Part I : Introduction to Data Structures

1.1 Concept of Data

1.1.1 Data

- Data is a **collection of information** but it is in **raw** form.
- When data is **processed** it becomes **information.**
- Meaningful, logical and processed form of data is called information.
- Data is numerical, character, some symbol or any other kind of information.

1.1.2 Data Object

- Data object is a term that refers to a set of elements. This set can be finite or infinite.
- **For example -** Consider a set of students studying in second year of college. This is a finite set where as the set of natural numbers is an infinite set.

1.1.3 Data Type

- Data type means the type of data.
- This type can be numeric, alphanumeric and so on.
- In C there are various data types such as int, float,char and so on.
- Generally a programming language supports a set of **built in data types** and allows the user to define a new type. Such data types are called as **user defined data types.** For example - In C struct or enum are used to for user defined data types.
- Thus there are two types of data types -

1. **Built in data types :** These are int, float, double, char which are defined by programming language itself.
2. **User defined data types :** Using the set of built in data types user can define his own data type. For example -

```
typedef struct student
{
        int roll;
        char name[10];
};s;
s stud1;
```

Here **s** represents the user defined data type defined with the help of **struct** and **stud1** is the variable name.

1.1.4 Data Structure

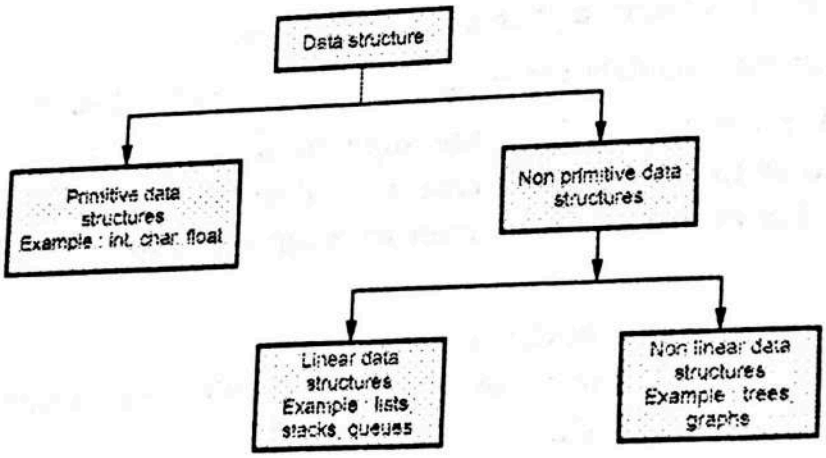
- Data structure is a collection of set of elements and various operations that can be performed on these elements.
- **Definition :** Data structure is a set of domains D, set of functions F, and set of axioms A (or rules) . This triple (D,F,A) denotes data structure D.
- **For example :** Arrays is a data structure containing the set of numeric elements on which the operations such as addition, multiplication, subtraction can be performed.

Review Questions

1. *Define the following terms :* SPPU : Dec.-08,09,10, Marks 4, Dec.-14, Marks 3
   i) Data object ii) Data type
2. *Define the following terms :* SPPU : May-18, Marks 6
   Data object, data type and data structure. SPPU : Dec.-18, Marks 3
3. *Define the following (1) Data object (2) Data structure.*

1.2 Types of Data Structure

SPPU : Dec.-06,11,13,15,16, May-09,13,14,16, Marks 10



1.2.1 Primitive and Non Primitive

- **The primitive data types are the fundamental data structures. The non primitive data structures can be built using primitive data structures.**
- **Examples of Primitive data structures :** In C language the primitive data structures are int, float, double, char.
- **The non-primitive data structures are user defined data structures.**

- **Examples of Non-primitive data structures :** In C language, the non primitive data structures are structure, union and enumerated data types.

1.2.2 Linear and Non Linear

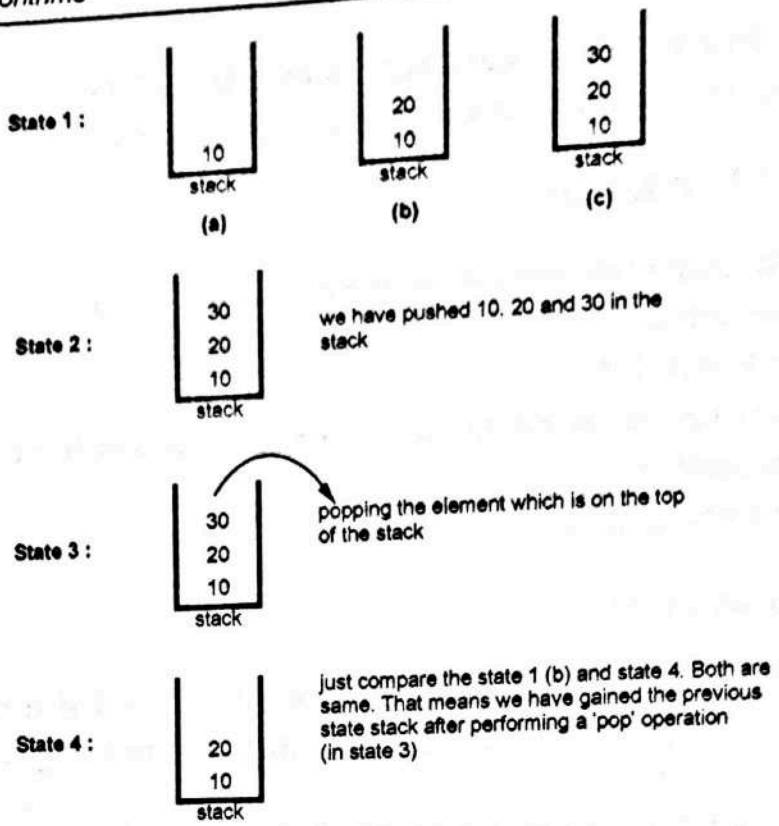
- Linear data structures are the data structures in which data is arranged in a list or in a straight sequence.
  For example : arrays, list
- Non linear data structures are the data structures in which data may be arranged in hierarchical manner.
  For example : trees, graphs.

1.2.3 Static and Dynamic

- The static data structures are data structures having fixed size memory utilization. One has to allocate the size of this data structure before using it.
  For example :
- Arrays in C is a static data structure. We first allocate the size of the arrays before its actual use. Sometimes whatever size we have allocated for the arrays may get wasted or we may require larger than the one which is existing. Thus the data structure is static in nature.
- The dynamic data structure is a data structure in which one can allocate the memory as per his requirement. If one does not want to use some block of memory, he can deallocate it.
  For example :
- Linked list, the linked list is a collection of many nodes. Each node consists of data and pointer to next node. In linked list user can create as much nodes (memory) as he wants, he can dellocate the memory which cannot be utilized further.
- The advantage of dynamic data structure over the static data structure is that there is no wastage of memory.

1.2.4 Persistent and Ephemeral

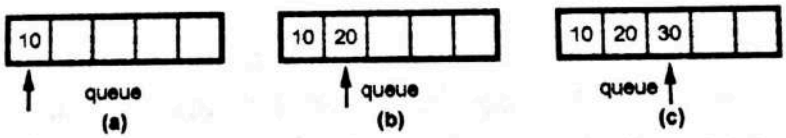
- The persistent data structures are the data structures which retain their previous state and modifications can be done by performing certain operations on it.
- For example stack can be a persistent data structure in following case.
- If we push 10, 20, 30 onto it and by performing pop operation we can gain its previous state. That means -



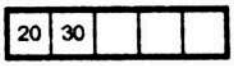
- Thus stack is a persistent data structure.
- The ephemeral data structures are the data structures in which we **cannot retain its previous state.**

For example - queues.

State 1 : In queues elements are inserted by one end and deleted by other end. Let us insert 10, 20, 30 in the queue.



State 2 : Now if we delete the element, we can delete it only by other end. That means 10 can be deleted. And queue will be -



Now this state is not matching with any of the previous states. That means we can not retain the previous state of the data structure even after performing certain operations. Such a data structure is called ephemeral data structure.

Review Questions

1. *Explain the following terms with suitable example.*
   i) Data ii) Data object iii) Data type iv) Abstract data type v) Data structure vi) Classification of data structure. **SPPU : Dec. 06, 16, May 09, Marks 10**

2. *Explain linear and non-linear data structures.*
   [illegible]
3. *Differentiate between primitive and non primitive data structure.* [illegible]
4. *What is persistent data structure ?* [illegible]
5. *Give classification of data structures with one example of each type.*
   [illegible]
6. *Explain static and dynamic data structures with suitable examples.*
   [illegible]

1.5 Definition of ADT [illegible]

The abstract data type is a triple of D - set of domains, F - set of functions, A - axioms in which only what is to be done is mentioned but how is to be done is not mentioned, i.e. rules are specified.

In ADT, all the implementation details are hidden. In short

ADT = Type + Function names + Behavior of each function

1.5.1 Realization of ADT in C

The abstract datatype consists of following things,

1. Data used along with its data type.
2. Declaration of functions which specify only the purpose. That means "What is to be done" in particular function has to be mentioned but "how is to be done" must be hidden.
3. Behavior of function can be specified with the help of data and functions together.

Thus ADT allows programmer to hide the implementation details. Hence it is called **abstract.**

For example : If we want to write ADT for a set of integers, then we will use following method

AbstractDataType Set {

Instances : Set is a collection of integer type of elements.

Preconditions : None

Operations :

1. **Store( ) : This** operation is for storing the integer element in a set.
2. **Retrieve( ) : This** operation is for retrieving the desired element from the given set.
3. **Display( ) : This** operation is for displaying the contents of set.

}

There is a specific method using which an ADT can be written. We begin with keyword **AbstractDataType** which is then followed by name of the data structure for which we want to write an ADT. In above given example we have taken **Set** data structure.

- Then inside a pair of curly brackets ADT must be written.
- We must first write **instances** in which the basic idea about the corresponding data structure must be given. Generally in this section definition of corresponding data structure is given.
- Using **Preconditions** or **Postconditions** we can mention specific conditions that must be satisfied before or after execution of corresponding function.
- Then a listing of all the required operations must be given. In this section, we must specify the purpose of the function. We can also specify the data types of these functions.

1.3.2 ADT for Arrays

In ADT instances represent the elements on which various operations can be performed. The basic operations that can be performed on arrays are store() and display(). Hence -

AbstractDataType Array

{

Instances : An array A of some size, index i and total number of elements in the array n.

Operations :

Store() - This operation stores the desired elements at each successive location.

display() - This operation displays the elements of the array.

}

ADT is useful to handle the data type correctly. Always what is to be done is given in ADT but how is to be done is not given in ADT.

Review Questions

1. *Define the following terms with an example :*
   i) Data type ii) Data structure iii) Abstract data type SPPU : Dec. 06, Marks 6
2. *What is abstract data type ? Explain ADT for an array.* SPPU : May-06, 07, Marks 6
3. *What is abstract data type ?* SPPU : Dec.-08, 09, Marks 4
4. *What is an abstract data type ? Explain with an example.* SPPU : May-10, 13, Marks 4
5. *Define the following terms with example :*
   i) Data object ii) Data Structure iii) Abstract data type SPPU : Dec.-14, Marks 6

Part II : Analysis of Algorithm

1.4 Introduction to Algorithm

Definition of Algorithm

An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statements in simple English language.

Example : Let us take a very simple example of an algorithm which adds the two numbers and stores the result in a third variable.

Step 1 : Start.

Step 2 : Read the first number is variable 'a'.

Step 3 : Read the second number in variable 'b'.

Step 4 : Perform the addition of both the numbers i.e. and store the result in variable 'c'.

Step 5 : Print the value of 'c' as a result of addition.

Step 6 : Stop.

1.4.1 Characteristics of Algorithm

1. Each algorithm is supplied with **zero or more inputs.**
2. Each algorithm must **produce** at least one **output.**
3. Each algorithm should have **definiteness** i.e. each instruction must be **clear and unambiguous.**
4. Each algorithm should have **finiteness** i.e. if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after finite number of steps.

Each algorithm should have **effectiveness** i.e. every instruction must be sufficiently basic that it can in principal be carried out by a person using only pencil and paper. Moreover each instruction of an algorithm must also be feasible.

Example 1.4.1 *Write an pseudo code to count the sum of n numbers.*

Solution :

```
Algorithm sum (1, n)
//Problem Description: This algorithm is for finding the
//sum of given n numbers
//Input: 1 to n numbers
```

```
//Output: The sum of n numbers
        result ← 0
        for i ← 1 to n do i ← i+1
                result ← result+i
return result
```

Example 1.4.2 *Write a pseudo code to check whether given number is even or odd.*

Solution :

```
Algorithm eventest (val)
//Problem Description: This algorithm test whether given
//number is even or odd
//Input: the number to be tested i.e. val
//Output: Appropriate messages indicating even or oddness
if (val%2=0) then
        write ("Given number is even")
else
        write("Given number is odd")
```

Example 1.4.3 *Write a pseudo code to find factorial of n number.*

Solution :

```
Algorithm fact (n)
//Problem Description: This algorithm finds the factorial
//of given number n
//Input: The number n of which the factorial is to be
//calculated.
//Output:factorial value of given n number.
if(n ← 1) then
        return 1
else
        return n*fact(n – 1)
```

Review Question

1. *Write characteristics of an algorithm.* SPPU : Dec.-11, Marks 4

1.5 Analysis of Algorithm

SPPU : May-10,12,13, Dec.-09,10,11,13,15, Marks 8

- The efficiency of algorithm can be decided by measuring the performance c algorithm. We can measure the performance of an algorithm by counting tw factors -

1. Amount of time required by the algorithm to execute. This is known as **time complexity.**
2. Amount of space required by the algorithm to execute. This is known as **space complexity.**

- Suppose we want to find out the time taken by following program statement.

```
x=x+1
```

- Determining the amount of time required by the above statement in terms of clock time is not possible because following is always dynamic -

1. The machine that is used to execute the programming statement.
2. Machine language instruction set.
3. Time required by each machine instruction.
4. The translation of compiler will make for this statement to machine language.
5. The kind of operating system (multi-programming or time sharing).

- The above information varies from machine to machine. Hence it is not possible to find out the exact figure. Hence the performance of the machine is measured in terms of **frequency count.**

1.5.1 Frequency Count and Its Importance

Definition : The frequency count is a count that denotes how many times particular statement is executed.

For Example : Consider following code for counting the frequency count

```
void fun()
{
 int a;
 a=10; ..........................................1
 printf("%d",a); ....................................1
}
```

The frequency count of above program is **2.**

Example 1.5.1 *Obtain the frequency count for the following code :*

```
void fun()
{
 int a;
 a=0;
 for(i=0;i<n;i++)
  {
```

```
    a  = a+i;
  }
  printf("%d",a);
}
```

Solution :

```
void fun()
{
 int a;
 a=0; ..............................................1
 for(i=0;i<n;i++) ..................................n+1
  {
    a  = a+i; ......................................n
  }
  printf("%d",a); ...................................1
}
```

The frequency count of above code is **2n+3**

The for loop in above given fragment of code is executed n times when the condition is true and one more time when the condition becomes false. Hence for the for loop the frequency count is n+1. The statement inside the for loop will be executed only when the condition inside the for loop is true. Therefore this statement will be executed for n times. The last printf statement will be executed for once.

Example 1.5.7 *Obtain the frequency count for the following code :*

```
void fun(int a[][],int b[][])
{
 int c[3][3];
  for(i=0;i<m;i++)
  {
   for(j=0;j<n;j++)
   {
    c[i][j]=a[i][j]+b[i][j];
   }
  }
```

Solution :

```
void fun(int a[][],int b[][])
{
 int c[3][3];
  for(i=0;i<m;i++) ..................................m+1
  {
   for(j=0;j<n;j++) ................................m(n+1)
   {
```

```
    c[i][j]=a[i][j]+b[i][j]; ..........................m.n
  }
}
```

The frequency count = (m+1) + m (n+1) + mn = 2 m + 2 mn + 1 = 2m (1+n) + 1

Example 1.5.3 *Obtain the frequency count for the following code :*

```
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
 {
   c[i][j]=0;
   for(k=1;k<=n;k++)
     c[i][j]=c[i][j]+a[i][k]*b[k][j];
 }
}
```

Solution :

Statement	Frequency count
for(i=1;i<=n;i++)	n+1
for(j=1;j<=n;j++)	n.(n+1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n+1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3+3n^2+2n+1$

After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

Example 1.5.4 *Obtain the frequency count for the following code :*

```
i=1;
do
{
 a++;
 if(i==5)
   break;
 i++;
}while(i<=n)
```

Solution :

Statement	Frequency count
i=1;	1
a++;	5
if(i==5)	5
break;	1
i++;	5
while(i<=n)	5
Total	22

Example 1.5.5 *Obtain the frequency count for the following code :*

```
m=n/2;
for(i=0;i+m<m;i++)
{
  a++;
  k++;
}
```

Solution :

Statement	Frequency count
m=n/2	1
for(i=0;i+m<m;i++)	(n/2)+1
a++;	n/2
k++;	n/2
Total	(3n/2)+2

If we consider only the order of magnitude for this code then the time complexity in terms of Big-oh notation can be specified as **O(n/2)**

Example 1.5.6 *Find the Frequency Count (F.C.) of the above code : Explain each step.*

```
double IterPow(double X, int N)
{
double Result = 1;
while (N > 0)
{
Result = Result *X;
N--;
}
return Result;
}
```

Solution :

Statement	Frequency count
double Result=1	1
while(N>0)	N+1
Result=Result*X	N
N--	N
return Result	1
Total	**3N+3**

What is frequency count of a statement ? Analyze time complexity of the following code :

```
i)  for(i = 1; i <= n; i++)
        for(j = 1; j <= m; j++)
        for(k = 1; k <= p; k++)
        sum = sum + i;
ii) i = n;
        while(i ≥ 1)
        {i--;}
```

Solution : i)

Statement	Frequency count
for(i=1;i<=n;i++)	n+1
for(j=1;j<=m;j++)	n(m+1)
for(k=1;k<=p;k++)	n.m.(p+1)
sum=sum+i	n.m.p
Total	2(n+nm+nmp)+1

II)

Statement	Frequency count
i=n	1
while(i>=1)	n+1
i--	n
Total	2(n+1)

Example 1.5.1 *Determine the frequency counts for all the statements in the following program segment :*

```
i=10;
for(i=10;i<=n;i++)
 for(j=1;j<i;j++)
  x=x+1;
```

Solution :

Statement	Frequency count
i=10;	1
for(i=10;i<=n;i++)	m we assume the count of execution is m
for(j=1;j<i;j++)	m((m+1)/2)
x=x+1;	m(m)

- Assume (n – 10 +2) = m . Then the total frequency count is -
- The outer loop will execute for m times. The inner loop is dependant upon the outer loop. Hence it will be (m+1)/2. Hence overall frequency count is $(1+m+m^2)(m(m+1)/2)$.

1.5.2 Time Complexity

- The amount of time required by an algorithm to execute is called the time complexity of that algorithm.
- For determining the time complexity of particular algorithm following steps are carried out -

1. Identify the basic operation of the algorithm.
2. Obtain the frequency count for this basic operation.
3. Consider the order of magnitude of the frequency count and express it in terms of big oh notation.

Example 1.5.3 *Write an algorithm to find smallest element in a array of integers and analyze its time complexity.*

Solution :

```
Algorithm MinValue(int a[n])
{
    min_element=a[0];
    for(i=0;i<n;i++)
```

```
    {
        if (a[i]<min_element) then
            min_element=a[i];
    }
    Write(min_element);
}
```

We will first obtain the frequency count for the above code.

Statement	Frequency count
min_element=a[0]	1
for(i=0;i<n;i++)	n+1
if (a[i]<min_element) then	n
Write(min_element);	1
Total	2n+3

By neglecting the constant terms and by considering the order of magnitude we can express the frequency count in terms of Omega notation as O(n). Hence the time complexity of above code is O(n).

Example 1.5.10 *What is frequency count ? Write its importance in analysis of algorithm. Find time complexity of an algorithm to find union of two sets of length m and n.*

SPPU : Dec-16, Marks 6

Solution :

```
void Union(int a[],int b[],int term1,int term2)
{
int c[MAX];
int i,j,k,n,flag;
k=0;
flag=0;
for(i=0;i<3;i++)
{
    c[k++]=a[i];
}
for(j=0;j<3;j++)
{
    flag=0;
    for(i=0;i<3;i++)
```

```
        {
                if(a[i]==b[j])
                        flag=1;
        }
        if(flag==0)
                c[k++]=b[j];
}
n=k;
printf("\n The A union B={");
for(k=0;k<n;k++)
  printf(" %d",c[k]);
printf(" }");
}
```

The basic operation in above algortihm is two nested for loops that combines the common and uncommon elements. Hence the time complexity of this algorithm is O(n^2).

1.5.3 Space Complexity of Algorithm

SPPU : Dec.-10, Marks 4

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors : **Constant and instance characteristics.** The space requirement S(p) can be given as,

$$S(p) = C + Sp$$

where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

There are two types of components that contribute to the space complexity - Fixed part and variable part.

The **fixed part** includes space for

- Instructions
- Variables
- Array size
- Space for constants

The **variable part** includes space for

- The variables whose size is dependent upon the particular problem instance being solved. The control statements (such as for, do, while, choice) are used to solve such instances.
- Recursion stack for handling recursive call.

Consider an example of algorithm to compute the space complexity.

Example 1.5.14 *Compute the space needed by the following algorithms justify your answer.*

```
Algorithm Sum(a,n)
{
    s : = 0.0 ;
        For i : = 1 to n do
            s : = s + a[i] ;
            return s
}
```

Solution : In the given code we require space for

```
s : = 0                          ← O(1)
    For i : = 1 to n             ← O(n)
        s : = s + a[i] ;         ← O(n)
        returns ;                ← O(1)
```

Hence the space complexity of given algorithm can be denoted in terms of big-oh notation. It is **O(n).**

Review Questions

1. *What do you mean by frequency count and its importance in analysis of an algorithm.*
   SPPU : Dec.-09,10, May-12,13, Marks 6
2. *What is time complexity ? How is time complexity of an algorithm computed ?*
   SPPU : Dec.-11, Marks 6
3. *What is time complexity of an algorithm ? Explain its importance with suitable examples.*
   SPPU : Dec.-15, Marks 3
4. *What is space complexity of an algorithm ? Explain its importance with example.*
   SPPU : Dec.-10, Marks 4, Dec.-14, Marks 3

1.6 Asymptotic Notations

SPPU : Dec.-10,14,15,16; May-10,12,16; Marks 8

- To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.
- Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".
- Various notations such as Ω, Θ and O used are called **asymptotic notions.**

1.6.1 Big oh Notation

- The **Big oh** notation is denoted by 'O'. It is a method of representing the **upper bound of** algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Definition

- Let f(n) and g(n) be two non-negative functions.
- Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write,

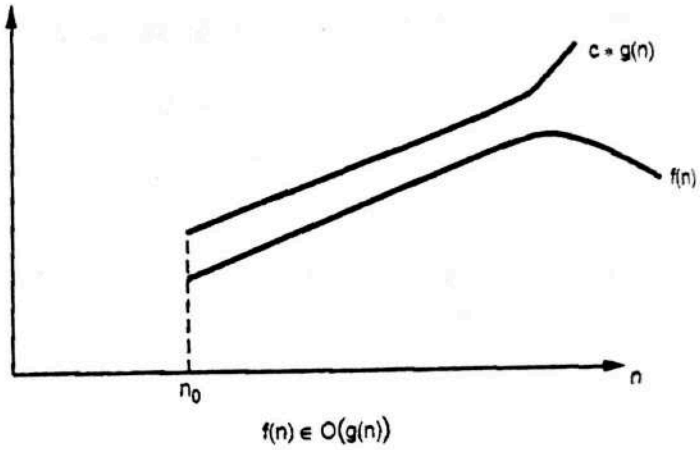


Fig. 1.6.1

$$f(n) \le c^* g(n)$$

- Then f(n) is big oh of g(n). It is also denoted as f(n) ∈ O (g(n)). In other words f(n) is less than g(n) if g(n) is multiple of some constant c.

Example : Consider function f(n) = 2n + 2 and g(n) = n^2. Then we have to find some constant c, so that f(n) ≤ c * g(n).

As f(n) = 2n + 2 and g(n) = n^2 then we find c for n = 1 then

$$f(n) = 2n + 2$$
$$= 2(1) + 2$$
$$f(n) = 4$$

and

$$g(n) = n^2$$
$$= (1)^2$$

$$g(n) = 1$$

i.e. $f(n) > g(n)$

If n = 2 then,

$$f(n) = 2(2) + 2$$
$$= 6$$
$$g(n) = (2)^2$$
$$g(n) = 4$$

i.e. $f(n) > g(n)$

If n = 3 then,

$$f(n) = 2(3) + 2$$
$$= 8$$
$$g(n) = (3)^2$$
$$g(n) = 9$$

i.e. $f(n) < g(n)$ is true.

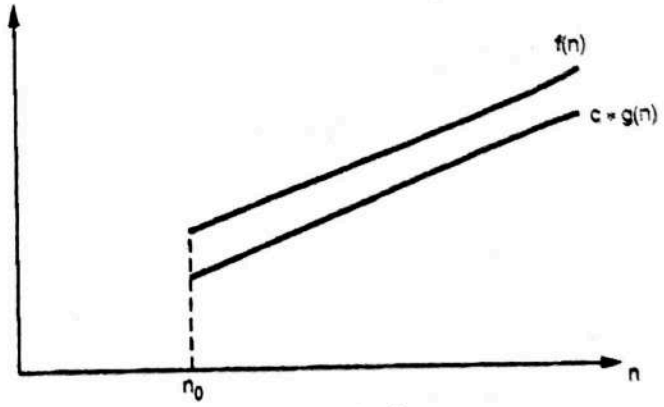
Hence we can conclude that for n > 2, we obtain

$$f(n) < g(n)$$

Thus always upper bound of existing time is obtained by big oh notation.

1.6.2 Omega Notation

Omega notation is denoted by 'Ω'. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote **shortest** amount of time taken by algorithm.



$f(n) \in \Omega(g(n))$

Fig. 1.6.2

Definition

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$$f(n) \geq c * g(n) \quad \text{For all } n \geq n_0$$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

Example :

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$$f(n) = 2(0)^2 + 5$$
$$= 5$$
$$g(n) = 7(0)$$
$$= 0 \quad \text{i.e. } f(n) > g(n)$$

But if $n = 1$

$$f(n) = 2(1)^2 + 5$$
$$= 7$$
$$g(n) = 7(1)$$
$$7 \text{ i.e. } f(n) = g(n)$$

If $n = 3$ then,

$$f(n) = 2(3)^2 + 5$$
$$= 18 + 5$$
$$= 23$$
$$g(n) = 7(3)$$
$$= 21$$

i.e. $f(n) > g(n)$

Thus for $n > 3$ we get $f(n) > c * g(n)$.

It can be represented as

$$2n^2 + 5 \in \Omega(n)$$

Similarly any

$$n^3 \in \Omega(n)^2$$

1.6.3 Θ Notation

The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition

Let f(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n) \leq c_2 g(n)$$

Then we can say that

$$f(n) \in \Theta(g(n))$$

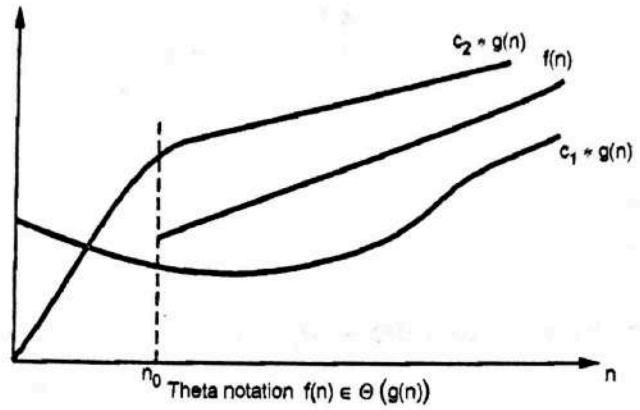


Fig. 1.6.3

Example :

If f(n) = 2n + 8 and g(n) = 7n.

where $n \geq 2$

Similarly $f(n) = 2n + 8$

$g(n) = 7n$

i.e. $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The theta notation is more precise with both big oh and omega notation.

1.6.4 Properties of Order of Growth

1. If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))$.
2. Polynomials of degree $m \in \Theta(n^m)$.

   That means maximum degree is considered from the polynomial.

   For example : $a_1 n^3 + a_2 n^2 + a_3 n + c$ has the order of growth $\Theta(n^3)$.
3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
4. Exponential functions a^n have different orders of growth for different values of a.

Key Points

i) $O(g(n))$ is a class of functions $f(n)$ that grows less fast than $g(n)$, that means $f(n)$ possess the time complexity which is always lesser than the time complexities that $g(n)$ have.

ii) $\Theta(g(n))$ is a class of functions $f(n)$ that grows at same rate as $g(n)$.

iii) $\Omega(g(n))$ is a class of functions $f(n)$ that grows faster than or atleast as fast as $g(n)$. That means $f(n)$ is greater than $\Omega(g(n)$.

Example 1.6.5 *Analyze time complexity of the following code segments :*

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

```
ii) i = 1
        while (i <= n)
    { x++ ;
            i++;
                }
```

```
iii) int process (int no)
     {
          if (no <= 0)
               return (0) ;
          else
               return (no + process (no – 1));
               }
```

SPPU : Dec.-10, Marks 8

Solution :

i) $O(n^3)$ ii) $O(n)$ iii) $O(n)$

1.6.6 How to Choose the Best Algorithm ?

- If we have two algorithms that perform same task and the first one has a computing time of $O(n)$ and the second of $O(n^2)$, then we will usually prefer the first one.
- The reason for this is that as n increases the time required for the execution of second algorithm will get far more than the time required for the execution of first.
- We will study various values for computing function for the constant values. The graph given below will indicate the rate of growth of common computing time functions.

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648

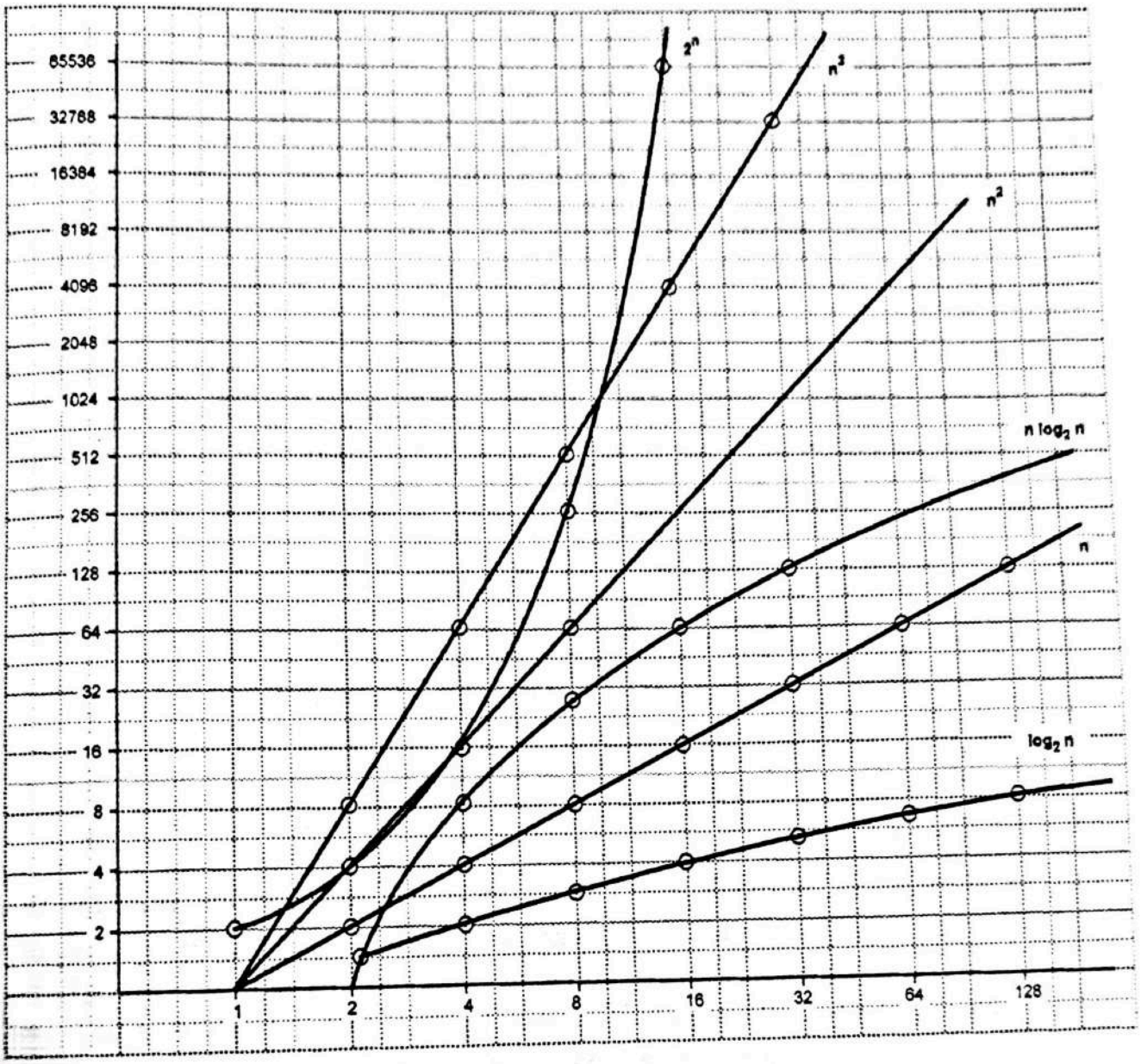


Fig. 1.6.4 Rate of growth of common computing time function

Notice how the times $O(n)$ and $O(n \log n)$ grow much more slowly than the others. For large data sets algorithms with a complexity greater than $O(n \log n)$ are often impractical. The very slow algorithm will be the one having time complexity 2^n.

University Questions

1. *Explain Big-oh, omega and theta notation with example.*

   SPPU: May-13, 14, Dec.-16, Marks 6

2. *Explain the different asymptotic notations.*

   SPPU: May-10, Dec.-14, May-16, Marks 3, Dec.-15, Marks 6

1.7 Best, Worst and Average Case of Algorithm

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case** time complexity.

For example : While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called **average case** time complexity.

Consider the following algorithm

```
Algorithm Seq_search(X[0 ... n - 1 ],key)
    // Problem Description: This algorithm is for searching the
    //key element from an array X[0...n - 1] sequentially.
    //Input: An array X[0...n - 1] and search key
    //Output: Returns the index of X where key value is present
    for i ← 0 to n - 1 do
     if(X[i]=key)then
     return i
```

Best Case Time Complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at first location in the list(X[0...n–1]) then algorithm runs for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{best} = 1$$

Worst Case Time Complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at n^{th} location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{worst} = n$$

The algorithm guarantees that for any instance of input which is of size n, the running time will not exceed $C_{worst}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average Case Time Complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and

P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.

The probability of getting unsuccessful search is (1 – P).

Now, we can find average case time complexity $C_{avg}(n)$ as -

$C_{avg}(n)$ = Probability of successful search (for elements 1 to n in the list) + Probability of unsuccessful search

$$C_{avg}(n) = \left[1\cdot\frac{P}{n} + 2\cdot\frac{P}{n} + \ldots + i\cdot\frac{P}{n}\right] + n\cdot(1-P)$$

There may be n elements at which chances of 'not getting

$$= \frac{P}{n}[1+2+\ldots+i\ldots n] + n(1-P)$$

$$= \frac{P}{n}\frac{n(n+1)}{2} + n(1-P)$$

$$C_{avg}(n) = \frac{P(n+1)}{2} + n(1-P)$$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if P = 0 that means there is no successful search i.e. we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation,

$$C_{avg}(n) = 0(n+1)/2 + n(1-0)$$

$$C_{avg}(n) = n$$

Thus the average case running time complexity becomes equal to n.

Suppose if P = 1 i.e. we get a successful search then

$$C_{avg}(n) = 1(n+1)/2 + n(1-1)$$

$$C_{avg}(n) = (n+1)/2$$

That means the algorithm scans about half of the elements from the list.

For calculating average case time complexity we have to consider probability of getting success of the operation. And any operation in the algorithm is heavily dependent on input elements. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

Part III: Sequential Organization

1.8 Single and Multidimensional Arrays

The arrays are defined as the collection of similar data type of elements. This means in an array either all the elements can be of integer type or float type or all of character type. The syntax of array is

Data_type name_of_array[size_of_array];

1.8.1 One Dimensional Arrays

Array can be one dimensional and two dimensional.

For example

You can visualize one dimensional array as :

Note that the elements in array are always stored in contiguous memory locations.

Now let us see how to handle this array. We will write a simple program in which we are simply going to store the elements and then we will print those stored elements.

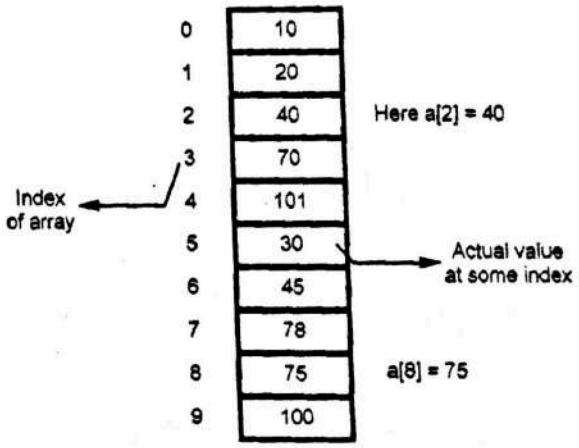


Fig. 1.8.1 Array a[10]

```
#include<iostream>
using namespace std;
int main()
{
    int a[10],index;
    cout<<"\n Store The Elements In An Array a");
    for(index=0;index<=9;index++)
        cin>>a[index];
    cout<<"\n You have stored these numbers in the array...");
    for(index=0;index<=9;index++)
        cout<<"\n<<a[index];
    return 0;
}
```

1.8.2 Two Dimensional Arrays

The two dimensional array is something which you can compare with the two storied building! Extra space which is arranged in rows and columns. Let us draw a figure which will represent the two dimensional array.

The syntax of two dimensional array is :

```
Data_type Name_Of_array[row_size][column_size];
```

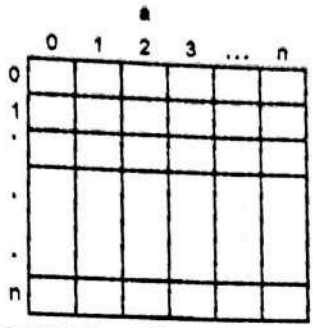


Fig. 1.8.2 Two dimensional array

For example

```
int a[10][10];
```

By this allocation the memory block of 10 x 10 = 100 is getting created. The nested for loops can be effectively used to access all the elements of an two dimensional array. We will take a sample example to explain this idea.

```
for(i=0;i<=2;i++)
{
 for(j=0;j<=2;j++)
 {
  cin>>a[i][j];
 }
}
```

Execution of Nested for Loop

Initially the value of i=0 at that time
j=0 it will store the element in a[0][0]
j=1 it will store the element in a[0][1]
j=2 it will store the element in a[0][2]
Next time i will be incremented by 1 and now i=1
Again
j=0 it will store the element in a[1][0]
j=1 it will store the element in a[1][1]
j=2 it will store the element in a[1][2]
Next time i will be incremented by 1 and now i=2

j=0 it will store the element in a[2][0]
j=1 it will store the element in a[2][1]
j=2 it will store the element in a[2][2]

Array positions

a00	a01	a02
a10	a11	a12
a20	a21	a22

Fig. 1.8.3 a[0][0] to a[2][2]

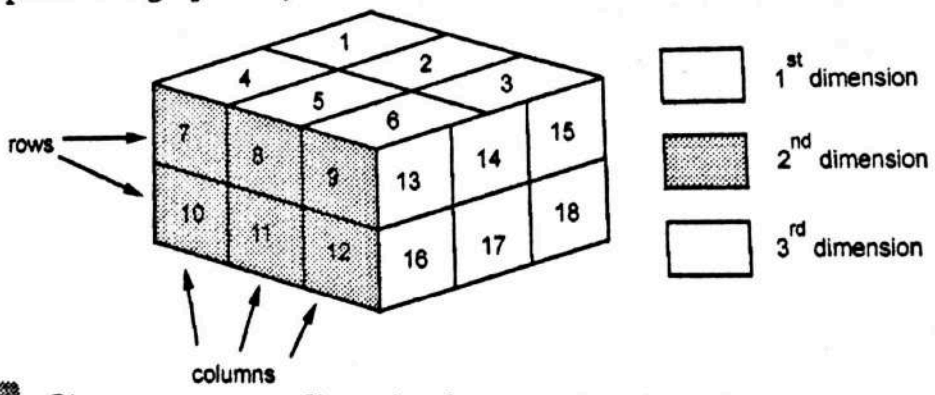
Since now value of i and j has reached to 2 the scanning procedure will get terminated as condition is i<=2 and j<=2. Thus all the elements from a[0][0] to a[2][2] get scanned.

1.8.3 Multidimensional Array

The multidimensional array is similar to the two dimensional array with multiple dimensions. For example here is 3-D array

```
a[3][2][3] = { {{1,2,3},{4,5,6}},
               {{7,8,9}, {10,11,12}},
                      {{13,14,15} {16,17,18}}
             }
```

We can represent it graphically as



Example 1.8.1 *Given are two one-dimensional arrays A and B which are in ascending order. Write C program to merge them into a single sorted array C that contains every item from arrays A and B, in ascending order.* **SPPU : May-07, Dec.-07, May-10**

Solution :

```
merge()
{
        int a[20],b[20],c[40],i,j,k,m,n;
        printf("\nEnter number of elements in a:");
        scanf("%d",&m);
        for(i=0;i<m;i++)
                scanf("%d",&a[i]);
        printf("\nEnter number of elements in b:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
```

```
        scanf("%d",&b[i]);
    i=j=k=0;
    while(i<m&&j<n)
    {
        if(a[i]<=b[j])
        {
            c[k]=a[i];
            i++;
            k++;
        }
        else
        {
            c[k]=b[i];
            j++;
            k++;
        }
    }
    while(i<m)
    {
        c[k]=a[i];
        i++;
        k++;
    }
    while(j<n)
    {
        c[k]=b[i];
        j++;
        k++;
    }
    for(i=0;i<k;i++)
        printf("%d\t",c[i]);
}
```

Review Questions

1. *Explain arrays with example.* SPPU : May-11, Marks 6
2. *Explain concept of arrays with suitable example.* SPPU : Dec-16, Marks 4
3. *Explain row major and column major representation of a matrix with example.* SPPU : Dec-17, Marks 6

SPPU : Dec-14, Marks 4

1.9 Address Calculation

When a two dimensional array gets stored in computer's memory, all the elements of it get stored at the successive memory locations. And this allocation of elements in the memory is based on two types of representations - **Row major representation** and **Column major representation.**

For example : Consider a 2D array 'a' as

```
int a[2][3] = {
                {10,20,30},
                {40,50,60}
              };
```

In above 2D array there are two rows and three columns in which the elements are arranged. For the row major representation -

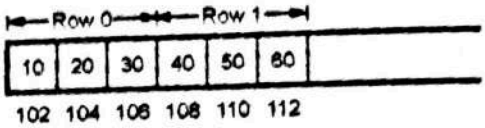


Fig. 1.9.1

For the column major representation -

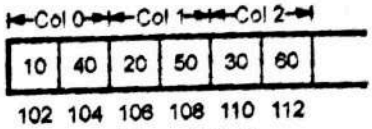


Fig. 1.9.2

Thus elements get stored at each successive location in the memory. Now we can access any element if the base address or starting address of an array is known. Suppose base address of an array is 102. Then we can find out location or address of any element based on index. But the method of address calculation varies for row major and column major representations.

Address Calculation for Row Major Array :

For calculating the address of any element in the array following formula is used -

address of a[i][j] = base address + i * n +j

Where the array is declared as a[m][n]. Here m,n represents the size of rows and columns respectively.

For example : Consider int a[2][3] = { {10, 20, 30}

{40, 50, 60}};

If we want to find out the address of a[1][2] i.e. location of element 60 then

= 102 + 1* 3 + 2 = 102 + 5

That is, for finding address of element 60 from base address we should move 5 places ahead i.e. at address 112.

Address calculation for Column Major Array :

For calculating the address of any element in the array following formula is used -

address of a[i][j] = base address + j * m + i

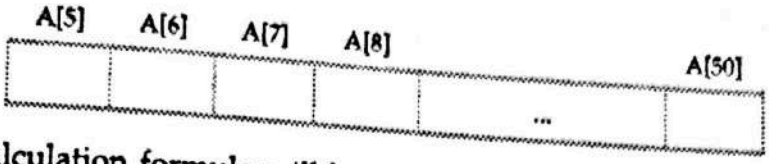
For example : If we want to find out the address of a[1][2] from above given 2D array i.e. location of element 60 then

= 102 + 2 * 2 + 1 = 102 + 5

That is for finding address of element 60 from base address we should move 5 places ahead i.e. at address **112**.

Example 1.5.1 *Consider the linear array, A (5:50), whose base address is 300 and the number of words per memory cell is 4. Find the address of A[15].*

Solution : The linear array can be



The address calculation formula will be

$\alpha + (i-5) * \text{size in words}$

Where α is base address.

As $\alpha = 300$ and $i = 15$,

We obtain A[15] as

$$A[15] = 300 + (15-5) * 4 = 300 + 40 = 340$$

Hence A [15] will be at location **340**.

Example 1.5.2 *Consider integer array int arr[4][5] declared in 'C' program. If the base address is 1020, find the address of the element arr[3][4] with row major and column major representation of array.*

Solution : Row Major Representation

The element a[i][j] will be at

a[i][j] = base address + (row_index*total number of cols + col_index)*element_size

= (base address+(i*col_size+j)*element_size)

when i=3 and j=4 ,element_size=int occupies 2 bytes of memory hence it is 2

total number of rows=4

a[3][4] = 1020+(3*5+4)*2 = 1020+38

a[3][4] = 1058

Column Major Representation

The element a[i][j] will be at

a[i][j] = base address+(col_index*total number of rows + row_index)*element_size

= (base address+(j*row_size+i)*element_size)

when i=3 and j=4, element_size=int occupies 2 bytes of memory hence it is 2

total number of columns=5

$$a[3][4] = 1020+(4*4+3)*2$$

$$= 1020+38$$

$$a[2][3] = 1058$$

Formula for Multidimensional Array

Let α represents the base address

The n dimensional array can be declared as $A[u_1][u_2] \ldots [u_n]$.

Now the two types of representations are as given below.

Row Major Representation

$$A[i_1][i_2][i_3]\ldots[i_n] = \alpha + (i_1 - 1)u_2 * u_3 * \ldots u_n + (i_2 - 1)u_3 * u_4 \ldots u_n + (i_3 - 1) * u_4 * u_5 \ldots u_n + (i_n - 1)$$

Column Major Representation

For column major representation the formula will be

$$A[i_1][i_2][i_3]\ldots[i_n] = \alpha + (i_n - 1) * u_2 * u_3 * \ldots u_n + (i_{n-1} - 1) * u_3 * u_4 * \ldots u_n + (i_1 - 1)$$

Example for 3D Array

Consider a three dimensional array A [a : b, c : d, e : f]

Let, $\alpha 1$, $\alpha 2$ and $\alpha 3$ are total sizes of first, second and third dimension of array.

Suppose we want to calculate address of A [p, q, r] - The specific element.

The effective index i = p – a, j = q – c, k = r – e

Row Major Order

For calculating address of A [p, q, r] -

Base address + w * [(i * $\alpha 2$+ j) * $\alpha 3$ + k]

Column Major Order -

For calculating address of A [p, q, r]

Base address + w * [(k * (α2 + j) * α1 + i]

For example

Consider an array A[2] [3] [4] with base address 100. The size of each element is 4 bytes and we have to obtain memory location of A[1] [1] [2].

Given that

α1 = 2, α2 = 3 and α3 = 4

a = 1, b = 2, c = 1, d = 3, e = 1, f = 4

p = 1, q = 1, r = 2, w = 4, Base address = 100.

Row Major Order

For calculating address of A [1] [1] [2] -

Base address + w * [(i * α2 + j) * α3 + k]

= 100 + 4 * [((p – a)* α2 + (q – c)) * α3 + (r – e)]

= 100 + 4 * [((1 – 1)* 3 + (1 – 1)) * 4 + (2 – 1)]

= 100 + 4 * [1]

= 104

Column Major Order

For calculating address of A[1] [1] [2] -

Base address + w*[(k * α2 + j) * α1 + i]

= Base address + 4 * [(1 * 3 + 0) * 2 + 0]

= 100 + 4 * 6

= 124

Review Questions

1. *Explain the concept of column major address calculation for multidimensional array with suitable example.* SPPU : Dec.-14, Marks 4

Part IV: Linked Organization

1.10 Concept of Linked Organization

SPPU : Dec.-05,08,13, May-06,09, Marks 6

- **Linked List** is referred as the **Linked organization.**
- In this organization, data is arranged in a linked list.
- The linked organization is also called the dynamic data structure because the nodes in the linked list are allocated or de-allocated as per the requirement.
- The dynamic memory allocation functions supports the creation of linked list.

Advantages of linked organization

1. In linked organization, insertion and deletion of elements can be efficiently done.
2. There is no wastage of memory. The memory can be allocated and deallocated as per requirement.

Disadvantages of linked organization

1. Linked organization does not support random or direct access.
2. Each data field should be supported by a link field to point to next node.

Review Questions

1. *What is a dynamic data structure ? List the advantages of linked lists.*

   SPPU : Dec.-05, May-06,09, Marks 6

2. *Write advantages of linked momory organization.*

   SPPU : Dec.-08, 13, Marks 4

1.11 Linked List as ADT

SPPU : Dec.-16, Marks 4

AbstractDataType LinkedList

{

Instances : List is a collection of elements which are arranged in a linear manner.

Operations : Various operations that can be carried out on list are -

1. **Insertion :** This operation is for insertion of element in the list.
2. **Deletion :** This operation removed the element from the list.
3. **Searching :** Based on the value of the key element the desired element can be searched.
4. **Modification :** The value of the specific element can be changed without changing its location.
5. **Display :** The list can be displayed in forward or in backward manner.

}

Review Question

1. *Explain with suitable example : Linked List as an ADT.* [Conceptual Question]

1.12 Singly Linked List

1.12.1 Representation of Linked List

Representation of Linked List using C++

```
class node
{
 public:
        int data;
        node *next;
};
class sll
{
 private:
        node *head;          <---- Data members of list
public:
        void create();
        void print();        <---- Operations on list
        .
        .
        .
};
```

1.12.2 Singly Linked List Operations

Various operations that can be performed on list are -

1. Creation	2. Insertion	3. Deletion
4. Reverse	5. Search	6. Display

Insertion and Deletion

1. Creation of linked list

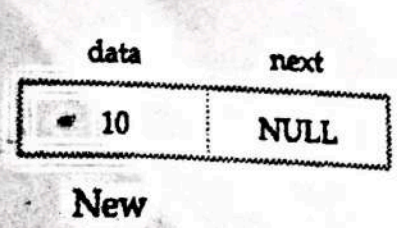
```
void sll :: create()
{
    node *temp, *New;
    int val,flag;
    char ans ='y';
    flag=TRUE;
    do
```

```
    {
        cout<<"\nEnter the data :";
        cin>>val;
        // allocate memory to new node
         New = new node;
         if ( New == NULL )
              cout<<"Unable to allocate memory\n";
        New-> data  = val ;
        New-> next = NULL;
        if( flag==TRUE ) // Executed only for the first time
        {
         head=New;
         temp = head;
         flag=FALSE;
        }
        else
        {
              /*temp last keeps track of the most recently
                created node*/
          temp->next = New;
          temp = New;
        }
        cout<<"\n Do you want to enter more elements?(y/n)";
        ans = getche();
    }while(ans=='y'||ans=='Y');
    cout<<"\nThe Singly Linked List is created\n";
    getch();
    clrscr();
}
```

Creation of linked list (logic explanation part) :

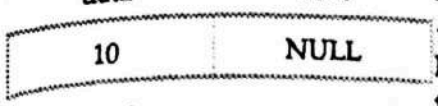
Initially one variable flag is taken whose value is initialized to TRUE (i.e. 1). The purpose of flag is for making a check on creation of first node. That means if flag is TRUE then we have to create head node or first node of linked list. Naturally after creation of first node we will reset the flag (i.e. assign FALSE to flag) Consider that we have entered the element value 10 initially then,

Step 1 :




```
New = new node ;
/* memory gets allocated for
New node */
New → data = Val;
/* value 10 will be put in data field of New */
```

Step 2 :

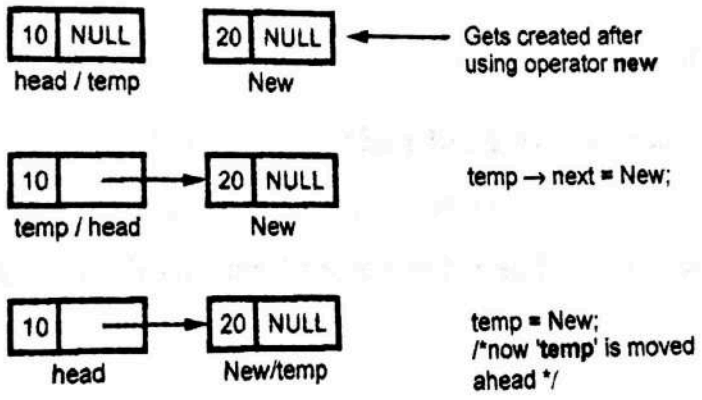


New/
head/temp

```
if (flag = = TRUE)
{
head = New
temp = head;
/* We have also called this node as temp because head's
address will be preserved in 'head' and we can change
'temp' node as per requirement */
flag = FALSE;
/* After creation of first node flag is reset */
```

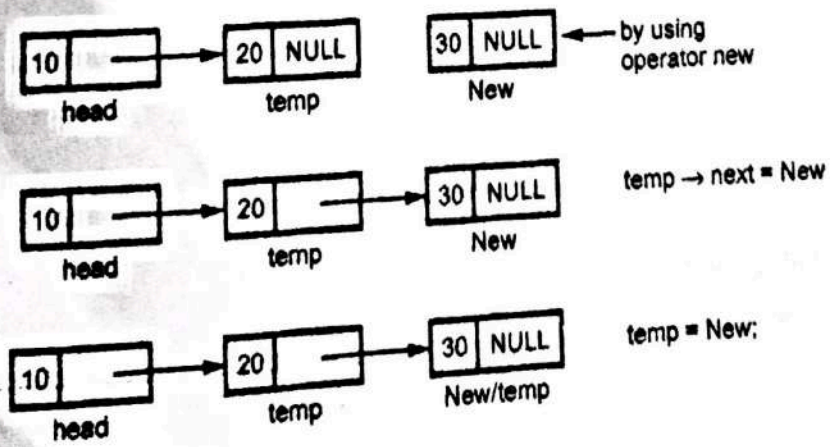
Step 3 :

If **head** node of linked list is created we can further create a linked list by attaching the subsequent nodes. Suppose we want to insert a node with value 20 then,



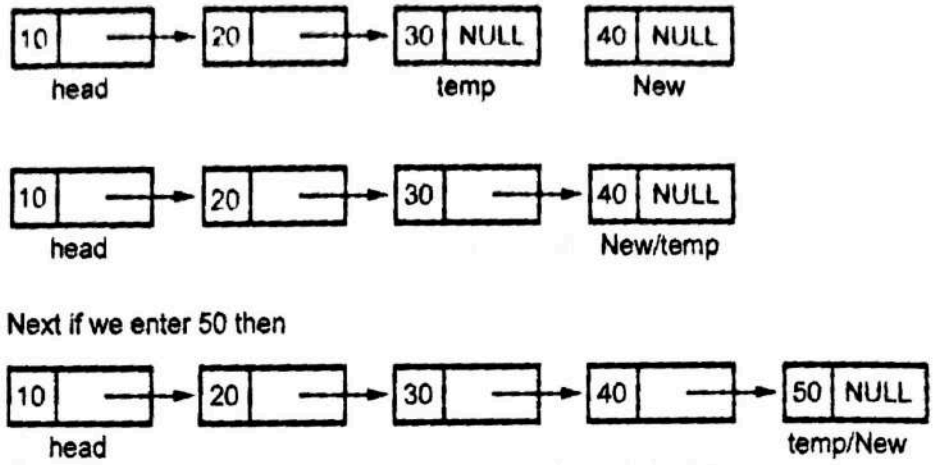
Step 4 :

If user wants to enter more elements then let say for value 30 the scenario will be,



Then for value 40 -

Step 5 :



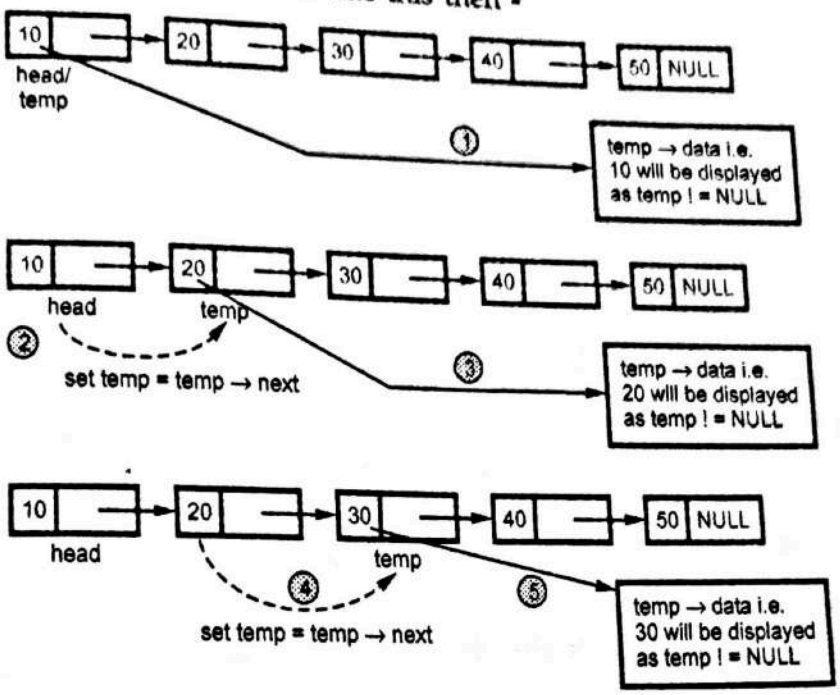
is the final linked list.

2. Display of linked list

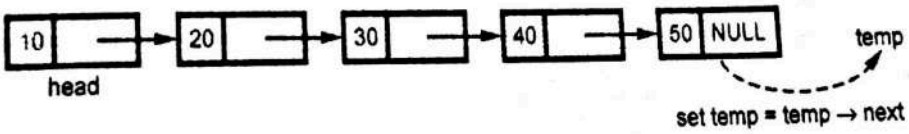
We are passing the address of **head** node to the display routine and calling head as the **'temp'** node. If the linked list is not created then naturally **head = temp** node will be **NULL.** Therefore the message "The list is empty " will be displayed.

```
void sll ::display()
{
    node *temp ;
    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";
        getch(); clrscr();
        return;
    }
    while ( temp != NULL )
    {
        cout<<temp->data<<" ";
        temp = temp -> next;
    }
    getch();
}
```

If we have created some linked list like this then -



Continuing in this fashion we can display remaining nodes 40, 50. When



As now value of temp becomes NULL we will come out of **while** loop. As a result of such display routine we will get,

10 → 20 → 30 → 40 → 50 → NULL

will be printed on console.

3. Insertion of any element at anywhere in the linked list

There are three possible cases when we want to insert an element in the linked list -

a) Insertion of a node as a head node

b) Insertion of a node as a last node

c) Insertion of a node after some node.

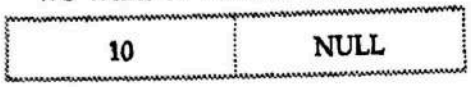
We will see the case a) first -

```
void sll:: insert_head()
{
node *New,*temp;
 New=new node;
cout<<"\n Enter The element which you want to insert";
```

```
cin>>New->data;
if(head==NULL)
        head=New;
else
{
temp=head;
New->next=temp;
head=New;
}
}
```

There is no node in the linked list. That means the linked list is empty

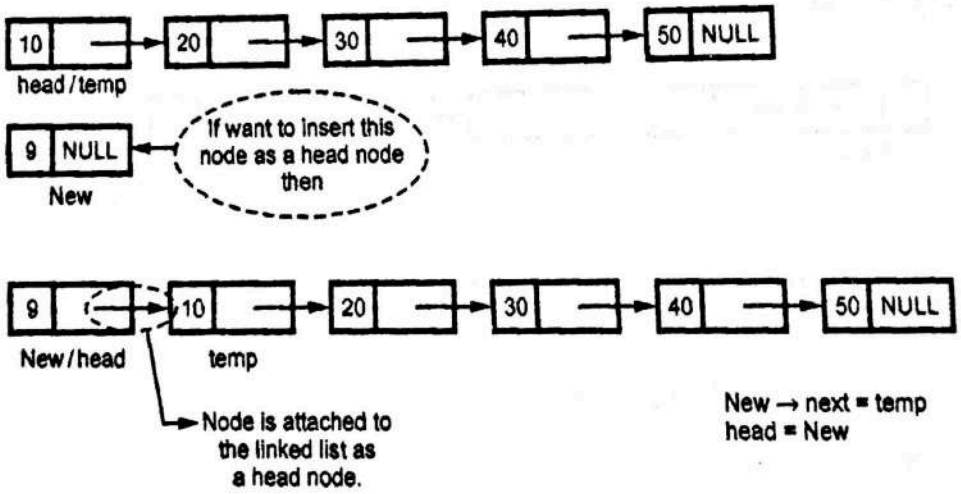
If there is no node in the linked list then value of head is NULL. At that time if we want to insert 10 then



head / New

```
Cin >>New → data
if (head = = NULL)
head = New;
```

Otherwise suppose linked list is already created like this



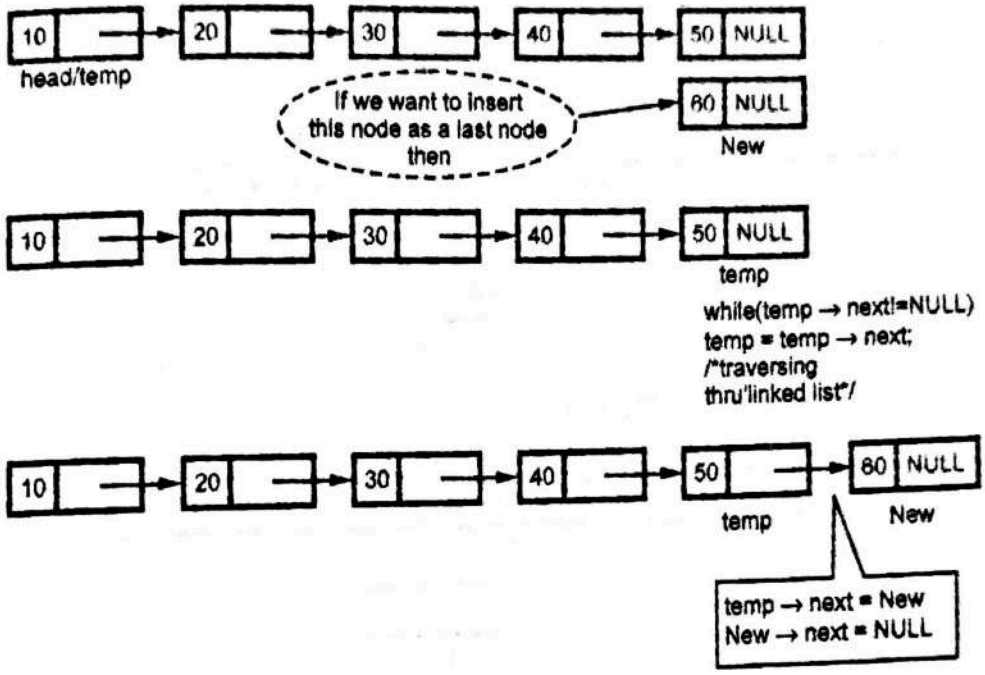
Now we will insert a node at the end -

```
void sll::insert_last()
{
 node *New,*temp;
 cout<<"\nEnter The element which you want to insert";
cin>>New->data;
if(head==NULL)
      head=New;
else
{
      temp=head;
      while(temp->next!=NULL)
            temp=temp->next;
```

Finding the end of the linked list. Then temp will be a last node.

```
    temp->next=New;
    New->next=NULL;
}
}
```

To attach a node at the end of linked list assume that we have already created a linked list like this -

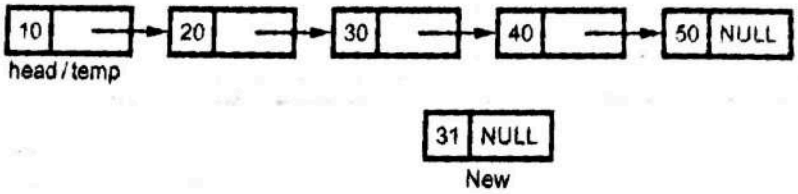


Now we will insert a new node after some node at intermediate position.

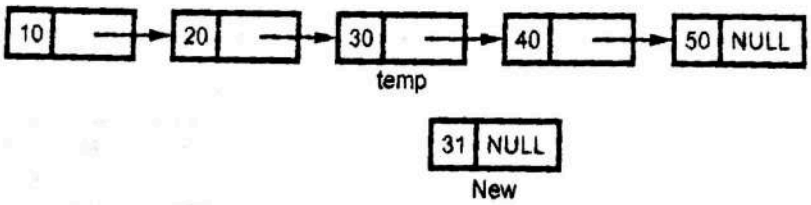
```
void sll:: insert_after()
{
int key;
node  *temp,*New;
New= new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
{
 head=New;
}
else
{
cout<<"\n Enter The element after which you want to insert the node";
cin>>key;
temp=head;
do
{
    if(temp->data==key)
    {
        New->next=temp->next;
```

```
            temp->next=New;
            break;
        }
    else
        temp=temp->next;
  }while(temp!=NULL);
 }
}
```

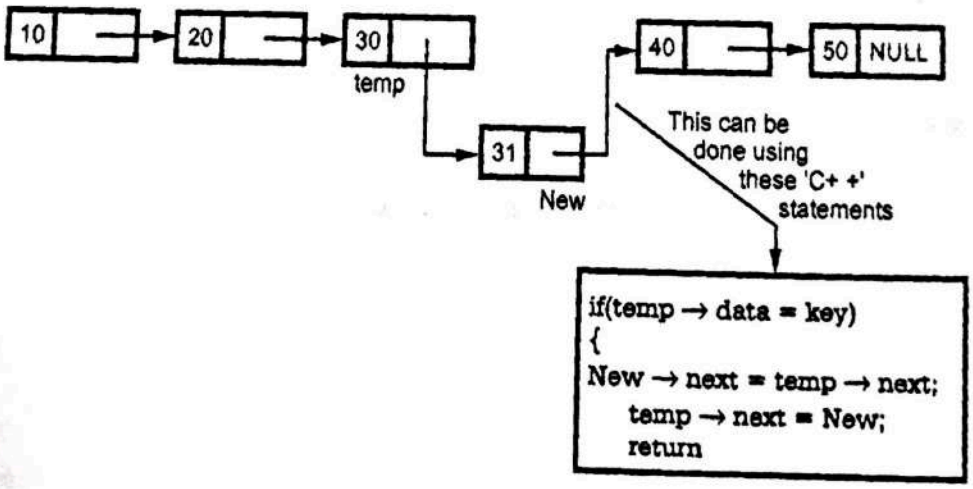
If we want to insert 31 in the linked list which is already created. We want to insert this 31 after node containing 30 then



As we will search for the value 30, key = 30.



Then,



Thus desired node gets inserted at desired position.

4. Deletion of any element from the linked list

```
void sll:: dele()
{
    node *temp, *prev ;
    int key;
    temp=head;
```

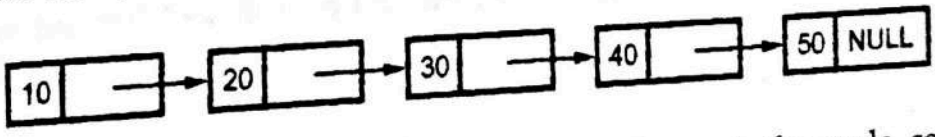
```
    clrscr();
    cout<<"\nEnter the data of the node you want to delete: ";
    cin>>key;
    while(temp!=NULL)
    {
     if(temp->data==key)
          break;
     prev=temp;
     temp=temp->next;
}
if(temp==NULL)
cout<<"\nNode not found";
else
{
     if(temp==head) //first node
          head=temp->next;
     else
          prev->next=temp->next;  //intermediate or end node
     delete temp;
     cout<<"\nThe Element is deleted\n";
     }
     getch();
}
```

Firstly Node to be deleted is searched in the linked list

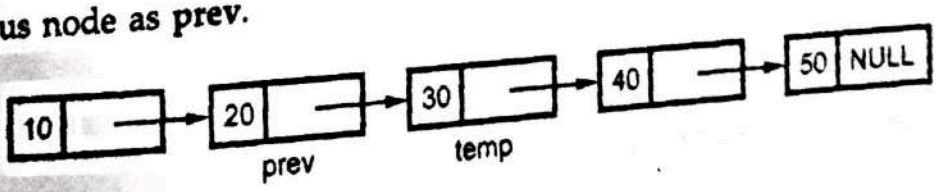
Once the node to be deleted is found then get the previous node of that node in variable **prev**

If we want to delete the head node then set its adjacent node as a new head node and then release the memory

Suppose we have,

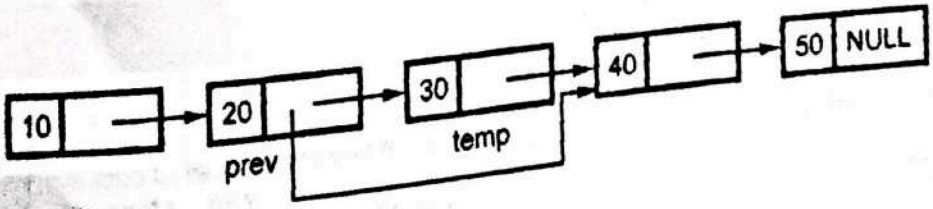


Suppose we want to delete node 30. Then we will search the node containing 30. Mark the node to be deleted as **temp**. Then we will obtain previous node of **temp**. Mark previous node as **prev**.

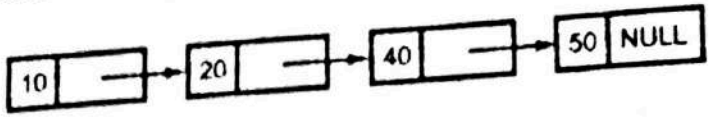


Then,

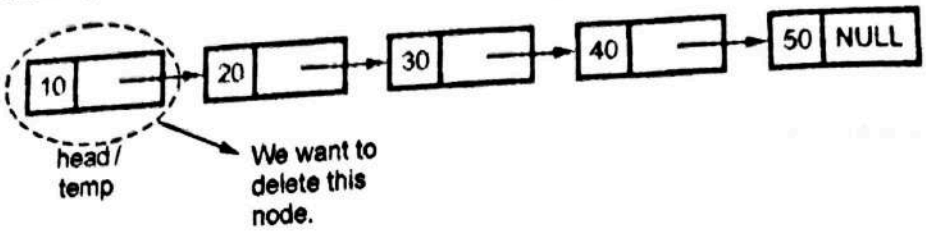
prev → next = temp → next



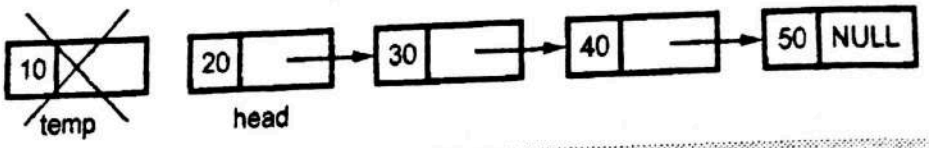
Now we will delete the temp node. Then the linked list will be



Another case is, if we want to delete a head node then -



This can be done using following statements




```
head = temp → next;
delete temp;
```

5. Searching of desired element in the linked list

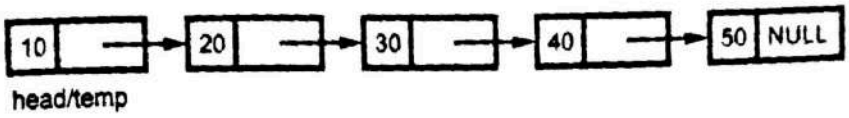
The search function is for searching the node containing desired value. We pass the head of the linked list to this routine so that the complete linked list can be searched from the beginning.

```
void all::search(int key)
{
node *temp;
int found;
temp = head;
if ( temp == NULL )
{
 cout<<"The Linked List is empty\n";
 getch();
 clrscr();
}
found = FALSE;
while ( temp != NULL && found==FALSE)
{
if ( temp->data != key)
  temp = temp -> next;
else
  found = TRUE;
}
```

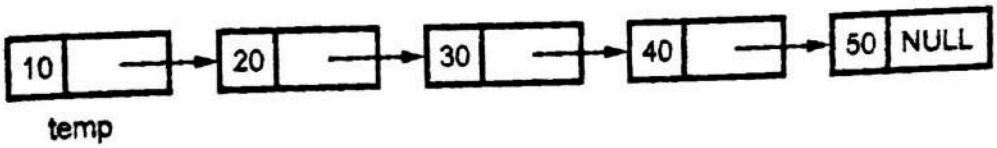
If node containing desired data is obtained in the linked list then set found variable to TURE

```
if ( found==TRUE )
{
 cout<<"\nThe Element is present in the list\n";
 getch();
}
else
{
 cout<<"The Element is not present in the list\n";
 getch();
}
}
```

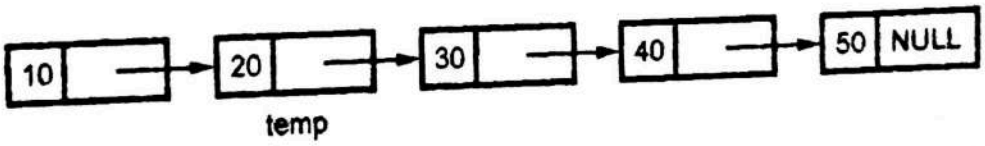
Consider that we have created a linked list as,



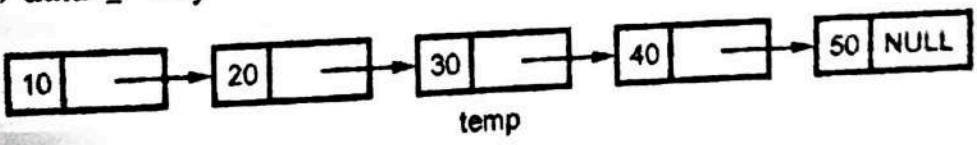
Suppose key = 30 i.e. we want a node conaining value 30 then compare **temp → data** and **key** value. If there is no match then we will mark next node as **temp**.



Is temp → data $\stackrel{?}{=}$ key ·····► No



Is temp → data $\stackrel{?}{=}$ key ·····► No



Is temp → data $\stackrel{?}{=}$ key ·····► Yes

Hence print the message "The Element is present in the list"

Thus in search operation the entire list can be scanned in search of desired node. And still, if required node is not obtained then we have to print the message.

"The Element is not present in the list"

Let us see the complete program for it -

'C++' Program

```
/*****************************************************************
 Demonstration Program to perform various operations on
 singly link lists.
*****************************************************************/

// List of include files

#include<iostream.h>
#include <conio.h>
#define TRUE 1
#define FALSE 0
// class defination

class sll
{
private:
 struct node
 {
       int data;
       structnode  *next;
 }*head;
public:
sll();
void create();
void display();
void search(int key);
void insert_head();
void insert_after();
void insert_last();
void dele();
~sll();
};

/*
-----------------------------------------------------------
The  Constructor defined
-----------------------------------------------------------
*/
sll::sll()
{
 head=NULL;//initialize head to NULL
}
/*
-----------------------------------------------------------
```

```
    The Destructor defined
-----------------------------------------------------------
*/
sll::~sll()
{
 node *temp,*temp1;
 temp=head->next;
 delete head;
 while(temp!=NULL) //free the memory allocated
 {
  temp1=temp->next;
  delete temp;
  temp=temp1;
 }
}
/*
-----------------------------------------------------------
    The Create function
-----------------------------------------------------------
*/
void sll :: create()
{
    node *temp, *New;
    int val,flag;
    char ans ='y';
    flag=TRUE;
    do
    {
        cout<<"\nEnter the data :";
        cin>>val;
        // allocate memory to new node
        New = new node;
        if ( New == NULL )
            cout<<"Unable to allocate memory\n";
        New-> data  = val ;
        New-> next = NULL;
        if( flag==TRUE ) // Executed only for the first time
        {
         head=New;
         temp = head;
         flag=FALSE;
        }
        else
        {
            /*temp last keeps track of the most recently
              created node*/
```

```
            temp->next = New;
            temp = New;
        }
        cout<<"\n Do you want to enter more elements?(y/n)";
        ans = getche();
    }while(ans==='y'||ans==='Y');
    cout<<"\nThe Singly Linked List is created\n";
    getch();
    clrscr();
}

/*
-----------------------------------------------------------
    The display function
-----------------------------------------------------------
*/
void sll ::display()
{
    node *temp ;
    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";


        getch(); clrscr();
          return;
    }
    while ( temp != NULL )
    {
        cout<<temp->data<<" ";
        temp = temp -> next;
    }
    getch();

}

void sll::search(int key)
{
 node *temp;
 int found;
 temp = head;
 if ( temp == NULL )
 {
  cout<<"The Linked List is empty\n";
  getch();
  clrscr();
```

```
}
found = FALSE;
while ( temp != NULL && found==FALSE)
{
if ( temp->data != key)
  temp = temp -> next;
else
  found = TRUE;
}
if ( found==TRUE )
{
cout<<"\nThe Element is present in the list\n";
getch();
}
else
{
cout<<"The Element is not present in the list\n";
getch();
}
}

/*
-----------------------------------------------------------
    The dele function
-----------------------------------------------------------
*/

void sll:: dele()
{
    node *temp, *prev ;
    int  key;
    temp=head;
    clrscr();
    cout<<"\nEnter the data of the node you want to delete: ";
    cin>>key;
    while(temp!=NULL)
    {
     if(temp->data==key)//traverse till required node to delete
     break;              //is found
     prev=temp;
     temp=temp->next;
}
if(temp==NULL)
cout<<"\nNode not found";
else
{
     if(temp==head) //first node
```

```
            head=temp->next;
        else
            prev->next=temp->next;  //intermediate or end node
        delete temp;
        cout<<"\nThe Element is deleted\n";
        }

        getch();

}

/*
-----------------------------------------------------------
        Function to insert at end
-----------------------------------------------------------
*/
void sll::insert_last()
{
node *New,*temp;
cout<<"\nEnter The element which you want to insert";
cin>>New->data;
if(head==NULL)
        head=New;
else
{
        temp=head;
        while(temp->next!=NULL)
                temp=temp->next;
        temp->next=New;
        New->next=NULL;
}
}

/*
-----------------------------------------------------------
        Function to insert after a node
-----------------------------------------------------------
*/
void sll:: insert_after()
{
int key;
node  *temp,*New;
New= new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
{
```

```
head=New;
}
else
{
cout<<"\n Enter The element after which you want to insert the node";
cin>>key;
temp=head;
do
{
    if(temp->data==key)
    {
        New->next=temp->next;
        temp->next=New;
        break;
    }
  else
      temp=temp->next;
}while(temp!=NULL);
}
}
/*
-----------------------------------------------------------
Function to insert at the beginning
-----------------------------------------------------------
*/

void sll:: insert_head()
{
node *New,*temp;
New=new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
      head=New;
else
{
temp=head;
New->next=temp;
head=New;
}
}
/*
-----------------------------------------------------------
    The main function
    Input : None
    Output: None
    Parameter Passing Method : None
```

```
---------------------------------------------------------------
*/
void main()
{
    sll s;
    int choice,val,ch1;
    char ans = 'y';
    do
    {
    clrscr();
    cout<<"\nProgram to Perform Various operations on Linked List";
    cout<<"\n1.Create";
    cout<<"\n2.Display";
    cout<<"\n3.Search";
    cout<<"\n4.Insert an element in a list";
    cout<<"\n5.Delete an element from list";
    cout<<"\n6.Quit";
    cout<<"\nEnter Your Choice ( 1-6) ";
    cin>>choice;
    switch( choice)
    {
        case 1:  s.create();
                 break;

        case 2:  s.display();
                 break;
        case 3:  cout<<"Enter the element you want to search";
                 cin>>val;
                 s.search(val);
                 break;
        case 4:  clrscr();
                 cout<<"\nThe list is:\n";
                 s.display();
                 cout<<"\nMenu";
                 cout<<"\n1.Insert at beginning\n2.Insert after";
                        cout<<"\n3.Insert at end";
                        cout<<"\nEnter your choice";
                        cin>>ch1;
                        switch(ch1)
                        {
                        case 1:s.insert_head();
                                break;
                        case 2:s.insert_after();
                                break;
                        case 3:s.insert_last();
                                break;
                        default:cout<<"\nInvalid choice";
```

```
                    }
                break;
    case 5: s.dele();
            break;
    default: cout<<"\nInvalid choice";
    }
    cout<<"\nContinue?";
    cin>>ans;
    }while(ans=='y'||ans=='Y');
    getch();
    return;
}
```

Output

```
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1-6) 1

Enter the data :10

Do you want to enter more elements?(y/n)y
Enter the data :20

Do you want to enter more elements?(y/n)y
Enter the data :30

Do you want to enter more elements?(y/n)y
Enter the data :40

Do you want to enter more elements?(y/n)y
Enter the data :50

Do you want to enter more elements?(y/n)n
The Singly Linked List is created
Continue?
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1-6) 2
```

```
10 20 30 40 50
Continue?

Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1-6) 3
Enter the element you want to search 30

The Element is present in the list
```

1.12.3 Other Linked List Operations

Example 1.12.1 *Given a singly linked list, write a function swap to swap every two nodes e.g. 1->2->3->4->5->6 should become 2->1->4->3->6->5.*

Solution : Assumption

1. The linked is created using create routine.
2. The display routine is written to display the nodes of the linked list.
3. The swap routine for swapping two nodes is as follows -

```
void swap(node **head)
{

  // If linked list is empty or there is only one node in list
  if (*head == NULL || (*head)->next == NULL)
     return;

  node *prev = *head;//maintaining the previous node
  node *current = (*head)->next;//and curret node.
     //Thus two nodes are focused at a time

  *head = current; // Change head before proceeding

  // Traverse the list
  while (true)
  {
     node *temp = current->next;
     current->next = prev; // Change next of current as previous node

     if (temp == NULL || temp->next == NULL)
     {
```

Part IV: Linked Organization

1.10 Concept of Linked Organization

SPPU : Dec.-05,08,13, May-06,09, Marks 6

- **Linked List is referred as the Linked organization.**
- **In this organization, data is arranged in a linked list.**
- **The linked organization is also called the dynamic data structure because the nodes in the linked list are allocated or de-allocated as per the requirement.**
- **The dynamic memory allocation functions supports the creation of linked list.**

Advantages of linked organization

1. In linked organization, insertion and deletion of elements can be efficiently done.
2. There is no wastage of memory. The memory can be allocated and deallocated as per requirement.

Disadvantages of linked organization

1. Linked organization does not support random or direct access.
2. Each data field should be supported by a link field to point to next node.

Review Questions

1. *What is a dynamic data structure ? List the advantages of linked lists.*

   SPPU : Dec.-05, May-06,09, Marks 6

2. *Write advantages of linked momory organization.*

   SPPU : Dec.-08, 13, Marks 3

1.11 Linked List as ADT

SPPU : Dec.-16, Marks 3

AbstractDataType LinkedList

{

Instances : List is a collection of elements which are arranged in a linear manner.

Operations : Various operations that can be carried out on list are -

1. **Insertion :** This operation is for insertion of element in the list.
2. **Deletion :** This operation removed the element from the list.
3. **Searching :** Based on the value of the key element the desired element can be searched.
4. **Modification :** The value of the specific element can be changed without changing its location.
5. **Display :** The list can be displayed in forward or in backward manner.

}

Review Question

1. *Explain with suitable example : Linked List as an ADT.* **SPPU : Dec.-16, Marks 3**

1.12 Singly Linked List

1.12.1 Representation of Linked List

Representation of Linked List using C++

```
class node
{
 public:
        int data;
        node *next;
};
class sll
{
 private:
        node *head;
public:
        void create();
        void print();
        .
        .
        .
};
```

Data members of list

Operations on list

1.12.2 Singly Linked List Operations

Various operations that can be performed on list are -

1. Creation
2. Insertion
3. Deletion
4. Reverse
5. Search
6. Display

Insertion and Deletion

1. Creation of linked list

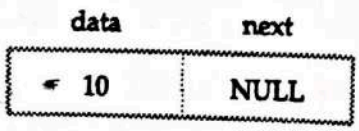
```
void sll :: create()
{
    node *temp, *New;
    int val,flag;
    char ans ='y';
    flag=TRUE;
    do
```

```
    {
        cout<<"\nEnter the data :";
        cin>>val;
        // allocate memory to new node
        New = new node;
        if ( New == NULL )
            cout<<"Unable to allocate memory\n";
        New-> data  = val ;
        New-> next = NULL;
        if( flag==TRUE ) // Executed only for the first time
        {
        head=New;
        temp = head;
        flag=FALSE;
        }
        else
        {
            /*temp last keeps track of the most recently
            created node*/
          temp->next = New;
          temp = New;
        }
        cout<<"\n Do you want to enter more elements?(y/n)";
        ans = getche();
    }while(ans=='y'||ans=='Y');
    cout<<"\nThe Singly Linked List is created\n";
    getch();
    clrscr();
}
```

Creation of linked list (logic explanation part) :

Initially one variable flag is taken whose value is initialized to TRUE (i.e. 1). The purpose of flag is for making a check on creation of first node. That means if flag is TRUE then we have to create head node or first node of linked list. Naturally after creation of first node we will reset the flag (i.e. assign FALSE to flag) Consider that we have entered the element value 10 initially then,

Step 1 :



New

```
New = new node ;
/* memory gets allocated for
New node */
New → data = Val;
/* value 10 will be put in data field of New */
```

Step 2 :

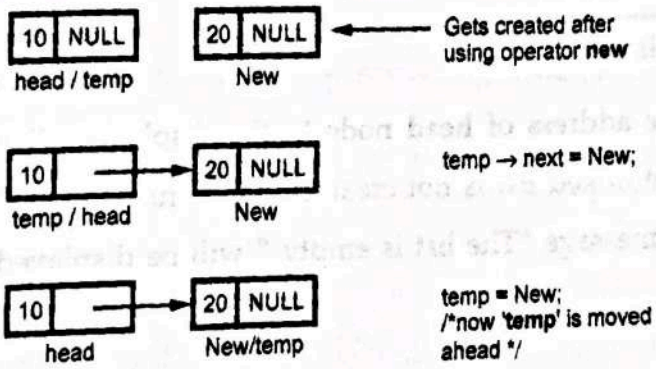
data	next
10	NULL

New/
head/temp

```
if (flag = = TRUE)
{
head = New
temp = head;
/* We have also called this node as temp because head's
address will be preserved in 'head' and we can change
'temp' node as per requirement */
flag = FALSE;
/* After creation of first node flag is reset */
```

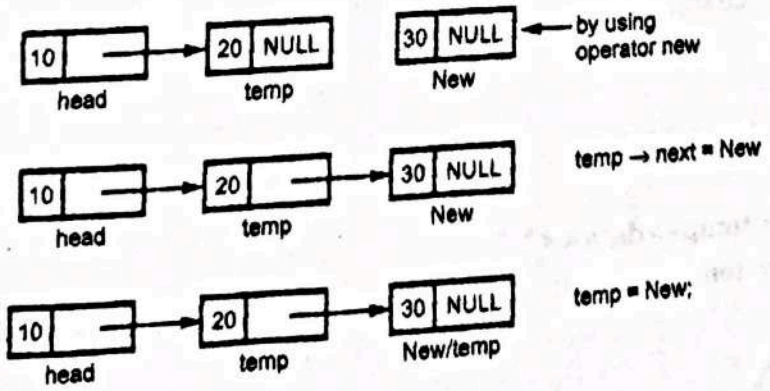
Step 3 :

If head node of linked list is created we can further create a linked list by attaching the subsequent nodes. Suppose we want to insert a node with value 20 then,



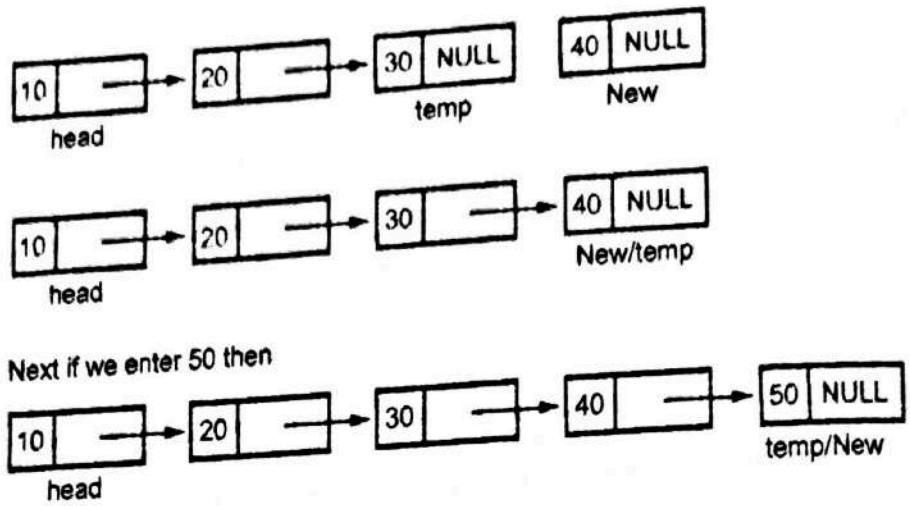
Step 4 :

If user wants to enter more elements then let say for value 30 the scenario will be,



Then for value 40 -

Step 5 :



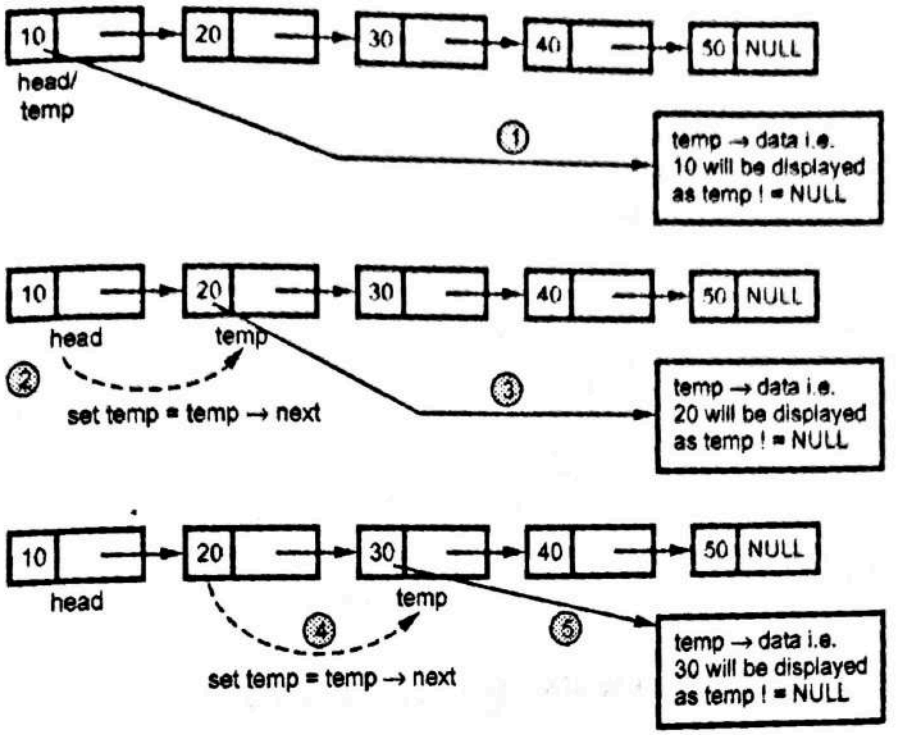
is the final linked list.

2. Display of linked list

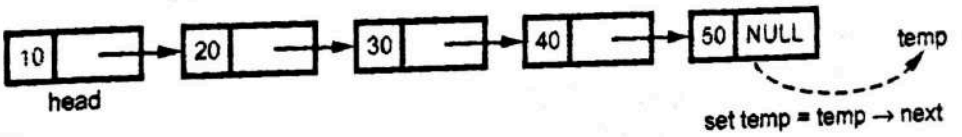
We are passing the address of **head** node to the display routine and calling head as the 'temp' node. If the linked list is not created then naturally **head = temp** node will be NULL. Therefore the message "The list is empty " will be displayed.

```
void sll ::display()
{
    node *temp ;
    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";
        getch(); clrscr();
        return;
    }
    while ( temp != NULL )
    {
        cout<<temp->data<<" ";
        temp = temp -> next;
    }
    getch();

}
```

If we have created some linked list like this then -



Continuing in this fashion we can display remaining nodes 40, 50. When



As now value of temp becomes NULL we will come out of while loop. As a result of such display routine we will get,

10 → 20 → 30 → 40 → 50 → NULL

will be printed on console.

3. Insertion of any element at anywhere in the linked list

There are three possible cases when we want to insert an element in the linked list -

a) Insertion of a node as a head node

b) Insertion of a node as a last node

c) Insertion of a node after some node.

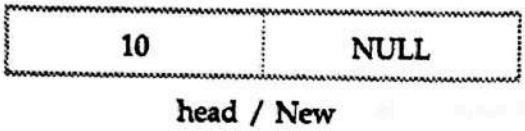
We will see the case a) first -

```
void sll:: insert_head()
{
node *New,*temp;
 New=new node;
cout<<"\n Enter The element which you want to insert";
```

```
cin>>New->data;
if(head==NULL)
        head=New;
else
{
temp=head;
New->next=temp;
head=New;
}
}
```

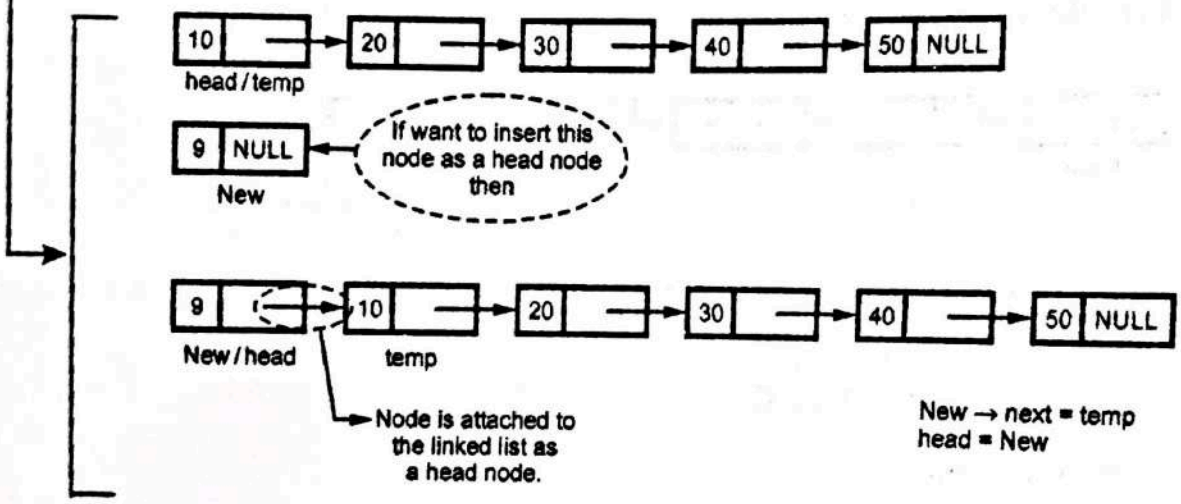
There is no node in the linked list. That means the linked list is empty

If there is no node in the linked list then value of head is NULL. At that time i we want to insert 10 then




```
Cin >>New → data
if (head = = NULL)
head = New;
```

Otherwise suppose linked list is already created like this



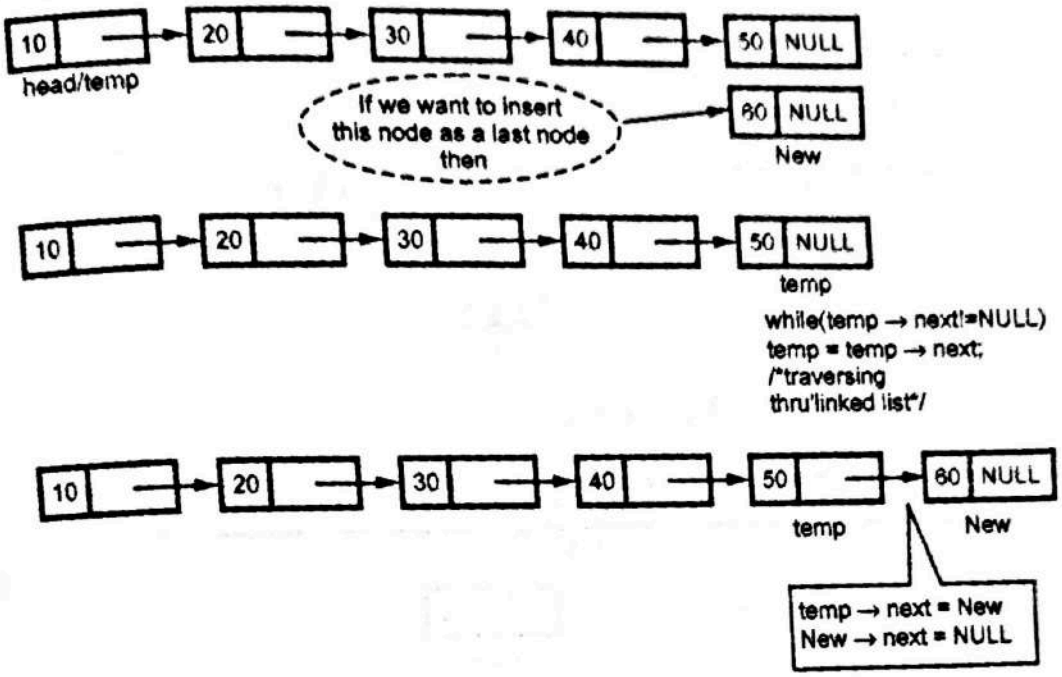
Now we will insert a node at the end -

```
void sll::insert_last()
{
 node *New,*temp;
 cout<<"\nEnter The element which you want to insert";
cin>>New->data;
if(head==NULL)
      head=New;
else
{
      temp=head;
      while(temp->next!=NULL)
            temp=temp->next;
```

Finding the end of the linked list. Then temp will be a last node.

```
    temp->next=New;
    New->next=NULL;

}
}
```

To attach a node at the end of linked list assume that we have already created a linked list like this -

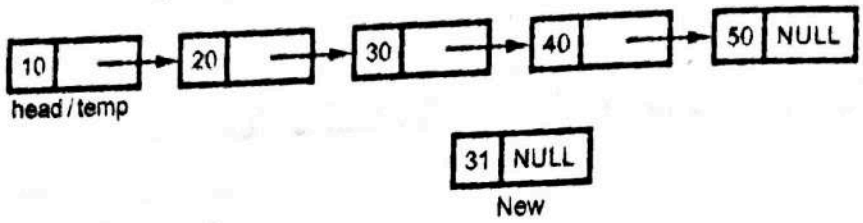


Now we will insert a new node after some node at intermediate position.

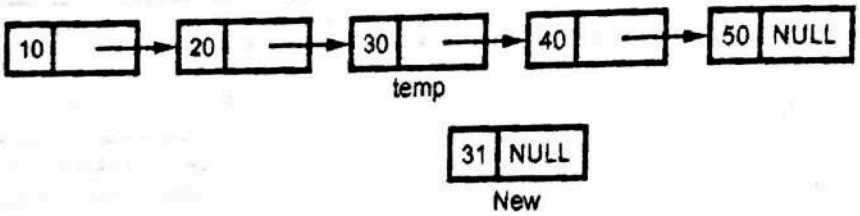
```
void sll:: insert_after()
{
int key;
node *temp,*New;
New= new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
{
 head=New;
}
else
{
 cout<<"\n Enter The element after which you want to insert the node";
 cin>>key;
 temp=head;
 do
 {
     if(temp->data==key)
     {
         New->next=temp->next;
```

```
            temp->next=New;
            break;
        }
    else
        temp=temp->next;
  }while(temp!=NULL);
 }
}
```

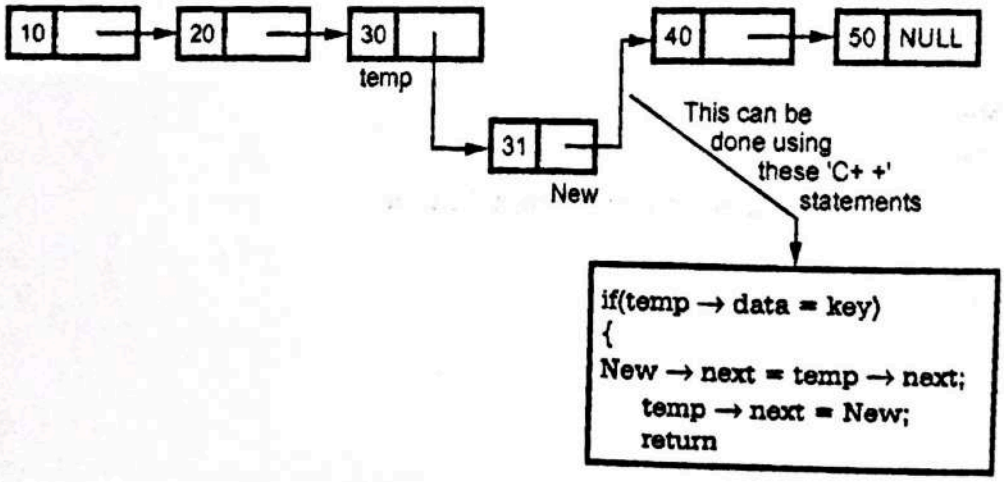
If we want to insert 31 in the linked list which is already created. We want to insert this 31 after node containing 30 then



As we will search for the value 30, key = 30.



Then,



Thus desired node gets inserted at desired position.

4. Deletion of any element from the linked list

```
void sll:: dele()
{
    node *temp, *prev ;
    int  key;
    temp=head;
```

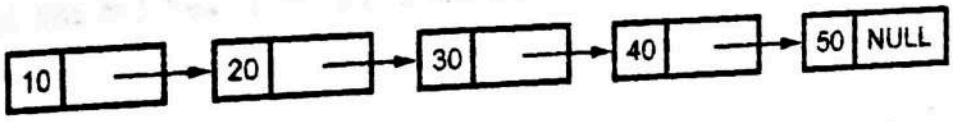
```
    clrscr();
    cout<<"\nEnter the data of the node you want to delete: ";
    cin>>key;
    while(temp!=NULL)
    {
     if(temp->data==key)
          break;
     prev=temp;
     temp=temp->next;
}
if(temp==NULL)
cout<<"\nNode not found";
else
{
     if(temp==head) //first node
          head=temp->next;
     else
          prev->next=temp->next; //intermediate or end node
     delete temp;
     cout<<"\nThe Element is deleted\n";
     }
     getch();

}
```

Firstly Node to be deleted is searched in the linked list

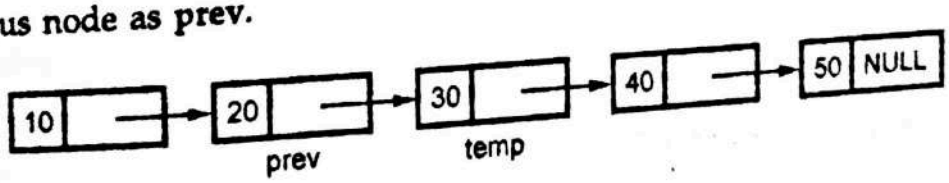
Once the node to be deleted is found then get the previous node of that node in variable **prev**

If we want to delete the head node then set its adjacent node as a new head node and then release the memory

Suppose we have,

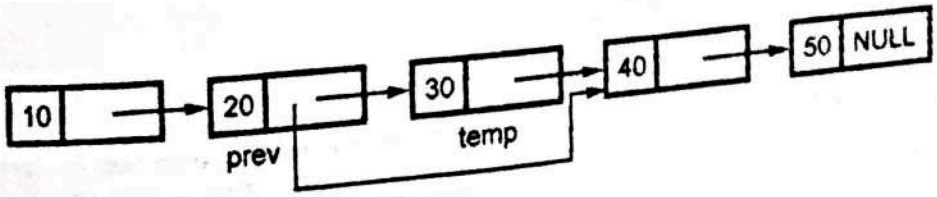


Suppose we want to delete node 30. Then we will search the node containing 30. Mark the node to be deleted as **temp**. Then we will obtain previous node of **temp**. Mark previous node as **prev**.

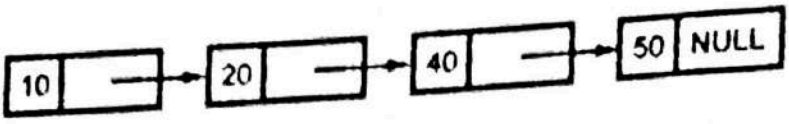


Then,

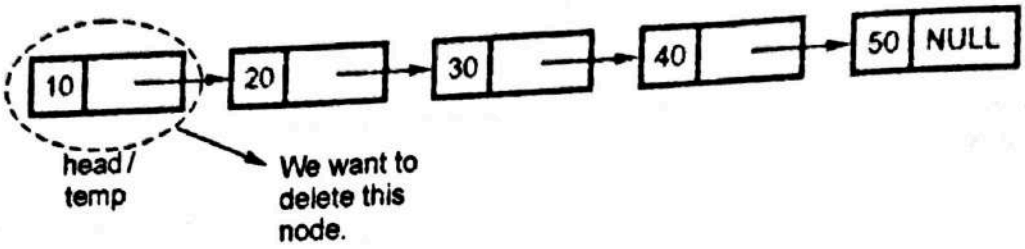
prev → next = temp → next



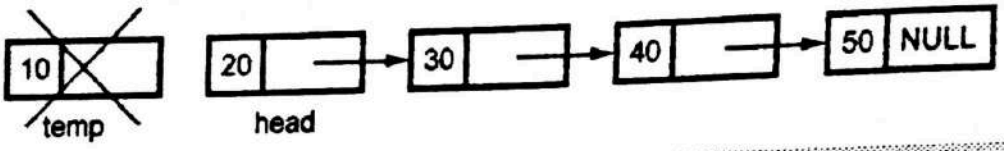
Now we will delete the temp node. Then the linked list will be



Another case is, if we want to delete a head node then -



This can be done using following statements




```
head = temp -> next;
delete temp;
```

5. Searching of desired element in the linked list

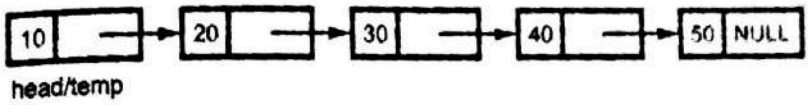
The search function is for searching the node containing desired value. We pass the head of the linked list to this routine so that the complete linked list can be searched from the beginning.

```
void all::search(int key)
{
 node *temp;
 int found;
 temp = head;
 if ( temp == NULL )
 {
  cout<<"The Linked List is empty\n";
  getch();
  clrscr();
 }
 found = FALSE;
 while ( temp != NULL && found==FALSE)
 {
  if ( temp->data != key)
   temp = temp -> next;
  else
   found = TRUE;
 }
```

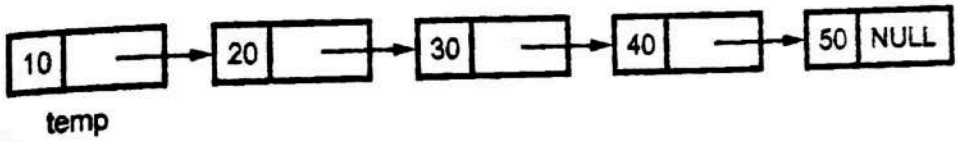
If node containing desired data is obtained in the linked list then set found variable to TURE

```
if ( found==TRUE )
{
cout<<"\nThe Element is present in the list\n";
getch();
}
else
{
cout<<"The Element is not present in the list\n";
getch();
}
}
```

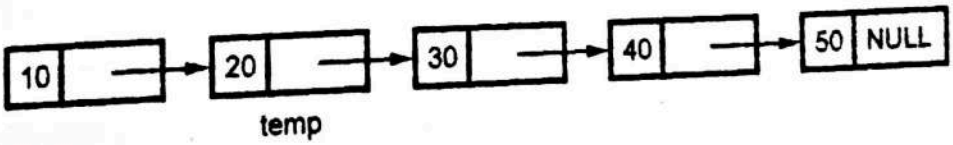
Consider that we have created a linked list as,



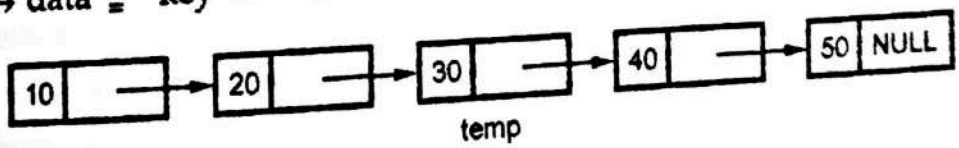
Suppose key = 30 i.e. we want a node conaining value 30 then compare temp → data and key value. If there is no match then we will mark next node as temp.



Is temp → data $\stackrel{?}{=}$ key ·····► No



Is temp → data $\stackrel{?}{=}$ key ·····► No



Is temp → data $\stackrel{?}{=}$ key ·····► Yes

Hence print the message **"The Element is present in the list"**

Thus in search operation the entire list can be scanned in search of desired node. And still, if required node is not obtained then we have to print the message.

"The Element is not present in the list"

Let us see the complete program for it -

'C++' Program

```
/*****************************************************************
 Demonstration Program to perform various operations on
 singly link lists.
*****************************************************************/

// List of include files

#include<iostream.h>
#include <conio.h>
#define TRUE 1
#define FALSE 0
// class defination

class sll
{
private:
 struct node
 {
      int data;
      structnode  *next;
 }*head;
public:
sll();
void create();
void display();
void search(int key);
void insert_head();
void insert_after();
void insert_last();
void dele();
~sll();
};

/*
-----------------------------------------------------------
The  Constructor defined
-----------------------------------------------------------
*/
sll::sll()
{
head=NULL;//initialize head to NULL
}
/*
-----------------------------------------------------------
```

```
    The Destructor defined
-----------------------------------------------------------
*/
sll::~sll()
{
 node *temp,*temp1;
 temp=head->next;
 delete head;
 while(temp!=NULL) //free the memory allocated
 {
  temp1=temp->next;
  delete temp;
  temp=temp1;
 }
}
/*
-----------------------------------------------------------
    The Create function
-----------------------------------------------------------
*/
void sll :: create()
{

    node *temp, *New;
    int val,flag;
    char ans ='y';
    flag=TRUE;
    do
    {
        cout<<"\nEnter the data :";
        cin>>val;
        // allocate memory to new node
         New = new node;
        if ( New == NULL )
              cout<<"Unable to allocate memory\n";
        New-> data  = val ;
        New-> next = NULL;
        if( flag==TRUE ) // Executed only for the first time
        {
         head=New;
         temp = head;
         flag=FALSE;
        }
        else
        {
              /*temp last keeps track of the most recently
                created node*/
```

```
            temp->next = New;
            temp = New;
        }
        cout<<"\n Do you want to enter more elements?(y/n)";
        ans = getche();
    }while(ans==='y'||ans==='Y');
    cout<<"\nThe Singly Linked List is created\n";
    getch();
    clrscr();
}

/*
-----------------------------------------------------
    The display function
-----------------------------------------------------
*/
void sll ::display()
{
    node *temp ;
    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";


        getch(); clrscr();
          return;
    }
    while ( temp != NULL )
    {
        cout<<temp->data<<" ";
        temp = temp -> next;
    }
    getch();

}

void sll::search(int key)
{
 node *temp;
 int found;
 temp = head;
 if ( temp == NULL )
 {
 cout<<"The Linked List is empty\n";
 getch();
 clrscr();
```

```
}
found = FALSE;
while ( temp != NULL && found==FALSE)
{
if ( temp->data != key)
  temp = temp -> next;
else
  found = TRUE;
}
if ( found==TRUE )
{
cout<<"\nThe Element is present in the list\n";
getch();
}
else
{
cout<<"The Element is not present in the list\n";
getch();
}
}

/*
-----------------------------------------------------------
     The dele function
-----------------------------------------------------------
*/

void sll:: dele()
{
     node *temp, *prev ;
     int key;
     temp=head;
     clrscr();
     cout<<"\nEnter the data of the node you want to delete: ";
     cin>>key;
     while(temp!=NULL)
     {
      if(temp->data==key)//traverse till required node to delete
      break;              //is found
      prev=temp;
      temp=temp->next;
     }
     if(temp==NULL)
     cout<<"\nNode not found";
     else
     {
      if(temp==head) //first node
```

```
            head=temp->next;
        else
            prev->next=temp->next;  //intermediate or end node
    delete temp;
    cout<<"\nThe Element is deleted\n";
    }

    getch();

}

/*
-----------------------------------------------------------
        Function to insert at end
-----------------------------------------------------------
*/
void sll::insert_last()
{
node *New,*temp;
cout<<"\nEnter The element which you want to insert";
cin>>New->data;
if(head==NULL)
    head=New;
else
{
    temp=head;
    while(temp->next!=NULL)
        temp=temp->next;
    temp->next=New;
    New->next=NULL;
}
}

/*
-----------------------------------------------------------
        Function to insert after a node
-----------------------------------------------------------
*/
void sll:: insert_after()
{
int key;
node  *temp,*New;
New= new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
{
```

```
head=New;
}
else
{
cout<<"\n Enter The element after which you want to insert the node";
cin>>key;
temp=head;
do
{
    if(temp->data==key)
    {
        New->next=temp->next;
        temp->next=New;
        break;
    }
  else
      temp=temp->next;
}while(temp!=NULL);
}
}
/*
-----------------------------------------------------------
Function to insert at the beginning
-----------------------------------------------------------
*/

void sll:: insert_head()
{
node *New,*temp;
New=new node;
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
      head=New;
else
{
temp=head;
New->next=temp;
head=New;
}
}
/*
-----------------------------------------------------------
    The main function
    Input : None
    Output: None
    Parameter Passing Method : None
```

```
*/
void main()
{
    sll s;
    int choice,val,ch1;
    char ans = 'y';
    do
    {
    clrscr();
    cout<<"\nProgram to Perform Various operations on Linked List";
    cout<<"\n1.Create";
    cout<<"\n2.Display";
    cout<<"\n3.Search";
    cout<<"\n4.Insert an element in a list";
    cout<<"\n5.Delete an element from list";
    cout<<"\n6.Quit";
    cout<<"\nEnter Your Choice ( 1-6) ";
    cin>>choice;
    switch( choice)
    {
        case 1:  s.create();
                 break;

        case 2:  s.display();
                 break;
        case 3:  cout<<"Enter the element you want to search";
                 cin>>val;
                 s.search(val);
                 break;
        case 4:  clrscr();
                 cout<<"\nThe list is:\n";
                 s.display();
                 cout<<"\nMenu";
                 cout<<"\n1.Insert at beginning\n2.Insert after";
                 cout<<"\n3.Insert at end";
                 cout<<"\nEnter your choice";
                 cin>>ch1;
                 switch(ch1)
                 {
                 case 1:s.insert_head();
                        break;
                 case 2:s.insert_after();
                        break;
                 case 3:s.insert_last();
                        break;
                 default:cout<<"\nInvalid choice";
```

```
                    }
                break;
    case 5: s.dele();
            break;
    default: cout<<"\nInvalid choice";
    }
    cout<<"\nContinue?";
    cin>>ans;
    }while(ans==='y'||ans==='Y');
    getch();
    return;
}
```

Output

```
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1-6) 1

Enter the data :10

Do you want to enter more elements?(y/n)y
Enter the data :20

Do you want to enter more elements?(y/n)y
Enter the data :30

Do you want to enter more elements?(y/n)y
Enter the data :40

Do you want to enter more elements?(y/n)y
Enter the data :50

Do you want to enter more elements?(y/n)n
The Singly Linked List is created
Continue?
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1-6) 2
```

Example 1.12.5 *Write an algorithm to find the location of an element in the given linked list. Is the binary search will be suitable for this search ? Explain the reason.*

Solution :

```
void search_element(node *head,int key)
{
 node *Temp;
 int count=1;
 //head is a starting node of the linked list
 //key is the element that is to be searched in the linked list.
 Temp=head;
 while(Temp->next!=NULL)
 {
  if(temp->data==key)
  {
   printf("\n The element is present at location %d",count);
   break;
  }
  count++;
  Temp=Temp->next;
 }
}
```

The binary search can be suitable for this algorithm only if the elements in the linked list are arranged in sorted order. Otherwise this method will not be suitable because sorting the linked list is not efficient as it requires the swapping of the pointers.

1.13 Doubly Linked List

SPPU : May-06, 07, 10, 14, Dec.-05, 08, 10, March-10

The typical structure of each node in doubly linked list is like this.

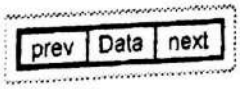
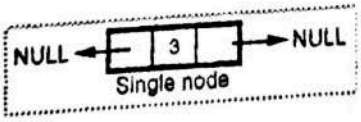


Fig. 1.13.1

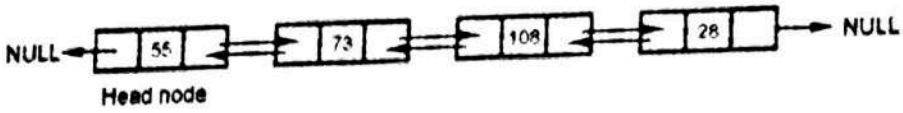


'C++' structure of doubly linked list :

```
typedef struct node
{
    int Data;
    struct node *prev;
    struct node *next;
}
```

The linked representation of a doubly linked list is

Thus the doubly linked list can traverse in both the directions, forward as well as backwards.

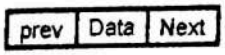


Logic for doubly linked list

In doubly linked list there are following operations

1. Create 2. Display 3. Insert 4. Delete

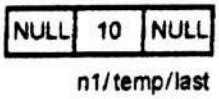
The node in doubly linked list (DLL) will look like this.



1.13.1 Operations on Doubly Linked List

1. Creation of node

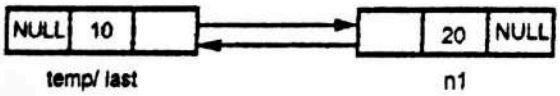
Initially set new node as,



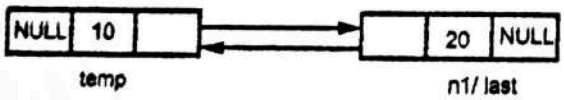
temp = n1;
last = temp;

For the first time flag = 0

Now set flag = 1, if we want to create more node then -

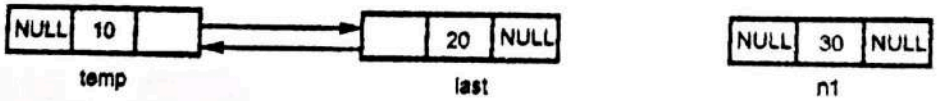


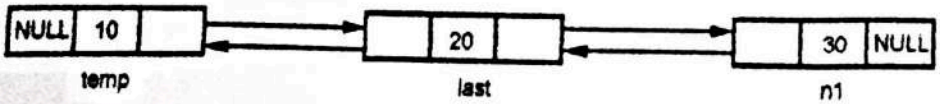
last → next = n1
n1 → prev = last



last = n1

If we want to attach node 30 then



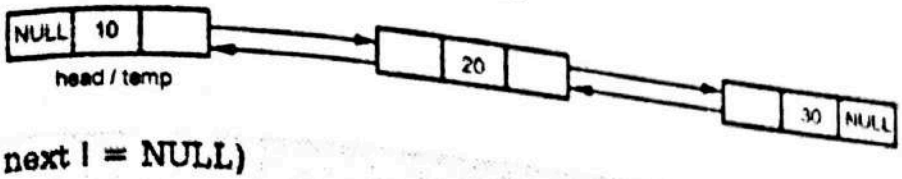


last→ next = n1
n1 → prev = last

This we can continue to create a doubly linked list.

2. Display of Doubly linked list

We assign head node address to temp node.

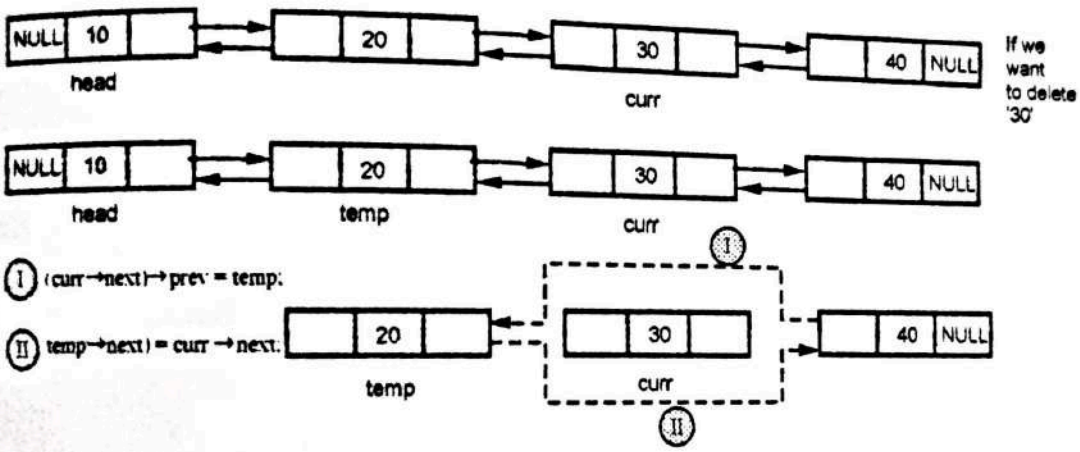



```
while (temp → next I = NULL)
{
    * display data at temp node.
    temp = temp → next
}
```

This will display 10 20 30 as a doubly linked.

3. Deletion of a node in doubly linked list

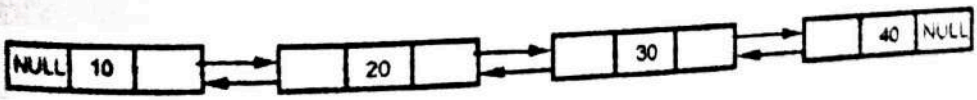
Consider,



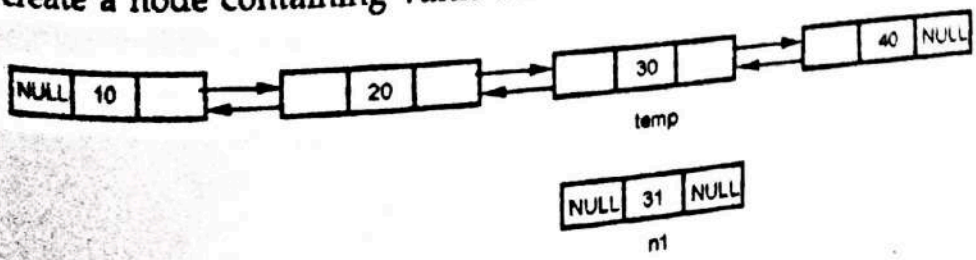
Then delete curr.

4. Insertion of a node

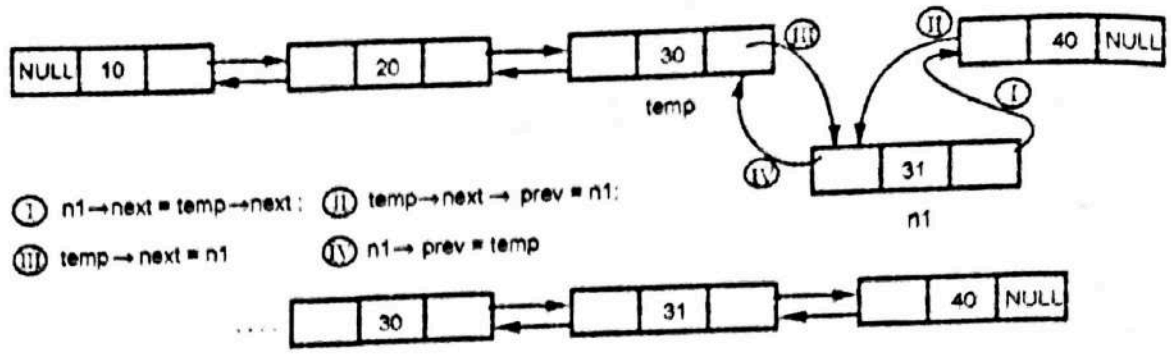
Consider a list



Suppose we want to insert 31 after 30 then first search for node 30 call it as temp node. Then create a node containing value 31.



Then



Thus "Node inserted".

```
/*****************************************************************
 Demonstration Program to perform various operations such as insertion,
 deletion,finding the lenghth of the doublly linked list.
*****************************************************************/

// List of include files
#include<iostream.h>
#include <conio.h>
// class defination
class dll
{
 private:
 struct node
 {
    int Data;
    struct  node  *next;
    struct  node  *prev;
 }*head;
 public:
 dll();
 void create();
 void print();
 void insert_beg();
 void insert_after();
 void insert_end();
 void del();
 int length();
 ~dll();
};

/*
-----------------------------------------------------------
The Constructor defined
-----------------------------------------------------------
*/
```

```
dll::dll()
{
 head=NULL;
}
/*
-----------------------------------------------------------
The destructor defined
-----------------------------------------------------------
*/
dll::~dll()
{
node *temp,*temp1; //free the memory
temp=head;
while(temp!=NULL)
{
   temp1=temp->next;
   delete temp;
   temp=temp1;
}
}
/*
-----------------------------------------------------------
    The Create function
-----------------------------------------------------------
*/
void dll :: create()
{
    // Local declarations here
    node  *n1,*last,*temp;
    char ans ='y';
    int flag=0;
    int val;
    do
    {
        cout<<"\nEnter the data :";
        cin>>val;

        // allocate new node
        n1 = new node;
        if ( n1 == NULL )
              cout<<"Unable to allocate memory\n";
        n1 -> Data  = val ;
        n1 -> next = NULL;
        n1 -> prev = NULL;
        if (flag==0) // Executed only for the first time
        {
```

```
                temp=n1;
                last=temp;
                flag=1;
            }
            else
            {
                // last keeps track of the most recently
                // created node

                last->next=n1;
                n1->prev=last;
                last=n1;
            }
            cout<<"\n\nEnter more?";
            ans = getche();
        }while(ans=='y's||ans=='Y');
        cout<<"\nThe List is created\n";
        head=temp;
        getch();
}

/*
-----------------------------------------------------------
        The length function
        Output:Returns number of items in list
        Calls : None
-----------------------------------------------------------
*/

int dll ::length()
{
        node *curr ;
        int  count;

        count = 0;
        curr = head;
        if ( curr == NULL )
        {
            cout<<"The list is empty\n";
            getch();
            return 0;
        }
        while ( curr != NULL )
        {
            count++;
            curr = curr -> next; // traverse till end
        }
```

```
    getch();
    return count;

}

/*
    The print function
    Output:None, Displays data
    Calls : None

*/
void dll ::print()
{
    node *temp ;

    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";
        getch(); clrscr();
        return;
    }
    else
    {
    cout<<"\nThe list is:";
    while(temp != NULL)
    {
        cout<<temp->Data<<" ";
        temp = temp -> next;
    }
    }
    getch();

}

/*
    The del function
    Output:deletes an item from the list

*/

void dll:: del()
{
    node *curr, *temp;
    int  data;
    curr=head;
```

```
        cout<<"\nEnter the data of the node you want to delete: ";
        cin>>data;
        while(curr!=NULL)
        {
         if(curr->Data==data) //traverse till the required node to delete
         break;              //is found
         curr=curr->next;
        }
        if(curr==NULL)
        cout<<"\nNode not found";
        else
        {
          if(curr==head)
          {
             if(head->next==NULL&&head->prev==NULL)//only one node
                        head=NULL;
             else
             {
             head=curr->next;// first node
             head->prev=NULL;
             }
          }
          else
          {
             temp=curr->prev; //intermediate or end node
             if(curr->next!=NULL)
             (curr->next)->prev=temp;
             temp->next=curr->next;
          }
             delete curr; //free memory
             cout<<"\nThe item is deleted\n";
        }

        getch();

}

/*-----------------------------------------------------------
        Function to insert at end
-----------------------------------------------------------*/
void dll::insert_end()
{
node *temp,*n1;
int val,flag=0;
cout<<"\nEnter the data of the new node to insert";
cin>>val;
temp=head;
```

```
if(temp==NULL)
flag=1;
else
{
while(temp->next!=NULL)  // traverse till end
    temp=temp->next;

}

// allocate new node
n1 = new node;
if ( n1 == NULL )
    cout<<"\nUnable to allocate memory\n";
n1-> Data  = val ;
n1 -> next = NULL;
n1 -> prev = NULL;
if(flag==0)
{
temp->next=n1;  // attach at end
n1->prev=temp;
}
else
head=n1;      //if list empty make this node as head node
cout<<"\nNode inserted";
}


/*-----------------------------------------------------------
    Function to insert after a node
-----------------------------------------------------------*/
void dll:: insert_after()
{
node *temp,*n1;
int val,val1;
cout<<"\nEnter the data of the new node to insert";
cin>>val;
cout<<"\nEnter the data of the node after which to insert";
cin>>val1;
temp=head;
while(temp!=NULL)
{
 if(temp->Data==val1) //traverse till the required node after which to insert
      break;
 temp=temp->next;
}
 if(temp!=NULL)
 {
 /* allocate new node */
```

```
 n1 = new node;
 if ( n1 == NULL )
       cout<<"Unable to allocate memory\n";
  n1 -> Data  = val ;
  n1 -> next = NULL;
  n1 -> prev = NULL;
  /* after temp attach*/
  n1->next=temp->next;
  temp->next->prev=n1;
  temp->next=n1;
  n1->prev=temp;
  cout<<"\nNode inserted";
 }
 else
 cout<<"\nNode after which to insert not found";
}
/* ----------------------------------------------------------------
Function to insert at the beginning
----------------------------------------------------------------*/

void dll:: insert_beg()
{
 node *temp,*n1;
 int data;

 cout<<"\nEnter the data of the new node to insert";
 cin>>data;

 /* allocate new node */
 n1 = new node;
 if ( n1 == NULL )
 cout<<"Unable to allocate memory\n";
  n1 -> Data  = data ;
  n1 -> next = NULL;
  n1 -> prev = NULL;
 if(head)
 {
  n1->next=head; //attach before head
  head->prev=n1;
 }
 head=n1; //make this node as head

 cout<<"\nNode inserted";
}


/*
```

```
/*
---------------------------------------------------------
    The main function
    Input : None
    Output: None
    Parameter Passing Method : None
---------------------------------------------------------
*/
void main()
{
    dll d;
    int ch,ch1,cnt;
    char ans = 'y';
    do
    {
    clrscr();
    cout<<"\n"<<"MENU";
    cout<<"\n1.Create\n2.Display\n3.Insert\n4.Delete\n5.Length";
    cout<<"\nEnter ur choice:";
    cin>>ch;
    switch(ch)
    {
    case 1: d.create();
            break;
    case 2: d.print();
            break;
    case 3: clrscr();
        d.print();
        cout<<"\nMenu";
        cout<<"\n1.Insert at beginning\n2.Insert after";
        cout<<"\n3.Insert at end";
        cout<<"\nEnter your choice";
        cin>>ch1;
        switch(ch1)
        {
        case 1: d.insert_beg();
                d.print();
                break;
        case 2: d.insert_after();
                d.print();
                break;
        case 3: d.insert_end();
                d.print();
                break;
        default:cout<<"\nInvalid choice";
        }
        break;
    case 4: d.print();
```

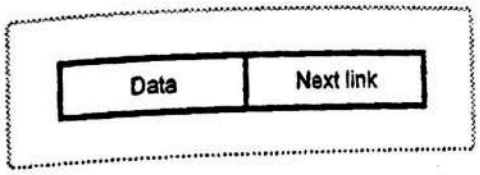
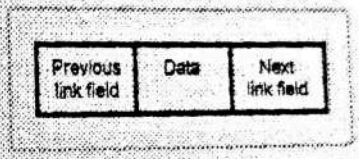
```
                d.del();
                d.print();
                break;
        case 5: cnt=d.length();
                cout<<"The length is: "<<cnt;
                break;
        default: cout<<"\nInvalid choice";
        }
        cout<<"\n Do You Want To Continue?";
        cin>>ans;
        }while(ans==='y'||ans==='Y');
        getch();
        return;
}
```

Output

```
MENU
1.Create
2.Display
3.Insert
4.Delete
5.Length
Enter ur choice:1
Enter the data :1

Enter more?y
Enter the data :2

Enter more?y
Enter the data :3

Enter more?y
Enter the data :4

Enter more?y
Enter the data :5

Enter more?n
The List is created

 Do You Want To Continue?

MENU
1.Create
2.Display
```

```
3.Insert
4.Delete
5.Length
Enter ur choice:2

The list is:1 2 3 4 5
Do You Want To Continue?y
```

Comparison between Singly and Doubly Linked List

Sr. No.	Singly Linked List	Doubly Linked List
1.	Singly linked list is a collection of nodes and each node is having one data field and next link field. For example : 	Doubly linked list is a collection of nodes and each node is having one data field, one previous link field and one next link field. For example : 
2.	The elements can be accessed using next link.	The elements can be accessed using both previous link as well as next link.
3.	No extra field is required; hence node takes less memory in SLL.	One field is required to store previous link hence node takes more memory in DLL.
4.	Less efficient access to elements.	More efficient access to elements.

Example 1.11.1 *Write a method in C++ to join two doubly linked lists into a single doubly linked list. In a join elements of second list are appended to the end of first list.*

Solution :

```
/*****************************************************************
Program to join two doubly linked list
*****************************************************************/

// List of include files

#include<iostream.h>

// class defination
class dll
{
private:
struct node
{
```

```
        int Data;
        structnode  *next;
        structnode  *prev;
}*head;
public:
dll();
void create();
void print();
void join(dll,dll);
};

/*
-----------------------------------------------------------
 Constructor
-----------------------------------------------------------
*/
dll::dll()
{
 head=NULL;
}
/*
-----------------------------------------------------------
        The Create function
-----------------------------------------------------------
*/
void dll :: create()
{
        node *New,last,*temp;
        int num,flag=0;
        char ans;
        // flag to indicate whether a new node
         //is created for the first time or not
        do
        {
                cout<<"\n Enter the Element :";
                cin>>num;
                /* allocate new node */
                New =new node;
                New -> Data  = num;
                New -> next = NULL;
                New->prev=NULL;
                if ( flag == 0 )
                {
                 head = New; // First node is created
                 temp=New;
                 flag=1;
```

```
        }
        else
        {
            temp->next=New;//remaining nodes are created
            New->prev=temp;
            temp=New;
        }
        cout<<"\n Do You Want to insert More Elements?";
        ans=getch();
    }while(ans==='y');
    cout<<"\n The List Is Created";

}

/*
    The print function
*/
void dll ::print()
{
    node *temp ;
    temp = head;
    if ( temp == NULL )
    {
        cout<<"\nThe list is empty\n";
        getch(); clrscr();
        return;
    }
    else
    {
    cout<<"\nThe list is:";
    while(temp != NULL)
    {
        cout<<temp->Data<<" ";
        temp = temp -> next;
    }
    }
    getch();
}
/*
The join Function
*/
void dll::join(dll d1,dll d2)
{
node *temp1,*temp2,*prev_node1,*next_node2;
```

```
temp1=d1.head;
temp2=d2.head;
if(temp1==NULL)
temp1=temp2;
while(temp1->next!=NULL)
{
 temp1=temp1->next;//reaching at the end of first DLL
}
temp1->next=temp2;

}
/*
-----------------------------------------------------
      The main function
-----------------------------------------------------
*/
void main()
{
    dll d1,d2;
    int ch,ch1,cnt;
    char ans = 'y';
    do
    {

      cout<<"\nEnter the data for 1 st Doubly Linked List ";
      d1.create();
      d1.print();
      cout<<"\nEnter the data for 2nd Doubly Linked List ";
      d2.create();
      d2.print();
      d1.join(d1,d2);
      d1.print();
      cout<<"\nDo you want to Continue?";
      cin>>ans;
      }while(ans=='y'||ans=='Y');
      return;
}
```

Output

```
Enter the data for 1 st Doubly Linked List
Enter the Element :10

Do You Want to insert More Elements?
Enter the Element :20

Do You Want to insert More Elements?
Enter the Element :30
```

```
Do You Want to insert More Elements?
The List Is Created
The list is:10  20  30
Enter the data for 2nd Doubly Linked List
Enter the Element :40

Do You Want to insert More Elements?
Enter the Element :50

Do You Want to insert More Elements?
The List Is Created
The list is:40  50
The list is:10  20  30  40  50
```

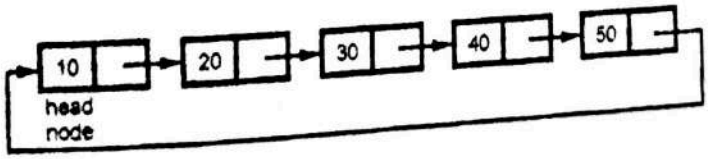
Review Questions

1. *Explain the insertion of node in doubly linked list at :*
   i) The start of the list; ii) The end of the list; iii) After the position.
   SPPU : May-06, Dec-08, Marks 8
2. *Give the structure definition to represent doubly linked list to store numbers. Compare doubly linked list with singly linked list.* SPPU : Dec-06, Marks 8
3. *Write pseudo C function to insert a node before and after any node in doubly linked list and for deletion of specified node.* SPPU : May-07, Marks 10
4. *Compare singly linked list with doubly linked list.*
   SPPU : May-10, Marks 6, Dec-10, Marks 2, May-13, Marks 6

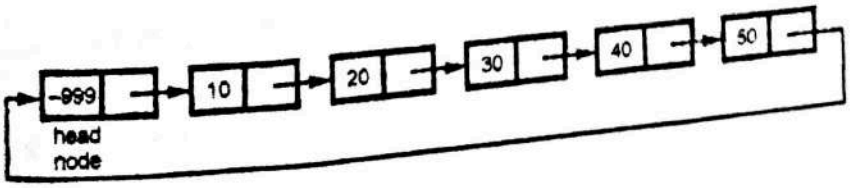
1.14 Circularly Linked List

SPPU : May-06,08,09,13,14,16, Dec-07, Marks 10

The circular linked list is as shown below -



or



The Circular Linked List (CLL) is similar to singly linked list except that the last node's next pointer points to first node.

Various operations that can be performed on circular linked list are,

1. Creation of circular linked list.
2. Insertion of a node in circular linked list.
3. Deletion of any node from linked list.
4. Display of circular linked list.

We will see each operation along with some example.

1. Creation of circular linked list

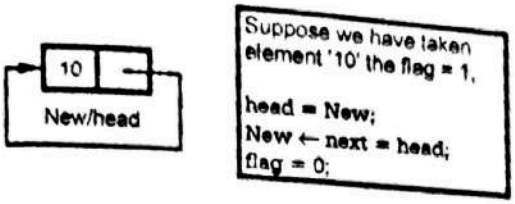
```
void sll ::Create()
{
 char ans;
 int flag=1;
 node *New,*temp;
 clrscr();
 do
 {
      New = new node;
      New->next=NULL;
      cout<<"\n\n\n\tEnter The Element\n";
      cin>>New->data;
      if(flag==1)   /*flag for setting the starting node*/
      {
            head = New;
            New->next=head;
            flag=0;/*reset flag*/
      }
      else          /* find last node in list */
      {
       temp=head;
       while (temp->next != head)/*finding the last node*/
            temp=temp->next; /*temp is a last node*/
       temp->next=New;
       New->next=head; /*each time making the list
                 circular*/
      }
      cout<<"\n Do you want to enter more nodes?(y/n)";
      ans=getch();
      }while(ans=='y'||ans=='Y');
}
```

Single node in the Circular list

Initially we will allocate memory for New node using a function **get_node()** . There is one variable **flag** whose purpose is to check whether first node is created or not. That

means flag is 1 (set) then first node is not created. Therefore after creation of first node we have to reset the flag (making flag = 0).

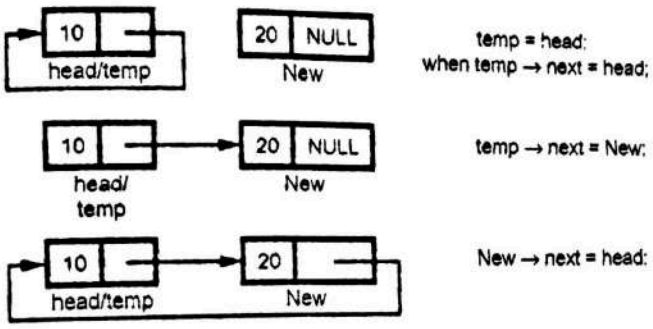
Initially,



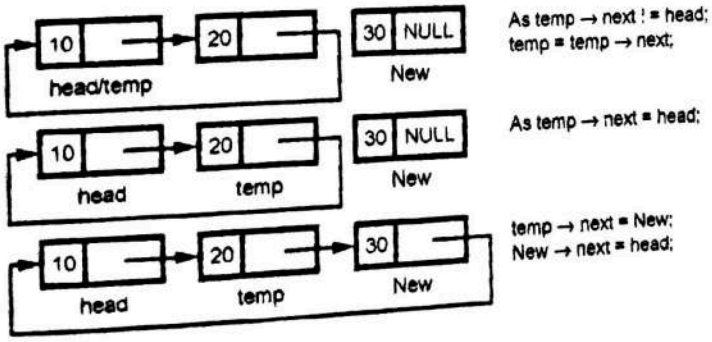
Here variable **head** indicates starting node.

Now as flag = 0, we can further create the nodes and attach them as follows.

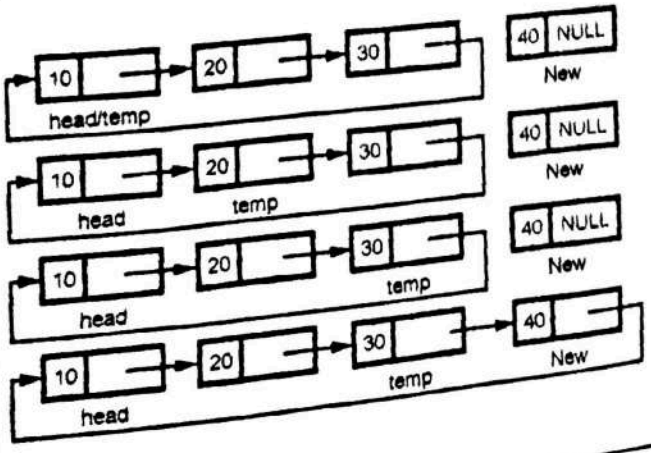
When we have taken element '20'



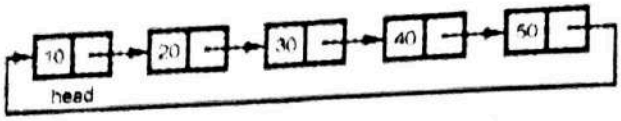
If we want to insert 30 then



If we want to insert 40 then



Thus we can create a circular linked list by inserting one more element 50. It is as shown below -



2. Display of Circular linked list

```
void sll::Display()
{
      node *temp;
      temp = head;
      if(temp == NULL)
       cout<<"\n Sorry ,The List Is Empty\n";
      else
      {
       do
       {
              cout<<"\t"<<temp->data;
              temp = temp->next;
       }while(temp != head);/*Circular linked list*/
      }
      getch();
}
```

The next node of last node is head node

3. Insertion of circular linked list

While inserting a node in the linked list, there are 3 cases -

a) inserting a node as a head node

b) inserting a node as a last node

c) inserting a node at intermediate position

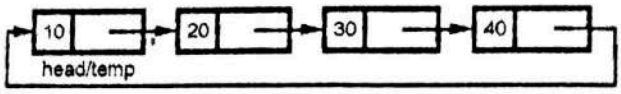
The functions for these cases is as given below -

```
void sll::insert_head()
{
 node *New,*temp;
 New=new node;
 New->next=NULL;
 cout<<"\n Enter The element which you want to insert ";
 cin>>New->data;
 if(head==NULL)
        head=New;
 else
 {
              temp=head;
              while(temp->next!=head)
                     temp=temp->next;
```

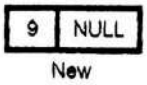
```
        temp->next=New;
        New->next=head;
        head=New;
        cout<<"\n The node is inserted!";

}
}
/*Insertion of node at last position*/
void sll::insert_last()
{
    node *New,*temp;
    New=new node;
    New->next=NULL;
    cout<<"\n Enter The element which you want to insert ";
    cin>>New->data;
    if(head==NULL)
        head=New;
    else
    {
    temp=head;
    while(temp->next!=head)
            temp=temp->next;
    temp->next=New;
    New->next=head;
    cout<<"\n The node is inserted!";
    }
}
void sll::insert_after()
{
 int key;
 node *New,*temp;
 New= new node;
 New->next=NULL;
 cout<<"\n Enter The element which you want to insert ";
 cin>>New->data;
 if(head==NULL)
 {
   head=New;
 }
 else
 {
     cout<<"\n Enter The element after which you want to insert the node ";
     cin>>key;
     temp=head;
     do
     {
         if(temp->data==key)
         {
```

```
                New->next=temp->next;
                temp->next=New;
                cout<<"\n The node is inserted";
                return;
            }
            else
                temp=temp->next;
    }while(temp!=head);
  }
}
```

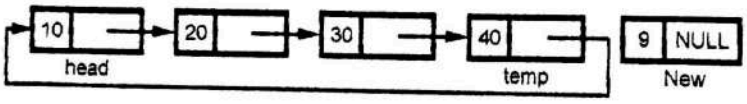
Suppose linked list is already created as -

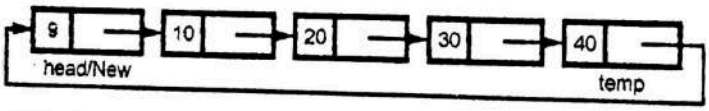


If we want to insert a New node as a head node then,



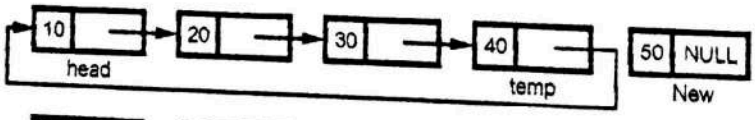
Then

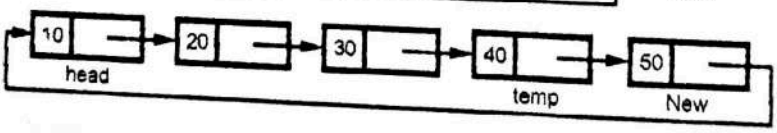




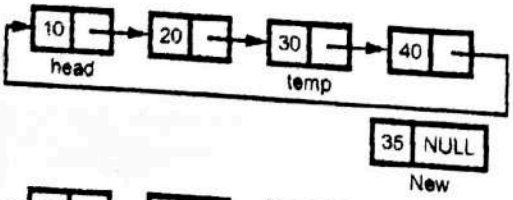
temp → next = New;
New → next = head;
head = New;

If we want to insert a New node as a last node consider a linked list

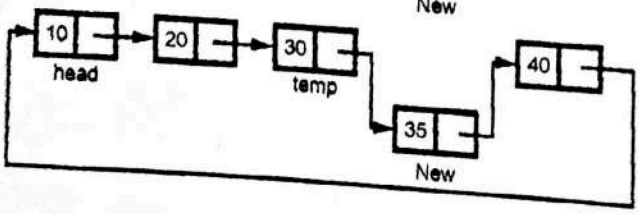




If we want to insert an element 35 after node 30 then,



As key = 30
and temp → data = 30



New → next = temp → next;
temp → next = New;

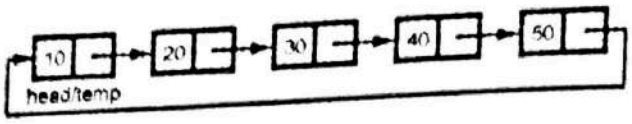
4. Deletion of any node

```
void sll::Delete()
{
int key;
struct node *temp,*temp1;
cout<<"\n Enter the element which is to be deleted";
cin>>key;
temp=head;
if(temp->data==key)/*If header node is to be deleted*/

{    temp1=temp->next;
     if(temp1==temp)
/*if single node is present in circular linked list
and we want to delete it*/
     {
        temp=NULL;
        head=temp;
        cout<<"\n The node is deleted";
     }
     else /*otherwise*/
     {
      while(temp->next!=head)
       temp=temp->next;/*searching for the last node*/
        temp->next=temp1;
        head=temp1;/*new head*/
        cout<<"\n The node is deleted";
     }
}
else
{
    while(temp->next!=head) /* if intermediate node is to
                  be deleted*/
    {
     if((temp->next)->data==key)
     {
        temp1=temp->next;
        temp->next=temp1->next;
        temp1->next=NULL;
        delete temp1;
        cout<<"\n The node is deleted";
     }
     else
        temp=temp->next;
    }
}
}
```

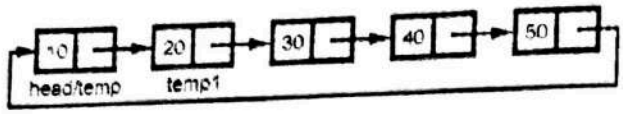
If a single node is present in the list and we want to delete it

The previous node of the node to be deleted is searched. Here **temp 1** is the node to be deleted and **temp** is the previous node of **temp 1**

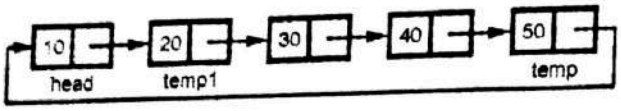
Suppose we have created a linked list as,



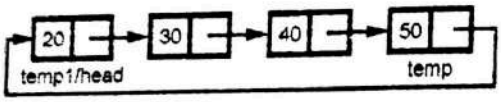
If we want to delete temp → data i.e. node 10 then,



temp1 = temp → next;

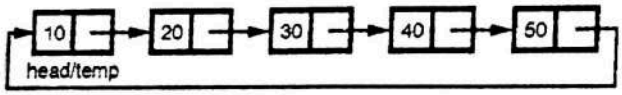


while (temp → next ! head)
temp = temp → next;

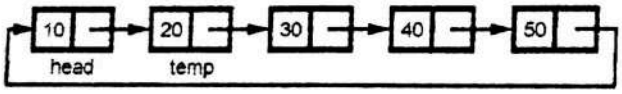


temp → next = temp1;
head = temp1;

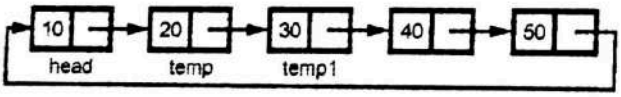
If we want to delete an intermediate node from a linked list which is given below.



We want to delete node with '30' then

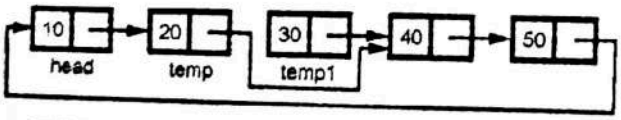


Key = 30
and
temp → next → data
= key. hence

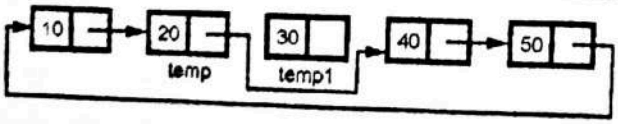


temp1 = temp → next;

Now



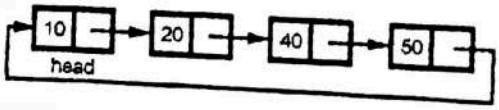
temp → next
= temp1 → next;



temp1 → next = NULL

Then delete temp1 ; / * to deallocate memory */

The linked list can be -



Thus node with value 30 is deleted from CLL.

5. Searching a node from circular linked list

```
void sll::Search(int num)
{
    node *temp;
    int found;
    temp=head;
    if ( temp == NULL )
     {
      cout<<"The Linked List is empty\n";
      getch();
      clrscr();
     }
    found=0;
    while(temp->next!=head && found==0)
    {
     if(temp->data == num)
      found=1;/*if node is found*/
     else
      temp=temp->next;
    }
    if(found==1)
     {
     cout<<"\n The node is present";
     getch();
    }
    else
    {
     cout<<"\n The node is not present";
     getch();
    }
}
```

while searching a node from circular linked list we go on comparing the data field of each node starting from the head node. If the node containing desired data is found we declare that the node is present.

'C++' Program

```
/*
*****************************************************************
 Program To Perform The Various Operations On The Circular Linked List
*****************************************************************
*/

#include<iostream.h>
#include <conio.h>
#include<stdlib.h>
// class defination
```

```
class sll
{
private:
struct node
{
 int data;
 struct node *next;
}*head;
public:
sll();
void Create();
void Display();
void Search(int key);
void insert_head();
void insert_after();
void insert_last();
void Delete();
~sll();
};

/*
-----------------------------------------------------------
The  Constructor defined
-----------------------------------------------------------
*/
sll::sll()
 {
 head=NULL;//initialize head to NULL
 }
/*
-----------------------------------------------------------
The Destructor defined
-----------------------------------------------------------
*/
sll::~sll()
{
  node *temp,*temp1;
  temp=head->next;
  delete head;
  while(temp!=NULL) //free the memory allocated
  {
   temp1=temp->next;
   delete temp;
   temp=temp1;
  }
}
```

```
/*
    ----------------------------------------
    The Create function
    ----------------------------------------
*/
void sll ::Create()
{
char ans;
int flag=1;
node *New,*temp;
clrscr();
do
{
    New = new node;
    New->next=NULL;
    cout<<"\n\n\n\tEnter The Element\n";
    cin>>New->data;
    if(flag==1)   /*flag for setting the starting node*/
    {
        head = New;
        New->next=head;  /*the circular list of a single
                node*/
        flag=0;/*reset flag*/
    }
    else          /* find last node in list */
    {
    temp=head;
    while (temp->next != head)/*finding the last node*/
        temp=temp->next; /*temp is a last node*/
    temp->next=New;
    New->next=head; /*each time making the list
            circular*/
    }
    cout<<"\n Do you want to enter more nodes?(y/n)";
    ans=getch();
    }while(ans=='y'||ans=='Y');
}
void sll::Display()
{
    node *temp;
    temp = head;
    if(temp == NULL)
    cout<<"\n Sorry ,The List Is Empty\n";
    else
    {
    do
    {
```

```
            cout<<'\t'<<temp->data;
            temp = temp->next;
        }while(temp != head);/*Circular linked list*/
        }
        getch();
}

void sll::insert_head()
{
node *New,*temp;
New=new node;
New->next=NULL;
cout<<"\n Enter The element which you want to insert ";
cin>>New->data;
if(head==NULL)
        head=New;
else
{
            temp=head;
            while(temp->next!=head)
                    temp=temp->next;
            temp->next=New;
            New->next=head;
            head=New;
            cout<<"\n The node is inserted!";
}
}
/*Insertion of node at last position*/
void sll::insert_last()
{
        node *New,*temp;
        New=new node;
        New->next=NULL;
        cout<<"\n Enter The element which you want to insert ";
        cin>>New->data;
        if(head==NULL)
                head=New;
        else
        {
        temp=head;
        while(temp->next!=head)
                temp=temp->next;
        temp->next=New;
        New->next=head;
        cout<<"\n The node is inserted!";
        }
}
```

```
void sll::insert_after()
{
int key;
node *New,*temp;
New= new node;
New->next=NULL;
cout<<"\n Enter The element which you want to insert ";
cin>>New->data;
if(head==NULL)
{
  head=New;
}
else
{
    cout<<"\n Enter The element after which you want to insert the node ";
    cin>>key;
    temp=head;
    do
    {
        if(temp->data==key)
        {
            New->next=temp->next;
            temp->next=New;
            cout<<"\n The node is inserted";
            return;
        }
        else
            temp=temp->next;
  }while(temp!=head);
 }
}

void sll::Search(int num)
{
    node *temp;
    int found;
    temp=head;
    if ( temp == NULL )
    {
     cout<<"The Linked List is empty\n";
     getch();
     clrscr();
    }
    found=0;
    while(temp->next!=head && found==0)
    {
    if(temp->data == num)
```

```
      found=1;/*if node is found*/
     else
       temp=temp->next;
    }
    if(found==1)
     {
     cout<<"\n The node is present";
     getch();
     }
     else
     {
      cout<<"\n The node is not present";
      getch();
     }
}
void sll::Delete()
{
 int key;
 struct node *temp,*temp1;
 cout<<"\n Enter the element which is to be deleted";
 cin>>key;
 temp=head;
 if(temp->data==key)/*If header node is to be deleted*/
 {
        temp1=temp->next;
        if(temp1==temp)
   /*if single node is present in circular linked list
   and we want to delete it*/
        {
              temp=NULL;
              head=temp;
              cout<<"\n The node is deleted";
        }
        else  /*otherwise*/
        {
          while(temp->next!=head)
            temp=temp->next;/*searching for the last node*/
              temp->next=temp1;
              head=temp1;/*new head*/
              cout<<"\n The node is deleted";
        }
 }
 else
 {
        while(temp->next!=head) /* if intermediate node is to
                          be deleted*/
        {
```

```
    if((temp->next)->data==key)
    {
        temp1=temp->next;
        temp->next=temp1->next;
        temp1->next=NULL;
        delete temp1;
        cout<<"\n The node is deleted";

    }
    else
        temp=temp->next;

    }

}
}

void main()
{
    sll s;
    char ch='y';
    int num,choice,choice1;
    do
    {
    clrscr();
    cout<<"\n Program For Circular Linked List\n";
    cout<<" 1.Insertion of any node\n\n";
    cout<<" 2. Display of Circular List\n\n";
    cout<<" 3. Insertion of a node in Circular List\n\n";
    cout<<" 4. Deletion of any node \n\n";
    cout<<" 5. Searching a Particular Element in The List\n\n";
    cout<<" 6.Exit ";
    cout<<"\n Enter Your Choice ";
    cin>>choice;
    switch(choice)
    {
        case 1 :s.Create();
            break;
        case 2 :s.Display();
            break;
        case 3: cout<<"\n 1. Insert a node as a head node";
            cout<<"\n 2. Insert a node as a last node";
            cout<<"\n 3. Insert a node at intermediate position in the linked list";
            cout<<"\n Enter your choice for insertion of node";
            cin>>choice1;
            switch(choice1)
            {
              case 1:s.insert_head();
                  break;
              case 2:s.insert_last();
```

```
                    break;
                case 3:s.insert_after();
                    break;
                }
                break;
            case 4:s.Delete();
                break;
            case 5:cout<<"\n Enter The Element Which Is To Be Searched ";
                cin>>num;
                s.Search(num);
                break;
            case 6:exit(0);
        }
        cout<<"\nDo you want to go to Main Menu?\n";
        ch = getch();
    }while(ch == 'y' || ch == 'Y');
}
```

Output

```
Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 1


    Enter The Element
10
Do you want to enter more nodes?(y/n)

    Enter The Element
20
Do you want to enter more nodes?(y/n)

    Enter The Element
30
Do you want to enter more nodes?(y/n)

    Enter The Element
40
```

```
Do you want to enter more nodes?(y/n)
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 2
    10    20    30    40
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 3

1. Insert a node as a head node
2. Insert a node as a last node
3. Insert a node at intermediate position in the linked list
Enter your choice for insertion of node 1

Enter The element which you want to insert 9

The node is inserted!
Do you want to go to Main Menu?
```

```
Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 2
     9     10     20     30     40
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 3

1. Insert a node as a head node
2. Insert a node as a last node
3. Insert a node at intermediate position in the linked list
Enter your choice for insertion of node2.

Enter The element which you want to insert 50

The node is inserted!
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List
```

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 2
9 10 20 30 40 50
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node
5. Searching a Particular Element in The List

6.Exit

Enter Your Choice 5

Enter The Element Which Is To Be Searched 75

The node is not present
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 6

Advantages of Circular Linked List over Singly Linked List

In circular linked list the next pointer of last node points to the head node. Hence we can move from last node to the head node of the list very efficiently. Hence accessing any node is much faster than singly linked list.

Applications of circular list :

1) The circular list is used to create the linked lists in which the last node points to the first node.
2) In round robin method the circular linked list is used.
3) The circular linked list is used in solving Josephus problem.
4) For representing the polynomial the circular linked list.

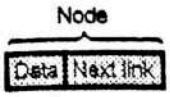
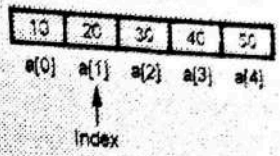
Review Questions

1. *Explain about circular linked list.*
2. *Write a program to implement circular linked list.*
3. *Write a function for insertion operation in circular linked list.*
4. *Explain the operations of circular linked list.*
5. *Write a 'C' function to insert and delete a Nth element in circular linked list ? Give the pictorial representation of the same.* SPPU : May-06, 09, Marks
6. *Discuss insertion and deletion of an element in the singly, doubly and circular linked list. (Give supporting C code and pictorial representation).* SPPU : Dec.-07, May-08, Marks
7. *Explain a insertion of node in circular linked list : 1) The start of the list 2) The end of the list 3) After the position.* SPPU : May-09, Marks
8. *Write a Pseudo C code to add a node in a singly circular linked list with integer data.* SPPU : May-10, Marks
9. *Write a pseudo C function to display the singly circular link list storing character data in reverse manner and comment on time and space complexity of your function.* SPPU : May-13, Marks
10. *Write advantages circular singly linked list over a linear linked list.* SPPU : May-14, Marks
11. *List the applications of circular lists.* SPPU : May-16, Marks

1.15 Linked List Vs. Arrays

SPPU : Dec.-06,09, May-10, May-12,13, Marks 8

The comparison between linked list and arrays is as shown below -

Sr. No.	Linked list	Array
1.	The linked list is a collection of nodes and each node is having one data field and next link field. For example : 	The array is a collection of similar types of data elements. In arrays the data is always stored at some index of the array. For example : 
2.	Any element can be accessed by sequential access only.	Any element can be accessed randomly i.e. with the help of index of the array.
3.	Physically the data can be deleted.	Only logical deletion of the data is possible.
4.	Insertions and deletion of data is easy	Insertions and deletion of data is difficult.
5.	Memory allocation is dynamic. Hence developer can allocate as well as deallocate the memory. And so no wastage of memory is there.	The memory allocation is static. Hence once the fixed amount of size is declared then that much memory is allocated. Therefore there is a chance of either memory wastage or memory shortage.

Example 1.15.1 *Compare linked list with arrays with reference to the following aspects :*

i) Accessing any element randomly

ii) Insertion and deletion of an element

iii) Utilization of computer memory.

SPPU : Dec.-06,09, May-12, Marks 8

Solution : i) Accessing any element randomly : Using array, any element can be accessed easily.

a) It takes constant time.

b) In linked list, we can access elements in sequence. So it is very difficult and time consuming to access a element randomly.

ii) Insertion and deletion of an element :

a) Insertion and deletion of elements is time consuming.

b) In linked list, we can insert and delete elements easily and it is less time consuming.

iii) Utilization of computer memory :

a) Array requires contiguous memory. First we have to declare array and compiler allocates memory at declaration.

b) If we utilize less memory than allocated, then unutilized memory is unnecessarily reserved for array. It can't be used by other programs.

c) If we want to add more elements than the actual size of array, it is not possible.

d) In linked list, memory is utilized efficiently. Memory is not pre-allocated like static data structure. Memory is allocated as per the need. Memory is deallocated when it is no longer needed.

Review Questions

1. *Compare the linked and sequential organization of data structures.*

SPPU : Dec.-05,09, Marks 4

2. *Compare array with linked list.* SPPU : Dec.-10, Marks 4, May-13, Marks 6

1.16 Applications of Linked List

Various applications of linked list are -

1. Representation of polynomial and performig various operations such as addition, multiplication and evaluation on it.
2. Performing addition of long positive integers.
3. Representing non-integer and non-homogeneous list.

2 Searching and Sorting

Syllabus

Searching and sorting : *Need of searching and sorting, Concept of internal and external sorting, sort stability, Searching methods : Linear and binary search algorithms, Fibonacci Series.*
Sorting methods : *Bubble, insertion, Quick, Merge, shell and comparison of all sorting methods. Analyze Insertion sort, Quick Sort, binary search, hashing for Best, Worst and Average case.*

Contents

2.1 Need of Searching and Sorting. . . . . **Dec.-06,08,09,15,** · · · · · · · · · · Marks 6

2.2 Concept of Internal and External Sorting **May-07,08,09,**
. . . . . **Dec.-08,14,** · · · · · · · · · · · · · · · · Marks 8

2.3 Sort Stability . . . . . **May-09,16, Dec.-14,** · · · · · · · · Marks 6

2.4 Searching Methods **Dec.-13,16,**
. . . . . **May-14,16,** · · · · · · · · · · · · · · · · · Marks 6

2.5 Linear Search

2.6 Binary Search . . . . . **Dec.-13,16, May-14,16,** · · · · · · Marks 7

2.7 Fibonacci Search

2.8 Bubble Sort . . . . . **Dec.-11,** · · · · · · · · · · · · · · · · · · · Marks 8

2.9 Insertion Sort **Dec.-06,07,09,11,**
. . . . . **May-07,12,** · · · · · · · · · · · · · · · · · Marks 8

2.10 Quick Sort . . . . . **May-09,** · · · · · · · · · · · · · · · · · · · Marks 6

2.11 Merge Sort

2.12 Shell Sort

2.13 Comparison of all Sorting Methods

2.14 Analyze Insertion sort, Quick Sort, Binary Search, Hashing

2.15 Study and Analyze Selection Sort, Bucket Sort, Radix Sort
. . . . . **May-08,10,11,15,16,**
. . . . . **Dec.-08,11,13,16,** · · · · · · · · · · Marks 8

Part I : Basics

2.1 Need of Searching and Sorting

The sorting is a technique by which the elements are arranged in some particular order. Usually the sorting order is of two types -

Ascending order : It is the sorting order in which the elements are arranged from low value to high value. In other words elements are in increasing order.

For example : 10, 50, 40, 20, 30

can be arranged in ascending order after applying some sorting technique as

10, 20, 30, 40, 50

Descending Order : It is the sorting order in which the elements are arranged from high value to low value. In other words elements are in decreasing order. It is reverse of the ascending order.

For example : 10, 50, 40, 20, 30

can be arranged in descending order after applying some sorting technique as

50, 40, 30, 20, 10

While sorting the elements, we always consider a specific order and expect our data to be arranged in that order.

Searching is a technique in which the location of desired element is obtained. The searching is based on the search key. This key is compared with the array elements. If the key and element matches then the position of that element is returned.

Importance of Searching and Sorting

- Sorting is useful for arranging the data in desired order. After sorting the required element can be located easily.
- Searching technique is essential for locating the position of the required element from the heap of data.

Application of Sorting

1. The sorting is useful in database applications for arranging the data in desired order.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements, the sorting is required.
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

Review Questions

1. *Discuss the importance of sorting and searching in computer application*

2. *Write short note on "application of sorting".*

2.2 Concept of Internal and External Sorting

- **Internal Sorting :** The **internal sorting** is a sorting in which the data resides in the main memory of the computer.
- Various methods that make use of internal sorting are -

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
5. Merge sort
6. Bucket Sort.

- **External Sorting :** For many applications it is not possible to store the entire data on the main memory for two reasons, amount of main memory available is **smaller** than amount of data. Secondly the **main memory is a volatile** device and thus will loose the data when the power is shut down. To overcome these problems the data is stored on the **secondary storage devices.** The technique which is used to sort the data which resides on the secondary storage devices are called **external sorting.**

The data stored on secondary memory is part by part loaded into main memory, sorting can be done over there. The sorted data can be then stored in the intermediate files. Finally these intermediate files can be merged repeatedly to get sorted data. Thus **huge amount** of data can be sorted using this technique.

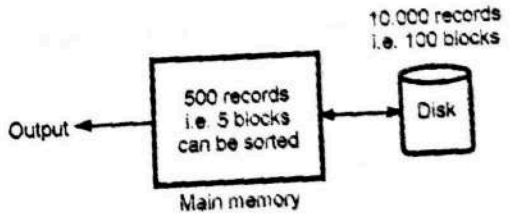


Fig. 2.2.1

For example :

- Consider that there are 10,000 records that has to be sorted. Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records.

- The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will be repeated 20 times to get all the records sorted in chunks.
- In the second step, we start merging a pair of intermediate files in the main memory to get output file. (See Fig. 2.2.2).

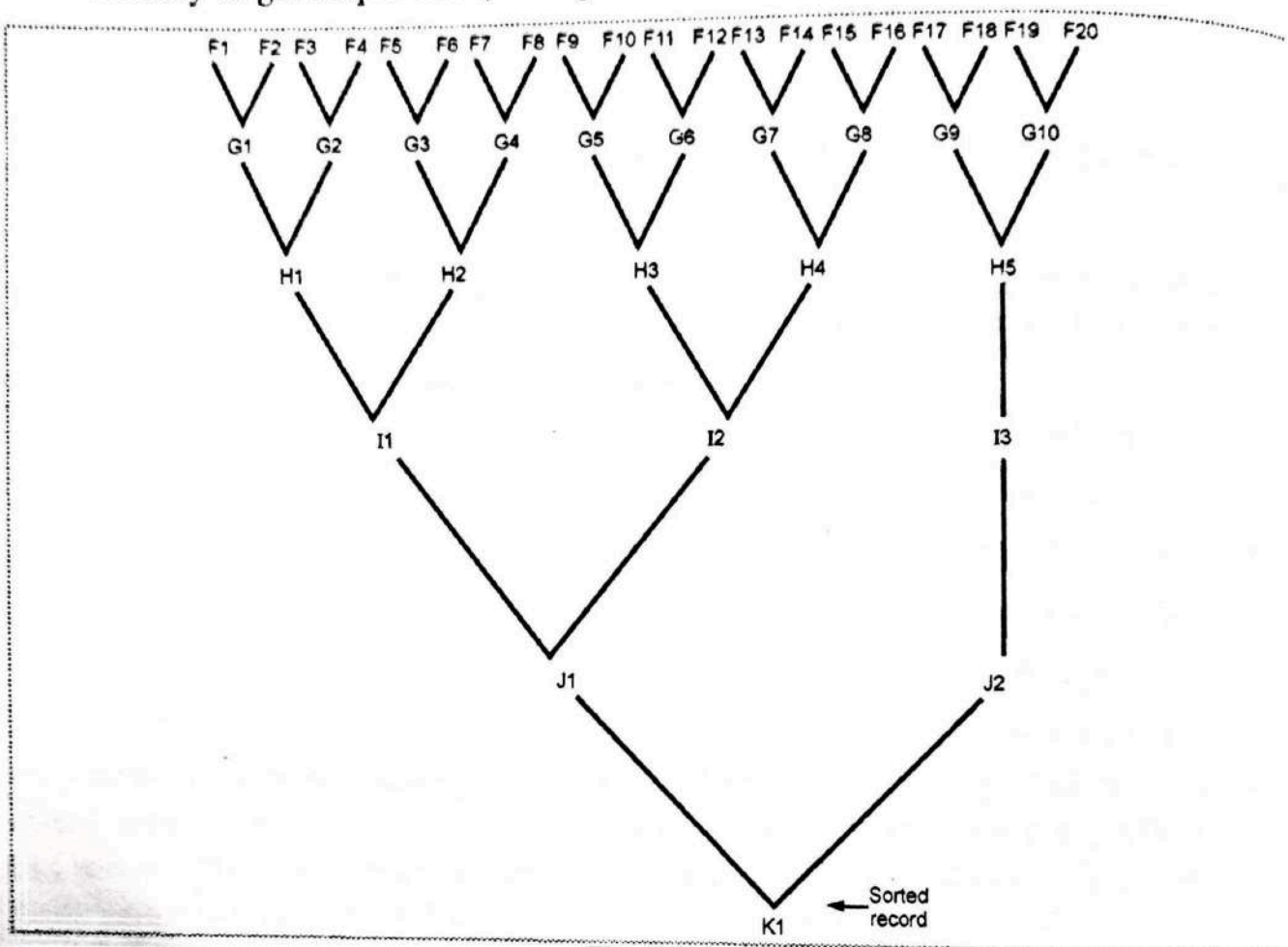


Fig. 2.2.2

Difference between Internal and External Sorting

Sr.No.	Internal Sorting	External Sorting
1.	In internal sorting method, the amount of data is limited.	In external sorting method, there is huge amount of data that need to be sorted.
2.	All the data to sort is stored on main memory.	In external sorting data is stored outside memory (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.
3.	It takes less amount of time.	It takes more amount of time as each time data is loaded into main memory from secondary memory.
4.	It is simple technique.	It is complicated.
5.	Example : Bubble sort, quick sort, insertion sort.	Example : Two way merge sort.

Review Questions

1. *Compare external and internal sorting. Give the suitable example of external and internal sorting each.* [illegible]
2. *Explain the terms : i) Internal sorting ii) External sorting.* [illegible]

2.5 Sort Stability

[illegible]

i) Sorting Stability

The sorting stability means comparing the records of same value and expecting them in the same order even after sorting them.

For example :

(Pune, BalGandharva)

(Pune, Shaniwarwada)

(Nasik, Panchavati)

(Mumbai, Gateway-of-India)

Now in the above list, we will sort the list according to the first alphabet of the city. The ascending order for the alphabets P(for Pune), N(for Nasik), M(for Mumbai) will be M, N, P. The sorting stability can be achieved by arranging the records as follows.

(Mumbai, Gateway-Of-India)

(Nasik, Panchavati)

(Pune, BalGandharva)

(Pune, Shaniwarwada)

Note that in the above list same record as Pune is twice. But we have preserved the original sequence as it is after comparing them. Thus the stability is achieved in sorting the records.

ii) Efficiency and Passes

We usually denote the efficiency of sorting algorithms in terms of time complexity. The time complexities are given in terms of big-on notations.

Commonly there are $O(n^2)$ and O(nlogn) time complexities for various algorithms. The sorting techniques such as bubble sort, insertion sort, selection sort, shell sort has the time complexity $O(n^2)$.

While sorting the elements in some specific order there is lot of arrangement of elements. The phases in which the elements are moving to acquire their proper position are called **passes.**

For example : 10, 30, 20, 50, 40

Pass 1 : 10, 20, 30, 50, 40

Pass 2 : 10, 20, 30, 40, 50

In the above method we can see that data is getting sorted in two passes distinctly.

Review Questions

1. *Explain the following terms with respect to sorting*
   i) Sort stability ii) Efficiency iii) Passes SPPU : May 09, May 13
2. *Explain the term : Sort stability.* SPPU : Dec 14, May 15
3. *What is sort stability and efficiency ?* SPPU : May 16, May 17

Part II : Searching

2.4 Searching Methods

SPPU : Dec. 13,15, May 14,16, May 17

- When we want to find out particular record efficiently from the given list of elements then there are various methods of searching that element. These methods are called **searching methods.** Various algorithms based on these searching methods are known as **searching algorithms.**
- The **basic characteristic** of any searching algorithm is -
  i. It should be **efficient**
  ii. **Less number of computations** must be involved in it.
  iii. The **space** occupied by searching algorithms must be **less.**
- The most commonly used searching algorithms are -
  i. Sequential or linear search
  ii. Indexed sequential search
  iii. Binary search
- **The element** to be searched from the given list is called the **key** element.

Let us discuss various searching algorithms.

2.5 Linear Search

- **Definition :** Linear searching is a searching technique in which every element is compared with key linearly from first to last element of list.
- The time complexity of this algorithm is **O(n).** The time complexity will increase linearly with thes value of n.
- For higher value of n the linear search is not the satisfactory solution.

- **Example :**

Array

	Roll no.	Name	Marks
0	15	Parth	96
1	2	Anand	40
2	13	Lalita	81
3	1	Madhav	50
4	12	Arun	78
5	3	Jaya	94

Fig. 2.5.1 Represents students database for linear search

From the above Fig. 2.5.1 the array is maintained to store the students record. The record is not sorted at all. If we want to search the student's record whose roll number is 12 then with the key-roll number we will see the every record whether it is of roll number = 12. We can obtain such a record at array [4] location.

Let us implement the linear search using C++ program

C++ Program

```
#include<iostream>
using namespace std;
class Search
{
 private:
            int a[10],n,key;
 public:
            void get_data();
            void Seq_Search(int);
};
void Search::Seq_Search(int key)
{
 int flag=0,mark;
 for(int i=0;i<n;i++)
 { //searching element from beginning of array
  if(a[i]==key)
  {
   flag=1;//setting the flag if element found
   mark=i;//marking the location of key element
   break;
  }
 }
 if(flag==1)//flag=1 means element is found
 cout<<"\n The element is present at location: "<<mark+1;
```

```
else
 cout<<"\n The element is not present in the array";
}
void Search::get_data()
{
 cout<<"\n How Many Elements are there in an array? ";
 cin>>n;
 cout<<"\n Enter the elements ";
 for(int i=0;i<n;i++)
  cin>>a[i];
}

void main( )
{
 int k;
 Search obj;
obj.get_data();
 cout<<"\n\t Enter the element which is to be searched ";
 cin>>k;
 obj.Seq_Search(k);
}
```

Output

```
How Many Elements are there in an array? 7

Enter the elements 10 20 30 40 50 60 70

      Enter the element which is to be searched 50
The element is present at location: 5
```

Advantages of linear searching

1. It is simple to implement.
2. It does not require specific ordering before applying the method.

Disadvantage of linear searching

1. It is less efficient.

2.6 Binary Search

SPPU : Dec. 13, 16, May 14, 16, May 17

Definition : Binary search is a searching algorithm in which the list of elements is divided into two sublists and the key element is compared with the middle element. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves (sublists) depending upon the result produced through the match.

This algorithm is considered as an efficient searching algorithm.

The prerequisite for this searching technique is that the **list should be sorted.**

Example 2.5.1 *Search 99 from the list –40, 11, 33, 37, 42, 45, 99, 100 using binary search.*

Solution : As mentioned earlier the necessity of this method is that all the elements should be sorted. So let us take an array of sorted elements.

Array

0	1	2	3	4	5	6	7
– 40	11	33	37	42	45	99	100

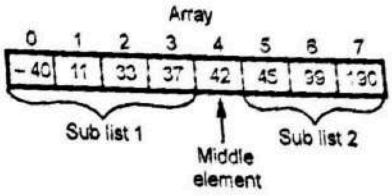
Step 1 : Now the key element which is to be searched is = 99 ∴ key = 99.

Step 2 : Find the middle element of the array. Compare it with the key

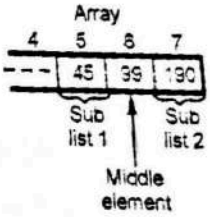
if middle $\overset{=}{?}$ key

i.e. if 42 $\overset{=}{?}$ 99

if 42 < 99 search the sublist 2



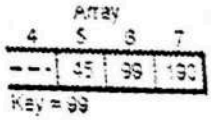
Now handle only sublist 2. Again divide it, find mid of sublist 2



if middle $\overset{=}{?}$ key

i.e. if 99 $\overset{=}{?}$ 99

So match is found at 7th position of array i.e. at array [6].



Thus by binary search method we can find the element 99 present in the given list at array $[6]^{th}$ location.

Example 2.5.2 *How many comparisons are required to find 73 in the following array usin binary search 12, 25, 32, 37, 41, 48, 58, 60, 66, 73, 74, 79, 83, 91, 95 ?*

Solution :

Step 1 :

0	1	2	3	4	5	6	7	8	9	10	11
12	25	32	37	41	48	58	60	66	73	74	79

↑

Compare 73 with 58

Number of comparison = 1.

Step 2 : As 73 > 58. Search 73 between A[7] to A[11]

7	8	9	10	11
60	66	73	74	79

↑

Compare 73 with 73

Number 73 is found.

∴ **Number of comparison = 2.**

Thus with two comparisons number 73 is found.

Example 3.1.3 *Apply binary search method to search 60 from the list 10, 20, 30, 40, 50, 60, 70.*

Solution : Consider a list of elements stored in array A as,

0	1	2	3	4	5	6
10	20	30	40	50	60	70
↑ Low						↑ High

The KEY element (i.e. the element to be searched) is 60.

Now to obtain middle element we will apply a formula :

$$m = (low + high)/2$$

$$m = (0 + 6)/2$$

$$m = 3$$

Then Check $A[m] \stackrel{?}{=} KEY$

i.e. $A[3] \stackrel{?}{=} 60$ NO $A[3] = 40$ and $40 < 60$

∴ **Search the right sublist.**

0	1	2	3	4	5	6
10	20	30	40	50	60	70

Left sublist ↑ m Right sublist

The right sublist is

...	50	60	70

Now we will again divide this list and check the mid element.

	4	5	6
...	50	60	70
		↑ m	

$m = (low + high)/2$

$m = (4 + 6)/2$

$\therefore m = 5$

is $A[m] \stackrel{?}{=} KEY$

i.e. $A[5] \stackrel{?}{=} 60$ Yes, i.e. the number is present in the list.

Thus we can search the desired number from the list of elements.

Algorithm for binary search using recursive definition :

1. if(low > high)
2. return;
3. mid = (low + high)/2;
4. if(x==a[mid])
5. return (mid);
6. if(x < a[mid])
7. search for x in a[low] to a[mid-1];
8. else
9. search for x in a[mid + 1] to a[high];

```
#include<iostream>
#define size 10
class BINSEARCH
{
 private:
          int a[size];

 public:
          int low,high;
          BINSEARCH();
          int get_value();
          int binsearch(int x,int low,int high);
};
BINSEARCH::BINSEARCH()
{
 low = 0;
}
```

```
int BINSEARCH::get_value()
{
 int n;
 cout<<"\n Enter the total number of elements ";
 cin>>n;
 cout<<"\nEnter the list of elements \n";
 for(int i=0;i<n;i++)
            cin>>a[i];
 return n;
}
int BINSEARCH::binsearch(int x,int low,int high)
{
 int mid;
 if(low>high)
            return -1;
 mid = (low+high)/2;
 if(x ==a[mid])
            return(mid);
 else if(x<a[mid])
            binsearch(x,low,mid-1);
 else
            binsearch(x,mid+1,high);
}

void main(void)
{
 int key,ans,n;
 BINSEARCH obj;
cout<<"\n\t\t\t Binary Search Method ";
 n=obj.get_value();
 cout<<"\n Enter the element which you want to search ";
 cin>>key;
 obj.high =n-1;
 ans=obj.binsearch(key,obj.low,obj.high);
 if(ans!= -1)
            cout<<"\n The number "<<key<<" is present in the list at location
"<<ans+1;
 else
            cout<<"\n The number is not present in the list ";
}
```

Output 1

```
Binary Search Method
Enter the total number of elements 5
Enter the list of elements
10
20
30
```

```
40
50

Enter the element which you want to search 40

The number 40 is present in the list at location 4
```

Output 2

```
Binary Search Method
Enter the total number of elements 5
Enter the list of elements
10
20
30
40
50
Enter the element which you want to search 78

The number is not present in the list
```

Searching method	Time complexity
Linear Search	O(n)
Binary Search	O(n log n)

Advantage of binary searching

1. It is an efficient technique.

Disadvantages of binary searching

1. It requires specific ordering before applying the method.
2. It is complex to implement.

Difference between Linear Search and Binary Search

Sr. No.	Linear search method	Binary search method
1.	The linear search is a searching method in which the element is searched by scanning the entire list from first element to the last.	The binary search is a searching method in which the list is sub divided into two sub-lists. The middle element is then compared with the key element and then accordingly either left or right sub-list is searched.
2.	Many times entire list is searched.	Only sub-list is searched for searching the key element.

3.	It is simple to implement.	It involves computation for finding the middle element.
4.	It is less efficient searching method.	It is an efficient searching method.

Review Questions

1. *Write pseudo C code for binary search and analyze its time complexity.* **SPPU : May 16, Marks 3**
2. *Compare linear and binary search.* **SPPU : May 14, Marks 3**
3. *Explain linear and binary search techniques with examples.* **SPPU : May 16, Marks 3**
4. *Write Pseudo C algorithms for : i) Linear Search ii) Binary Search.* **SPPU : December 16, Marks 7**

2.7 Fibonacci Search

In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. If the key element is really present in the list, we can reach to the location of key in the list and thus we get the message that the "element is present in the list" otherwise get the message. "element is not present in the list."

In Fibonacci search rather than considering the mid element, we consider the indices as the numbers from fibonacci series. As we know, the Fibonacci series is -

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

To understand how Fibonacci search works, we will consider one example, suppose, following is the list of elements.

arr [ ]

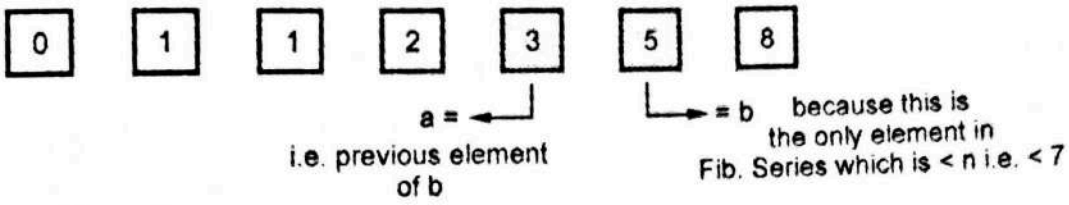
10	20	30	40	50	60	70
1	2	3	4	5	6	7

Here n = total number of elements = 7

We will always compute 3 variables i.e. a, b and f.

Initially f = n = 7.

For setting a and b variables we will consider elements from Fibonacci series.



Now we have

f = 7

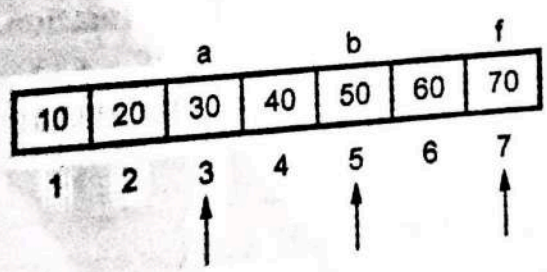
b = 5

a = 3

With these initial values we will start searching the key element from the list. Each time we will compare key element with arr [f]. That means

If (Key < arr [f])	If (Key > arr [f])
f = f – a	f = f + a
b = a	b = b – a
a = b – a	a = a – b

Suppose we have f = 7, b = 5, a = 3



If Key = 20

i.e. Key < arr[f]

i.e. 20 < 70

$\therefore f = f - a = 7 - 3 = 4$

b = a = 3

a = b − a = 2

Again we compare $\boxed{\text{if (Key < arr [f])}}$

i.e. 20 < arr [4]

i.e. if (20 < 40) → Yes

At Present f = 4, b = 3, a = 2

$\therefore f = f - a = 4 - 2 = 2$

b = a = 2

a = b − a = 3 − 2 = 1

Now we get f = 2, b = 2, a = 1

a	f/b					
10	20	30	40	50	60	70
1	2	3	4	5	6	7

If (Key < arr [f]) i.e. if (20 < 20) → No

If Key > arr [f]) i.e. if (20 > 20) → No

That means "Element is present at f = 2 location"

If Key = 60

i.e. Key < arr [f]

60 < 70

$\therefore f = f - a = 7 - 3 = 4$

b = a = 3

a = b − a = 2

Again we compare $\boxed{\text{if (Key > arr [f])}}$

∴i.e. 60 > arr [f] i.e. 40

$\therefore f = f + a = 4 + 2 = 6$

b = b − a = 3 − 2 = 1

a = a − b = 2 − 3 = − 1

a	b				f	
10	20	30	40	50	60	70
1	2	3	4	5	6	7

If (Key < arr [f])

i.e. if (60 < 60) → No

If (key > arr [f])

i.e. if (60 > 60) → No

That means "Element is present at f = 6 location."

Analysis : The time complexity of fibonacci search is O(logn).

Algorithm

Let the length of given array be n [0...n-1] and the element to be searched be key

Then we use the following steps to find the element with minimum steps :

1. Find the smallest Fibonacci number greater than or equal to n. Let this number be f (m^{th} element).
2. Let the two Fibonacci numbers preceding it be a ($(m-1)^{th}$ element) and b ($(m-2)^{th}$ element)

While the array has elements to be checked :

Compare key with the last element of the range covered by **b**

(a) If **key** matches, return index value.

(b) Else if **key is less** than the element, move the third Fibonacci variable two Fibonacci down, indicating removal of approximately two-third of the unsearched array.

(c) Else key is greater than the element, move the third Fibonacci variable one Fibonacci down. Reset offset to index. Together this results into removal of approximately front one-third of the unsearched array.

3. Since there might be a single element remaining for comparison, check if **a** is '1'. If Yes, compare key with that remaining element. If match, return index value.

For example –

According to the algorithm, we have to sort the elements of the array prior to applying Fibonacci search. Consider sorted array of elements as -

0	1	2	3	4	5	6
10	20	30	40	50	60	70

$\therefore n = 7$, we want to find key = 60

Now check Fibonacci series.

As $n < 8$

Set $f = 8$

$a = 5$

$b = 3$

			b		a			f
0	1	2	3	4	5	6		8
10	20	30	40	50	60	70	...	

Set Offset $= -1$

$\therefore \quad i = 2$

$\therefore \quad i = \min(\text{offset} + b, n - 1)$

$A[i] = A[2] < (\text{key} = 60)$

Here key is greater than the element we move f one Fibonacci down (step 2c of algorithm)

$\therefore$ $f = 5$

$a = 3$

$b = 2$

		b	a		f	
0	1	2	3	4	5	6
10	20	30	40	50	60	70

Set offset = i i.e. = 2

Now $i = 4$

$\therefore$ $i = \min(\text{offset} + b, n - 1)$

Here key is again greater than the element. So we move f, one fibbonacci down

$\therefore$ $f = 3$

$a = 2$

$b = 1$

	b	a	f			
0	1	2	3	4	5	6
10	20	30	40	50	60	70

Set offset = i i.e. **4**

Now new $i = 5$

$\therefore$ $i = \min(\text{offset} + b, n - 1)$

Here $a[i] = \text{key}$

Hence return value of i as the position of key element.

Note that, due to Fibonacci numbering, the search portion is restricted and we need to compare very less number of elements.

C++ Program

```
#include<iostream>
#include<cmath>
using namespace std;
int FibSearch(int arr[10], int key,int  n)
{
    int i;
    // Initialize Fibonacci numbers
    int b = 0;
    int a = 1;
    int f = b + a;

    // f is going to store the smallest
    // Fibonacci number greater than or equal to n
    while (f < n)
    {
        b = a;
        a = f;
        f = b + a;
    }
    // Marks the eliminated range from front
    int offset = -1;

    // while there are elements to be inspected.
    // we compare arr[i] with key.
    while (f > 1)
    {
    // Check if b is a valid location
    i = std::min(offset+b, n-1);

      // If key is greater than the value at
      // index b, cut the subarray array
    // from offset to i
    if (arr[i] < key)
    {
            f = a;
            a = b;
            b = f - a;
            offset = i;

    }
    // If key is lesser than the value at
    // index b, cut the subarray
    // after i+1
    else if (arr[i] > key)
    {
            f = b;
            a = a - b;
```

```
                b = f - a;
        }
        // element found. return index
        else
                return i;
}
    // comparing the last element with key
    if(a ==key && arr[offset+1] == key)
        return offset+1;

    // element not found. return -1
    return -1;
}
int main()
{
        int i,n,array[10],key;
        cout<<"\n Program For Fibonacci Search";
        cout<<"\nHow many elements are there in Array?";
        cin>>n;
        for (i=0;i<n;i++)
        {
                cout<<"\n Enter element in Array";
                cin>>array[i];
        }
        cout<<"Resultant array is\n";
        for (i=0;i<n;i++)
        {
                cout<<" "<<array[i];
        }
        cout<<"\n Enter the key element to be searched: ";
        cin>>key;
        cout<<"\n The element is present at index: "<<FibSearch(array,key,n);
        return 0;
}
```

Output

```
Program for Fibonacci Search
How many elements are there in Array? 7
 Enter element in Array 10
 Enter element in Array 20
 Enter element in Array 30
 Enter element in Array 40
 Enter element in Array 50
 Enter element in Array 60
 Enter element in Array 70
Resultant array is
 10 20 30 40 50 60 70
Enter the key element to be searched : 60
The element is present at index : 5
```

Part - III : Sorting Methods

2.8 Bubble Sort

This is the simplest kind of sorting method in this method. We do this bubble sort procedure in several iterations, which are called passes.

Example Bubble sort :

Consider 5 unsorted elements are

45 – 40 190 99 11

First store those elements in the array a.

	a
0	45
1	–40
2	190
3	99
4	11

Pass 1

In this pass each element will be compared with its neighbouring element.

Compare 45 and – 40

Is 45 > – 40 ∴Interchange

i.e. compare a[0] and a[1], after interchange

∴a[0] = – 40

	a
0	45
1	–40
2	190
3	99
4	11

Compare a[1] and a[2]

Is 45 > 190 ∴No interchange

	a
0	–40
1	45
2	190
3	99
4	11

Compare a[2] and a[3]

Is 190 > 99 ∴Interchange

a[2] = 99

a[3] = 190

	a
0	–40
1	45
2	190
3	99
4	11

Compare a[3] and a[4]

Is 190 > 11 ∴Interchange

a[3] = 11

a[4] = 190

	a
0	–40
1	45
2	99
3	190
4	11

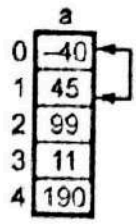
After first pass the array will hold the elements which are sorted to some extent.

	a
0	–40
1	45
2	99
3	11
4	190

Pass 2

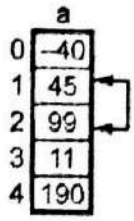
Compare a[0] and a[1]

No interchange



Compare a[0] and a[1]

No interchange.

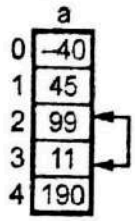


Compare a[2] and a[3]

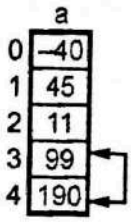
Since 99 > 11 interchange

∴a[2] = 11

a[3] = 99



No interchange



Pass 3

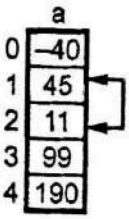
First compare a[0] and a[1], no interchange

∴ Compare a[1] and a[2]

45 > 11 ∴ Interchange

a[1] = 11

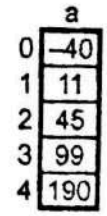
a[2] = 45



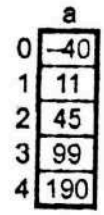
Next compare a[2] and a[3] similarly

a[3] and a[4]

No interchange.



This is end of pass 3. This process will be thus continued till pass 4. Finally at the end of last pass the array will hold all the sorted elements like this,



Since the comparison positions look like bubbles, therefore it is called **bubble sort.**

Algorithm :

1. Read the total number of elements say n.
2. Store the elements in the array.
3. Set the i = 0 .
4. Compare the adjacent elements.
5. Repeat step 4 for all n elements.
6. Increment the value of i by 1 and repeat step 4, 5 for i< n
7. Print the sorted list of elements.
8. Stop.

C++ Program

```
#include<iostream>
using namespace std;
class Sort
{
 private:
        int n;
        int A[10];
 public:
        void get_data();
        void bubble();
};
void Sort::get_data()
{
 cout<<"\n How many elements are there?";
 cin>>n;
 cout<<"\n Enter the elements\n";
 for(int i=0;i<n;i++)
   cin>>A[i];
}
void Sort::bubble()
{
 int  temp;
 for(int i=0;i<=n-2;i++)
 {
        for(int j=0;j<=n-2-i;j++)
        {
        if(A[j]>A[j+1])
        {
        temp=A[j];
```

```
        A[j]=A[j+1];
        A[j+1]=temp;
       }
 }
}
 cout<<"\n The sorted List is ...\n";
 for(i=0;i<n;i++)
       cout<<"        "<<A[i];
}
int main()
{
 Sort obj;
 cout<<"\n\t\t Bubble Sort\n";
 obj.get_data();
 obj.bubble();
 return 0;
}
```

Output

```
          Bubble Sort
How many elements are there?5
Enter the elements
20
10
40
30
50

The sorted List is ...
     10      20      30      40      50
```

Analysis

In Bubble Sort, n-1 comparisons will be done in the 1st pass, $n-2$ in 2nd pass, $n-3$ in 3rd pass and so on. So the total number of comparisons will be,

$(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$

Sum $= n(n-1)/2$

i.e $O(n^2)$

Hence the time complexity of Bubble Sort is $O(n^2)$.

1) Worst Case Time Complexity [Big - O] : $O(n^2)$

2) Best Case Time Complexity [Big-omega] : $O(n)$

3) Average Time Complexity [Big-theta] : $O(n^2)$

2.9 Insertion Sort

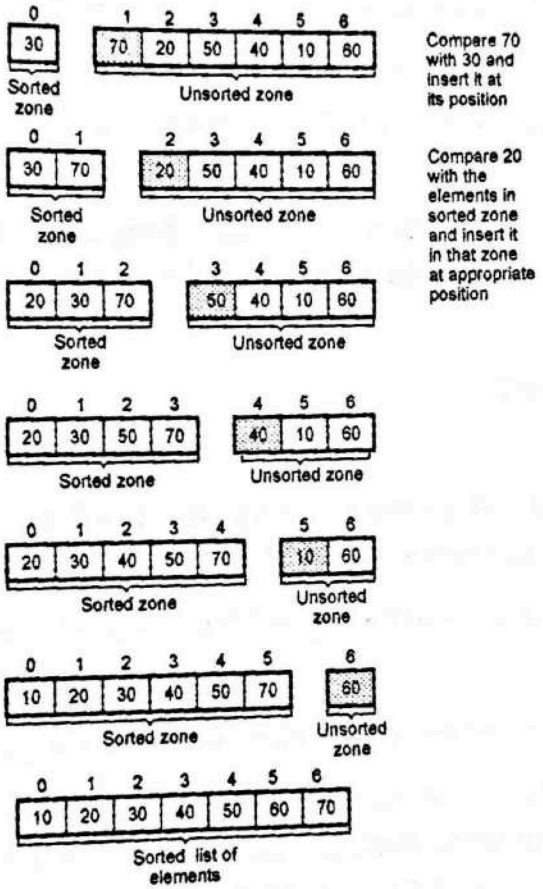
In this method the elements are inserted at their appropriate place. Hence is the name **insertion sort**. Let us understand this method with the help of some example -

For Example

Consider a list of elements as,

0	1	2	3	4	5	6
30	70	20	50	40	10	60

The process starts with first element



Algorithm

Although it is very natural to implement insertion using recursive(top down) algorithm but it is very efficient to implement it using bottom up(iterative) approach.

```
Algorithm Insert_sort(A[0...n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0...n-1] in ascending order
 for i ← 1 to n-1 do
{
        temp ← A[i]//mark A[i]th element
```

```
        j ← i-1//set j at previous element of A[i]
        while(j>=0)AND(A[j]>temp)do
        {
    //comparing all the previous elements of A[i] with
    //A[i].If any greater element is found then insert
    //it at proper position
        A[j+1] ← A[j]
        j ← j-1
        }
        A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

- When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**
- If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.
- If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

Advantages of insertion sort

1. *Simple* to implement.
2. This method is *efficient* when we want to sort small number of elements. And this method has excellent performance on almost sorted list of elements.
3. More efficient than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort.
4. This is a *stable* (does not change the relative order of equal elements).
5. It is called *in-place* sorting algorithm (only requires a constant amount O(1) of extra memory space). The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

C Function

```
void Insert_sort(int A[10],int n)
{
int i,j,temp;
for(i=1;i<=n-1;i++)
{
        temp=A[i];
        j=i-1;
        while((j>=0)&&(A[j]>temp))
```

```
        {
         A[j+1]=A[j];
         j=j-1;
         }
         A[j+1]=temp;
}
printf("\n The sorted list of elements is...\n");
for(i=0;i<n;i++)
         printf("\n%d",A[i]);
}
```

Let us now see the implementation of this method using C.

C Program

```
/*************************************************************
Implementation of insertion sort
*************************************************************/
#include<iostream>
using namespace std;
class Sort
{
 private:
      int A[10];
      int n;
 public:
      void get_data();
      void Insert_sort();
      void display();
};
void Sort::Insert_sort()
{
 int i,j,temp;
 for(i=1;i<=n-1;i++)
 {
          temp=A[i];
          j=i-1;
          while( (j>=0)&&(A[j]>temp))
          {
               A[j+1]=A[j];
               j=j-1;
          }
          A[j+1]=temp;
     }
}
void Sort::get_data()
{
```

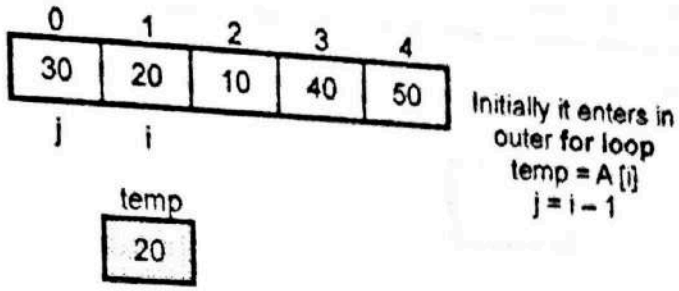
```
 cout<<"\n How many elements are there?";
 cin>>n;
 cout<<"\n Enter the elements\n";
 for(int i=0;i<n;i++)
       cin>>A[i];
}
void Sort::display()
{
       cout<<"\n The sorted list of elements is...\n";
 for(int i=0;i<n;i++)
              cout<<"\n    "<<A[i];
}
void main( )
{
       Sort obj;
cout<<"\n\t\t Insertion Sort";
 obj.get_data();
 obj.Insert_sort();
 obj.display();
}
```

Output

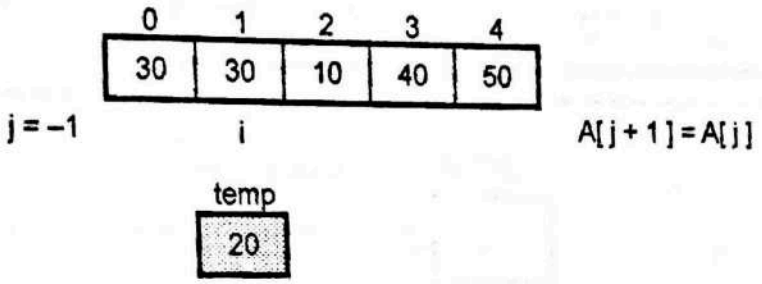
```
                    Insertion Sort
How many elements are there?7
Enter the elements
3
1
2
5
4
7
6
The sorted list of elements is...
  1
  2
  3
  4
  5
  6
  7
```

Logic Explanation :

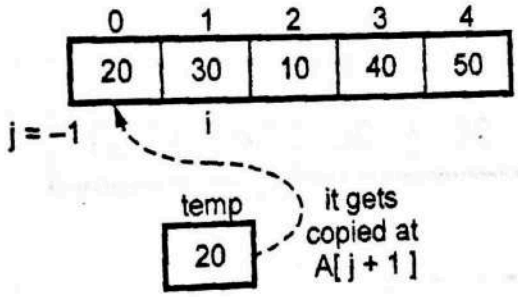
For understanding the logic of above C program consider a list of unsorted elements as,



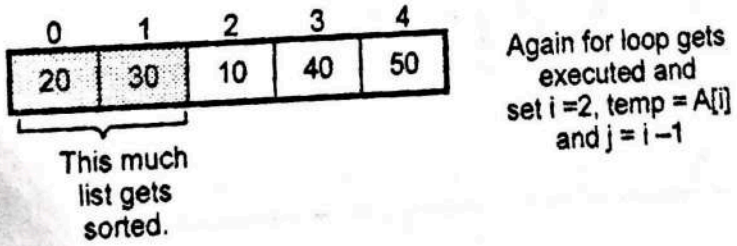
Then the control moves to **while loop**. As j >= 0 and A[j] > temp is **True**, the while loop will be executed.

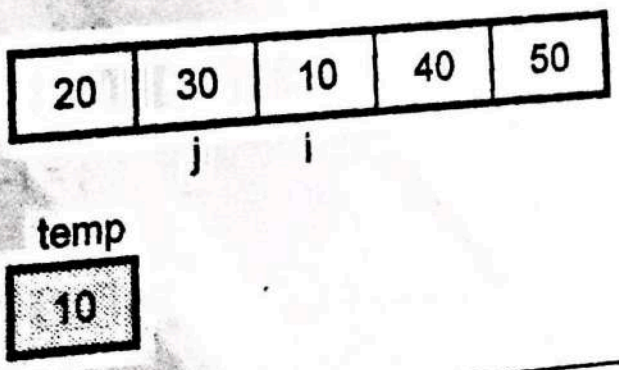


Now since j >= 0 is false, control comes out of while loop.

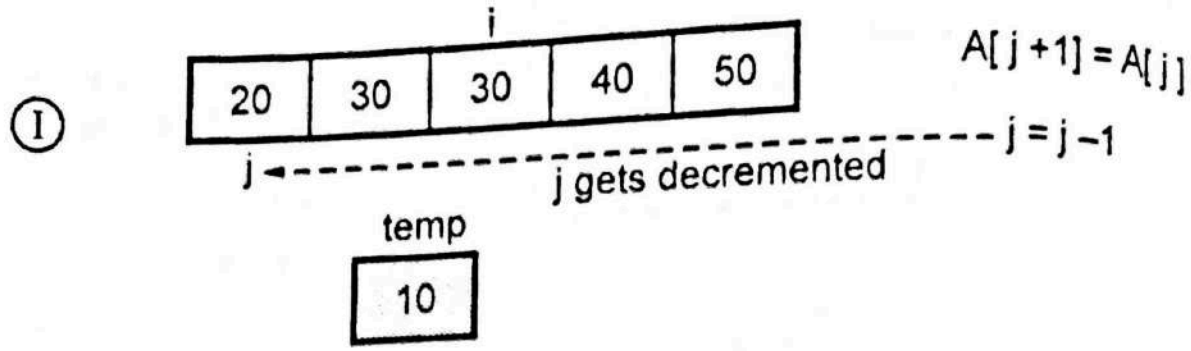


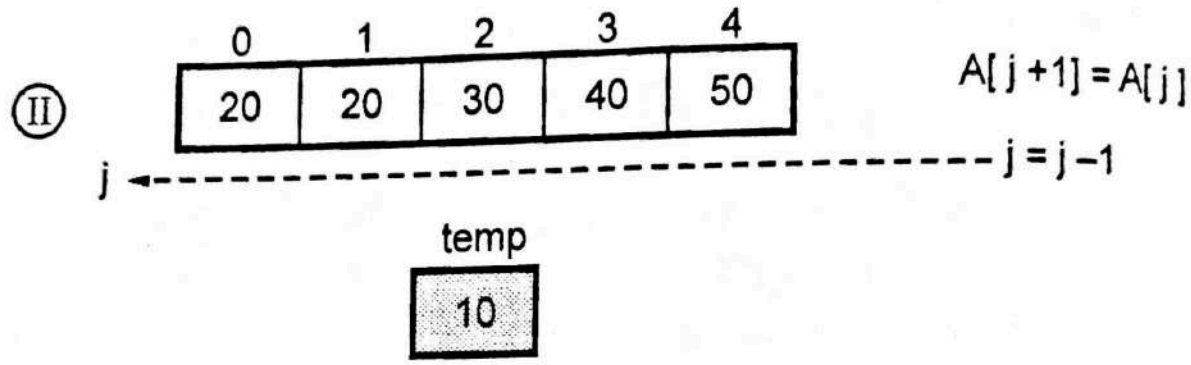
Then list becomes,

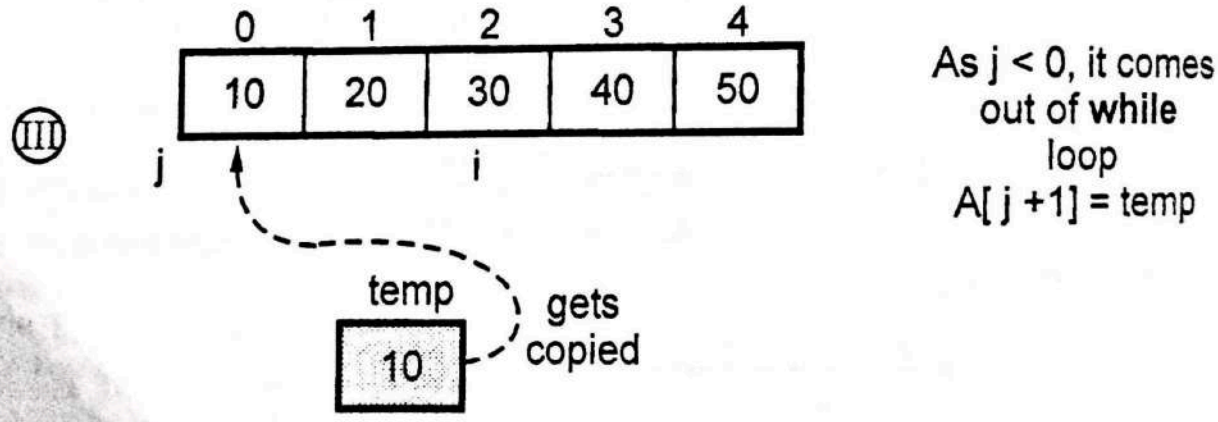




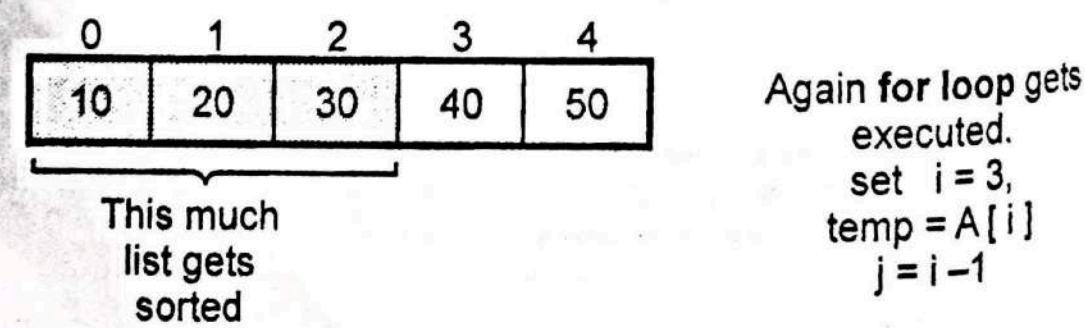
It moves to **while loop** As j > = 0 and (A [j] > temp) is **true**, the while loop gets executed.


I
i
20 30 30 40 50
A[j +1] = A[j]
j = j –1
j
j gets decremented
temp
10

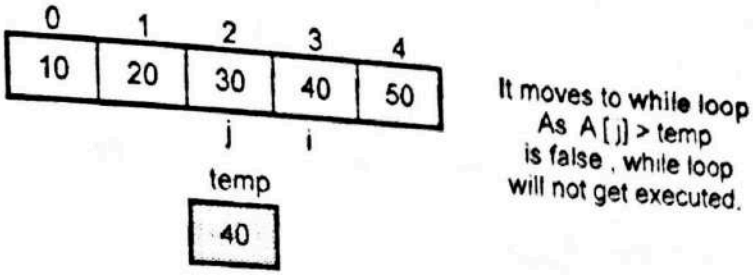

II
0 1 2 3 4
20 20 30 40 50
A[j +1] = A[j]
j = j –1
j
temp
10

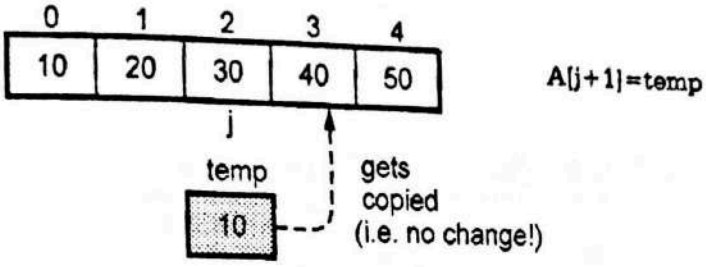

III
0 1 2 3 4
10 20 30 40 50
j
i
As j < 0, it comes out of while loop
A[j +1] = temp
temp
10
gets copied

Thus,

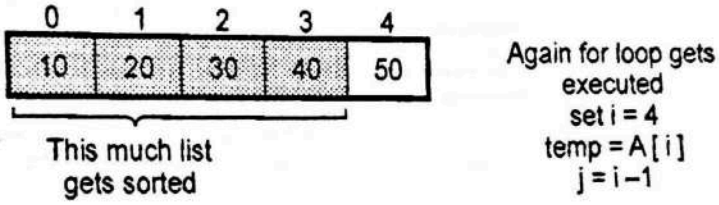

0 1 2 3 4
10 20 30 40 50
This much list gets sorted
Again for loop gets executed.
set i = 3,
temp = A [i]
j = i –1

Then,

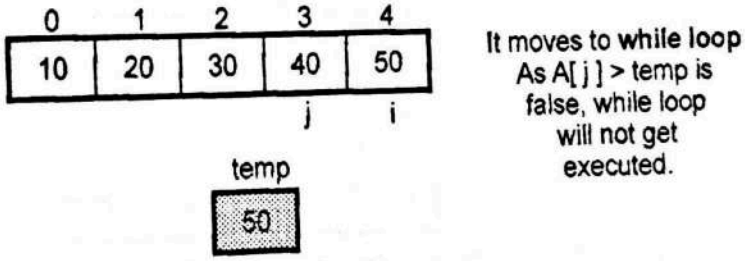


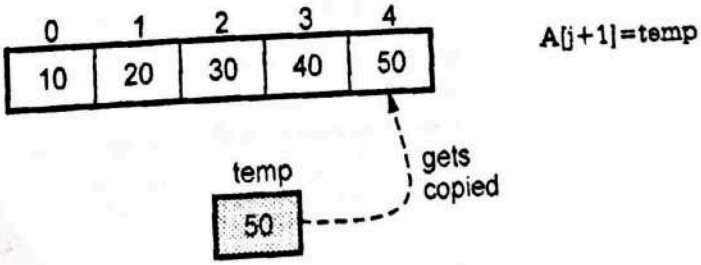


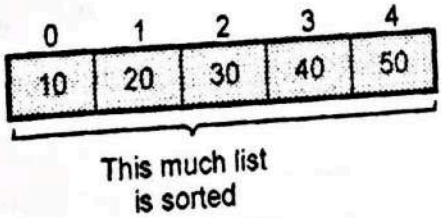
Then,



Then,







Thus we have scanned the entire list and inserted the elements at corresponding locations. Thus we get the sorted list by insertion sort.

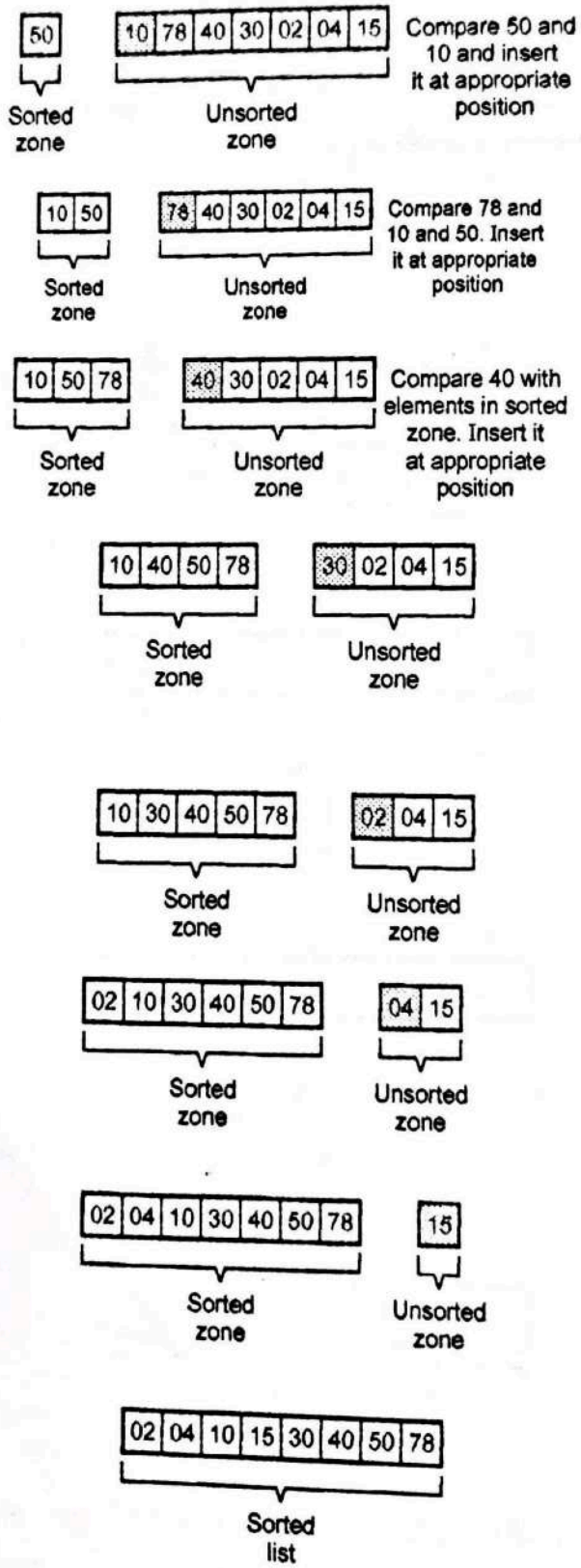
Example 2.5.1 *Sort the following numbers using insertion sort. Show all passes*

50, 10, 78, 40, 30, 02, 04, 15

Solution : Consider the list of elements as -

0	1	2	3	4	5	6	7
50	10	78	40	30	02	04	15

The process starts with first element.



Example 2.12 *Compare the insertion sort and selection sort with*

i) Efficiency ii) Sort stability iii) Passes

Solution : i) Efficiency : The code for selection sort is as follows,

```
for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
        {
                if(ai>aj)
                        swap ai and aj
        }
```

In the ith pass, n-i comparisons will be needed to select the smallest element. n-1 passes are required to sort the array. Thus, the number of comparisons needed to sort an array having n elements,

$$= (n-1)+(n-2)+...+2+1 = \frac{n(n-1)}{2} = \frac{1}{2}\left(n^2-n\right)$$

$$\approx 0\left(n^2\right)$$

Insertion sort :

```
for(i=1;i<n;i++)
{
        k=a[i];
        for(j=i-1;j>=0&&y<a[i])
                a[j+1]=a[j];
                a[j+1]=1;
}
```

Inner loop is data sensitive. If the input list is presented for sorting is presorted then the test a[i] > y in the inner loop will fail immediately. Thus, only one comparison will be made in each pass.

Thus the total number of comparisons (Best case)

$= n - 1 \approx n$ for large n.

If the numbers to be sorted are initially in descending order then the inner loop will make i iterations in i^{th} pass.

∴ Total number of comparisons.

$$= 1+2+3+...+(n-1) = \frac{n(n-1)}{2} = \frac{1}{2}\left(n^2-n\right) \approx n^2 \text{ for large n.}$$

ii) Sort stability : Both the techniques are stable. Both of them have same time complexity.

iii) Passes : Both selection and insertion sort requires n-1 passes.

Insertion sort is more stable than selection sort.

Insertion sort requires less overhead while sorting data. Selection sort requires more number of comparisons. In both the algorithms, elements are read in linear fashion and adjacent elements are compared. Hence identical numbers will maintain their relative sequence even in the sort list.

2.10 Quick Sort

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows :

Divide : Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.

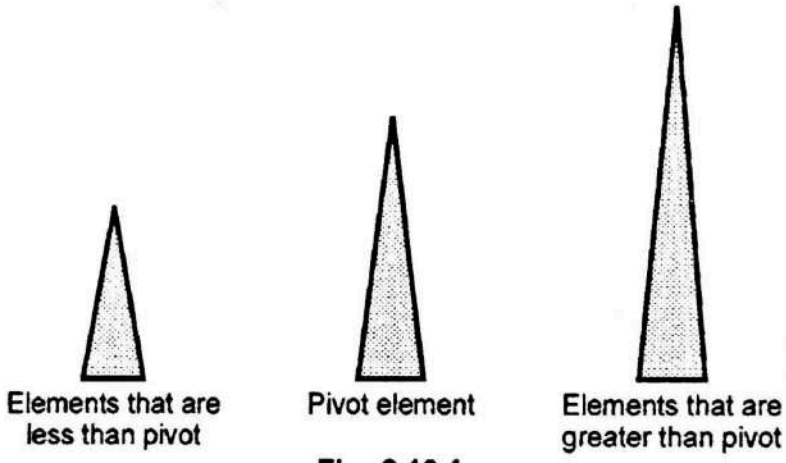


Fig. 2.10.1

Conquer : Recursively sort the two sub arrays.

Combine :

Combine all the sorted elements in a group to form a list of sorted elements.

Algorithm

The quick sort algorithm is performed using following two important functions - *Quick* and *partition*. Let us see them -

```
Algorithm Quick(A[0...n-1],low,high)
//Problem Description : This algorithm performs sorting of
//the elements given in Array A[0...n-1]
//Input: An array A[0...n-1] in which unsorted elements are
//given. The low indicates the leftmost element in the list
```

```
//and high indicates the rightmost element in the list
//Output: Creates a sub array which is sorted in ascending
//order
if(low<high)then
//split the array into two sub arrays
m ← partition(A[low...high])// m is mid of the array
Quick(A[low...m-1])
Quick(A[mid+1...high] )
```

In above algorithm call to partition algorithm is given. The *partition* performs arrangement of the elements in ascending order. The recursive *quick* routine is for dividing the list in two sub lists. The pseudo code for *Partition* is as given below -

```
Algorithm Partition (A[low...high])
//Problem Description: This algorithm partitions the
//subarray using the first element as pivot element
//Input: A subarray A with low as left most index of the
//array and high as the rightmost index of the array.
//Output: The partitioning of array A is done and pivot
//occupies its proper position. And the rightmost index of
//the list is returned
pivot ← A[low]
i ← low
j ← high+1
while(i<=j) do
{
 while(A[i]<=pivot) do
  i ← i+1
 while(A[j]>=pivot) do
  j ← j-1;
 if(i<=j) then
          swap(A[i],A[j])//swaps A[i] and A[j]
}
swap(A[low],A[j])//when i crosses j swap A[low] and A[j]
return j//rightmost index of the list
```

The partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the right of pivot. In other words pivot is occupying its proper position and the partitioned list is obtained in an ordered manner.

Analysis

When pivot is chosen such that the array gets divided at the mid then it gives the best case time complexity. The best case time complexity of quick sort is $O(n\log_2 n)$.

The worst case for quick sort occurs when the pivot is minimum or maximum of all the elements in the list. This can be graphically represented as -

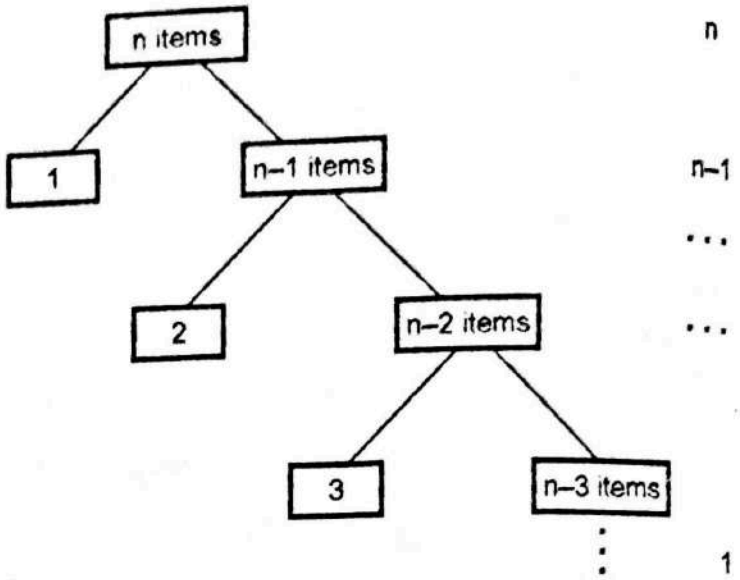


Fig. 2.10.2

This ultimately results in $O(n^2)$ time complexity. When array elements are randomly distributed then it results in average case time complexity, and it is $O(n\log_2 n)$.

Example 2.10.1 *Consider following numbers, sort them using quick sort. Show all passes to sort the values in ascending order*

25, 57, 48, 37, 12, 92, 86, 33

GTU : May-09, Marks 6

Solution : Consider the first element as a pivot element.

25	57	48	37	12	92	86	33
↑	↑						↑
Pivot	i						j

Now, if A[i] < Pivot element then increment i. And if A[j] > Pivot element then **decrement j. When** we get these above conditions to be false, Swap A[i] and A[j]

25	33	48	37	12	92	86	57
↑	i						j
Pivot							

25	33	48	37	12	92	86	57
↑	i			j			
Pivot							

Swap A[i] and A[j]

Low							High
25	12	48	37	33	92	86	57
	j	i					

As j > i Swap A[Low] and A[j]

After pass 1 :

[12]	25	[48	37	33	92	86	57]
		↑	↑				↑
		Pivot	i				j
12	25	[48	37	33	92	86	57]
					i		j
		Low					
12	25	[48	37	33	92	86	57]
				j	i		

As j > i, we will swap A[j] and A[Low]

After pass 2 :

12	25	[33	37]	48	[92	86	57]

After pass 3 :

12	25	33	37	48	[92	86	57]

Assume 92 to be pivot element

12	25	33	37	48	[92	86	57]
					↑	i	j
					Pivot		
12	25	33	37	48	92	86	57
							j i

As j > i swap 92 with 57.

After pass 4 :

12	25	33	37	48	57	[86	92]

After pass 5 :

12 25 33 37 48 57 86 92

is a sorted list.

C++ Program

```
/*****************************************************************************
 Program to sort the elements in ascending order using Quick Sort
*****************************************************************************/
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 10
class Quick
{
 private:
        int arr[SIZE];
 public:
        int get_data(int);
        void quicksort(int,int);
        int partition(int,int);
        void swap(int,int);
        void display(int);
};

/*
 This function is to input the elements
*/

int Quick::get_data(int n)
        {
                int i;
                cout<<"\n Enter Total numbers to sort : ";
                cin>>n;
                for(i=0;i<n;i++)
                {
                 cout<<"\nEnter Element ";
                 cin>>arr[i];
                }
                return n;
        }
/*
This function is to sort the elements in a sublist
*/
void Quick::quicksort(int p,int q)
```

```
{
int j;
if(p<q)
{
 j=partition(p,q+1);//setting pivot element
 quicksort(p,j-1);//splitting of list
 quicksort(j+1,q);//splitting  of list
}
}
/*
This function is to partition a list and decide the pivot element
*/
int Quick::partition(int m,int p)
{
 int pivot=arr[m];
 int i=m,j=p;
 do
  {
   do
    {
     i++;
    }while(arr[i]<pivot);
   do
    {
     j--;
    }while(arr[j]>pivot);
   if(i<j)
    swap(i,j);
  }while(i<j);
  arr[m]=arr[j];
  arr[j]=pivot;
  return j;
}
void Quick::swap(int i,int j)
{
 int p;
 p=arr[i];
 arr[i]=arr[j];
 arr[j]=p;
}

void Quick::display(int n)
{
     for(int i=0;i<n;i++)
```

```
        cout<<" "<<arr[i];
}
void main()
{
 Quick obj;//for integer elements
 int n,i;
 clrscr();
 cout<<"\n\t\t Quick Sort Method \n";
 n=obj.get_data(n);
 obj.quicksort(0,n-1);
 cout<<"\n\n\t Sorted Aarry Is: \n";
 obj.display(n);
 getch();
}
```

Output

```
            Quick Sort Method

 Enter Total numbers to sort : 5

Enter Element 30

Enter Element 50

Enter Element 10
Enter Element 20

Enter Element 40

      Sorted Array Is:
      10 20 30 40 50
```

Example 2.16.2 *What is the complexity of Quicksort ? Show the tree of calls for the quick-sort algorithm using the final element as a pivot on the array 4, 2, 3, 9, 5, 8, 6, 1.*

Solution : The best and average case time complexity of quick sort is **O(nlogn)** and the worst case time complexity is $O(n^2)$.

Example :

0	1	2	3	4	5	6	7
Pivot 4	*i* 2	3	9	5	8	6	*j* 1
4	2	3	*i* 9	5	8	6	*j* 1
4	2	3	*j* 1	*i* 5	8	6	9
1	2	3	4	5	8	6	9
Pivot 1	*i* 2	*j* 3					
1	2	3					
				Pivot 5	*i* 8	6	*j* 9
				j 5	*i* 8	6	9
				5	*Pivot* 8	*i* 6	*j* 9
					8	*j* 6	*i* 9
					6	8	9
				5	6	8	9
1	2	3	4				

From above computations, tree of recursive calls is as follows -

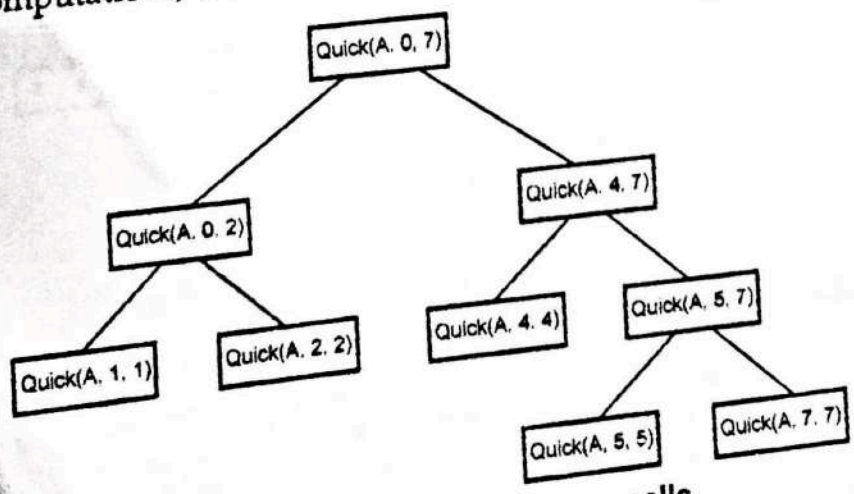


Fig. 2.10.3 Tree of recursive calls

2.11 Merge Sort

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

Merge sort on an input array with n elements consists of three steps :

Divide : Partition array into two sub lists s1 and s2 with n/2 elements each.

Conquer : Then sort sub list s1 and sub list s2.

Combine : Merge s1 and s2 into a unique sorted group.

Example 2.11.1 *Sort the following elements using merge sort.*

10, 5, 7, 6, 1, 4, 8, 3, 2, 9.

Solution :

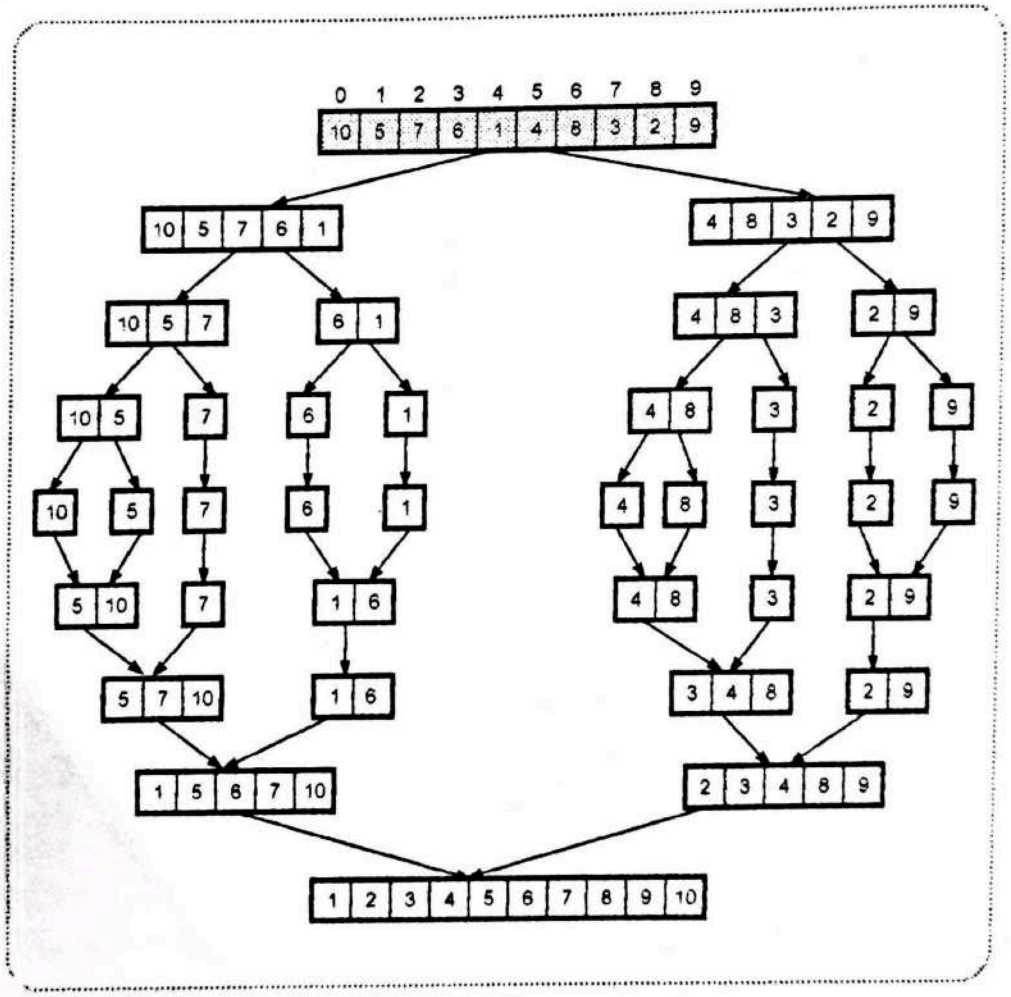


Fig. 2.11.1

2.11.1 Algorithm

```
Algorithm MergeSort(int A[0...n-1],low,high)
//Problem Description : This algorithm is for sorting the
//elements using merge sort
//Input : Array A of unsorted elements, low as beginning
```

```
//pointer of array A and high as end pointer of array A
//Output : Sorted array A[0...n - 1]
if(low < high)then
{
        mid←(low+high)/2              // split the list at mid
        MergeSort(A,low,mid)          // first sublist
        MergeSort(A,mid+1,high)       // second sublist
        Combine(A,low,mid,high)       // merging of two sublists
}
Algorithm Combine(A[0...n-1],low, mid, high)
{
    k ← low;    // k as index for array temp
    i ← low;    // i as index for left sublist of array A
    j ← mid+1 // j as index for right sublist of array A
    while(i <= mid and j <= high)do
    {
        if(A[i]<=A[j])then
    // if smaller element is present in left sublist
        {
            // copy that smaller element to temp array
            temp[k] ← A[i]
            i ← i+1
            k ← k+1
        }
        else  //smaller element is present in right sublist
        {
            //copy that smaller element to temp array
            temp[k] ← A[j]
            j ← j+1
            k ← k+1
        }
    }
    //copy remaining elements of left  sublist to temp
    while(i<=mid)do
    {
            temp[k] ← A[i]
            i ← i+1
            k ← k+1

    }
    //copy remaining elements of right sublist to temp
    while(j<=high)do
    {
            temp[k] ← A[j]
            j ← j+1
            k ← k+1

    }
```

Logic explanation

To understand above algorithm consider a list of elements as

70	20	30	40	10	50	60
0	1	2	3	4	5	6
↑ Low			↑ Mid			↑ High

Then we will first make two sublists as

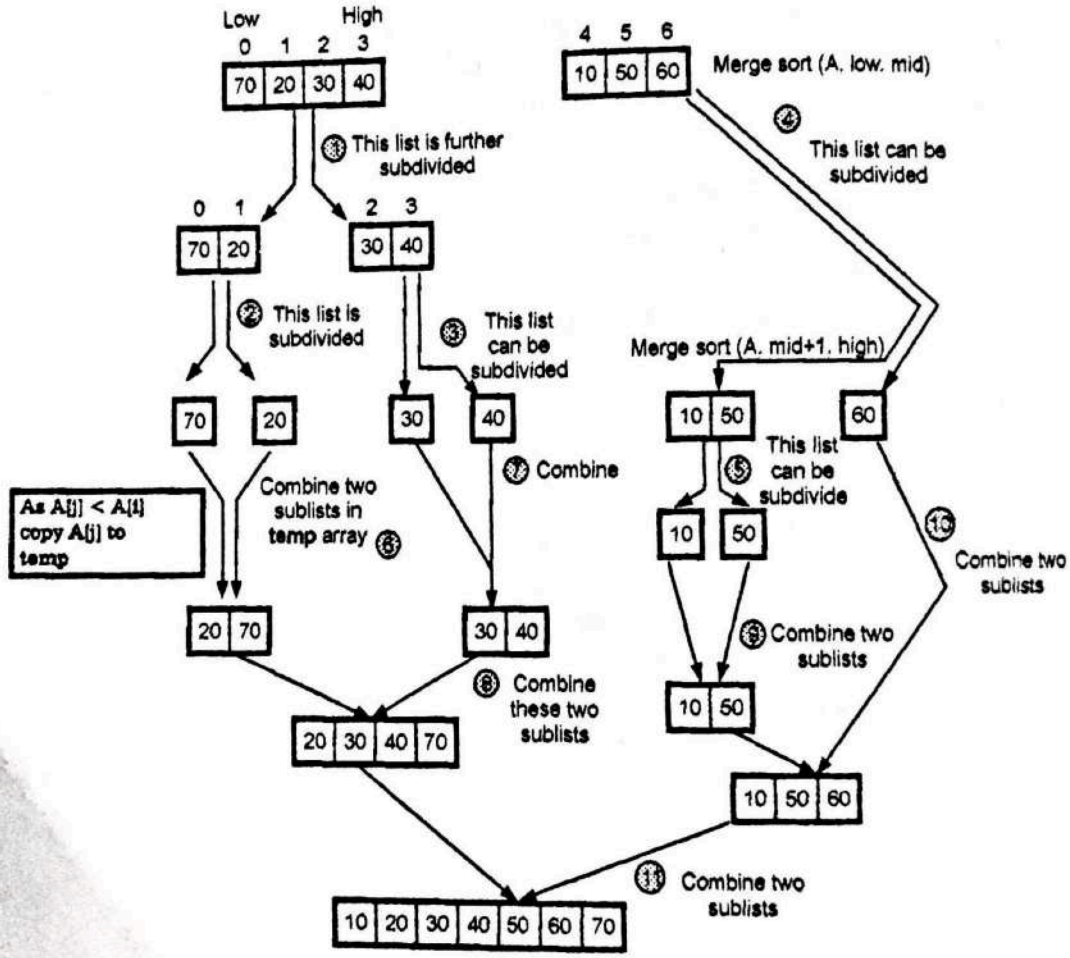
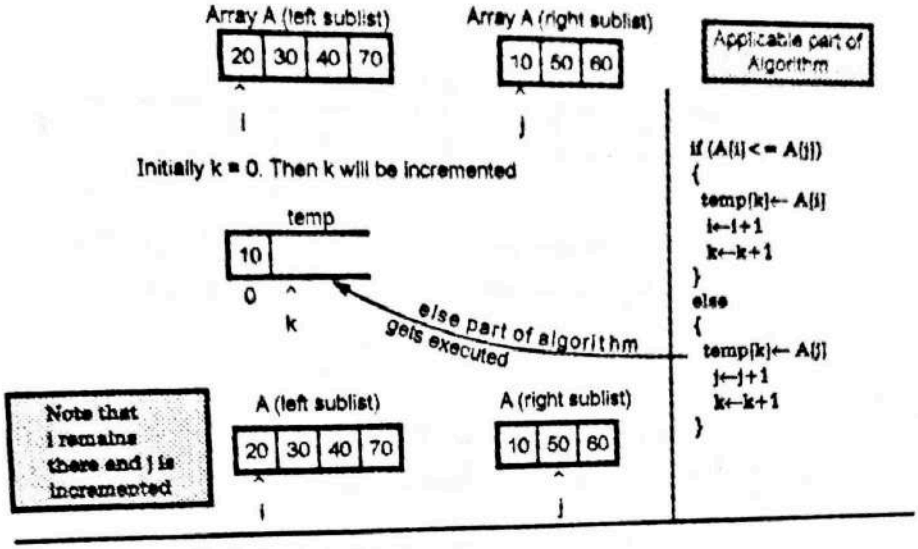
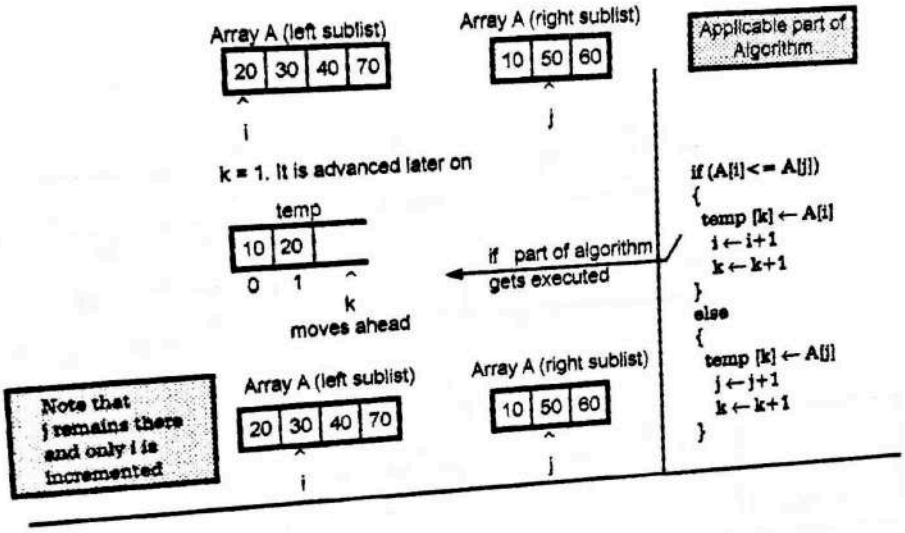


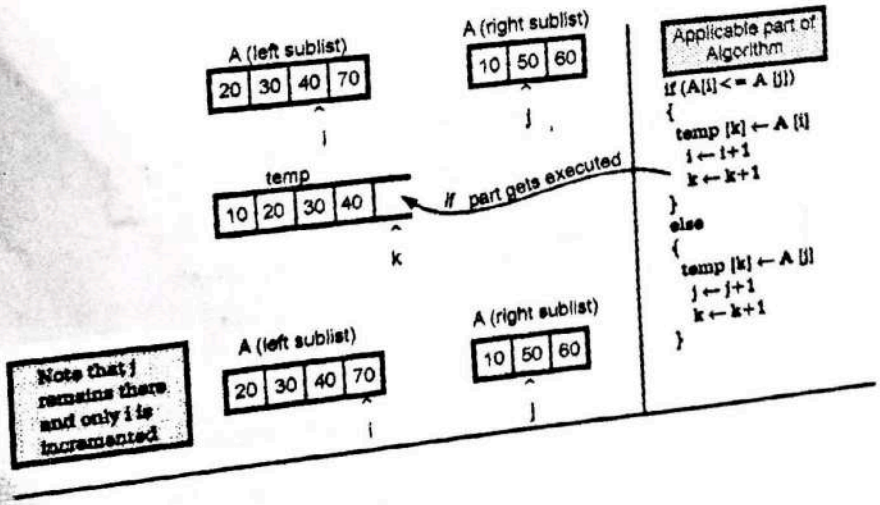
Fig. 2.11.2 Merge and combine operations

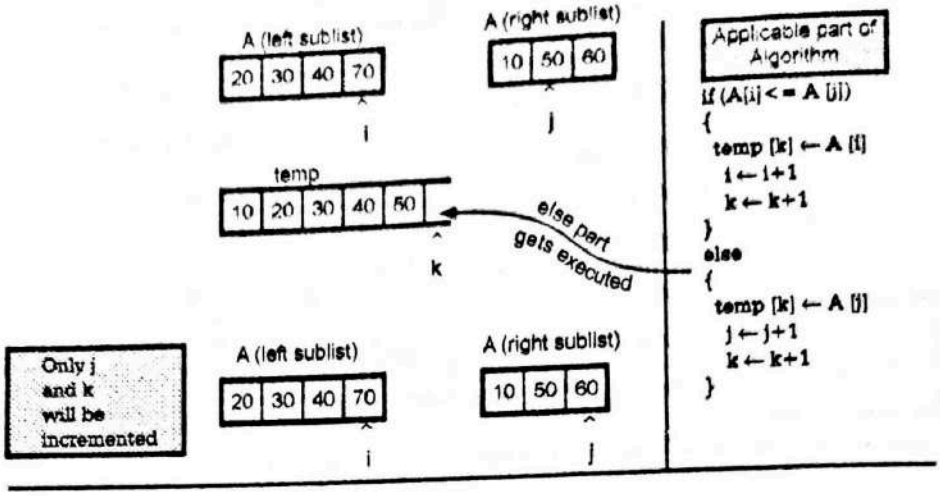
Let us see the **combine** operation more closely with the help of some example.

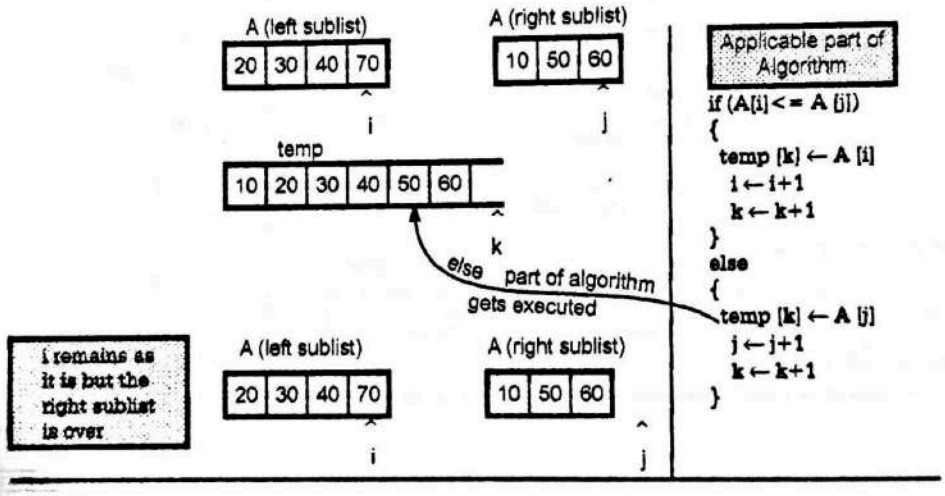
Consider that at some instance we have got two sublist 20, 30, 40, 70 and 10, 50, 60, then

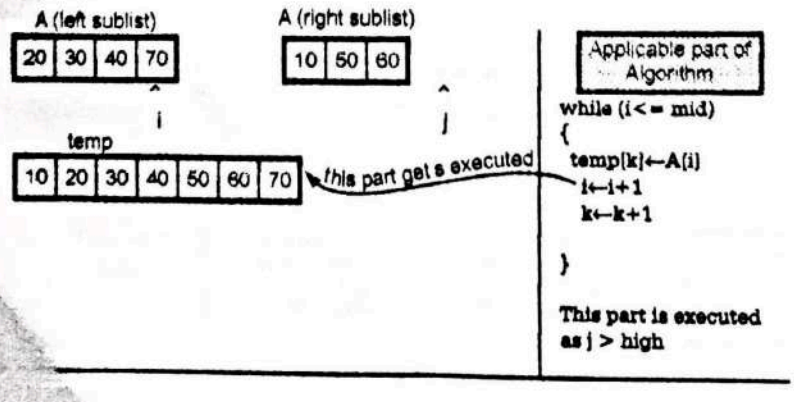

Array A (left sublist)
20 30 40 70
i
Array A (right sublist)
10 50 60
j
Applicable part of Algorithm
Initially k = 0. Then k will be incremented
temp
10
0
k
else part of algorithm gets executed
if (A[i] <= A[j])
{
temp[k]← A[i]
i←i+1
k←k+1
}
else
{
temp[k]← A[j]
j←j+1
k←k+1
}
Note that i remains there and j is incremented
A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j


Array A (left sublist)
20 30 40 70
i
Array A (right sublist)
10 50 60
j
Applicable part of Algorithm
k = 1. It is advanced later on
temp
10 20
0 1
k
moves ahead
if part of algorithm gets executed
if (A[i] <= A[j])
{
temp [k] ← A[i]
i ← i+1
k ← k+1
}
else
{
temp [k] ← A[j]
j ← j+1
k ← k+1
}
Note that j remains there and only i is incremented
Array A (left sublist)
20 30 40 70
i
Array A (right sublist)
10 50 60
j


A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
Applicable part of Algorithm
temp
10 20 30 40
k
if part gets executed
if (A[i] <= A [j])
{
temp [k] ← A [i]
i ← i+1
k ← k+1
}
else
{
temp [k] ← A [j]
j ← j+1
k ← k+1
}
Note that j remains there and only i is incremented
A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j


A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
temp
10 20 30 40 50
k
else part gets executed
Only j and k will be incremented
A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
Applicable part of Algorithm
if (A[i] <= A [j])
{
temp [k] ← A [i]
i ← i+1
k ← k+1
}
else
{
temp [k] ← A [j]
j ← j+1
k ← k+1
}


A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
temp
10 20 30 40 50 60
k
else part of algorithm gets executed
i remains as it is but the right sublist is over
A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
Applicable part of Algorithm
if (A[i] <= A [j])
{
temp [k] ← A [i]
i ← i+1
k ← k+1
}
else
{
temp [k] ← A [j]
j ← j+1
k ← k+1
}


A (left sublist)
20 30 40 70
i
A (right sublist)
10 50 60
j
temp
10 20 30 40 50 60 70
this part get s executed
Applicable part of Algorithm
while (i <= mid)
{
temp[k]←A[i]
i←i+1
k←k+1
}
This part is executed as j > high

Finally we will copy all the elements of array **temp** to array **A**. Thus array **A** contains sorted list.

A

10	20	30	40	50	60	70

2.1.2 Analysis

Let, the recurrence relation for Merge Sort be

$$T(n) = T(n/2) + T(n/2) + cn \qquad \text{for } n > 1$$

i.e.

$$T(n) = 2T(n/2) + cn \qquad \text{... (1)}$$

$$T(1) = 0 \qquad \text{... (2)}$$

Let us apply substitution on equation (1). Assume $n = 2^k$.

$$T(n) = 2T(n/2) + cn$$

$$T(n) = 2T\left(\frac{2^k}{2}\right) + c \cdot 2^k \qquad \text{... } \because n = 2^k$$

$$T(2^k) = 2T(2^{k-1}) + c \cdot 2^k$$

It we put $k = k - 1$ then,

$$T(2^k) = 2T(2^{k-1}) + c \cdot 2^k$$

$$= 2[2T(2^{k-2}) + c \cdot 2^{k-1}] + c \cdot 2^k$$

$$= 2^2 T(2^{k-2}) + 2 \cdot c \cdot 2^{k-1} + c \cdot 2^k$$

$$= 2^2 T(2^{k-2}) + 2 \cdot c \cdot \frac{2^k}{2} + c \cdot 2^k$$

$$= 2^2 T(2^{k-2}) + c \cdot 2^k + c \cdot 2^k$$

$$T(2^k) = 2^2 T(2^{k-2}) + 2c \cdot 2^k$$

Similarly we can write,

$$T(2^k) = 2^3 T(2^{k-3}) + 3c \cdot 2^k$$

$$= 2^4 T(2^{k-4}) + 4c \cdot 2^k$$

...

...

$$\ldots$$

$$= 2^k T(2^{k-k}) + k \cdot c \cdot 2^k$$

$$= 2^k T(2^0) + k \cdot c \cdot 2^k$$

$$T(2^k) = 2^k T(1) + k \cdot c \cdot 2^k \quad \ldots (3)$$

But as per equation (2), T(1) = 0,

∴ Equation (3) becomes,

$$T(2^k) = 2^k \cdot 0 + k \cdot c \cdot 2^k$$

$$T(2^k) = k \cdot c \cdot 2^k$$

But we assumed $n = 2^k$, taking logarithm on both sides.

i.e. $\log_2 n = k$

∴ $T(n) = \log_2 n \cdot cn$

∴ $T(n) = \Theta(n \cdot \log_2 n)$

Hence the average and worst case time complexity of merge sort is $\Theta(n \log_2 n)$.

Time complexity of merge sort

Best case	Average case	Worst case
$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$

2.11.3 Program

```
/*****************************************************************
This program is for Merge Sort
*****************************************************************/
# include<iostream.h>
# include<conio.h>
# include<stdlib.h>
class MS
{
    private:
        int A[10];
    public:
        int n;
        int low,high;
        void get_data();
        void MergeSort(int low,int high);
```

```
        void Combine(int low,int mid,int high);
        void Display();


};
/*
This function is to split the list into sublists
*/
void MS::get_data()
{
        cout<<"\n Enter the length of list : ";
        cin>>n;
        cout<<"\n Enter list elements :\n";
        for(int i=0;i<n;i++)
        cin>>A[i];

}
void MS::MergeSort(int low,int high)
{
    int mid;
    if(low < high)
    {
        mid = (low+high)/2;//split the list at mid
        MergeSort(low,mid);//first sublist
        MergeSort(mid+1,high);//second sublist
        Combine(low,mid,high);//merging of two sublists
    }
}
/* This function is for merging the two sublists
*/
void MS::Combine(int low,int mid,int high)
{
        int i,j,k;
        int temp[10];
        k=low;
        i=low;
        j=mid+1;
        while(i <= mid && j <= high)
        {
        if(A[i]<=A[j])
        {
            temp[k]=A[i];
            i++;
            k++;
```

We compare elements from left sublist and right sublist. If element in the left sublist is lesser than the element in the right sublist then copy that smaller element of left sublist to temp array

```
        }
        else
          {
                    temp[k]=A[j];
                    j++;
                    k++;
          }
        }

        while(i<=mid)
        {
                    temp[k]=A[i];
                    i++;
                    k++;
        }
        while(j<=high)
        {
                    temp[k]=A[j];
                    j++;
                    k++;
        }
        //copy the elements from temp array to A
        for(k=low;k<=high;k++)
                    A[k]=temp[k];
}
/* function to display sorted array */
void MS::Display()
{
        cout<<"\n\n The Sorted Array Is ...\n";
        for(int i=0;i<n;i++)
              cout<<" "<<A[i];
}


void main()
{
        MS obj;
        clrscr();
        cout<<"\n\t\t Merge Sort \n";
        obj.get_data();
        obj.low=0;
        obj.high=obj.n-1;
        obj.MergeSort(obj.low,obj.high);
        obj.Display();
        getch();
}
```

We compare elements from left sublist and right sublist. If element in the right sublist is lesser than the element in the left sublist then copy that smaller element of right sublist to **temp** array

Reached at the end of right sublist and elements of left sublist are remaining. Then copy the remaining elements of left sublist to **temp**

Reached at the end of left sublist and elements of right sublist are remaining. Then copy the remaining elements of right sublist to **temp**

Output

```
Merge Sort

Enter the length of list : 7

Enter list elements :
70
20
30
40
10
50
60

The Sorted Array Is ...
 10 20 30 40 50 60 70
```

Example 2.11.1 *What is the difference between quick sort and merge sort algorithm ?*

Solution :

(1) The merge sort requires additional space for sorting the elements. This is called auxiliary space. While the quick sort does not require any additional space for sorting the elements.

(2) The merge sort is a stable sort while quick sort is not a stable sorting method. That means the order of elements in quick sort gets changed.

(3) The time complexity of merge sort is O (n log n) in all best, worst and average cases. While the time complexity of quick sort degrades if the list is already sorted. It is O(n^2) in worst case.

Review Question

1. *Explain merge sort method with suitable example.*

2.12 Shell Sort

This method is a improvement over the simple insertion sort. In this method the elements at fixed distance are compared. The distance will then be decremented by some fixed amount and again the comparison will be made. Finally, individual elements will be compared. Let us take some example.

Example : If the original file is

	0	1	2	3	4	5	6	7
X array	25	57	48	37	12	92	86	33

Step 1 : Let us take the distance k = 5

So in the first iteration compare

(x [0], x[5])

(x [1], x [(6])

(x [2], x[7])

(x[3])

(x[4])

i.e. first iteration

After first iteration,

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	33	37	12	92	86	48

Step 2 : Initially k was 5. Take some d and decrement k by d. Let us take d = 2

$\therefore$ k = k – d i.e. k = 5 – 2 = 3

So now compare

(x[0], x[3], x[6]), (x[1], x[4], x[7])

(x[2], x[5])

Second iteration

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	33	37	12	92	86	48

After second iteration

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	12	33	37	48	92	86	57

Step 3 : Now k = k – d $\therefore$ k = 3 – 2 = 1

So now compare

(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])

This sorting is then done by simple insertion sort. Because simple insertion sort is highly efficient on sorted file. So we get

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
12	25	33	37	48	57	86	92

C++ Program

```
/*****************************************************************************
Program for sorting the list of elements using shell sort
*****************************************************************************
#include<iostream>
using namespace std;
#define MAX 10
class SHELL
{
    private:
        int a[MAX],n;
    public:
        SHELL();
        void get_data();
        void shellsort();
        void Display();
};
SHELL::SHELL()
{
    for(int i=0;i<MAX;i++)
        a[i]=0;
}
void SHELL::get_data()
{
    cout<"\n Enter total number of elements in an array ";
    cin>>n;
    cout<<"\n Enter some elements in array\n";
    for(int i=0;i<n;i++)
    cin>>a[i];
}
void SHELL::Display()
{
    int i;
    for(i=0;i<n;i++)
    {
        cout<" "<a[i];
    }
}
void SHELL::shellsort()
{
    int i,j,d,k,value;
    d=(n+1)/2;
    for(i=d;i>=1;i=i/2)
    {
        for(j=i;j<=n-1;j++)
        {
            value=a[j];
```

```
                k=j-i;
                while(k>=0 && value<a[k])
                {
                        a[k+i]=a[k];
                        k=k-i;
                }
                a[k+i]=value;
        }
 }
}
int main(void)
{
        int i;
        SHELL obj;
        cout<<"\n\t Program for Shell Sort";
        obj.get_data();
        cout<<"\nBefore sorting  ";
        obj.Display();
        cout<<"\n";
        cout<<"\nAfter sorting  ";
        obj.shellsort();
        obj.Display();
        return 0;
}
```

Output

```
Program for Shell Sort
Enter total number of elements in an array 4
Enter some elements in array
10
 5
 2
20
Before sorting   10 5 2 20
After sorting   2 5 10 20
```

Analysis

Best Case : The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less. In that case the inner loop does not need to any work and a simple comparison will be sufficient to skip the inner sort loop. the other loops gives O(nlogn). The best case of O(n) is reached by using a constant number of increments. Hence the best case time complexity of shell sort is **O (n log n)**

Worst Case and Average Case : The running time of Shellsort depends on the choice of increment sequence. The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect. The wort case and average case time complexity is **O(n)**.

2.13 Comparison of all Sorting Methods

Sorting technique	Best case	Average case	Worst case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Shell sort	$O(n\log n)$	$O(n)$	$O(n)$

2.14 Analysis of Insertion sort, Quick Sort, Binary Search, Hashing

2.14.1 Analysis of Insertion Sort

Best case :

It happens for **almost sorted array**. The number of key comparisons can be

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = (n-1)$$

$$\therefore \quad C_{best}(n) = \Theta(n)$$

Thus the best case time complexity of insertion sort is Θ **(n).**

Average case :

When an array contains randomly distributed elements then it results in average case time complexity. As compared to worst case atleast half number of comparisons are needed to obtain average case complexity.

$$\therefore \quad C_{avg}(n) = \frac{(n-1)n}{2} \times \frac{1}{2} = \frac{n^2 - n}{4} \approx \frac{n^2}{4}$$

$$\therefore \quad C_{avg}(n) = \Theta(n^2)$$

Thus the average case time complexity of insertion sort is $\Theta(n^2)$.

The algorithm can be analysed for worst case, best case and average case time complexities.

Worst case :

When the list is sorted in descending order and if we want to sort such a list in ascending order then it results in worst case behavior of algorithm.

1. The basic operation of algorithm is a key comparison `A[j]>temp`. [Note that the operation given in the innermost loop is called basic operation].
2. The basic operation depends upon the size of list i.e. n–1. Hence the basic operation is considered for the range 1 to (n – 1).
3. The number of key comparisons for such input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$\because \sum_{l}^{u} 1 = u - l + 1$$

$$\therefore \sum_{j=0}^{i-1} 1 = (i-1) - 0 + 1 = 1$$

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

$$= \frac{(n-1)\,n}{2}$$

$$\because \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$C_{worst}(n) = \Theta(n^2)$$

That means worst case time complexity of insertion sort is $\Theta(n^2)$.

Key Point :

Time complexity of insertion sort

Best case	Average case	Worst case
$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$

2.10.2 Analysis of Quick Sort

Best case :

If the array is always partitioned at mid, then it brings the best case efficiency.

The recurrence relation for this algorithm is

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\text{Time required to sort left sub-list}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Time required to sort right sub-list}} + \underbrace{n}_{\text{Time required for partitioning the sub array}}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad ...(1)$$

Assume $T(1) = 0$

Solving recurrence relatiion (1),

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2\left[2 \cdot T\left(\frac{n}{4}\right) + n\right] + n$$

$$= 4T\left(\frac{n}{4}\right) + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$\vdots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

If we assume $n = 2^k$

then $$T(n) = 2^k (T)\left(\frac{2^k}{2^k}\right) + kn$$

$$= 2^k T(1) + kn$$

$$= 2^k \cdot 0 + kn \qquad \because T(1) = 0$$

$$= k \cdot n$$

But as $n = 2^k$, the k becomes $\log_2 n$

$\therefore$ $T(n) = kn$ becomes

$$T(n) = n \cdot \log_2 n$$

Hence the best case time complexity is $O(\log_2 n)$.

Worst case (sorted array)

The worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list. This can be graphically represented as,

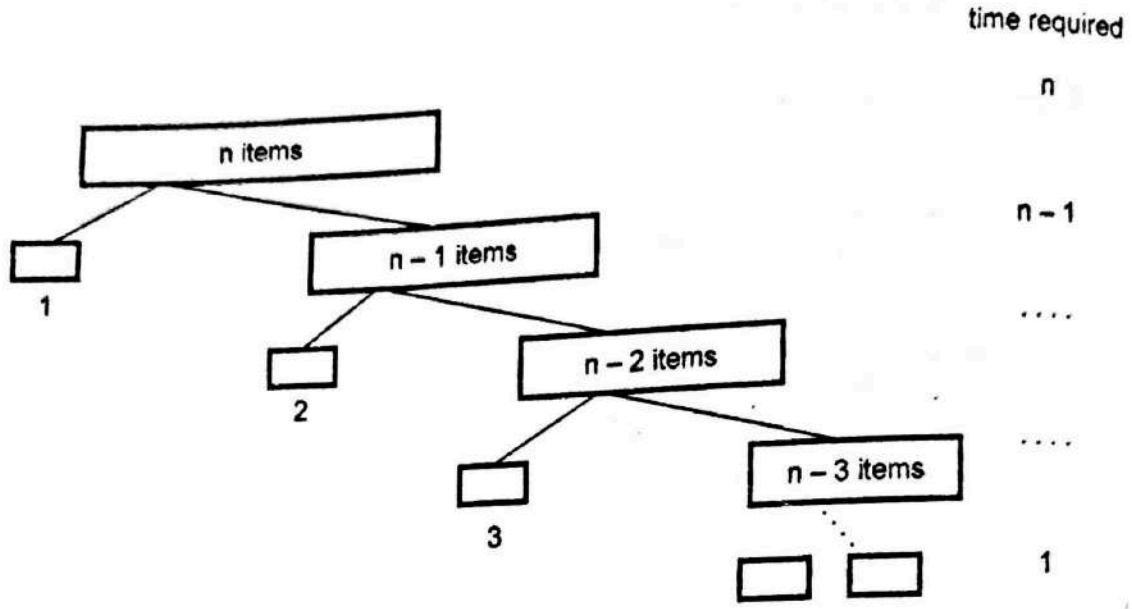


Fig. 2.14.2

We can write it as,

$$C(n) = C(n-1) + n$$

or $$C(n) = n + (n-1) + (n-2) + \ldots + 2 + 1$$

But as we know

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

$$\approx \frac{1}{2} n^2$$

$$\therefore \quad C(n) = \Theta(n^2)$$

The time complexity of worst case of quick sort is $\Theta(n^2)$

Average case (Random array)

For any pivot position i; $i \in \{0, ..., n - 1\}$

- Time for partitioning an array = cn.
- The head and tail subarrays contain i and n – 1 – i items respectively.
- The recurrence relation for average case for sorting is -

$$T(n) = \frac{1}{n}\left[\sum_{i=0}^{n-1}(T(i)+T(n-1-i)+cn)\right]$$

$$= \frac{2}{n}(T(0)+T(1)+.....+T(n-2)+T(n-1)+cn)$$

$$nT(n) = 2(T(0)+T(1)+.....+T(n-2)+T(n-1)+cn^2)$$

$$(n-1)T(n-1) = 2(T(0)+T(1)+.....+T(n-2)+c(n-1)^2)$$

$$nT(n)-(n-1)T(n-1) = 2T(n-1)+2cn-c \approx 2T(n-1)+2cn$$

$$\therefore \quad nT(n) \approx (n+1)T(n-1)+2cn$$

Rearranging above equation and dividing by n(n + 1)

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n}+\frac{2c}{n+1}$$

Telescoping above equation as,

$$\frac{T(n)}{n+1}+\frac{T(n-1)}{n}+\frac{T(n-2)}{n-1}+....+\frac{T(2)}{3}+\frac{T(1)}{2}$$

$$-\frac{T(n-1)}{n}+\frac{T(n-2)}{n-1}-....-\frac{T(2)}{3}-\frac{T(1)}{2}-\frac{T(0)}{1}$$

$$= \frac{2c}{n+1}+\frac{2c}{n}+.....+\frac{2c}{3}+\frac{2c}{2}$$

This gives us,

$$\frac{T(n)}{n+1}-\frac{T(0)}{1} = 2c\left(\frac{1}{2}+\frac{1}{3}+....+\frac{1}{n+1}+\frac{1}{n}\right)$$

$$\therefore \quad \frac{T(n)}{n+1} = \frac{T(0)}{1}+2c\left(\frac{1}{2}+\frac{1}{3}+....+\frac{1}{n}+\frac{1}{n+1}\right)$$

$$\approx 2c\,(H_{n+1} - 1)$$

Where H is a harmonic number and $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots\ldots + \frac{1}{n}$

$$\therefore \quad \frac{T(n)}{n+1} = c' \log n$$

$$T(n) = (n+1)\,c' \log n$$

$$T(n) \approx \Theta(n \log n)$$

$\therefore$ **Average case time complexity is $\Theta(n\log n)$**

2.4.3 Analysis of Binary Search

Best Case : In the best case, the item X is the middle in the array A. A constant number of comparisions (actually just 1) are required.

Hence the best case time complexity is **O(1)**.

Worst and Average Case : In the worst case, the item X does not exist in the array A at all. For average case, the average number of iterations for successful find are considered. Hence following recurrence relation can be used for finding the worst case and average case time complexity.

$$T(n) = T(n/2) + c$$

Time required to compare left and right sublists

Constant time required to compare middle element

Now solving the above recurrence relation , we get -

$$T(n) = T\left(\frac{n}{2}\right) + c \qquad \ldots(1)$$

$$T(i) = 1 \qquad \ldots(2)$$

If we assume $n = 2^k$ then equation (1) becomes.

$$T(2^k) = T\left(\frac{2^k}{2}\right) + c$$

$$T(2^k) = T(2^{k-1}) + c$$

$$= T(2^{k-2}) + 2c$$

$$= T(2^{k-3}) + 3c$$

$$\vdots$$

$$T(n) = T(2^{k-k}) + kc$$

$$= T(2^0) + kc$$

$$= 1 + kc \qquad \because \; T(2^0) = T(1) = 1$$

As $n = 2^k$, $k = \log_2 n$, Hence we can replace k by $\log_2 n$

$$\therefore \qquad T(n) = 1 + \log_2 n \cdot c \qquad \ldots(3)$$

From equation 3, if we neglect constants and consider only order of magnitude we get,

$$\boxed{T(n) = O(\log_2 n)}$$

2.14.4 Analysis of Hashing

The time complexity of various operations of various hashing techniques are -

1) Linear Probing

	Worst Case	Average Case	Best Case
Search	O(n)	O(1)	O(1)
Insert	O(n)	O(1)	O(1)
Delete	O(n)	O(1)	O(1)

2) Separate Chaining

	Worst Case	Average Case	Best Case
Search	O(n)	O(1)	O(1)
Insert	O(n)	O(1)	O(1)
Delete	O(n)	O(1)	O(1)

3.1 Concept of Stack

SPPU : Dec-16, May-19, Marks 6

Let us have the formal definition of the stack.

Definition :

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottommost element and 60 will be the topmost element in the stack. A stack is shown in Fig. 3.1.1.

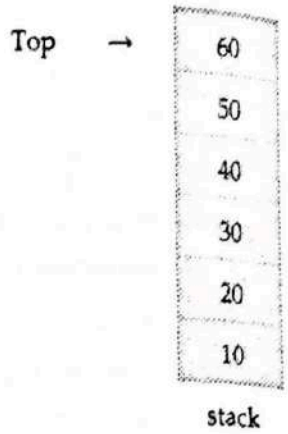


Fig. 3.1.1 Stack

Example

The typical example can be a stack of coins. The coins can be arranged one on another, when we add a new coin it is always placed on the previous coin and while removing the coin the recently placed coin can be removed. The example resembles the concept of stack exactly.

Review Questions

1. *Explain stack and write pseudo C algorithm for PUSH and POP operations of stack.* SPPU : Dec-16, Marks 6
2. *Explain the following linear data structure - stack.* SPPU : May-19, Marks 4

3.2 Stack as an ADT

Stack is a data structure which posses LIFO i.e. Last In First Out property. The abstract data type for stack can be as given below.

AbstractDataType stack

{

Instances : Stack is a collection of elements in which insertion and deletion of elements is done by one end called **top.**

Preconditions :

1. **Stfull () :** This condition indicates whether the stack is full or not. If the stack is full then we cannot insert the elements in the stack.
2. **Stempty () :** This condition indicates whether the stack is empty or not. If the stack is empty then we cannot pop or remove any element from the stack.

Operations :

1. **Push :** By this operation one can push elements onto the stack. Before performing push we must check stfull () condition.
2. **Pop :** By this operation one can remove the elements from stack. Before popping the elements from stack We should check stempty () condition.

}

3.3 Implementation of Stack using Sequential Organization

Declaration 1 :

```
#define size 100
int stack[size], top = - 1;
```

In the above declaration stack is nothing but an array of integers. And most recent index of that array will act as a top.

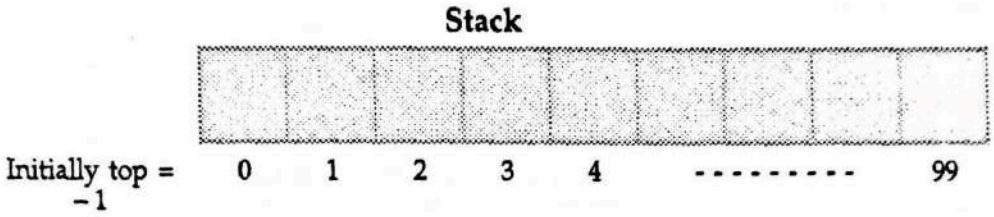


Fig. 3.3.1 Stack using one dimension array

The stack is of the size 100. As we insert the numbers, the top will get incremented. The elements will be placed from 0th position in the stack. At the most we can store 100 elements in the stack, so at the most last element can be at (size – 1) position, i.e., at index 99.

Declaration 2 :

```
#define size 10
struct stack {
        int s[size];
        int top;
      }st;
```

In the above declaration stack is declared as a structure.

Now compare declaration 1 and 2. Both are for stack declaration only. But the second declaration will always preferred. Why ? Because in the second declaration we have used a structure for stack

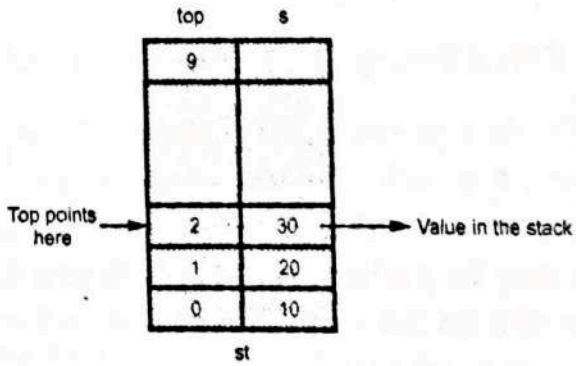


Fig. 3.3.2 Stack using structure

elements and top. By this we are binding or co-relating top variable with stack elements. Thus top and stack are associated with each other by putting them together in a structure. The stack can be passed to the function by simply passing the structure variable.

We will make use of the second method of representing the stack in our program.

The stack can also be used in the databases. For example if we want to store marks of all the students of fourth semester we can declare a structure of stack as follows.

```
# define size 60
typedef struct student
    {
        int roll_no;
        char name[30];
        float marks;
    }stud;
    stud s1[size];
    int top = -1;
```

The above structure will look like this.

	roll_no	name	marks
59			
⋮			
top = 3 →	40	Mita	66
2	30	Rita	91
1	20	Geeta	88.3
0	10	Seeta	76.5

S1

Thus we can store the data about whole class in our stack. The above declaration means creation of stack. Hence we will write only push and pop function to implement the stack. And before pushing or popping we should check whether stack is empty or full.

3.3.1 Stack Empty Operation

Initially stack is empty. At that time the top should be initialized to – 1 or 0. If we set top to – 1 initially then the stack will contain the elements from 0th position and if we set top to 0 initially, the elements will be stored from 1st position, in the stack. Elements may be pushed onto the stack and there may be a case that all the elements are removed from the stack. Then the stack becomes empty. Thus whenever top reaches to – 1 we can say the stack is empty.

```
int stempty()
    {
     if(st.top==-1)
      return 1;
     else
      return 0;
    }
```

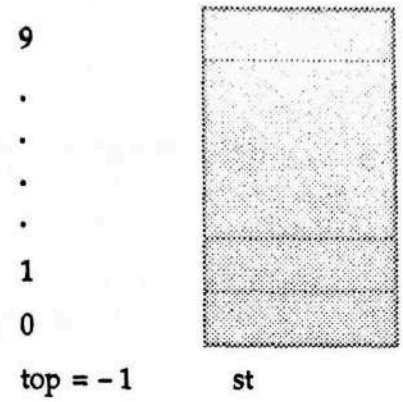


Fig. 3.3.3 Stack empty condition

Key Point *If top = – 1 means stack empty.*

3.3.2 Stack Full Operation

In the representation of stack using arrays, size of array means size of stack. As we go on inserting the elements the stack gets filled with the elements. So it is necessary before inserting the elements to check whether the stack is full or not. Stack full condition is achieved when stack reaches to maximum size of array.

```
int stfull()
{
 if(st.top>=size-1)
  return 1;
 else
  return 0;
}
```

Thus stfull is a Boolean function if stack is full it returns 1 otherwise it returns 0.

Key Point *If top = – 1 means stack empty.*

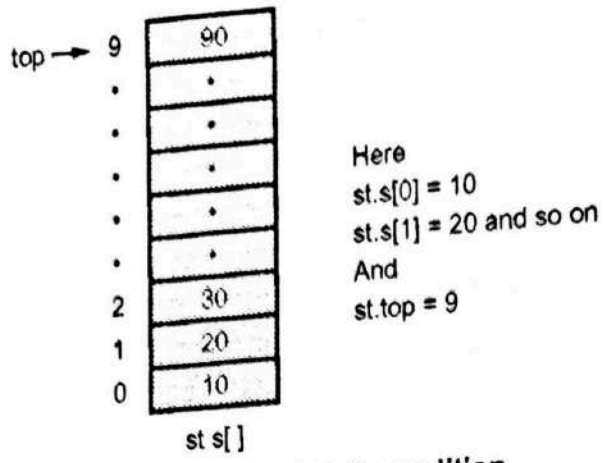


Fig. 3.3.3 Stack full condition

3.3.3 Push and Pop Operations

We will now discuss the two important functions which are carried out on a stack. push is a function which inserts new element at the top of the stack. The function is as follows.

```
void push(int item)
{
st.top++; / * top pointer is set to next location */
st.s[st.top] =item; /* placing the element at that location */
}
```

Note that the push function takes the parameter **item** which is actually the element which we want to insert into the stack - means we are pushing the element onto the stack. In the function we have checked whether the stack is full or not, if the stack is not full then only the insertion of the element can be achieved by means of push operation.

Now let us discuss the operation pop, which deletes the element at the top of the stack. The function pop is as given below -

Note that always top element can be deleted.

```
int pop()
{
        int item;
        item = st.s[st.top];
        st.top--;
        return(item);
}
```

In the choice of pop- it invokes the function 'stempty' to determine whether the stack is empty or not. If it is empty, then the function generates an error as stack underflow ! If not, then pop function returns the element which is at the top of the stack. The value at the top is stored in some variable as item and it then decrements the value of the top, which now points to the element which is just under the element being retrieved from the stack. Finally it returns the value of the element stored in the variable item. Note that this is what called as logical deletion and not a physical deletion, i.e.

even when we decrement the top, the element just retrieved from the stack remains there itself, but it no longer belongs to the stack. Any subsequent push will overwrite this element.

The push operation can be shown by following Fig. 3.3.4

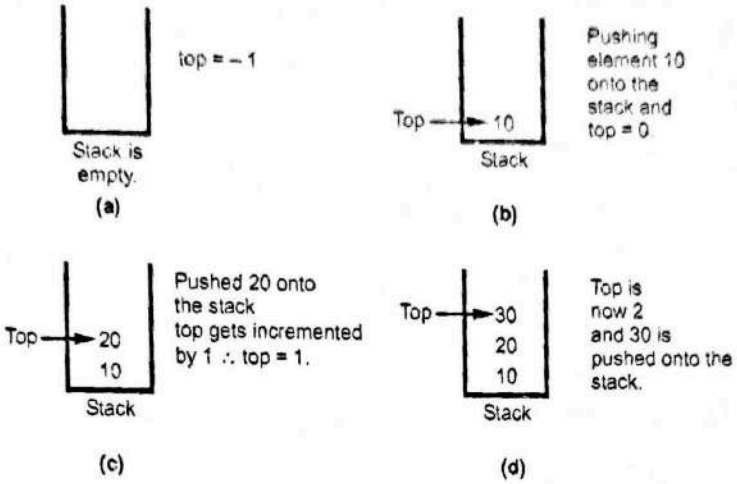


Fig. 3.3.4 Performing push operation

The pop operation can be shown by following Fig. 3.3.5

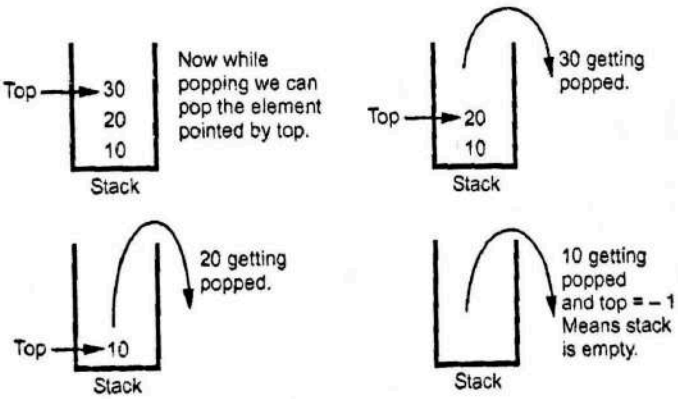


Fig. 3.3.5 Performing pop operation

Key Point *If stack is empty, then we cannot pop and if stack is full we cannot push.*

'C++' Program

```
/*****************************************************************
Program for implementing a stack using arrays. It involves various operations such as
push, pop, stack empty, stack full and display.
*****************************************************************/
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#define size 5
/* stack structure */
```

```
class STACK_CLASS
{
 private:
 struct stack
 {
     int s[size];
     int top;
  } st;
 public:
        STACK_CLASS();
        int stfull();
        void push(int item);
        int stempty();
        int pop();
        void display();
};
//constructor is used to initialise stack
STACK_CLASS::STACK_CLASS()
{
        st.top=-1;
        for(int i=0;i<size;i++)
                st.s[i]=0;
}

/*
 The stfull Function
 Input:none
 Output:returns 1 or 0 for stack full or not
 Called By:main
 Calls:none
*/
int STACK_CLASS::stfull()
{
 if(st.top>=size-1)
        return 1;
 else
        return 0;
}
/*
 The push Function
 Input:item which is to be pushed
 Output:none-simply pushes the item onto the stack
 Called By:main
 Calls:none
*/
void STACK_CLASS::push(int item)
{
 st.top++;
 st.s[st.top] =item;
}
```

```
/*
The stempty Function
Input:none
Output:returns 1 or 0 for stack empty or not
Called By:main
Calls:none
*/
int STACK_CLASS::stempty()
{
if(st.top==-1)
 return 1;
else
 return 0;
}
/*
The pop Function
Input:none
Output:returns the item which is popped from the stack
Called By:main
Calls:none
*/
int STACK_CLASS::pop()
{
int item;
item=st.s[st.top];
st.top--;
return(item);
}
/*
The display Function
Input:none
Output:none-displays the contents of the stack
Called By:main
Calls:none
*/
void STACK_CLASS::display()
{
int i;
if(stempty())
  cout<"\n Stack Is Empty!";
else
{
for(i=st.top;i>=0;i--)
cout<"\n"<st.s[i];
}
}
/*
The main Function
Input:none
Output:none
```

```
 Called By:O.S.
 Calls:push,pop,stempty,stfull,display
*/
void main(void)
{
      int item,choice;
      char ans;
      STACK_CLASS obj;
      clrscr();
      cout<"\n\t\t Implementation of Stack";
      do
      {
            cout<"\n Main Menu";
            cout<"\n1.Push\n2.Pop\n3.Display\n4.exit";
            cout<"\n Enter Your Choice: ";
            cin>choice;
            switch(choice)
            {
             case 1:    cout<"\n Enter the item to be pushed ";
                        cin>item;
                        if(obj.stfull())
                        cout<"\n Stack is Full!";
                  else
                        obj.push(item);
                        break;
             case 2:if(obj.stempty())
                        cout<"\n Empty stack!Underflow !!";
                  else
                  {
                   item=obj.pop();
                   cout<"\n The popped element is "<item;
                  }
                  break;
             case 3:obj.display();
                        break;
             case 4:exit(0);
            }
            cout<"\n Do You want To Continue?";
            ans=getche();
      }while(ans ==Y ||ans ==y);
      getch();
}
/******************** End Of Program *************************/
```

Output

```
Implementation Of Stack
Main Menu
1.Push
2.Pop
3.Display
```

```
4.exit
 Enter Your Choice: 1

 Enter the item to be pushed 10

Do You want To Continue?y
 Main Menu
1.Push
2.Pop
3.Display
4.exit
 Enter Your Choice: 1

 Enter the item to be pushed 20

Do You want To Continue?y
 Main Menu
1.Push
2.Pop
3.Display
4.exit
 Enter Your Choice: 1

 Enter the item to be pushed 30

 Do You want To Continue?y
 Main Menu
1.Push
2.Pop
3.Display
4.exit
 Enter Your Choice: 2
 The popped element is 30
 Do You want To Continue?y
 Main Menu
1.Push
2.Pop
3.Display
4.exit
 Enter Your Choice: 3
20
10
Do You  want To Continue?y
```

Review Question

1. *Explain the ADT operations for array implementation stack.*

3.4 Implementation of Stack using Linked Organization

SPPU : May-15, Dec.-10,15, Marks 6

Stack is a special case of list and therefore we can represent stack using arrays as well as using linked list. The advantage of implementing stack using linked list is that we need not have to worry about the size of the stack. Since we are using linked list as many elements we want to insert those many nodes can be created. And the nodes are dynamically getting created so there won't be any stack full condition.

The typical structure for linked stack can be

```
struct stack
{
    int data ;
    struct stack * next ;
}node ;
```

Each node consists of data and the next field. Such a node will be inserted in the stack. Following Fig. 3.4.1 represents stack using linked list.

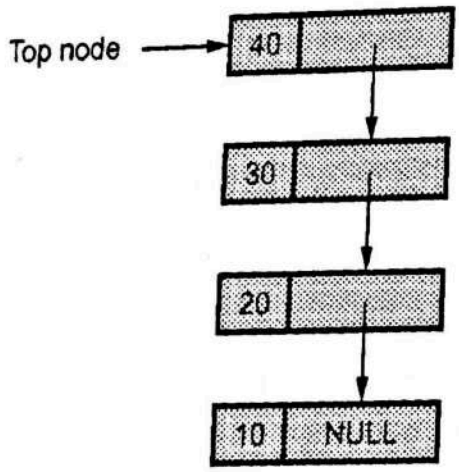


Fig. 3.4.1 Representing linked stack

There are various operations that can be performed on the linked stack. These operations are -

1. Push
2. Pop
3. Stack empty
4. Stack full

1. Push operation

The pseudo code for push operation is as given below.

```
void Lstack::Push(int Item, node **top)
{
    node *New;
    New = new node;
    New->next=NULL;
    New->data=Item;
    New-> next = *top;
    *top = New;
}
```

Using new operator the memory can be allocated.
For example :

10	NULL

New/top

In the main function, when the item to be inserted is entered, a call to push function is given.

New

10	NULL

If item = 10, then by new node will be allocated. Then

```
*top = New
```

This is our new **top** node. Hence stack will look like this

New

top

10	NULL

If again **Push** function is called for pushing the value 20 then

New

20	NULL

top

10	NULL

This can be done using following statements -

```
New->next=*top;
top=New;
```

And we will get

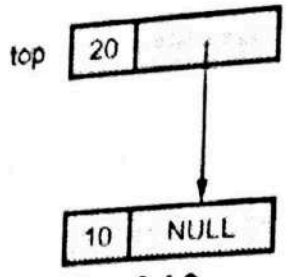


Fig. 3.4.2

When 30 is pushed then

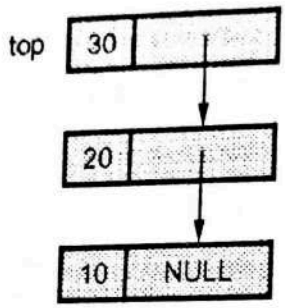


Fig. 3.4.3

Thus by pushing the items repeatedly we can create a stack using linked list.

2. Pop operation

```
int Lstack::Pop(node **top)
{
  int item;
  node *temp;
  item = (*top) ->data ;
  temp = *top;
  *top = (*top) -> next ;
  delete temp;
  return (item);
}
```

We want to delete top node. Hence in order to remember the data being deleted it is already stored in item

Taking the address of **top** in **temp** node and assigning new **top** to the next node.

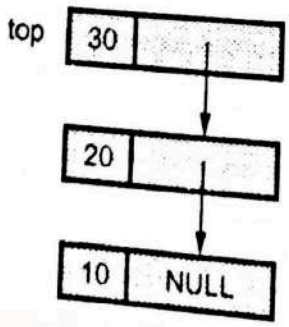


Fig. 3.4.4

If stack is as follows.

If we write

item = *(top)->data	then item=30
temp=*top	
*top=*top->next	now top =20
delete temp;	deallocating memory of node 30. That means node 30 is deleted.

Let us now see the C++ implementation of it.

C++ Program

```
/*****************************************************************
Program for creating the stack using the linked list.
Program performs all the operations such as push,pop and
display.It takes care of stack underflow condition. There can
   not be stack full condition in stack using linked list.
*****************************************************************/
#include <iostream.>
#include <stdlib.h>
using namespace std;
//Declaration for data structure of linked stack
class Lstack
{
 private:
 typedef struct stack
   {
        int data;
       struct stack *next;
    }node:
    node *top;
  public:
    Lstack();
    ~Lstack();
    void create(),remove(),show();
    void Push(int , node **);
    void Display(node **);
    int Pop(node **);
    int Sempty(node *);
};
/*
*****************************************************************
The constructor defined
*****************************************************************
*/
Lstack::Lstack()
{
```

```
  top = NULL;
}
/*
-----------------------------------------------------------------------
The destructor defined
-----------------------------------------------------------------------
*/
Lstack::~Lstack()
{
 node *temp;
 temp=top;
 if(temp==NULL)
          delete temp;
 else
 {
  while(temp!=NULL)
  {
          temp=temp ->next;
          top=NULL;
          top=temp;
    }
   delete temp;//de allocating memory
   }
}
/*
-----------------------------------------------------------------------
The create function
-----------------------------------------------------------------------
*/
void Lstack::create()
{
 int data;
 cout<<"\n Enter the data";
 cin>>data;
 Push(data,&top);
}
/*
-----------------------------------------------------------------------
The remove function
-----------------------------------------------------------------------
*/
void Lstack::remove()
{
    int item;
    if(Sempty(top))
          cout<<"\n stack underflow!";
    else
```

```
{
            item = Pop(&top);
            cout<<"\n The popped node is  "<< item;
        }
  }
  /*
  ---------------------------------------------------------------------
  The show function
  ---------------------------------------------------------------------
  */
  void Lstack::show()
  {
     Display(&top);
  }

  /*
  ---------------------------------------------------------------------
  The Push function
  ---------------------------------------------------------------------
  */
  void Lstack::Push(int Item, node **top)
  {
     node *New;
     New = new node;
     New->next=NULL;
     New->data=Item;
     New -> next = *top;
  *top = New;
  }
  /*
  ---------------------------------------------------------------------
  The Sempty Function
  ---------------------------------------------------------------------
  */
  int Lstack::Sempty(node *temp)
  {
  if(temp==NULL)
   return 1;
  else
      return 0;
  }
  /*
  ---------------------------------------------------------------------
  The Pop function
  ---------------------------------------------------------------------
  */
  int Lstack::Pop(node **top)
```

```
{
  int item;
  node *temp;
  item = (*top) ->data;
  temp = *top;
  *top = (*top) -> next;
  delete temp;
  return(item);
}
/*
-----------------------------------------------------------------
The Display function
-----------------------------------------------------------------
*/
int Lstack::Display(node **head )
{
  node *temp ;
  temp = *head;
  if(Sempty(temp))
     cout<<"\n The stack is empty!";
  else
  {
    while ( temp != NULL )
    {
      cout<<"  "<<temp -> data;
      temp = temp -> next;
    }
  }

}
/*
-----------------------------------------------------------------
The main function
-----------------------------------------------------------------
*/
int main ( )
{
        int choice;
        char ans,ch;
        Lstack st;
        cout<<"\n\t\t Stack Using Linked List";
        do
        {
         cout<<"\n\n The main menu";
         cout<<"\n1.Push\n2.Pop\n3.Display\n4.Exit";
         cout<<"\n Enter Your Choice";
         cin>>choice;
```

```
        switch(choice)
        {
         case 1:st.create();
                    break;
         case 2:st.remove();
                    break;
         case 3:st.show();
                    break;
         case 4:exit(0);
        }
        cout<<"\n Do you want to continue?";
        cin>>ans;
   }while(ans ==='Y' ||ans ==='y');
return 0
}
```

Output

```
The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice1

Enter the data10

Do you want to continue?
The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice1

Enter the data20

Do you want to continue?
The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice1

Enter the data30
Do you want to continue?
The main menu
```

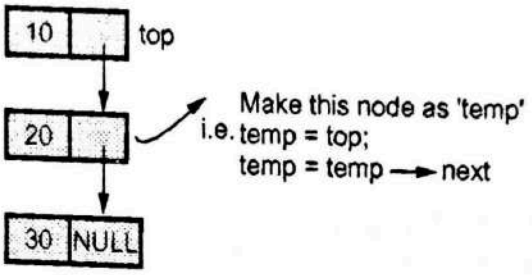
```
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice1

Enter the data40

Do you want to continue?
The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice1

Enter the data50

Do you want to continue?


The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice3
 50  40  30  20  10
Do you want to continue?

The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice2

The popped node is  50
Do you want to continue?

The main menu
1.Push
2.Pop
3.Display
4.Exit
Enter Your Choice4
```

We have defined constructor for stack in which simply top is intialized by NULL value.

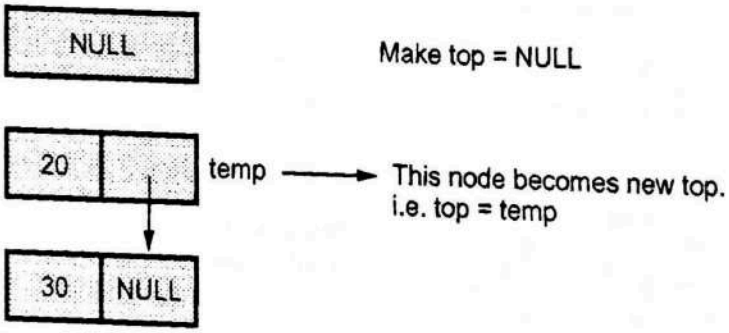
```
LStack :: LStack()
{
top = NULL ;
}
```

And in the destructor we are deallocating the memory which was reserved for each node of stack.

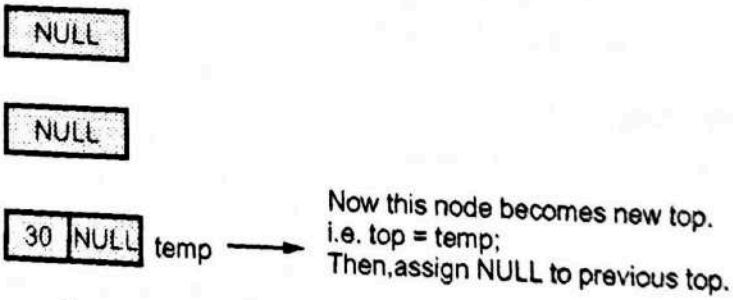
For example : If the linked stack is created as follows -



Continuing in while loop.



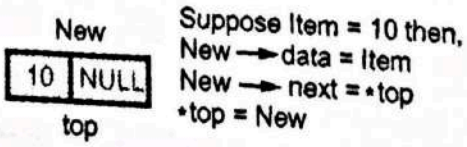
Hence,



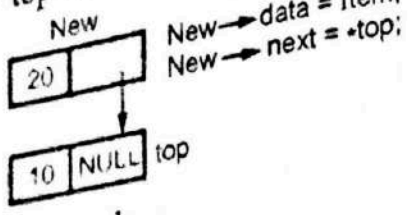
This shows that all the nodes are assigned NULL. Then the memory for these nodes is deallocated by 'delete'.

In push function : The memory for 'New ' node is allocated using new operator. In 'New ' node we will put the data.

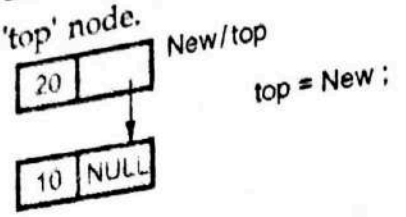
Initially top = NULL, hence



Now make 'New ' node as 'top' node. When we push next item as 20 then,

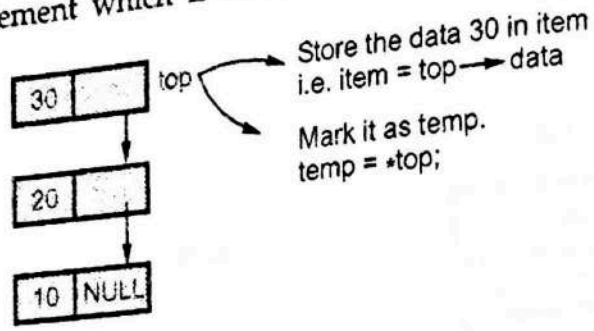


Now make 'New ' node as 'top' node.

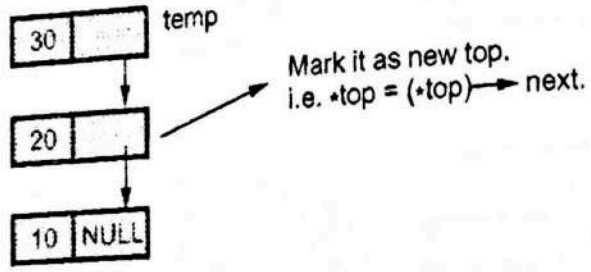


In POP function

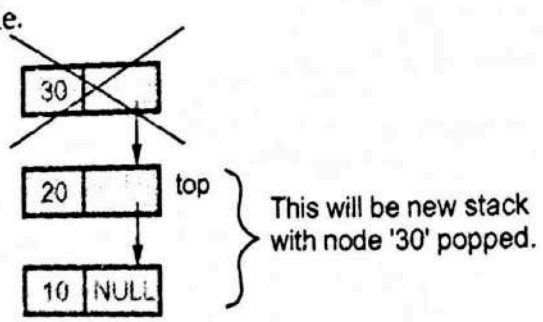
We will POP the element which is at the top. Consider a scenario that we have a stack of



Then



Then delete 'temp' node.



Review Questions

1. *Implement push and pop operations for stack using linked organization.* **SPPU : Dec.-10, Marks 4**
2. *Implement stack as an ADT using linked organization.* **SPPU : May-15, Marks 6**
3. *What is stack ? write a program for implementation of stack using linked organization and perform i) PUSH and ii) POP operations* **SPPU : Dec.-15, Marks 6**

3.5 Concept of Implicit and Explicit Stack

SPPU : Dec.-11,12; May-11,12,13,14; Marks 8

When we apply recursion an **implicit stacks** is used. Implicit stack is a special kind of stack used by **runtime memory environment.** This stack is used to hold the information about function CALL. This type of stack is used implicitly by the compiler to support function CALL and RETURN from the function. This stack cannot be directly manipulated by the programmer. Following program makes use of explicit stack.

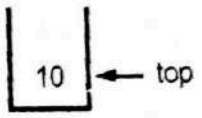
```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
        int fun(int);
        cout<<"\n Use of Implicit stack";
        int a=10;
        fun(a);
        return 0;
}
int fun(int a)
{
 if(a>0)
 {
        fun(a-1);
        cout<<" " <<a;
 }
 else
        return a;
}
```

Output

Use of Implicit stack 1 2 3 4 5 6 7 8 9 10

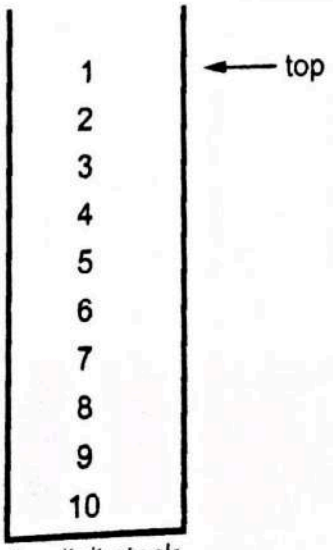
(I) When **main ( )** CALLs fun (a) implicit stack is invoked and value of a = 10 is pushed onto the stack.

∴



implicit stack.

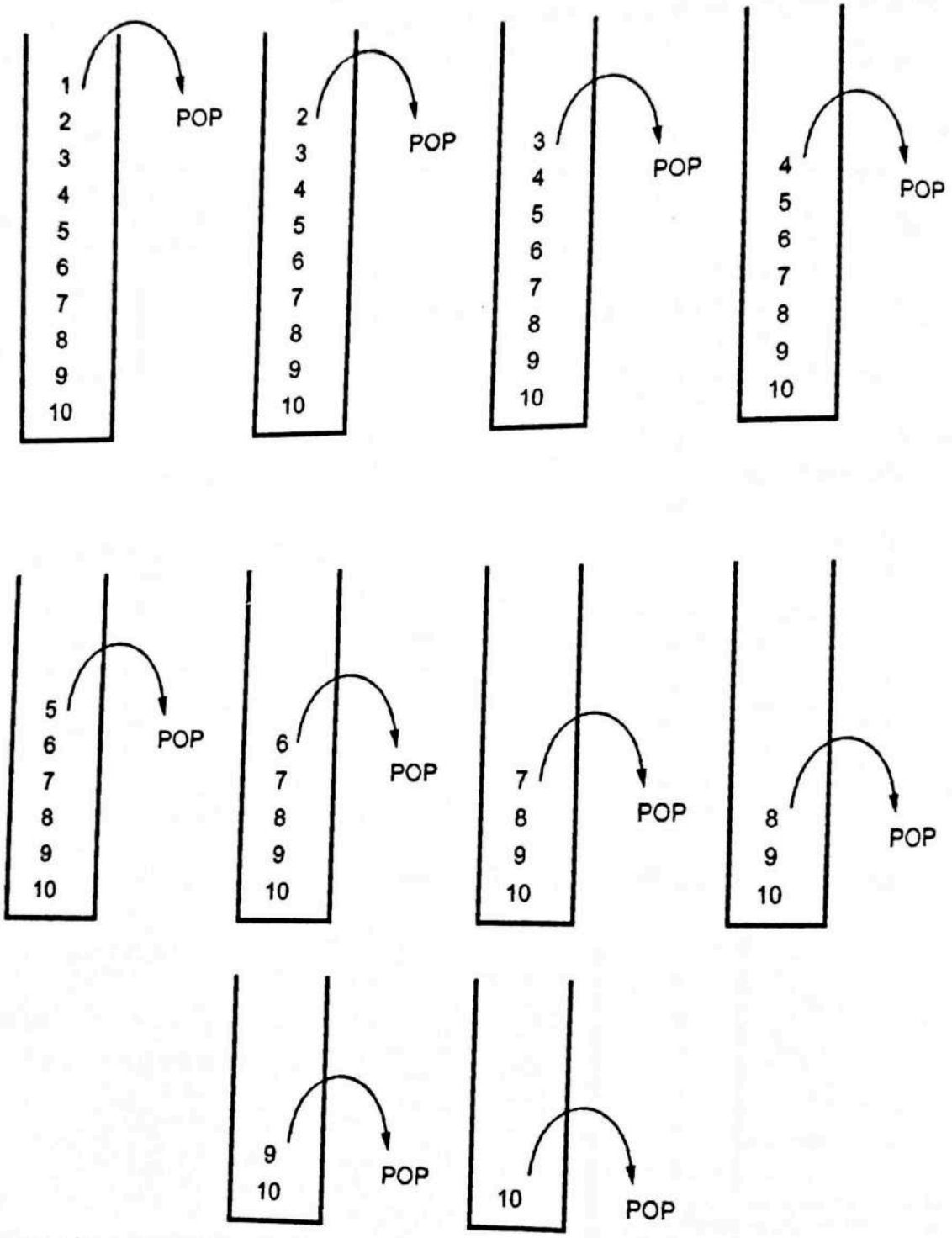
(II) As a = 10 > 0 is true, when a recursive call to **fun(a – 1)** is given a – 1 = 9 is pushed onto the stack. This happens repeatedly when a > 0. The implicit stack will then be



implicit stack

(III) When value of a = 0 the **else** part gets executed

(IV) Now, the control of the program reaches to **cout** statement. This is the time when recursive function wants to return to the **main( )**. At this instance, the stack is popped off and a = 1 gets printed. The control then reaches to the } bracket at the end. Thus control switches between **cout** and } and each time stack is popped off. This repetition occurs because all the pending call are in stack.



(V) Finally since all the pending or incomplete calls are getting fulfilled, the control returns back the **main( )** function.

The implicit stack is mainly used to store data that is local to a function. Space for local data items is allocated from the stack during the execution of that function, and is de-allocated when the function exits.

Explicit stack

The above program can be shown below with **explicit usage of stack -**

```
#include<iostream>
  using namespace std ;
  int stack[10],top;  <------------------  Explicit stack is maintained
  int main()
  {

    void fun(int);
    cout<<"\n Use of Explicit stack";
    top=-1;
    int a=10;
    fun(a);
    return 0;
  }
  void push(int item)
  {
   top++;
   stack[top]=item;
  }
  int pop()
  {
  int item;
  item=stack[top];
  top--;
  return item;
  }
```

```
int stempty()
  {
  if(top==-1)
    return 1;
  else
    return 0;
  }
  void fun(int a)
  {
   do
   {
   if(a>0)
   {
     push(a);
     a--;
   }

   else
     cout<<"     " << pop();
   }while(!stempty());

  }
```

Each CALL gets pushed onto the stack.

Recursion is managed by do_while statement

Each time value of a must be explicitly changed.

On each RETURN stack is popped OFF

Output

Use of Explicit stack 1 2 3 4 5 6 7 8 9 10

Example 3.5.1 *Consider the program*

```
int fact (int n)
{
int x, y;
if (n == 0)
      return (1);
x = n - 1;
y = fact(x);
return (n * y);
}
```

SPPU : Dec.-11, Marks 8

Solution : Suppose a calling program contains the following statement
print ("%d", fact(4));

Then we get -

Display the contents of the stack for the variables n, x and y as execution of the fact function proceeds.

Sr. No.	Input	Output
1.	n = 4	$4 \times 3 \times 2 \times 1$
		= 24
2.	x = 3	
	y = Fact (3)	3×1
	n = 3	
3.	x = 2	
	y = Fact (2)	2×1
	n = 2	
4.	x = 1	
	y = Fact (1)	1×1
	n = 1	
5.	x = 0	n = 0
	y = Fact (0)	1
	Now n = 0	
	Hence return	1

Review Questions

1. *Explain the concept of implicit and explicit stack and give its importance.*
   SPPU : May-11, Dec.-12, Marks 4; May-14, Marks 2
2. *What is importance of stack in recursion ? Explain with suitable example.*
   SPPU : May-12, May-13, Marks 4
3. *Explain the concept of stack. What is implicit and explicit stack ? Explain both of them with suitable example.*
   SPPU : Dec.-19, Marks 6

3.6 Applications of Stack

Various applications of stack are

1. Expression conversion
2. Expression evaluation
3. Parsing well formed parenthesis
4. Decimal to binary conversion
5. Reversing a string
6. Storing function calls.

3.7 Expression Conversion

SPPU : May-13,14,15,16, Dec.-11,14,15 Marks 8

There are three types of expressions :

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

One of the application of stack is conversion of expression. First of all, let us see these expressions with the help of examples.

1. Infix Expression :

In this type of expressions the arrangement of operands and operator is as follows.

Infix expression = operand1 operator operand2

For example
1. (a+b)
2. (a+b) * (c – d)
3. (a+b/e) * (d+f)

Parenthesis can be used in these expressions infix expression are the most natural way of representing the expressions.

2. Postfix Expression :

In this type of expressions the arrangement of operands and operator is as follows

Postfix expression = operand1 operand operator

For example
1. ab+
2. ab + cd – *
3. ab + e / df + *

In postfix expression there is no parenthesis used. All the corresponding operands come first and then operator can be placed.

3. Prefix Expression :

In prefix expression the arrangement of operands and operators is as follows.

Prefix expression = operator operand1 operand2

For example
1. + ab
2. * + ab – cd
3. * / + abe + df

In prefix expression, there is no parenthesis used. All the corresponding operators come first and then operands are arranged.

3.7.1 Conversion from Infix to Postfix Expression

The most important thing in the conversion procedure is to know the priorities of the operators being used in it. This helps in determining the order of evaluation of operands.

Operator	Precedence in stack	Precedence in expression
^	3	4
*, /	2	2
+, –	1	1
(	0	4

There are certain rules which has to be followed while coverting infix expression to postfix form :

1. The expression is to be read from left to right.
2. Read one character at a time from infix expression.
3. Make use of stack to store the operators.
4. There should not be any parenthesis in the postfix form.

Let us see an algorithm for it.

Algorithm for conversion of infix to postfix form :

1. Read the expression from left to right.
2. If the input symbol read is '(' then push it onto the stack.
3. If the input symbol read is an operand then place it in postfix expression.

4. If the input symbol read is an operator then

a) Check if the precedence of the operator which is in the stack has greater precedence than the precedence of the operator read, if so then remove that symbol from stack and place it in the postfix expression. Repeat step 4(a) till you get the operator in the stack has greater precedence than the operator being read.

b) Otherwise push the operator being read onto the stack.

5. If the input symbol read is a closing parenthesis ')' then pop all the operators from the stack, place them in postfix expression till the opening parenthesis is not popped. The '(' should not be place in the postfix expression.

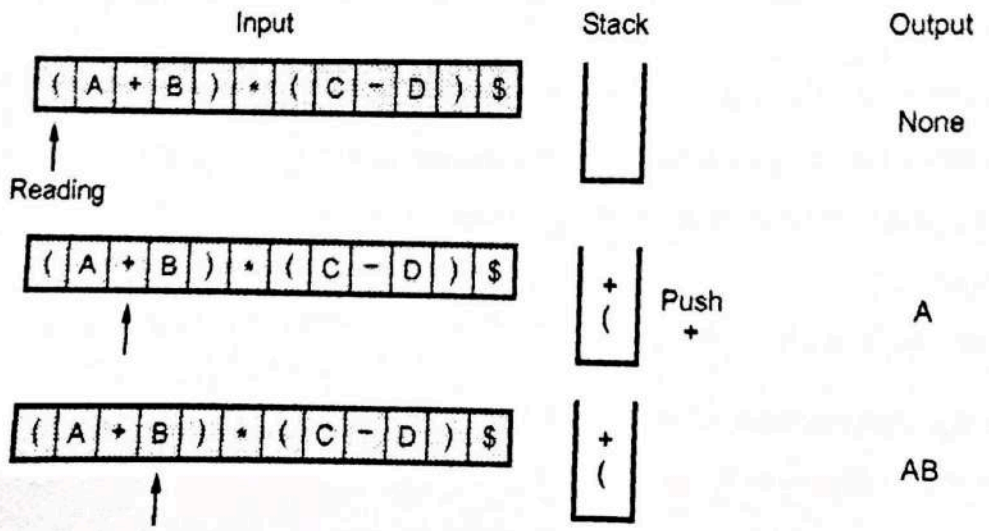
6. Finally print the postfix expression.

Let us see the procedure with some examples.

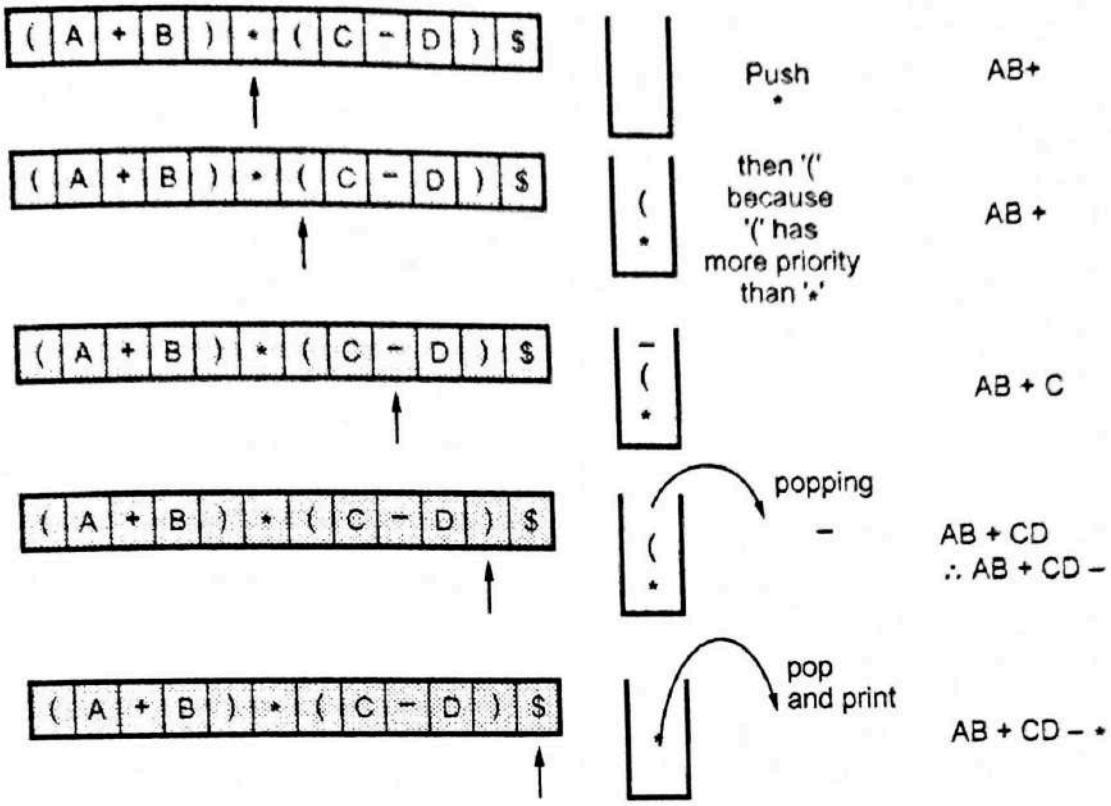
Example : A * B + C $

Input	Stack	Output
None	Empty	none
A	Empty	A
*	*	A
B	*	AB
+	Pop * and Push	AB*
C	+	AB*C
$	Empty	AB*C +

Example :



When closing parenthesis comes pop everything and print except '('.



So finally the converted expression is

AB+CD-*

C++ Program

```
/*****************************************************************
 Program for conversion of Infix expression to Postfix form.
*****************************************************************/
#include<iostream>
using namespace std ;
class EXP
{
 private:
 char post[40];
 int top,st[20];
 public:
 EXP();
 void postfix(char inf[40]);
 void push(int);
 char pop();
 };
```

```
EXP::EXP()
{
top=0;
}
/*
-----------------------------------------------
Postfix function
-----------------------------------------------
*/
void EXP::postfix(char inf[40])
 {
   int i,j=0;
   for(i=0;inf[i]!='\0';i++)
   {
   switch(inf[i])
   {
        case '+' :  while(st[top] >= 1)
                     post[j++] = pop();
                    push(1);
                    break;

        case '-' ;  while(st[top] >= 1)
                    post[j++] = pop();
                    push(2);
                    break;
        case '*' ;  while(st[top] >= 3)
                    post[j++] = pop();
                    push(3);
                    break;

        case '/' :  while(st[top] >= 3)
                    post[j++] = pop();
                    push(4);
                    break;

        case '^' :  while(st[top] >= 4)
                    post[j++] = pop();
                    push(5);
                    break;
```

```
            case '(' :   push(0);
                         break;

            case ')' :   while(st[top] != 0)
                         post[j++] = pop();
                         top--;
                         break;

            default  : post[j++] = inf[i];
      }
      }
      while(top>0)
      post[j++] = pop();
cout<<"\n\tPostfix expression is =>\n\n\t\t "<<post;
   }
   /*
   ----------------------------------------
   Push function
   ----------------------------------------
   */
   void EXP::push(int ele)
   {
      top++;
      st[top] = ele;
   }
   /*
   ----------------------------------------
   pop Function
   ----------------------------------------
   */
   char EXP::pop()
   {
      int el;
      char e;
      el = st[top];
      top--;
      switch(el)
      {
            case 1 : e = '+';
```

```
            break;
             case 2 : e = '-';
            break;
             case 3 : e = '*';
            break;
             case 4 : e = '/';
            break;
             case 5 : e = '^';
            break;
    }
    return(e);
  }

 /*
The main Function
          Input:none
   Output:none
   Called By:O.S.
   Calls:postfix

 */
 int main(void)
 {
     EXP obj;
     char inf[40];
     cout<<"\n\tEnter the infix expression :: \n";
     cin>>inf;
     cout<<"The infox expression entered by you is..."<<inf;
     obj.postfix(inf);
     return 0
  }
```

Output

```
Enter the infix expression ::
(a+b)*(c-d)
The infox expression entered by you is...(a+b)*(c-d)
        Postfix expression is =>

                ab+cd-*
```

Example 3.7.1 *Convert the following infix expression into prefix and postfix notations.*

Solution : $\underbrace{a*b}-c-d+e*f-g/h*i$ $\qquad t_1 = ab*$

$t_1 - c - d + \underbrace{e*f} - g/h*i$ $\qquad t_2 = ef*$

$t_1 - c - d + t_2 - \underbrace{g/h}*i$ $\qquad t_3 = gh/$

$t_1 - c - d + t_2 - t_3 * i$	$t_4 = t_3 i *$
$t_1 - c - d + t_2 - t_4$	$t_5 = t_1 c -$
$t_5 - d + t_2 - t_4$	$t_6 = t_5 d -$
$t_6 + t_2 - t_4$	$t_7 = t_6 t_2 +$
$t_7 - t_4$	$t_8 = t_7 t_4 -$
t_8	

Now replacing backaward we get,

t_8

$t_7\ t_4 -$

$t_6\ t_2 + t_4 -$

$t_5 d - t_2 + t_4 -$

$t_1\ c - d - t_2 + t_4 -$

$ab * c - d - t_2 + t_4 -$

$ab * c - d - ef * + t_4 -$

$ab * c - d - ef * + t_3\ i * -$

$\boxed{ab * c - d - ef * + gh/\ i * -}$ is required postfix expression

Now for converting infix expression to prefix,

Consider

$a * b - c - d - e * f - g/h * i$	$t_1 = * ab$
$t_1 - c - d + e * f - g/h * i$	$t_2 = * ef$
$t_1 - c - d + t_2 - g/h * i$	$t_3 = /gh$
$t_1 - c - d + t_2 - t_3 * i$	$t_4 = * t_3 i$
$t_1 - c - d + t_2 - t_4$	$t_5 = - t_1 c$
$t_5 - d + t_2 - t_4$	$t_6 = - t_5 d$
$t_6 + t_2 - t_4$	$t_7 = + t_6\ t_2$
$t_7 - t_4$	$t_8 = - t_7\ t_4$

Now replacing the temporary variables in backward direction,

t_8

$- t_7 t_4$

$- + t_6\ t_2\ t_4$

$- + -\ t_5 d\ t_2\ t_4$

$- + - - t_1\ c\ d\ t_2\ t_4$

$- + - - *\ abcd\ t_2\ t_4$

$- + - - *\ abcd * ef\ t_4$

$- + - - *\ abcd * ef * t_3\ i$

$- + - - *\ abcd * ef * /\ gh\ i$ is required prefix expression

Example 3.7.2 *Clearly indicate the contents of stack during conversion of given infix to prefix expression. Consider ^ as exponent operator.*

$(((A + B) * C - (D - E)) \wedge (F + G))$ **SPPU : May-14, Marks 6**

Solution :

Step 1 : Reverse the infix expression

)) G + F(^)) E – D(–C *) B + A (((

Step 2 : Make every) and (as (and) respectively. The expression becomes

((G + F) ^ ((E – D) – C * (B + A)))

Step 3 : Convert the expression to postfix form.

Expression	Current Symbol	Action Taken	Stack	Output
((G+F)^((E–D)–C*(B+A)))	(	Push	(	
((G+F)^((E–D)–C*(B+A)))	(	Push	((	
G+F)^((E–D–C*(B+A)))	G	Print	((	G
+F)^((E–D–C*(B+A)))	+	Push	((+	G
F)^((E–D–C*(B+A)))	F	Print	((+	GF
)^((E–D–C*(B+A)))	)	POP+, Print, POP (	(	GF +
^((E–D–C*(B+A)))	^	Push ^	(^	GF +
((E–D–C*(B+A)))	(	Push (	(^(	GF +

(E-D-C*(B+A)))	(	Push (	(^ ((	GF +
E-D-C*(B+A)))	E	Print E	(^ ((	GF + E
-D-C*(B+A)))	-	Push -	(^ ((-	GF + E
D-C*(B+A)))	D	Print D	(^ ((-	GF + ED
-C*(B+A)))	-	Pop - Print. Push -	(^ ((-	GF + ED -
C*(B+A)))	C	Print C	(^ ((-	GF + ED - C
*(B+A)))	*	Push *	(^ ((- *	GF + ED - C
(B+A)))	(	Push (	(^ ((- * (	GF + ED - C
B+A)))	B	Print B	(^ ((- * (	GF + ED - CB
+A)))	+	Push +	(^ ((- * (+	GF + ED - CB
A)))	A	Print A	(^ ((- * (+	GF + ED - CBA
)))	)	POP +, Print POP (	(^ ((- *	GF + ED - CBA +
))	)	POP *, Print. POP - Print.	(^ ((	GF + ED - CBA + * -
)	)	POP ^, Print POP (	empty	GF + ED - CBA + * - ^

Now reverse the obtained postfix expression.

^ - * + ABC - DE + FG is required prefix expression.

Example 3.7.3 *Change the following infix to postfix using stack. Clearly indicate the content of stack :*

*i) (A + B) * C - D * F + C.*

*ii) (A - 2) * (B + C - D * E) * F.*

SPPU : May-15, Marks 6

Solution :

Input Read	Action	Stack	Output
(A + B) * C – D * F + C	Push (	(	
(A + B) * C – D * F + C	Print A	(	A
(A + B) * C – D * F + C	Push +	(+	A
(A + B) * C – D * F + C	Print B	(+	AB
(A + B) * C – D * F + C	POP + , Print, POP (		AB+
(A + B) * C – D * F + C	Push *	*	AB+
(A + B) * C – D * F + C	Print (	*	AB + C
(A + B) * C – D * F + C	POP *, Print then push –	–	AB + C *
(A + B) * C – D * F + C	Print D	–	AB + C * D
(A + B) * C – D * F + C	Push *	– *	AB + C * D
(A + B) * C – D * F + C	Print F	– *	AB + C * DF
(A + B) * C – D * F + C	POP *, Print. POP –, Print push +	+	AB + C * DF * –
(A + B) * C – D * F + C	Print C	+	AB + C * DF * – C
(A + B) * C – D * F + C	POP +, Print.		AB + C * DF * – C +

The postfix expression is **AB + C * DF * – C +**

Input Read	Action	Stack	Output
(A – 2) * (B + C – D * E) * F	Push (	(	
(A – 2) * (B + C – D * E) * F	Print A	(	A

(A – 2) * (B + C – D * E) * F	Push –	(–	A
(A – 2) * (B + C – D * E) * F	Print 2	(–	A2
(A – 2) * (B + C – D * E) * F	POP –, Print, POP (		A2 –
(A – 2) * (B + C – D * E) * F	Push *	*	A2 –
(A – 2) * (B + C – D * E) * F	Push (	*(	A2 –
(A – 2) * (B + C – D * E) * F	Print B	*(	A2 – B
(A – 2) * (B + C – D * E) * F	Push +	*(+	A2 – B
(A – 2) * (B + C – D * E) * F	Print (	*(+	A2 – BC
(A – 2) * (B + C – D * E) * F	POP +, Print Push –	*(–	A2 – BC +
(A – 2) * (B + C – D * E) * F	Print D	*(–	A2 – BC + D
(A – 2) * (B + C – D * E) * F	Push *	*(– *	A2 – BC + D
(A – 2) * (B + C – D * E) * F	Print E	*(– *	A2 – BC + DE
(A – 2) * (B + C – D * E) * F	POP *, Print. POP –, Print POP (	*	A2 – BC + DE * –
(A – 2) * (B + C – D * E) * F	POP *, Print. Push *	*	A2 – BC + DE * – *
(A – 2) * (B + C – D * E) * F	Print F	*	A2 – BC + DE * – * F
(A – 2) * (B + C – D * E) * F ↑	POP *, print		A2 – BC + DE * – * F *

The postfix expression is **A2 – BC + DE * – * F ***

Example 3.7.4 *Transform the following expression to prefix and postfix*

*A + (((B – C) * (D – E) +F) / G) $ (H – J). Consider $ as a raised operator.*

SPPU : Dec.-15, Marks 6

Solution : Conversion from Infix to Prefix

A + (((B – C) * (D – E) + F) / G) $ (H – J)	
A + ((T1 * (D – E) + F) / G) $ (H – J),	T1 = –BC
A + ((T1 * T2 + F) / G) $ (H – J)	T2 = –DE
A + ((T3 + F) / G) $ (H – J)	T3 = * T1 T2
A + (T4 / G) $ (H – J)	T4 = + T3 F
A + T5 $ (H – J)	T5 = / T4 G
A + T5 $ T6	T6 = – HJ
A + T7	T7 = $ T5 T6
T8	T8 = + AT 7

By backward substitution we get

T8

+ A T 7

+ A $ T5 T6

+ A $ / T4 G T6

+ A $ / + T3 FG T6

+ A $ / + * T1 T2 FG T6

+ A $ / + * –BC T2 FG T6

+ A $ / + * – BC – DE FG T6

+ A $ / + * –BC – DE FG – HJ

This required prefix expression.

Infix to Postfix

A + (((B – C) * (D – E) + F) / G) $ (H – J)	
A + ((T1 * (D – E) + F) / G) $ (H – J)	T1 = BC –
A + ((T * T2 + F) / G) $ (H – J)	T2 = DE –
A + ((T3 + F) / G) $ (H – J)	T3 = T1 T2 *
A + (T4 / G) $ (H – J)	T4 = T3 F +
A + T5 $ (H – J)	T5 = T4 G /
A + T5 $ T6	T6 = HJ –
A + T7	T7 = T5 T6 $
T8	T8 = AT 7 +

By backward substitution we get,

T8

A T7 +

A T5 T6 $ +

A T4 G / T6 $ +

A T3 F + G / T6 $ +

A T1 T2 * F + G / T6 $ +

ABC – T2 * F + G / T6 $ +

ABC – DE – * F + G / T6 $ +

ABC – DE – * F + G / HJ – $ +

This is required post fix expression.

Example 3.7.5 *Clearly indicate the contents of stack during conversion of given infix expression to prefix : Infix expression : A + (B * ((D – E % F) / H))*

SPPU : May-16, Marks 6

Solution : Step 1 : Reverse the given infix expression.

)) H /) F % E – D ((* B (+ A

Step 2 : Make (as), and) as (

((H / (F % E – D)) * B) + A

Step 3 : Now convert this expression to postfix.

Expression	Current Symbol	Action Taken	Stack	Output
((H/(F%E–D))*B)+A	(	Push (	(	
(H/F%E–D))*B)+A	(	Push (	((	
H/(F%E–D))*B)+A	H	Print H	((	H
/(F%E–D))*B)+A	/	Push /	((/	H
(F%E–D))*B)+A	(	Push (	((/(	H
F%E–D))*B)+A	F	Print F	((/(	HF

%E – D))* B) + A	%	Push %	((/ (%	HF
E– D))* B) + A	E	Print E	((/ (%	HFE
– D))* B) + A	–	POP %, Print then Push –	((/ (–	HFE %
D))* B) + A	D	Print D	((/ (–	HFE % D
)) * B) + A	)	POP –, Print POP (	((/	HFE % D –
)* B) + A	)	POP /, Print. POP (	(	HFE % D – /
*B) + A	*	Push *	(*	HFE % D – /
B) + A	B	Print B	(*	HFE % D – / B
) + A	)	Pop *, Print . POP (	empty	HFE % D – / B *
+ A	+	Push +	+	HFE % D – / B *
A	A	Print A	+	HFE % D – / B * A
End of Input		POP + Print	empty	HFE % D – / B * A +

Now reverse the obtained postfix expression.

The + A * B / – D % EFH be the prefix expression.

Example 3.7.6 *Convert the following expression from infix to Postfix and Prefix. Make use of appropriate data structure which can be used for conversion.*

*a. 2 * 3 / (2 – 1) + 5 * 3*

SPPU : Dec.-18, Marks 6

Solution : Infix to Postfix conversion

Input character read	action	stack	Output
2	Print 2		2
*	Push	*	2
3	Print 3	*	23
/	Pop *, print, push	/	23*
(	Push (	/ (	23*
2	Print 2	/ (	23*2
–	Push –	/ (–	23*2
1	Print	/ (–	23*21
+	POP–, Print POP (POP /, Print Push +	+	23*21 – 1
5	Print 5	+	23*21 – /5
*	Push *	+*	23*21 – /5
3	Print 3	+*	23*21 – /53
End of input	POP *, Print POP +, Print		23*21 – /53*+

The required Postfix expression is,

23 * 21 – / 53 * +

Inflix to Prefix conversion

Reverse given expression

3*5+) 1–2 (/3*2

Now make) as (and (as)

3*5+(1–2)/3*2

Now convert this expression to Postfix form.

Input character read	action	stack	Output
3	Print		3
*	Push *	*	3
5	Print	*	35
+	POP *, Print, Push +	+	35*

(	Push (	+ (	35*
1	Print 1	+ (	35*1
–	Push	+ (–	35*1
2	Print 2	+ (–	35*12
)	POP –, Print POP (	+	35*12 –
/	Push /	+ /	35*12 –
3	Print 3	+ /	35*12 – 3
*	POP /, Print Push *	+ *	35*12 – 3 /
2	Print 2	+ *	35*12 – 3 / 2
End of input	POP *, Print POP +, Print	empty	35*12 – 3 / 2* +

Now reverse the expression obtained.

+ *2/3–21 *53

This is required prefix expression.

Example 3.7.7 *Convert the following expression from Infix to Postfix and Prefix. Make use of appropriate data structure : 2 * 3/(2 – 1) + 5 * 3* **SPPU May-19, Marks 6**

Solution : Infix and Postfix

Input Read	Action	Stack	Output
2 * 3 / (2 – 1) + 5 * 3 ↑	Print 2	Empty	2
2 * 3 / (2 – 1) + 5 * 3 ↑	Push *	*	
2 * 3 / (2 – 1) + 5 * 3 ↑	Print 3	*	2 3
2 * 3 / (2 – 1) + 5 * 3 ↑	POP *, Print then push /	/	23 *
2 * 3 /(2 – 1) + 5 * 3 ↑	Push (	/ (	
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 2	/ (	23 * 2

2 * 3 /(2 – 1) + 5 * 3 ↑	Push –	/ (–	23 * 2
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 1	/ (–	23 * 21
2 * 3 /(2 – 1) + 5 * 3 ↑	POP –, Print, POP (	/	23 * 21 –
2 * 3 /(2 – 1) + 5 * 3 ↑	POP /, Print, Push +	+	23 * 21 – /
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 5	+	23 * 21 – / 5
2 * 3 /(2 – 1) + 5 * 3 ↑	Push *	+ *	23 * 21 – / 5
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 3	+ *	23 * 21 – / 5 3
2 * 3 /(2 – 1) + 5 * 3 ↑	POP *, Print POP +, Print	Empty	23 * 21 – / 53*+

Infix to prefix

Step 1 : Reverse the given infix expression 3*5 +)1 – 2 (/3*2

Step 2 : Make (as) and) as (3*5 + (1 – 2)/3*2.

Step 3 : Now convert this expression to postfix.

Input Read	Action	Stack	Output
3*5+(1 – 2)/3*2 ↑	Print 3	Empty	3
3*5+(1 – 2)/3*2 ↑	Push	*	3
3*5+(1 – 2)/3*2 ↑	Print 5	*	3 5
3*5+(1 – 2)/3*2 ↑	POP *, Print Push +	+	3 5 *
3*5+(1 – 2)/3*2 ↑	Push (	+ (	3 5 *

3*5+(1 – 2)/3*2 ↑	Print 1	+ (	3 5 * 1
3*5+(1 – 2)/3*2 ↑	Push –	+ (–	3 5 * 1
3*5+(1 – 2)/3*2 ↑	Print 2	+ (–	3 5 * 1 2
3*5+(1 – 2)/3*2 ↑	POP –, Print	+	35 *12 –
3*5+(1 – 2)/3*2 ↑	Push /	+ /	35 *12 –
3*5+(1 – 2)/3*2 ↑	Print 3	+ /	35 *12 – 3
3*5+(1 – 2)/3*2 ↑	Push *	+/*	35 *12 – 3
3*5+(1 – 2)/3*2 ↑	Print 2	+/*	35 *12 – 32
3*5+(1 – 2)/3*2 ↑	POP * Print, POP, /Print POP +, Print	Empty	35 *12 – 32*/+

Now reverse the obtained expression + / * 2 3 – 2 1 * 5 3 is the required prefix expression.

Review Questions

1. *Write a program to convert infix expression to postfix.*

SPPU : Dec.-11, 14, Marks 6; May-14, Marks 4

2. *What is stack ? Explain different ways of implementation with application.*

SPPU : May-13, Marks 8

3. *Write an algorithm to convert infix to postfix expression.* **SPPU : May-14, Marks 4**

4. *Write a C/C++ function to convert infix expression to postfix expression.*

SPPU : Dec.-14, Marks 6

3.8 Expression Evaluation

SPPU : Dec.-14,18, May-12,16,19, Marks 6

Algorithm for evaluation of postfix

1. Read the postfix expression from left to right.
2. If the input symbol read is an operand then push it onto the stack.
3. If the operator is read POP two operands and perform arithmetic operations if operator is

+	then result	= operand 1 + operand 2
–	then result	= operand 1 – operand 2
*	then result	= operand 1 * operand 2
/	then result	= operand 1 / operand 2

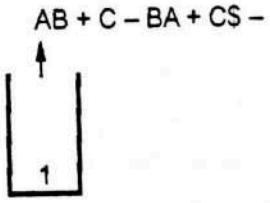
4. Push the result onto the stack.
5. Repeat steps 1-4 till the postfix expression is not over.

For example : Consider postfix expression -

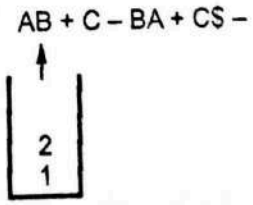
AB + C – BA + C $ – for A = 1, B = 2 and C = 3

Now as per the algorithm of postfix expression evaluation, we scan the input from left to right. If operand comes we push them onto the stack and if we read any operator, we must pop two operands and perform the operation using that operator. Here $ is taken as exponential operator.

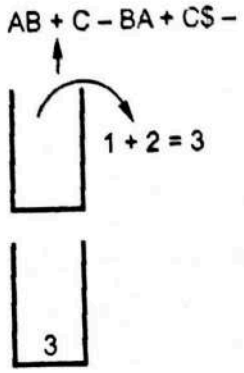
As A = 1, push 1



As B = 2, push 2

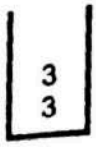


POP two operands and perform addition of 1 and 2. Then push the result onto the stack.



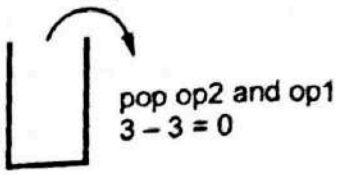
Now, AB + C – BA + C$ –

As C = 3, push 3 onto the stack.

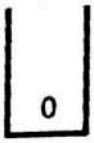


AB + C – BA + C$ –

Perform subtraction.

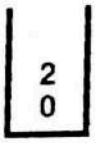


Then push the result onto the stack.



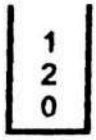
AB + C – BA + C$ –

Push B = 2.



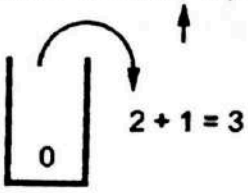
AB + C – BA + C$ –

Push A = 1.

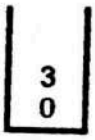


AB + C – BA + C$ –

As operator comes perform operation by popping two operands.

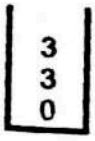


Then push the result onto the stack.



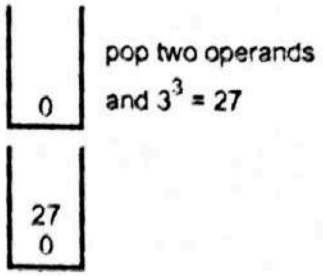
AB + C – BA + C$ –

Push C = 3 onto the stack.

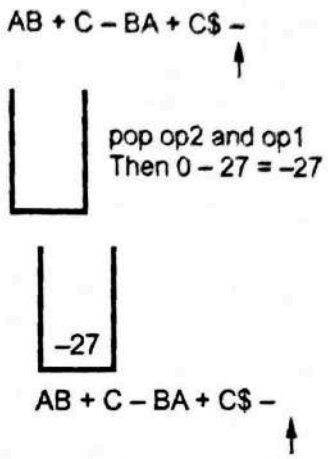


AB + C – BA + C$ –

Then push the computed result.



Perform the subtraction.



Now since there is no further input we should pop the contents of stack and print it as a result of evaluation.

Hence output will be – 27.

C++ Program

```
/*****************************************************************
  Program to evaluate a given postfix expression.
*****************************************************************/
#include<iostream>
#include<cstring>
#include<math.h>
#define size 80
using namespace std ;
class EVAL
{
/*declaration of stack data structure*/
        private:
        struct stack
        {
                double s[size];
                int  top;
        }st;
        public:
                double post(char post[]);
                void push(double);
                double pop();
};
```

```
/*-----------------------------------------------------------------
The post function which is for evaluating postfix expression
Input : A post Expression of single digit operand
Output: Resultant value after evaluating the expression
Parameter Passing Method :By reference
Called By : main()
Calls :push(), pop()
-----------------------------------------------------------------
*/
double EVAL::post(char exp[])
{
   char   ch,*type;
   double  result, val, op1, op2;
   int i;
   st.top = 0;
   i=0;
   ch = exp[i];

   while ( ch !='$' )
   {
         if ( ch >= '0' && ch <= '9')
         type ="operand";
         else if ( ch == '+' ch == '-' ||
          ch == '*' || ch == '/' ||
          ch == '^' )
         type="operator";
         if( strcmp(type,"operand")==0)/*if the character is operand*/
         {
         val = ch - 48;
         push( val);
         }
         else
         if (strcmp(type,"operator")==0)/*if it is operator*/
         {
          op2 = pop();
          op1 = pop(); //popping two operands to perform arithmetic operation
          switch(ch)
         {
         case '+' : result = op1 + op2;
           break;
         case '-' : result = op1 - op2;
         break;
         case '*' : result = op1 * op2;
         break;
         case '/' : result = op1 / op2;
         break;
         case '^' : result = pow(op1,op2);
```

The characters '0', '1', ... '9' will be converted to their values, so that they will perform arithmetic operation.

```
        break;
    }/* switch */
    push(result);
    }
        i++;
        ch=exp[i];
    } /* while */
    result = pop(); /*pop the result*/
    return(result);
}
/*
-----------------------------------------------------------------
    The push function
    Input : A value to be pushed on global stack
    Output: None, modifies global stack and its top
    Parameter Passing Method :By Value
    Called By : Post()
    Calls :none
-----------------------------------------------------------------
*/
void EVAL::push(double val)
{
    if ( st.top+1 >= size )
            cout<<"\nStack is Full\n";
    st.top++;
    st.s[st.top] = val;
}
/*-----------------------------------------------------------------
    The pop function
    Input : None, uses global stack and top
    Output: Returns the value on top of stack
    Parameter Passing Method :None
    Called By : post()
    Calls :none
-----------------------------------------------------------------
*/
double EVAL::pop()
{
    double val;
    if ( st.top == -1 )
        cout<<"\nStack is Empty\n";
    val = st.s[st.top];
    st.top --;
    return(val);
}
/*-----------------------------------------------------------------
    The main function
```

Finally result will be pushed onto the stack.

```
    Input : None
    Output: None
    Parameter Passing Method :None
    Called By : OS
    Calls :post()
  ----------------------------------------------------------
  */
  int main ( )
  {
    char exp[size];
    int   len;
    double  Result;
    EVAL obj;
    cout<<"Enter the postfix Expression\n";
    cin>>exp;
    len = strlen(exp);
    exp[len] = '$'; /* Append $ at the end as a endmarker*/
    Result =obj. post(exp);
    cout<<"Tho Value of the expression is  " <<Result;
    exit(0);
    return 0
  }
```

Output

```
Enter the postfix Expression
12+34*+
Tho Value of the expression is  15
```

Logic of evaluation of postfix expression [Refer the above program]

Let us take some example of postfix expression and try to evaluate it

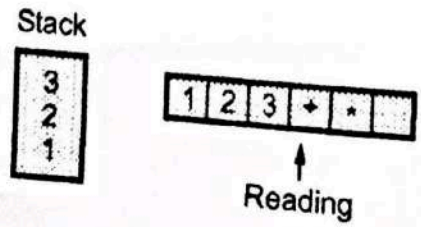
123+*

Step 1 : Assume the array exp [ ] contains the input

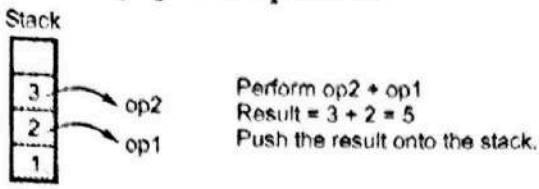
exp

	2	3	+	*	$

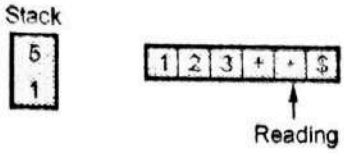
The $ symbol is used as an end marker. Read from first element of the array if i operand push it onto the stack.



Step 2 : If the operator is read pop two operands

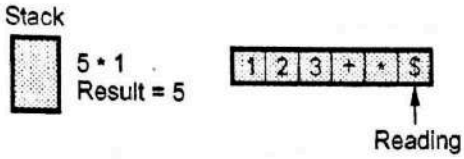


Step 3 :



Again pop two operands and perform the operation.

Step 4 :



At this point the stack is empty and the input is read as $. So here stop the evaluation procedure and return the result = 5.

Example 3.8.1 *Clearly indicate the content of stack during evaluation of postfix expression : ab-cd/*e+, where a=8, b=6, c=10, d=5 and e=7.* **SPPU : Dec.-14, Marks 6**

Solution :

Input	Action	Stack
ab - cd/*e+	Read and push 8 onto the stack	8
ab - cd/*e+	Read b and push 6 onto the stack	8 6
ab - cd/*e+	Pop two operands perform 6 – 8 = – 2. Push – 2 onto the stack	– 2
ab - cd/*e+	Push 10	– 2 10
ab - cd/*e+	Push 5	– 2 10 5
ab - cd/*e+	Pop two operands perform 10/5 = 2. Push 2 onto the stack	– 2 2
ab - cd/*e+	Pop two operands perform – 2 * 2 = – 4 push – 4 onto the stack	– 4
ab - cd/*e+	Push 7	– 4 7
ab - cd/*e+	Pop two operands perform 7 + (– 4) = 3. Push 3 onto stack	3

Pop 3 from the stack and display 3 as a result.

Example 3.8.2 *Clearly indicate the contents of stack for evaluating the following postf expression :*

Assume A = 10, B = 2, C = 13

i) AB + C – BA – C + –

*ii) ABC + * CBA – + **

SPPU : May-16, Marks

Solution : i) AB + C – BA – C + –

Input	Action	Stack
A	Read A and Push 10	10
B	Read B and Push 2	10 2
+	POP two operands and perform 10–2 = 5. Push 5	5
C	Push 13	5 13
–	POP two operands OP2 = 13, OP1 = 5, Perform OP1 – OP2 = 5 – 13 = 8. Push 8.	8
B	Push 2	8 2
A	Push 10	8 2 1
–	POP two operands and OP2 = 10, OP1 = 2 Perform OP1 – OP2 = 2 – 10 = – 8, Push – 8	8 –8
C	Push 13	8 –8
+	POP two operands OP2 = 13, OP1 = –8. OP1 + OP2 = –8 + 13 = 5 Push 5.	8 5
–	POP two operands, OP2 = 5, OP1 = 8. Perform OP1 – OP2 = 8 – 5 = 3 Push 3	3
End of Input	POP and Print 3 as result.	empty

ii) ABC + * CBA – + *

Input	Action	Stack
A	Read A and Push 10	10
B	Read B and Push 2	10 2
C	Read C and Push 13	10 2 13
+	POP two operands OP2 = 13, OP1 = 2. Perform OP1 + OP2 = 2 + 13 = 15. Push 15.	10 15

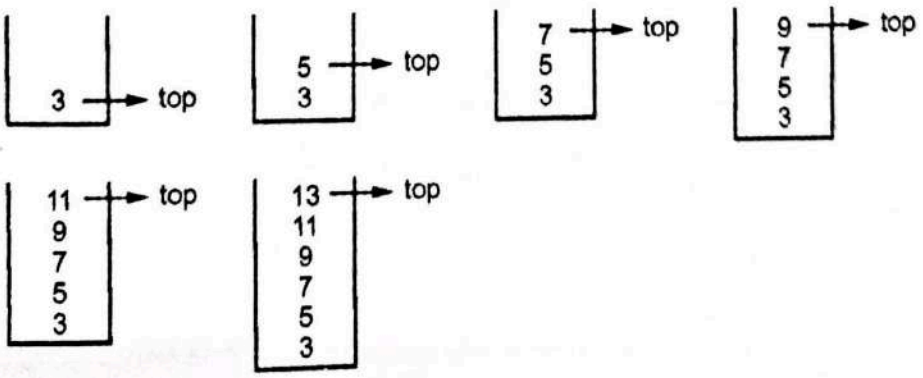
*	POP two operands OP2 = 15, OP1 = 10. Perform OP1 * OP2 = 10 * 15 = 150. Push 150	150
C	Push 13	150 13
B	Push 2	150 13 2
A	Push 10	150 13 2 10
-	POP two operands OP2 = 10, OP1 = 2. Perform OP1 - OP2 = 2 - 10 = -8. Push -8	150 13 -8
+	POP two operands OP2 = -8, OP1 = 13 Perform OP1 + OP2 = -8 + 13 = 5	150 5
*	POP two operands OP2 = 5, OP1 = 150. Perform OP1 * OP2 = 150 * 5 = 750. Push 750.	750
End of Input	POP 750. Print it as result of evaluation = 750.	empty

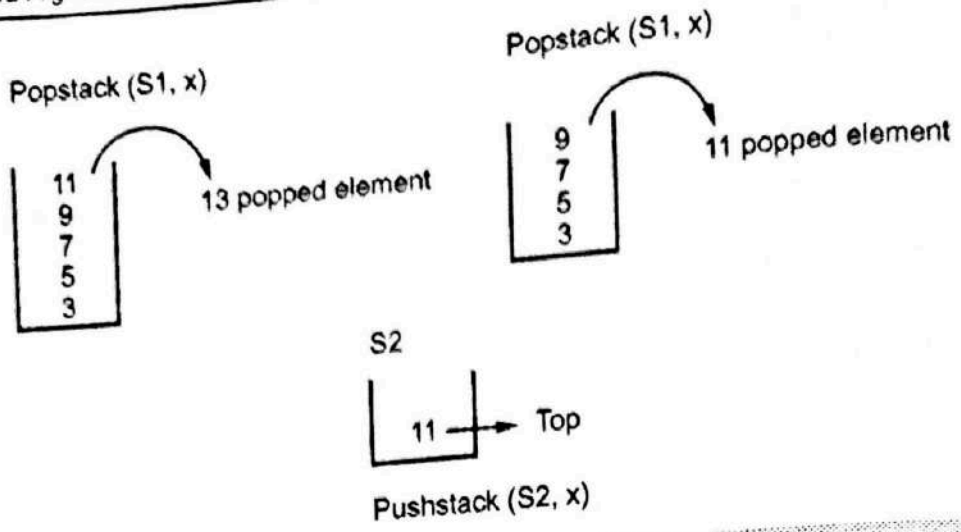
Example 3.8.3 *Imagine we have two empty stacks of integers, S1 and S2. Draw a picture of each stack after the following operations :*

```
Pushstack (S1, 3);
Pushstack(S1, 5)
Pushstack(S1, 7);
Pushstack(S1, 9);
Pushstack(S1, 11);
Pushstack(S1, 13);
While(!emptystack(S1))
{
Popstack(S1, x);
Popstack(S1, x);
Puphstack(S2, x);
}
```

SPPU : May-12, Marks 6

Solution : S1





Example 3.8.4 *Evaluate given expressions : i) 2 3 * 2 1 – / 5 3 * + ii) + * 2 / 3 – 2 1 * 5 3* **SPPU : Dec-18, Marks 6**

Solution :

i) 2 3 * 2 1 – / 5 3 * +

Input Read	Action Taken	Stack
2	Push	2
3	Push	2 3
*	POP 3, POP 2, perform 2*3 = 6. Push 6	6
2	Push 2	6 2
1	Push 1	6 2 1
–	POP 1, POP 2, perform 2 – 1 = 1. Push 1	6 1
/	POP 1, POP 6, perform 6/1 = 6. Push 6	6
5	Push 5,	6 5
3	Push 3	6 5 3
*	POP 3, POP 5 perform 5*3 =15. Push 15	6 15
+	POP 15, POP 6 perform 6+15 = 21. Push 21	21
End of input	POP 21 and Print it as result	

Ans. : 21

ii) + * 2 / 3 – 21 * 53

+*2/3–21*53	
+*2/3(–21)*53	(-21) = (2-1) = 1
+*2/3 1(*53)	(*5 3) = (5*3) = 15
+*2 (/ 3 1)15	(/ 3 1) = (3/1) = 3
+(*2 3)15	(*23) = (2*3) = 6
(+6 15)	(+ 6 15) = (6+15) = 21
21	

Example 3.8.5 *If the values of A, B, C and D are 2, 3, 4 and 5 respectively, calculate the value of the following postfix expressions : i) AB * C – D + ii) ABC + * D –*

SPPU : May-19, Marks 6

Solution : i) AB * C – D +

Input	Action	Stack
A	Read A and Push 2	2
B	Read B and Push 3	2 3
*	POP two operands and perform 2 * 3 = 6. Push 6.	6
C	Read C and Push 4	6 4
–	POP two operands and perform 6 – 4 = 2. Push 2.	2
D	Read D and Push 5	2 5
+	POP two operands and perform 2 + 5 = 7. Push 7.	7
End of Input	POP and Print 7 as result	Empty

ii) ABC + * D –

Input	Action	Stack
A	Read A, and Push 2	2
B	Read B, and Push 3	2 3
C	Read C, and Push 4	2 3 4
+	POP two operands and perform 3 + 4 = 7. Push 7.	2 7

*	POP two operands and perform 2 * 7 = 14. Push 14.	14
D	Read D, and Push 5	14 5
–	POP two operands and perform 14 – 5 = 9 Push 9.	9
End of Input	POP and Print 9 as result	Empty

3.9 Recursion

Definition : Recursion is a programming technique in which the function calls it repeatedly for some input.

By recursion the same task can be performed repeatedly.

Properties of Recursion

Following are two fundamental principles of recursion

1. There must be at least one condition in recursive function which do not involve call to recursive routine. This condition is called a "way out" of the sequence recursive calls. The is called **base case property.**
2. The invoking of **each recursive call must reduce** to some manipulation and go closer to base case condition.

For example - While computing factorial using recursive method -

```
if(n==0)
  return 1;  //This is a base case
else
  return n*fact(n-1);//on each call the computation will go closer to base case
```

Example

Algorithm for Factorial Function using Iterative Definition

```
1. prod = 1;
2. x=n;
3. while(x>0)
4. {
5. prod = prod*x;
6. x – –;
7. }
8. return(prod);
```

Algorithm for Factorial Function using Recursive Definition

```
1. if(n==0)
2. fact =1;
3. else
   {
4. x=n-1;
5. y= value of x!;
6. fact = n* y ;
7. } /*else ends here */
```

This definition is called the **recursive definition** of the factorial function. This definition is called recursive because again and again the same procedure of multiplication is followed but with the different input and result is again multiplied with the next input. Let us see how the recursive definition of the factorial function is used to evaluate the 5 !

Step 1. 5! = 5 * 4!

Step 2. 4! = 4 * 3!

Step 3. 3! = 3 * 2!

Step 4. 1! = 1 * 0!

Step 5. 2! = 2 * 1!

Step 6. 0! = 1.

Actually the step 6 is the only step which is giving the direct result. So to solve 5 !, we have to backtrack from step 6 to step 1,collecting the result from each step. Let us see how to do this.

Step 6′. 0! =1

Step 5′. 1! = 1*0! = 1 from step 6′

Step 4′. 2! = 2*1! = 2 from step 5′

Step 3′. 3! = 3*2! = 6 from step 4′

Step 2′. 4! = 4*3! = 24 from step 3′

Step 1′. 5! = 5*4! = 120 from step 2′

Example 3.9.1 *Generate n^{th} term of Fibonacci sequence with recursion.*

Solution :

```
#include<iostream>
using namespace std;
int fib(int n)
{
        int x,y;
        if(n<=1)
                return n;
        x=fib(n-1);
        y=fib(n-2);
        return (x+y);
}
int main(void)
{
        int n,num;
        cout<<"\n Enter location in fibonacci series: ";
        cin>>n;
        num=fib(n);
        cout<<"\n The number at "<<n<<" position is "<<num<<" in fibonacci series";
        return 0;
}
```

Output

```
Enter location in fibonacci series: 3
The number at 3 position is 2 in fibonacci series
```

3.9.1 Use of Stack in Recursive Functions

For illustrating the use of stack in recursive functions, we will consider an example finding factorial. The recursive routine for obtaining factorial is as given below -

```
int fact (int num)
{
        int a, b;
        if (num = = 0)
                return 1 ;
                else
        {
                a = num - 1 ;
                b = fact (a) ;
                f = a * b;
                return f ;
        }
}
/* Call to the function */
ans = fact (4) ;
```

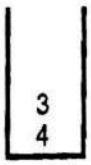
For the above recursive routine we will see the use of stack. For the stack the only principle idea is that – when there is a call to a function the values are pushed onto the stack and at the end of the function or at the return the stack is popped

Suppose we have num = 4. Then in function fact as num ! = 0 else part will be executed. We get

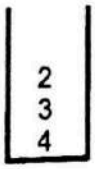
a = num – 1

a = 3

And a recursive call to fact (3) will be given. Thus internal stack stores



Next fact (2) will be invoked. Then

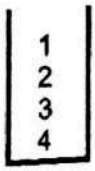


∵ a = num – 1

a = 3 – 1

b = fact (2)

Then,



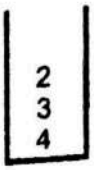
∵ a = num – 1

a = 2 – 1

b = fact (1)

Now next as num = 0, we return the value 1.

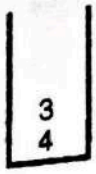
∴ a = 1, b = 1, we return f = a * b = 1 from function fact. Then 1 will be popped off.



Now the top of the stack is the value of a

Then we get b = fact (2) = 1

∴ f = a * b = 2 * 1 = 2 will be returned.



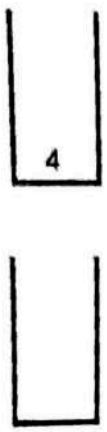
Here a = 3

Then b = fact (3)

b = 2

∴ f = a * b = 6 will be returned.

The stack will be



Here a = 4

Then b = fact (4)

= 6

∴ f = a * b = 24 will be returned.

As now stack is empty. We return f = 24 from the function **fact.**

3.9.2 Programming Examples

Example 3.9.2 *Explain the how to check the well formedness of parenthesis using stack ?*

Solution :

- The well formedness of the parenthesis is checked by means of the stack data structure.
- The simple logic for checking it as push all the opening brackets onto the stack and as soon as the closing bracket comes perform the pop operation.
- If you read one closing parenthesis perform one pop operation which should result in popping off one opening parenthesis.

```
#include <iostream>
#include <cstring>
using namespace std;
#define SIZE 30
class Balanced
{
    private:
        int top;
        char stack[SIZE];
    public:
    // function prototypes
    Balanced()
    {
        top = -1;
    }
    void push(char);
    void pop();
```

```
Move disk 1 from B peg to C
Move disk 3 from A peg to B
Move disk 1 from C peg to A
Move disk 2 from C peg to B
Move disk 1 from A peg to B
Move disk 4 from A peg to C
Move disk 1 from B peg to C
Move disk 2 from B peg to A
Move disk 1 from C peg to A
Move disk 3 from B peg to C
Move disk 1 from A peg to B
Move disk 2 from A peg to C
Move disk 1 from B peg to C
```

Part II : Queue

3.10 Concept of Queues

SPPU : May-19, Marks 4

The queue can be formally defined as ordered collection of elements that has two ends named as **front** and **rear.** From the front end one can delete the elements and from the rear end one can insert the elements.

For Example :

The typical example can be a queue of people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus. You can call it as the front end of the queue.

Following Fig. 3.10.1 represents the queue of few elements.

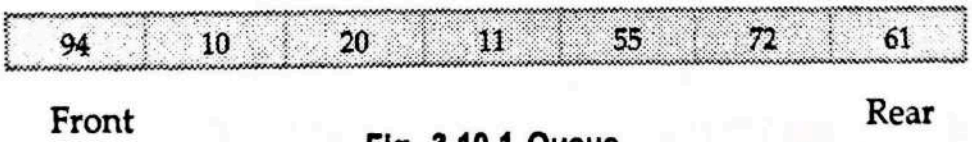


Fig. 3.10.1 Queue

Review Question

1. *Explain the following Linear data structure - Queue* **SPPU : May-19, Marks 4**

3.11 Queue as ADT

AbstractDataType Queue

{

Instances : The queue is a collection of elements in which the element can be inserted by one end called rear and elements can be deleted by other end called front.

Operations :

1. Insert : The insertion of the element in the queue is done by the end called rear. Before the insertion of the element in the queue it is checked whether or not the queue is full.

2. Delete : The deletion of the element from the queue is done by the end called front. Before performing the delete operation it checked whether the queue is empty or not.

}

3.12 Implementation of Queue using Array

- As we have seen, queue is nothing but the collection of items.
- Both the ends of the queue are having their own functionality.
- The Queue is also called as FIFO i.e. a First In First Out data structure. All the elements in the queue are stored sequentially.
- Various operations on the queue are -

1. Queue overflow.
2. Insertion of the element into the queue.
3. Queue underflow.
4. Deletion of the element from the queue.
5. Display of the queue.

Let us see each operation one by one.

C++' representation of queue.

```
struct queue
{
    int que [size];
    int front;
    int rear;
} Q;
```

1. Insertion of element into the queue

The insertion of any element in the queue will always take place from the rear end.

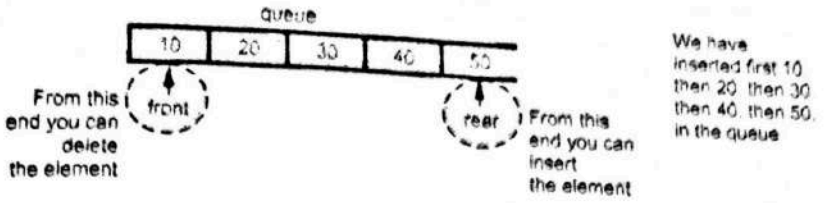


Fig. 3.12.1 Representing the insertion

Before performing insert operation you must check whether the queue is full or not. If the rear pointer is going beyond the maximum size of the queue then the queue overflow occurs.

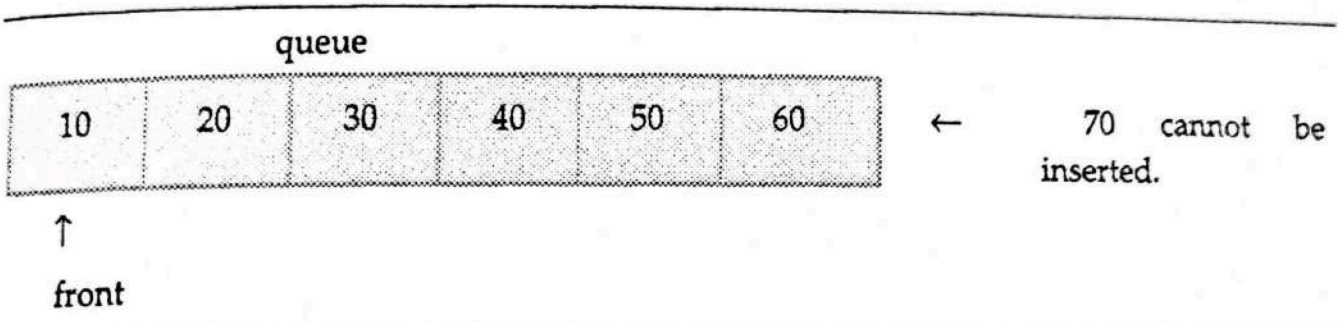


Fig. 3.12.2 Representing the queue overflow

2. Deletion of element from the queue

The deletion of any element in the queue takes place by the front end always.

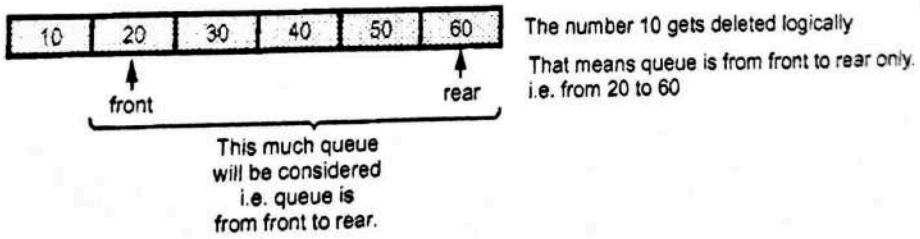


Fig. 3.12.3 Representing the deletion

Before performing any delete operation one must check whether the queue is empty or not. If the queue is empty, you can not perform the deletion. The result of illegal attempt to delete an element from the empty queue is called the queue underflow condition.

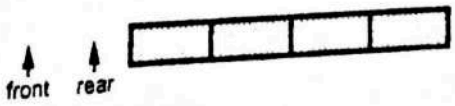


Fig. 3.12.4 Representing the queue underflow

Let us see the C++ implementation of the queue.

```
  cout<<" "<<Q.que[i];
}
void main(void)
{
 int choice,item;
 char ans;
 MyQ obj;
 clrscr();
 do
 {
  cout<<"\n Main Menu";
  cout<<"\n1.Insert\n2.Delete\n3.Display";
  cout<<"\n Enter Your Choice: ";
  cin>>choice;
  switch(choice)
      {
      case 1:if(obj.Qfull())       //checking for Queue overflow
                  cout<<"\n Can not insert the element";
            else
            {
                  cout<<"\n Enter The number to be inserted ";
                  cin>>item;
                  obj.insert(item);
            }
            break;
      case 2:if(obj.Qempty())
                  cout<<"\n Queue Underflow!!";
                  else
                  obj.delet();
                  break;
      case 3:if(obj.Qempty())
                  cout<<"\nQueue Is Empty!";
                  else
                  obj.display();
                  break;
      default:cout<<"\n Wrong choice!";
                  break;
      }
 cout<<"\n Do You Want to continue?";
 ans =getche();
 }while(ans ==='Y'||ans ==='y');
}
/************************ End Of Program ************************/
```

Output

```
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 1
```

```
Enter The number to be inserted 10

Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 1

Enter The number to be inserted 20

Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 1

Enter The number to be inserted 30

Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 1

Enter The number to be inserted 40

Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 3
10 20 30 40
Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 2

The deleted item is 10
Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 2
```

```
The deleted item is 20
Do You Want to continue?y
Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice: 3
30 40
Do You Want to continue?
```

Review Question

1. *Explain about the operations of queue with an example.*

3.13 Implementation of Queue using Linked Organization

SPPU : May-10,12, Dec-14, March-16

The main advantage in linked representation is that we need not have to worry about size of the queue. As in linked organization we can create as many nodes as we want so there will not be a queue full condition at all. The queue using linked list will be very much similar to a linked list. The only difference between the two is in queue, the left most node is called **front node** and the right most node is called **rear node**. And we can not remove any arbitrary node from queue. We have to remove front node always :

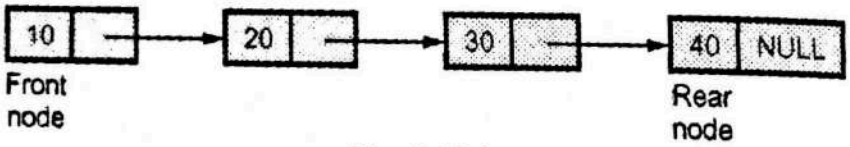


Fig. 3.13.1

The typical node structure will be,

```
typedef struct node
    {
        int data;
        struct node  next;
    }Q;
```

Various operations that can be performed on queue are

1. Insertion of a node in queue
2. Deletion of node from the queue
3. Checking whether queue is empty or not
4. Display of a queue

All these operations are implemented in following C++ program.

C++ Program

```
/*****************************************************************
Program For implementing the Queue using the linked list.
The queue full condition will never occur in this program.
*****************************************************************/
```

```
#include<iostream.>
using namespace std;
//Declaration of Linked Queue data structure
class Lqueue
{
 private:
 typedef struct node
 {
  int data;
  struct node *next;
 }Q;
 Q *front,*rear;
 public:
 Lqueue();
 ~Lqueue();
 void create(),remove(),show();
 void insert();
 Q *delet();
 void display(Q *);
};

/*
-----------------------------------------------
The constructor defined
-----------------------------------------------*/
Lqueue::Lqueue()
{
 front=NULL;
 rear= NULL;
}
/*
-----------------------------------------------
The create function
-----------------------------------------------*/
void Lqueue::create()
{
insert();
}
/*
-----------------------------------------------
The remove function
-----------------------------------------------*/
void Lqueue::remove()
{
  front =delet();
}
/*
-----------------------------------------------
The show function
-----------------------------------------------*/
```

```
void Lqueue::show()
{
  display(front);
}
/*
*****************************************************
 The insert Function
*****************************************************
*/
void Lqueue::insert()
{
 char ch;
 Q *temp;
 clrscr();
 temp =new Q;//allocates memory for temp node
 temp->next=NULL;
 cout<<"\n\n\n\tInsert the element in the Queue\n";
 cin>>temp->data;

 if(front == NULL)//creating first node
 {
  front= temp;
  rear=temp;
 }
 else              //attaching other nodes
  {
      rear->next=temp;
      rear=rear->next;
      }
}

/*
*****************************************************
The QEmpty Function
*****************************************************
int Qempty(Q *front)
{
 if(front ==NULL)
  return 1;
 else
  return 0;
}
/*
*****************************************************
The delet Function
*****************************************************
*/
Q *Lqueue::delet()
```

```
{
 Q *temp;
 temp=front;
if(Qempty(front))
 {
      cout<<"\n\n\t\tSorry!The Queue Is Empty\n";
      cout<<"\n Can not delete the element";
 }
 else
  {
      cout<<"\n\tThe deleted Element Is  "<<temp->data;
      front=front->next;
      temp->next=NULL;
      delete temp;
  }
return front;
}
/*
-----------------------------------------------------
The display Function
-----------------------------------------------------
*/
void Lqueue::display(Q *front)
{
if(Qempty(front))
 cout<<"\n The Queue Is Empty\n";
else
{
 cout<<"\n\t The Display Of Queue Is \n ";
 for(;front !=rear->next;front=front->next)
      cout<<"  "<<front->data;
}

}
/*
-----------------------------------------------------
The destructor defined
-----------------------------------------------------
*/
Lqueue::~Lqueue()
{
if((front!=NULL)&&(rear!=NULL))
{
front=NULL;
rear=NULL;
delete front;
delete rear;
```

```
  }
 }

/*
 ---------------------------------------------------------
 The main Function
 Calls:create,remove,show
 Called By:O.S.
 ---------------------------------------------------------
 */
int main(void)
{
 char ans;
 int choice;
 Lqueue Que;
 do
 {

      cout<<"\n\tProgram For Queue Using Linked List\n";
      cout<<"\n          Main Menu";
      cout<<"\n1.Insert\n2.Delete \n3.Display";
      cout<<"\n Enter Your Choice";
      cin>>choice;
      switch(choice)
      {
       case 1 :Que.create();
                break;
       case 2 :Que.remove();
                break;
       case 3 :Que.show();
                break;
      default:cout<<"\nYou have entered Wrong Choice"<<endl;
              break;
 }
 cout<<"\nDo You Want To See Main Menu?(y/n)"<<endl;
 cin>>ans;
 }while(ans == 'y' || ans == 'Y');
return 0;
}
```

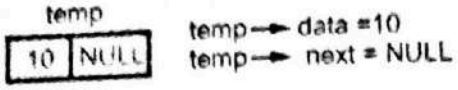
Output

```
      Program For Queue Using Linked List
         Main Menu
1.Insert
2.Delete
3.Display
Enter Your Choice 1
```

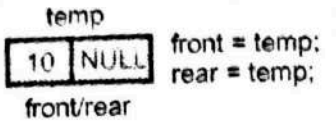
```
Insert the element in the Queue
10
Do You Want To See Main Menu?(y/n)
            Program For Queue Using Linked List
        Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice 1

 Insert the element in the Queue
20

Do You Want To See Main Menu?(y/n)
      Program For Queue Using Linked List
         Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice 1

 Insert the element in the Queue
30

Do You Want To See Main Menu?(y/n)
      Program For Queue Using Linked List
         Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice 1

 Insert the element in the Queue
40

Do You Want To See Main Menu?(y/n)

      Program For Queue Using Linked List

         Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice3

         The Display Of Queue Is
  10  20  30  40
Do You Want To See Main Menu?(y/n)
```

In Insert function

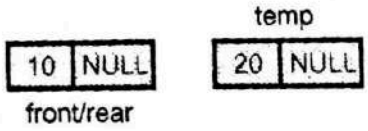
First we will allocate a node called 'temp' with next field assigned with 'Null' value. Suppose we want to insert data 10 then



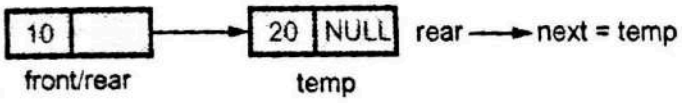
Now mark this node as queue's front and rear.



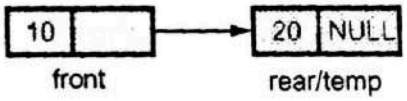
Now next, suppose we want to insert 20 in queue then a new node 'temp' will be created with data 20.



Then, the queue will be formed by setting appropriate front and rear.



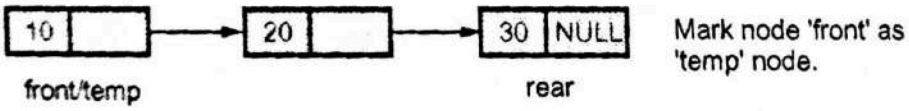
Then



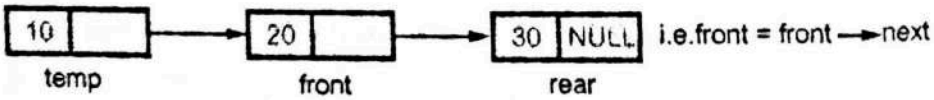
Continuing in this fashion we can create a linked queue.

In delet function

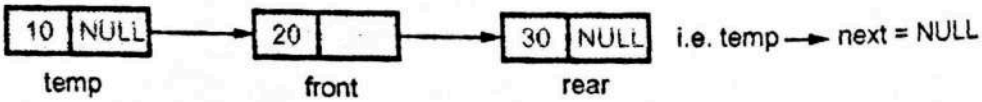
We assume that a queue is created like this -



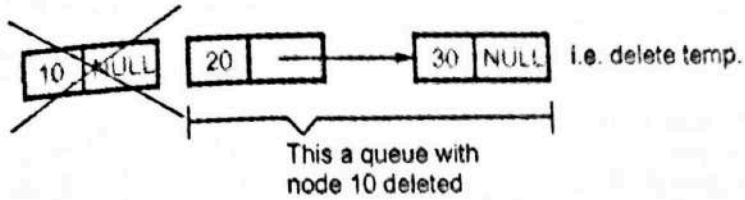
Simply display a message that "The deleted Element is 10" (i.e. temp → data). Then set front as



Then



Then,



Review Questions

1. *Define linear queue. How to represent it using linked organization ? Explain any one application in detail.* **SPPU : Dec-14, Marks 6, May-10, Marks 10**
2. *Write an pseudocode for simple queue using linked organization and represent each pseudocode step diagrammatically.* **SPPU : May-12, Marks 8**

3.14 Concept of Circular Queue

SPPU : May-10,11,13,14,15,16, Dec-10,11,14, Marks 8

As we have seen, in case of linear queue the elements get deleted logically. This can be shown by following Fig. 3.14.1.

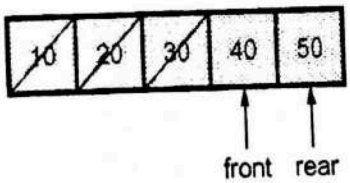


Fig. 3.14.1 Linear queue

We have deleted the elements 10, 20 and 30 means simply the front pointer is shifted ahead. We will consider a queue from front to rear always. And now if we try to insert any more element then it won't be possible as it is going to give "queue full !" message. Although there is a space of elements 10, 20 and 30 (these are deleted elements), we can not utilize them because queue is nothing but a linear array !

Hence there is a concept called **circular queue.** The main advantage of **circular queue** is we can **utilize the space** of the queue fully. The circular queue is shown by following Fig. 3.14.2.

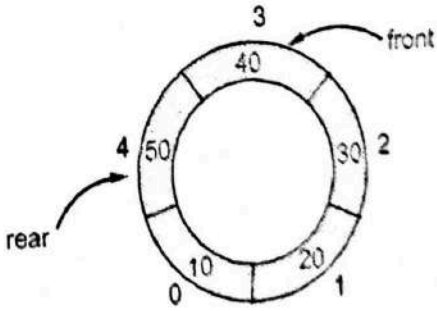


Fig. 3.14.2 Circular queue

Considering that the elements deleted are 10, 20 and 30.

There is a formula which has to be applied for setting the front and rear pointers, for a circular queue.

```
rear = (rear + 1) % size
front = (front + 1) % size
```

$$rear = (rear + 1) \% size$$
$$= (4 + 1) \% 5$$
$$rear = 0$$

So we can store the element 60 at 0th location similarly while deleting the element.

$$front = (front + 1) \% size$$
$$= (3 + 1) \% 5$$
$$front = 4$$

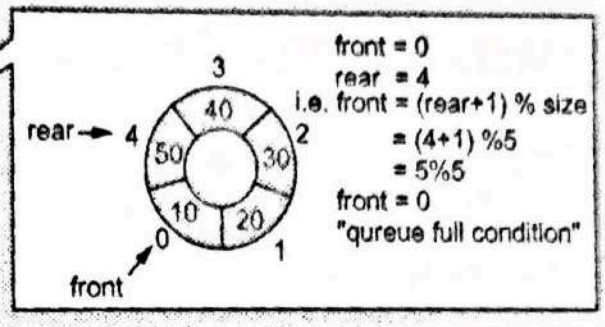
So delete the element at 4th location i.e. element 50

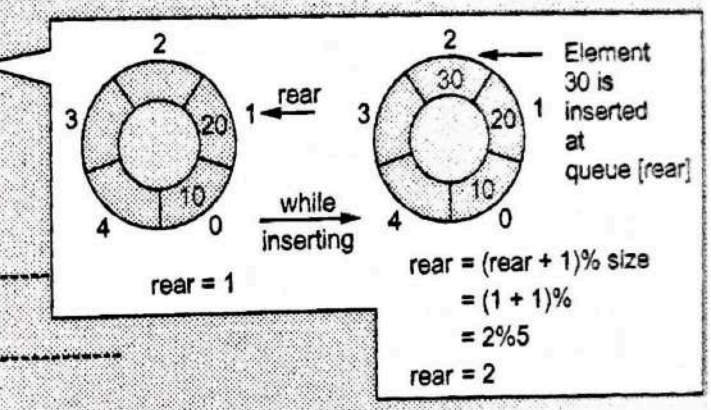
Let us see the 'C++' program now,

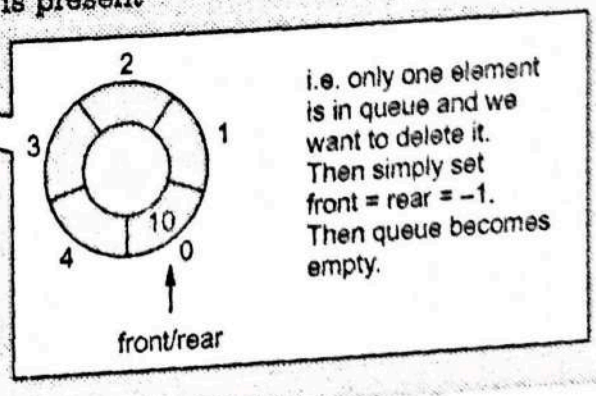
3.14.1 Implementation of Circular Queue

```
#include<iostream.>
#define size 10
using namespace std;
class Queue
{
int Que[size];
int front,rear;
public:
Queue()//constructor defined
{
      front=-1;
      rear=0;
}
void init();
void insert(int ch);
int delet();
void display();
};

void Queue::init()
{
for(i=0;i< size;i++)
      Que[i]=0;
}
```

```
/*
-----------------------------------------------
insert function
-----------------------------------------------
*/

void Queue::insert(int item)
{
if (front==(rear+1)% size)
{
      cout << "Queue is full\n";
}
else
{
      //setting front pointer for a single element in Queue
      if(front==-1)
       front=rear=0;
      else
            rear=(rear+1)% size;
      Que[rear]=item;
}
}
/*
-----------------------------------------------
delet function
-----------------------------------------------
*/
int Queue::delet()
{
 int val;
 if(front==-1)
 {
      cout << "Queue is empty\n";
      return 0; // return null on empty Queue
 }
 val= Que[front];//item to be deleted
 if(front==rear)//when single element is present
 {
      front=rear=-1;
 }
 else
      front=(front+1)% size;
 return val;
}
```







```
/*
-----------------------------------------------------
Display function
-----------------------------------------------------
*/
void Queue::display()
{
 int i;
 i=front;
 while(i!=rear)
 {
        cout<<Que[i]<<" ";
        i=(i+1)%MAX;
 }
 cout<<Que[i]<<endl;
}

/*
-----------------------------------------------------
The main function
-----------------------------------------------------
*/
int main()
{
int i,choice,item;
char ans='y';
Queue obj;
obj.init();
do
{
      cout<<"   Menu"<endl;
      cout<<"1.Insert \n 2.Delete \n 3.Display"<<endl;
      cout<<"Enter Your choice"<<endl;
      cin>>choice;
      switch(choice)
      {
       case 1:cout<<"Enter the element"<<endl;
             cin>item;
                obj.insert(item);
              break;
       case 2:cout<<"The element to be deleted is"<<endl;
             cout<<obj.delet();
                break;
       case 3:obj.display();
             break;
      }
      cout<<"Do You Want to Continue?"<<endl;
```

```
        ans=getche();
    }while(ans==='Y'||ans==='y');
    return 0;
    }
```

Output

```
Menu
   1.Insert
   2.Delete
   3.Display
   Enter Your choice: 1

   Enter the element
   10
   Do You Want to Continue?  y
     Menu
   1.Insert
   2.Delete
   3.Display
   Enter Your choice: 1

   Enter the element
   20
   Do You Want to Continue?  y
     Menu
   1.Insert
   2.Delete
   3.Display
   Enter Your choice: 1

   Enter the element
   30
   Do You Want to Continue?  y
     Menu
   1.Insert
   2.Delete
   3.Display
   Enter Your choice: 3
   10  20  30
   Do You Want to Continue?  y
     Menu
   1.Insert
   2.Delete
   3.Display
   Enter Your choice: 2
   The element to be deleted is
   10
   Do You Want to Continue?  y
```

```
Menu
1.Insert
2.Delete
3.Display
Enter Your choice: 3
20 30
Do You Want to Continue? n
```

Example 3.14.1 *Consider a circular queue of characters and is of size 6. "–" denotes an empty queue location. Show the queue contents as the following opns. take place :*

i) F is added to the queue *ii) Two letters are deleted.*

iii) K, L and M are added to the queue. *iv) Two letters are deleted.*

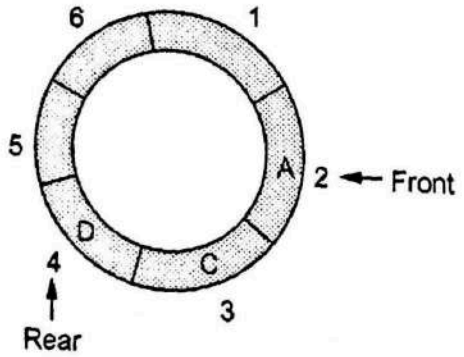
v) R is added to the queue. *vi) Two letters are deleted*

vii) S is added to the queue. *viii) Two letters are deleted.*

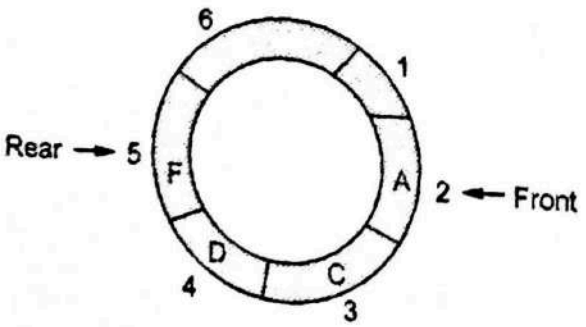
Initial queue configuration is ;

FRONT = 2, REAR = 4, Queue : –, A, C, D, –, – **SPPU : Dec.-10, Marks 8**

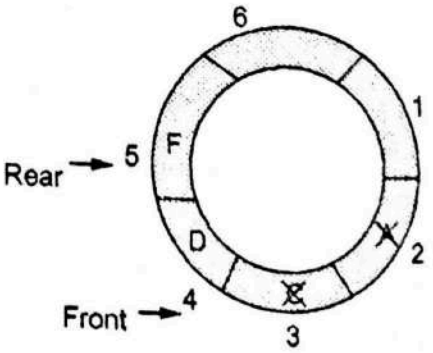
Solution : The initial configuration is,



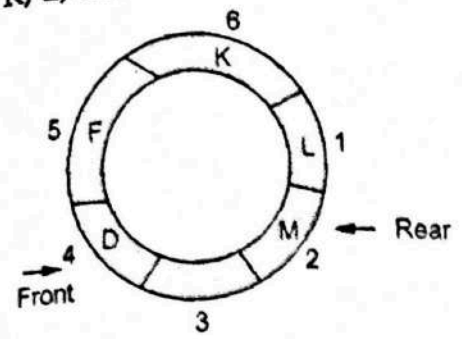
i) F is added



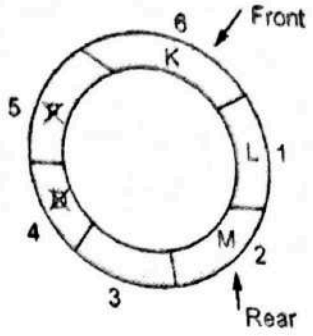
ii) Two letters are deleted



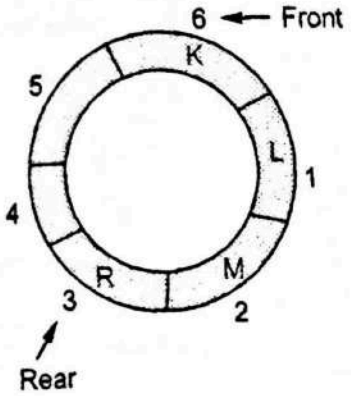
iii) K, L, M are added



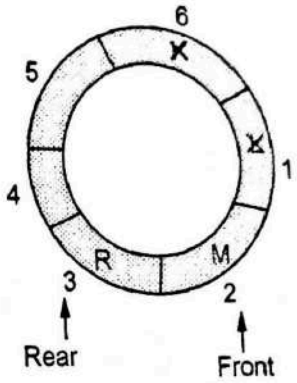
iv) Two letters are deleted



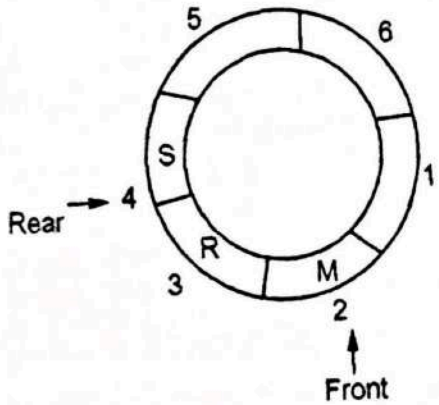
v) R is added



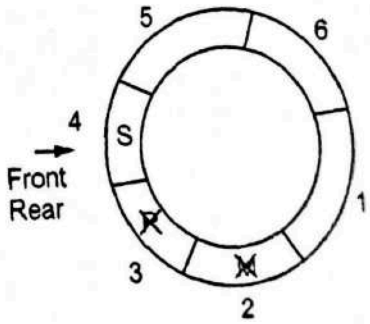
vi) Two letters are deleted



vii) S is added to the queue



viii) Two letters are deleted



Finally

----- S -----

Front Rear

Example 3.5.12 *Consider following circular queue of characters and size 5.*

1	2	3	4	5
		A	C	

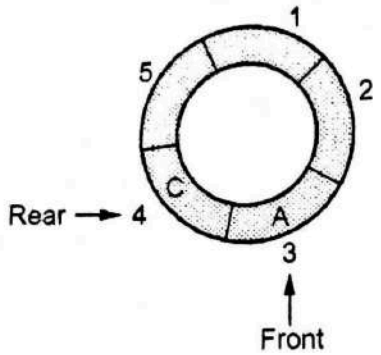
Front point to A and Rear Points to C

Show the queue contents as per the following operations at every step.

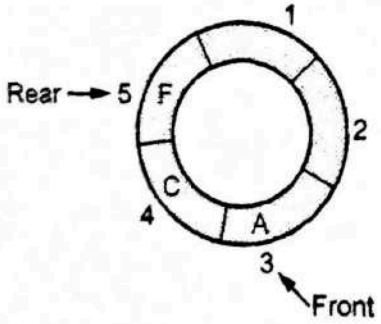
i) F is added to the queue.
ii) Two letters are deleted.
iii) K, L, M are added to the queue.
iv) Two letters are deleted.
v) R is added to the queue.
vi) Two letters are deleted.
vii) R is added to the queue.
viii) Two letters are deleted.

SPPU : May-14,16, Marks 6

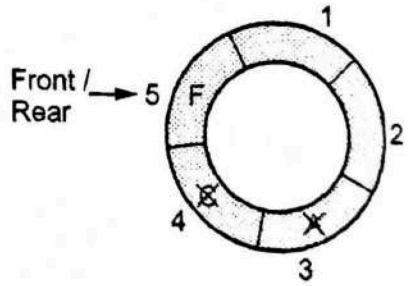
Solution : Consider the circular representation as,



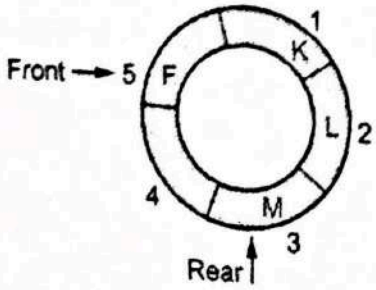
Step 1 : F is added



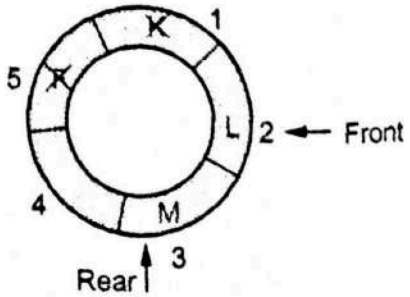
Step 2 : Two letters deleted



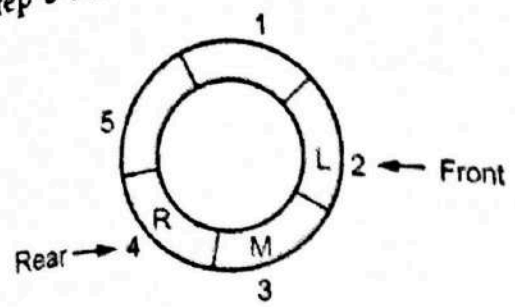
Step 3 : K, L, M are added



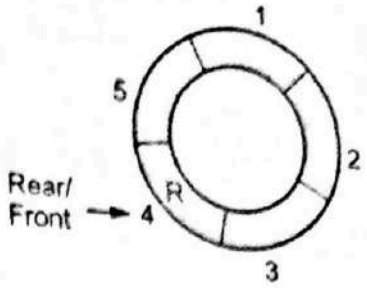
Step 4 : Two letters are deleted



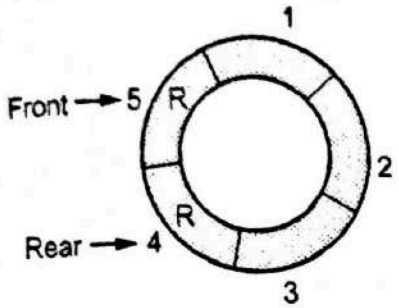
Step 5 : R is added



Step 6 : Two letters deleted



Step 7 : R is added to the queue



Step 8 : Two letters are deleted

The queue becomes empty.

Review Questions

1. *Explain the advantages of circular queue over linear queue with an example.* **SPPU : May-10,11, Marks 4**
2. *Implement circular queue as ADT.* **SPPU : Dec-10, Marks 8**
3. *Implement circular queue as an ADT using linked list.* **SPPU : Dec-11, Marks 8, May-13, Marks 6**
4. *Define circular queue. Explain the advantages of circular queue over linear queue with example.* **SPPU : Dec-14, Marks 6**
5. *Explain the implementation of circular queue using sequential organization.* **SPPU : May-15, Marks 6**

3.15 Double Ended Queue

SPPU : May-10,13,16, Dec-12,13, Marks 10

In a linear queue, the usual practice is for insertion of elements we use one end called rear and for deletion of elements we use another end called as front. But in the doubly ended queue we can make use of both the ends for insertion of the elements as well as we can use both the ends for deletion of the elements. That means it is possible to insert the elements by rear as well as by front. Similarly it is possible to delete the elements from front as well as from rear. Just see the following figure to understand the concept of doubly ended queue i.e. deque.

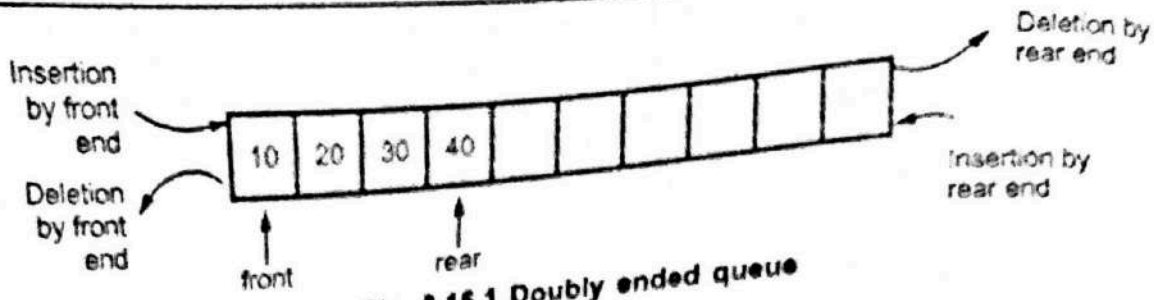
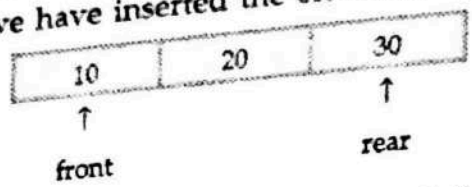


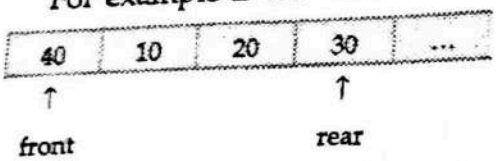
Fig. 3.15.1 Doubly ended queue

As we know, normally we insert the elements by rear end and delete the elements from front end. Let us say we have inserted the elements 10, 20, 30 by rear end.

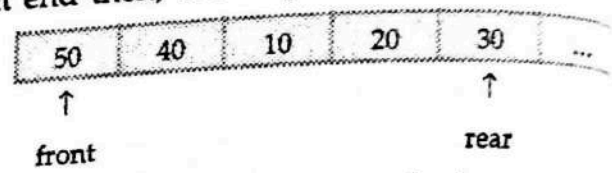


Now if we wish to insert any element from front end then first we have to shift all the elements to the right.

For example if we want to insert 40 by front end then, the deque will be

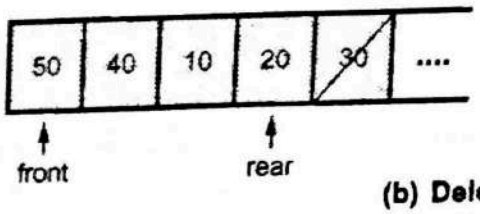


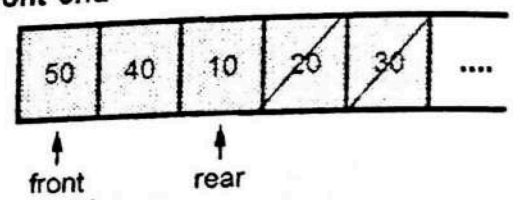
Inserted 40 in the queue by front end



Inserted 50 in the queue by front end

(a) Insertion by front end





(b) Deletion by rear end

Fig. 3.15.2 Operations on deque

We can place –1 for the element which has to be deleted.

Let us see the implementation of deque using arrays

```
/*****************************************************************
Program To implement Doubly ended queue using arrays
*****************************************************************/
 #include<iostream>
 #include<stdlib.h>
 #define size 5
 using namespace std;
 class Dqueue
 {
   private:
         int que[size];
   public:
```

```
        int front,rear;
        Dqueue();
        int Qfull();
        int Qempty();
        int insert_rear(int item);
        int delete_front();
        int insert_front(int item);
        int delete_rear();
        void display();

};

Dqueue::Dqueue()
{
   front=-1;
   rear=-1;
   for(int i=0;i<size;i++)
         que[i]=-1;


}

int Dqueue::Qfull()
{
   if(rear==size-1)
         return 1;
   else
         return 0;
}
   int Dqueue::Qempty()
{
   if((front>rear)||(front==-1&&rear==-1))
         return 1;
  else
         return 0;
}

int Dqueue::insert_rear(int item)
{
    if(front==-1&&rear==-1)
          front++;
    que[++rear]=item;
    return rear;
}

int Dqueue::delete_front()
{
    int item;
    if(front==-1)
```

```
            front++;
        item=que[front];
        que[front]=-1;
        front++;
        return item;
  }

  int Dqueue::insert_front(int item)
  {
      int i,j;
      if(front==-1)
            front++;
      i=front-1;
      while(i>=0)
      {
            que[i+1]=que[i];
        i--;
      }
      j=rear;
      while(j>=front)
      {
            que[j+1]=que[j];
            j--;
      }
      rear++;
      que[front]=item;
      return front;
}
int Dqueue::delete_rear()
{
      int item;
      item=que[rear];
      que[rear]=-1;/*logical deletion*/
      rear--;
      return item;
}

void Dqueue::display()
{
      int i;
      cout<<"\n Straight Queue is:";
            for(i=front;i<=rear;i++)
      cout<<" "<<que[i];
}

int main()
{
```

```
int choice,item;
char ans;
ans='y';
Dqueue obj;
cout<<"\n\t\t Program For doubly ended queue using arrays";
do
{
    cout<<"\n1.insert by rear\n2.delete by front\n3.insert by front\n4.delete by
    rear";
    cout<<"\n5.display\n6.exit";
    cout<<"\n Enter Your choice ";
    cin>>choice;
    switch(choice)
    {
    case 1:if(obj.Qfull())
            cout<<"\n Doubly ended Queue is full";
            else
                {
                cout<<"\n Enter The item to be inserted";
                cin>>item;
                obj.rear=obj.insert_rear(item);
                }
           break;

    case 2:if (obj.Qempty())
            cout<<"\n Doubly ended Queue is Empty";
            else
            {
            item=obj.delete_front();
            cout<<"\n The item deleted from queue is "<<item;
            }
            break;
    case 3:if(obj.Qfull())
            cout<<"\n Doubly ended Queue is full";
            else
            {
             cout<<"\n Enter The item to be inserted";
             cin>>item;
             obj.front=obj.insert_front(item);
            }
            break;
    case 4:if(obj.Qempty())
            cout<<"\n Doubly ended Queue is Empty";
            else
            {
            item=obj.delete_rear();
            cout<<"\n The item deleted from queue is "<<item;
            }
```

```
                break;
            case 5:obj.display();
                break;
            case 6:exit(0);
    }
    cout<<"\n Do You Want To Continue?";
    cin>>ans;
  }while(ans==='y'||ans==='Y');
  return 0;
}
```

Output

```
Program For doubly ended queue using arrays
1.insert by rear
2.delete by front
3.insert by front
4.delete by rear
5.display
6.exit
Enter Your choice 1

Enter The item to be inserted 10

Do You Want To Continue?y

1.insert by rear
2.delete by front
3.insert by front
4.delete by rear
5.display
6.exit
Enter Your choice 1

Enter The item to be inserted 20

Do You Want To Continue?y

1.insert by rear
2.delete by front
3.insert by front
4.delete by rear
5.display
6.exit
Enter Your choice 1

Enter The item to be inserted 30
Do You Want To Continue?y
1.insert by rear
2.delete by front
```

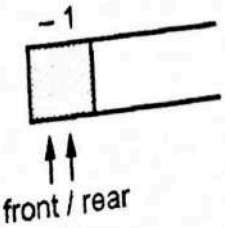
```
3.insert by front
4.delete by rear
5.display
6.exit
Enter Your choice 5

Straight Queue is:  10  20  30
Do You Want To Continue?
```

[illegible] *The dq is an input restricted doubly ended queue, implemented as a linear queue. The delete functions return the element that is deleted and also stores 0 (zero) at the location of deleted element. Clearly indicate the contents of queue after each add and delete operation given below :*

```
struct dqueue
{
        int front
        in arr[10];
        int rear;

};
Struct dque dq = {-1, {0}, -1};
int i = 0;
addqatend (&dq, 11);
addqatend (&dq, 12);
addqatend (&dq, 13);
i = delqatbeg (&dq);
i = delqatbeg (&dq);
i = delqatend (&dq);
addqatend (&dq, 22);
addqatend (&dq, 23);
i = delqatbeg (&dq);
i = delqatend (&dq);
addqatend (&dq, 24);
```

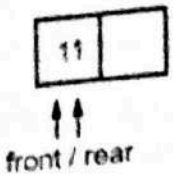
SPPU : May-16, Marks

Solution : Step 1 : Initial Configuration



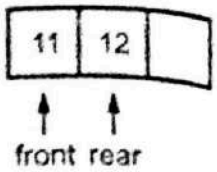
Step 2 :

addatend (& dq, 11)



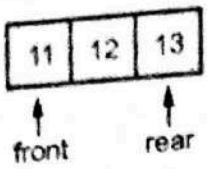
Step 3 :

addatend (& dq, 12)



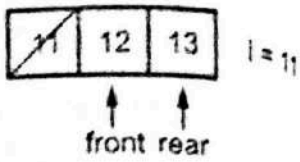
Step 4 :

addatend (& dq, 13)



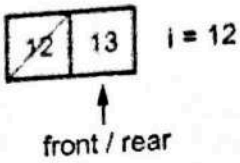
Step 5 :

i = delatbeg (& dq)



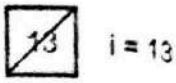
Step 6 :

i = delatbeg (& dq)



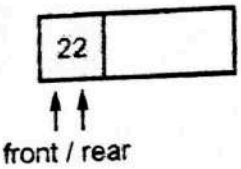
Step 7 :

i = delatbeg (& dq)



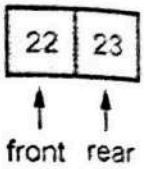
Step 8 :

addatend (& dq, 22)



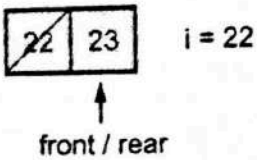
Step 9 :

addatend (& dq, 23)



Step 10 :

i = delqatend (& dq)

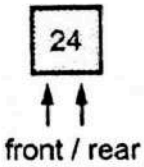


Step 11 :

i = delqatend (& dq)

Step 12 :

i = addatend (& dq, 24)



Review Questions

1. *Define dequeue and give its example.* **SPPU : May-10, Marks 2**
2. *What is dequeue ? Write pseudo code to perform insetion and deletion of element in a linked implementation of dequeue.* **SPPU : May-13, Marks 10**
3. *Write short note on – Double ended queue.* **SPPU : Dec-17, Marks 2, Dec-19, Marks 4**

3.16 Priority Queue

SPPU : May-10,13, Dec-10,11,15,19, Marks 10

The priority queue is a data structure having a collection of elements which are associated with specific ordering. There are two types of priority queues -

1. Ascending priority queue
2. Descending priority queue

Application of Priority Queue

1. The typical example of priority queue is scheduling the jobs in operating system. Typically operating system allocates priority to jobs. The jobs are placed in the queue and position1 of the job in priority queue determines their priority. In operating system there are three kinds of jobs. These are real time jobs, foreground jobs and background jobs. The operating system always schedules the real time jobs first. If there is no real time job pending then it schedules foreground jobs. Lastly if no real time or foreground jobs are pending then operating system schedules the background jobs.
2. In network communication, to manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling, to manage the discrete events the priority queue is used.

Types of Priority Queue

The elements in the priority queue have specific ordering. There are two types of priority queues –

1. **Ascending Priority Queue** – It is a collection of items in which the items can be inserted arbitarily but only smallest element can be removed.
2. **Descending Priority Queue** – It is a collection of items in which insertion of items can be in any order but only largest element can be removed.

In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list. The data structure heap is used to implement the priority queue effectively.

ADT for Priority Queue

Various operations that can be performed on priority queue are -

1. Insertion 2. Deletion 3. Display

Hence the ADT for priority queue is given as below -

Instances :

P_que[MaX] is a finite collection of elements associated with some priority.

Precondition :

The front and rear should be within the maximum size MAX.

Before insertion operation , whether the queue is full or not is checked.

Before any deletion operation, whether the queue is empty or not is checked.

Operations :

1. Create() – The queue is created by declaring the data structure for it.
2. insert() – The element can be inserted in the queue
3. delet() – If the priority queue is ascending priority queue then only smalles element is deleted each time. And if the priority queue is descending priorit queue then only largest element is deleted each time.
4. Display() – The elements of queue are displayed from front to rear.

The implementation of priority queue using arrays is as given below -

1. Insertion operation

While implementing the priority queue we will apply a simple logic. That is wh inserting the element we will insert the element in the array at the proper position. F example if the elements are placed in the queue as -

9	12			
que[0]	que[1]	que[2]	que[3]	que[4]
front	rear			

And now if an element 8 is to be inserted in the queue then it will be at location as -

8	9	12		
que[0]	que[1]	que[2]	que[3]	que[4]
front		rear		

If the next element comes as 11 then the queue will be -

8	9	11	12	
que[0]	que[1]	que[2]	que[3]	que[4]
front			rear	

The C++ function for this operation is as given below -

```
int Pr_Q::insert(int rear,int front)
{
    int item,j;
    cout<<"\nEnter the element: ";
    cin>>item;
    if(front ==-1)
            front++;
    j=rear;
    while(j>=0 && item<que[j])
```

```
{
    que[j+1]=que[j];
    j--;
    }
    que[j+1]=item;
    rear=rear+1;
    return rear;
}
```

2. Deletion operation

In the deletion operation we are simply removing the element at the front.

For example if queue is created like this –

8	9	11	12	
que[0] front	que[1]	que[2]	que[3] rear	que[4]

Then the element at que[0] will be deleted first

~~8~~	9	11	12	
~~que[0]~~	que[1] front	que[2]	que[3] rear	que[4]

and then new front will be qsue[1].

The deletion operation in C++ is as given below -

```
int Pr_Q::delet(int front)
{
    int item;
    item=que[front];
    cout<<"\n The item deleted is  "<<item;
    front++;
    return front;
}
```

The complete implementation of priority queue is as given below -

C++ Program

```
/*****************************************************
Program for implementing the ascending priority Queue
*****************************************************/
/*Header Files*/
#include<iostream.>
#define SIZE 5
using namespace std;
class Pr_Q
```

```
    {
     private:
            int que[SIZE];
     public:
            int rear,front;
            Pr_Q();
            int insert(int rear,int front);
            int Qfull(int rear);
            int delet(int front);
            int Qempty(int rear,int front);
            void display(int rear,int front);
    };
    Pr_Q::Pr_Q()
    {
     front=0;
     rear=-1;
    }
    int Pr_Q::insert(int rear,int front)
    {
     int item,j;
     cout<<"\nEnter the element: ";
     cin>>item;
     if(front ==-1)
          front++;
     j=rear;
     while(j>=0 && item<que[j])
     {
      que[j+1]=que[j];
      j--;
     }
     que[j+1]=item;
     rear=rear+1;
     return rear;
}
     int Pr_Q::Qfull(int rear)
     {
     if(rear==SIZE-1)
      return 1;
     else
     return 0;
     }

     int Pr_Q::delet(int front)
     {
      int item;
      item=que[front];
```

```
cout<<"\n The item deleted is "<<item;
front++;
return front;
}
int Pr_Q::Qempty(int rear,int front)
{
if((front==-1)||(front>rear))
return 1;
else
return 0;
}
void Pr_Q::display(int rear,int front)
{
int i;
cout<<"\n The queue is: ";
for(i=front;i<=rear;i++)
cout<<" "<<que[i];
}

int main(void)
{
int choice;
char ans;
Pr_Q obj;

do
{
cout<<"\n\t\t Priority Queue\n";
cout<<"\n Main Menu";
cout<<"\n1.Insert\n2.Delete\n3.Display";
cout<<"\n Enter Your Choice: ";
cin>>choice;
switch(choice)
{
    case 1:if(obj.Qfull(obj.rear))
                cout<<"\n Queue IS full";
           else
                obj.rear=obj.insert(obj.rear,obj.front);
           break;
    case 2:if(obj.Qempty(obj.rear,obj.front))
                cout<<"\n Cannot delete element";
           else
                obj.front=obj.delet(obj.front);
           break;
    case 3:if(obj.Qempty(obj.rear,obj.front))
                cout<<"\n Queue is empty";
```

```
                else
                obj.display(obj.rear,obj.front);
                break;
  default:cout<<"\n Wrong choice: ";
          break;
  }
  cout<<"\n Do You Want TO continue?";
  cin>>ans;
 }while(ans=='Y' ||ans=='y');
 return 0;
}
```

Output

```
        Priority Queue

Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice: 1

Enter the element: 20

Do You Want TO continue?

        Priority Queue

Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice: 1

Enter the element: 30

Do You Want TO continue?

        Priority Queue
Main Menu
1.Insert
2.Delete
3.Display
 Enter Your Choice: 1

Enter the element: 10

Do You Want TO continue?
```

```
            Priority Queue
 Main Menu
 1.Insert
 2.Delete
 3.Display
 Enter Your Choice: 1

Enter the element: 40

Do You Want TO continue?

            Priority Queue
 Main Menu
 1.Insert
 2.Delete
 3.Display
 Enter Your Choice: 3

The queue is:  10 20 30 40
Do You Want TO continue?

            Priority Queue

 Main Menu
 1.Insert
 2.Delete
 3.Display
 Enter Your Choice: 2

The item deleted is  10
Do You Want TO continue?
```

Review Questions

1. *What are the applications of the data structure priority queue.* **SPPU : May-10, Marks 4**
2. *What is priority queue ? What is its use ? Give the function to add an element in priority queue.* **SPPU : Dec-15, Marks 6, May-10, Marks 10**
3. *Write a C program to implement priority queue.* **SPPU : Dec-10, Marks 8**
4. *Write a note on priority queues.* **SPPU : Dec-11, Marks 4**
5. *What is priority queue ? Explain insert and delete operations in detail using multidimensional array implementation.* **SPPU : May-13, Marks 8**
6. *Write short note on - Priority queue* **SPPU : Dec-19, Marks 8**

3.17 Applications of Queues

SPPU : May-11, 15, 16, Dec-12, Marks 6

1. In operating system for **scheduling jobs**, priority queues are used.
2. In Local Area Network (LAN) multiple computers share few printers. Then these **printers** accumulate jobs in a queue. Thus printers can process multiple print commands with the help of queues.
3. For **categorizing data**, queues are used.
4. In **simulation and modeling** queues are used.
5. In computer networks, while **broadcasting message**, the message packets are accumulated in queue. And then these packets are forwarded.

Example 3.16.5 *Specify which of the following applications would be suitable for a first-in-first-out queue and Justify your answer :*

i) A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first come, first-served basis.

ii) An inventory of parts is to be processed by part number.

iii) A dictonary of words used by spelling checker is to be created.

iv) Customers are to take numbers at a bakery and be served in order when their numbers come-up.

SPPU : May-11, Dec-12, Marks 4, May-15, Marks 6

Solution :

i) No, this application will not be suitable for first come first serve queue because sometimes some serious patients may be come and he/she needs doctors in urgent.

ii) Yes, for this application queue is suitable because here the part no must be stored in specific manner (order).

iii) No, because words can be choose in random order.

iv) Yes, for bakery customers must be serve if first come first serve basis.

Review Question

1. *List down applications of queue.* SPPU : Dec-12, May-13, Marks 4

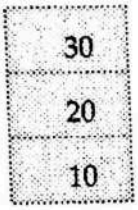
3.18 Double Ended Queue as Stack and Queue

We can make double ended queue to work as stack as well as queue. Following table lists out various operations of doubly ended queue, these operations can be used as stack's push and pop operations or queues insert and delete operations.

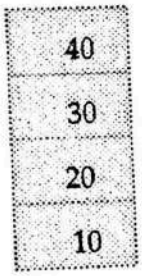
Dequeue	Stack	Queue
display()	display()	display()
QEmpty()	For Stack Full Condition	For Queue Full Condition
Qfull()	For Stack Empty Condition	For Queue Empty Condition
Insert_front()	-	-
Insert_rear()	push()	insert_queue()
delete_rear()	pop()	-
delete_front()	-	Delete_queue()

Example : Stack

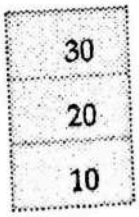
Suppose stack contains,



Push (40)

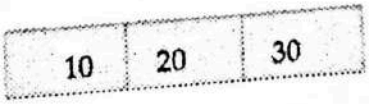


Pop()

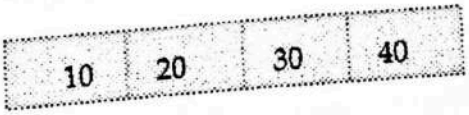


Example Queue

Suppose queue contains



Insert_queue(40)



Delete_queue()

20	30	40

C++ Program

```
#include<iostream>
using namespace std;
#define size 5
class DQ_QS
{
        private:
                int dque[size];
        public:
        int front,rear;
        DQ_QS()
        {
                front=-1;
                rear=-1;
                for(int i=0;i<size;i++)
                        dque[i]=-1;
        }
        /* The Qfull function    */
        int Qfull()
        {
                if(rear==size-1)
                        return 1;
                else
                        return 0;
        }
        /*  The Qempty   */
        int Qempty()
        {
                if((front>rear)||(front==-1&&rear==-1))
                        return 1;
                else
                        return 0;
        }
        /* The Qfull function */
        int insert_rear(int item)
        {
                if(front==-1&&rear==-1)
                        front++;
                dque[++rear]=item;
                return rear;
        }
        /* The delete_front function*/
```

```
int delete_front()
{
        int item;
        if(front==-1)
                front++;
        item=dque[front];
        dque[front]=-1;
        front++;
        return item;
}
int insert_front(int item)
{
        int i,j;
        if(front==-1)
                front++;
        i=front-1;
        while(i>=0)
        {
                dque[i+1]=dque[i];
                i--;
        }
        j=rear;
        while(j>=front)
        {
                dque[j+1]=dque[j];
                j--;
        }
        rear++;
        dque[front]=item;
        return front;
}
int delete_rear()
{
        int item;
        item=dque[rear];
        dque[rear]=-1;/*logical deletion*/
        rear--;
        return item;
}
void display()
{
        int i;
        for(i=front;i<=rear;i++)
                printf(" %d",dque[i]);
}
};
int main()
```

```
{
    int choice,i,item;
    char ans;
    ans='y';
    DQ_QS obj;
    cout<<"\n\t\t Program For doubly ended queue As Stack and Queue";
    do
    {
        cout<<"\n\t ***** Stack Operations ***** ";
        cout<<"\n1.Push\n2.Pop \n3.Display Stack ";
        cout<<"\n\t ***** Queue Operations ***** ";
        cout<<"\n4.Insert_Queue \n5.Delete_Queue \n6.Display Queue";
        cout<<"\n Enter Your choice: ";
        cin>>choice;
        switch(choice)
        {
            case 1:if(obj.Qfull())
                        printf("\n Stack is full");
                    else
                    {
                        cout<<"\n Enter The item to be pushed: ";
                        cin>>item;
                        obj.rear=obj.insert_rear(item);
                    }
            break;
            case 2:if(obj.Qempty())
                        printf("\n Stack is Empty");
                    else
                    {
                        item=obj.delete_rear();
                        cout<<"\n The item deleted from stack is: "<<item;
                    }
            break;
            case 3:cout<<"\n Stack is ...\n";
                    obj.display();
            break;
            case 4:if(obj.Qfull())
                        cout<<"\n Queue is full";
                    else
                    {
                        cout<<"\n Enter The item to be inserted: ";
                        cin>>item;
                        obj.rear=obj.insert_rear(item);
                    }
            break;
            case 5:if(obj.Qempty())
                        cout<<"\n Queue is Empty";
```

```
                    else
                    {
                            item=obj.delete_front();
                            cout<<"\n The item deleted from queue is: "<<item;
                    }
                break;
                case 6:cout<<"\n Queue is ...\n";
                        obj.display();
                break;
                case 7:exit(0);
            }
            cout<<"\n Do You Want To Continue? ";
            cin>>ans;
        }while(ans==='y'||ans==='Y');
        return 0;

}
```

Output

```
        Program For doubly ended queue As Stack and Queue
    ***** Stack Operations *****
1.Push
2.Pop
3.Display Stack
    ***** Queue Operations *****
4.Insert_Queue
5.Delete_Queue
6.Display Queue
Enter Your choice: 1

Enter The item to be pushed: 10

Do You Want To Continue?
    ***** Stack Operations *****
1.Push
2.Pop
3.Display Stack
    ***** Queue Operations *****
4.Insert_Queue
5.Delete_Queue
6.Display Queue
Enter Your choice: 1

Enter The item to be pushed: 20

Do You Want To Continue?
    ***** Stack Operations *****
1.Push
2.Pop
```

```
3.Display Stack
        ***** Queue Operations *****
4.Insert_Queue
5.Delete_Queue
6.Display Queue
 Enter Your choice: 1

 Enter The item to be pushed: 30

 Do You Want To Continue?
        ***** Stack Operations *****
 1.Push
 2.Pop
 3.Display Stack
        ***** Queue Operations *****
 4.Insert_Queue
 5.Delete_Queue
 6.Display Queue
 Enter Your choice: 3

 Stack is ...
  10  20  30
 Do You Want To Continue?
        ***** Stack Operations *****
 1.Push
 2.Pop
 3.Display Stack
        ***** Queue Operations *****
 4.Insert_Queue
 5.Delete_Queue
 6.Display Queue
 Enter Your choice: 4

 Enter The item to be inserted: 11

 Do You Want To Continue?
        ***** Stack Operations *****
 1.Push
 2.Pop
 3.Display Stack
        ***** Queue Operations *****
 4.Insert_Queue
 5.Delete_Queue
 6.Display Queue
 Enter Your choice: 4

 Enter The item to be inserted: 22
```

```
Do You Want To Continue?
      ***** Stack Operations *****

1.Push
2.Pop
3.Display Stack
      ***** Queue Operations *****

4.Insert_Queue
5.Delete_Queue
6.Display Queue
Enter Your choice: 5

The item deleted from queue is: 10
Do You Want To Continue?
      ***** Stack Operations *****

1.Push
2.Pop
3.Display Stack
      ***** Queue Operations *****

4.Insert_Queue
5.Delete_Queue
6.Display Queue
Enter Your choice: 6

Queue is ...
 20  30  11  22
Do You Want To Continue?n
```

4 Trees

Syllabus

ree : Trees and binary trees - concept and terminology, Expression tree, Binary tree as an ADT, inary search tree, Recursive and Non recursive algorithms for binary tree traversals ,Binary search ee as ADT (Insert, Search, Delete, level wise Display).

***hreaded binary tree :** Concept of threaded binary tree (inorder, preorder and postorder). Preorder nd In-order traversals of in-order threaded binary tree, Applications of trees.*

Contents

4.1 Trees and Binary Trees Concept and Terminology ..... **Dec.-10, May-14,** ..... Marks 6

4.2 Representation of Binary Tree

4.3 Conversion of General Tree into Binary Tree ..... **May-11,16, Dec.-14,** ..... Marks 6

4.4 Binary Trees as an ADT

4.5 Binary Tree Traversals ..... **May-13,14,18,19, Dec.-11,12,15,18,** ..... Marks 6

4.6 Binary Search Trees ..... **Dec.-14,** ..... Marks 4

4.7 Binary Search Tree as ADT ..... **Dec.-06,12, May-10,11,13,** .. Marks 8

4.8 Threaded Binary Tree ..... **Dec.-10,18, May-16,** ..... Marks 8

4.9 Applications of Trees

4.10 Expression Tree ..... **May-10,11, Dec.-11,** ..... Marks 10

4.1 Tree and Binary Tree Concepts and Terminologies

SPPU : Dec.-10, May-14, Marks 6

Definition of a Tree :

A tree is a finite set of one or more nodes such that -

i) **Root node :** It is a specially designated node.

ii) There are remaining n nodes which can be partitioned into disjoint sets $T_1, T_2, T_3, \ldots T_n$ where $T_1, T_2, T_3, \ldots T_n$ are called subtrees of the root.

The concept of tree can be represented by following figure -

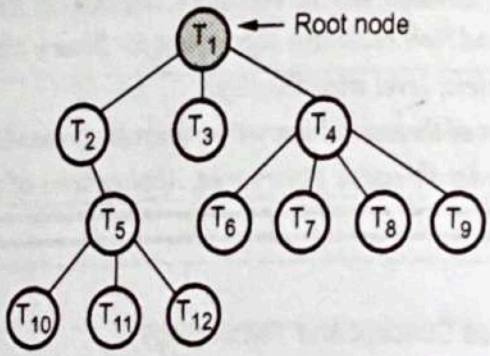


Fig. 4.1.1 Tree

4.1.1 Basic Tree Concepts

Let us get introduced with some of the definitions or terms which are normally used.

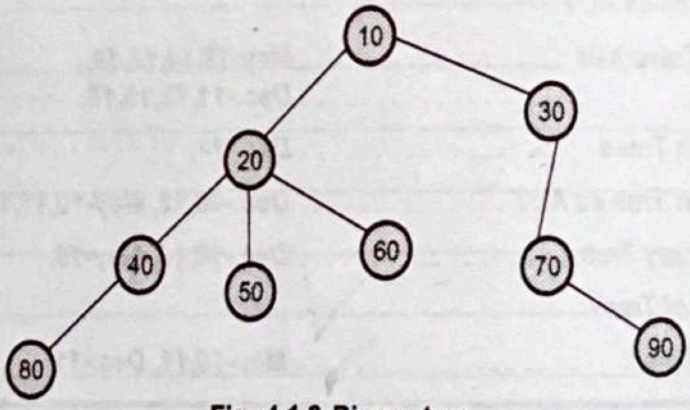


Fig. 4.1.2 Binary tree

From Fig. 4.1.2,

1. Root

Root is a unique node in the tree to which further subtrees are attached. For above given tree, node 10 is a root node.

2. Parent node

The node having further sub-branches is called parent node. In Fig. 4.1.3, 20 is parent node of 40, 50 and 60.

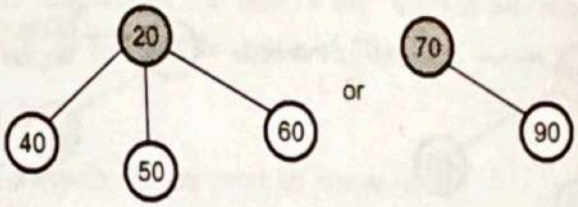


Fig. 4.1.3 Binary tree representing parent node

3. Child nodes

The child nodes in above given tree are marked as shown below.

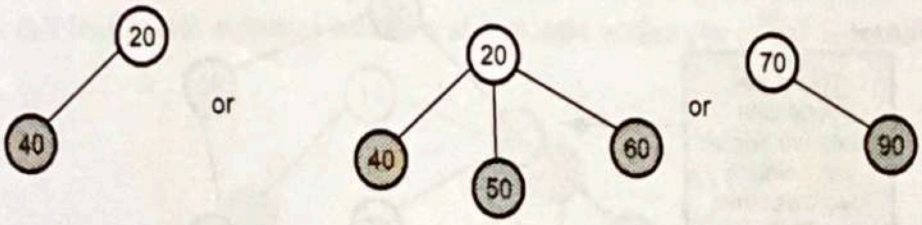


Fig. 4.1.4 Binary tree representing child node

4. Leaves

These are the terminal nodes of the tree.

For example

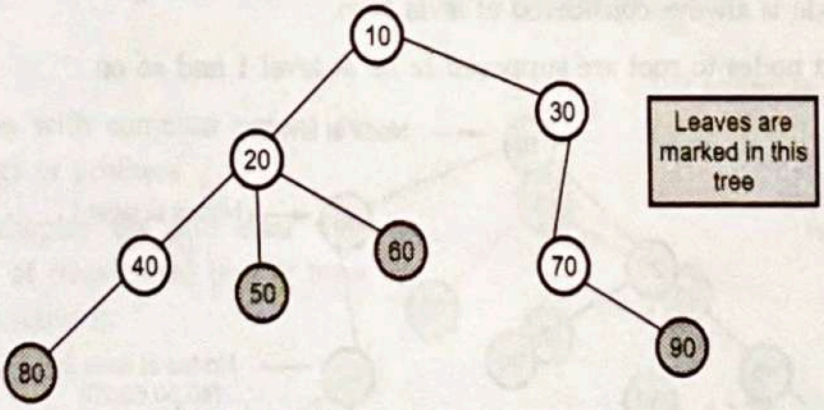


Fig. 4.1.5 Binary tree with left nodes

5. Degree of the node

The total number of sub-trees attached to that node is called the degree of a node.

For example

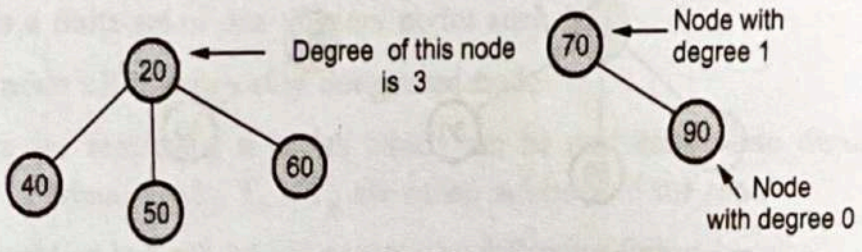


Fig. 4.1.6 Binary tree

6. Degree of tree

The maximum degree in the tree is degree of tree.

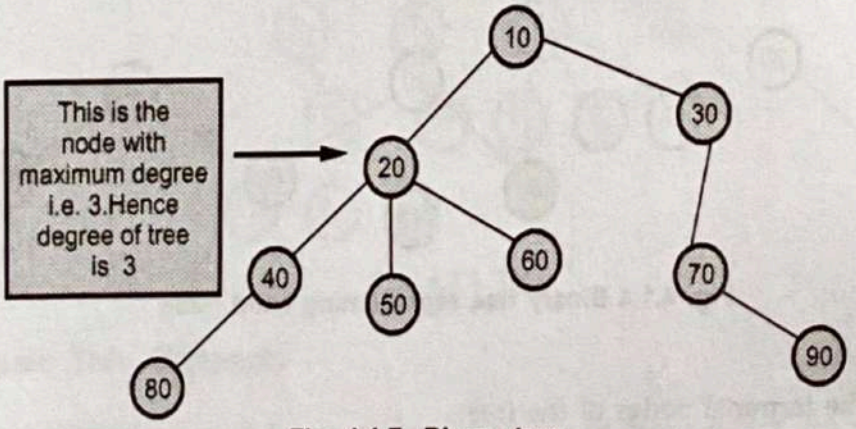


Fig. 4.1.7 Binary tree

7. Level of the tree

The root node is always considered at level zero.

The adjacent nodes to root are supposed to be at level 1 and so on.

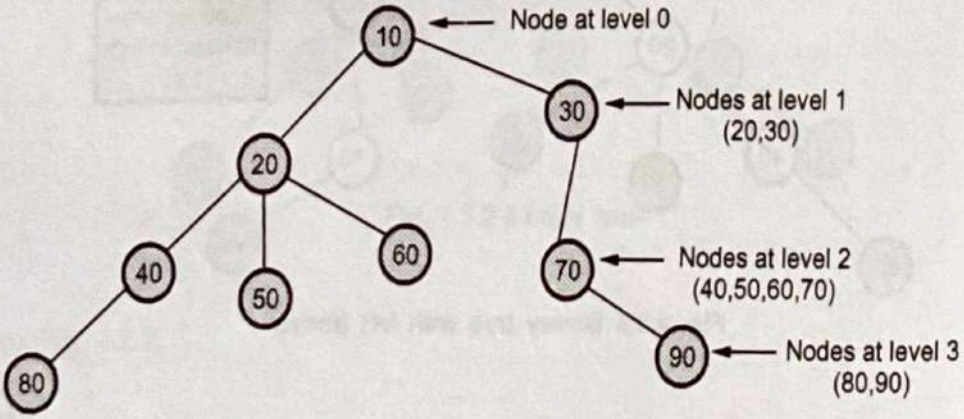


Fig. 4.1.8 Binary tree in which levels are shown

8. Height of the tree

The maximum level is the height of the tree. In Fig. 4.1.8 the height of tree is 3. Sometimes height of the tree is also called **depth** of tree.

9. Predecessor

While displaying the tree, if some particular node occurs previous to some other node then that node is called **predecessor** of the other node.

For example : While displaying the tree in Fig. 4.1.8 if we read node 20 first and then if we read node 40, then 20 is a predecessor of 40.

10. Successor

Successor is a node which occurs next to some node.

For example : While displaying tree in Fig. 4.1.8 if we read node 60 after reading node 20 then 60 is called successor of 20.

11. Internal and external nodes

Leaf node means a node having no child node. As leaf nodes are not having further links, we call leaf nodes **external nodes** and non leaf nodes are called **internal nodes.**

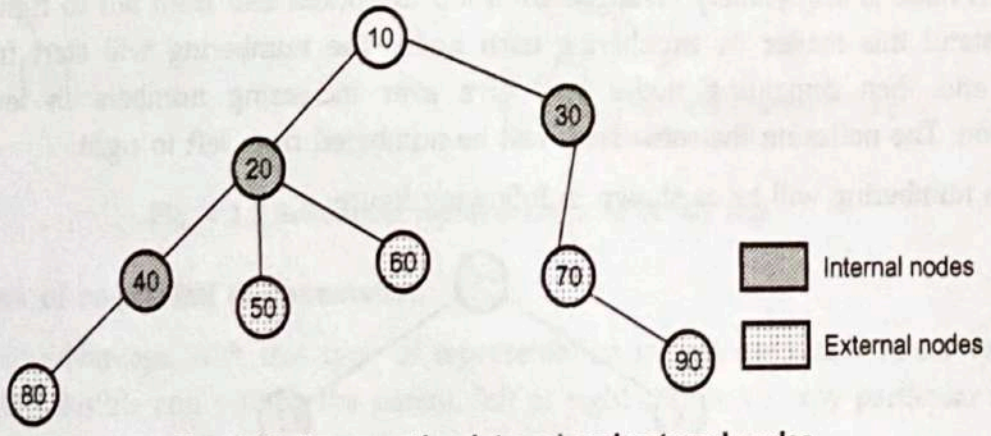


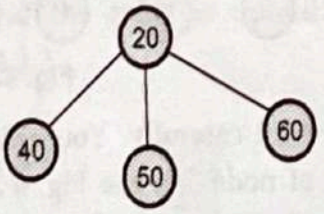
Fig. 4.1.9 Representing internal and external nodes

12. Sibling

The nodes with common parent are called siblings or brothers.

In this chapter we will deal with special type of trees called binary trees. Let us understand it.

For example



The nodes 40, 50 and 60 are siblings of each other.

Fig. 4.1.10

Review Questions

1. *With examples, define the following terms with respect to trees*
   i) Skewed binary trees ii) Complete binary tree iii) Ancestor, descendant leaf node.

SPPU : Dec.-10, Marks 6

2. *With examples define following terms :*

   i) Completer binary tree ii) Strictly binary tree iii) Predecessor and Successor.

   SPPU : May-14, Marks 6

   Hint : *For complete and strict binary tree refer section 2.5.*

4.2 Representation of Binary Tree

There are two ways of representing the binary tree.

1. Sequential representation.
2. Linked representation.

Let us see these representations one by one.

1. Sequential representation of binary trees or array representation :

Each node is sequentially arranged from top to bottom and from left to right. Let us understand this matter by numbering each node. The numbering will start from root node and then remaining nodes will give ever increasing numbers in level wise direction. The nodes on the same level will be numbered from left to right.

The numbering will be as shown in following figure.

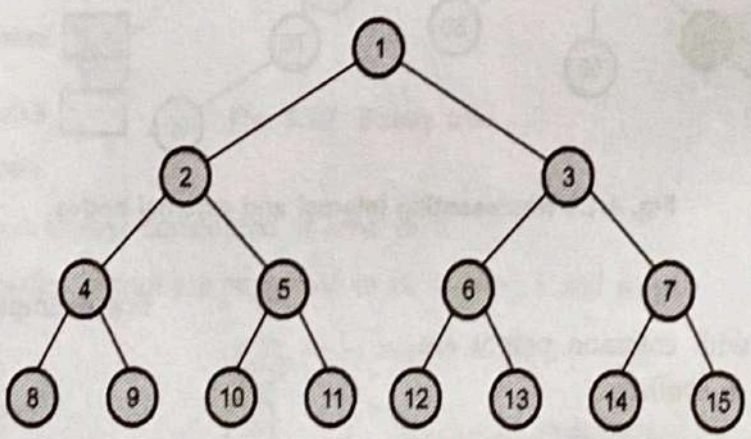


Fig. 4.2.1

Now, observe Fig. 4.2.2 carefully. You will get a point that a binary tree of depth n having $2^n - 1$ number of nodes. In the Fig. 4.2.2 the tree is having the depth 4 and total number of nodes are 15. Thus remember that in a binary tree of depth n there will be maximum $2^n - 1$ nodes. And so if we know the maximum depth of the tree then we can represent binary tree using arrays data structure. Because we can then predict the maximum size of an array that can accommodate the tree.

Thus array size can be > = n. The root will be at index 0. Its left child will be at index 1, its right child will be at index 2 and so on. Another way of placing the elements in the array is by applying the formula as shown below.

When n = 0 the root node will placed at 0th location

Parent(n) = floor (n–1)/2

Left(n) = (2 n+1)

Right(n) = (2 n+2).

Where n>0

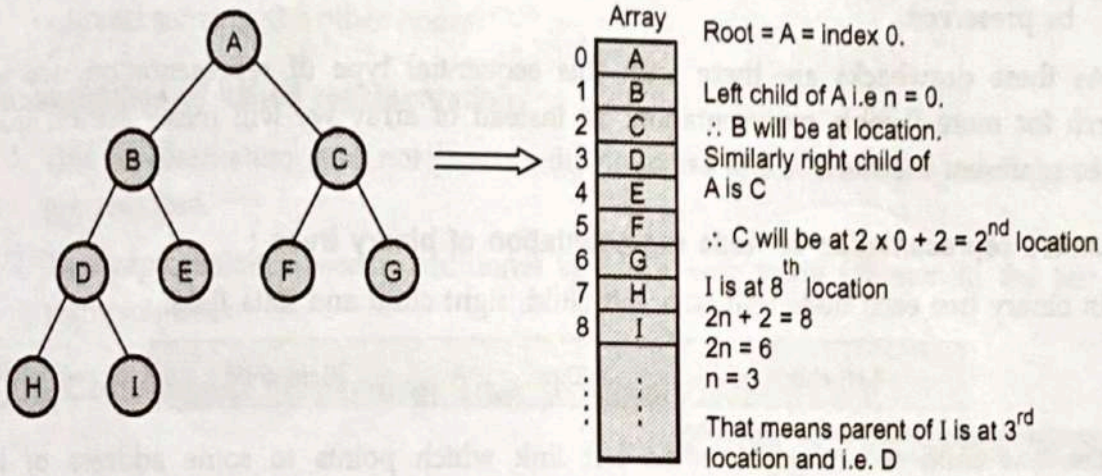


Fig. 4.2.2 Sequential representation of binary tree

Advantages of sequential representation

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent, left or right children of any particular node is fast because of the random access.

Disadvantages of sequential representation

1. The major disadvantage with this type of representation is wastage of memory.

For example : In the skewed tree half of the array is unutilized. You can easily understand this point simply by seeing Fig. 4.2.3.

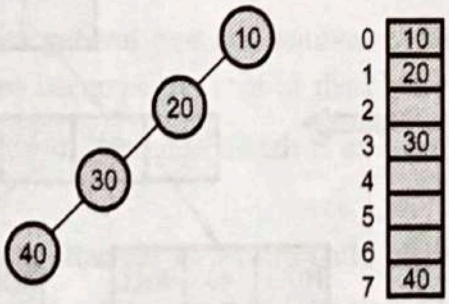


Fig. 4.2.3.

2. In this type of representation the maximum depth of the tree has to be fixed because we have already decided the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes have to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

2. Linked representation or node representation of binary trees :

In binary tree each node will have left child, right child and data field.

Left child	Data	Right child

The left child is nothing but the left link which points to some address of left sub-tree whereas right child is also a right link which points to some address of right sub-tree. And the data field gives the information about the node. Let us see the 'C' structure of the node in a binary tree.

```
typedef struct node
{
 int data;
 struct node *left;
 struct node *right;
}bin;
```

The tree with linked representation is as shown below.

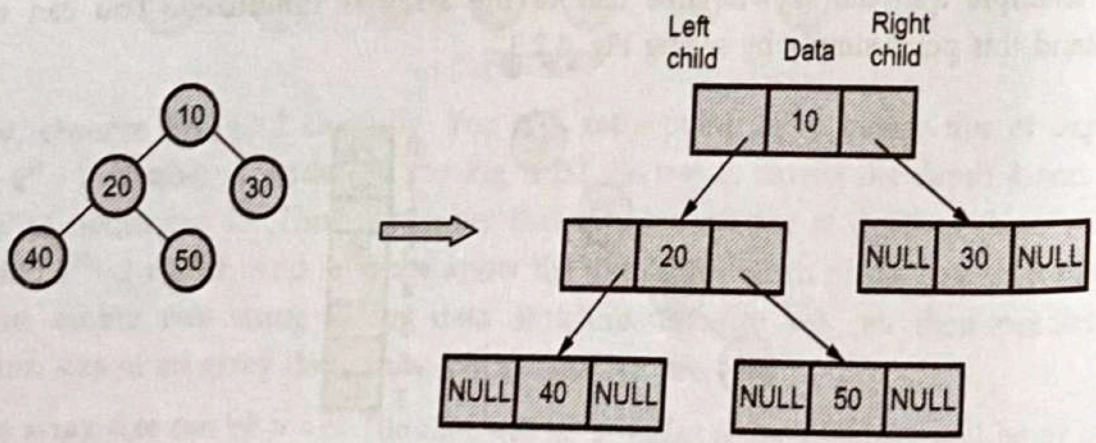


Fig. 4.2.4 Linked representation of binary tree

Advantages of linked representation

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as many memory (nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.
2. Insertions and deletions which are the most common operations can be done without moving the other nodes.

Disadvantages of linked representation

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

4.3 Conversion of General Tree to Binary Tree

SPPU : May-11,16, Dec.-14, Marks 6

General trees are those trees in which there could be any number of nodes that can be attached as a child nodes. The number of subtrees for each node may be varying. For example - Following tree is a **General tree**.

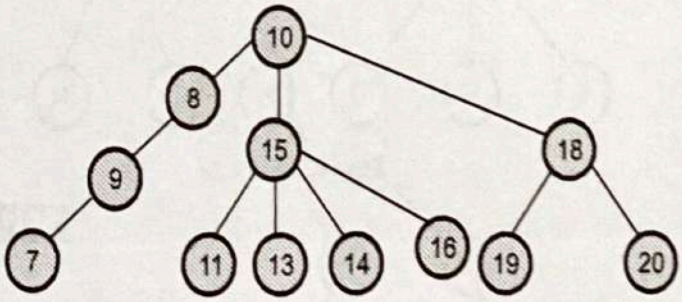


Fig. 4.3.1 General tree

We can convert the given general tree into equivalent binary tree as follows -

i) The root of general tree becomes the root of the binary tree.

ii) Find the first child node of the node attach it as a left child to the current node in binary tree.

iii) The right sibling can be attached as a right child of that node.

Let us convert the given general tree in Fig. 4.3.2 into equivalent binary tree.

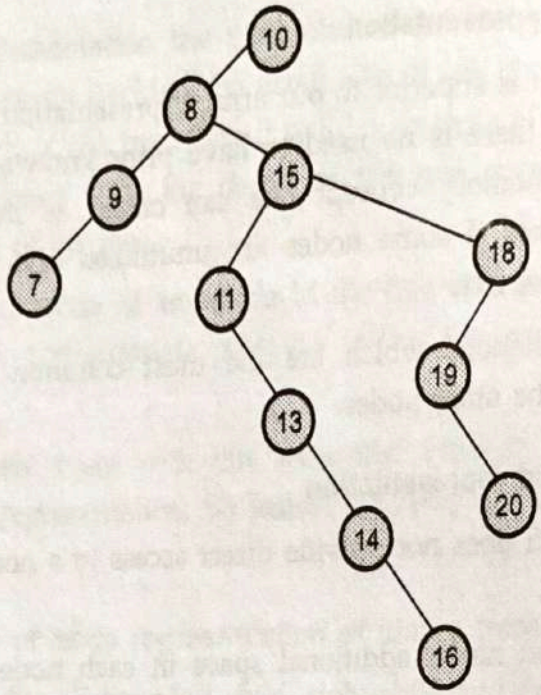


Fig. 4.3.2 Binary tree

Example 4.3.1 *Convert the following generalized tree into a binay tree.*

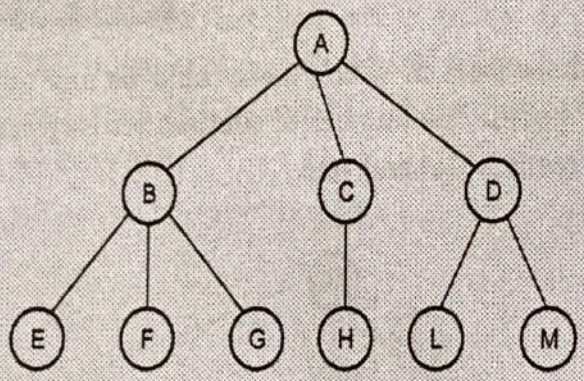


Fig. 4.3.3

SPPU : May-16, Marks 4

Solution :

For converting general tree into binary tree following rules are applied –

i) Root of general tree is a root of binary tree.

ii) The first left child of general tree is a left child of binary tree.

iii) The right sibling can be attached as a right child of that node.

The converted binary tree is shown in Fig. 4.3.4.

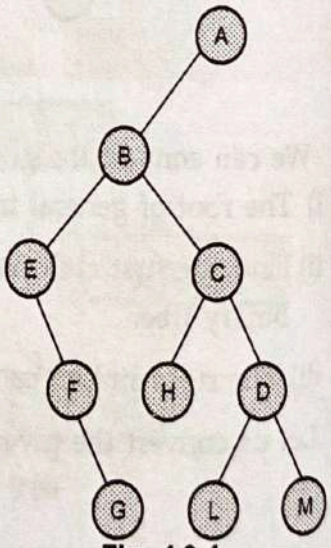


Fig. 4.3.4

Example 4.3.2 *List down the steps to convert general tree to binary tree ? Convert the given general tree to binary tree -*

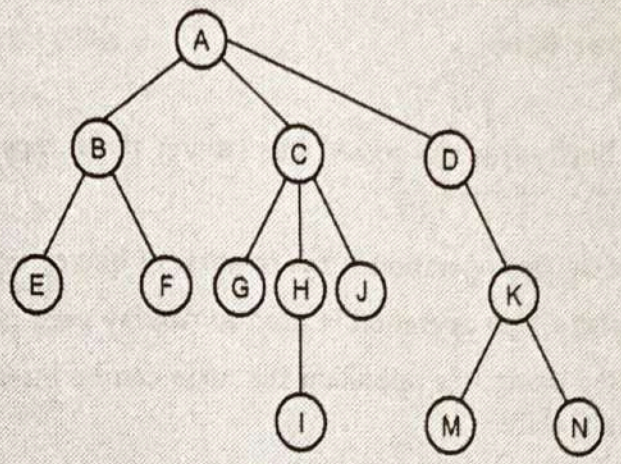


Fig. 4.3.5

SPPU : Dec.-14, Marks 6

Solution : Refer Example 4.3.1.

The equivalent binary tree will be

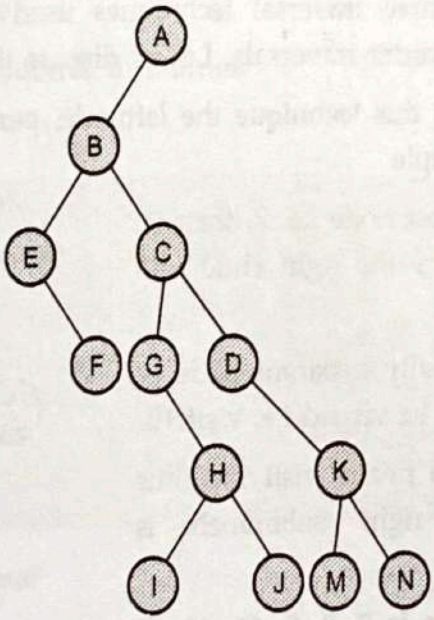


Fig. 4.3.6

Review Question

1. *What is the necessity of converting a tree into binary tree ? Show the conversion of tree with at least 12 nodes into binary tree and list down the steps of the same.*

SPPU : May-11, Marks 6

4.4 Binary Trees as an ADT

The ADT for binary tree is

AbstractDataType Btree

{

Instances : Binary tree is a rooted tree having at the most two child nodes.

Operations :

1. **Create :** This operation is for creation of binary tree.
2. **Display :** This operation is used to display each node in the tree.
3. **Insert :** Using this operation the node can be inserted by attaching it as left or right child.
4. **Delete :** The node having left child or right child or both can be deleted.

}

4.5 Recursive and Non-recursive Algorithms for Binary Tree Traversals

SPPU : May-13,14,18,19, Dec.-11,12,15,18, Marks 6

Concept : There are three traversal techniques used for binary tree. These are . Inorder, Preorder and Postorder traversals. Let us discuss these traversals in detail

1. **Inorder traversal :** In this technique the leftnode, parent and then right node visit is followed. For example

I) Here visit the leftmost node i.e. 7, then its parent node 8 then 9 the right child will be visited.

II) The root node (actually a parent node of left sub-branch) will be visited i.e. visit 10.

III) Visit 11 then 12 and finally visit 13. Thus the traversal of right subbranch is performed.

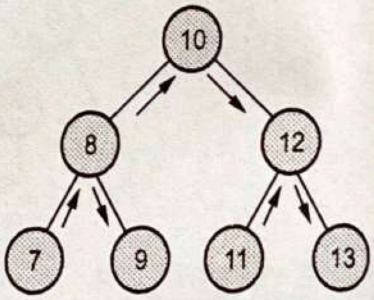


Fig. 2.8.1 Binary tree

Hence, Inorder sequence is **7, 8, 9, 10, 11, 12, 13**

2. **Preorder traversal :** In this method visit parent node the left and then right node.

I) Visit 10

II) Visit 8, then 7 finally 9

III) Visit 12 then 11 and finally 13.

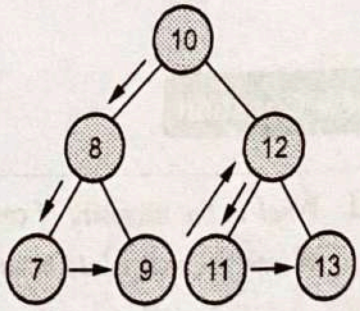


Fig. 2.8.2 Binary tree

Hence preorder traversal is **10, 8, 7, 9, 12, 11, 13.**

3. **Postorder traversal** : In this technique visit left node then right node and finally parent node.

I) Visit 7, then 9 and then 8.

II) Visit 11, then 13 and then 12.

III) Visit node 10.

The Postorder sequence is :

7, 9, 8, 11, 13, 12, 10.

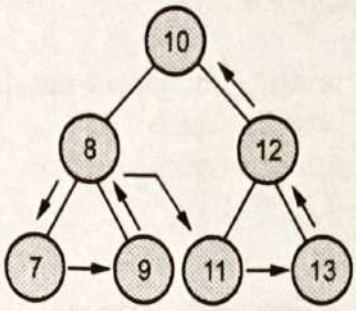


Fig. 2.8.3

Algorithms and C Functions

1. Recursive inorder traversal

Algorithm :

1. If tree is not empty then
   a. traverse the left subtree in inorder
   b. visit the root node
   c. traverse the right subtree in inorder

C Function

```
void inorder(node *temp)
{
 if(temp!=NULL)
 {
      inorder(temp->left);
      printf(" %d",temp->data);
      inorder(temp->right);
 }
}
```

2. Recursive preorder traversal

Algorithm :

1. If tree is not empty then
   a. visit the root node
   b. traverse the left subtree in preorder
   c. traverse the right subtree in preorder

C Function

```
void preorder(node *temp)
{
```

```
    if(temp!=NULL)
    {
        printf(" %d",temp->data);
        preorder(temp->left);
        preorder(temp->right);
    }
}
```

3. Recursive postorder traversal

Algorithm :

1. If tree is not empty then

a. traverse the left subtree in postorder

b. traverse the right subtree in postorder

c. visit the root node

C function

```
void postorder(node *temp)
{
    if(temp!=NULL)
    {
        postorder(temp->left);
        postorder(temp->right);
        printf(" %d",temp->data);
    }
}
```

Example 4.5.1 *Consider the following tree given in the problem. Show a Postorder, Preorder and In order Traversal of tree.*

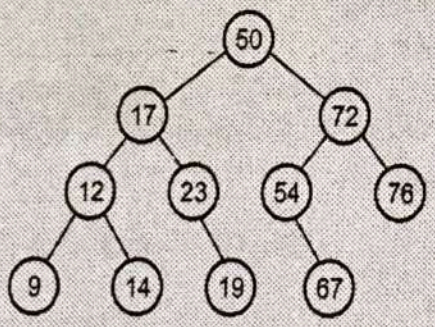


Fig. 4.5.4

SPPU : May-14, Marks 3

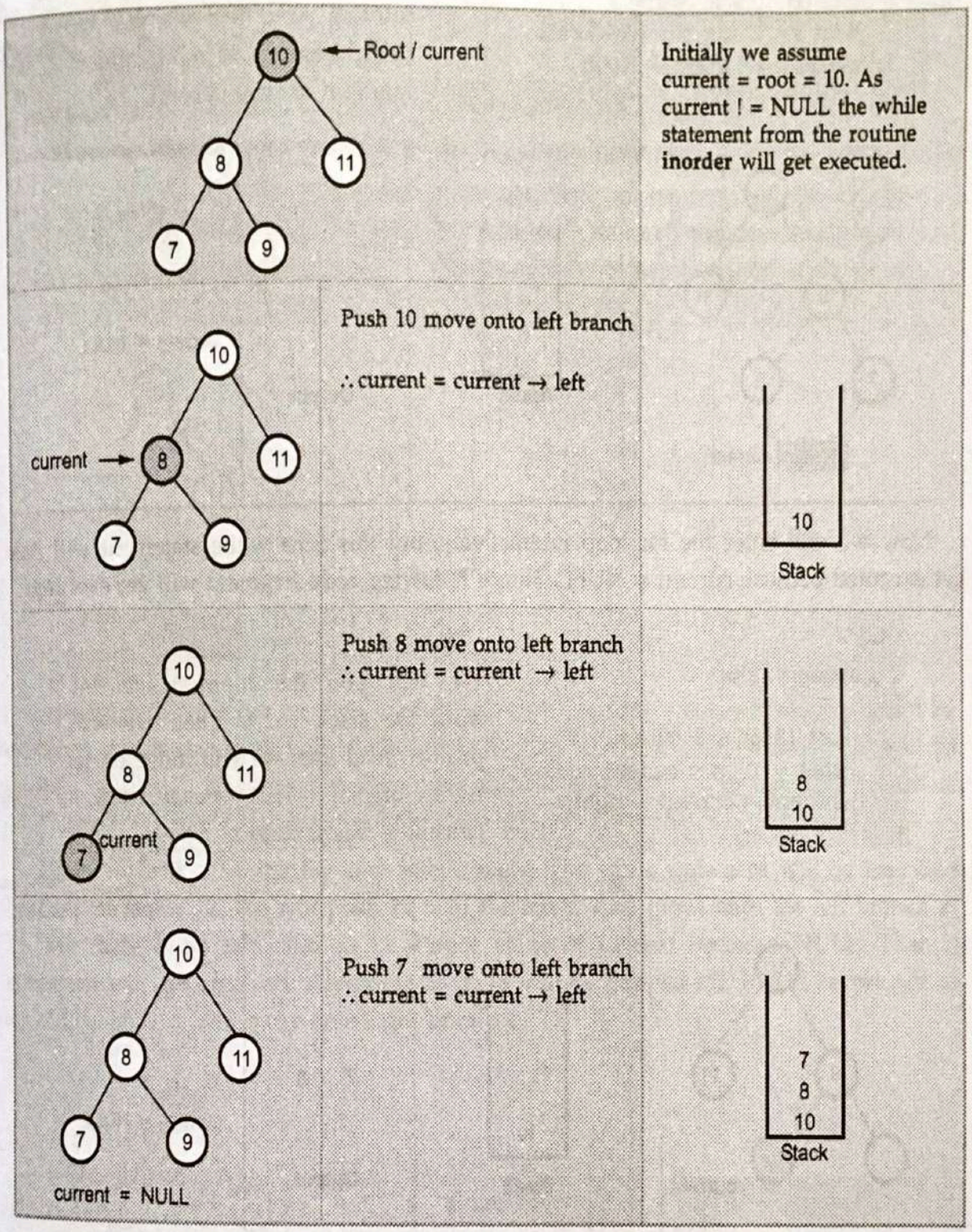
Solution : Preorder Sequence : 50, 17, 12, 9, 14, 23, 19, 72, 54, 67, 76.

Inorder Sequence : 9, 12, 14, 17, 23, 19, 50, 54, 67, 72, 76.

Postorder Sequence : 9, 14, 12, 19, 23, 17, 67, 54, 76, 72, 50.

4.5.1 Non Recursive Traversals

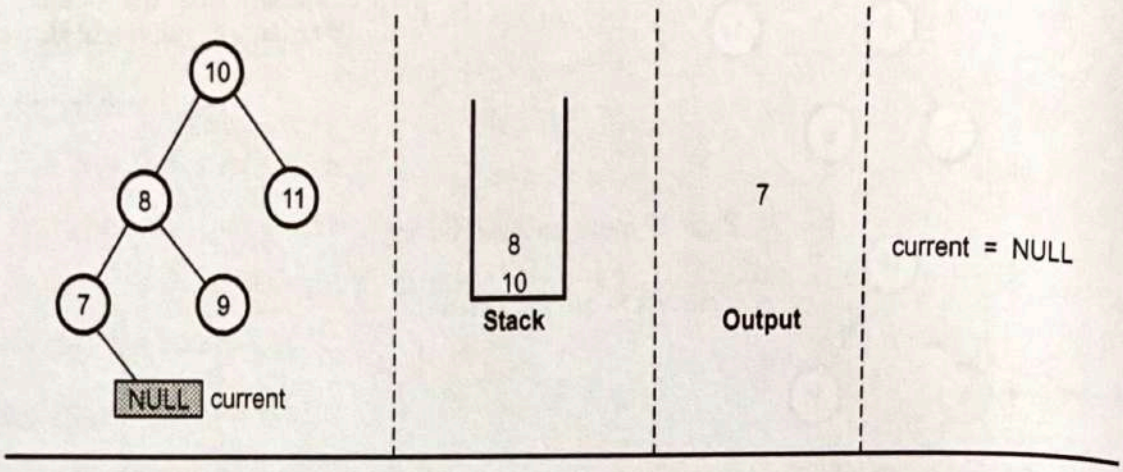
We will consider following tree as an example.



Now as **current** becomes **NULL**, we will come out of the while statements and following code fragment will get executed.

Code	Meaning

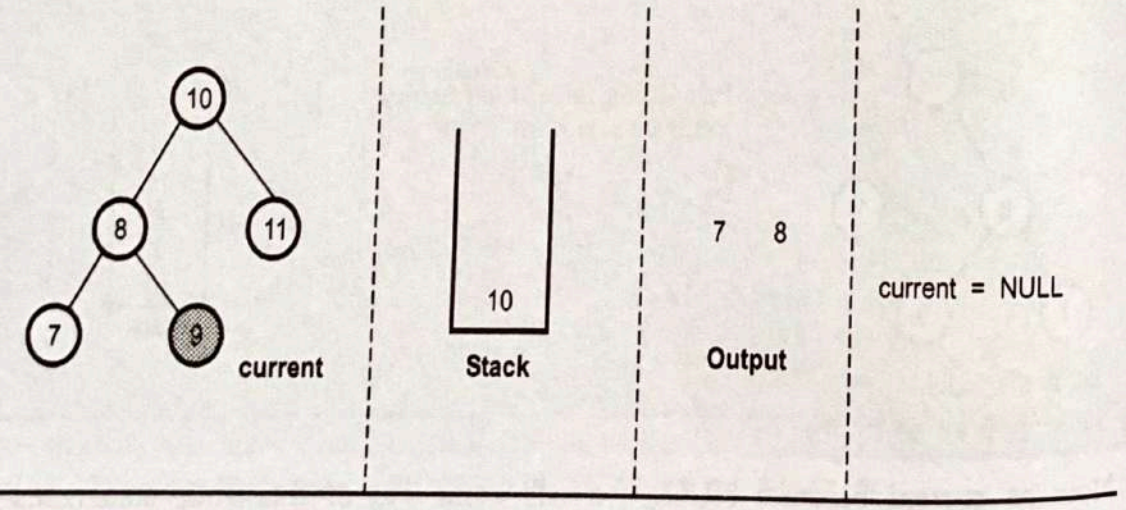
if (! stempty (top))
{
pop (&top,s, & current);
cout <<" " << current → data;
current = current → right;
}
	We will pop the topmost element from the stack i.e. 7. Then we will print it. And then we will move to right branch or right child of 7, calling it as new **current.**



Now we will enter the for loop interactively, but this time while statement will not get executed because current = NULL. Hence following code fragment will get executed.

Code	Meaning

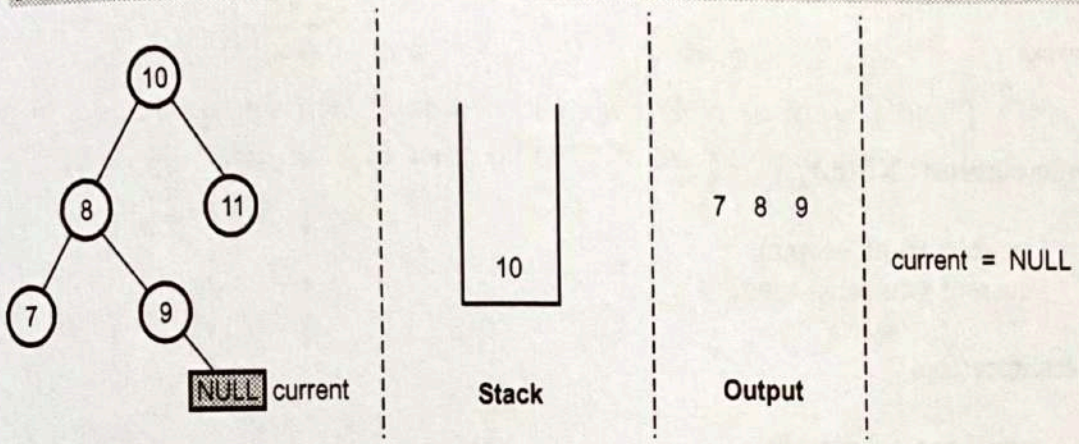
if (! stempty (top))
{
pop (&top,s, & current);
cout <<" " << current → data;
current = current → right;
}
	We will pop the topmost element from the stack i.e. 8. Then we will print it. And then we will move on to right branch. That mean new **current =** Right child of 8 = 9.



Now we will enter the for loop iteratively and while loop will be executed because current = 9. Hence 9 will be pushed onto the stack. And now current = current → left. As there is no left child for the node current; current = NULL. Hence following code fragment will be executed.

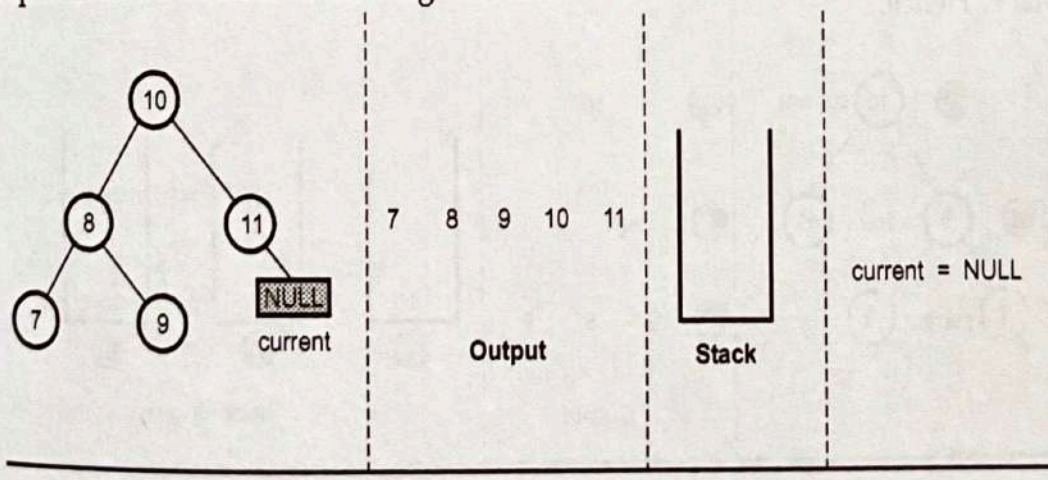
Code	Meaning

if (! stempty (top))
{
pop (&top,s, & current);
cout <<" " << current → data;
current = current → right;
}
	Now 9 which lies on the top of the stack will be popped off. It will be printed as an output. And now current will set to right child of 9 which is NULL.



Now we will enter the for loop interactively but this time while statement will not be executed. The if statement will be executed. According to this code fragment, 10 will be popped off, it will be printed as an output and we will move on to the right branch of 10. Hence now, current = 11 As an output we will get **7 8 9 10 .**

Again we will enter the for loop interactively. The while statement will be executed because current = 12. We will push 11 onto the stack. And move onto the left branch of 11. But there is no left child to 11. Hence value of **current** becomes **NULL**. Then if statement will get executed. According to it, 11 will be popped off, it will be printed as an output and we will move onto right branch of 11.

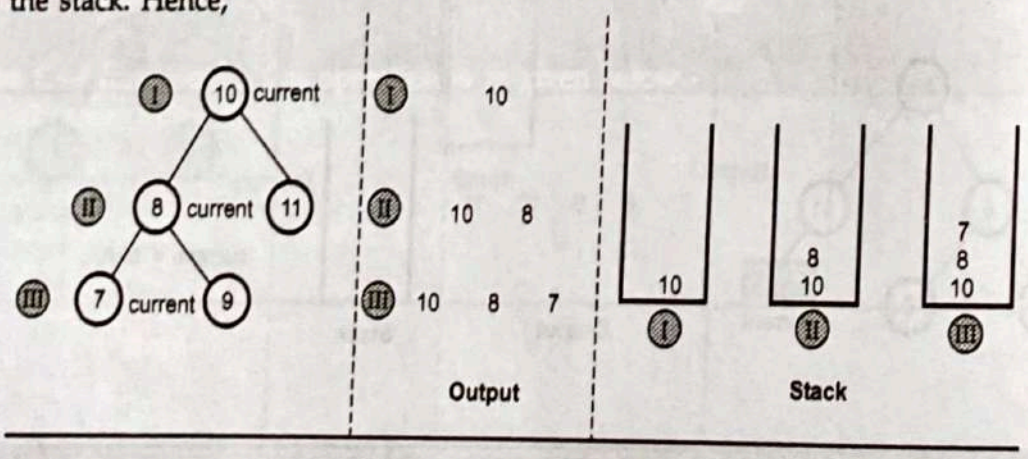


Again we will enter the for loop iteratively. But current = NULL ; hence while statements will not be executed, stack is empty; hence if statements will not be executed. Hence **else** will be executed, by which control returns to main function. Thus we will get **7 8 9 10 11** as an output of nonrecursive inorder traversal.

```
void TREE_CLASS::inorder(node *root)
{
 node *current,*s[10];
 int top=-1;
 if(root==NULL)
 {
        cout<<"\n Tree is empty\n";
        return;
 }
 current=root;
 for(;;)
 {
        while(current!=NULL)
        {
                push(current,&top,s);
                current=current->left;
        }
        if(!stempty(top))
        {
                pop(&top,s,&current);
                cout<<" "<<current->data;
                current=current->right;
        }
        else
                return;
 }
```

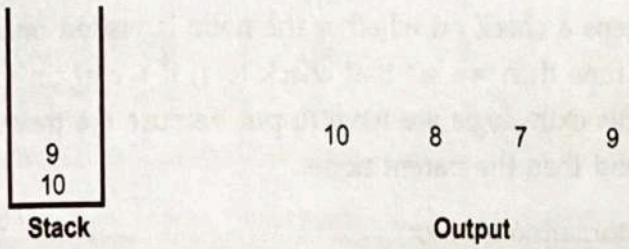
Non-recursive Preorder Traversal

The logic for preorder traversal is similar to the inorder traversal. But the only difference is that we will print the value of each visited current node before pushing it onto the stack. Hence,

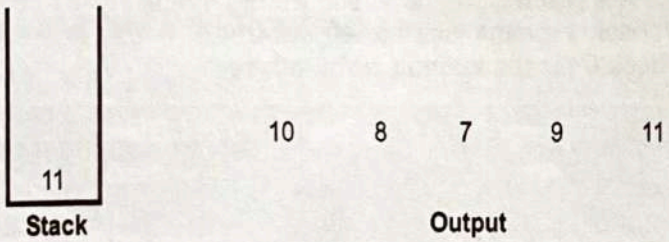


Visiting each node, calling it as current, printing it as an output, pushing current node onto the stack and moving on to the left child. These are the sequence of operation which we must follow at every stage. Above is the scenario for stage I, II and III.

Now if we move on to left branch of 7 we will get current = NULL. We will pop 7 check if it has any right. As 7 has no right child. We will pop the next element i.e. 8. As 8 has a right child i.e. 9, we will print 9 as an output and push it on to the stack. The stack will now be,

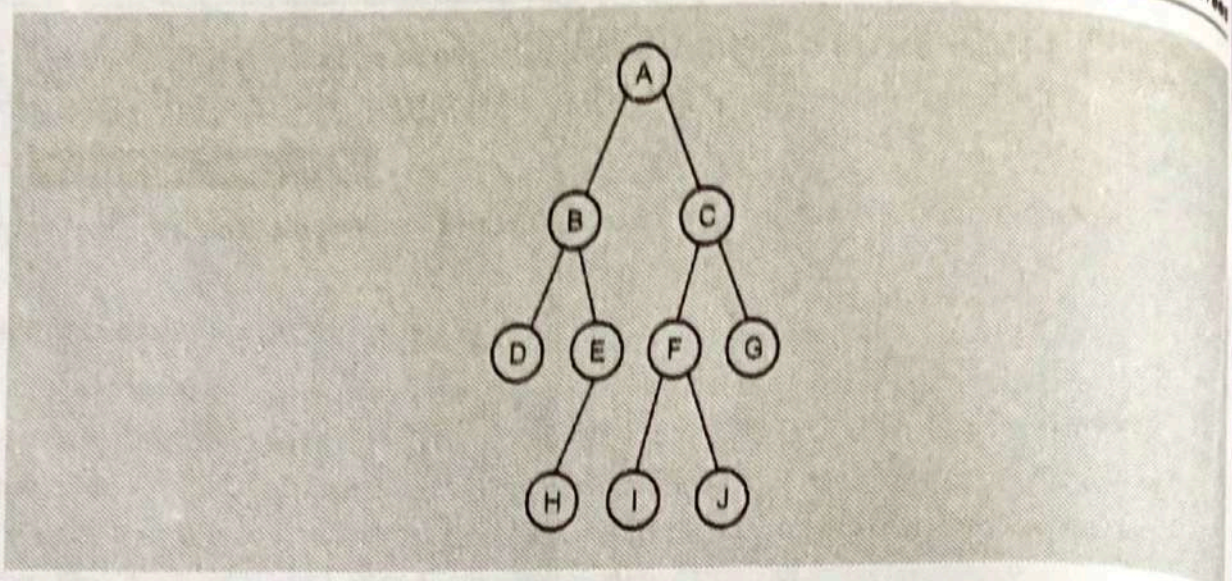


Then 9 will be popped off, but 9 has no right child so we will pop 10. The 10 has right child i.e. 11. Hence print 11 as an output and push it onto the stack.



Finally 11 will be popped off. The node 11 has no left or right child so nothing will be pushed. As a result we get **10 8 7 9 11** as an output for nonrecursive preorder traversal.

```
void TREE CLASS::preorder(node *root)
{
node *current,*s[10];
int top=-1;
if(root==NULL)
{
    cout<<"\n The Tree is empty\n";
    return;
}
current=root;
for(;;)
{
    while(current!=NULL)
    {
     cout<<" "<<current->data;
     push(current,&top,s);
     current=current->left;
    }
    if(!stempty(top))
    {
```



Solution : Preorder : A, B, D, E, H, C, F, I, J, G

Inorder : D, B, H, E, A, I, F, J, C, G

Postorder : D, H, E, B, I, J, F, G, C, A

Review Questions

1. *Write functions for non recursive inorder and preorder traversal for binary trees.*

SPPU : Dec.-11, May-13, Marks 8

2. *Implement binary tree as ADT.* **SPPU : Dec.-12, Marks 8**

3. *Write an algorithm for preorder traversal of binary tree and give suitable example.*

SPPU : May-18, Dec.-18, Marks 6

4.6 Binary Search Tree

SPPU : Dec.-14, Marks 4

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at left subtree < root node value < Right subtree values. See the Fig. 4.6.1.

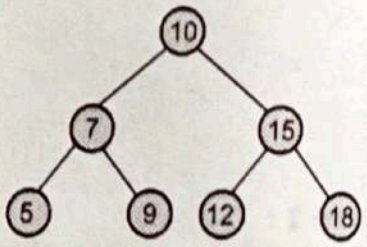


Fig. 4.6.1 Binary search tree

If you observe the Fig. 4.6.1 carefully, you will find that the left value < parent value < right value is followed throughout the tree.

4.6.1 Operations on Binary Search Tree

Various operations that can be performed on binary search tree are -

1. Insertion of a node in a binary tree.
2. Deletion of some element from the binary search tree.
3. Searching of an element in the binary tree.
4. Display of binary tree.

Let us discuss each operation one by one -

1. Insertion of a node in a binary tree

Algorithm :

1. Read the value for the node which is to be created, and store it in a node called **New**.
2. Initially if (root!=**NULL**) then root=**New**.
3. Again read the next value of node created in **New**.
4. If (New->value < root->value) then attach **New** node as a left child of root otherwise attach **New** node as a right child of root.
5. Repeat step 3 and 4 for constructing required binary search tree completely.

```
void bintree::insert(node  *root,node *New)
{
    if(New->data<root->data)
    {
        if(root->left==NULL)
         root->left=New;
        else
         insert(root->left,New);
    }
    if(New->data>root->data)
    {
    if(root->right==NULL)
            root->right=New;
    else
            insert(root->right,New);
    }
}
```

Attaching left node to the current node.

If the root node has already some left child, then move on to left subtree

While inserting any node in binary search tree, first of all we have to look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the

value of current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.

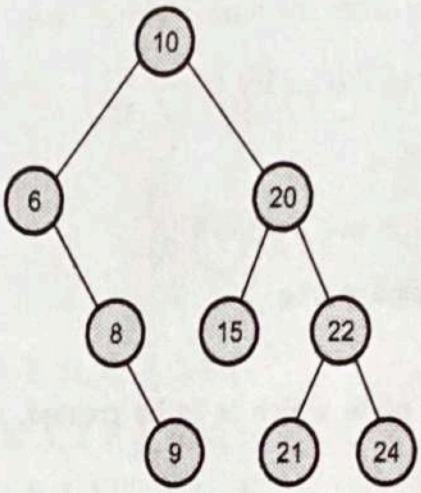


Fig. 4.6.2 Before insertion

In the Fig. 4.6.2 if we want to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is NULL node as left child of 24, we can attach 23 as left child of 24.

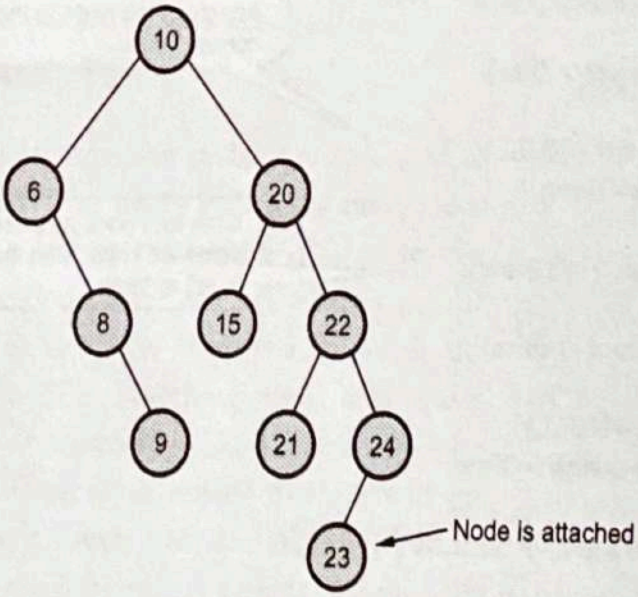


Fig. 4.6.3 After insertion

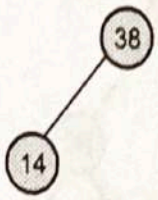
Example 4.6.1 *Define binary search tree. Draw the BST for given nodes :*
38, 14, 56, 23, 82, 8, 45, 70, 18, 15. **SPPU : Dec.-14, Marks 4**

Solution : Binary search tree - Refer section 4.6.

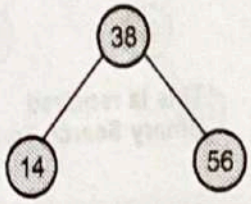
Example :

Step 1 : Insert 38

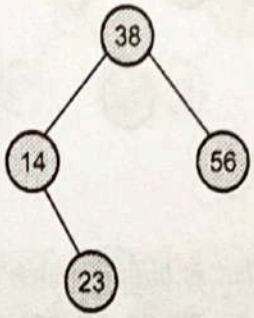
Step 2 : Insert 14



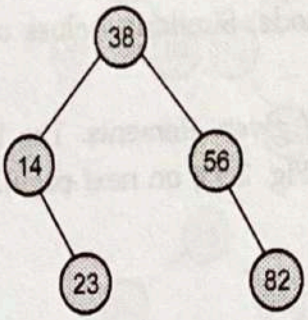
Step 3 : Insert 56



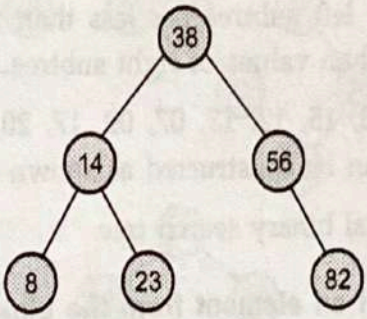
Step 4 : Insert 23



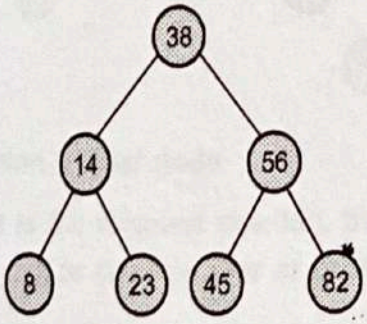
Step 5 : Insert 82



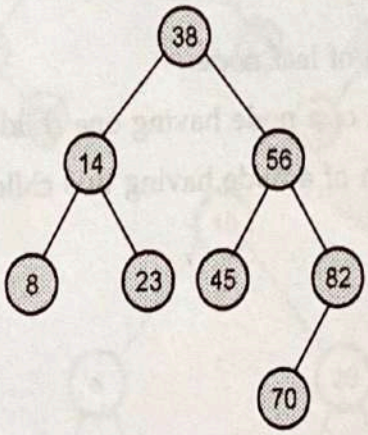
Step 6 : Insert 8



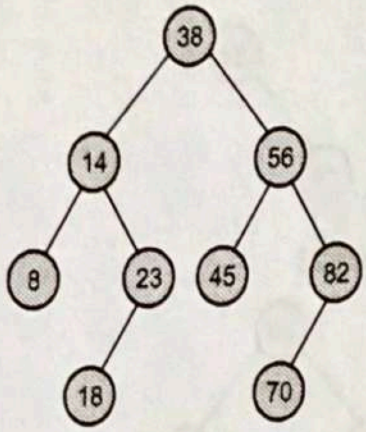
Step 7 : Insert 45



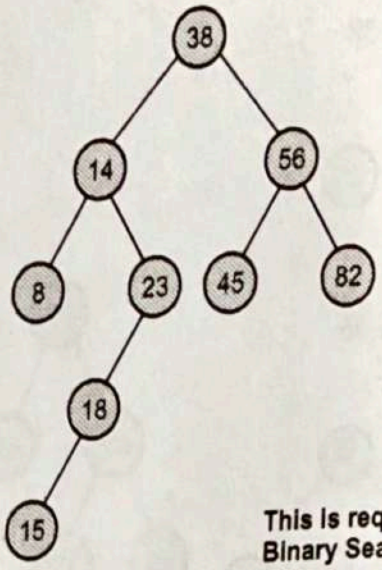
Step 8 : Insert 70



Step 9 : Insert 18



Step 10 : Insert 15



This is required Binary Search Tree

Example 4.6.2 *What is binary search tree ? Draw binary search tree for the following data : 10, 08, 15, 12, 13, 07, 09, 17, 20, 18, 04, 05.*

Solution : Binary search tree - The binary search tree is a kind of binary tree in which the values at left subtree are less than the value of root node. Similarly values of root node is less than values of right subtree.

Let, 10, 08, 15, 12, 13, 07, 09, 17, 20, 18, 04, 05 are the given elements. The binary search tree can be constructed as shown in Fig. 2.9.4 (Refer Fig. 2.9.4 on next page).

That is final binary search tree.

2. Deletion of an element from the binary tree

For deletion of any node from binary search tree there are three cases which are possible.

i. Deletion of leaf node.

ii. Deletion of a node having one child.

iii. Deletion of a node having two children.

Let us discuss the above cases one by one.

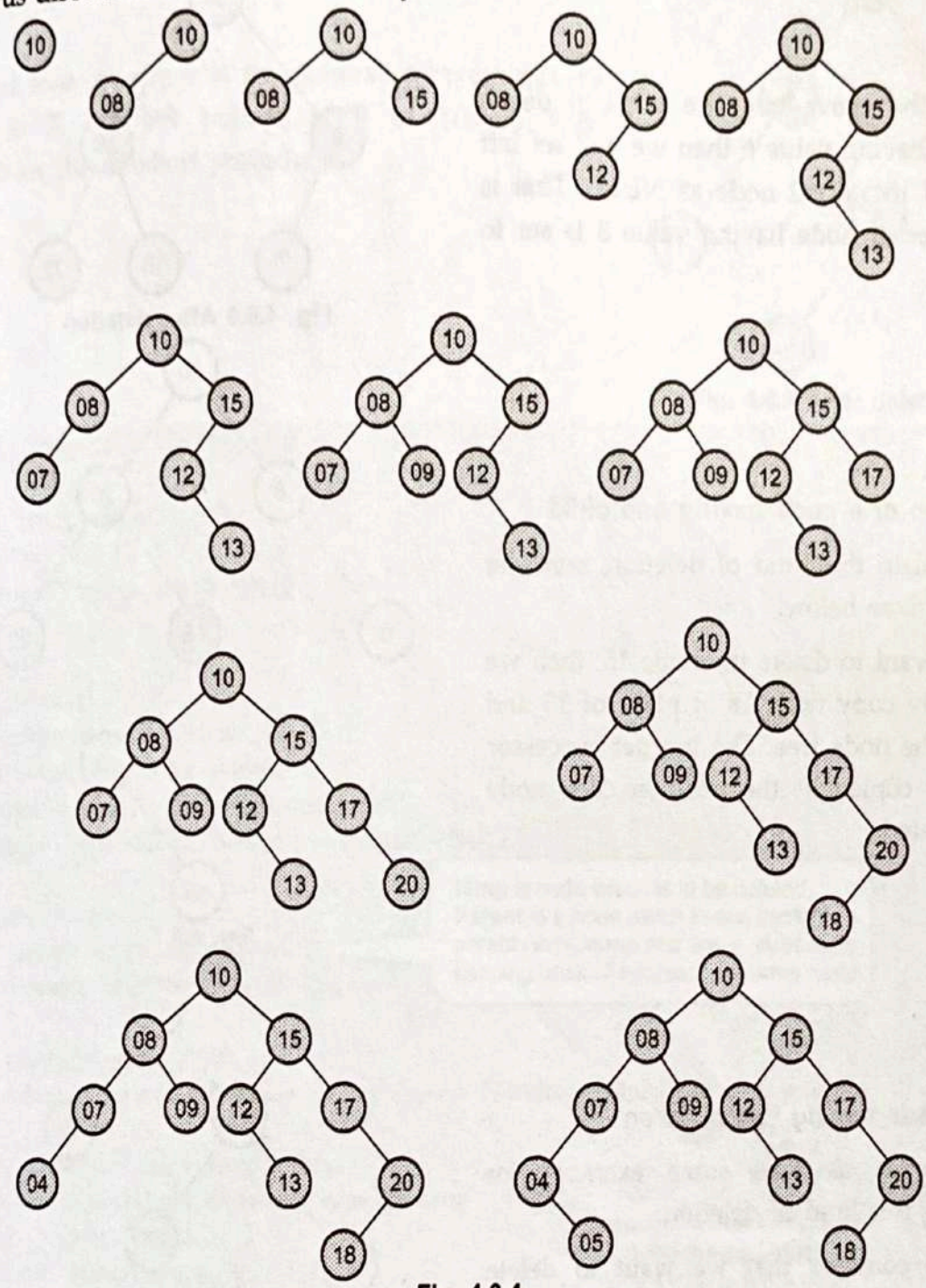


Fig. 4.6.4

i. Deletion of leaf node

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

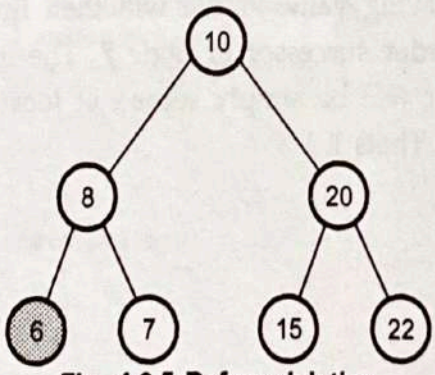


Fig. 4.6.5 Before deletion

From the above tree, we want to delete the node having value 6 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 8 is set to NULL.

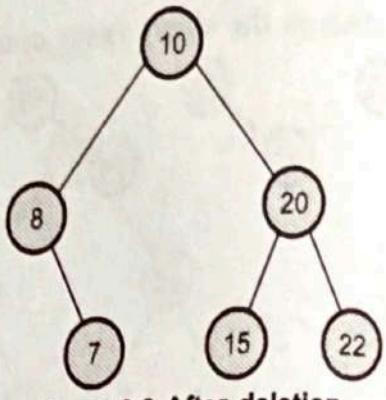


Fig. 4.6.6 After deletion

ii. Deletion of a node having one child

To explain this kind of deletion, consider a tree as given below.

If we want to delete the node 15, then we will simply copy node 18 at place of 15 and then set the node free. The inorder successor is always copied at the position of a node being deleted.

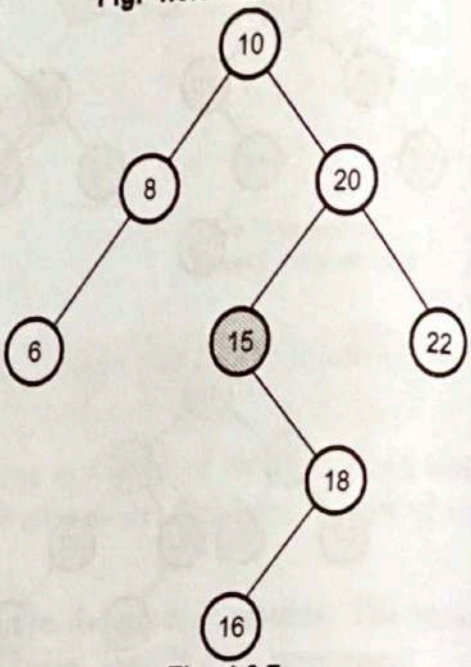


Fig. 4.6.7

iii. The node having two children

Again, let us take some example for discussing this kind of deletion.

Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. The inorder succesor will be simply copied at location of node 7. Thats it !

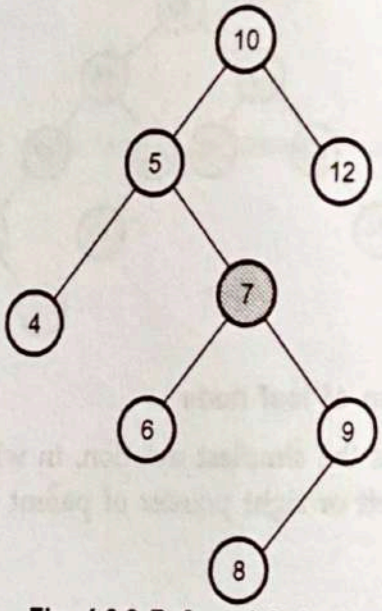


Fig. 4.6.8 Before deletion

That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.

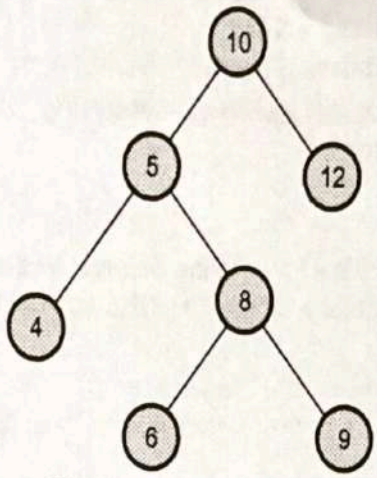


Fig. 4.6.9 After deletion

```
void bintree::del(node *root,int key)
{
 node *temp_succ;
  if(root==NULL)
   cout<<"Tree is not Created!";
  else
  {
   temp=root;
   search(&temp,key,&parent);
  //temp node is to be deleted
  /*deleting a node with two children*/
  if(temp->left!=NULL && temp->right!=NULL)
  {
   parent=temp;
temp_succ=temp->right;
   while(temp_succ->left!=NULL)
   {
       parent=temp_succ;
       temp_succ=temp_succ->left;<----------Moving to leftmost sub-tree
   }
   temp->data=temp_succ->data;
          //copying the immediate successor
   temp->right=NULL;
   cout<<" Now Deleted it!";
   return;
  }
  /*deleting a node having only one child*/
  /*The node to be deleted has left child*/
  if(temp->left!=NULL && temp->right==NULL)
  {
   if(parent->left==temp)<------------ Finding the parent of temp
       parent->left=temp->left;
   else
       parent->right=temp->left;
```

temp is node which is to be deleted. **Parent** is a node which keeps track of parent node. **temp** and **temp_succ** is for keeping track of successor of **temp** node.

Copying the successor node's data to **temp** node. Thus contents of **temp** node gets deleted.

```
 temp=NULL;
 delete temp;
 cout<<" Now Deleted it!";
 return;
}

/*The node to be deleted has right child*/
if(temp->left==NULL && temp->right!=NULL)
{
 if(parent->left==temp)
     parent->left=temp->right;
 else
     parent->right=temp->right;
 temp=NULL;
 delete temp;
     cout<<" Now Deleted it!";
 return;
}
/*deleting a node which is having no child*/
if(temp->left==NULL && temp->right==NULL)
{
     if(parent->left==temp)
      parent->left=NULL;
     else
      parent->right=NULL;
 cout<<" Now Deleted it!";
 return;
}
}
}
```

3. Searching a node from binary search tree

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub-branch otherwise on left sub-branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".

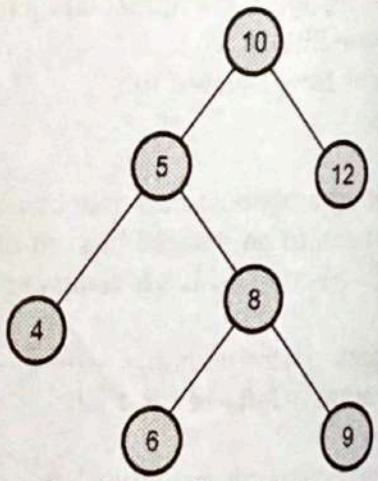


Fig. 4.6.10 After deletion

In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub-branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub-branch. Now compare 9 with 8 but

as 9 is greater than 8 we will move on right sub-branch. As the node we will get holds the value 9. Thus the desired node can be searched. Let us see the 'C' implementation of it.

The routine is as given below.

```
void bintree::search(node **temp,int key,node **parent)
{
 if(*temp==NULL)
  cout<<endl<<"Tree is Not Created"<<endl;
 else
 {
  while(*temp!=NULL)
  {
     if((*temp)->data==key)
     {
      cout<<"\nThe"<<(*temp)->data<<"Element is Present";
      break;
     }
  *parent=*temp;//stores the parent value<------
  if((*temp)->data>key) <-----------------------------
     *temp=(*temp)->left;    <---------------------
  else
     *temp=(*temp)->right;
  }
 }
 return;
}
```

- Marking the parent node
- If current node is greater than key
- Search for the left subtree

We can display a tree in inorder fashion.

4.7 Binary Search Tree as ADT

SPPU : Dec.-06,12, May-10,11,13, Marks 8

The binary search tree can be represented as an abstract data type. It is as given below.

AbstractDataType BST

{

Instances : Binary search tree is a binary tree in which value of left child is less than its parent node and value of right child is greater than its parent node.

Operations :

1. **Create -** Using this operation a binary search tree can be created.
2. **Display -** This operation is used for displaying all the nodes of BST.
3. **Insert -** For insertion of any node in the BST, this operation is useful.
4. **Delete -** Using this operation, any node from BST can be deleted.

5. **Search** - This function is used for searching any node from BST.

}

The complete implementation of BST as an ADT as follows -

```
/*********************************************************************
Program For Implementation Of Binary Search Tree and perform insertion
deletion,searching,display of tree.
*********************************************************************/
#include<iostream>
using namespace std;
class bintree
 {
      typedef struct bst
      {
       int data;
       struct bst *left,*right;
      }node;
      node  *root,*New,*temp,*parent;
  public:
 bintree()
 {
      root=NULL;
 }
 void create();
 void display();
 void delet();
 void find();
 void insert(node *,node *);
 void inorder(node *);
 void search(node **,int,node **);
 void del(node *,int);
};
void bintree::create()
{
      New=new node;
      New->left=NULL;
      New->right=NULL;
      cout<<"\n Enter The Element ";
      cin>>New->data;
      if(root==NULL)
            root=New;
      else
            insert(root,New);
}
void bintree::insert(node  *root,node *New)
{
     if(New->data<root->data)
```

```
    {
        if(root->left==NULL)
         root->left=New;
        else
         insert(root->left,New);
    }
    if(New->data>root->data)
    {
     if(root->right==NULL)
                root->right=New;
     else
                insert(root->right,New);
    }
}


void bintree::display()
{
 if(root==NULL)
      cout<<"Tree Is Not Created";
 else
      {
      cout<<"\n The Tree is : ";
      inorder(root);
      }
}

void bintree::inorder(node  *temp)
{
 if(temp!=NULL)
 {
      inorder(temp->left);
      cout<<" "<<temp->data;
      inorder(temp->right);
 }
}
void bintree::find()
{
  int key;
  cout<<"\n Enter The Element Which You Want To Search";
  cin>>key;
  temp=root;
  search(&temp,key,&parent);
  if(temp==NULL)
   cout<<"\n Element is not present";
  else
  cout<<"\nParent of node "<<temp->data<<"is "<<parent->data;
```


Scanned by CamScanner

```
}
void bintree::search(node **temp,int key,node **parent)
{
 if(*temp==NULL)
  cout<<endl<<"Tree is Not Created"<<endl;
 else
 {
 while(*temp!=NULL)
 {
      if((*temp)->data==key)
      {
       cout<<"\nThe"<<(*temp)->data<<"Element is Present";
       break;
      }
 *parent=*temp;//stores the parent value
 if((*temp)->data>key)
      *temp=(*temp)->left;
 else
      *temp=(*temp)->right;
 }
}
return;
}

void bintree::delet()
{
int key;
cout<<"\n Enter The Element U wish to Delete";
cin>>key;
if(key==root->data)
{
 bintree();//assigning a value NULL to root
}
else
del(root,key);
}
/*
-----------------------------------------------------------------------
This function is for deleting a node from binary search tree There exist three possible
cases for deletion of a node
-----------------------------------------------------------------------
*/
void bintree::del(node *root,int key)
{
node *temp_succ;
 if(root==NULL)
  cout<<"Tree is not Created!";
```


Scanned by CamScanner

```
else
{
temp=root;
search(&temp,key,&parent);
//temp node is to be deleted
/*deleting a node with two children*/
if(temp->left!=NULL && temp->right!=NULL)
{
parent=temp;
temp_succ=temp->right;
while(temp_succ->left!=NULL)
{
    parent=temp_succ;
    temp_succ=temp_succ->left;
}
temp->data=temp_succ->data;
        //copying the immediate successor
temp->right=NULL;
cout<<" Now Deleted it!";
return;
}
/*deleting a node having only one child*/
/*The node to be deleted has left child*/
if(temp->left!=NULL && temp->right==NULL)
{
if(parent->left==temp)
    parent->left=temp->left;
else
    parent->right=temp->left;
temp=NULL;
delete temp;
cout<<" Now Deleted it!";
return;
}

/*The node to be deleted has right child*/
if(temp->left==NULL && temp->right!=NULL)
{
if(parent->left==temp)
    parent->left=temp->right;
else
    parent->right=temp->right;
temp=NULL;
delete temp;
    cout<<" Now Deleted it!";
return;
}
```


Scanned by CamScanner

```
/*deleting a node which is having no child*/
if(temp->left==NULL && temp->right==NULL)
{
    if(parent->left==temp)
     parent->left=NULL;
    else
     parent->right=NULL;
cout<<" Now Deleted it!";
return;
}
}
}
int main()
{
    int choice;
    char ans='N';
    bintree tr;
    cout<<"\n\t Program For Binary Search Tree ";
    do
    {
     cout<<"\n1.Create\n2.Search\n3.Delete\n4.Display";
     cout<<"\n\n Enter your choice :";
     cin>>choice;
     switch(choice)
     {
      case 1:do
             {
               tr.create();
               cout<<"Do u Want To enter More elements?(y/n)"<<endl;
               ans=getche();
              }while(ans=='y');
              break;
      case 2:tr.find();
             break;
      case 3:tr.delet();
             break;
      case 4:tr.display();
             break;
     }
  }while(choice!=5);
return 0;
}
```

Output

```
    Program For Binary Search Tree
1.Create
2.Search
```


Scanned by CamScanner

```
3.Delete
4.Display

Enter your choice :1

Enter The Element 10
Do u Want To enter More elements?(y/n)
y
Enter The Element 8
Do u Want To enter More elements?(y/n)
y
Enter The Element 7
Do u Want To enter More elements?(y/n)
y
Enter The Element 9
Do u Want To enter More elements?(y/n)
y
Enter The Element 12
Do u Want To enter More elements?(y/n)
y
Enter The Element 11
Do u Want To enter More elements?(y/n)
y
Enter The Element 13
Do u Want To enter More elements?(y/n)
n
1.Create
2.Search
3.Delete
4.Display

Enter your choice :4

The Tree is :  7  8  9  10  11  12  13
1.Create
2.Search
3.Delete
4.Display

Enter your choice :2

Enter The Element Which You Want To Search 13

The13Element is Present
Parent of node 13is 12
1.Create
```


Scanned by CamScanner

```
2.Search
3.Delete
4.Display

Enter your choice :3

Enter The Element U wish to Delete 12

The 12 Element is Present Now Deleted it!
1.Create
2.Search
3.Delete
4.Display


Enter your choice :4

The Tree is :  7  8  9  10  11  13
1.Create
2.Search
3.Delete
4.Display

Enter your choice :3

Enter The Element U wish to Delete 11

The11Element is Present Now Deleted it!
1.Create
2.Search
3.Delete
4.Display

Enter your choice :4

The Tree is :  7  8  9  10  13
1.Create
2.Search
3.Delete
4.Display

Enter your choice :3

Enter The Element U wish to Delete7

The7Element is Present Now Deleted it!
1.Create
```


Scanned by CamScanner

```
2.Search
3.Delete
4.Display

Enter your choice :4

The Tree is :  8  9  10  13
1.Create
2.Search
3.Delete
4.Display

Enter your choice :5
```

Example 4.7.1 *Write efficient functions that takes only a pointer to the root of binary tree T, and compute :*

1) The number of nodes in T 2) The number of leaves in T.

What is running time of your algorithm ?

Solution : 1) The number of nodes in T

There are two function written -

count_nodes function has a logic for visiting each node of the tree and counting them.

display function displays the actual count value

```
/*
-----------------------------------------------------------------------
Function which counts the total number of nodes in the tree
-----------------------------------------------------------------------
*/
void tree::count_nodes(btree *temp,int *count)
{
 if(root==NULL)
     cout<"TREE IS NOT CREATED";
if(temp!=NULL)
{
 count_nodes(temp->left,count);
 *count++;
 count_nodes(temp->right,count);
}
}
/*
-----------------------------------------------------------------------
The calling function - It calls the function for counting total number of nodes
-----------------------------------------------------------------------
```


Scanned by CamScanner

```
*/

void tree::display()
{
 int *count=0;
 count_nodes(root,count);
 cout<<"Total nodes= "<<*count;
}
```

2) The number of leaves in T

There are two function written –

count_leaves function has a logic for visiting each leaf node of the tree and counting them.

display function displays the actual count value for the leaves

```
/*
-----------------------------------------------------------------------
The count_leaves Function
-----------------------------------------------------------------------
*/
void tree::count_leaves(btree *temp,int *count)
{
 if(root==NULL)
      cout<<"TREE IS NOT CREATED";
 if(temp!=NULL)
 {
 if((temp->left==NULL)&&(temp->right==NULL))
       *count=*count+1;
 else
 {
  count_leaves(temp->left,count);
  count_leaves(temp->right,count);
 }
 }
}
/*
-----------------------------------------------------------------------
The display function
-----------------------------------------------------------------------
*/

void tree::display()
{
 int *count;
 *count=0;
 count_leaves(root,count);
 cout<<"Total nodes= "<<*count;
}
```


Scanned by CamScanner

Example 4.7.2 *Write a pseudo C routine to find the depth of the binary tree.*

SPPU : Dec.-06, Marks 4

Solution :

```
int tree::Height(btree *temp)
{
 int d1,d2;
 if(temp==NULL)
  return 0;
 if(temp->left==NULL && temp->right==NULL)
  return 0;
 d1=Height(temp->left);//computing height of leftsubtree
 d2=Height(temp->right);//computing height of rightsubtree
  if(d1>d2)
  return d1+1;
  else
  return d2+1;
//maximum depth=height of the tree
}
/*
-----------------------------------------------------------------------
      The display function
-----------------------------------------------------------------------
*/

void tree::display()
{
 cout<<"The height of the tree is "<<Height(root);
}
/*
```

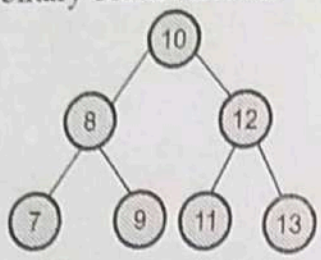
Example 4.7.3 *Write recursive function to find mirror image of a given binary tree. Show the contents of stack stepwise.*

Solution : Recursive function to find mirror image

```
void tree : : mirror (btree*root)
{
btree * temp:
if (root | = NULL)
{
      mirror(root  → left);
      mirror(root → right);
      temp = root → left;
      root → left = root → right;
      root → right = temp ;
}
}
```

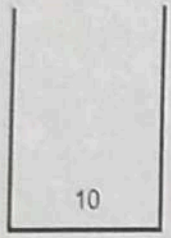

Scanned by CamScanner

Stack contents - Assume the binary search tree as -



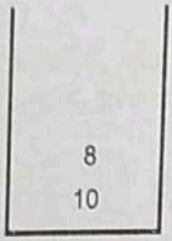
Binary search tree

Step 1 :



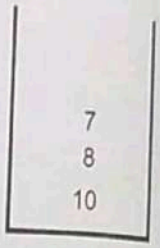
Stack

Step 2 :



Stack

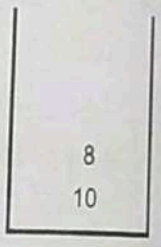
Step 3 :



Stack

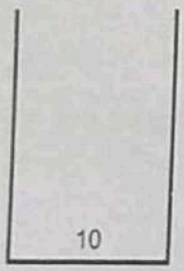
Step 4 :

The stack will be popped, node 7 has no left and right child.

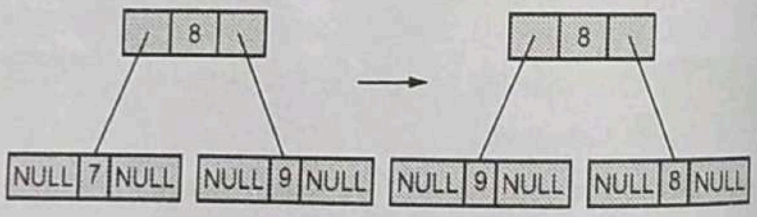


Stack

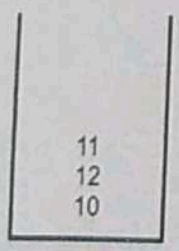
Step 5 : The stack will be popped. Then the left and right child of the nod 8 will be swapped.



stack

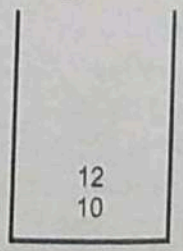


Step 6 :



Stack

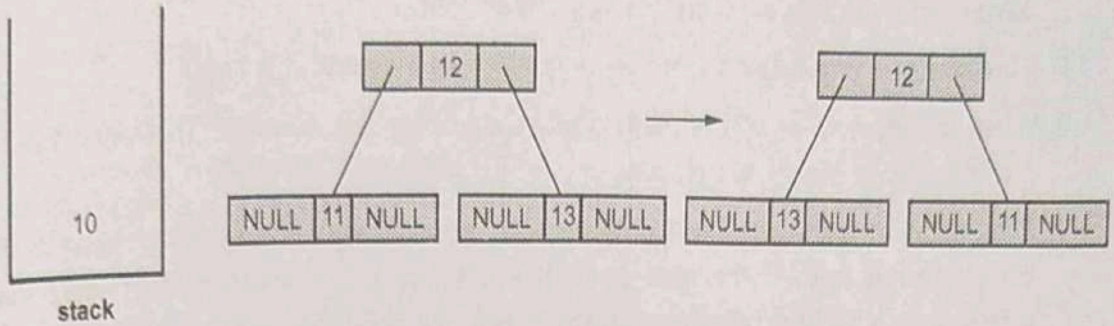
Step 7 : The stack will be popped, node 11 ha no left and right child



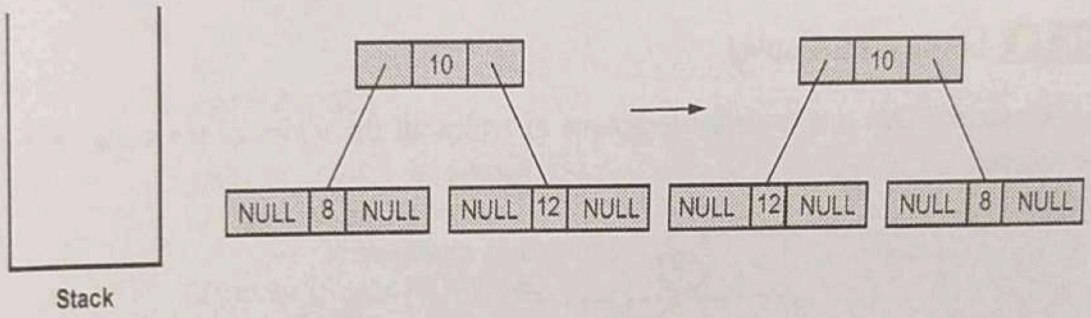
Stack

Scanned by CamScanner

Step 8 : The stack will be popped. Then left and right child of the node 12 will be swapped.



Step 9 : The stack will be popped. Then the left and right child of node 10 will be popped.

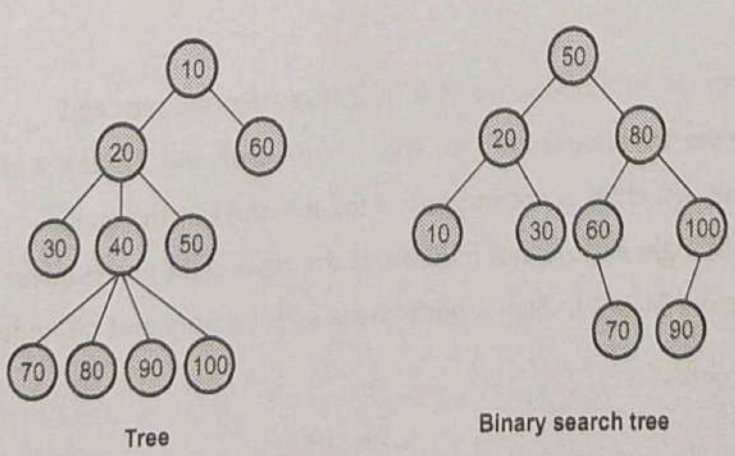


As now stack is empty, the control will return to main function.

Example 4.7.4 *Explain the difference between a tree and a binary search tree. Write an algorithm for conversion of binary tree to binary search tree.*

Solution : The tree is a data structure having one root node and any number of subtrees. The binary search tree is a data structure in which there is one root node and every node has at the most two subtrees. The arrangement of nodes is based on their values. That is left node is less than parent node and right node is greater than parent node.

For example



Scanned by CamScanner

Algorithm

1. Traverse the binary tree in an inorder manner. When the node value is read in the traversing store it in an array say a[], one by one.
2. Now start reading array **a** from left to right, one element at a time.
3. When the element at a[0] is read, create a node for this element. This node will be a root node, of binary search tree.
4. Read element from array **a**. Compare each element with the nodes present in the binary search tree. If the new node has a value less than its parent node then attach it as left child otherwise attach it as right child. Repeat this step for all the elements of array **a**.
5. Display the binary search tree created in step 3 and 4.

4.7.1 Level-wise Display

Concept : This is a traversal technique in which all the nodes of the same level are displayed.

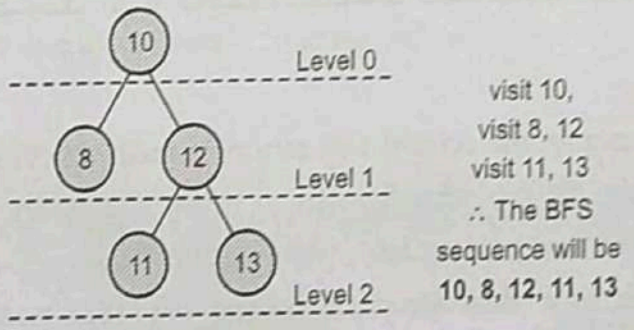


Fig. 4.7.1

In this traversal method, we start from the root node and go on printing the data in level wise manner.

An additional data structure called **queue** is required to perform this traversal.

Algorithm :

Step 1 : Visit the root node . Insert it in Queue from the rear end.

Step 2 : Delete the node from front end of the queue and display it as output.

Step 3 : If its left child is not null insert the left child in the queue.

Step 4 : If its right child is not null, insert the right child in the queue.

Step 5 : Repeat Step 2 to Step 4 until queue is not empty and all nodes are visited.

Scanned by CamScanner

Pseudo Code :

```
Algorithm LevelWiseDisplay()
{
    btree *temp, *dummy;
    dummy = new btree;
    if ( dummy == NULL)
      cout<<"Insufficient Memory\n";
    dummy->left = root;
    dummy->right = NULL;
    dummy->data = ' ';
    temp = dummy ->left;
    enque(temp);
    enque(dummy);
    temp = deque();
    while ( front != rear)
    {
        if ( temp != dummy)
        {
            cout<<" "<<temp->data;
            if ( temp ->left != NULL)
                    enque(temp -> left);
            if ( temp ->right != NULL)
                enque(temp -> right);
        }
        else
        {
            enque(temp);
            cout<<"\n";
        }
        temp = deque();
    }
}
```

Example 4.7.5 *Consider following tree and find its Node sequence.*

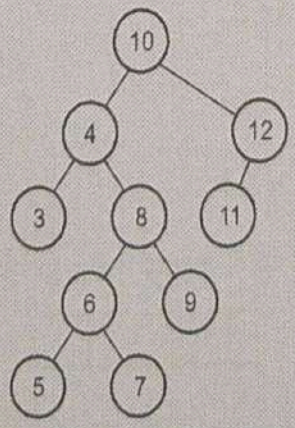
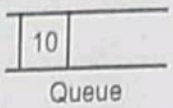


Fig. 4.7.1

Scanned by CamScanner

Solution :

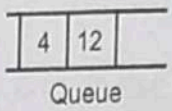
Step 1 : We will visit root node insert it in queue.



Delete the node from queue, display it as output.

∴ **Node sequence :** 10

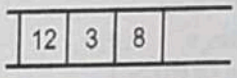
Step 2 : Find left and right nodes of 10, insert them in queue.



Delete the node 4 from, display as output.

∴ **Node sequence :** 10, 4

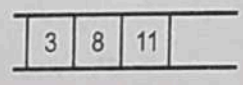
Step 3 : Find left and right child of 4, insert them in queue.



Delete 12, display as output.

∴ **Node sequence :** 10, 4, 12

Step 4 : Find left and right child of 12, insert them in queue.



Delete 3, display it.

∴ **Node sequence :** 10, 4, 12, 3

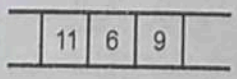
Step 5 : Find left and right child of 3.

But 3 has no left or right child.

Now delete 8, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8

Step 6 : Find left and right child of 8, insert them in queue.



Delete 11, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8, 11

Scanned by CamScanner

Step 7 : There is no left or right child of 11. So nothing will be inserted in queue.

Now delete 6, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8, 11, 6

Step 8 : Find left and right child of 6. Insert them in queue.

	9	5	7

Delete 9, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8, 11, 6, 9

Step 9 : There is no left or right child of 9. Delete 5, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8, 11, 6, 9, 5

Step 10 : There is no left or right child of 5.

	7	

Delete 7, display it.

∴ **Node sequence :** 10, 4, 12, 3, 8, 11, 6, 9, 5, 7

C++ Program

```
/*****************************************************************
Program to create a binary tree and print the data level wise or in a
breadth first search order.
*****************************************************************/
#include<iostream.h>
#include<conio.h>
#define size 50

/* Declare a node for binary tree */
class tree
{
      private:
       typedef struct node
       {
            int  data;
            struct node *left;
            struct node *right;
       }btree;
       btree *root,*New;
  public:
  tree();
  void create(),LevelWiseDisplay();
```


Scanned by CamScanner

```
  void insert(node *,node *);
  void enque(btree *Node);
  btree *deque ();
  btree *que[size];
  void remove(btree *);
  ~tree();
  int front,rear;
};

/*
-----------------------------------------------
The constructor defined
-----------------------------------------------
*/
tree::tree()
{
 root=NULL;
 front=rear=-1;
}
/*
-----------------------------------------------
The create function
-----------------------------------------------
*/
void tree::create()
{
 New=new btree;
 New->left=NULL;
 New->right=NULL;
 cout<<"\n Enter The Element ";
 cin>>New->data;
 if(root==NULL)     /* Tree is not Created */
        root=New;
 else
        insert(root,New);
}
/*
-----------------------------------------------------------------------------------------
The insert Function
-----------------------------------------------------------------------------------------
*/
void tree::insert(node *root,node *New)
{
char ch;
cout<<"\n Where to insert left/right of "<<root->data<<" ";
ch=getche();
if ((ch=='r')||(ch=='R'))
```


Scanned by CamScanner

```
{
            if(root->right==NULL)
            {
            root->right=New;  //attaching new node as a right child
            }
            else
            insert(root->right,New);  //moving on the right branch

}
else
{
if (root->left==NULL)
{
    root->left=New;   //attaching new node as a left child
}
else
    insert(root->left,New); //moving on the left branch
}
}

/*
-----------------------------------------------------------------------
    The LevelWiseDisplay function
-----------------------------------------------------------------------
*/
void tree::LevelWiseDisplay()
{
    btree *temp, *dummy;
    dummy = new btree;
    if ( dummy == NULL)
      cout<<"Insufficient Memory\n";
    dummy->left = root;
    dummy->right = NULL;
    dummy->data = ' ';
    temp = dummy ->left;
    enque(temp);
    enque(dummy);
    temp = deque();

    while ( front != rear)
    {
        if ( temp != dummy)
        {
            cout<<"  "<<temp->data;
            if ( temp ->left != NULL)
                    enque(temp -> left);
            if ( temp ->right != NULL)
                enque(temp -> right);
```


Scanned by CamScanner

```
            }
            else
            {
                enque(temp);
                cout<<"\n";
            }
            temp = deque();
        }
}

/*
-----------------------------------------------------------------------
The enque function
-----------------------------------------------------------------------
*/
void tree::enque(btree *Node)
{
        if ( rear == size-1)
            cout<<"Queue is full\n";
        rear = rear + 1;
        que[rear] = Node;
}

/*
-----------------------------------------------------------------------
The deque function
-----------------------------------------------------------------------
*/
btree *tree::deque ()
{
        btree *Node;

        if ( front == rear )
            cout<<"Queue is empty\n";
        front++;
        Node = que[front];
        return(Node);
}
/*
-----------------------------------------------------------------------
The destructor defined
-----------------------------------------------------------------------
*/
tree::~tree()
{
remove(root);
}
```


Scanned by CamScanner

```
/*
-----------------------------------------------------------------------
The remove function for de allocating the memory
-----------------------------------------------------------------------
*/
void tree::remove(btree *root)
{
 if(root!=NULL)
 {
  remove(root->left);
  remove(root->right);
 }
 delete root;
}

/*
-----------------------------------------------------------------------
      The main function
-----------------------------------------------------------------------
*/
void main ()
{
  tree tr;
  char ans='y';
      do
       {
        clrscr();
        tr.create();
        cout<<"\n Do u Want To enter More Elements?(y/n)\n";
        ans=getch();
        }while(ans=='y');
       cout<<"\n\n The Breadth First Traversal is ...\n";
       tr.LevelWiseDisplay();
       getch();
}
```

Output

```
Enter The Element 10

Do u Want To enter More Elements?(y/n)

Enter The Element 8

Where to insert left/right of 10 l
Do u Want To enter More Elements?(y/n)
```


Scanned by CamScanner

```
Enter The Element 9

Where to insert left/right of 10 l
Where to insert left/right of 8 r
Do u Want To enter More Elements?(y/n)

Enter The Element 12

Where to insert left/right of 10 r
Do u Want To enter More Elements?(y/n)

Enter The Element 7

Where to insert left/right of 10 l
Where to insert left/right of 8 l
Do u Want To enter More Elements?(y/n)

Enter The Element 11

Where to insert left/right of 10 r
Where to insert left/right of 12 l
Do u Want To enter More Elements?(y/n)

Enter The Element 13

Where to insert left/right of 10 r
Where to insert left/right of 12 r
Do u Want To enter More Elements?(y/n)

The Breadth First Traversal is ...
 10
 8  12
 7  9  11  13
```

Review Questions

1. *Write an algorithm to delete a node from binary search tree.* **SPPU : May-10, 11, Marks 6**
2. *Write recursive algorithms to find number of leaf nodes and height of binary tree.* **SPPU : May-10, Marks 6**
3. *Write a recursive algorithm to find the mirror image of binary tree and explain it with suitable example.* **SPPU : May-11, Marks 6**
4. *Write a function for creating binary search tree.* **SPPU : Dec.-12, Marks 4**
5. *What is bianry search tree ? Write pseudocode for deletion and insertion of a node in a binary search tree.* **SPPU : May-13, Marks 8**

Scanned by CamScanner

4.8 Threaded Binary Tree

SPPU : Dec.-10,18, May-16, Marks 8

In threaded binary tree the leaf nodes are having the NULL values in the left and right link fields. It is just a wastage of memory. So to avoid the NULL value the threads are used. The threads are nothing but the links to predecessor and successor nodes.

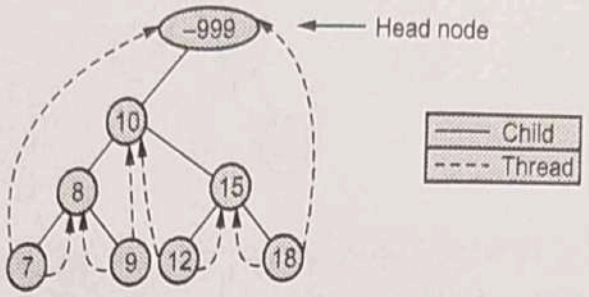


Fig. 4.8.1 Threaded binary tree

There are three types of threading possible - inorder, preorder and postorder threading.

The basic idea in inorder threading is that the **left thread** should point to the **predecessor** and the **right thread** points to the **successor.** Here we are assuming the head node as the starting node and the root node of the tree is attached to left of head node which is as shown in Fig. 4.8.1.

C structure

```
typedef struct thread
{
 int data;
int lth,rth;
struct thread *left;
struct thread *right;
}Th;
```

Example 4.8.1 *For the binary tree represented as an array, perform in-order threading on the tree :*

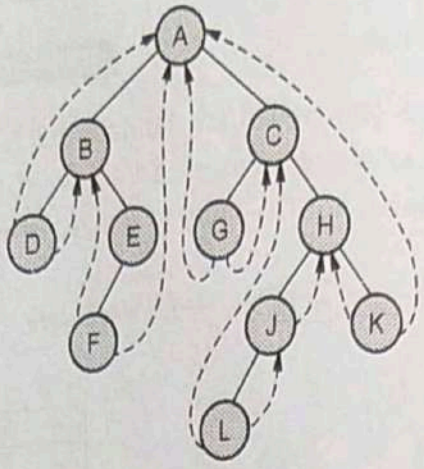
A B C D E G H — — F — — — J K — — — — — — — — — — — — L — —

SPPU : Dec.-10, Marks 6

Solution :

0	1	2	3	4	5	6	9	13	14	27
A	B	C	D	E	G	H	F	J	K	L

Scanned by CamScanner



Example 4.8.2 *Draw the threaded binary tree equivalent for the tree represented by the following array assuming root node of tree is stored at index 0 in the array.*

SPPU : May-16, Marks 4

Index	Data
0	10
1	–
2	20
3	–
4	–
5	30
6	–
7	–
8	–
9	–
10	–
11	–
12	40
13	–
14	–
15	–
16	–
17	–
18	–
19	–
20	–
21	–

Scanned by CamScanner

Index	Data
22	–
23	–
24	–
25	50
26	–
27	–
28	–
29	–
30	–
31	–

Solution : When n = 0, the root node will be placed at 0th location. The nodes are stored in array using following formula,

Parent (n) = floor (n – 1) / 2

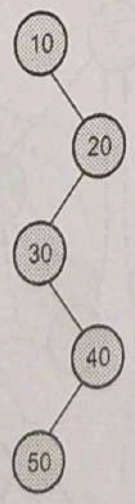
Leftchild (n) = 2n + 1

Rightchild (n) = 2n + 2

When n = 0

Rightchild (0) = 2 × 0 + 2 = 2. That means 20 is attached as right child of 10.

Let n = 2, Leftchild (2) = 2n + 1 = 2 × 2 + 1 = 5. At 5th index the data is 30. That means 30 is attached as leftchild of 20. In this way we can get the construction of binary tree as -

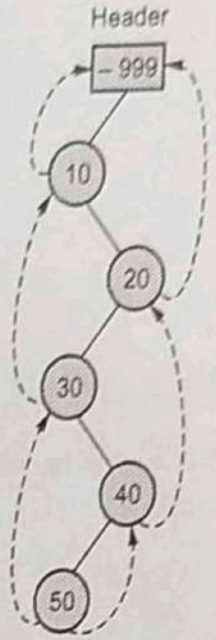


Binary tree

Fig. 4.8.2

Scanned by CamScanner

The predecessor is pointed by left thread and successor of node is pointed by right thread. Thus the converted threaded binary tree will be -



Threaded binary tree

Fig. 4.8.3

4.8.1 Concept of Threaded Binary Tree (Inorder, Preorder and Postorder)

The inorder threading is a way in which the left thread points to inorder predecessor of the current node and right thread points to inorder successor of the current node.

For example : Inorder traversal - 3, 2, 5, 4, 1, 7, 6, 10, 9, 8

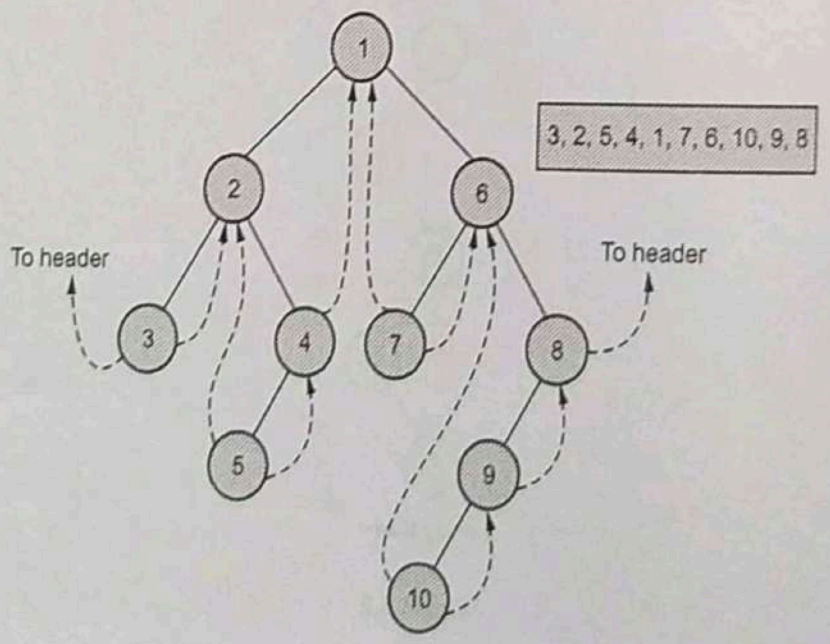


Fig. 4.8.4

Scanned by CamScanner

The preorder threading is a way in which the left thread points to preorder predecessor of the current node and right thread points to preorder successor of it.

For example : Preorder sequence - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

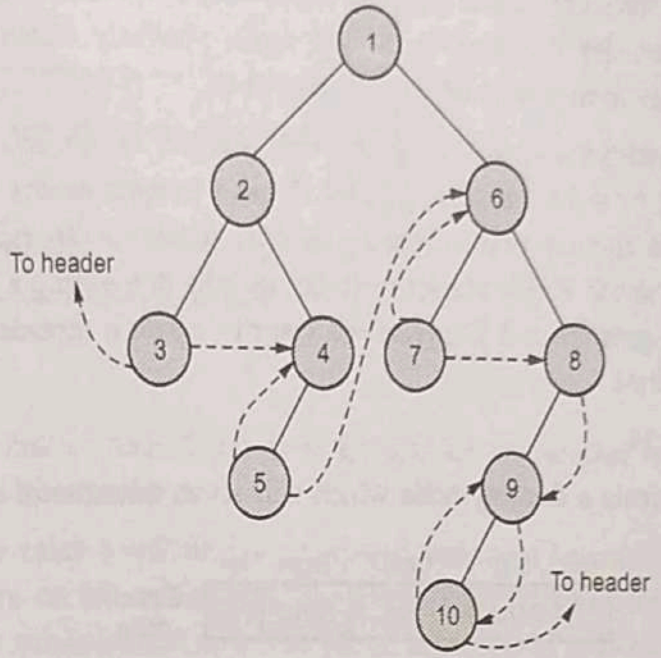


Fig. 4.8.5

The postorder threading is a way in which the left thread points to postorder predecessor and right thread points to postorder successor of it.

For example : Postorder traversal - 3, 4, 2, 7, 10, 9, 8, 6, 1

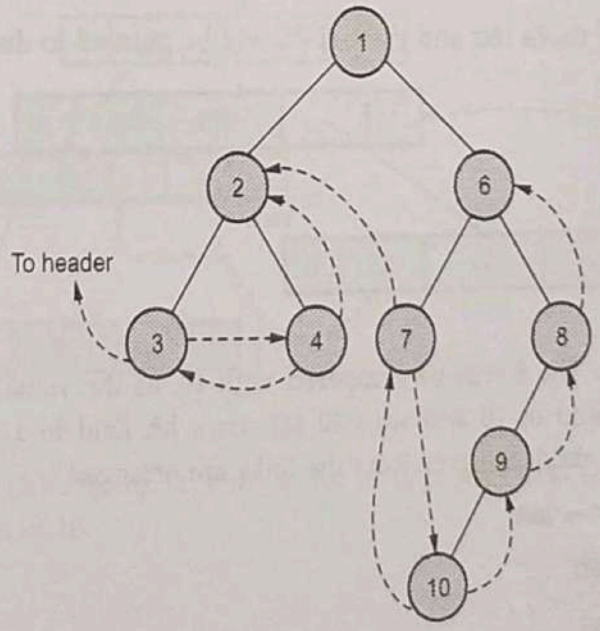


Fig. 4.8.6

Scanned by CamScanner

4.8.2 Insertion and Deletion Operations

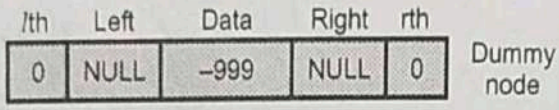
1. Insertion Operation

In thread binary tree, the NULL pointers are avoided. Instead of left NULL pointer the link points to inorder predecessor of that node. Similarly instead of right NULL pointer the link points to inorder successor of that node.

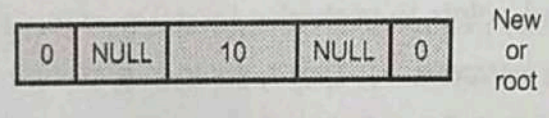
There are two additional fields in each node named as lth and rth these fields indicate whether left or right thread is present. Thread present means the NULL link is replaced by a link to inorder predecessor or inorder successor. To represent that there exists thread the lth or rth Fields are set to 0. Let us take one example to explain how a threaded binary tree gets created. Suppose we want to create a threaded binary tree for the nodes haivng values -

10, 8, 6, 12, 9, 11, 14

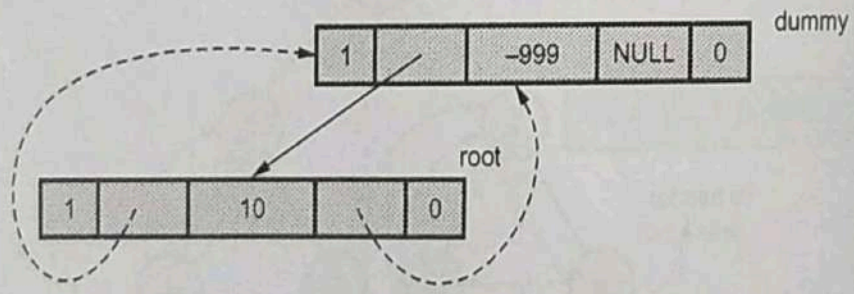
Initially we will create a dummy node which will act as a header of the tree.



Now let us take first value i.e. 10. The node with value 10 will be the root node we will attach this root node as left child of dummy node.



The NULL links of root's left and right child will be pointed to dummy.



Now next comes 8. The 8 will be compared with 10, as the value is less than 10, we will attach 8 as left child of 10 and we will set root's lth field to 1, indicating that the node 10 is having left child. See then how the links are arranged.

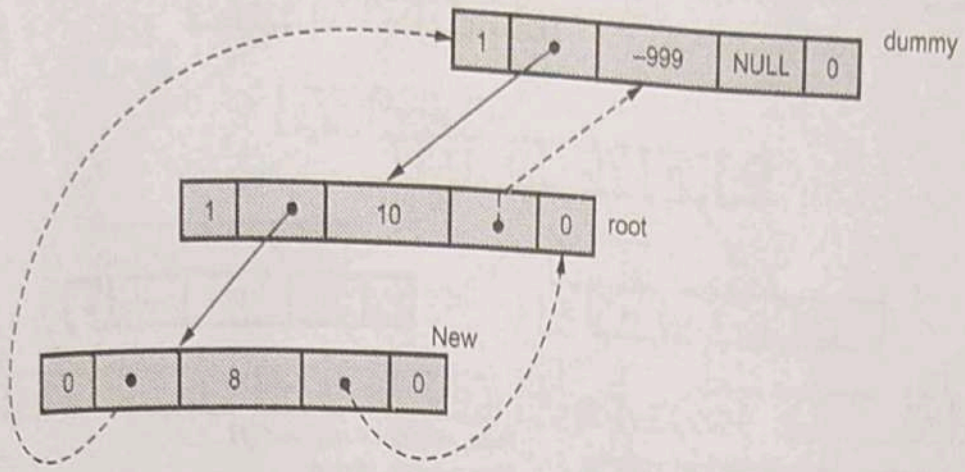
New → left = root → left

New → right = root;

root → left = New;

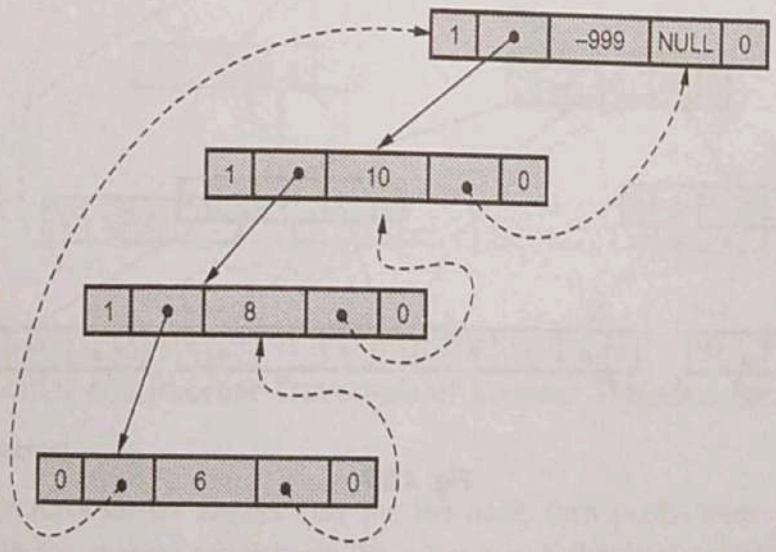
root → lth = 1;

Scanned by CamScanner



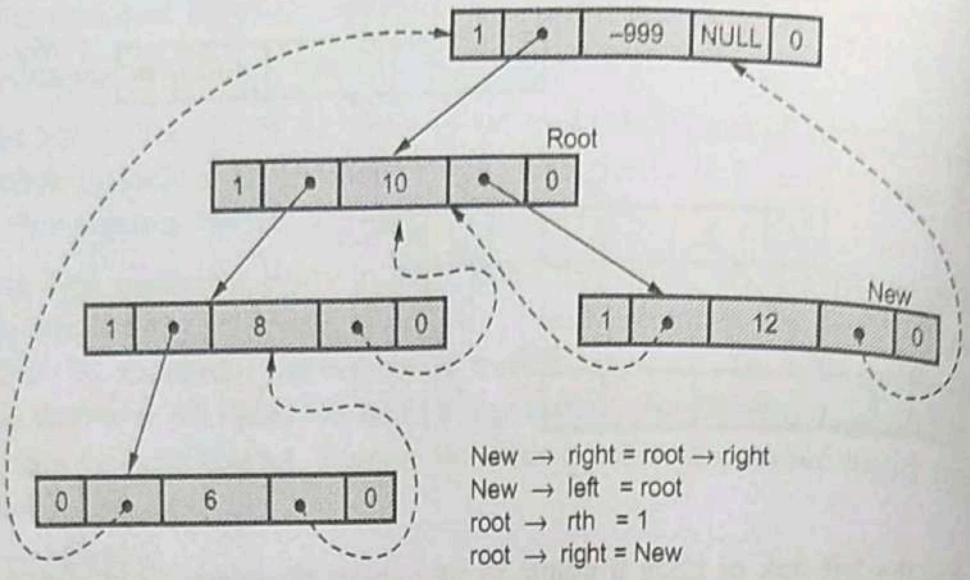
Note that the left link of node 8 points to its inorder predecessor and right link of node 8 points to its inorder successor.

Next comes 6. The value 6 will be first compared with root i.e. with 10. As 6 is less than 10, we will move on left sub-branch. The 6 will then be compared with 8, so we have to move on left sub-branched of 8. But lth of node 8 is 0, indicating that there is no left child to 8, so we will attach 6 as left child to 8.



Then comes 12. We will compare 12 with 10. As 12 is greater than 10, we will attach the node 12 as right child of 10.

Scanned by CamScanner



Note that the left field of node 12 points to its inorder predecessor and right field of node 12 points to its inorder sussessor. Thus by attaching 9, 11, 14 as appropriate children, the threaded binary tree will look like this -

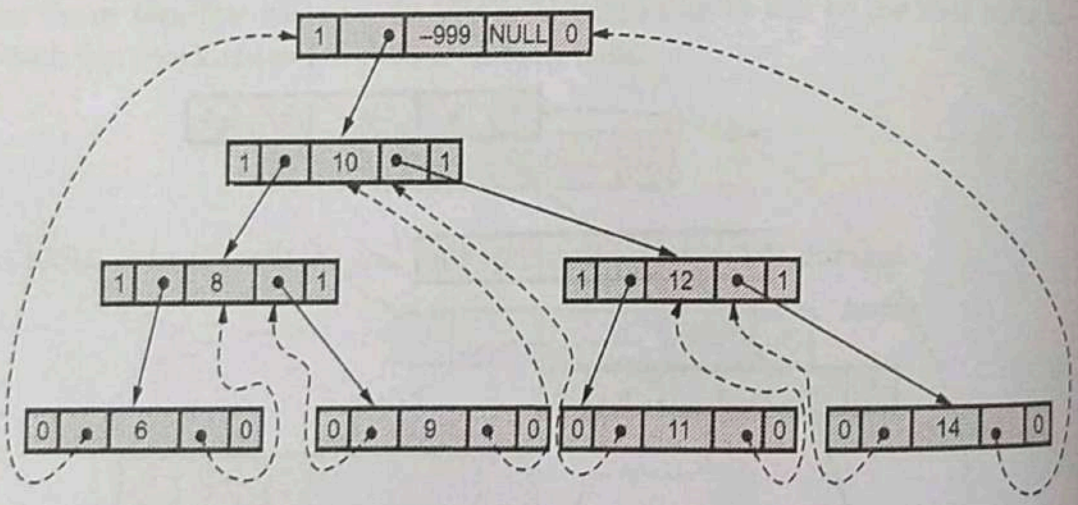


Fig. 4.8.7

Such a threaded tree is called inorder treaded binary tree.

2. Deletion Operation

The deletion operation is similar to the deletion operation in BST. The only additional thing is to adjust the left thread (*l*th) and right thread (rth) fields. For example -

Consider the given threaded binary tree as -

Scanned by CamScanner

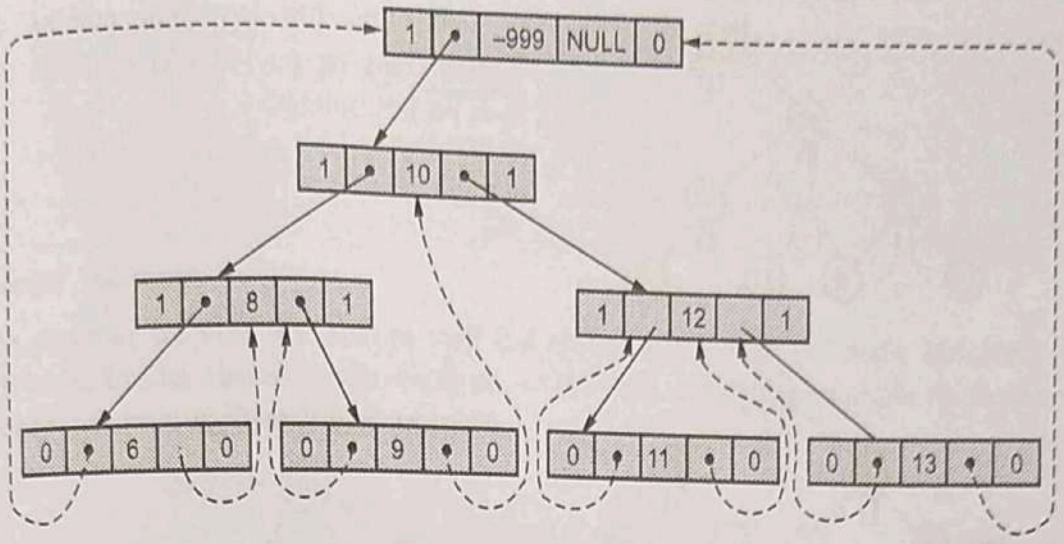


Fig. 4.8.8

If we **delete node 12** then the inorder successor 13 will occupy 12th location.

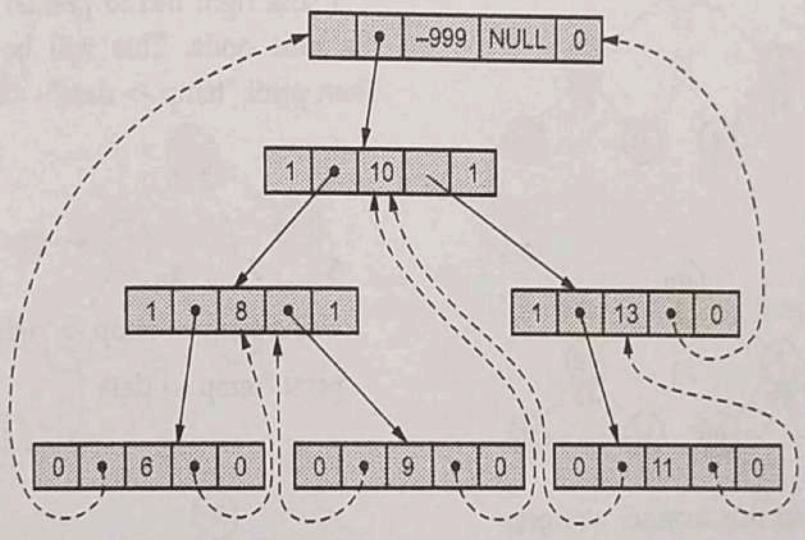


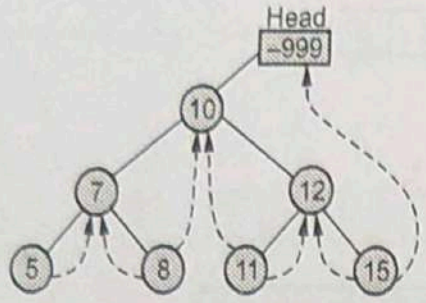
Fig. 4.8.9

4.8.3 Preorder and In-order Traversals of In-order Threaded Binary Tree

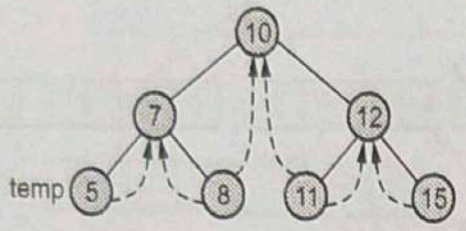
Inorder Traversal

In inorder traversal we always visit the left node, then parent node and lastly the right node. In this manner we traverse the entire tree. Following example illustrates the inorder traversal of threaded binary tree.

Scanned by CamScanner

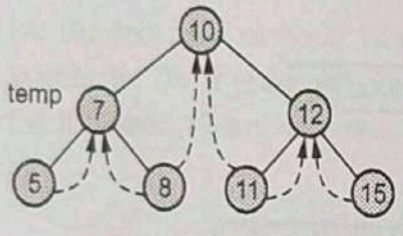


If temp has left child pointer, then keep moving on left pointer untill there is no left pointer.



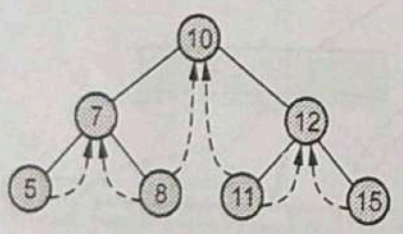
5

print "temp -> data"



5 7

Using right thread pointer move on to the next node. This will be new temp. Then print "temp -> data"



5 7 8

Using temp = temp -> right,

print "temp -> data"

Continuing in this manner, we get,

5 7 8 10 11 12 15 as inorder sequence.

C++ Code

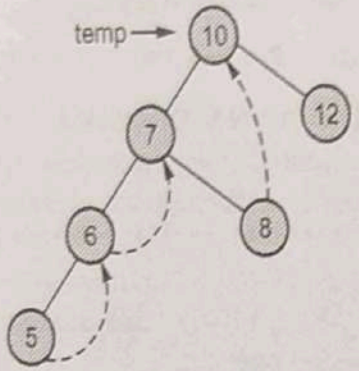
```
void inorder(node *temp,node *dummy)
{
  while(temp!=dummy)
  {
      while(temp->lth==1)
            temp=temp->left;
      cout<<"\t"<<temp->data;
      while(temp->rth==0)
      {
            temp=temp->right;
            if(temp==dummy)
```


Scanned by CamScanner

```
            return;
        cout<<"\t"<<temp->data;
    }
    temp=temp->right;
  }
}
```

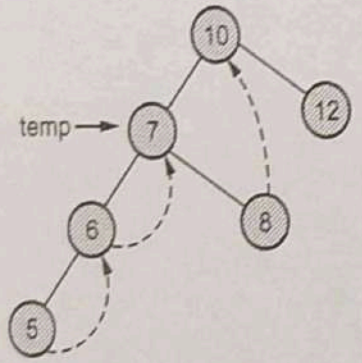
Preorder Traversal

In preorder traversal we always visit the parent node, then left node and lastly the right node. In this manner we traverse the entire tree. Following example illustrates the preorder traversal of threaded binary tree.



10

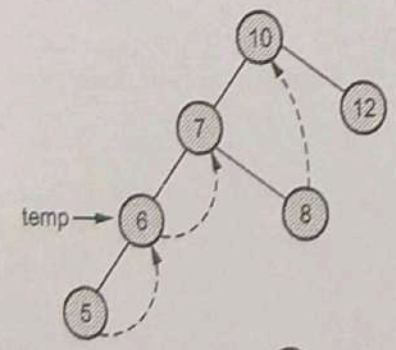
Print "temp -> data"



10 7

Print "temp -> data" if left subtree exists.

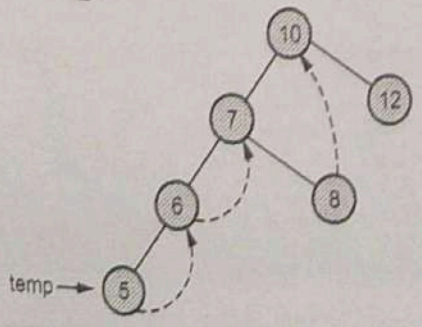
Make temp -> left as temp.



10 7 6

Print "temp -> data" if left subtree exists.

Make temp -> left as temp.

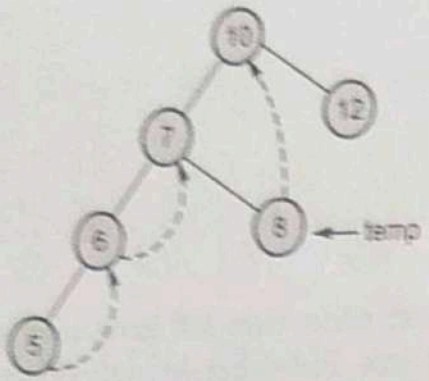


10 7 6 5

Print " temp -> data " If left and right subtree don't exists, and right thread exist, traverse up the right threads.

If you arrive at last node, then stop. Else make temp -> right as temp.

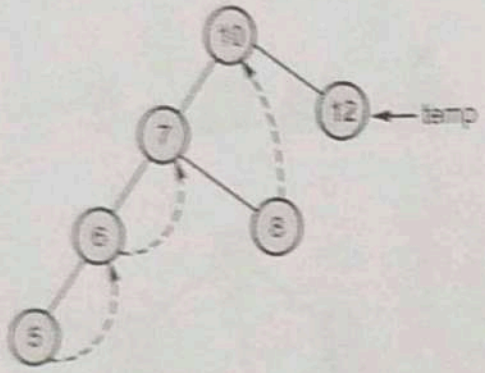
Scanned by CamScanner



10 7 6 5 8

Print " temp -> data " If left and right subtree don't exists, and right thread exists, traverse up the right threads.

If you arrive at last node, then stop. Else make temp -> right as temp.



Print "temp -> data"

∴ 10 7 6 5 8 12

This is preorder travers.

C++ Code

```
void preorder(node *temp,node *dummy)
{
  int flag=0;
  while(temp!=dummy)
  {
        if(flag==0)
         cout<<"\t"<<temp->data;
        if((temp->lth==1)&&(flag==0))
        {
            temp=temp->left;
        }
        else
        {
              while(1)
              {
                    if(temp->rth==1)
                    {
                          flag=0;
                          temp=temp->right;
                          break;
                    }
                    else
                    {
```


Scanned by CamScanner

```
                    if(temp!=dummy)
                    {
                        flag=1;
                        temp=temp->right;
                        break;
                    }
                }//end else
            }//end while
        }//end else
    }//end while
}
```

Let us see the C++ implementation of inorder threaded binary tree. The tree is displayed using inorder and preorder traversal.

```
/*****************************************************************************
Program For Implementation Of Inorder Threaded Binary Search Tree
and perform insertion,deletion and inorder and preorder traversing of tree
*****************************************************************************/
#include <iostream>
using namespace std;
class node
{
public:
  int data;
  int lth,rth;
  node *left,*right;
};

class thread
{
 private:
 node *dummy;
 node *New,*root,*temp,*parent;
 public:
 thread();
 void create(); //All The implementation Details are hidden!
 void insert(node *,node *);
 void display();
 void find();
 void delet();
 void del(node *root,node *dummy,int key);
};

/*
-----------------------------------------------------------------
The constructor defined
-----------------------------------------------------------------
```


Scanned by CamScanner

```
*/
thread::thread()
{
 root=NULL;
}
/*
-----------------------------------------------------------------
The create function
-----------------------------------------------------------------
*/
void thread::create()
{
 New=new node;
 New->left=NULL;
 New->right=NULL;
 New->lth=0;
 New->rth=0;
 cout<<"\n Enter The Element ";
 cin>>New->data;
 if(root==NULL)
 {                    // Tree is not Created
  root=New;
  dummy=new node;
  dummy->data=-999;
  dummy->left=root;
  root->left=dummy;
  root->right=dummy;
 }
 else
 insert(root,New);
}
/*
-----------------------------------------------------------------
This function is for creating a binary search tree
-----------------------------------------------------------------
*/
void thread::insert(node *root,node *New)
{
 if(New->data<root->data)
  {
      if(root->lth==0)
       {
           New->left=root->left;
           New->right=root;
           root->left=New;
           root->lth=1;
       }
```


Scanned by CamScanner

```
      else
            insert(root->left,New);
}
if(New->data>root->data)
{
     if(root->rth==0)
     {
            New->right=root->right;
            New->left=root;
            root->rth=1;
            root->right=New;
     }
     else
      insert(root->right,New);
 }
}


/*
-----------------------------------------------------------------
The display function
-----------------------------------------------------------------
*/
void thread::display()
{
 void inorder(node *,node *dummy);
 void preorder(node *,node *dummy);
 if(root==NULL)
       cout<<"Tree Is Not Created";
 else
       {
         cout<<"\nInorder Traversal...";
         inorder(root,dummy);
         cout<<"\nPreorder Traversal...";
         preorder(root,dummy);
       }
}
node *search(node *root,node *dummy,int key,node **parent)
{
 node *temp;
 int flag=0;
 temp=root;
 while((temp!=dummy))
 {
  if(temp->data==key)
  {
      cout<<"\n The "<<temp->data<<" Element is Present";
```


Scanned by CamScanner

```
        flag=1;
        return temp;
    }
    *parent=temp;
    if(temp->data>key)
        temp=temp->left;
    else
        temp=temp->right;
    }
    return NULL;
}

/*
-----------------------------------------------------------------
The find function which calls the routine for searching an elemnt
-----------------------------------------------------------------
*/
void thread::find()
{

    int key;
    node *search(node *root,node *dummy,int key,node **parent);
    cout<<"\n Enter The Element Which You Want To Search";
    cin>>key;
    temp=search(root,dummy,key,&parent);
    if(temp==NULL)
     cout<<"\nElement is Not Present";
    else
     cout<<" It's Parent Node is "<<parent->data;
}

/*
-----------------------------------------------------------------
The delet function which calls the routine for deletion of an element
-----------------------------------------------------------------
*/
void thread::delet()
{

    int key;
    cout<<"\n Enter The Element U wish to Delete";
    cin>>key;
    del(root,dummy,key);
}

/*
-----------------------------------------------------------------
```


Scanned by CamScanner

```
This function is for deleting a node from binary search tree
There exists three possible cases for deletion of a node
-----------------------------------------------------------------
*/

void thread::del(node *root,node *dummy,int key)
{
 node *temp,*parent,*temp_succ;
 node *search(node *root,node *dummy,int key,node **parent);
 int flag=0;
 temp=search(root,dummy,key,&parent);
 if(root==temp)
 {
  cout<<"\n Its Root Node Which Can Not Be Deleted!!";
  return;
 }
 //deleting a node with two children
 if(temp->lth==1 && temp->rth==1)
 {
  parent=temp;
  temp_succ=temp->right;//Finding Inorder successor
  while(temp_succ->lth==1)
  {
      flag=1;
      parent=temp_succ;
      temp_succ=temp_succ->left;
  }
  if(flag==0)
  {
      temp->data=temp_succ->data;
      parent->right=temp_succ->right;
      parent->rth=0;
  }
  else//inorder successor is on left subbranch.
            // and it has to be traversed
  {
      temp->data=temp_succ->data;
      parent->rth=0;
      parent->lth=0;
      parent->left=temp_succ->left;
  }
  cout<<" Now Deleted it!";
  return;
 }
 //deleting a node having only one child
 //The node to be deleted has left child
 if(temp->lth==1 && temp->rth==0)
```


Scanned by CamScanner

```
{
 if(parent->left==temp)
 {
      (temp->left)->right=parent;
       parent->left=temp->left;
 }
 else
 {
      (temp->left)->right=temp->right;
      parent->right=temp->left;
 }
 temp=NULL;
 delete temp;
 cout<<" Now Deleted it!";
 return;
}

//The node to be deleted has right child
if(temp->lth==0 && temp->rth!=0)
{
 if(parent->left==temp)
 {
      parent->left=temp->right;
      (temp->right)->left=temp->left;
      (temp->right)->right=parent;
 }
 else
 {
      parent->right=temp->right;
      (temp->right)->left=parent;

 }
 temp=NULL;
 delete temp;
 cout<<" Now Deleted it!";
 return;
}
//deleting a node which is having no child
if(temp->lth==0 && temp->rth==0)
{
     if(parent->left==temp)
     {
      parent->left=temp->left;
      parent->lth=0;
     }
     else
      {
```


Scanned by CamScanner

```
        parent->right=temp->right;
        parent->rth=0;
      }
 cout<<" Now Deleted it!";
 return;
 }
}
/*
-----------------------------------------------------------------
The inorder function
-----------------------------------------------------------------
*/
void inorder(node *temp,node *dummy)
{
  while(temp!=dummy)
  {
      while(temp->lth==1)
       temp=temp->left;
  cout<<" "<<temp->data;
  while(temp->rth==0)
  {
  temp=temp->right;
      if(temp==dummy)
       return;
      cout<<" "<<temp->data;
  }
  temp=temp->right;
  }
}
/*
-----------------------------------------------------------------
The Preorder function
-----------------------------------------------------------------
*/
void preorder(node *temp,node *dummy)
{
  int flag=0;
  while(temp!=dummy)
  {
        if(flag==0)
         cout<<" "<<temp->data;
        if((temp->lth==1)&&(flag==0))
        {
            temp=temp->left;
        }
        else
        {
```


Scanned by CamScanner

```
            while(1)
            {
                if(temp->rth==1)
                {
                    flag=0;
                    temp=temp->right;
                    break;
                }
                else
                {
                    if(temp!=dummy)
                    {
                        flag=1;
                        temp=temp->right;
                        break;
                    }
                }//end else
            }//end while
        }//end else
    }//end while
}



/*
-----------------------------------------------------------------
The main function
-----------------------------------------------------------------
*/
int main()
{
  int choice;
  char ans='N';
  thread th;
  do
  {
      cout<<"\n\t Program For Threaded Binary Tree";
      cout<<"\n1.Create \n2.Display \n3.Search \n4.Delete";
      cin>>choice;
      switch(choice)
      {
       case 1:do
              {
               th.create();
                cout<<"\n Do u Want To enter More  Elements?(y/n)";
                cin>>ans;
              }while(ans=='y');
```


Scanned by CamScanner

```
                break;
        case 2:th.display();
                break;
        case 3:th.find();
                break;
        case 4:th.delet();
                break;
        }
 cout<<"\n\nWant To See Main Menu?(y/n)";
 cin>>ans;
 }while(ans=='y');
return 0;
}
```

Output

```
      Program For Threaded Binary Tree
1.Create
2.Display
3.Search
4.Delete 1

Enter The Element 10

Do u Want To enter More  Elements?(y/n)y

Enter The Element 8

Do u Want To enter More  Elements?(y/n)y

Enter The Element 7

Do u Want To enter More  Elements?(y/n)y

Enter The Element 9

Do u Want To enter More  Elements?(y/n)y

Enter The Element 12

Do u Want To enter More  Elements?(y/n)y

Enter The Element 11

Do u Want To enter More  Elements?(y/n)y

Enter The Element 13
```


Scanned by CamScanner

```
Do u Want To enter More  Elements?(y/n)n

Want To See Main Menu?(y/n)y

        Program For Threaded Binary Tree
1.Create
2.Display
3.Search
4.Delete 2

Inorder Traversal...  7  8  9  10  11  12  13
Preorder Traversal...  10  8  7  9  12  11  13
Want To See Main Menu?(y/n)n
```

4.8.4 Advantages and Disadvantages

Advantages of threaded binary tree

1. In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.
2. The threads are pointing to successor and predecessor nodes. This makes us to obtain predecessor and successor node of any node quickly.
3. There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

Disadvantage of threaded binary tree

1. Implementing threads for every possible node is complicated.

Review Question

1. *What is TBT ? What is advancement in TBT over BT ? Draw any suitable in-ordered TBT and traverse it in pre-order traversal.* **SPPU : Dec.-18, Marks 8**

4.9 Applications of Trees

Various applications of binary tree are

1. Decision Trees
2. Game Trees
3. Expression Trees

Scanned by CamScanner

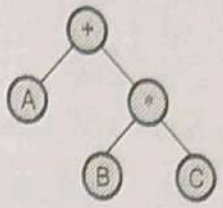
4.10 Expression Tree

SPPU : May-10,11, Dec.-11, Marks 10

An expression tree is a binary tree in which the operands are attached as leaf nodes and operators become the internal nodes.

For example

From expression tree :



Inorder traversal : A+B*C(Left-Data-Right)

Preorder traversal : +A*BC(Data-Left-Right)

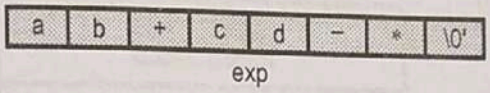
Postorder traversal : ABC*+ (Left-Right-Data)

If we traverse the above tree in inorder, preorder or postorder then we get infix, prefix or postfix expressions respectively.

Key Point *Inorder traversal of expression tree fives infix expression. The preorder traversal of expression tree gives prefix expression and post order traversal of expression tree gives postfix expression.*

Creation of an expression tree

Consider a postfix expression, stored in array **exp**



Now we will read each symbol from left to right one character at a time. If we read an operand then we will make a node of it and push it onto the stack. If we read operator then pop two nodes from stack, the first popped node will be attached as right child to operator node and second popped node will be attached as a left child to operator node. For the above given expression let us build a tree.

NULL	a	NULL

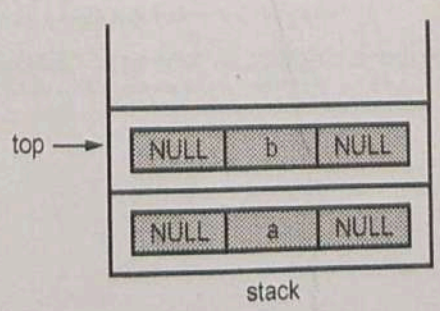
Push it onto the stack

NULL	b	NULL

Push it onto the stack

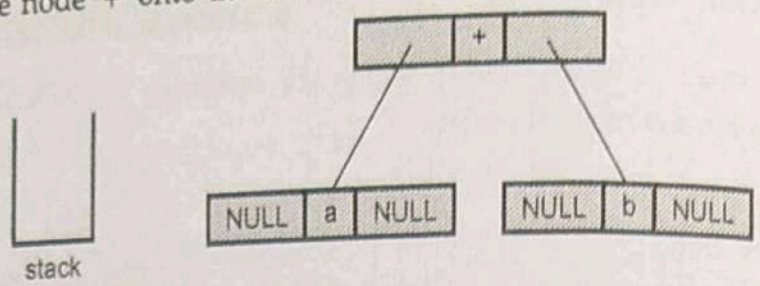
Then stack will be -

Now as we read '+' pop two operands and attach them as follows -

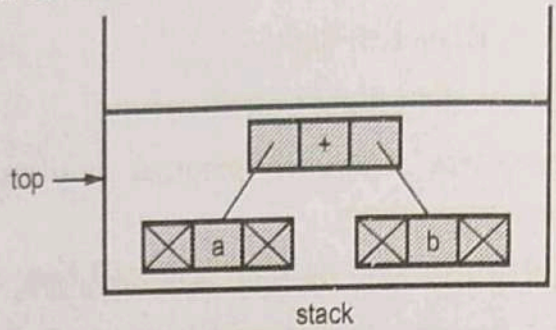


Scanned by CamScanner

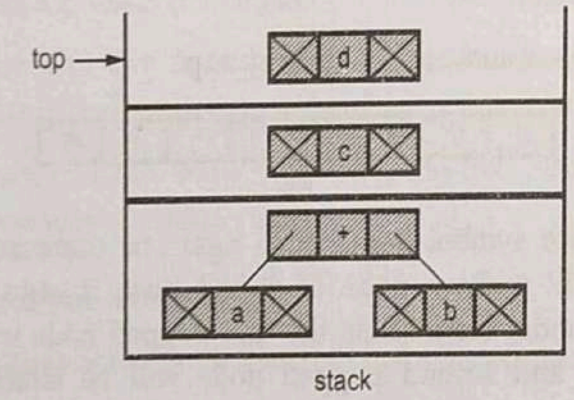
Now push the node '+' onto the stack.

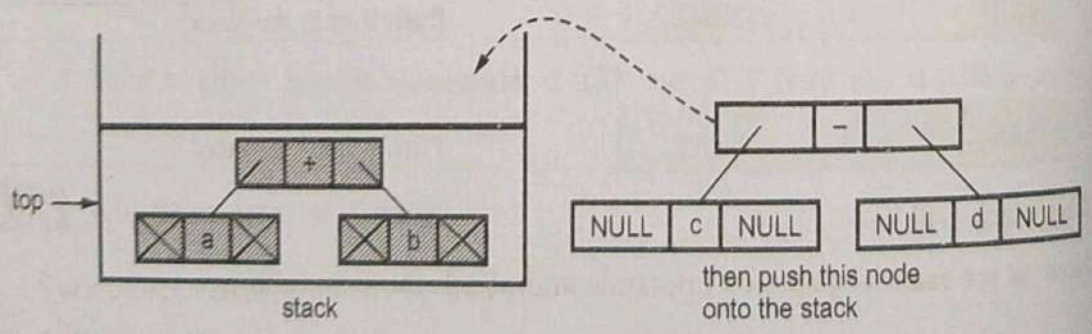


Next we read c and d. Hence stack will be -



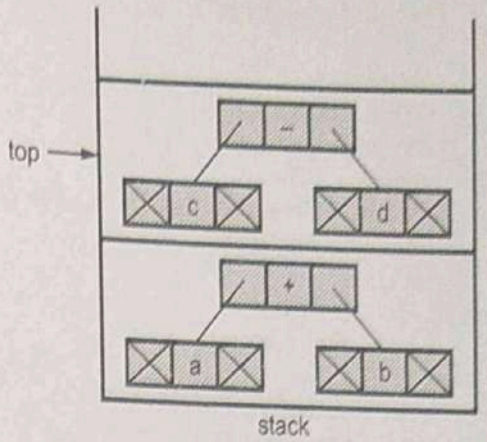
Now we read '–'. So pop two nodes

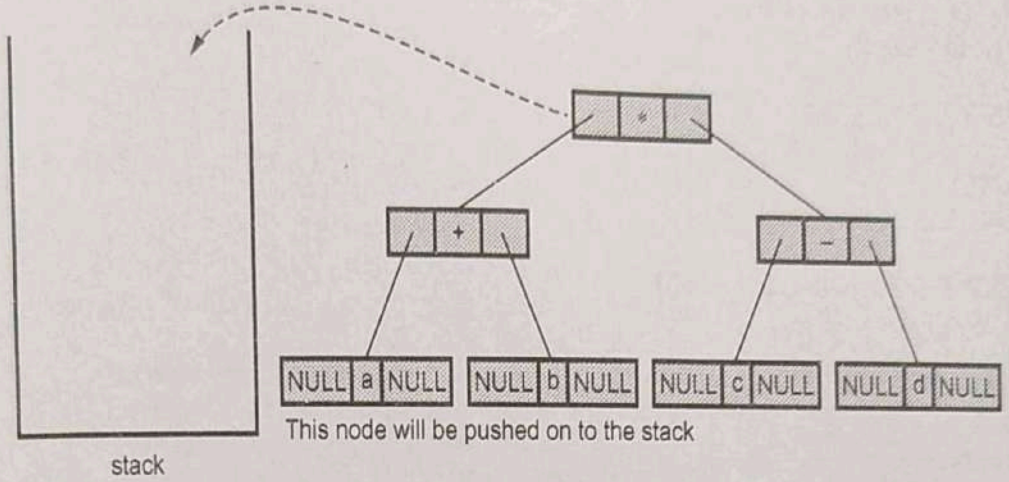




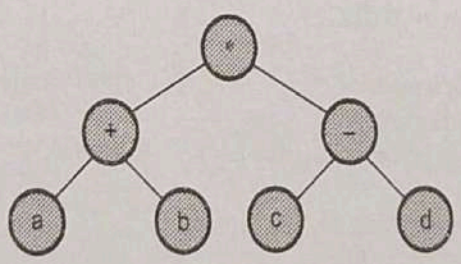
Scanned by CamScanner

Next we read '*'. Hence pop two nodes from the stack and attach them to * node.





As now we read '\0', pop the content of stack. The node which we will get is the root node of an expression tree.



Let us implement it

C++ Program

```
*****************************************************
Program for creating an expression tree and printing it using an inorder traversal
*****************************************************/
#include <iostream>
#include <ctype.h>
#define size 20
Using namespace std;
class EXP TREE
{private:
```


Scanned by CamScanner

```
        typedef struct node
        {
        char data;
        struct node *left;
        struct node *right;
        }btree;
        /*stack stores the operand nodes of the tree*/
        btree  *stack[size];
        int  top;
public:
        btree *root;
        EXP  TREE();
        void create(char exp[]);
        void push(btree *);
        void display(btree *root);
        btree* pop();
};
EXP  TREE::EXP  TREE()
{
 root=NULL;
 top=-1;//initializing stack
}
void EXP  TREE::create(char exp[])
{
 int pos;
 char ch;
 pos = 0;
 ch = exp[pos];
 while ( ch !='\0')
 {
        /* Create a new node */
 root = new btree;
 root -> left = root -> right = NULL ;
 root -> data = ch;
 if (isalpha( ch) ) /* is it a operand */
        push ( root); /* push operand */
 else if ( ch=='+' ||ch=='-'||ch=='*'||ch=='/')
{
  /* it is operator, so pop two nodes from stack
         set first node as right child and
         set second as left child and push the
        operator node on to the stack
  */
        root->right = pop();
        root->left = pop();
        push(root);
 }
 else
        cout<<"Invalid character in expression\n";
 pos ++;
  ch = exp[pos]; /* Read next character */
}
root= pop();
```


Scanned by CamScanner

```
}
void EXP TREE::push(btree *Node)
{
      if ( top+1 >= size )
      cout<<"Error: Stack is Full\n";
      top++;
      stack[top] = Node;
}

btree* EXP TREE::pop()
{
      btree *Node;
      if ( top == -1 )
        cout<<"Error: Stack is Empty\n";
      Node = stack[top];
      top--;
      return(Node);
}
void EXP TREE::display(btree *root)
{
      btree *temp;
      temp = root;
      if (temp != NULL )
      {
              display(temp->left);
              cout<<" "<<temp->data;
              display(temp->right);
}
}

int main()
{

 char  exp[80]; /* exp stores postfix expression */
 EXP TREE obj;
cout<<"Enter the postfix expression\n";
 cin>>exp;
 obj.create(exp);
 cout<<"\n The Tree is created...\n";
 cout<<"\n The inorder traversal of it is \n ";
 obj.display(obj.root);
 return 0;
}
```

Output

```
Enter the postfix expression
ab+cd-*
 The Tree is created...

 The inorder traversal of it is
  a + b * c - d
```


Scanned by CamScanner

Example 4.10.1 *Parenthesis are not given in an expression in prefix or postfix. Justify. Draw the expression tree and find the infix and postfix expressions for the following prefix expression.*

*–AB + *CD/EF

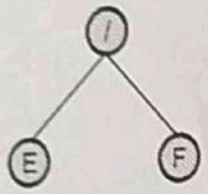
SPPU : Dec.-11, Marks 8

Solution : Basically parenthesis indicate the operations which need to be carried out first. So in case of postfix or prefix expression they are actually conversions of the original standard equation, where the brackets have already been taken into consideration and the formed prefix/postfix expression of a given mathematical statement.

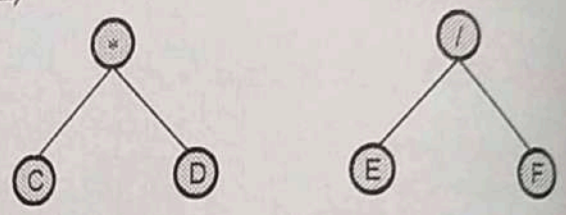
Given expression

* – AB + * CD/EF

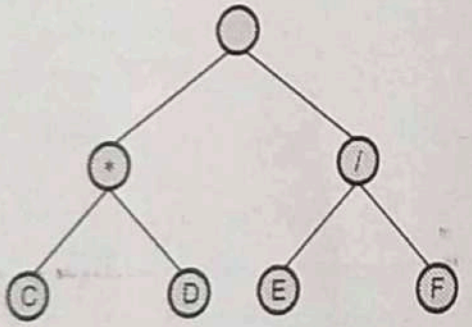
I)



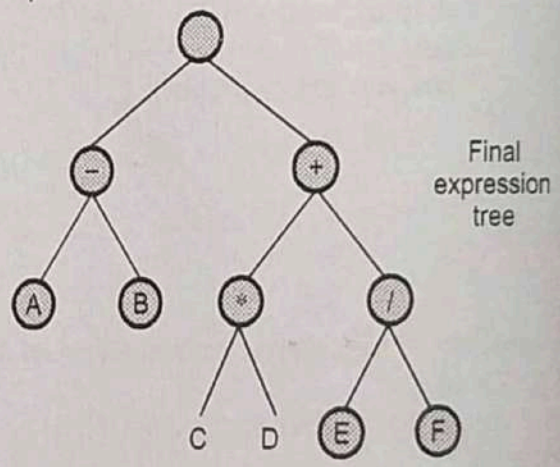
II)



III)



IV)



Infix expression = (A – B) * [(C * D) + (E/F)]

Postfix expression = AB – CD * + EF / *

Review Questions

1. *Write an algorithm to accept a prefix expression and construct its binary tree and perform recursive and non recursive inorder traversal of tree.* **SPPU : May-10, Marks 10**
2. *Write an algorithm to accept a postfix expression and construct its binary tree and perform recursive and non recursive in-order traversal of the tree.* **SPPU : May-11, Marks 10**

❑❑❑

Scanned by CamScanner

5 Graph and Symbol Table

Syllabus

Graph - *Concept and terminologies, Graph as an ADT, Representation of graphs using adjacency matrix and adjacency list, Breadth First Search traversal, Depth First Search traversal, Prim's and Kruskal's algorithms for minimum spanning tree, Shortest path using Dijkstra's algorithm, topological sorting.*

Symbol Table - *Notion of Symbol Table, OBST, AVL Tree.*

Heap - *Heap data structure, Min and Max Heap, Heap sort, applications of heap.*

Contents

5.1 Introduction to Graph ..... **May-13,19,** ..... Marks 6

5.2 Representation of Graphs using Adjacency Matrix and Adjacency ..... **May-10,13,15,19, Dec.-11,15,** Marks 8

5.3 Creation of Graph ..... **Dec.-10,** ..... Marks 8

5.4 Breadth First Search Traversals ..... **May-12,14,16,** ..... Marks 10

5.5 Depth First Search Traversal

5.6 Applications of Graph

5.7 Minimum Spanning Tree ..... **May-10,11,13,14,15,16,** .... **Dec.-14,** ..... Marks 8

5.8 Shortest Path using Dijkstra's Algorithm ..... **May-19,** ..... Marks 6

5.9 Topological Sorting ..... **May-13,16, Dec.-19,** ..... Marks 6

5.10 Notion of Symbol Table ..... **May-12,** ..... Marks 6

5.11 OBST ..... **Dec.-09, May-16,** ..... Marks 10

5.12 AVL Trees ..... **Dec.-05,07,08,14,19,** .... **May-10,11,14,15,16,18,19,** .. Marks 10

5.13 Heap Data Structure ..... **Dec.-15, May-16,19,** ..... Marks 8

5.14 Heap Sort ..... **May-11,13,14,** ..... **Dec.-11,14,19,** ..... Marks 12

5.15 Applications of Heap

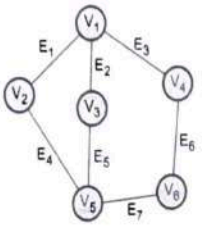
5.1 Concept and Terminologies

SPPU : May-13,19 Marks 6

A graph is a collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

- **Vertices** are nothing but the nodes in the graph.
- Two adjacent vertices are joined by **edges**.
- Any graph is denoted as G = {V, E}.

For example :



$$G = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\}$$

Fig. 5.1.1 Graph G

5.1.1 Graph as an ADT

The abstract data type for graph is as given below.

AbstractDataType Graph

{

Instances

Graph is a collection of vertices and edges.

Operations

1) tot_vertices ( ) : This function returns total number of vertices.
2) tot_edges ( ) : This function returns total number of edges.
3) Is_edge (V_1, V_2) : This function returns true if there exists an edge between V_1 and V_2.
4) In_degree ( ) : This function returns total number of edges that are incident to the vertex.
5) Out_degree ( ) : This function returns out degree of the vertex.
6) Display ( ) : This function displays the graph in DFS or in BFS manner.

}

5.1.2 Comparison between Graph and Trees

Sr. No.	Graph	Tree
1.	Graph is a non-linear data structure.	Tree is a non-linear data structure.
2.	It is a collection of vertices/nodes and edges.	It is a collection of nodes and edges.
3.	Each node can have any number of edges.	General trees consist of the nodes having any number of child nodes. But in case of binary trees every node can have at the most two child nodes.
4.	There is no unique node called root in graph.	There is a unique node called root in trees.
5.	A cycle can be formed.	There will not be any cycle.
6.	Applications : For finding shortest path in networking graph is used.	Applications : For game trees, decision trees, the tree is used.

5.1.3 Types of Graph

Basically graphs are of two types -

1. Directed graphs 2. Undirected graphs.

In the directed graph the **directions** are shown on the edges. As shown in the Fig. 5.1.2, the edges between the vertices are ordered. In this type of graph, the edge E_1 is in between the vertices V_1 and V_2. The V_1 is called head and the V_2 is called the tail. Similarly for V_1 head the tail is V_3 and so on.

We can say E_1 is the set of (V_1, V_2) and not of (V_2, V_1).

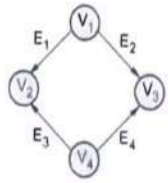


Fig. 5.1.2 Directed graph

Similarly, in an undirected graph, the **edges are not ordered**. Please see the Fig. 5.1.3 for clear understanding of undirected graph. In this type of graph the edge E_1 is set of (V_1, V_2) or (V_2, V_1).

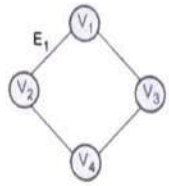


Fig. 5.1.3 Undirected graph

Similarly the object shown in the Fig. 5.1.4 is a multi-graph.

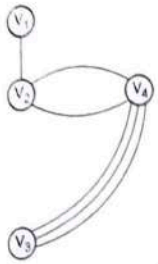


Fig. 5.1.4 A multi-graph

5.1.4 Properties of Graph

1) Complete graph : If an undirected graph of n vertices consists of $\frac{n(n-1)}{2}$ number of edges then that graph is called a complete graph.

For example : The graph shown in Fig 5.1.5 is a complete graph.

Here $n = 4$

$\therefore e = \frac{n(n-1)}{2} = \frac{4(3)}{2} = 6$

Theorem : A complete graph with n vertices has $\frac{n(n-1)}{2}$ number of edges.

Proof : This can be proved by principle of mathematical induction.

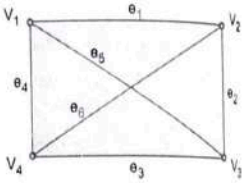


Fig. 5.1.5 Complete graph

Basis of induction : This theorem is true for $n = 1$ i.e. a graph with single node because when $n = 1$, total number of edges $= 1(1-1)/2 = 0$.

For instance :

Fig. 5.1.6 Complete graph when n = 1

Induction hypothesis : We assume that for k vertices it is true that total number of edges $= \frac{k(k-1)}{2}$.

Inductive step : We have to show that the theorem is also true for $(n + 1)$ vertices.

Let, G(n) be the graph with n vertices. If we add one more vertex to G(n) then to make this graph complete, we need to add edges to this newly added vertex from all the previous n vertices then total number of edges in G(n+1) will be

$$\text{Edges} = \text{Edges in } G(n) + n = \frac{n(n-1)}{2} + n = \frac{n(n-1)+2n}{2}$$

$$= \frac{n^2 - n + 2n}{2} = \frac{n^2 + n}{2}$$

$$\text{Total edges in } G(n+1) = \frac{n(n+1)}{2} \quad \text{...(5.1.1)}$$

If we put $n + 1 = k$ then $n = k - 1$. Hence we can rewrite equation (5.1.1) as,

$$= \frac{(k-1)k}{2} \quad \text{i.e.} \quad \frac{k(k-1)}{2}$$

This is true by induction hypothesis. Hence total number of edges in G (n+1) graph is

$$= \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} \text{ is true.}$$

Thus it is proved by principle of mathematical induction that total number of edges in complete graph is $\frac{n(n-1)}{2}$.

2) Subgraph : A subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' are proper subset of the set of edges of G.

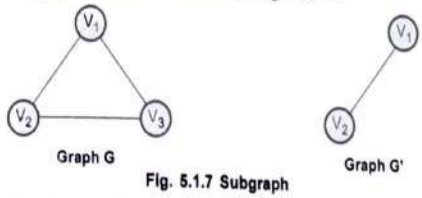


Fig. 5.1.7 Subgraph

3) Connected graph : An undirected graph is said to be connected if for every pair of distinct vertices V_i and V_j in V(G) there is an edge V_i to V_j in G.

Note that there is path from any two vertices.

For example :

- $V_1 - V_2$ • $V_2 - V_1$ • $V_3 - V_1$ • $V_4 - V_3 - V_1$
- $V_1 - V_3$ • $V_2 - V_1 - V_3$ • $V_3 - V_1 - V_2$ • $V_4 - V_2$
- $V_1 - V_4$ • $V_2 - V_4$ • $V_3 - V_4$ • $V_4 - V_3$

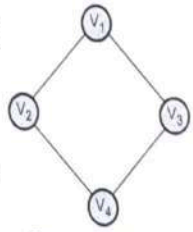
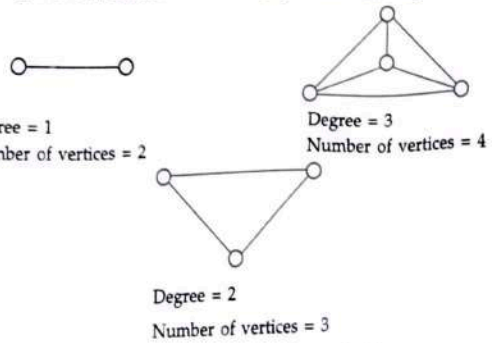


Fig. 5.1.8 Connected graph

Example 5.1.1 *Prove that the number of odd degree vertices in a connected graph should be even.*

Solution : To prove this, consider connected graphs with odd degree vertices.



These figures show clearly that in a connected graph there are even number of vertices with odd degree.

Weighted graph

A weighted graph is a graph which consists of weights along its edges.

For example :

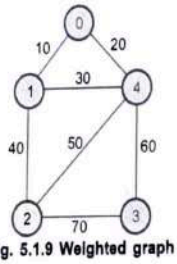


Fig. 5.1.9 Weighted graph

Path : A path is denoted using sequence of vertices and there exists an edge from one vertex to the next vertex.

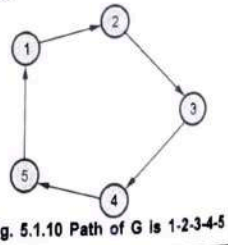


Fig. 5.1.10 Path of G is 1-2-3-4-5

Cycle : A closed walk through the graph with repeated vertices, mostly having the same starting and ending vertex is called a cycle.

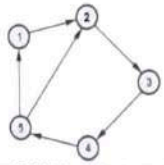


Fig. 5.1.11 Cycle 2-3-4-5-2 or 1-2-3-4-5-1

Component : The maximal connected subgraph of a graph is called component of a graph.

For example : Following are 3 components of a graph.

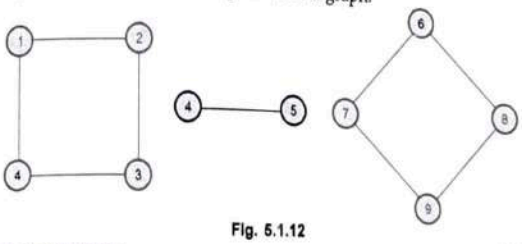
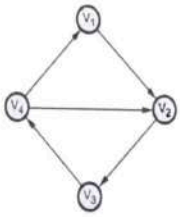


Fig. 5.1.12

In-degree and out-degree

The degree of vertex is the number of edges associated with the vertex.

In-degree of a vertex is the number of edges that incident to that vertex. Out-degree of the vertex is total number of edges that are going away from the vertex.



Vertices	In-degree	Out-degree
V_1	1	1
V_2	2	1
V_3	1	1
V_4	1	2

Fig. 5.1.13 Self loop

Self loop : Self loop is an edge that connects the same vertex, to itself.

Isolated vertex : In a graph the vertex with degree 0 is called **isolated** vertex. **For example :**

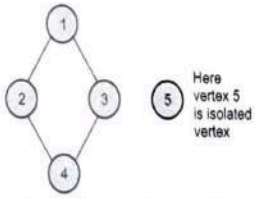
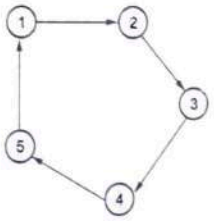


Fig 5.1.14 : Graph containing isolated vertex

Example 5.1.2 *Define the following terms with respect to graph : i) Path ii) Adjacant vertices iii) Cycle iv) Loop v) Degree of vertex vi) Connected graph.*

SPPU : May-19, Marks 6

Solution : i) Path : A path is denoted using sequence of vertices and there exists an edge from one vertex to next vertex.

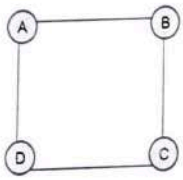


Path is 1 - 2 - 3 - 4 -5.

Graph G

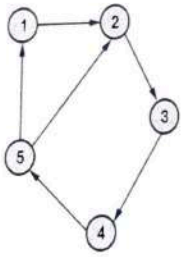
ii) Adjacent vertices : In a graph, two vertices are adjacent if they are joined by an edge.

For example :



Here verties A and B, A and D, B and C, D and C are adjacent vertices.

iii) Cycle : A closed walk through the graph with reapeated verties, mostly having same starting and ending vertex is called a cycle.



Cycle 2-3-4-5-2
Or 1-2-3-4-5-1

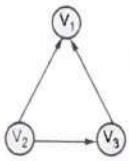
Graph G

iv) Loop : The loop is an edge that connects the same vertex to itself.

For example :

v) Degree of vertex : The degree of vertex is number of edges asscoaited with the vertex.

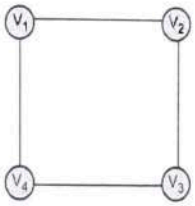
For example :



In Degree of V_1 = 2
Out Degree of V_1 = 0

vi) Connected Graph : Connected graph is a graph in which there is a path from one vertex to any other vertex.

For example :



Review Question

1. *Define : i) Complete graph ii) Path* **SPPU : May-13, Marks 4**

5.2 Representation of Graphs using Adjacency Matrix and Adjacency List

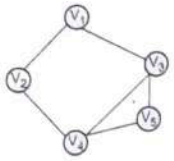
SPPU : May-10,13,15,19, Dec.-11,15, Marks 8

Using Adjency matrix and adjemcy list :

Normally a graph can be represented by two representations and those are

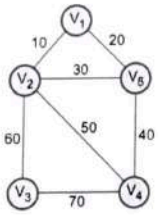
1) Adjacency matrix : In this representation, matrix or 2 dimensional array is used to represent the graph.

Consider a graph G of n vertices and the matrix M. If there is an edge present between vertices V_i and V_j then M[i][j] = 1 else M[i][j] = 0. Note that for an undirected graph if M[i][j] = 1 then for M[j][i] is also 1. Here are some graphs shown by adjacency matrix.



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Fig. 5.2.1 An adjacency matrix representation



Weighted graph

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 10 & 0 & 0 & 20 \\ 2 & 10 & 0 & 60 & 50 & 30 \\ 3 & 0 & 60 & 0 & 70 & 0 \\ 4 & 0 & 50 & 70 & 0 & 40 \\ 5 & 20 & 30 & 0 & 40 & 0 \end{array}$$

Adjacency matrix

Fig. 5.2.2 Adjacency matrix representation for weighted graph

Adjacency matrix for weighted graph

- In the weighted graph, weights or distances are given along every edge. Hence in an adjacency matrix representation any edge which is present between vertices V_i and V_j is denoted by its weight. Hence M[i][j] = Weight of edge.

- If there is no edge between V_i and V_j then, M[i][j] = 0.

2) Adjacency list : In this representation, a **linked list** is used to represent a graph.

There are two methods of representing graph using adjacency list.

Method 1 : As we know the graph is a set of vertices and edges, we will maintain the two structures, for vertices and edges respectively.

For example : Now in Fig. 5.2.3, graph has the nodes as a, b, c, d, e. So we will maintain the linked list of these head nodes as well as the adjacent nodes.

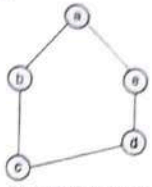


Fig. 5.2.3 Graph G

```
typedef struct head
{
  char data;
  struct head *down;
  struct head *next;
};
typedef struct node
{
int ver;
struct node *link;
};
```

Explanation : This is purely the adjacency list graph. The down pointer helps us to go to each node in the graph where as the next node is for going to adjacent node of each of the head node.

Method 2 : In this method of representing the adjacency list, we take mixed data structure. That means instead of taking the head list as a linked list we will take an array of head nodes. So only one 'C' structure will be there representing the adjacent nodes. See the Fig. 5.2.5 representing the adjacency list for the above graph. First let us see the 'C' structure.

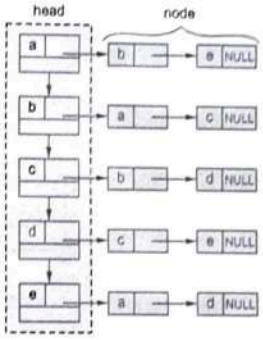


Fig. 5.2.4 Adjacency list (Method 1)

```
typedef struct node1
{
    char vertex;
    struct node1*nex ;
}node ;
node *head [10] ;
```

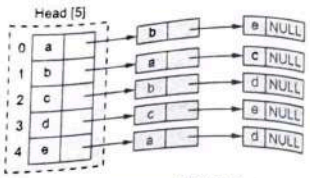


Fig. 5.2.5 Adjacency list (Method 2)

Explanation : This is the graph which can be represented with the array and linked list data structures. Array is used to store the head nodes. The node strucutre will be the same through out.

Example 5.2.1 *Define a graph. For the given adjacency matrix draw the graph and its adjacency list.*

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	0	0	1	0	0	0
C	1	0	0	1	0	1	0	0
D	0	0	1	0	0	0	0	1
E	0	1	0	0	0	0	1	0
F	0	0	1	0	0	0	1	1
G	0	0	0	1	0	1	0	0
H	0	0	0	1	0	1	0	0

Find all the nodes adjacent to node A, node F and node G. **SPPU : Dec.-11, Marks 8**

Solution : Graph for given adjacency matrix :

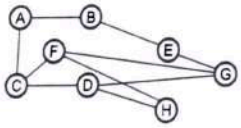


Fig. 5.2.6

Nodes adjacent to A = B and C

Nodes adjacent to F = C, H, G

Nodes adjacent to G = E, D, F

Adjacency list :

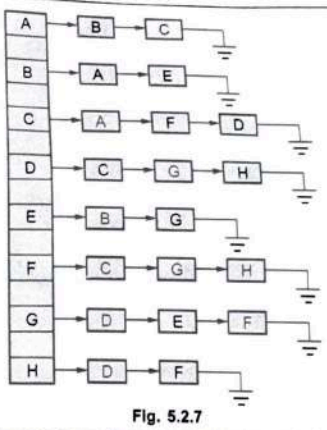


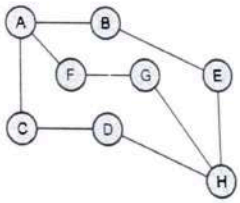
Fig. 5.2.7

Example 5.2.2 *Define a graph. For the given adjacency matrix draw the graph and its adjacency list :*

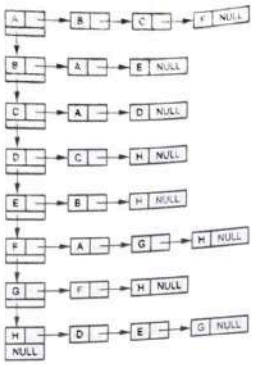
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	1	0	0
B	1	0	0	0	1	0	0	0
C	1	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
E	0	1	0	0	0	0	0	1
F	1	0	0	0	0	0	1	0

Find all the nodes adjacent to node A, node F and node G. **SPPU : May-15, Marks 8**

Solution : Graph : (Refer section 5.1)

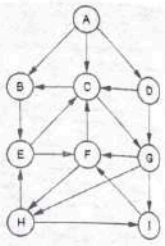


Adjacency list :



Example 5.2.3 *Consider the graph G given in Fig. below. Draw the adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to H with minimum stops. Find the minimum path P from A to H given that every edge has length of 1.*

SPPU : Dec.-15, Marks 6

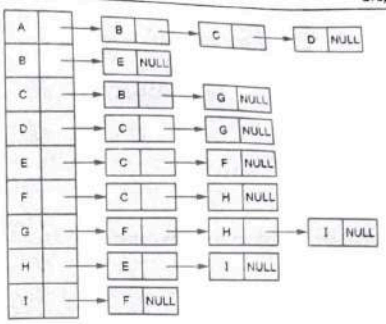


Solution : Adjacency List

The paths form A to H by considering every edge has lenght of 1 are listed as below :

Path	Total Length
A - B - E - F - H	4
A - B - E - F - C - G - H	6
A - C - G - H	3
A - C - G - F - H	4
A - D - C - G - H	4
A - D - G - H	3

The paths with minimum stops are (i) A - C - G - H (ii) A - D - G - H with path length 3.



Example 5.2.4 *Draw the directed graph for the adjacency matrix representation given below :*

	A	B	C	D	E	F
A	0	3	4	0	2	1
B	0	0	2	0	0	3
C	0	0	0	2	6	1
D	2	6	1	0	1	2
E	0	0	0	0	0	3
F	0	0	0	0	0	0

SPPU : May-19, Marks 6

Solution : Adjacency List

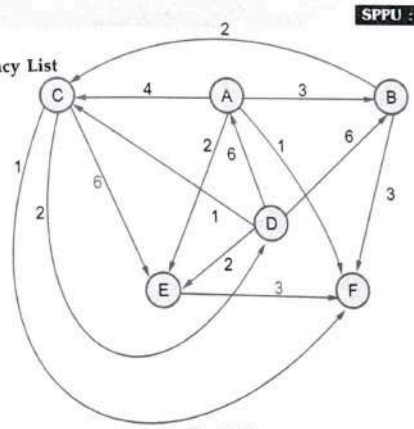


Fig. 5.2.8

Review Question

1. *Describe various ways in which graphs are represented.*

SPPU : May-10, Marks 6, May-13, Marks 4

5.3 Creation of Graph

SPPU : Dec.-10, Marks 8

Graph creation using adjacency matrix

Creation of graph using adjacency matrix is quite simple task. The adjacency matrix is nothing but a two dimensional array. The algorithm for creation of graph using adjacency matix will be as follows :

1) Declare an array of M[size] [size] which will store the graph.
2) Enter how many nodes you want in a graph.
3) Enter the edges of the graph by two vertices each, say V_i, V_j indicates some edge.
4) If the graph is directed set M[i][j] = 1. If graph is undirected set M[i][j] = 1 and M[j][i] = 1 as well.
5) When all the edges for the desired graph is entered print the graph M[i][j].

```
void Gbfs::create()
{
    int ch,flag;   //flag for directed or undirected graph
    flag = TRUE;
    n=0;
    cout<<"\n\t\t This is a Progrm To Create a Graph"
    cout<<"\n\t\t The Display Is In Breadth First Manner";
    cout<<"\n Press d/D for directed and u/U for undirected graph\n";
    ch = toupper(getche());
    if ( ch == 'D')
      flag = FALSE;

    do
    {
        cout<<"\nEnter the Edge of a graph ";
        cout<<"(And for termination type-99)\n";
        cin>>v1>>v2;
        if ( v1 == -99)
          break;
        if ( v1 >= size || v2 >= size)
            error("Invalid Vertex Value\n" );
        else
          g[v1][v2] = TRUE;
          if (flag)
              g[v2][v1] = TRUE;
```



```
    n++;          //for keeping track of total number of nodes
    }while(1);
}
```

Graph creation using adjacency list

1. Declare node structure for creating adjacency list.
2. Initialize an array of nodes. This array will act as head nodes. Say '*head [10]'. The index of head [ ] will be the starting vertex.
3. The create function will create the adjacency list for given graph G as follows -

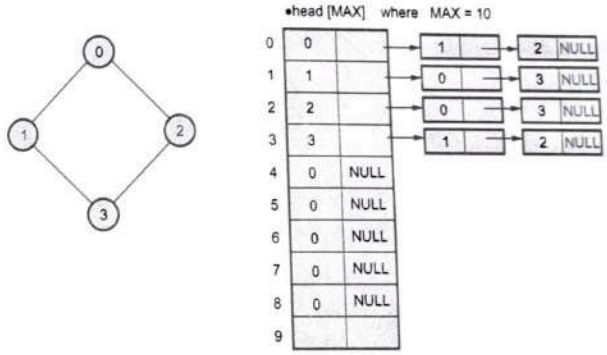


Fig. 5.3.1 Graph and its adjacency list representation

The pseudo code for create function is as follows :

```
void Lgraph::create()
{
    int V1, V2;
    char ans='y';
    node *New,*first;
    cout<<"\n\nEnter the vertices no. beginning with 0";
    do
    {
      cout<<"\nEnter the Edge of a graph \n";
          cin>>V1>>V2;
          if ( V1 >= MAX || V2 >= MAX)
                cout<<"Invalid Vertex Value\n" ;
          else
            {
                // creating link from V1 to V2
                New = new node;
```

```
            if ( New == NULL )
              cout<<"Insufficient Memory\n";
            New -> vertex = V2;
            New -> next = NULL;
            first = head[V1];
            if ( first == NULL )
              head[V1] = New;
            else
             {
                 while ( first -> next != NULL )
                   first = first -> next;
                 first -> next = New;
            }
        // creating link from V2 to V1
            New = new node;
            if ( New == NULL )
              cout<<"Insufficient Memory\n";
            New -> vertex = V1;
            New -> next = NULL;
            first = head[V2];
            if ( first == NULL )
              head[V2] = New;
            else
             {
               while ( first -> next != NULL )
                   first = first-> next;
              first -> next = New;
            }
        }
    cout<<"\nWant to add more edges?(y/n)";
    ans=getche();
    }while(ans=='y');
}
```

Example 5.3.1 *What is graph ? Draw how the following graph can be represented using linked organization :* **SPPU : Dec.-10, Marks 8**

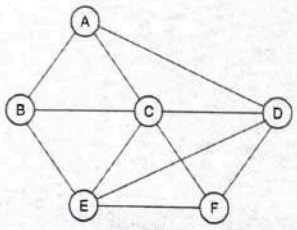


Fig. 5.3.2

Solution : Graph : The graph is a collection of nodes and vertices. The linked representation of given graph is -

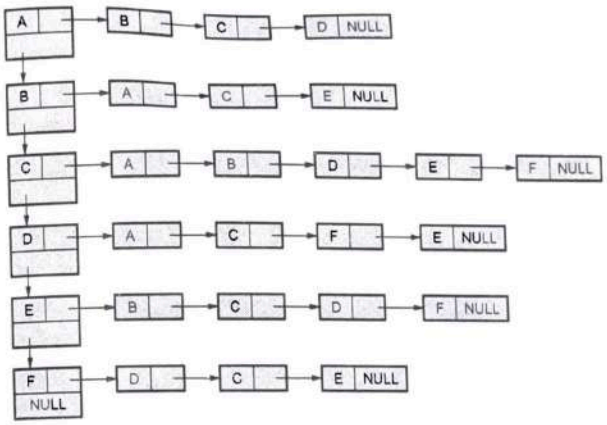


Fig. 5.3.3

5.4 Breadth First Search Traversal

SPPU : May-12,14,16, Marks 10

- In BFS we start from some vertex and find all the adjacent vertices of it. This process will be repeated for all the vertices so that the vertices lying on same breadth get printed.
- For avoiding repetition of vertices, we maintain array of **visited** nodes.
- A **queue** data structure is used to store adjacent vertices.

Algorithm :

1. Create a graph. Depending on the type of graph i.e. directed or undirected set the value of the flag as either 0 or 1 respectively.
2. Read the vertex from which you want to traverse the graph say V_i.
3. Initialize the visited array to 1 at the index of V_i.
4. Insert the visited vertex V_i in the queue.

5. Visit the vertex which is at the front of the queue. Delete it from the queue and place its adjacent nodes in the queue.
6. Repeat the step 5, till the queue is not empty.
7. Stop.

1. BFS by Adjacency Matrix :

C++ Program

```
/*****************************************************************
Program to create a Graph. The graph is represented using
Adjacency Matrix. The Program supports both undirected and
Directed Graphs and traverse the graph in Breadth First Search
Order.
*****************************************************************/

#include <iostream>

#include<ctype.h>

#define    size  20
#define    TRUE  1
#define    FALSE  0
using namespace std;
class Gbfs
{
 private:
 int g[size][size],v1,v2;
 int visit[size],Q[size];
 int front,rear;
 public:
 int n;
 Gbfs();
 void error(char *);
 void create(),bfs(int);
 void display();
 ~Gbfs();
};
/*
-----------------------------------------------------------------
The costructor defined
-----------------------------------------------------------------
*/
Gbfs::Gbfs()
{
 for ( v1 = 0; v1 < size; v1++)
      for ( v2 = 0; v2 < size; v2++)
```



```
          g[v1][v2] = FALSE;
 for ( v1 = 0; v1 < size; v1++)
      visit[v1] = FALSE;
}
/*
-----------------------------------------------------------------
The destructor defined
-----------------------------------------------------------------
*/
Gbfs::~Gbfs()
{
 for ( v1 = 0; v1 < size; v1++)
 {
      for ( v2 = 0; v2 < size; v2++)
      g[v1][v2]=FALSE;
 }
 for ( v1 = 0; v1 < size; v1++)
 visit[v1] = FALSE;
}
/*
-----------------------------------------------------------------
The display defined
-----------------------------------------------------------------
*/
void Gbfs::display()
{
 for ( v1 = 0; v1 < n; v1++)
 {
      for ( v2 = 0; v2 < n; v2++)
           cout<<" "<<g[v1][v2];
      cout<<endl;
 }
}
/*
-----------------------------------------------------------------
      The error function
-----------------------------------------------------------------
*/
void Gbfs::error( char *Msg )
{
      cout<<"\n"<<Msg;
      cout<<"Press any key to abort\n";
      getch();
      exit(1);
}
/*
```

```
-----------------------------------------------------------------
    The create function
-----------------------------------------------------------------
*/
void  Gbfs::create()
{
    int ch,flag;    //flag for directed or undirected graph
    flag = TRUE;
    n=0;
    cout<<"\n\t\t This is a Progrm To Create a Graph";
    cout<<"\n\t\t The Display Is In Breadth First Manner";
    cout<<"\n Press d/D for directed and u/U for undirected graph\n";
    ch = toupper(getche());
    if ( ch == 'D')
      flag = FALSE;

    do
    {
        cout<<"\nEnter the Edge of a graph ";
        cout<<"(And for termination type -99)\n";
        cin>>v1>>v2;
        if ( v1 == -99)
          break;
        if ( v1 >= size || v2 >= size)
            error("Invalid Vertex Value\n" );
        else
          g[v1][v2] = TRUE;
          if ( flag )
              g[v2][v1] = TRUE;
    n++;//for keeping track of total number of nodes
    }while(1);
}

/*
-----------------------------------------------------------------
    The bfs function
-----------------------------------------------------------------
*/
void Gbfs::bfs(int v1)
{
    int v2;

    visit[v1] = TRUE;
    front = rear = -1;
    Q[++rear] = v1;
    while ( front != rear )
    {
```



```
        v1 = Q[++front];
        cout<<"\n"<<v1;
        for ( v2 = 0; v2 < n; v2++)
        {
            if ( g[v1][v2] == TRUE && visit[v2] == FALSE )
            {
                Q[++rear] = v2;
                visit[v2] = TRUE;
            }
        }
    }
}
/*
-----------------------------------------------------------------
The main Function
-----------------------------------------------------------------
*/
int main()
{
    int v1;
    Gbfs gr;
    gr.create();
    cout<<"The Adjacency Matrix for the graph is "<<endl;
    gr.display();
    cout<<"Enter the Vertex from which you want to traverse ";
    cin>>v1;
    if ( v1 >= size )
     gr.error("Invalid Vertex\n");
    cout<<"The Breadth First Search of the Graph is \n";
    gr.bfs(v1);
    return (0);
}
```

Output

```
        This is a Progrm To Create a Graph
        The Display Is In Breadth First Manner
Press d/D for directed and u/U for undirected graph
u
Enter the Edge of a graph (And for termination type -99)

0 1

Enter the Edge of a graph (And for termination type -99)
0 2

Enter the Edge of a graph (And for termination type -99)
1 3
```

```
Enter the Edge of a graph (And for termination type -99)
2 3

Enter the Edge of a graph (And for termination type -99)
-99 -99
The Adjacency Matrix for the graph is
 0 1 1 0
 1 0 0 1
 1 0 0 1
 0 1 1 0
Enter the Vertex from which you want to traverse 0
The Breadth First Search of the Graph is
0
1
2
3
```

Explanation of logic of BFS

In BFS the queue is maintained for storing the adjacent nodes and an array 'visited' is maintained for keeping the track of visited nodes. i.e. once a particular node is visited it should not be revisited again. Let us see how our program works :

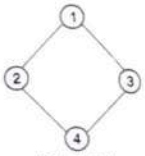
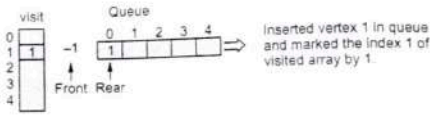
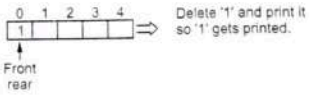


Fig. 5.4.1

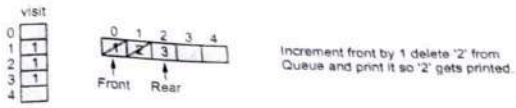
Step 1 : Start with vertex 1.



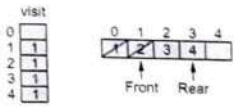
Step 2 :



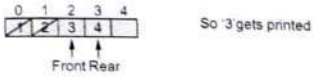
Step 3 : Find adjacent vertices of vertex 1 and mark them as visited, insert those in Queue.



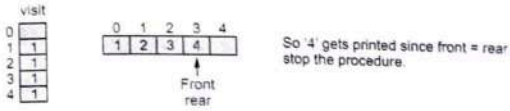
Step 4 : Find adjacent to '2' and insert those nodes in Queue as well as mark them as visited.



Step 5 : Increment front and delete the node print it.



Step 6 : Find adjacent to '3' i.e. 4 check whether it is marked as visited. If it is marked as visited do not insert in the queue.



Increment front, delete the node from Queue and print it.

So output will be - BFS for above graph as

1 2 3 4

2. BFS by Adjacency List

C++ Program

```
/*****************************************************************
    Program to create a Graph. The graph is represented using
    Adjacency List and traversing the graph in Breadth First Search
    Order.
*****************************************************************/

#include<iostream>
```

```
#define    MAX    10
#define    TRUE    1
#define    FALSE   0
using namespace std;
class Lgraph
{
  private:
      typedef struct node1
            {
                int vertex;
                struct node1 *next;
            }node;
            node *head[MAX];
            int visited[MAX];
            int Queue[MAX],front,rear;
  public:
      Lgraph();
      void create(),bfs(int);
};

Lgraph::Lgraph()//constructor defined
{
 int V1;
 for ( V1 = 0; V1 < MAX; V1++)
   visited[V1] = FALSE;
 front = rear = -1;
 for ( V1 = 0; V1 < MAX; V1++)
       head[V1] = NULL;
 }
 /*
 ---------------------------------------------------------------
 The create function
 ---------------------------------------------------------------
 */
 void Lgraph::create()
 {
       int V1, V2;
       char ans='y';
       node *New,*first;
       cout<<"\n\nEnter the vertices no. beginning with 0";
       do
       {
         cout<<"\nEnter the Edge of a graph \n";
         cin>>V1>>V2;
         if ( V1 >= MAX || V2 >= MAX)
```



```
                cout<<"Invalid Vertex Value\n" ;
         else
          {
                // creating link from V1 to V2
                New = new node;
                if ( New == NULL )
                  cout<<"Insufficient Memory\n";
                New -> vertex = V2;
                New -> next = NULL;
                first = head[V1];
                if ( first == NULL )
                  head[V1] = New;
                else
                 {
                while ( first -> next != NULL )
                first = first-> next;
                first -> next = New;
                }
          // creating link from V2 to V1
                New = new node;
                if ( New == NULL )
                  cout<<"Insufficient Memory\n";
                New -> vertex = V1;
                New -> next = NULL;
                first = head[V2];
                if ( first == NULL )
                  head[V2] = New;
                else
                 {
                  while ( first -> next != NULL )
                        first = first -> next;
                  first -> next = New;
                 }
          }
       cout<<"\nWant to add more edges?(y/n)";
       cin>>ans;
       }while(ans=='y');
 }
 /*
 ---------------------------------------------------------------
 The bfs Function
 ---------------------------------------------------------------
 */
 void Lgraph::bfs(int V1)
 {
       int i;
       node *first;
```

```
        Queue[++rear] = V1;
        while ( front != rear)
        {
                i = Queue[++front];
                if ( visited[i] == FALSE )
                {
                cout<<endl<<i;
                visited[i] = TRUE;
                }
                first = head[i];
                while ( first != NULL )
                {
                if ( visited[first->vertex] == FALSE )
                Queue[++rear] = first->vertex;
                first = first -> next;
                }
        }

}
/*
-----------------------------------------------------------------
        The main function
-----------------------------------------------------------------
*/
int main ( )
{
  // Local declarations
        int V1;
        char ans;
        Lgraph gr;
        Lgraph();
        gr.create();
        cout<<"Enter the Vertex from which you want to traverse :";
        cin>>V1;
        if ( V1 >= MAX )
                cout<<"\nInvalid Vertex\n";
        else
        {
        cout<<"The Breadth First Search of the Graph is \n";
        gr.bfs(V1);
        return 0;
        }
}
```

Output

```
Enter the vertices no. beginning with 0
Enter the Edge of a graph
```



```
0 1

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 2

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 3

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 4

Want to add more edges?(y/n)y
Enter the Edge of a graph
1 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
2 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
3 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
4 5

Want to add more edges?(y/n)n
Enter the Vertex from which you want to traverse :0
The Breadth First Search of the Graph is
0
1
2
3
4
5
```

Analysis of BFS

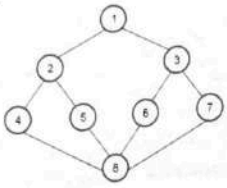
- If the graph is created using **adjacency matrix** in BFS routine then while statement executes for n times. Inside this while statement there is a for loop which executes for all the vertices. Hence time complexity of BFS is $O(V^2)$.

- If the graph is created using **adjacency list** then each vertex V is enqueued and dequeued at most once. Scanning for all the adjacent vertices takes $O(|E|)$ time. Hence the time complexity of BFS is $O(|V|+|E|)$.

Difference between DFS and BFS

DFS	BFS
The nodes are visited in depthwise manner.	The nodes are visited in breadth wise manner
The additional data structure stack is required for this type of traversal.	The additional data structure queue is required for this type of traversal.

Example 5.4.1 *Write an algorithm to perform DFS traversal for a graph. Perform DFS and BFS for the following graph :* **SPPU : May-12, Marks 10**



Solution : DFS = 1 2 4 8 5 6 3 7

BFS = 1 2 3 4 5 6 7 8

Example 5.4.2 *For given graph draw the adjacency list / matrix and perform BFS or DFS* **SPPU : May-14, Marks 6**

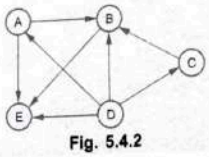
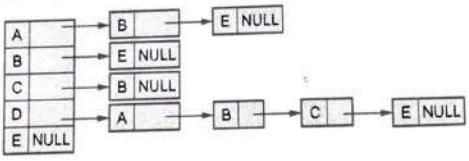


Fig. 5.4.2

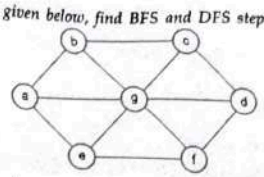
Solution : The adjacency list for given graph is



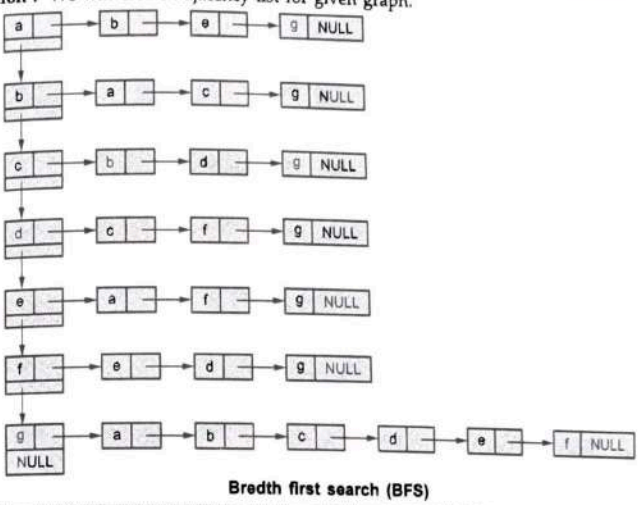
BFS sequence : D, A, B, C, E

DFS sequence : D, A, B, E, C

Example 5.4.3 *For the graph given below, find BFS and DFS stepwise.* **SPPU : Dec.-14, Marks 6**



Solution : We will create adjacency list for given graph.



Breadth first search (BFS)

Action	Queue	Visited array	Output
Step 1 : Insert a in queue. Mark visited (a) = 1	a	a 1 b c d e f	

Step 2 : Delete a and print

a	1
b	
c	
d	
e	
f	

a

Step 3 : Find adjacent of a and insert them in queue. Mark corresponding adjacent nodes as visited.

a	1
b	1
c	
d	
e	1
f	

a

Step 4 : Delete b and print. Find adjacent of b and if those nodes are not visited then insert them in the queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	
e	1
f	

a, b

Step 5 : Delete e and print. Find adjacent of e and if those nodes are not visited then insert them in queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	
e	1
f	1

a, b, e

Step 6 : Delete g and print. Find adjacent of g and if those nodes are not visited then insert them in queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	1
e	1
f	1

a, b, e, g

Step 7 : Delete c and print. Find adjacent of c and if those nodes are not visited then insert them in queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	1
e	1
f	1

a, b, e, g, c

Step 8 : Delete f and print. Find adjacent of f and if those nodes are not visited then insert them in queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	1
e	1
f	1

a, b, e, g, c, f

Step 9 : Delete d and print. Find adjacent of d and if those nodes are not visited then insert them in queue. Mark corresponding nodes as visited.

a	1
b	1
c	1
d	1
e	1
f	1

a, b, e, g, c, f, d

Step 10 : As queue is empty and all entries in the visited array are visited them stop. At display we will get BFS sequence.

Hence BFS sequence is **a, b, e, g, c, f, d**.

Depth first search

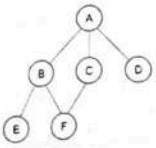
Action	Stack	Visited array	Output
Step 1 : Start with vertex a, mark it visited and print it. Then		a 1, b, c, d, e, f	a
Step 2 : Find adjacent of a i.e. b, and mark it visited. Push it onto the stack. Later b will be popped and printed.	b	a 1, b 1, c, d, e, f	a, b
Step 3 : Find adjacent b. The a is already visited, hence ignore. Next node will be c. Insert c onto the stack mark it as visited. Push it. Later c will be popped and printed.	c	a 1, b 1, c 1, d, e, f	a, b, c

Step 4 : Find adjacent of c. Push d, mark it visited. Later d will be popped and printed.	d	a 1, b 1, c 1, d 1, e, f	a, b, c, d
Step 5 : Find adjacent of d. Push f, mark it as visited. Later f will be popped and printed.	f	a 1, b 1, c 1, d 1, e, f 1	a, b, c, d, f
Step 6 : Find adjacent of f. Push e, mark it as visited. Later e will be popped and printed.	e	a 1, b 1, c 1, d 1, e 1, f 1	a, b, c, d, f, e
Step 7 : Find adjacent of e. Push g, mark it as visited. Later g will be popped and printed.	g	a 1, b 1, c 1, d 1, e 1, f 1	a, b, c, d, f, e, g

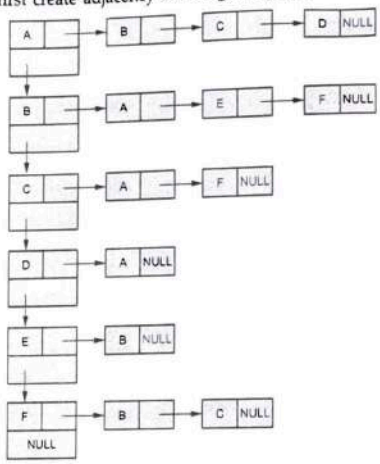
As all nodes are visited and stack is empty we will stop traversing.

The DFS sequence is **a, b, c, d, f, e, g**.

Example 5.4.4 *For a graph given below, Find BFS and DFS* **SPPU : May-16, Marks 4**



Solution : We will first create adjacency list for given graph

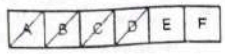


Breadth First Search

Action	Queue	Visited array	Output
Step 1 : Insert A in queue. Mark visited (A) = 1	A	A 1, B, C, D, E, F	

Step 2 : Delete A and print	~~A~~	A 1, B, C, D, E, F	A
Step 3 : Find nodes adjacent of A and insert them in queue. Mark them as visited.	~~A~~ B C D	A 1, B 1, C 1, D 1, E, F	A
Step 4 : Delete B, Print. Find nodes adjacent of B. If those nodes are not visited then mark it as 1, and insert them in queue.	~~A~~ ~~B~~ C D E F	A 1, B 1, C 1, D 1, E 1, F 1	A, B
Step 5 : Delete C print it. Find adjacent of C and if those nodes are not visited then mark it visited and insert in queue.	~~A~~ ~~B~~ ~~C~~ D E F	A 1, B 1, C 1, D 1, E 1, F 1	A, B, C

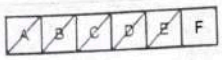
Step 6 : Delete D print it. Find adjacent of D. If not visited then insert in queue.



A	1
B	1
C	1
D	1
E	1
F	1

A, B, C, D

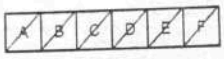
Step 7 : Delete E print it. Find adjacent of E. If not visited then insert in queue.



A	1
B	1
C	1
D	1
E	1
F	1

A, B, C, D, E

Step 8 : Delete F. Print it.



A	1
B	1
C	1
D	1
E	1
F	1

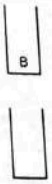
A, B, C, D, E, F

The BFS sequence is **A,B,C,D,E,F :**

Depth First Search

Action	Stack	Visited array	Output
Step 1 : Start with vertex A, mark it visited and print it.		A 1, B, C, D, E, F	A

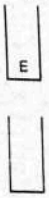
Step 2 : Find adjacent of A, i.e. B mark it visited. Push it onto the stack. Then POP it and print.



A	1
B	1
C	
D	
E	
F	

A, B

Step 3 : Find adjacent of B, it is A. But A is already visited. Hence we will find next adjacent node i.e. E. Push it. Mark it visited. POP it and print.



A	1
B	1
C	
D	
E	1
F	

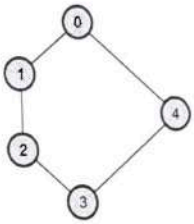
A, B, E

Continuing in this fashion, we traverse and get the DFS sequence as **A, B, E, F, C, D.**

5.5 Depth First Search Traversal

- In depth first search traversal we start from one vertex and traverse the path as deeply as we can go. When there is no vertex further, we traverse back and search for unvisited vertex.
- An array is maintained for storing the visited vertex.
- The DFS will be (if we start from vertex 0) 0 - 1 - 2 - 3 - 4
- The DFS will be (if we start from vertex 3) 3 - 4 - 0 - 1 - 2

For example



1) DFS by Adjacency Matrix (Recursive Program)

C++ Program

```
/*****************************************************************
      Program to create a Graph. The graph is represented using
      Adjacency Matrix. The Program supports both undirected and
      Directed Graphs and traverse the graph in Depth First Search
      Order.
*****************************************************************/

// List of include files

#include <iostream>
#include<ctype.h>
using namespace std;

// List of defined constants

#define        MAX 20
#define        TRUE      1
#define        FALSE     0

// Global declarations

// Declare an adjacency matrix for storing the graph
class Gdfs
{

  private:
  int g[MAX][MAX],v[MAX];
  int v1, v2;
  public:
  int n;
  Gdfs();
  void error(char *);
  void create(),display();
  void Dfs(int);
 ~Gdfs();
};
/*
-----------------------------------------------------------------
The constructor defined
-----------------------------------------------------------------
*/
Gdfs::Gdfs()
{
```



```
 for ( v1 = 0; v1 < MAX; v1++)
      v[v1] = FALSE;
 for ( v1 = 0; v1 < MAX; v1++)
  for ( v2 = 0; v2 < MAX; v2++)
      g[v1][v2] = FALSE;
}
/*
-----------------------------------------------------------------
The destructor defined
-----------------------------------------------------------------
*/
Gdfs::~Gdfs()
{
*/
 for ( v1 = 0; v1 < MAX; v1++)
 {
      for ( v2 = 0; v2 < MAX; v2++)
      g[v1][v2]=FALSE;
 }
 for ( v1 = 0; v1 < MAX; v1++)
  v[v1] = FALSE;
}
/*
-----------------------------------------------------------------
The display function
-----------------------------------------------------------------
*/
void Gdfs::display()
{
 for ( v1 = 0; v1 < n; v1++)
  {
      for ( v2 = 0; v2 < n; v2++)
            cout<<" "<<g[v1][v2];
      cout<<endl;
  }
}


/*
-----------------------------------------------------------------
      The error function
-----------------------------------------------------------------
*/
void Gdfs::error( char *Msg )
{
```

```
    cout<<Msg;
    cout<<"Press any key to abort\n";
    getch();
    exit(1);
}

/*
-----------------------------------------------------------------
    The Create function
-----------------------------------------------------------------
*/
void  Gdfs::create()
{
    int Ch, v1, v2, Undirected;
    Undirected = TRUE;
    n=0;
    cout<<"\n This Program Is For Creation OF Graph";
    cout<<"\n And The Display Is By Depth First Search";
    cout<<"\n\tPress d/D for directed nad u/U for undirected garph\n";
    Ch = toupper(getche());
    if ( Ch == 'D')
      Undirected = FALSE;
    do
    {
        cout<<"\nEnter the Edge of a graph by two vertices \n";
        cout<<"(and type -99 terminate)\n";
        cin>>v1>>v2;
        if ( v1 == -99)
          break;
        if ( v1 >= MAX || v2 >= MAX)
            error("Invalid Vertex Value\n" );
        else
          g[v1][v2] = TRUE;
          if ( Undirected )
              g[v2][v1] = TRUE;
    n++;
    }
    while(1);
}
/*
-----------------------------------------------------------------
    The Dfs function
-----------------------------------------------------------------
*/
void Gdfs::Dfs(int v1)
{
    int v2;
```



```
    cout<<endl<<v1;
    v[v1] = TRUE;
    for ( v2 = 0; v2 < n; v2++)
        if ( g[v1][v2] == TRUE &&  v[v2] == FALSE )
            Dfs(v2);
}
/*
-----------------------------------------------------------------
    The main function
-----------------------------------------------------------------
*/
int main ( )
{
    Gdfs gr;
    int v1;
    gr.create();
    cout<<"The Adjacency Matrix for the graph is "<<endl;
    gr.display();
    cout<<"Enter the Vertex from which you want to traverse : ";
    cin>>v1;
    if ( v1 >= MAX )
        gr.error("Invalid Vertex\n");
    cout<<"The Depth First Search of the Graph is "<<endl;
    gr.Dfs(v1);
    return 0;
}
```

Output

```
This Program Is For Creation OF Graph
And The Display Is By Depth First Search
    Press d/D for directed nad u/U for undirected garph
u
Enter the Edge of a graph by two vertices
(and type -99 terminate)
0 1

Enter the Edge of a graph by two vertices
(and type -99 terminate)
0 2

Enter the Edge of a graph by two vertices
(and type -99 terminate)
1 3

Enter the Edge of a graph by two vertices
(and type -99 terminate)
2 3
```

```
Enter the Edge of a graph by two vertices
(and type -99 terminate)
-99 -99
The Adjacency Matrix for the graph is
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter the Vertex from which you want to traverse : 1
The Depth First Search of the Graph is
1
0
2
3
```

Explanation of Logic For Depth First Traversal

In DFS the basic data structure for storing the adjacent nodes is stack. In our program we have used a recursive call to DFS function. When a recursive call is invoked actually push operation gets performed. When we exit from the loop pop operation will be performed. Let us see how our program works.

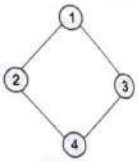
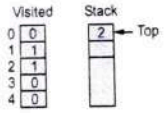


Fig. 5.5.1

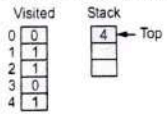
Step 1 : Start with vertex 1, print it so '1' gets printed. Mark 1 as visited.

Step 2 : Find adjacent vertex to 1, say i.e. 2 if it is not visited, call DFS(2) i.e. 2 will get inserted in the stack, mark is as visited.



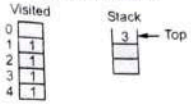
After exiting the loop, value 2 will be popped print '2'

Step 3 : Find adjacent to '2' i.e. vertex 4 if it is not visited call DFS(4) i.e., 4 will get pushed on to the stack mark it as visited.



After exiting the loop, value 4 will be popped print '4'

Step 4 : Find adjacent to '4' i.e. vertex 3 if it is not visited call DFS(3) i.e. 3 will be pushed onto the stack mark it visited.



After exiting the loop, value 3 will be popped print '3'

Since all the nodes are covered stop the procedure.

So output of DFS is

1 2 4 3

2) DFS by Adjacency Matrix (Non-recursive Program)

For non-recursive implementation of DFS, we use stack data structure. In this implementation we follow following steps

i) Push the vertex from which the searching should be started.

ii) POP the stack, print popped vertex if it is not visited. Then mark it as visited.

iii) Then find adjacent vertices to the current vertex and push them onto the stack.

iv) Repeat step (ii) and (iii) if stack is not empty.

```
/*****************************************************************
    Program to create a Graph. The graph is represented using
    Adjacency Matrix. The Program supports both undirected and
    Directed Graphs and traverse the graph in
    non-recursive Depth First Search Order.
*****************************************************************/

// List of include files

#include <iostream>
#include<ctype.h>
using namespace std;

// List of defined constants

#define    MAX 20
#define    TRUE 1
#define    FALSE 0

// Global declarations

int st[10];
```

```
int top;
// Declare an adjacency matrix for storing the graph
class Gdfs
{

  private:
  int g[MAX][MAX],v[MAX];
  int v1,v2;
  public:
  int n;
  Gdfs();
  void error(char *);
  void create(),display();
  void Dfs(int);
  friend void push(int item);
  friend int pop();
  ~Gdfs();
};
/*
-----------------------------------------------------------
The constructor defined
-----------------------------------------------------------
*/
Gdfs::Gdfs()
{
  for ( v1 = 0; v1 < MAX; v1++)
        v[v1] = FALSE;
  for ( v1 = 0; v1 < MAX; v1++)
   for ( v2 = 0; v2 < MAX; v2++)
        g[v1][v2] = FALSE;
  for(v1=0;v1<MAX;v1++)
  st[v1]=0;
  top=-1;
}
/*
-----------------------------------------------------------
The destructor defined
-----------------------------------------------------------
*/
Gdfs::~Gdfs()
{
  for ( v1 = 0; v1 < MAX; v1++)
  {
       for ( v2 = 0; v2 < MAX; v2++)
       g[v1][v2]=FALSE;
  }
  for ( v1 = 0; v1 < MAX; v1++)
```



```
  v[v1] = FALSE;
}
/*
-----------------------------------------------------------
The display function
-----------------------------------------------------------
*/
void Gdfs::display()
{
 for ( v1 = 0; v1 < n; v1++)
  {
       for ( v2 = 0; v2 < n; v2++)
              cout<<" "<<g[v1][v2];
       cout<<endl;
  }
}


/*
-----------------------------------------------------------
       The error function
-----------------------------------------------------------
*/

void Gdfs::error( char *Msg )
{
       cout<<Msg;
       cout<<"Press any key to abort\n";
       getch();
       exit(1);
}

/*
-----------------------------------------------------------
       The Create function
-----------------------------------------------------------
*/
void  Gdfs::create()
{
       int Ch, v1, v2, Undirected;
       Undirected = TRUE;
       n=0;
       cout<<"\n This Program Is For Creation OF Graph";
       cout<<"\n And The Display Is By Depth First Search";
       cout<<"\n\tPress d/D for directed nad u/U for undirected garph\n";
       Ch = toupper(getche());
       if ( Ch == 'D')
```

```
    Undirected = FALSE;
  do
  {
        cout<<"\nEnter the Edge of a graph by two vertices \n";
        cout<<"(and type -99 terminate)\n";
        cin>>v1>>v2;
        if ( v1 == -99)
          break;
        if ( v1 >= MAX || v2 >= MAX)
              error("Invalid Vertex Value\n" );
        else
          g[v1][v2] = TRUE;
          if ( Undirected )
                g[v2][v1] = TRUE;
  n++;
  }
  while(1);
}

/*
-----------------------------------------------------------------
      The Dfs function
-----------------------------------------------------------------
*/
void Gdfs::Dfs(int v1)
{
      int v2;
      push(v1);
      while(top!=-1)
      {
      v1=pop();
      if(v[v1]==FALSE)
      {
       cout<<endl<<v1;
       v[v1] = TRUE;
      }
      for ( v2 = 0; v2 < n; v2++)
            if ( g[v1][v2] == TRUE &&  v[v2] == FALSE )
                  push(v2);
      }
}
void push(int item)
{
 st[++top]=item;
}
int pop()
{
```



```
int item;
item=st[top];
top--;
return item;
}
/*
-----------------------------------------------------------------
      The main function
-----------------------------------------------------------------
*/
int main ( )
{
      Gdfs gr;
      int v1;
      gr.create();
      cout<<"The Adjacency Matrix for the graph is "<<endl;
      gr.display();
      cout<<"Enter the Vertex from which you want to traverse : ";
      cin>>v1;
      if ( v1 >= MAX )
            gr.error("Invalid Vertex\n");
      cout<<"The Depth First Search of the Graph is "<<endl;
      gr.Dfs(v1);
      return 0;
}
```

Output

```
This Program Is For Creation OF Graph
And The Display Is By Depth First Search
      Press d/D for directed nad u/U for undirected garph
u
Enter the Edge of a graph by two vertices
(and type -99 terminate)
0 1

Enter the Edge of a graph by two vertices
(and type -99 terminate)
0 2

Enter the Edge of a graph by two vertices
(and type -99 terminate)
1 3

Enter the Edge of a graph by two vertices
(and type -99 terminate)
2 3

Enter the Edge of a graph by two vertices
```

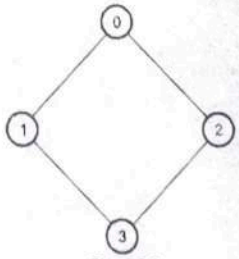


Fig. 5.5.2

```
(and type -99 terminate)
-99 -99
The Adjacency Matrix for the graph is
0110
1001
1001
0110
Enter the Vertex from which you want to traverse : 2
The Depth First Search of the Graph is
2
3
1
0
```

3) DFS by Adjacency List (Recursive Program)

C++ Program

```
/*****************************************************************
    Program to create a Graph. The graph is represented using
    Adjacency List. The Program supports both undirected and
    Directed Graphs and traverse the graph in Depth First Search
    Order.
*****************************************************************/

// List of include files

#include <iostream>


#define    MAX 10
#define    TRUE  1
#define    FALSE  0
using namespace std;
// Declaring an adjacency list for storing the graph
class Lgraph
{
  private:
      typedef struct node1
         {
          int vertex;
          struct node1 *next;
         }node;
     node  *head[MAX];  //Array Of head nodes
     int  visited[MAX];
     //visited array for checking whether the array is visited or not
 public:
    Lgraph();
```



```
    void create(),Dfs(int);
};
/*
-----------------------------------------------------------------
The constructor defined
-----------------------------------------------------------------
*/
Lgraph::Lgraph()
{
 int V1;
 for ( V1 = 0; V1 < MAX; V1++)
   visited[V1] = FALSE;
 for ( V1 = 0; V1 < MAX; V1++)
   head[V1] = NULL;
}
/*
-----------------------------------------------------------------
The create function
-----------------------------------------------------------------
*/
void Lgraph::create()
{
    int V1, V2;
    char ans='y';
    node *New,*first;
    cout<<"\n\nEnter the vertices no. beginning with 0";
    do
    {
      cout<<"\nEnter the Edge of a graph \n";
         cin>>V1>>V2;
         if ( V1 >= MAX || V2 >= MAX)
              cout<<"Invalid Vertex Value\n" ;
         else
           {
              // creating link from V1 to V2
              New = new node;
              if ( New == NULL )
                cout<<"Insufficient Memory\n";
              New -> vertex = V2;
              New -> next = NULL;
              first = head[V1];
              if ( first == NULL )
                head[V1] = New;
              else
               {
                  while ( first -> next != NULL )
                    first = first-> next;
```

```
                first -> next = New;
            }
        // creating link from V2 to V1
            New = new node;
            if ( New == NULL )
              cout<<"Insufficient Memory\n";
            New -> vertex = V1;
            New -> next = NULL;
            first = head[V2];
            if ( first == NULL )
              head[V2] = New;
            else
             {
               while ( first -> next != NULL )
                     first = first-> next;
              first -> next = New;
             }
          }
    cout<<"\nWant to add more edges?(y/n)";
    cin>>ans;
    }while(ans=='y');
}
/*
-----------------------------------------------------------------
The Dfs function
-----------------------------------------------------------------
*/
void Lgraph::Dfs(int V1)
{
    int V2;
    node *first;
    cout<<endl<<V1;
    visited[V1] = TRUE;
    first = head[V1];
    while ( first != NULL )
        if ( visited[first->vertex] == FALSE )
              Dfs(first->vertex);
        else
              first = first -> next;
}
/*
-----------------------------------------------------------------
    The main function
-----------------------------------------------------------------
```



```
*/
int main ( )
{
    // Local declarations
    int V1, V2;
    Lgraph gr;
    gr.create();;
    cout<<endl<<"Enter the Vertex from which you want to traverse :";
    cin>>V1;
    if ( V1 >= MAX )
          cout<<"Invalid Vertex\n";
    else
    {
     cout<<"The Depth First Search of the Graph is \n";
     gr.Dfs(V1);
     return 0;
    }
}
```

Output

```
Enter the vertices no. beginning with 0
Enter the Edge of a graph
0 1

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 2

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 3

Want to add more edges?(y/n)y
Enter the Edge of a graph
0 4

Want to add more edges?(y/n)y
Enter the Edge of a graph
1 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
2 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
3 5
```

```
Want to add more edges?(y/n)y
Enter the Edge of a graph
4 5

Want to add more edges?(y/n)n
Enter the Vertex from which you want to traverse :0
The Depth First Search of the Graph is
0e
1
5
2
3
4
```

4) DFS by Adjacency List (Non-recursive Program)

```
/*****************************************************************
      Program to create a Graph. The graph is represented using
      Adjacency List. The Program supports both undirected and
      Directed Graphs and traverse the graph in non recursive
      Depth First Search Order.
*****************************************************************/

// List of include files

#include <iostream>


#define     MAX 10
#define     TRUE   1
#define     FALSE  0
using namespace std;
// Global declarations
int st[10];
int top;

// Declaring an adjacency list for storing the graph
class Lgraph
{
  private:
        typedef struct node1
            {
             int vertex;
             struct node1 *next;
            }node;
      node  *head[MAX];  //Array Of head nodes
      int  visited[MAX];
      //visited array for checking whether the array is visited or not
```



```
  public:
        Lgraph();
        void create(),Dfs(int);
        friend void push(int item);
        friend int pop();

};
/*
-----------------------------------------------------------------
The constructor defined
-----------------------------------------------------------------
*/
Lgraph::Lgraph()
{
 int V1;
 for ( V1 = 0; V1 < MAX; V1++)
   visited[V1] = FALSE;
 for ( V1 = 0; V1 < MAX; V1++)
   head[V1] = NULL;
 for ( V1 = 0; V1 < MAX; V1++)
   st[V1] = 0;
 top=-1;
}
/*
-----------------------------------------------------------------
The create function
-----------------------------------------------------------------
*/
void Lgraph::create()
{
      int V1, V2;
      char ans='y';
      node *New,*first;
      do
      {
        cout<<"\nEnter the Edge of a graph \n";
            cin>>V1>>V2;
            if ( V1 >= MAX || V2 >= MAX)
                  cout<<"Invalid Vertex Value\n" ;
            else
             {
                  // creating link from V1 to V2
                  New = new node;
                  if ( New == NULL )
                    cout<<"Insufficient Memory\n";
                  New -> vertex = V2;
                  New -> next = NULL;
                  first = head[V1];
```

```
            if ( first == NULL )
              head[V1] = New;
            else
            {
                  while ( first -> next != NULL )
                    first = first -> next;
                  first -> next = New;
            }
        // creating link from V2 to V1
            New = new node;
            if ( New == NULL )
              cout<<"Insufficient Memory\n";
            New -> vertex = V1;
            New -> next = NULL;
            first = head[V2];
            if ( first == NULL )
              head[V2] = New;
            else
            {
              while ( first -> next != NULL )
                    first = first -> next;
              first -> next = New;
            }
        }
    cout<<"\nWant to add more edges?(y/n)";
    cin>>ans;
    }while(ans=='y');
}
/*
-----------------------------------------------------------
The Dfs function
-----------------------------------------------------------
*/
void Lgraph::Dfs(int V1)
{
    node *first;

    push(V1);
    while(top!=-1)
    {
    V1=pop();
    if(visited[V1]==FALSE)
    {
    cout<<endl<<V1;
    visited[V1] = TRUE;
    }
    first = head[V1];
```



```
    while ( first != NULL )
    {
      if ( visited[first->vertex] == FALSE )
        push(first ->vertex);
      first = first -> next;
    }
  }
}
void push(int item)
{
st[++top]=item;
}
int pop()
{
int item;
item=st[top];
top--;
return item;
}

/*
-----------------------------------------------------------
      The main function
-----------------------------------------------------------
*/
int main ( )
{
    // Local declarations
    int V1, V2;

    Lgraph gr;
    gr.create();

    cout<<endl<<"Enter the Vertex from which you want to traverse :";
    cin>>V1;
    if ( V1 >= MAX )
          cout<<"Invalid Vertex\n";
    else
    {
      cout<<"The Depth First Search of the Graph is \n";
      gr.Dfs(V1);
      return 0 ;
    }
}
```

Output

```
Enter the Edge of a graph
1 2

Want to add more edges?(y/n)y
Enter the Edge of a graph
```

```
1 3

Want to add more edges?(y/n)y
Enter the Edge of a graph
1 4

Want to add more edges?(y/n)y
Enter the Edge of a graph
2 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
3 5

Want to add more edges?(y/n)y
Enter the Edge of a graph
4 5

Want to add more edges?(y/n)n

Enter the Vertex from which you want to traverse :3
The Depth First Search of the Graph is

3
5
4
1
2
```

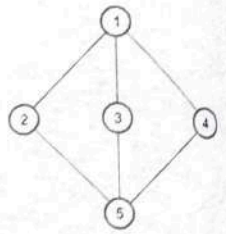


Fig. 5.5.3

5.6 Applications of Graph

The graph theory is used in the computer science very widely. There are many interesting applications of graph. We will list out few applications -

1. In computer networking such as Local Area Network (LAN), Wide Area Networking (WAN), internetworking.
2. In telephone cabling graph theory is effectively used.
3. In job scheduling algorithms.

5.7 Spanning Tree

SPPU : May-10,11,13,14,15,16, Dec.-14, Marks 8

A Spanning tree S is a subset of a tree T in which all the vertices of tree T are present but it may not contain all the edges.

The minimum spanning tree of a weighted connected graph G is called **minimum** spanning tree if its **weight is minimum**.

We are going to discuss the two popular algorithms of minimum spanning tree, namely Prim's algorithm and Kruskal's algorithm.

5.7.1 Prim's Algorithm

In Prim's algorithm the pair with the **minimum weight** is to be chosen. Then **adjacent** to these vertices which ever is the edge having minimum weight is selected. This process will be continued till all the vertices are not be covered. The necessary condition in this case is that the **circuit should not be formed**.

ADT Operations for Prim's Algorithm

AbstractDataType Prim

{

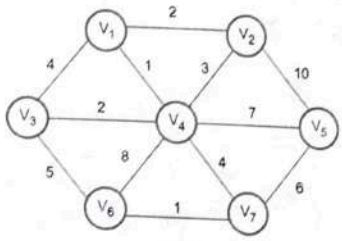
Instances : Prim's algorithm is applied on a weighted graph G for creating a minimum spanning tree, and path length of a spanning tree can be obtained.

Operations :

1. Select an edge with minimum weight.
2. Add the vertices in the set V.
3. Then select any edge with minimum weight such that the edge is adjacent to any vertex from set V.
4. Repeat step 1, 2 and 3 until all the vertices are selected. But circuit should be formed while selecting these vertices.

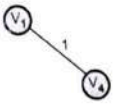
}

Example 5.7.1 *For the graph given below, construct a minimum spanning tree using Prim's algorithm. Show the table created during each pass of the algorithm.*



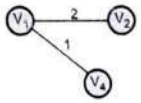
Solution : In Prim's algorithm the minimum path length is selected. Only the circuit should not be formed.

Step 1 :



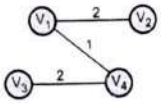
Path length = 1

Step 2 :



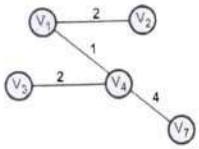
Path length = 3

Step 3 :



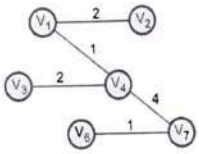
Path length = 5

Step 4 :



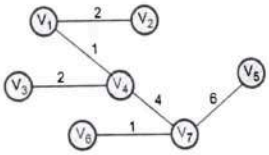
Path length = 9

Step 5 :



Path length = 10

Step 6 :



Path length = 16

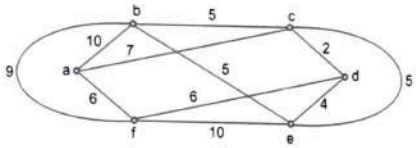
Thus the minimum spanning tree with path length 16 is obtained.

Example 5.7.2 *Explain Prim's algorithm with an example.*

Solution : Prim's algorithm is to obtain the minimum spanning tree. In this algorithm the minimum edge is selected each time. The edge selection is done for adjacent vertices only.

```
Prim(G,Cost, n, tree)
{
 /* G is a graph of n nodes. cost is an array of cost of edges. tree is an array of
minimum spanning tree */
min=cost[p,q] //represents minimum cost edge
tree[1][1]=p;
tree[1][2]=q;
for(i=1; i<n; i++)
{
        if(cost[i][q] < cost[i][p])
                optimum[i]=p;
        else
        optimum[i]=q // storing minimum cost
        optimum[p]=optimum[q]=0;
        for(i=2; i<n-1; i++)
{//store vertex j at optimum distance and neighbouring of j in 'tree'
                tree[i][1] = j;
                tree[i][2] = optimum[j];
                min = min+cost[j][optimum[j]];
        optimum[j]=0;
        for(k=1 to n)
        if((optimum[k]!=0)AND (cost[k,optimum[k]]>cost[k,j]))then
                optimum[k]=j;
        }
                return min;
}
```

For example : Consider the graph as given below -



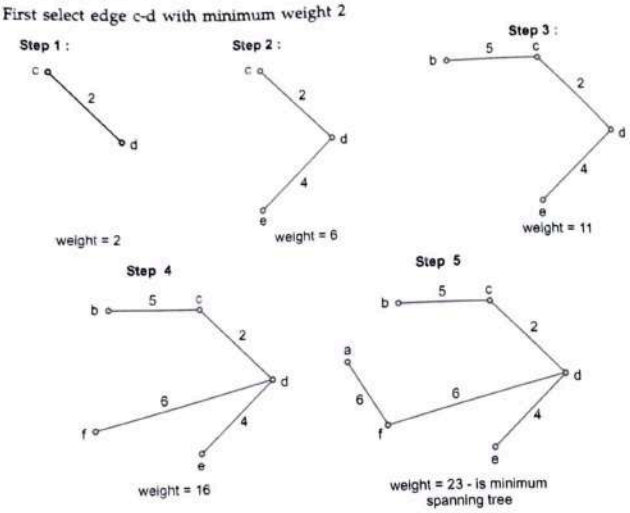
We will find the minimum spanning tree using Prim's algorithm using following rules -

1. Select the edge with minimum edge. The corresponding vertices will form the set of vertices.

2. Each time select an edge with minimum weight and the edge to be selected from the graph should be such that at least one of the vertex of the selected edge is adjacent to already selected vertices.
3. In selection of edges the circuit should not be formed.

From above graph,

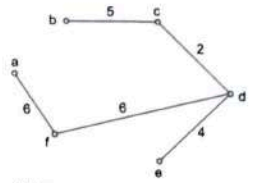
First select edge c-d with minimum weight 2



The edge selection can be summarised as below -

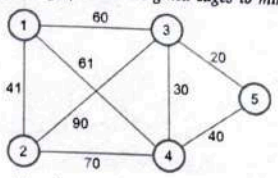
Edge	Set
<c-d>	{c, d}
<d-e>	{c, d, e}
<b-c>	{b, c, d, e}
<d-f>	{b, c, d, e, f}
<a-f>	{a, b, c, d, e, f}

Thus the minimum spanning tree will be -



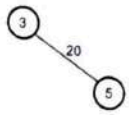
Minimum spanning tree with weight 23

Example. 5.7.3 *Convert the given graph with weighted edges to minimal spanning tree.*

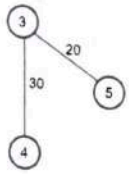


Solution : We will obtain minimal spanning tree using Prim's algorithm.

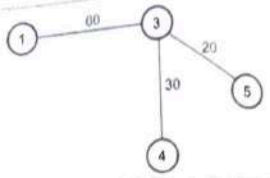
Step 1 :
Select an edge 3-5, as it is the edge with minimum weight.



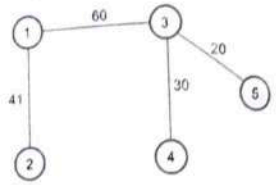
Step 2 :
Select next adjacent edge 3-4, as it is the edge with minimum weight.



Step 3 :
Select next adjacent edge 3-1



Step 4 :
Select next adjacent edge 1-2



As all the vertices are visited and we get a connected graph, this is a minimal spanning tree with **total weight = 151.**

C++ Program

```
/*****************************************************************
This program is to implement Prim's Algorithm
*****************************************************************/
# include<iostream>
# define SIZE 20
# define INFINITY 32767
using namespace std;
class MST
{
 private:
  int G[SIZE][SIZE],nodes;
 public:
 MST();
 void Prim();
 void get_data();
};
/*     This function finds the minimal spanning tree by Prim's Algorithm   */

MST::MST()
{
 for (int i=0 ; i<SIZE ; i++)       // Initialize the graph
   for (int j=0 ; j<SIZE ; j++)
     G[i][j] = 0;
}
```



```
void MST::Prim()
{
      int select[SIZE], i, j, k;
      int min_dist, v1, v2,total=0;

      for (i=0 ; i<nodes ; i++)   // Initialize the selected vertices list
            select[i] = 0;

      cout<<"\n\n The Minimal Spanning Tree Is :\n";
            select[0] = 1;
      for (k=1 ; k<nodes ; k++)
      {
                  min_dist = INFINITY;
                  for (i=0 ; i<nodes ; i++) // Select an edge such that one vertex is
                  {                         // selected and other is not and the edge
      for (j=0 ; j<nodes ; j++) // has the least weight.
      {
            if (G[i][j] && ((select[i] && !select[j]) || (!select[i] && select[j])))
                  {
                        if (G[i][j] < min_dist)//obtained edge with minimum wt
                        {
                              min_dist = G[i][j];
                              v1 = i;
                              v2 = j;   //picking up those vertices
                        }
                  }
            }
      }
      cout<<"\n Edge ("<<v1<<" "<<v2<<") and weight = "<<min_dist;
      select[v1] = select[v2] = 1;
      total =total+min_dist;
      }
      cout<<"\n\n\t Total Path Length Is =  "<<total;
}
void MST::get_data()
{
 int v1, v2, length, n;
 cout<<"\n Enter Number of Nodes in The Graph  ";
 cin>>nodes;
 cout<<"\n Enter Number of Edges in The Graph  ";
 cin>>n;
 //entering weighted graph
 cout<<"\n Enter edges and weights \n";
 for (int i=0 ; i<n; i++)
 {
  cout<<"\n Enter Edge by V1 and V2 : ";
```

```
  cin>>v1>>v2;
  cout<<"\n Enter corresponding weight :";
  cin>>length;
  G[v1][v2] = G[v2][v1] = length;
 }
}
int main( )
{

      MST obj;
       cout<<"\n\t Prim'S Algorithm\n";
         obj.get_data();
       cout<<"\n\t";
       obj.Prim();
      return 0;
}
```

Output

```
        Prim'S Algorithm

Enter Number of Nodes in The Graph  7

Enter Number of Edges in The Graph  9

Enter edges and weights

Enter Edge by V1 and V2 :0 1

Enter corresponding weight :27

Enter Edge by V1 and V2 :1 2

Enter corresponding weight :16

Enter Edge by V1 and V2 :2 3

Enter corresponding weight :12

Enter Edge by V1 and V2 :3 4

Enter corresponding weight :22

Enter Edge by V1 and V2 :4 5

Enter corresponding weight :25

Enter Edge by V1 and V2 :0 5
```



```
Enter corresponding weight :6

Enter Edge by V1 and V2 :1 6

Enter corresponding weight :14

Enter Edge by V1 and V2 :4 6

Enter corresponding weight :24

Enter Edge by V1 and V2 :3 6

Enter corresponding weight :18


The Minimal Spanning Tree Is :

Edge (0 5 )and weight = 6
Edge (4 5 )and weight = 25
Edge (3 4 )and weight = 22
Edge (2 3 )and weight = 12
Edge (1 2 )and weight = 16
Edge (1 6 )and weight = 14

Total Path Length Is = 95
```

5.7.2 Kruskal's Algorithm

In Kruskal's algorithm the **minimum** weight is obtained. In this algorithm also the **circuit should not be formed**. Each time the edge of minimum weight has to be selected, from the graph.It is **not necessary** in this algorithm to have edges of minimum weights to be **adjacent**. Let us solve one example by Kruskal's algorithm.

Example 5.7.4 *Consider the following graph (Fig. 5.7.1)*

i) Obtain minimum spanning tree by Kruskal's algorithm.

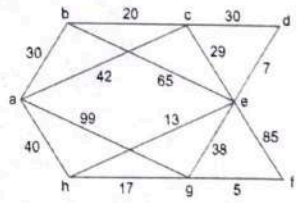
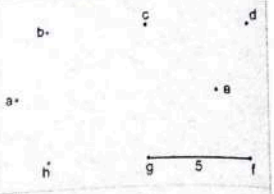


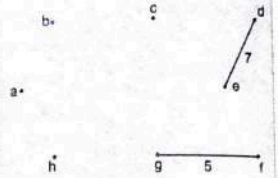
Fig. 5.7.1

Solution : Kruskal's algorithm

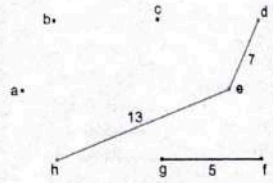
Step 1 : Select an edge gf with weight 5.



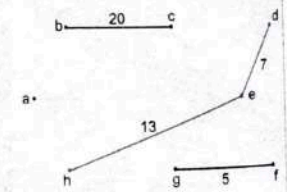
Step 2 : Select an edge ed with weight 7.



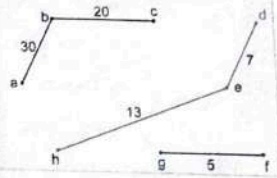
Step 3 : Select an edge eh with weight 13.



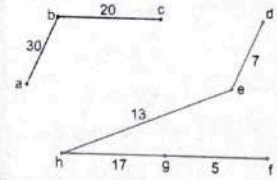
Step 4 : Select an edge bc with weight 20.



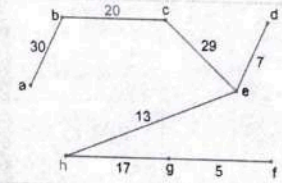
Step 5 : Select an edge ab with weight 30.



Setp 6 : Select an edge hg with weight 17.



Step 7 : Select an edge ce with weight 29.



Total weight = 134

Algorithm

```
Algorithm spanning_tree()
//Problem Description : This algorithm finds the minimum
//spanning tree using Kruskal's algorithm
//Input : The adjacency matrix graph G containing cost
//Output : prints the spanning tree with the total cost of
//spanning tree
 count←0
 k←0
 sum←0
      for i←0 to tot_nodes do
            parent[i]←i
while(count!=tot_nodes-1)do
 {
      pos←Minimum(tot_edges);//finding the minimum cost edge
            If(pos=-1)then//Perhaps no node in the graph
```

```
                break
            v1←G[pos].v1
            v2←G[pos].v2
            i←Find(v1,parent)
            j←Find(v2,parent)
            if(i!=j)then
            {
             tree[k][0] ←v1
//storing the minimum edge in array tree[]
             tree[k][1] ←v2
             k++
             count++;
             sum+←G[pos].cost
//accumulating the total cost of MST
            Union(i,j,parent);
            }
            G[pos].cost INFINITY
           }
           if(count=tot_nodes-1)then
           {
                for i←0 to tot_nodes-1
            {
                write(tree[i][0],tree[i][1])
            }
            write("Cost of Spanning Tree is ",sum)
           }
```

tree [ ] [ ] is an array in which the spanning tree edges are stored.

Computing total cost of all the minimum distances.

For each node of i, the minimum distance edges are collected in array **tree [ ] [ ]**. The spanning tree is printed here.

C function

```
void spanning_tree()
{
 int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
 int sum;
 int Find(int v2,int parent[]);
 void Union(int i,int j,int parent[]);
 count=0;
 k=0;
 sum=0;
 for(i=0 ; i<tot_nodes ; i++)
            parent[i]=i ;
 while(count!=tot_nodes-1)
 {
            pos = Minimum(tot_edges);//finding the minimum cost edge
            if(pos==-1)//Perhaps no node in the graph
                break;
            v1 = G[pos].v1;
            v2 = G[pos].v2;
```



```
            i = Find(v1,parent);
            j=Find(v2,parent);
            if(i != j)
            {
                tree[k][0] = v1;//storing the minimum edge in array
                                                tree[]
                tree[k][1] = v2;
                k++;
                count++;
                sum+=G[pos].cost;//accumulating the total cost of MST
                Union(i,j,parent);
            }
            G[pos].cost = INFINITY;
           }
           if(count==tot_nodes-1)
           {
            printf("\n Spanning tree is...");
            printf("\n-------------------------\n");
            for(i=0 ; i<tot_nodes-1 ; i++)
            {
                printf("[%d",tree[i][0]);
                printf(" - ");
                printf("%d",tree[i][1]);
                printf("]");
            }
            printf("\n-------------------------");
            printf("\nCost of Spanning Tree is = %d",sum);
           }
           else
            {
                printf("There is no Spanning Tree");
            }
}
```

Analysis

The efficiency of Kruskal's algorithm is $\Theta(|E| \log |E|)$ where E is total number of edges in the graph.

Example 5.7.5 *Consider the following diagram. Using Prim's algorithm and Kruskal's algorithm draw all the intermediate stages of the graph to find minimum spanning tree.*

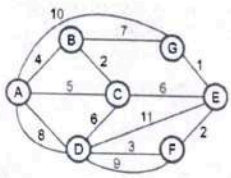
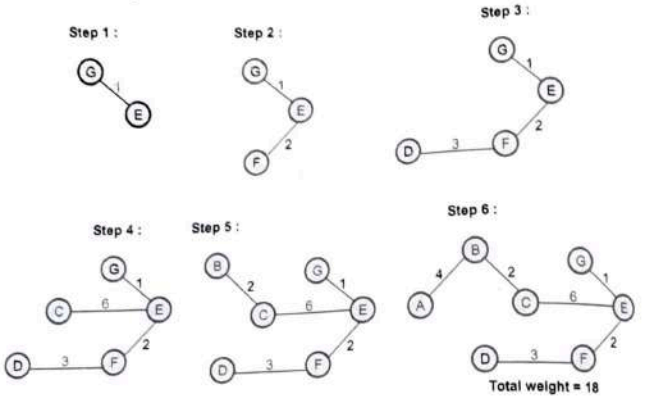
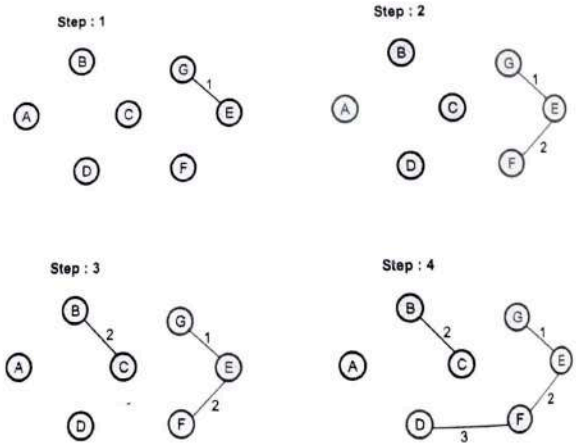


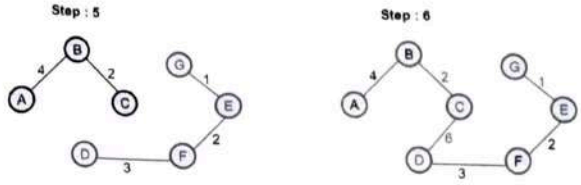
Fig. 5.7.2

Solution : Prim's Algorithm



Kruskal's Algorithm

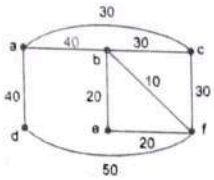




Total weight = 18

Example 5.7.6 *Consider the following graph G. The label of an edge is the cost of the edge. i) Give the representation of the graph G in a computer. What is the space complexity ?*

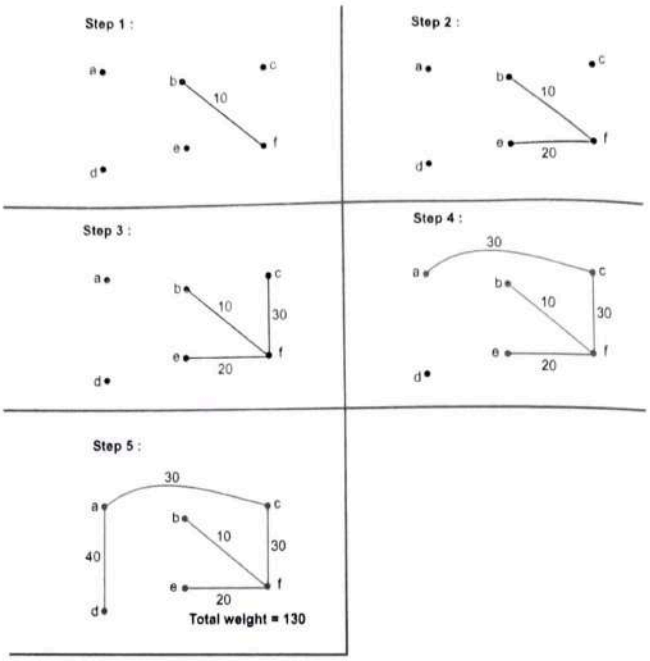
ii) Using Kruskals algorithm, draw a Minimum Spanning Tree (MST) of the graph G describing each step of the algorithm. Also write down the order in which the edges are selected.



Solution : **i)** The graph can be represented using adjacency matrix.

	a	b	c	d	e	f
a	0	40	30	40	0	0
b	40	0	30	0	20	10
c	30	30	0	0	0	30
d	40	0	0	0	0	50
e	0	20	0	0	0	20
f	0	10	30	50	20	0

ii) Kruskal's Algorithm



Example 5.7.7 *Write a pseudo code for Prim's algorithm and find the MST for the graph given and show all the steps.* **SPPU : May-14, Marks 8**

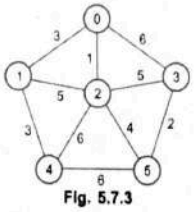
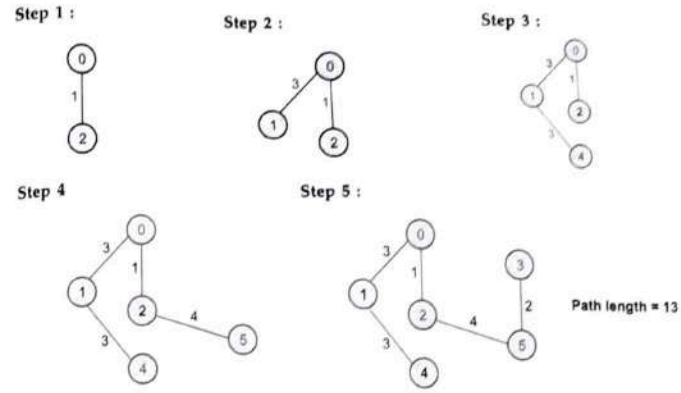


Fig. 5.7.3

Ans. : Refer section 5.7.1



Example 5.7.8 *Find the minimum spanning tree using Prim's and Kruskal's method for the following graph :* **SPPU : Dec.-14, Marks 8**

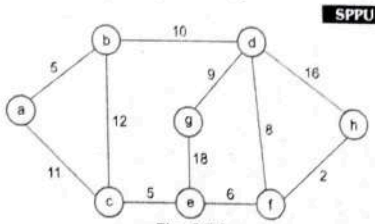
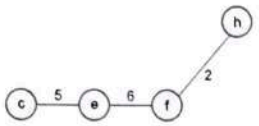


Fig. 5.7.4

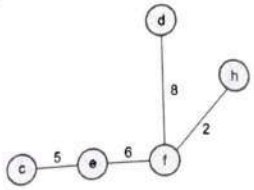
Solution : Prim's algorithm

Step 1 : **Step 2 :**

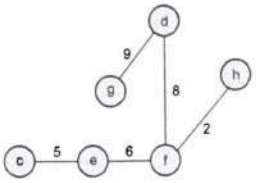
Step 3 :



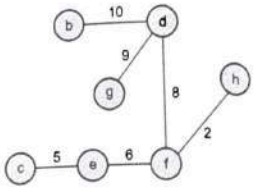
Step 4 :



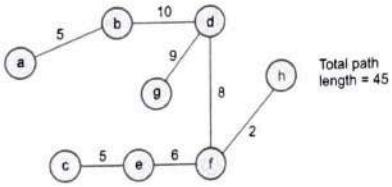
Step 5 :



Step 6 :

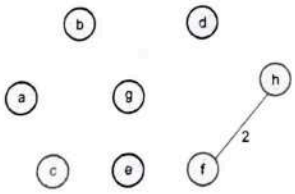


Step 7 :

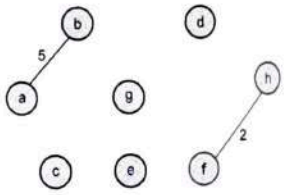


Kruskal's algorithm

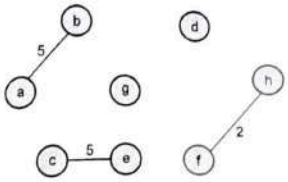
Step 1 :



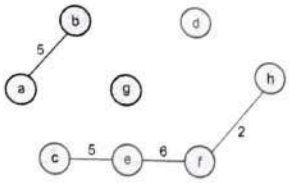
Step 2 :



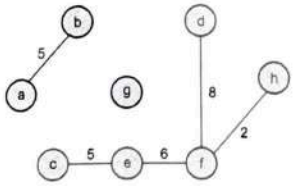
Step 3 :



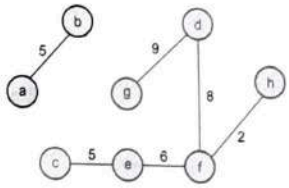
Step 4 :



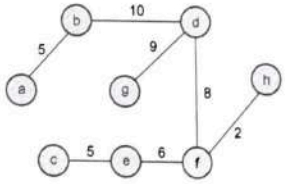
Step 5 :



Step 6 :



Step 7 :



Example 5.7.9 *Find the minimum spanning tree using Prim's and Kruskal's method for the following graph :*

SPPU : May-15, Marks 4

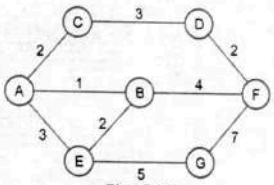
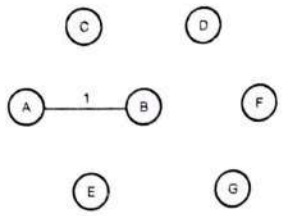


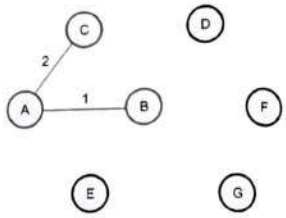
Fig. 5.7.5

Ans. :

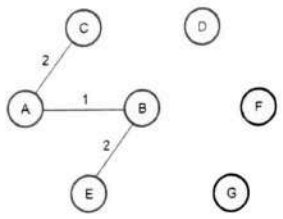
Step 1 :



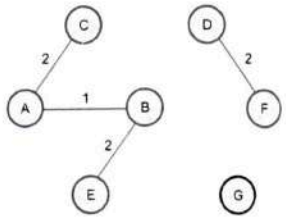
Step 2 :



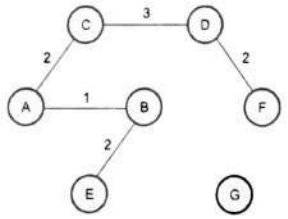
Step 3 :



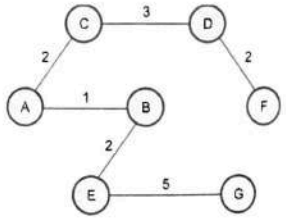
Step 4 :



Step 5 :



Step 6 :



Total path length = 15

Example 5.7.10 *For the graph given below, show stepwise representation of MST using Kruskal's algorithm.* **SPPU : May-16, Marks 4**

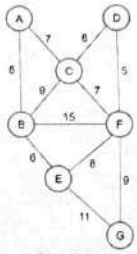
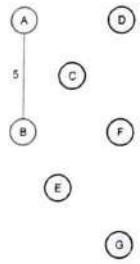


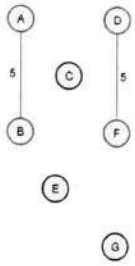
Fig. 5.7.6

Solution :

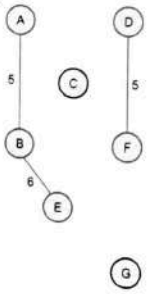
Step 1 :



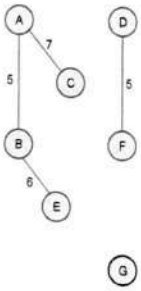
Step 2 :



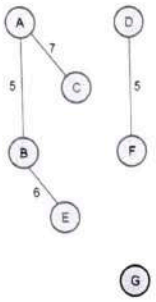
Step 3 :



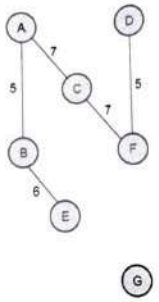
Step 4 :



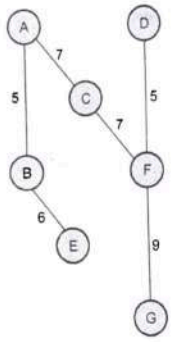
Step 5 :



Step 6 :



Step 7 :



Total path length = 39

C++ Program

```
#include<iostream>
#define INFINITY 999
using namespace std ;
class KRUSKAL
{
 private:
      typedef struct Graph
      {
       int v1;
       int v2;
       int cost;
```



```
      }GR;
      GR G[20];
public:
      int tot_edges,tot_nodes;
      void create();
      void spanning_tree();
      void get_input();
      int Minimum(int);
};
int Find(int v2,int parent[])
{
 while(parent[v2]!=v2)
 {
      v2=parent[v2];
 }
 return v2;
}
void Union(int i,int j,int parent[])
{
 if(i<j)
      parent[j]=i;
 else
      parent[i]=j;
}
void KRUSKAL::get_input()
{
 cout<<"\n Enter Total number of nodes: ";
 cin>>tot_nodes;
 cout<<"\n Enter Total number of edges: ";
 cin>>tot_edges;
}
void KRUSKAL::create()
{
 for(int k=0;k<tot_edges;k++)
 {
      cout<<"\n Enter Edge in (V1 V2)form ";
      cin>>G[k].v1>>G[k].v2;
      cout<<"\n Enter Corresponding Cost ";
      cin>>G[k].cost;
 }
}
int KRUSKAL::Minimum(int n)
{
 int i,small,pos;
 small=INFINITY;
 pos=-1;
 for(i=0;i<n;i++)
```

```
{
    if(G[i].cost<small)
    {
        small=G[i].cost;
        pos=i;
    }
}
return pos;
}
void KRUSKAL::spanning_tree()
{
    int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
    int sum;
    count=0;
    k=0;
    sum=0;
    for(i=0;i<tot_nodes;i++)
     parent[i]=i;
    while(count!=tot_nodes-1)
    {
     pos=Minimum(tot_edges);//finding the minimum cost edge
     if(pos==-1)//Perhaps no node in the graph
        break;
     v1=G[pos].v1;
     v2=G[pos].v2;
     i=Find(v1,parent);
     j=Find(v2,parent);
     if(i!=j)
     {
        tree[k][0]=v1;//storing the minimum edege in array tree[]
        tree[k][1]=v2;
        k++;
        count++;
        sum+=G[pos].cost;//accumulating the total cost of MST
        Union(i,j,parent);
     }
     G[pos].cost=INFINITY;
    }
    if(count==tot_nodes-1)
    {
     cout<<"\n Spanning tree is..."<<endl;
     cout<<"\n-----------------------"<<endl;
     for(i=0;i<tot_nodes-1;i++)
     {
        cout<<"["<<tree[i][0];
        cout<<" ";
        cout<<tree[i][1]<<"]"<<endl;
```



```
     }
     cout<<"\n-----------------------"<<endl;
     cout<<"Cost of Spanning Tree is = "<<sum<<endl;
    }
    else
    {
     cout<<"There is no Spanning Tree"<<endl;
    }
}
int main()
{
 KRUSKAL obj;
 cout<<"\n\t Graph Creation ";
 obj.get_input();
 obj.create();
 obj.spanning_tree();
return 0
}
```

Output

```
    Graph Creation
Enter Total number of nodes: 4

Enter Total number of edges: 5

Enter Edge in (V1 V2)form 1 4

Enter Corresponding Cost 3

Enter Edge in (V1 V2)form 1 2

Enter Corresponding Cost 8

Enter Edge in (V1 V2)form 2 3

Enter Corresponding Cost 6

Enter Edge in (V1 V2)form 2 4

Enter Corresponding Cost 2

Enter Edge in (V1 V2)form 3 4

Enter Corresponding Cost 1

Spanning tree is...

-----------------------
```

[3 - 4]
[2 - 4]
[1 - 4]

Cost of Spanning Tree is = 6

Review Questions

1. *Define minimum spanning tree with an example* **SPPU : May-10,11, Marks 2**
2. *Write a note on Kruskal's algorithm* **SPPU : May-11, Marks 6**
3. *What is minimal spanning tree ? How it is different from the shortest path sequence of a given graph? Justify your answer with an example.* **SPPU : May-13, Marks 4**

5.8 Shortest Path using Dijkstra's Algorithm **SPPU : May-19, Marks 6**

- Dijkstra's Algorithm is a popular algorithm for finding shortest path.
- This algorithm is called **single source shortest path** algorithm because in this algorithm, for a given vertex called **source** the shortest path to all other vertices is obtained.
- This algorithm applicable to graphs with non-negative weights only.

Example 5.8.1 *Obtain the shortest path for the given graph.*

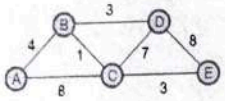
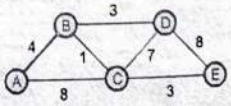
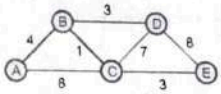
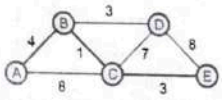


Fig. 5.8.1

Solution : Now we will consider each vertex as a source and will find the shortest distance from this vertex to every other remaining vertex. Let us start with vertex A.

Source vertex	Distance with other vertices		Path shown in graph
A	A-B,	path = 4	
	A-C,	path = 8	
	A-D,	path = ∞	
	A-E	path = ∞	

B	B-C,	path = 4 + 1	
	B-D,	path = 4 + 3	
	B-E	path = ∞	
C	C-D,	path = 5 + 7 = 12	
	C-E,	path = 5 + 3 = 8	
D	D-E,	path = 7 + 8 = 15	

Thus the shortest distance from A to E is obtained.

that is A - B - C - E with path length = 4 + 1 + 3 = 8. Similarly other shortest paths can be obtained by choosing appropriate source and destination

Algorithm

```
Algorithm  Dijkstra(int cost[1...n,1...n],int source,int dist[])

for i ← 0 to tot_nodes
{
        dist[i] ← cost[source,i]//initially put the
        s[i] ← 0 //distance from source vertex to i
        //i is varied for each vertex
        path[i] ← source//all the sources are put in path
}
    s[source] ← 1          ← Start from each source node
for(i ← 1 to tot_nodes)
{
        min_dist ← infinity;
        v1 ← -1//reset previous value of v1
        for(j ← 0 to tot_nodes-1)
        {
            if(s[j]=0)then
            {
                if(dist[j]<min_dist)then      ← Finding minimum distance from selected source node.That is : source-j represents min. dist. edge
                {
                 min_dist ← dist[j]
                 v1 ← j
                }
            }
        }
        s[v1] ← 1
        for(v2 ← 0 to tot_nodes-1)
```

```
        {
        If(s[v2]=0)then
        {
            If(dist[v1]+cost[v1][v2]<dist[v2])then
            {
                dist[v2] ← dist[v1]+cost[v1][v2]
                path[v2] ← v1
            }
        }
        }
    }
}
```

v_1 is next selected destination vertex with shortest distance. All such vertices are accumulated in array path[ ]

C Function

```
void Dijkstra(int tot_nodes,int cost[10][10],int source,int dist[])
{
 int i,j,v1,v2,min_dist;
 int s[10];
 for(i=0;i<tot_nodes;i++)
 {
        dist[i]=cost[source][i];//initially put the
        s[i]=0; //distance from source vertex to i
        //i is varied for each vertex
        path[i]=source;//all the sources are put in path
 }
 s[source]=1;
 for(i=1;i<tot_nodes;i++)
 {
        min_dist=infinity;
        v1=-1;//reset previous value of v1
        for(j=0;j<tot_nodes;j++)
        {
         if(s[j]==0)
         {
            if(dist[j]<min_dist)
            {
             min_dist=dist[j];//finding minimum disatnce
             v1=j;
            }
         }
        }
        s[v1]=1;
        for(v2=0;v2<tot_nodes;v2++)
        {
```



```
            if(s[v2]==0)
            {
                if(dist[v1]+cost[v1][v2]<dist[v2])
                {
                    dist[v2]=dist[v1]+cost[v1][v2];
                    path[v2]=v1;
                }
            }
        }
    }
}
```

Example 5.8.2 *Find the shortest path using Dijkstra algorithm between node A and node F.*

SPPU : May-19, Marks 6

Solution : Consider source node A.

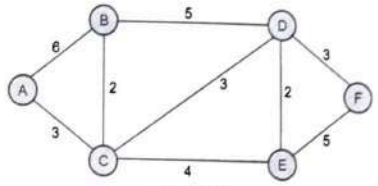


Fig. 5.8.2

Step 1 : S = {A} T = {B, C, D, E, F}

d (A, B) = 6 d (A, D) = ∞

d (A, C) = 3 d (A, E) = ∞

d (A, F) = ∞

The d(A, C) = 3 is minimum distance. Hence we will select vertex C.

Step 2 : S = {A, C} T = {B, D, E, F}

d (A, B) = 6

d (A, D) = d (A, C) + d (C, D) = 6

d (A, E) = d (A, C) + d (C, E) = 7

d (A, F) = ∞

We select d (A, B) as minimum distance. Hence select vertex B.

Step 3 : S = {A,C, B}, T = {D, E, F}

d (A, D) = min {[d (A, B) + d (B, D)], [d (A, C) + d (C, D)]}

= min {[6+5], [3+3]}

d (A, D) = 6

d (A, E) = 7

d (A, F) = ∞

We will select vertex D.

Step 4 : S = {A, C, B, D}, T = {E, F}

d (A, E) = 7

d (A, F) = d (A, D) + d (D, F) = 6 + 3

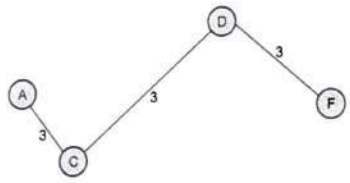
d (A, F) = 9

We will select vertex E.

Step 5 : S = {A, C, B, D, E} T = {F}

d (A, F) = 9

Step 6 : Thus we obtain the shortest path from A and F as,



5.9 Topological Sorting

SPPU : May-13,16, Dec.-19, Marks 6

Algorithm

Following are the steps to be followed in this algorithm -

1. From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it. If there are more than one such vertices then break the tie randomly.
2. Note the vertices that are deleted.
3. All these recorded vertices give topologically sorted list.

Let us understand this algorithm with some examples -

Example 5.9.1 *Sort the digraph for topological sort.*

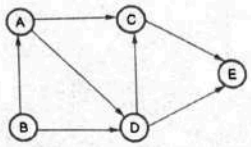


Fig. 5.9.1

Solution : We will follow following steps to obtain topologically sorted list.

Choose vertex B, because it has no incoming edge, delete it along with its adjacent edges.

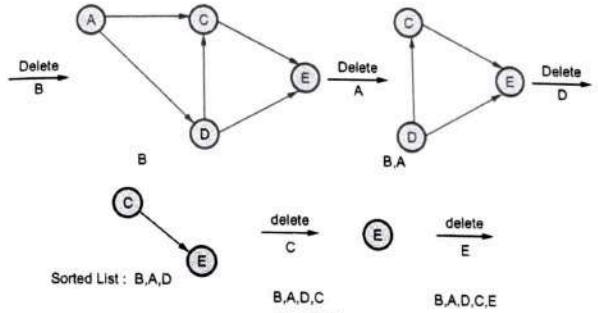


Fig. 5.9.2

Hence the list after topological sorting will be B, A, D, C, E.

Example 5.9.2 *Sort the given digraph using topological sort.*

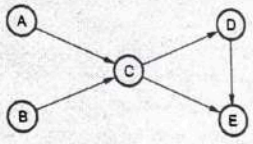


Fig. 5.9.3

Solution : Let us start with the vertex with no incoming edge. There are two such vertices A and B. The tie can be broken arbitrarily. Let us first delete vertex B.

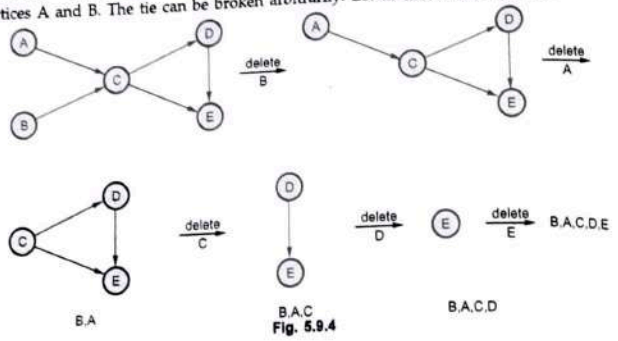


Fig. 5.9.4

Thus the topologically sorted list is B, A, C, D, E.

Example 5.9.3 *Consider the following graph, each node contains subject course number :*

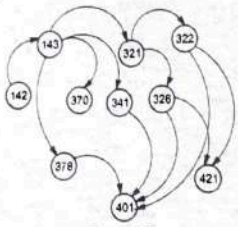


Fig. 5.9.5

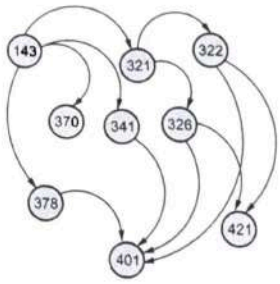
To complete a course, students need to take all the course subjects. Students need to take into consideration which courses are prerequisites for other courses when making a schedule for the upcoming semester so that 370 can not taken before 143, but the former can be taken along with 341 and 370. Help the students by giving the order of subjects they need to take to complete the course. [Hint : Topological sort]

SPPU : May-16, Marks 4

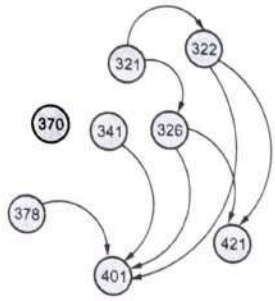
Solution : We will apply topological sorting technique.

Each time we will find a vertex with no incoming edges. Delete that vertex along with all the outgoing edges from it. If there are more than such vertices then break the tie randomly.

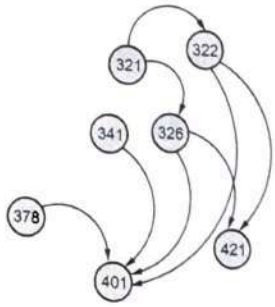
Step 1 : Delete 142



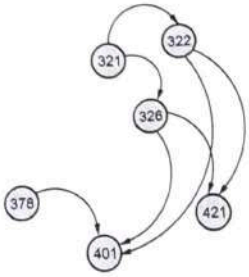
Step 2 : Delete 143



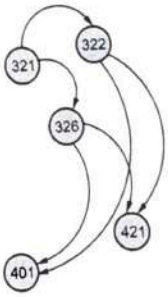
Step 3 : Delete 370



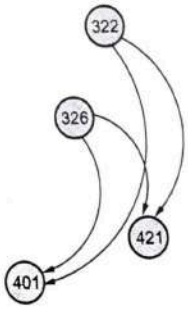
Step 4 : Delete 341



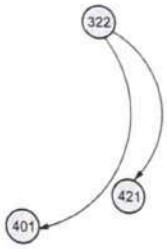
Step 5 : Delete 378



Step 6 : Delete 321



Step 7 : Delete 326



Step 8 : Delete 322

Step 9 : Delete 401

Step 10 : Delete 421

Thus the course schedule can be in following order.

142	143	370, 341	378	321	326, 322	401, 421

Review Questions

1. *What is topological sorting? Explain with example.* **SPPU : May-13, Marks 6**
2. *What is topological sorting ? Where topological sorting can be applicable ? Explain it with suitable example.* **SPPU : Dec.-19, Marks 6**

Part II : Symbol Table

5.10 Notion of Symbol Table

SPPU : May-12, Marks 6

- Definition

The symbol table is defined as **the set of Name** and **Value** pairs.

For example

	Name	Value
0	a	10
1	b	0
2	c	0
3		
4		
5		
6		

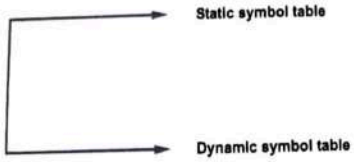
Fig. 5.10.1 Symbol table storing 3 identifiers

- **Use of Symbol Table**

The symbol tables are typically used in **compilers**. Basically compiler is a program which scans the application program (for instance : your C program) and produces machine code. During this scan compiler stores the identifiers of that application program in the symbol table. These identifiers are stored in the form of name, value address, type. Here the name represents the name of identifier, value represents the value stored in an identifier, the address represents memory location of that identifier and type represents the data type of identifier. Thus compiler can keep track of all the identifiers with all the necessary information.

- **Types of Symbol Tables**

There are two types of symbol tables.



The static symbol table stores the fixed amount of information whereas dynamic symbol table stores the dynamic information.

These symbol tables are normally used to implement static and dynamic data structures. Hence there can be static tree tables or dynamic tree tables, which are implemented using static and dynamic symbol tables respectively.

Examples of static symbol table : Optimal binary search tree (OBST), Huffman's coding can be implemented using static symbol table.

Example of dynamic symbol table : An AVL tree is implemented using dynamic symbol table.

- **Advantages of using Symbol Tables**

Following are some advantages of using symbol tables. As symbol table is normally used in compiler, these advantages are relevant to compilers -

1. During compilation of source program, **fast look up** for the required identifiers is possible due to use of symbol table.
2. The runtime allocation for the identifiers is managed using symbol tables.
3. Use of symbol table allows to handle certain issues like scope of identifiers, and implicit declarations.

- **Operations on Symbol Table**

Following operations can be performed on symbol table -

1. Insertion of an item in the symbol table.
2. Deletion of any item from the symbol table.
3. Searching of desired item from symbol table.

5.10.1 Applications of Symbol Table

1. Optimal Binary Search Tree
2. AVL Tree
3. Huffman's Tree
4. Heap Sort

Review Question

1. *Explain symbol table with its applications.* **SPPU : May-12, Marks 6**

5.11 OBST

SPPU : Dec.-09, May-16, Marks 10

Problem Description

Let $\{a_1, a_2, \ldots a_n\}$ be a set of identifiers such that $a_1 < a_2 < a_3$. Let p(i) be the probability with which we can search for a_i and $q_{(i)}$ be the probability of searching element x such that $a_i < x < a_{i+1}$ and $0 \le i \le n$. In other words $p_{(i)}$ is the probability of successful search and q(i) be the probability of unsuccessful search. Also $\sum_{1 \le i \le n} p_{(i)} + \sum_{1 \le i \le n} q_{(i)}$ then obtain a tree with minimum cost. Such a tree with optimum cost is called **optimal binary search tree.**

To solve this problem we will perform following steps.

Step 1 : Notations used

Let,

$T_{ij} = OBST(a_{i+1}, \cdots, a_j)$

C_{ij} denotes the $cost(T_{ij})$.

W_{ij} is the weight of each T_{ij}.

T_{0n} is the final tree obtained.

T_{00} is empty.

$T_{i,i+1}$ is a single-node tree that has element a_{i+1}.

During the computations the root values are computed and r_{ij} stores the root value of T_{ij}.

Step 2 : We will apply following formula for computing each sequence.

$$C(i, j) = \min_{i < k \le j} \{C(i, k-1) + C(k, j)\} + W(i, j)\}$$

$$W(i, j) = W[i, j-1] + p[j] + q[j];$$

$$r[i, j] = k$$

Example 5.11.1 *Consider n = 4 and (q_1, q_2, q_3, q_4) = (do, if, read, while). The values for p's and q's are given as p(1 : 4) = (3, 3, 1, 1) and q(0 : 4) = (2, 3, 1, 1, 1). Construct the optimal binary search tree.* **SPPU : May-16, Marks 8**

Solution : We will apply following formulae for the computation of W, C and r.

$$W_{i,i} = q_i,\ r_{i,i} = 0,\ C_{i,i} = 0$$

$$W_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$r_{i,i+1} = i + 1$$

$$C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$W_{i,j} = W_{i,j-1} + p_j + q_j$$

$$r_{i,j} = k$$

$$C_{i,j} = \min_{i < k \le j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

We will construct tables for values of W, C and r.

Let $i = 0$

$$W_{00} = q_0 = 2$$

When i = 1

$$W_{11} = 3$$

When i = 2

$$W_{22} = 1$$

When i = 3

$$W_{33} = 1$$

When i = 4

$$W_{44} = q_4 = 1$$

When i = 0 and j – i = 1 then

$$W_{01} = q_0 + q_1 + p_1 = 2 + 3 + 3$$

$$W_{01} = 8$$

When i =1 and j – i = 1

$$W_{12} = q_1 + q_2 + p_2 = 3 + 1 + 3$$

$$W_{12} = 7$$

When i = 2 and j – i = 1

$$W_{23} = q_2 + q_3 + p_3 = 1 + 1 + 1$$

$$W_{23} = 3$$

When i = 3 and j – i = 1

$$W_{34} = q_3 + q_4 + p_4 = 1 + 1 + 1$$

$$W_{34} = 3$$

Now, when i = 0 and j – i = 2

$$W_{ij} = W_{i,j-1} + p_j + q_j$$

$$W_{02} = W_{01} + p_2 + q_2 = 8 + 3 + 1$$

$$W_{02} = 12$$

When i =1 and j – i = 2

$$W_{13} = W_{12} + p_3 + q_3 = 7 + 1 + 1$$

$$W_{13} = 9$$

When i = 2 and j – i = 2 then

$$W_{24} = W_{23} + p_4 + q_4 = 3 + 1 + 1$$

$$W_{24} = 5$$

Now, when $i = 0$ and $j - i = 3$ then

$$W_{03} = W_{02} + p_3 + q_3 = 12 + 1 + 1$$
$$W_{03} = 14$$

When $i = 1$ and $j - i = 3$ then

$$W_{14} = W_{13} + q_4 + p_4 = 9 + 1 + 1$$
$$W_{14} = 11$$

When $i = 0$ and $j - i = 4$ then

$$W_{04} = W_{03} + q_4 + p_4 = 14 + 1 + 1$$
$$W_{04} = 16$$

The table for W can be represented as,

$j-i$ \ i	0	1	2	3	4
0	$W_{00} = 2$	$W_{11} = 3$	$W_{22} = 1$	$W_{33} = 1$	$W_{44} = 1$
1	$W_{01} = 8$	$W_{12} = 7$	$W_{12} = 3$	$W_{34} = 3$	
2	$W_{02} = 12$	$W_{13} = 9$	$W_{24} = 5$		
3	$W_{03} = 14$	$W_{14} = 11$			
4	$W_{04} = 16$				

We will now compute for C and r.

As $C_{i,i} = 0$ and $r_{i,i} = 0$

$C_{00} = 0$ $C_{11} = 0$ $C_{22} = 0$ $C_{33} = 0$ $C_{44} = 0$

$r_{00} = 0$ $r_{11} = 0$ $r_{22} = 0$ $r_{33} = 0$ $r_{44} = 0$

Similarly, $C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$ and $r_{i,i+1} = i+1$

$\therefore$ When $i = 0$

$$C_{01} = q_0 + q_1 + p_1 = 2 + 3 + 3$$
$$C_{01} = 8$$
$$r_{01} = 1$$

When $i = 1$

$$C_{12} = q_1 + q_2 + p_2 = 3 + 1 + 3$$
$$C_{12} = 7 \quad \text{and} \quad r_{12} = 2$$

When $i = 2$

$$C_{23} = q_2 + q_3 + p_3 = 1 + 1 + 1$$
$$C_{23} = 3 \quad \text{and} \quad r_{23} = 3$$

When $i = 3$

$$C_{34} = q_3 + q_4 + p_4 = 1 + 1 + 1$$
$$C_{34} = 3 \quad \text{and} \quad r_{34} = 4$$

Now we will compute C_{ij} and r_{ij} for $j - 1 \geq 2$.

As $$C_{i,j} = \min_{i < k \leq j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

Hence we will find k.

For C_{02} we have $i = 0$ and $j = 2$. Value of K will be between i and j i.e. between 0 and 2. For $r_{i,j-1}$ to $r_{i+1,j}$ i.e. for r_{01} to $r_{1,2}$. We will compute minimum value of C_{ij}.

Let $r_{01} = 1$ and $r_{12} = 2$. Then we will assume value of $k = 1$ and will compute C_{ij}. Similarly with $k = 2$ we will compute C_{ij} and will pick up minimum value of C_{ij} only.

Let us compute C_{ij} with following formula,

$$C_{ij} = C_{i,k-1} + C_{kj}$$

For $k = 1, i = 0, j = 2$.

$$C_{02} = C_{00} + C_{12} = 0 + 7$$
$$C_{02} = 7 \qquad \text{...(5.11.1)}$$

For $k = 2, i = 0, j = 2$

$$C_{02} = C_{01} + C_{22} = 8 + 0$$
$$C_{02} = 8 \qquad \text{...(5.11.2)}$$

From equations (5.11.1) and (5.11.2) we can select minimum value of C_{02} is 7. That means **$k = 1$** gives us minimum value of $C_{i,j}$.

Hence **$r_{ij} = r_{02} = k = 1$**

Now,

$$C_{02} = \min\{C_{(i,k-1)} + C_{kj}\} + W_{ij}$$
$$= 7 + W_{02} = 7 + 12$$
$$C_{02} = 19$$

Continuing in this fashion we can compute C_{ij} and r_{ij}. It is as given below.

j − i	0	1	2	3	4
0	$C_{00} = 0$, $r_{00} = 0$	$C_{11} = 0$, $r_{11} = 0$	$C_{22} = 0$, $r_{22} = 0$	$C_{33} = 0$, $r_{33} = 0$	$C_{44} = 0$, $r_{44} = 0$
1	$C_{01} = 8$, $r_{01} = 1$	$C_{12} = 7$, $r_{12} = 2$	$C_{23} = 3$, $r_{23} = 3$	$C_{34} = 3$, $r_{34} = 4$	
2	$C_{02} = 19$, $r_{02} = 1$	$C_{13} = 12$, $r_{13} = 2$	$C_{24} = 8$, $r_{24} = 3$		
3	$C_{03} = 25$, $r_{03} = 2$	$C_{14} = 19$, $r_{14} = 2$			
4	$C_{04} = 32$, $r_{04} = 2$ } T_{04}				

Therefore T_{04} has a root r_{04}. The value of r_{04} is 2. From (a_1, a_2, a_3, a_4) = (do, if, read, while) a_2 becomes the root node. Hence root is **if**.

$$r_{ij} = k$$

$$r_{04} = 2$$

Then r_{ik-1} becomes left child and r_{kj} becomes the right child. In other words r_{01} becomes the left child and r_{24} becomes right child of r_{ij}.

Here $r_{01} = 1$ so a_1 becomes left child of a_2 and $r_{24} = 3$ so a_3 becomes right child of a_2.

For the node r_{24} i = 2, j = 4 and k =3. Hence left child of it is $r_{ik-1} = r_{22} = 0$. That means left child of $r_{24} = a_3$ is empty. The right child of r_{24} is $r_{34} = 4$. Hence a_4 becomes right child of a_3. Thus all n = 4 nodes are used to build a tree. The optimal binary search tree is show in Fig. 5.11.1

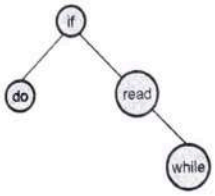


Fig. 5.11.1 OBST

5.11.1 Algorithm

```
Algorithm Wt(p,q,W,n)
//p is an input array [1..n]
//q is an input array [0..n]
// W[i,j] will be output for this function such that W[0..n,0..n]
{
  for i←1 to n do
    W[i,i] ← q[i]; // initialize
  for m←1 to n do
  {
   for i←0 to n-m do
    {
      k ← i+m;
      W[i,k]←W[i,k-1] + p[k] + q[k];
    }
  }
write(W[0:n]);//print the values of W
}
```

- For computing C_{ij}, r_{ij} following algorithm is used.

```
Algorithm OBST(p,q,W,C,r)
{
  //computation of first two rows
for i=0 to n do
{
          C[i,i] ← 0;
                r[i,i] ← 0;
                C[i,i+1] ← q[i]+q[i+1]+p[i+1];
                r[i,i+1]←i+1;
          }
          for m ← 2 to n do
          {
                For ← 0 to n-m do
{
     j←i+m;
     k←Min_Value(C,r,i,j); //call for algorithm Min_Value
// minimum value of C_ij is obtained for deciding value of k
C[i,j]←W[i,j] + C[i,k-1] + C[k,j];
r[i,j]←k;
}
}
write(C[0:n],r[0:n]);//print values of C_ij and r_ij
}

Algorithm Min_Value(C,r,i,j)
{
```

```
    minimum← ∞;
// finding the range of k
    for (m←r[i,j-1] to r[i+1,j]) do
    {
        if(C[i,m-1]+C[m,j])< minimum then
        {
            minimum ← C[i,m-1]+C[m,j];
            k←m;
        }
        return k;//This k gives minimum value of C
    }
}
```

Following algorithm is used for creating the tree T_{ij}.

```
Algorithm build_tree(r,a,i,j)
{
  T ← new(node); //allocate memory for a new node
  k ← r[i,j];
  T -> data ← a[k];
  if (j==i+1)
     return;
  T->left=build_tree(r,a,i,k-1);
  T->right=build_tree(r,a,k,j);
}
```

In order to obtain the optimal binary search tree we will follow :

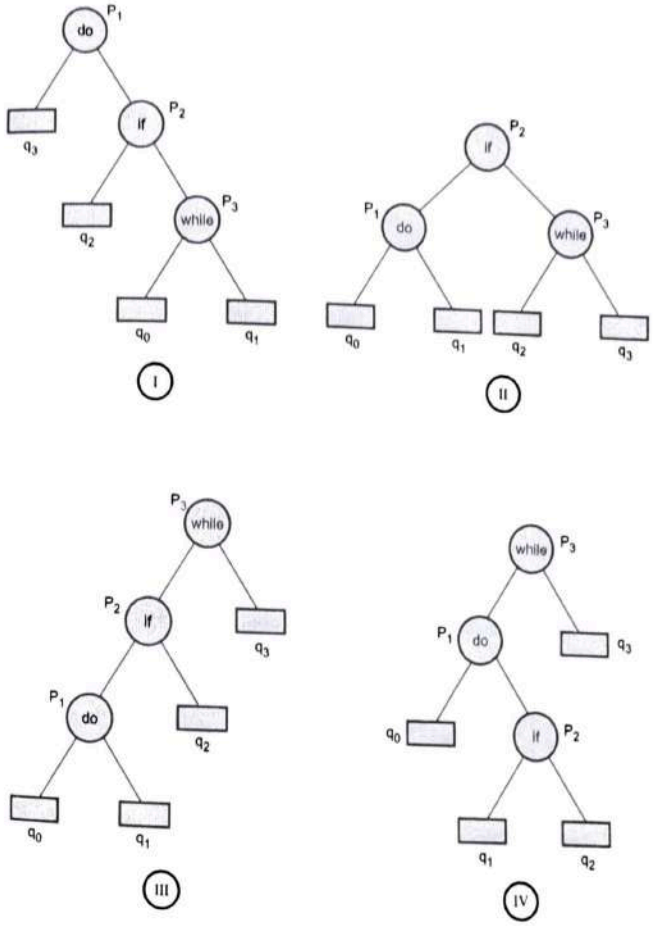
1. Compute the weight using Wt function.
2. Compute C_{ij} and r_{ij} using OBST.
3. Build the tree using build_tree function.

Example 5.11.2 *How many binary search trees (BSTs) can be constructed for given 'n' identifiers ? i) Construct all possible Binary Search Trees for the identifier set (q_1, q_2, q_3) = (do, if, while)*

ii) Compute the total cost of each BST constructed by you assuming equal probabilities for successful and unsuccessful search i.e. (p_1, p_2, p_3) = (1/7, 1/7, 1/7) for successful search and (q_0, q_1, q_2, q_3) = (1/7, 1/7, 1/7, 1/7) for unsuccessful search. Which BST is an optimal binary search tree ?

iii) Compute the total cost of each BST constructed by you assuming probabilities for successful and unsuccessful search as : (p_1, p_2, p_3) = (0.5, 0.1, 0.05) for successful search and (q_0, q_1, q_2, q_3) = (0.15, 0.1, 0.05, 0.05) for unsuccessful search. Which BST is an optimal binary search tree ? **SPPU : Dec.-09, Marks 10**

Solution : i) All possible binary search tree for (do, if, while) are -



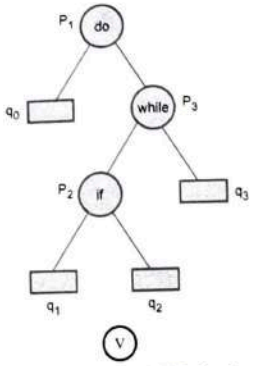


Fig. 5.11.2 All possible BSTs for 3 nodes

Thus there are 5 possible BSTs for 3 identifiers. Let a_i be a node then the cost can be computed by following formula -

ii) Cost (tree) = $\sum_{1 \le i \le n} P(i) * \text{level}(a_i) + \sum_{0 \le i \le n} q(i) * (\text{level}(E_i) - 1)$

where E_i represents the equivalence class for all the identifiers a_i.

$$\text{Cost(tree I)} = \underbrace{\left[\frac{1}{7}*3+\frac{1}{7}*2+\frac{1}{7}*1\right]}_{\text{successful nodes (such as do, if, while)}} + \underbrace{\left[\frac{1}{7}*(4-1)+\frac{1}{7}*(4-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*(2-1)\right]}_{\text{(unsuccessful nodes denoted by square)}} = \frac{6}{7}+\frac{9}{7}$$

Cost (tree I) = $\frac{15}{7}$

$$\text{Cost(treeII)} = \left[\left(\frac{1}{7}*1\right)+\left(2*\frac{1}{7}+2*\frac{1}{7}\right)\right]+\left[\frac{1}{7}*(3-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*(3-1)\right] = \frac{5}{7}+\frac{8}{7}$$

Cost(tree II) = $\frac{13}{7}$

$$\text{Cost(tree III)} = \left[\frac{1}{7}*1+\frac{1}{7}*2+\frac{1}{7}*3\right]+\left[\frac{1}{7}*(2-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*2*(4-1)\right] = \frac{6}{7}+\frac{9}{7}$$

Cost (tree III) = $\frac{15}{7}$

$$\text{Cost (tree IV)} = \left[\frac{1}{7}*1+\frac{1}{7}*2+\frac{1}{7}*3\right]+\left[\frac{1}{7}*(2-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*(4-1)+\frac{1}{7}*(4-1)\right] = \frac{6}{7}+\frac{9}{7}$$

Cost (tree IV) = $\frac{15}{7}$

$$\text{Cost (tree V)} = \left[\frac{1}{7}*1+\frac{1}{7}*2+\frac{1}{7}*3\right]+\left[\frac{1}{7}*(2-1)+\frac{1}{7}*(3-1)+\frac{1}{7}*(4-1)+\frac{1}{7}*(4-1)\right]$$

Cost (tree V) = $\frac{15}{7}$

From all these computations tree II is an optimal binary search tree because it has least cost.

iii) Using the same formula as mentioned in (ii) we will compute the cost of all the trees shown in Fig. 5.11.2.

$$\text{Cost (tree I)} = [(0.5*1+0.1*2+0.05*3)]+[0.05*1+0.05*2+0.15*3+0.1*3]$$
$$= [0.5+0.2+0.15]+[0.90] = 0.80+0.90$$

Cost (tree I) = 1.70

$$\text{Cost (tree II)} = [P_2*1+P_1*2+P_3*2]+[q_0*2+q_1*2+q_2*2+q_3*2]$$
$$= [0.1*1+0.5*2+0.05*2]+[0.15*2+0.1*2+0.05*2+0.05*2]$$

Cost (tree II) = 1.90

$$\text{Cost (tree III)} = [P_1*3+P_2*2+P_3*1]+[3*(q_0+q_1)+2*(q_2)+1*q_3]$$
$$= [0.5*3+0.1*2+0.05*1]+[3*(0.15+0.1)+2*0.05+0.05]$$

Cost (tree III) = 1.30

$$\text{Cost (tree IV)} = [P_1*2+P_2*3+P_3*1]+[2*q_0+3*(q_1+q_2)+1*q_3]$$
$$= [0.5*2+0.1*3+0.05*1]+[2*0.15+3*(0.1+0.05)+0.05]$$

Cost (tree IV) = 1.25

$$\text{Cost (tree V)} = [P_1*1+P_2*3+P_3*2]+[1*q_0+3*q_1+3*q_2+2*q_3]$$
$$= [0.5*1+0.1*3+0.05*2]+[1*0.15+3*0.1*3*0.05+2*0.05]$$

Cost (tree V) = 1.60

All these computations show that tree IV is optimal binary search tree.

5.12 AVL Trees

SPPU : Dec.-05,07,08,14,19, May-10,11,14,15,16,18,19, Marks 10

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to **height of subtrees**. The tree can be made balanced and because of this retrieval of any node can be done in **O(logn) times**, where n is total number of nodes. From the name of these scientists the tree is called **AVL tree**.

Definition

An empty tree is height balanced if T is a non empty binary tree with T_L and T_R as its left and right subtrees. The T is height balanced if and only if

i) T_L and T_R are height balanced.

ii) $h_L - h_R <= 1$ where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

Definition of Balance Factor

The balance factor BF(T) of a node in binary tree is defined to be $\mathbf{h_L - h_R}$ where h_L and h_R are heights of left and right subtrees of T.

For any node in AVL tree the balance factor i.e. BF(T) is **–1, 0 or +1.**

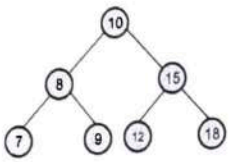


Fig. 5.12.1 AVL tree

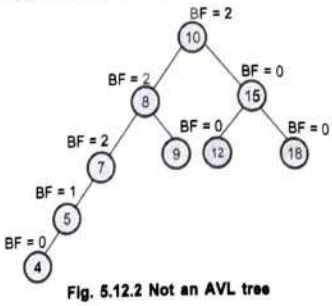


Fig. 5.12.2 Not an AVL tree

5.12.1 Height of AVL Tree

Theorem : The height of AVL tree with n elements (nodes) is O(logN).

Proof : Let an AVL tree with N nodes in it. N_h be the minimum number of nodes in an AVL tree of height h.

In worst case, one subtree may have height h–1 and other subtree may have height h–2. And both these subtrees are AVL trees. Since for every node in AVL tree the height of left and right subtrees differ by at most 1. Hence,

$$N_h = N_{h-1} + N_{h-2} + 1$$

where N_h denotes the minimum number of nodes in an AVL tree of height h.

$\therefore \quad N_0 = 0 \qquad N_1 = 2$

We can also write it as,

$$N > N_h = N_{h-1} + N_{h-2} + 1$$

$$> 2N_{h-2}$$
$$> 4N_{h-4}$$
$$\vdots$$
$$> 2^i N_{h-2^i}$$

If value of h is even, let $i = \frac{h}{2-1}$

Then equation becomes,

$$N > 2^{h/2-1} N_2$$

$$= N > 2^{h/2-1} \times 4 \quad \because N_2 = 4$$

$$= O(\log N)$$

If value of h is odd, let $i = (h-1)/2$ then equation becomes.

$$N > 2^{(h-1)/2} N_1$$

$$N > 2^{(h-1)/2} \times 1 \quad \because N_1 = 1$$

$$\therefore \quad h = O(\log N)$$

This proves that height of AVL tree is always O(log N). Hence search, insertion and deletion can be carried out in logarithmic time.

Theorem : The height of AVL tree h with minimum number of nodes n is 1.44 log n.

Proof : Let F_h be the AVL tree with height h having minimum number of nodes. The tree can be shown as below.

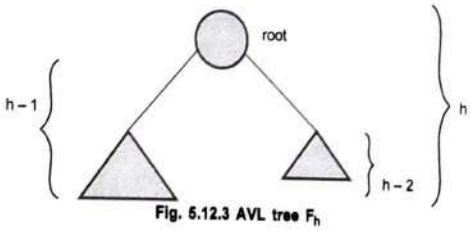


Fig. 5.12.3 AVL tree F_h

Hence minimum number of nodes in AVL tree is

= Length of left subtree + Length of right subtree + 1

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1$$

Also $\quad |F_0| = 1$ and $|F_1| = 2$

This denotes the Fibonacci tree.

Adding 1 to both sides.

$$|F_h| + 1 = (|F_{h-1}| + 1) + (|F_{h-2}| + 1)$$

Thus the numbers $|F_h|+1$ are Fibonacci numbers. Using approximate formula for Fibonacci numbers, we will get an approximate value as :

$$|F_h|+1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

Taking logarithm

$$= h = 1.44 \log |F_h|$$

Hence the worst case height of an AVL tree with n nodes is 1.44 log n.

The Fibonacci trees are as shown below.

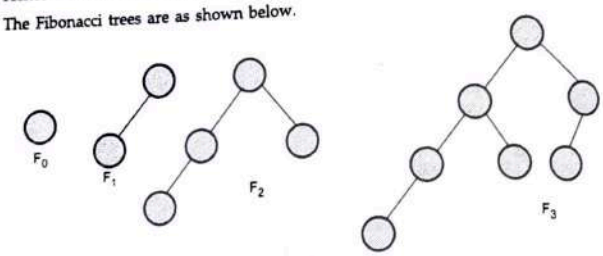


Fig. 5.12.4

Theorem : The minimum number of nodes in an AVL tree with height h is,

$$S_h = S_{h-2} + S_{h-1} + 1$$

Proof : We can prove this using the induction method.

Basis of induction

If **h = 0** then the minimum number of nodes is 1. If **h = 1** then minimum number of nodes are **2**.

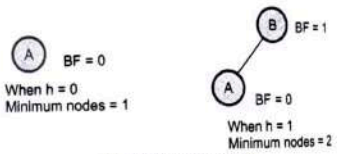


Fig. 5.12.5 AVL trees

This proves that

$$S_h = S_{h-2} + S_{h-1} + 1$$

Induction Hypothesis :

We assume that $S_m = S_{m-2} + S_{m-1} + 1$ is true.

Induction step : Now consider that we can obtain minimum number of nodes for an AVL tree with height k + 1, using following formula

$$S_{k+1} = S_{k+1-2} + S_{k+1-1} + 1$$

$$S_{k+1} = S_{k-1} + S_k + 1 \quad \text{... (5.12.1)}$$

If we assume m = k + 1 then equation (5.12.1) becomes,

$$S_m = S_{m-2} + S_{m-1} + 1$$

This statement is true. ... ∵ Induction hypothesis.

Hence

$$S_{k+1} = S_{k-1} + S_k + 1 \text{ is also true.}$$

Thus it is true that

$$S_k = S_{k-2} + S_k + 1$$

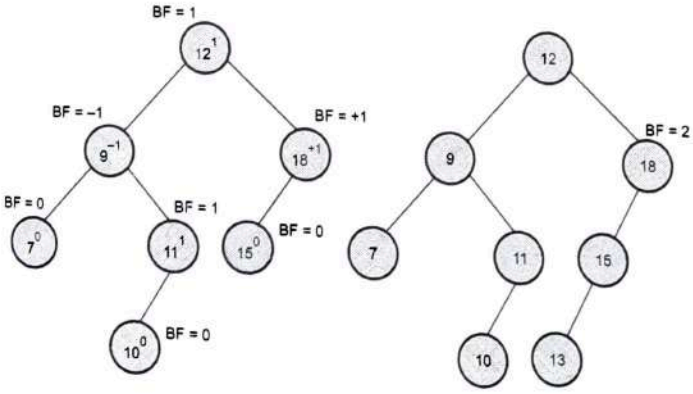
Hence for height h minimum number of nodes in an AVL tree can be obtained using the formula

$$S_h = S_{h-2} + S_{h-1} + 1$$

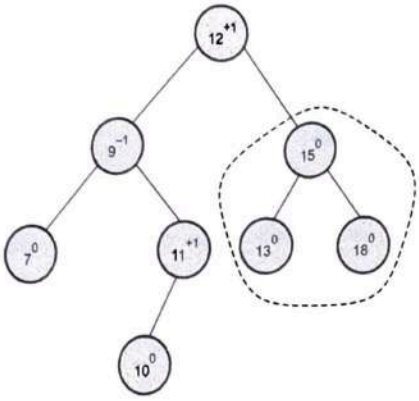
5.12.2 Representation of AVL Tree

- The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factor as – 1, 0 or + 1.
- After insertion of any node in an AVL tree if the balance factor of any node becomes other than –1, 0, or + 1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.

For example :



Original AVL tree **Insert 13 property violated**



Restore AVL property

Fig. 5.12.6 Restoring AVL property

- After an insertion of a new node if balance condition gets destroyed, then the nodes on that path (new node insertion point to root) needs to be readjusted. That means only the affected subtree is to be rebalanced.
- The rebalancing should be such that entire tree should satisfy AVL property.

In above given example -

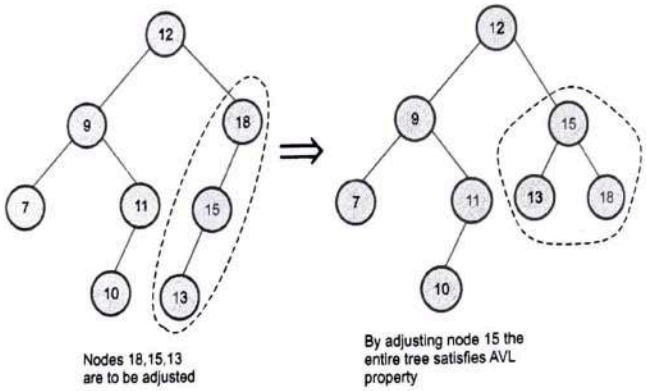


Fig. 5.12.7

5.12.3 Algorithms and Analysis of AVL Trees

There are 3 operations that are commonly carried out on AVL tree. And those are

1. Insertion of a node.
2. Deletion of a node at any position.
3. Searching the desired node.

After performing insertion and deletion operations, sometimes the tree structure needs to be rearranged, because an AVL tree is a height balanced tree in which the balance factor should always be –1, 0 or +1. Let us now understand these operations with the help of examples.

5.12.4 Insertion

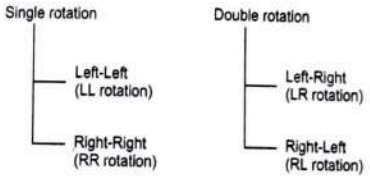
There are four different cases when rebalancing is required after insertion of new node.

1. An insertion of new node into left subtree of left child (LL).
2. An insertion of new node into right subtree of left child (LR).
3. An insertion of new node into left subtree of right child (RL).
4. An insertion of new node into right subtree of right child (RR).

There is a symmetry between case 1 and 4. Similarly symmetry exists between case 2 and 3.

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree.

There are two types of rotations.



Insertion algorithm

1. Insert a new node as new leaf just as in ordinary binary search tree.
2. Now trace the path from insertion point (new node inserted as leaf) towards root. For each node 'n' encountered, check if heights of left (n) and right (n) differ by at most 1.

a) If yes, move towards parent (n).

b) Otherwise restructure by doing either a single rotation or a double rotation.

Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any ancestor on 'n'.

Different rotations in AVL tree

1. LL rotation

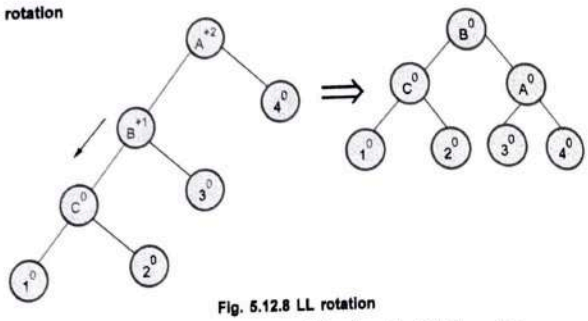


Fig. 5.12.8 LL rotation

When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor + 2.

The LL rotation has to be applied to rebalance the nodes.

2. RR rotation

When node '4' gets attached as right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in Fig. 5.12.9

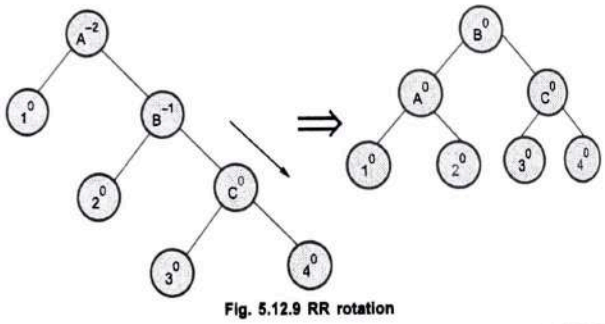


Fig. 5.12.9 RR rotation

3. LR rotation

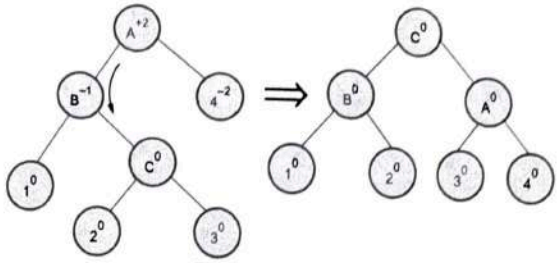


Fig. 5.12.10 LR rotation

When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

4. RL rotation

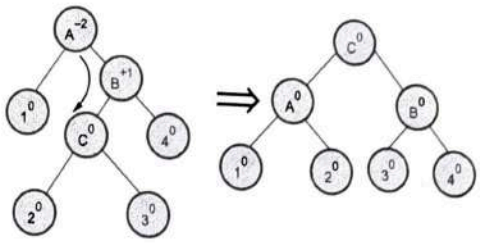


Fig. 5.12.11 RL rotation

When node '2' is attached as a left child of node 'C' then node 'A' gets unbalanced as its balance factor becomes – 2. Then RL rotation needs to be applied to rebalance the AVL tree.

Example 5.12.1 *Consider an AVL tree as given in Fig. 5.12.12. to insert 1, 25, 28, 12 in this tree.*

Solution :

Insert 1

To insert node '1' we have to attach it as a left child of '2'. This will unbalance the tree as follows. We will apply LL rotation to preserve AVL property of it.

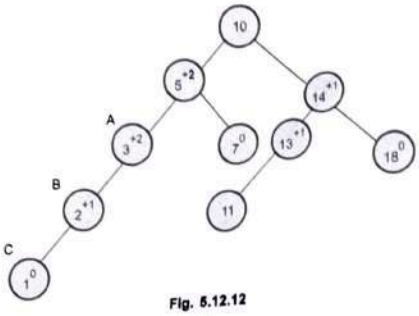


Fig. 5.12.12

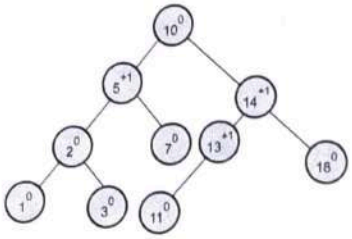


Fig. 5.12.13

Insert 25

We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property.

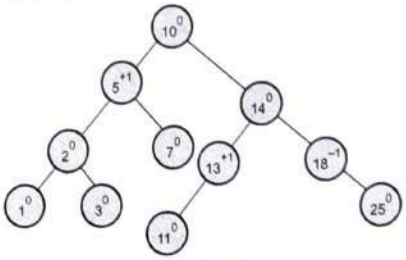


Fig. 5.12.14

Insert 28

The node '28' is attached as a right child of 25. RR rotation is required to rebalance.

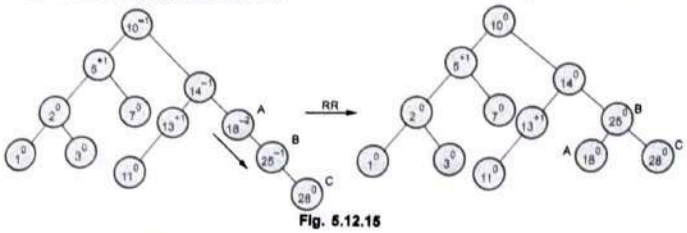


Fig. 5.12.15

Now we will insert 12.

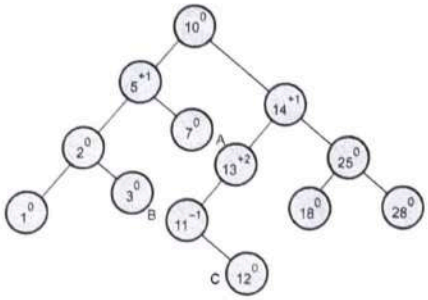


Fig. 5.12.16

To rebalance the tree we have to apply LR rotation

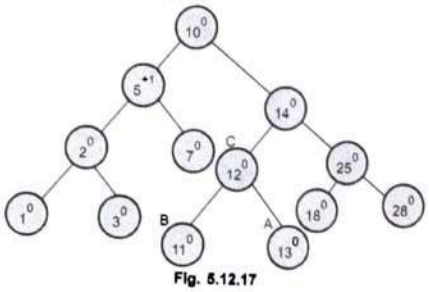


Fig. 5.12.17

Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

5.12.5 Deletion

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

Algorithm for deletion

The deletion algorithm is more complex than insertion algorithm.

1. Search the node which is to be deleted.
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.
   b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some subtree then balance that subtree using appropriate single or double rotations.

The deletion algorithm takes O(log n) time to delete any node.

Consider an AVL tree.

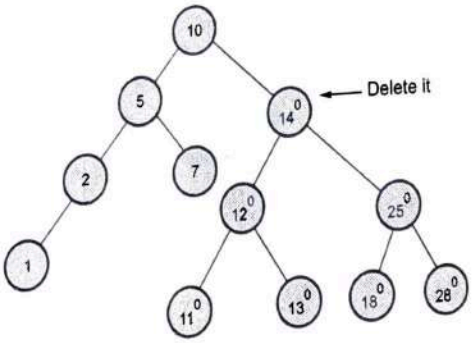


Fig. 5.12.18

The tree becomes

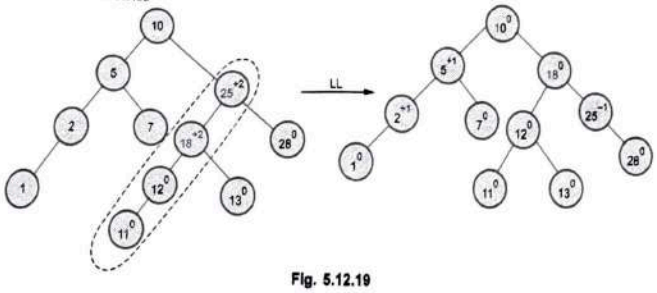


Fig. 5.12.19

Thus the node 14 gets deleted from AVL tree.

5.12.6 Searching

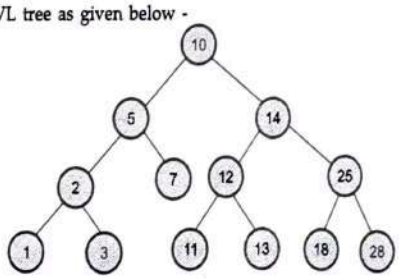
The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is used for searching a node from AVL tree.

Algorithm for searching

1. Consider the node to be searched as **'Key'** node.
2. Compare **key** with **root** node. If key < root then go to the left node and call this left node as current node. If key > root then go to the right node and call this right node as current node otherwise the desired node is the root node.
3. Compare **key** with **current** node. If key < current node then go to left node and call this left node as current node. If key > current node then go to right node and call this right node as current node. Otherwise if key = current node then desired node is present in an AVL tree. Thus repeat step 3 until the desired node is found or visiting the entire tree is not over.

For example :

Consider an AVL tree as given below -



Now if we want to search for node 18 then we will follow the following steps -

1. Compare 18 with root node 10. As 18 > 10 we will move on right sub-branch.
2. Compare 18 with 14. As 18 > 14, we will move on right sub-branch.
3. Compare 18 with 25. As 18 < 25, we will move on left sub-branch.
4. Compare 18 with 18. As the match is found, we will declare that "the desired node is present in an AVL tree."

The searching of a node from AVL tree takes **O(logn)** time.

5.12.7 AVL Tree Implementation

C++ Program

```
#include<iostream>
#define FALSE 0
#define TRUE 1
using namespace std;

//Tree node

    typedef struct Node
        {
                int data;
                int BF;
                struct Node *left;
                struct Node *right;
        }node;


class AVL
{
    node *root;
    public:
        AVL()
        {
        root=NULL;
        }
        node *insert(int data,int *current)
        {
            root=create(root,data,current);
            return root;
        }
        node *create(node *root,int data,int *current);
        node *remove(node *root,int data,int *current);
        node *find_succ(node *temp,node *root,int *current);
```



```
        node *right_rotation(node *root,int *current);
        node *left_rotation(node *root,int *current);
        void display(node *root);
};
node *AVL::create(struct Node *root,int data,int *current)
{
node *temp1,*temp2;
if(root==NULL)//initial node
{
    root= new node;
    root->data=data;
    root->left=NULL;
    root->right=NULL;
    root->BF=0;
    *current=TRUE;
    return(root);
}
if(data<root->data)
{
    root->left=create(root->left,data,current);
    // adjusting left subtree
    if(*current)
    {
    switch(root->BF)
    {
        case 1:temp1=root->left;
                if(temp1->BF==1)
                {
                    cout<<"\n single rotation: LL rotation";
                    root->left=temp1->right;
                    temp1->right=root;
                    root->BF=0;
                    root=temp1;
                }
                else
                {
                    cout<<"\n Double roation:LR rotation";
                    temp2=temp1->right;
                    temp1->right=temp2->left;
                    temp2->left=temp1;
                    root->left=temp2->right;
                    temp2->right=root;
                    if(temp2->BF==1)
                     root->BF=-1;
                    else
                     root->BF=0;
                    if(temp2->BF==-1)
```

```
                    temp1->BF=1;
                    else
                    temp1->BF=0;
                    root=temp2;
                }
                root->BF=0;
                *current=FALSE;
                break;
        case 0:
                root->BF=1;
                break;
        case -1:
                root->BF=0;
                *current=FALSE;
    }
}
}
if(data> root->data)
{
root->right=create(root->right,data,current);
//adjusting the right subtree
if(*current!=NULL)
{
    switch(root->BF)
    {
     case 1:
                root->BF=0;
                *current=FALSE;
                break;
     case 0:
                root->BF=-1;
                break;
     case -1:
                temp1=root->right;
                if(temp1->BF==-1)
                {
                    cout<<"\n single rotation:RR rotation";
                    root->right=temp1->left;
                    temp1->left=root;
                    root->BF=0;
                    root=temp1;
                }
                else
                {
                    cout<<"\n Double rotation:RL rotation";
                    temp2=temp1->left;
                    temp1->left=temp2->right;
```



```
                    temp2->right=temp1;
                    root->right=temp2->left;
                    temp2->left=root;
                    if(temp2->BF==-1)
                     root->BF=1;
                    else
                     root->BF=0;
                    if(temp2->BF==1)
                     temp1->BF=-1;
                    else
                     temp1->BF=0;
                    root=temp2;
                }
                root->BF=0;
                *current=FALSE;
            }
        }
    }
    return(root);
}
/*

    Display of Tree in inorder fashion
*/
void AVL::display(node *root)
{
if(root!=NULL)
{
    display(root->left);
    cout<<root->data<<"  ";
    display(root->right);
}
}
/*

 Deletion of desired node the tree

*/
node *AVL::remove(node *root,int data,int *current)
{
node *temp;
if(root->data==13)
    cout<<root->data;
if(root==NULL)
{
    cout<<"\n No such data";
    return (root);
```

```
}
else
{
    if(data<root->data)
    {
     root->left=remove(root->left,data,current);
     if(*current)
          root=right_rotation(root,current);
    }
    else
    {
     if(data>root->data)
     {
          root->right=remove(root->right,data,current);
          if(*current)
           root=left_rotation(root,current);
     }
     else
     {
          temp=root;
          if(temp->right==NULL)
          {
           root=temp->left;
           *current=TRUE;
           delete(temp);
          }
          else
          {
           if(temp->left==NULL)
           {
                root=temp->right;
                *current=TRUE;
                delete(temp);
           }
           else
           {
                temp->right=find_succ(temp->right,temp,current);
                if(*current)
                 root=left_rotation(root,current);
           }
          }
     }
    }
}
return (root);
}
node *AVL::find_succ(node *succ,node *temp,int *current)
```



```
{
 node *temp1=succ;
 if(succ->left!=NULL)
 {
      succ->left=find_succ(succ->left,temp,current);
      if(*current)
       succ=right_rotation(succ,current);
 }
 else
 {
      temp1=succ;
      temp->data=succ->data;
      succ=succ->right;
      delete temp1;
      *current=TRUE;
 }
 return (succ);
}
node *AVL::right_rotation(node *root,int *current)
{
 node *temp1,*temp2;
 switch(root->BF)
 {
      case 1:
                    root->BF=0;
                    break;
      case 0:
                    root->BF=-1;
                    *current=FALSE;
                    break;
      case -1:
                    temp1=root->right;
                    if(temp1->BF<=0)
                    {
                         cout<<"\n single rotation: RR rotation";
                         root->right=temp1->left;
                         temp1->left=root;

                         if(temp1->BF==0)
                         {
                          root->BF=-1;
                          temp1->BF=1;
                          *current=FALSE;
                         }
                         else
                         {
                          root->BF=temp1->BF=0;
```

```
                }
                root=temp1;
            }
            else
            {
            cout<<"\n Double Rotation:RL rotation";
            temp2=temp1->left;
            temp1->left=temp2->right;
            temp2->right=temp1;
            root->right=temp2->left;
            temp2->left=root;

            if(temp2->BF==-1)
                root->BF=1;
            else
                root->BF=0;
            if(temp2->BF==1)
                temp1->BF=-1;
            else
                temp1->BF=0;
            root=temp2;
            temp2->BF=0;
        }
    }
    return (root);
}
node* AVL::left_rotation(node *root,int *current)
{
 node *temp1,*temp2;
 switch(root->BF)
 {
    case -1:
                root->BF=0;
                break;

    case 0:
                root->BF=1;
                *current=FALSE;
                break;

    case 1:
                temp1=root->left;
                if(temp1->BF>=0)
                {
                 cout<<"\nsingle rotation LL rotation";
                 root->left=temp1->right;
                 temp1->right=root;
                 if(temp1->BF==0)
                 {
```



```
                    root->BF=1;
                    temp1->BF=-1;
                    *current=FALSE;
                }
                else
                {
                    root->BF=temp1->BF=0;
                }
                root=temp1;
                }
                else
                {
                cout<<"\nDouble rotation:LR rotation";
                temp2=temp1->right;
                temp1->right=temp2->left;
                temp2->left=temp1;
                root->left=temp2->right;
                temp2->right=root;

                if(temp2->BF==1)
                    root->BF=-1;
                else
                    root->BF=0;

                if(temp2->BF==-1)
                 temp1->BF=1;
                else
                 temp1->BF=0;
                root=temp2;
                temp2->BF=0;
            }
        }
 return root;
}

int main()
{
 AVL obj;
 node *root=NULL;
 int current;
  root=obj.insert(40,&current);
 root=obj.insert(50,&current);
 root=obj.insert(70,&current);
 cout<<endl;
 obj.display(root);
 cout<<endl;
 root=obj.insert(30,&current);
```

```
cout<<endl;
obj.display(root);
 root=obj.insert(20,&current);
cout<<endl;
obj.display(root);
root=obj.insert(45,&current);
cout<<endl;
obj.display(root);
 root=obj.insert(25,&current);
cout<<endl;
obj.display(root);
root=obj.insert(10,&current);
cout<<endl;
obj.display(root);
 root=obj.insert(5,&current);
cout<<endl;
obj.display(root);
root=obj.insert(22,&current);
cout<<endl;
obj.display(root);
root=obj.insert(1,&current);
cout<<endl;
obj.display(root);
 root=obj.insert(35,&current);
 cout<<"\n\nFinal AVL tree is: \n";
obj.display(root);
root=obj.remove(root,20,&current);
root=obj.remove(root,45,&current);
cout<<"\n\n AVL tree after deletion of a node: \n";
obj.display(root);
cout<<"\n";
return 0;
}
```

Output

```
 single rotation:RR rotation
40  50  70

30  40  50  70
 single rotation: LL rotation
20  30  40  50  70
 Double rotation:LR rotation
20  30  40  45  50  70
 Double rotation:LR rotation
20  25  30  40  45  50  70
10  20  25  30  40  45  50  70
 single rotation: LL rotation
```



```
5  10  20  25  30  40  45  50  70
Double rotation:LR rotation
5  10  20  22  25  30  40  45  50  70
single rotation: LL rotation
1  5  10  20  22  25  30  40  45  50  70
Double rotation:LR rotation

Final AVL tree is:
1  5  10  20  22  25  30  35  40  45  50  70
single rotation LL rotation

AVL tree after deletion of a node:
1  5  10  22  25  30  35  40  50  70
```

5.12.8 Comparison between BST, OBST and AVL Tree

Binary Search Tree : This is binary tree in which nodes can be inserted or deleted based on their values. In binary search tree left node is less than parent node. The right node is greater than parent node. Binary search tree can be implemented using arrays or linked data structures.

Optimal Binary Search Tree (OBST)

The optimal binary search tree is basically a binary search tree with optimum cost. There is a fixed set of identifiers from which OBST is built. The static symbol table is used to implement OBST.

Height Balanced Search Tree

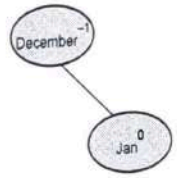
The height balanced tree is basically a binary search tree which is constructed using balance factor. Its implementation is a dynamic implementation.

Example 5.12.2 *Obtain height balanced tree for the following sequence of data : December, Jan., April, March, July, Aug., Oct., Nov., May, June.* **SPPU : Dec.-07, May-10, Marks 8**

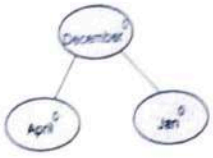
Solution :

Step 1 : Insert December

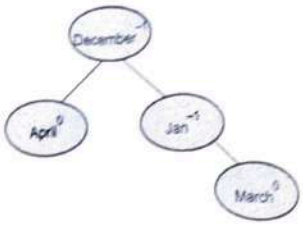
Step 2 : Insert Jan.



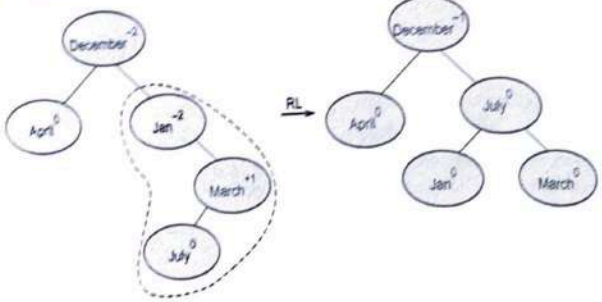
Step 3 : Insert April



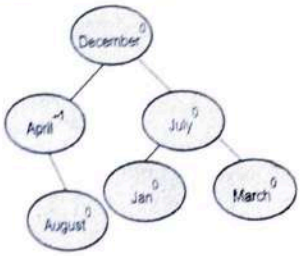
Step 4 : Insert March



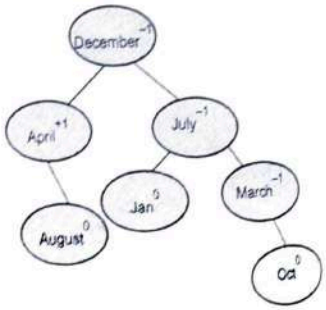
Step 5 : Insert July



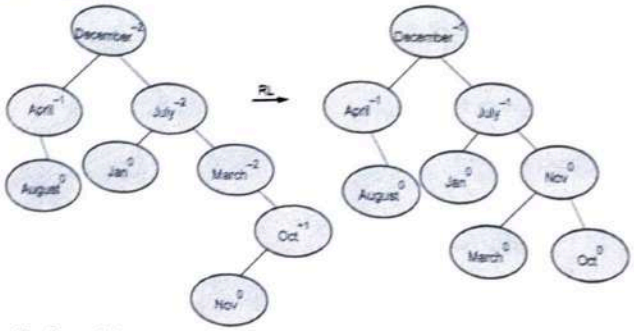
Step 6 : Insert August



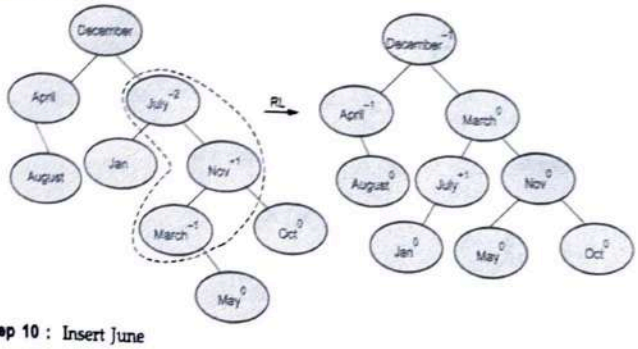
Step 7 : Insert Oct.



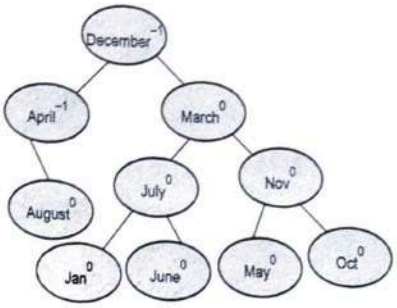
Step 8 : Insert Nov.



Step 9 : Insert May



Step 10 : Insert June



is a final AVL tree.

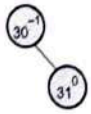
Example 5.12.3 *Define AVL tree. Obtain an AVL tree by accepting the following numbers (keys) one at a time. Show balance factor of each node, the type of rotation and the transformation required for each insertion in AVL tree.*
30, 31, 32, 23, 22, 28, 24, 29, 26, 27, 34, 36. **SPPU : Dec.-08, May-10, Marks 10**

Solution : An AVL tree is a height balanced tree in which every node has a balance factor which is 0, –1 or +1.

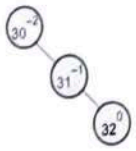
Example

Step 1 : Insert 30

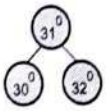
Step 2 : Insert 31



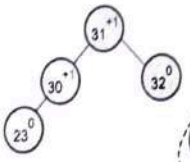
Step 3 : Insert 32



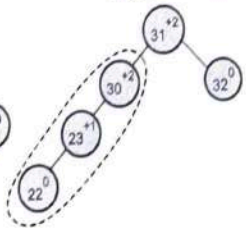
RR →



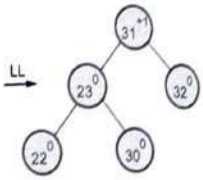
Step 4 : Insert 23



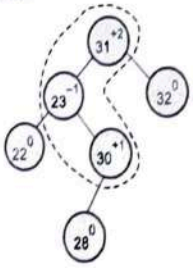
Step 5 : Insert 22



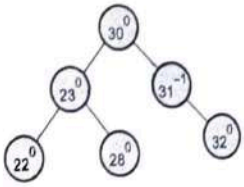
LL →



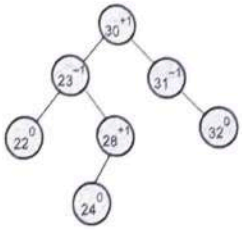
Step 6 : Insert 28



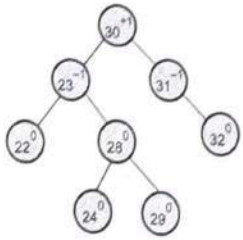
LR →



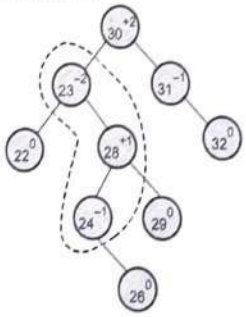
Step 7 : Insert 24



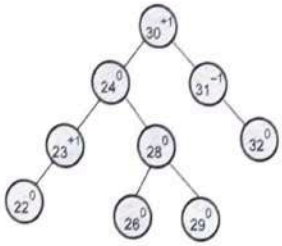
Step 8 : Insert 29



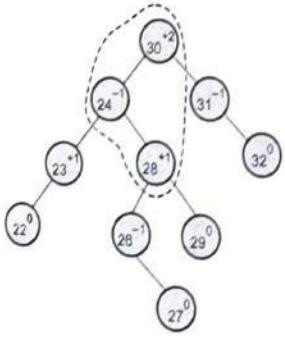
Step 9 : Insert 26



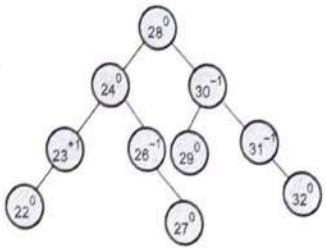
RL →



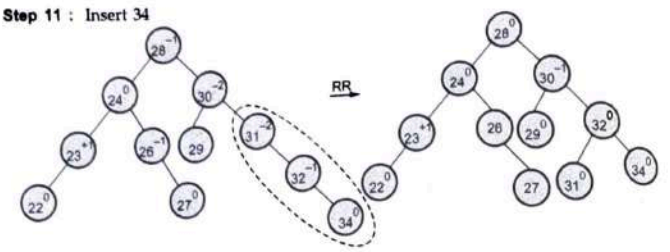
Step 10 : Insert 27



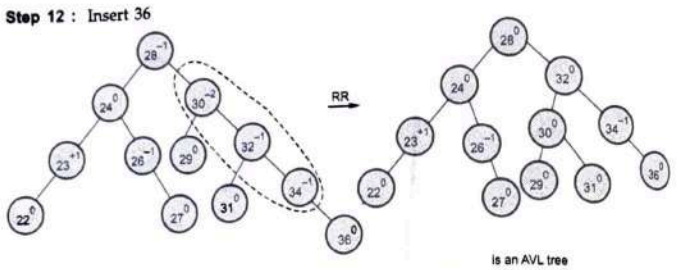
LR →



Step 11 : Insert 34



Step 12 : Insert 36



Example 5.12.4 *What is the size of the smallest AVL tree with height 8 ?*

SPPU : Dec.-05, Marks 4

Solution : If h is the height of an AVL tree then minimum number of nodes can be obtained for such AVL tree using the following formula.

$$N(h) = 1 + N(h-1) + N(h-2) \quad ...(1)$$

If there is only one node then h = 0, for 2 nodes the h = 1. We can formulate and say

$$N(0) = 1, \quad N(1) = 2 \quad ...(2)$$

Hence using equation (1) we can obtain minimum number of nodes for height 8.

$$N(8) = 1 + N(7) + N(6) \quad ...(3)$$

$$N(7) = 1 + N(6) + N(5) \quad ...(4)$$

$$N(6) = 1 + N(5) + N(4) \quad ...(5)$$

$$N(5) = 1 + N(4) + N(3) \quad ...(6)$$

$$N(4) = 1 + N(3) + N(2) \quad ...(7)$$

$$N(3) = 1 + N(2) + N(1) \quad ...(8)$$

$$N(2) = 1 + N(1) + N(0)$$

$$\therefore \quad N(2) = 1 + 2 + 1 \quad ... \because \text{From equation (2)}$$

$$\mathbf{N(2) = 4}$$

Put N(2) in equation (8)

$$\therefore \quad N(3) = 1 + 4 + 2$$

$$\mathbf{N(3) = 7}$$

Put N(3) in equation (7)

$$\therefore \quad N(4) = 1 + 7 + 4$$

$$\mathbf{N(4) = 12}$$

Put N(4) in equation (6)

$$\therefore \quad N(5) = 1 + 12 + 7$$

$$\mathbf{N(5) = 20}$$

Put N(5) in equation (5)

$$\therefore \quad N(6) = 1 + 20 + 12$$

$$\mathbf{N(6) = 33}$$

Put N(6) in equation (4)

$$\therefore \quad N(7) = 1 + 33 + 20$$

$$\mathbf{N(7) = 54}$$

Put N(7) in equation (3)

$$\therefore \quad N(8) = 1 + 54 + 33$$

$$\mathbf{N(8) = 88}$$

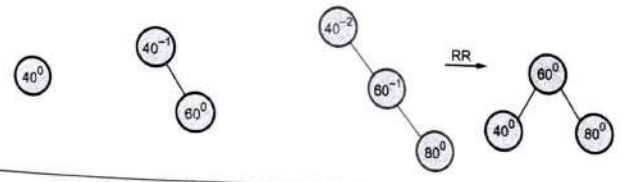
That means minimum number of nodes are 88 for an AVL tree of height 8.

Example 5.12.5 *Construct an AVL tree for given sequence of data. In each case show the balance factor of all nodes and name the type of rotation used for balancing the tree 40, 60, 80, 50, 45, 47, 44, 42, 75, 46, 41.*

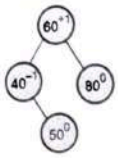
SPPU : May-11, Marks 10

Solution :

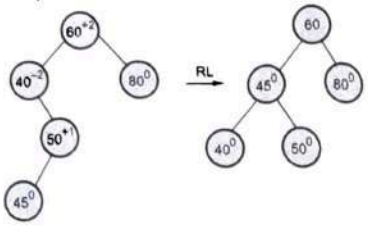
Step 1 : **Step 2 : Insert 60** **Step 3 : Insert 80**



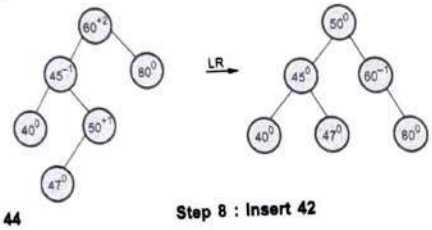
Step 4 : Insert 50



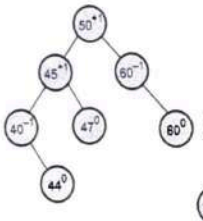
Step 5 : Insert 45



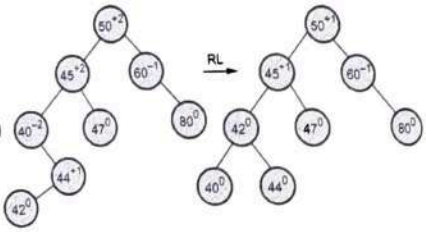
Step 6 : Insert 47



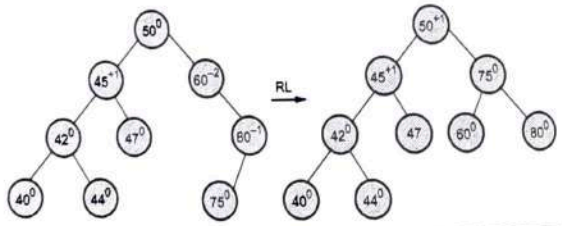
Step 7 : Insert 44



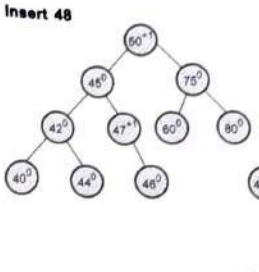
Step 8 : Insert 42



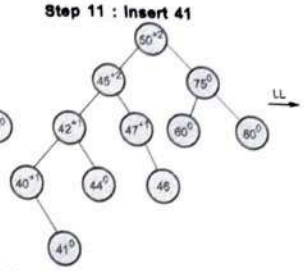
Step 9 : Insert 75

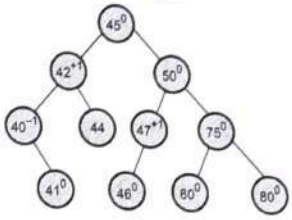


Step 10 : Insert 48



Step 11 : Insert 41





Final AVL Tree

Example 5.12.6 *Construct an AVL search tree by inserting the following elements in the order of their occurrence. Show the balance factor and type of rotation at each stage :*
MAR MAY NOV AUG APR JAN DEC JUL FEB JUN

SPPU : May-14, Marks 10

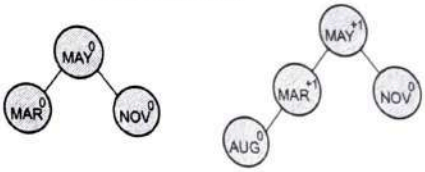
Solution :

Step 1 : Insert MAR

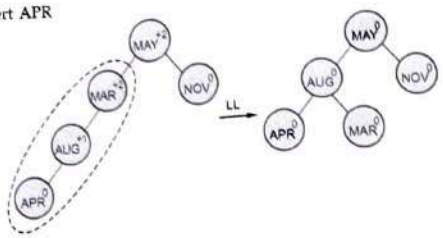
Step 2 : Insert MAY

Step 3 : Insert NOV

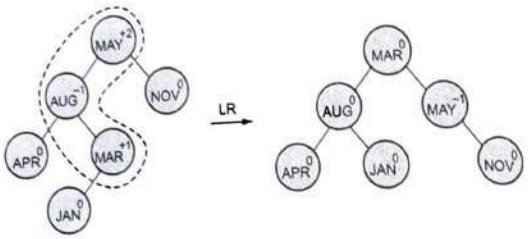
Step 4 : Insert AUG



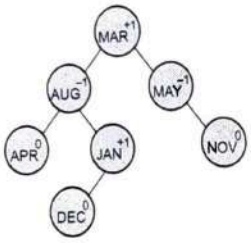
Step 5 : Insert APR



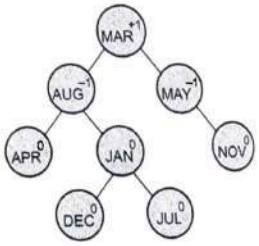
Step 6 : Insert JAN



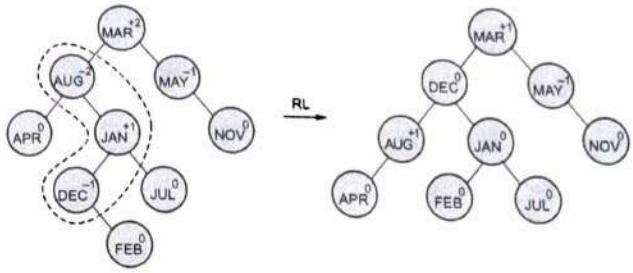
Step 7 : Insert DEC



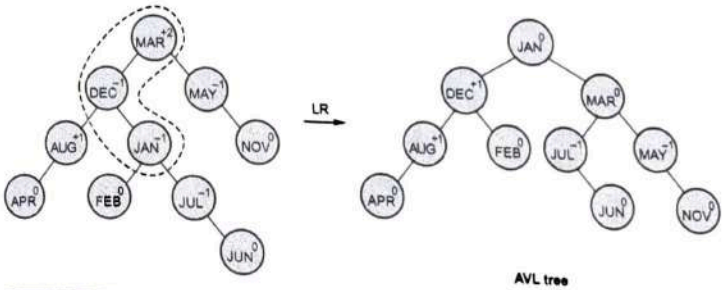
Step 8 : Insert JUL



Step 9 : Insert FEB



Step 10 : Insert JUN



AVL tree

Example 5.12.7 *Obtain an AVL tree by inserting one data element at a time in the following sequence :*

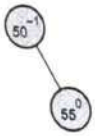
50, 55, 60, 15, 10, 40, 20, 45, 30, 70, 80

Label the rotations appropriately at each stage. **SPPU : Dec.-14, Marks 8**

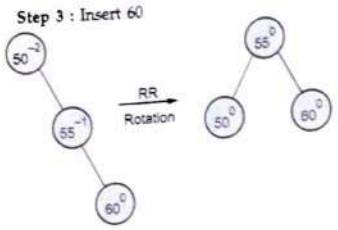
Solution :

Step 1 : Insert 50

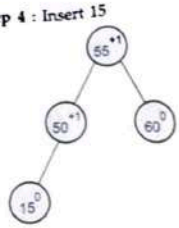
Step 2 : Insert 55



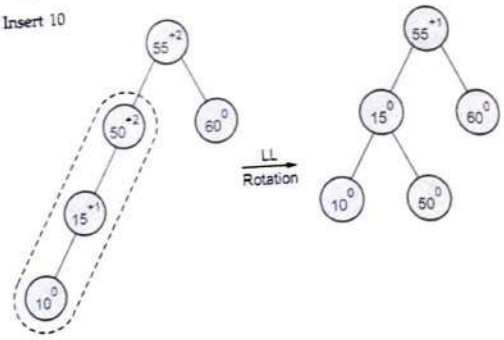
Step 3 : Insert 60



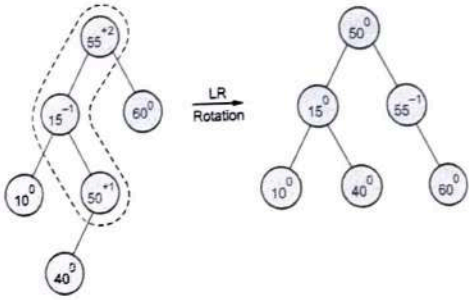
Step 4 : Insert 15



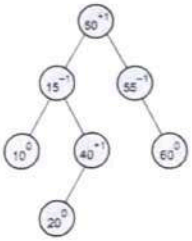
Step 5 : Insert 10



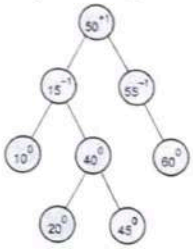
Step 6 : Insert 40



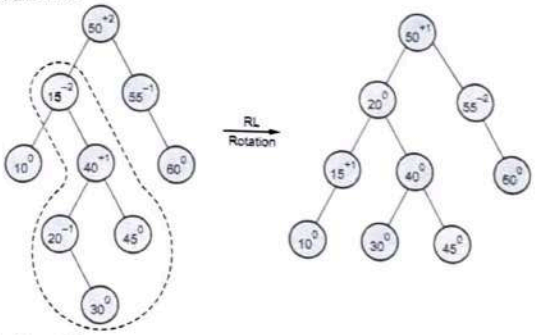
Step 7 : Insert 20



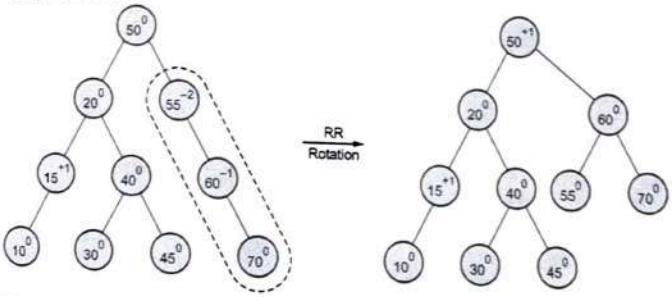
Step 8 : Insert 45



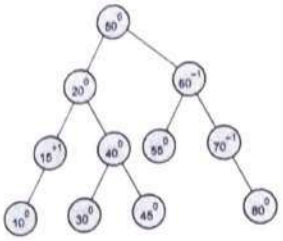
Step 9 : Insert 30



Step 10 : Insert 70



Step 11 : Insert 80



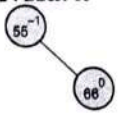
AVL tree

Example 5.12.8 *Construct an AVL search tree by inserting the following elements in the order of their occurrence. Show the balance factor and type of rotation at each stage :*
55, 66, 77, 15, 11, 33, 22, 35, 25, 44, 88, 99. **SPPU : May-15, Marks 6**

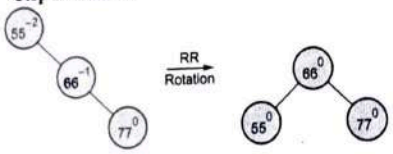
Solution :

Step 1 : Insert 55

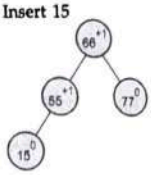
Step 2 : Insert 66



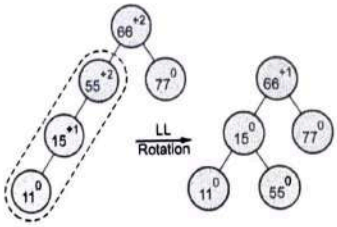
Step 3 : Insert 77



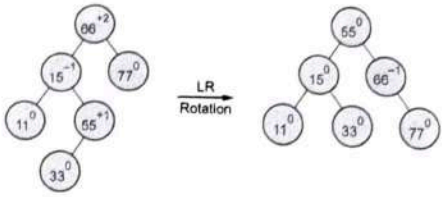
Step 4 : Insert 15



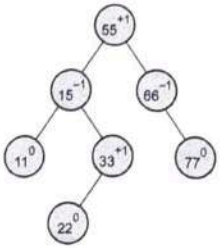
Step 5 : Insert 11



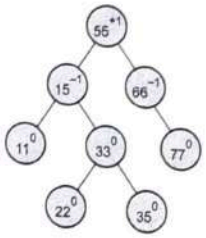
Step 6 : Insert 33



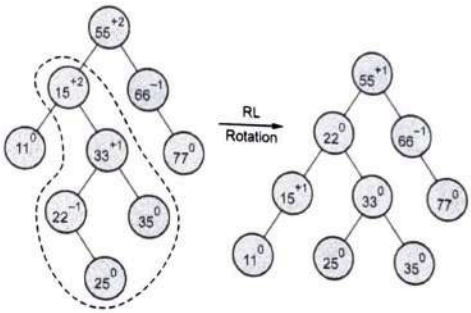
Step 7 : Insert 22



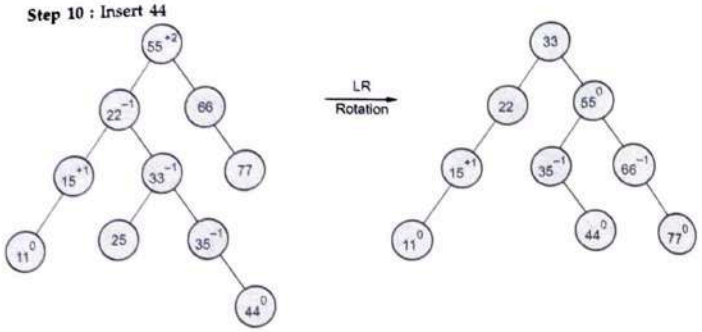
Step 8 : Insert 35



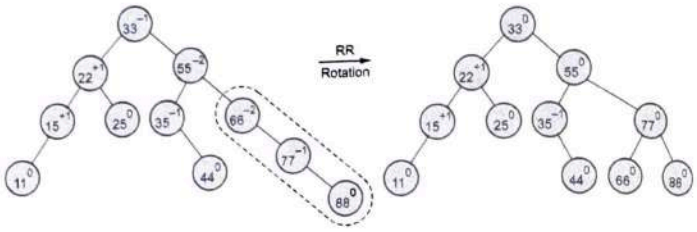
Step 9 : Insert 25



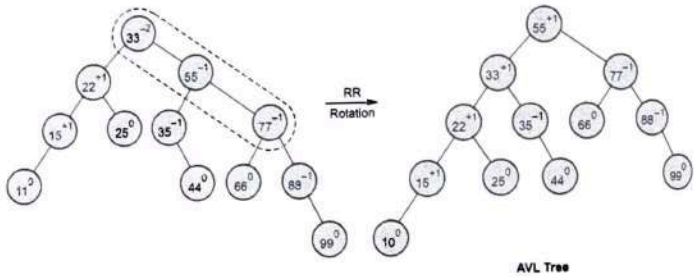
Step 10 : Insert 44



Step 11 : Insert 88

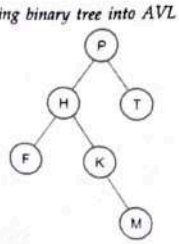


Step 12 : Insert 99



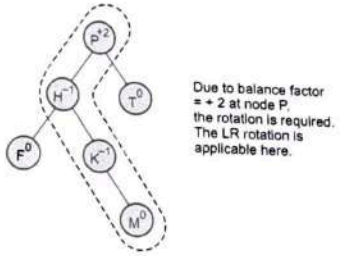
AVL Tree

Example 5.12.9 *Convert the following binary tree into AVL tree.*

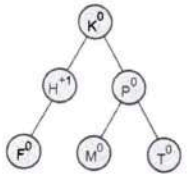


SPPU : May-16, Marks 4

Solution : Step 1 : We will add balance factor to each node.



Step 2 :



Balanced AVL tree

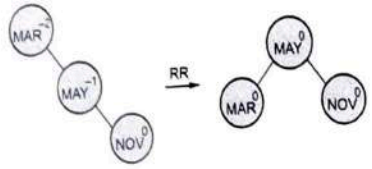
Example 5.12.10 *Construct an AVL for the following data : MAR, MAY, NOV, AUG, APR, JAN, DEC, JUN, FEB, JUL, OCT, SEP. Show the balance factor of each node and type of rotation.*

SPPU : May-18, Marks 10

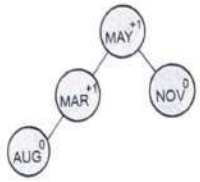
Solution : Step 1 : Insert MAR

Step 2 : Insert MAY

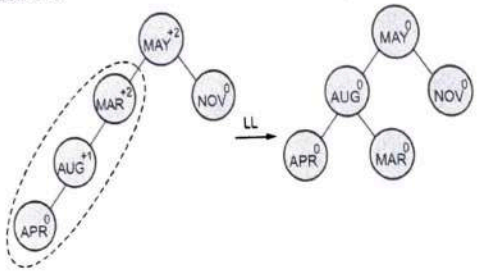
Step 3 : Insert NOV



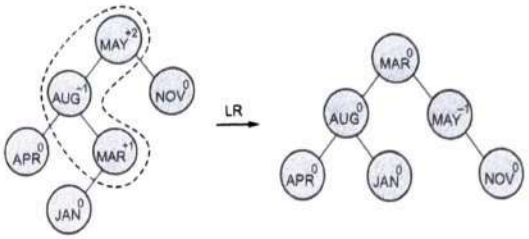
Step 4 : Insert AUG



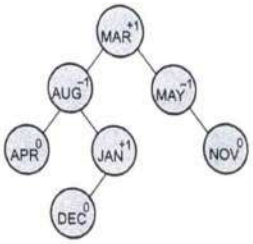
Step 5 : Insert APR



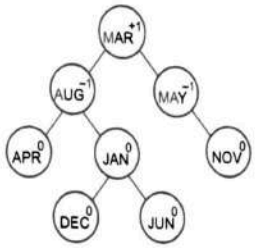
Step 6 : Insert JAN



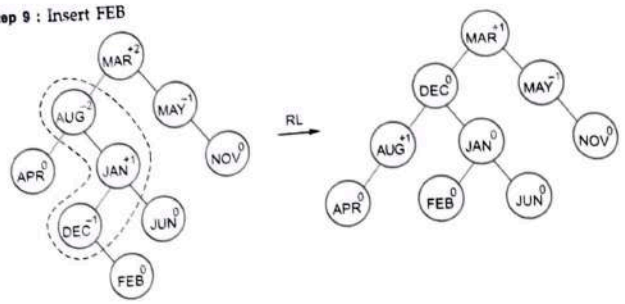
Step 7 : Insert DEC



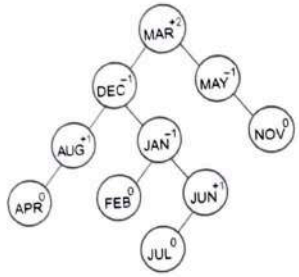
Step 8 : Insert JUN



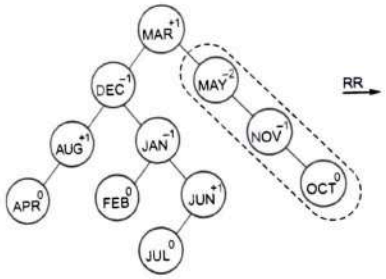
Step 9 : Insert FEB

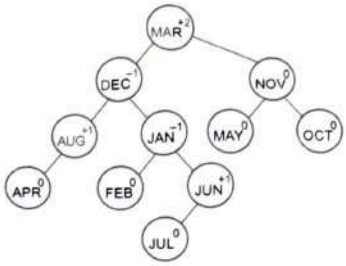


Step 10 : Insert JUL

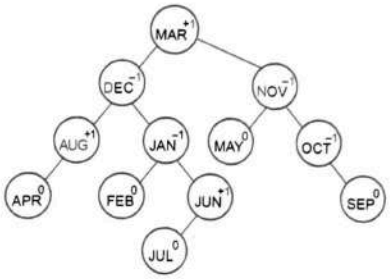


Step 11 : Insert OCT





Step 12 : Insert SEP



This is required AVL tree

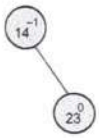
This is required AVL Tree.

Example 5.12.11 *Create an AVL tree using the following data, show the balance factor :*
14, 23, 7, 10, 33, 56, 80, 66, 70

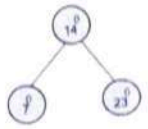
SPPU : May-19, Marks 8

Solution : Step 1 : Insert 14

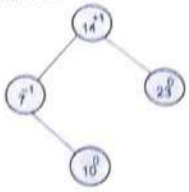
Step 2 : Insert 23



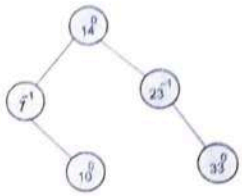
Step 3 : Insert 7



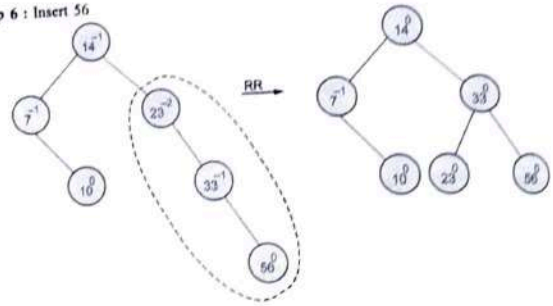
Step 4 : Insert 10



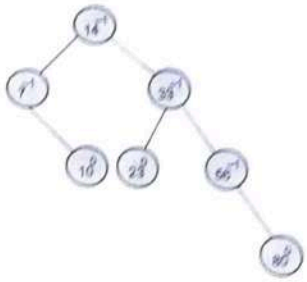
Step 5 : Insert 33



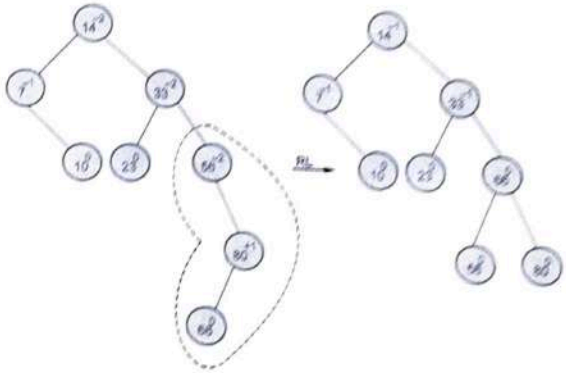
Step 6 : Insert 56



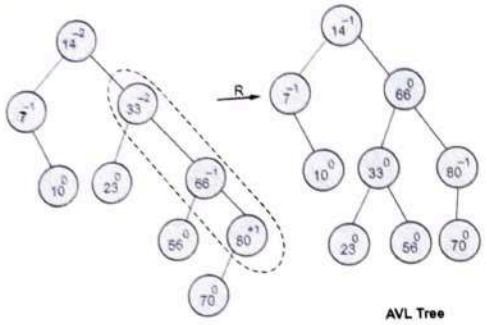
Step 7 : Insert 80



Step 8 : Insert 66

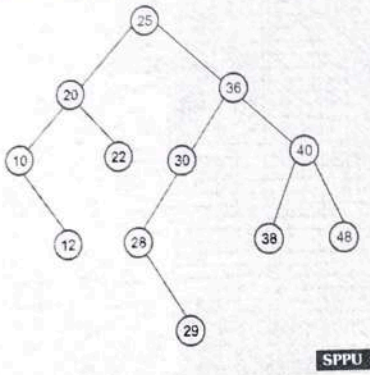


Step 9 : Insert 70



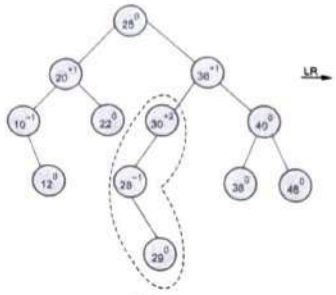
AVL Tree

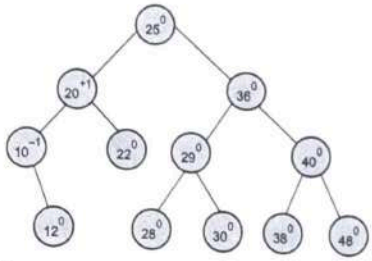
Example 5.12.12 *For a given tree, identify whether it is AVL tree or not ? If it is not an AVL tree, convert it into balanced AVL tree. After conversion, insert node 15 and 24 in the tree. Delete node 20 and 22 from the tree. After insertion and deletion operation, if the tree is imbalanced, make it balanced AVL tree.*



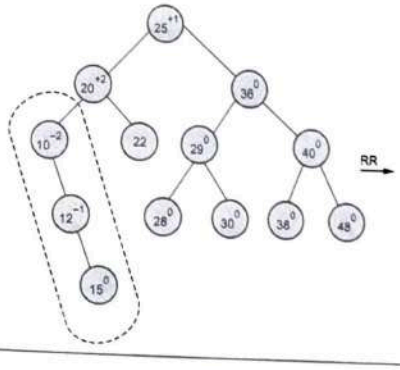
SPPU : Dec.-19, Marks 10

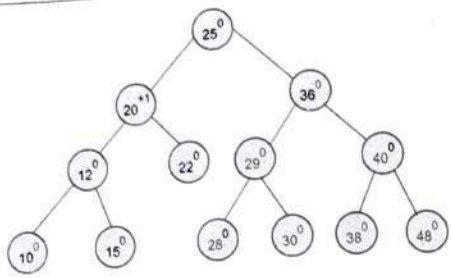
Solution : The given tree is not AVL tree,



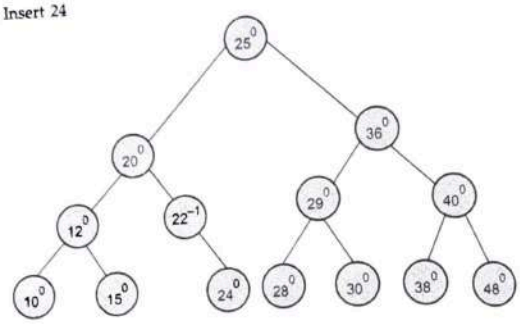


Step 1 : Insert 15

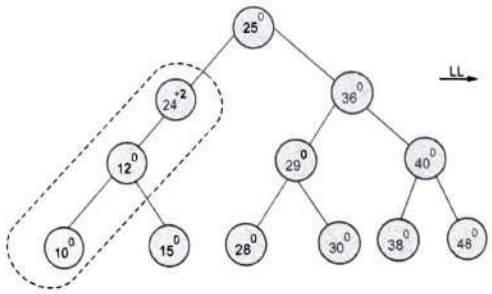




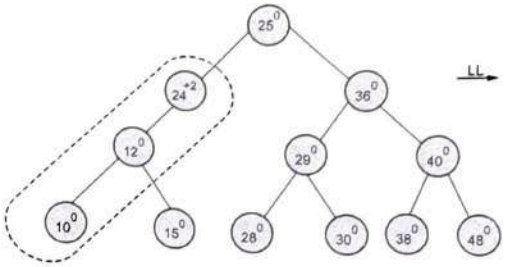
Step 2 : Insert 24

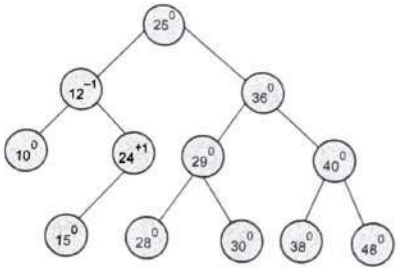


Step 3 : Delete 20



Step 4 : Delete 22





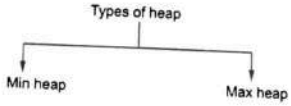
5.13 Heap Data Structure

SPPU : Dec.-15, May-16,19, Marks 8

Definition : Heap is a **complete binary tree** or **almost complete binary tree** in which **every parent node** is either greater or lesser than its child nodes.

5.13.1 Min and Max Heap

Heap can be **min heap** or **max heap.**



A **Max heap** is a tree in which value of each node is greater than or equal to the value of its children nodes.

For example :

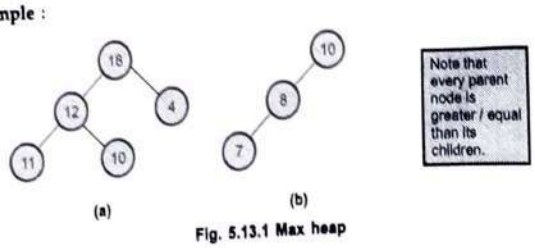


Fig. 5.13.1 Max heap

A **Min heap** is a tree in which value of each node is less than or equal to value of its children nodes.

For example :

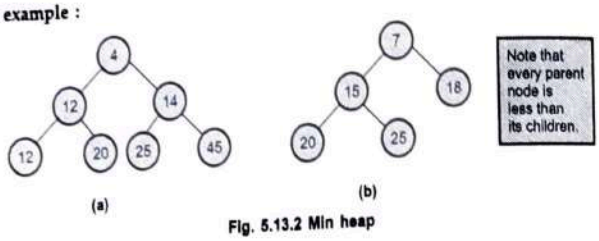
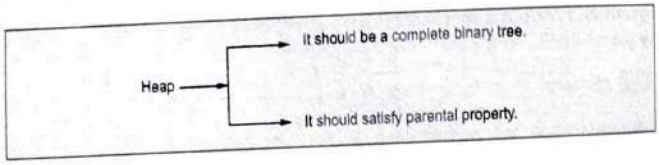


Fig. 5.13.2 Min heap

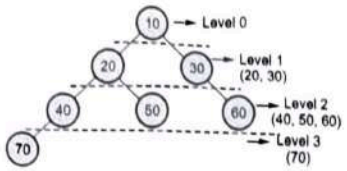
Parent being greater or lesser in heap is called **parental property**. Thus heap has two important properties.



Before understanding the construction of heap let us learn/revise few basics that are required while constructing heap.

- **Level of binary tree :** The root of the tree is always at level 0. Any node is always at a level one more than its parent nodes level.

For example :



Height of the tree : The maximum level is the height of the tree. The height of the tree is also called **depth** of the tree.

For example :

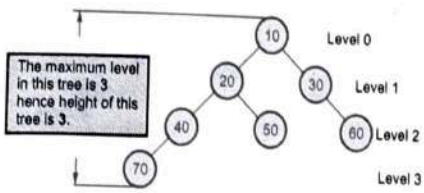


Fig. 5.13.3 Height of tree

- **Complete binary tree :** The complete binary tree is a binary tree in which all leaves are at the same depth or total number of nodes at each level i are 2^i.

For example :

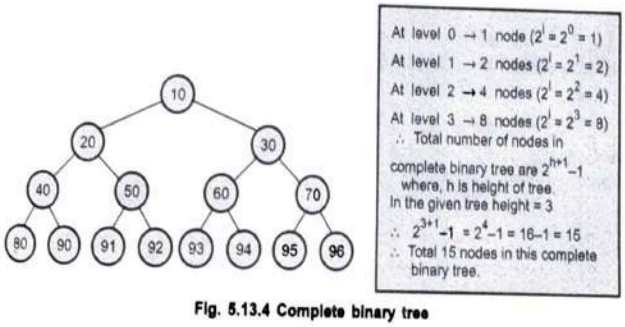


Fig. 5.13.4 Complete binary tree

- **Almost complete binary tree** : The almost complete binary tree is a tree in which -
  i) Each node has a left child whenever it has a right child. That means there is always a left child, but for a left child there may not be a right child.
  ii) The leaf in a tree must be present at height h or h–1. That means all the leaves are on two adjacent levels.

For example :

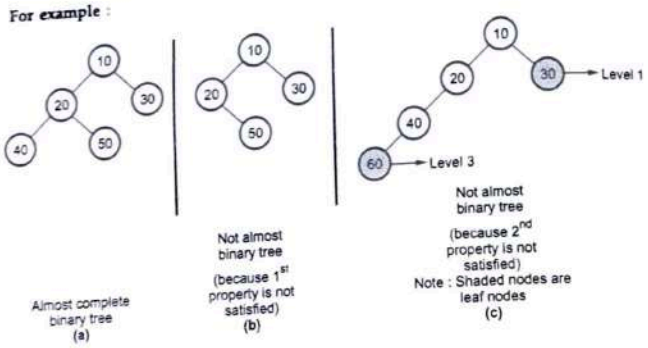


Fig. 5.13.5

In Fig. 5.13.5 (a) the given binary tree is satisfying both the properties. That is all the leaves are at either level 2 or level 1 and when right child is present left child is always present.

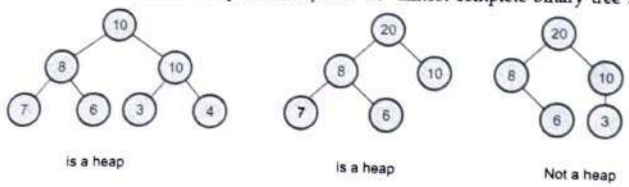
In Fig. 5.13.5 (b), the given binary tree is not satisfying the first property; that is the right child is present for node 20 but it has no left child. Hence it is not almost complete binary tree.

In Fig. 5.13.5 (c), the given binary tree is not satisfying the second property; that is the leaves are at level 3 and level 1 and not at adjacent levels. Hence it is not almost complete binary tree.

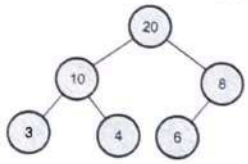
5.13.2 Heap using Priority Queues

There are some important properties that should be followed by heap.

1. There should be either "complete binary tree" or "almost complete binary tree".



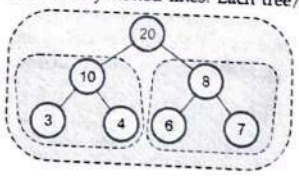
2. The root of a heap always contains its largest element.



This is heap because 20 is largest among all node values.

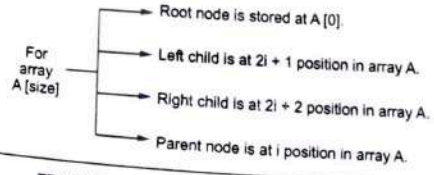
3. Each subtree in a heap is also a heap.

The tree and subtrees are shown by dotted lines. Each tree/subtree itself is a heap.

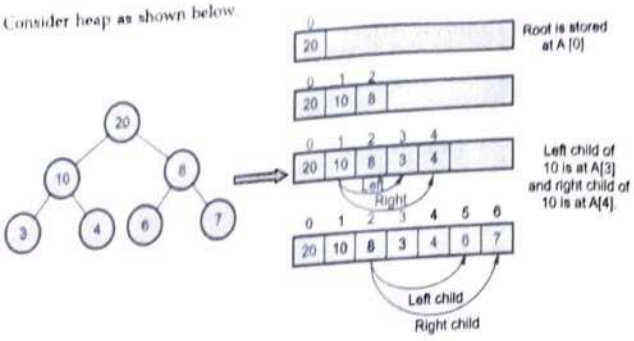


4. Heap can be implemented as **array** by recording its elements in the top-down and left to right fashion.

For example :

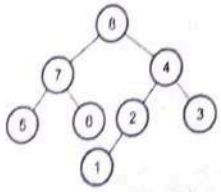


Consider heap as shown below



Thus array can represent complete heap.

Example 5.13.1 *For a given heap, give the array representation of it.*



Solution : Root node is stored at 0th index. For each parent node i we will store left node at (2i + 1) and right node at (2i + 2) positions in array.

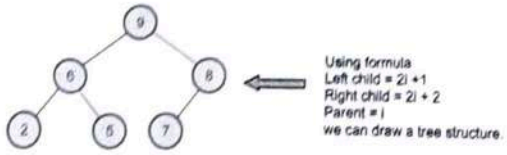
The array will then be -

0	1	2	3	4	5	6	7	8	9	10	11
8	7	4	5	6	2	3					1

Example 5.13.2 *Draw a tree for given heap represented by an array.*

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

Solution :



5.13.3 Heap as ADT

AbstractDataType Heap

{

Instances : Heap is a complete binary tree or almost complete binary tree in which every parent node is either greater or lesser than its child nodes.

Operations :

1. **Create-tree :** This operation reads the element one by one and creates the complete binary tree. In this tree total number of nodes at each level i are 2^i.
2. **ParentDominance ( ) :** This function is for arranging each node in the tree in such a manner that each parent node is greater than its child nodes.

}

5.13.3.1 Insertion of Element in the Heap

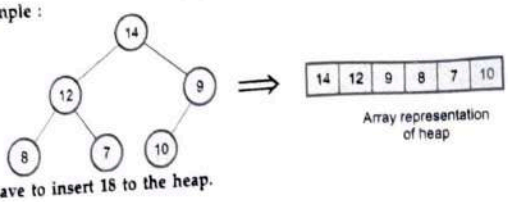
The algorithm for inserting an element in the heap can be as given below -

Algorithm for insertion of element :

1. Insert element at the last position in heap.
2. Compare with its parent, swap the two nodes if parental dominance condition gets violeted.

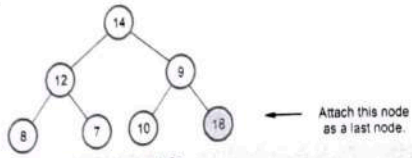
   [Parental dominance = Parent is greater than its children]
3. Continue comparing the new element with nodes up, each time by satisfying parental dominance condition.

For example :



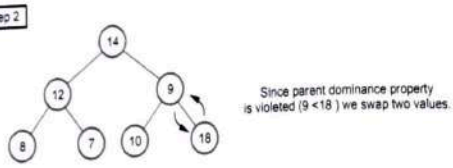
If we have to insert 18 to the heap.

Step 1

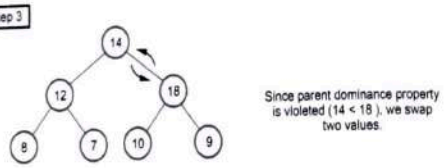


Key Point *We always fill heap from left to right.*

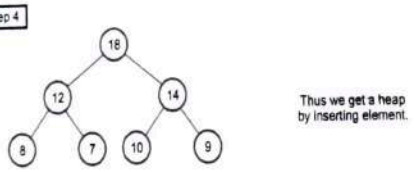
Step 2



Step 3



Step 4

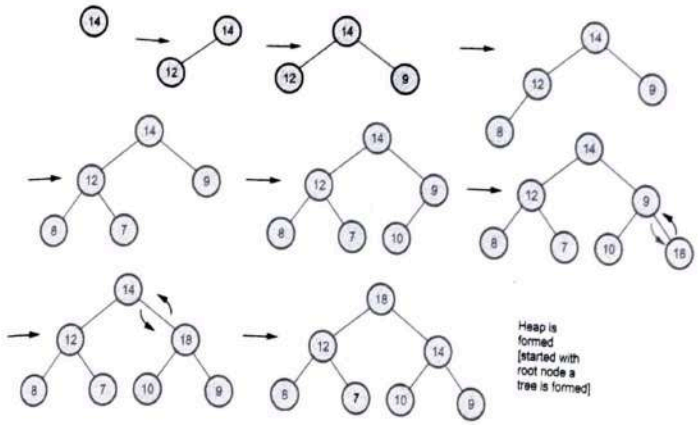


In top-down approach the heap is constructed from top to bottom scanning of a tree, each time checking parent dominace property.

Example 5.13.3 *Construct a heap using for following list of elements.*

14, 12, 9, 8, 7, 10, 18

Solution : We can construct heap using top-down approach with repeated insertion of element.



Pseudo Code

```
 void Heap::insert(int item)
{
 int temp;
//temp node starts at leaf and moves up
 temp=++size;
 while(temp!=1 && heap[temp/2]<item)
 {
// the element cannot be placed in array H
  H[temp]=H[temp/2];//moving element down
   temp=temp/2;//finding the parent
  }
  H[temp]=item;//placing the element at its position
  }
```

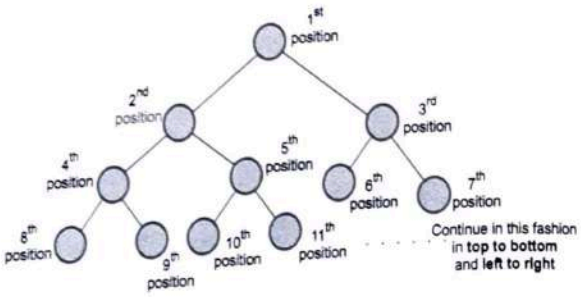
5.13.3.2 Deletion of Element From the Heap

Algorithm for deletion of element from heap :

The root of a heap can be deleted and the heap fixed up can be as follows :

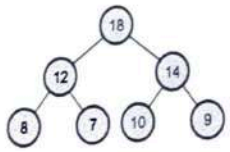
1. Exchange the root with the last leaf.
2. Decrease the heap's size by 1.
3. Heapify the smaller tree using bottom-up construction algorithm.

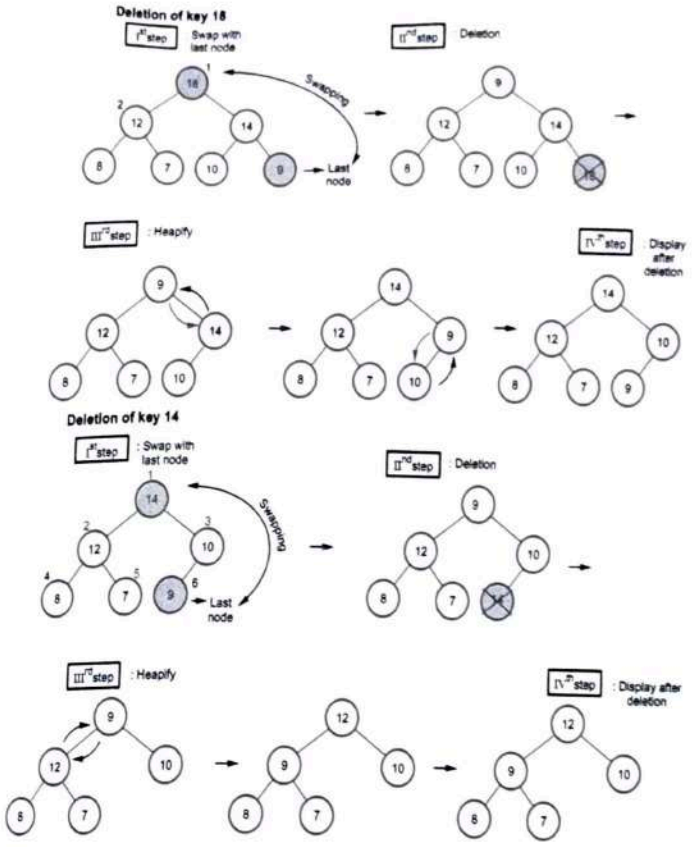
While constructing heap nodes are attached like this,



Understanding of positions in a tree or a heap is necessary for performing delete operation. Because the key to be deleted is always swapped with last node and the last node is always deleted from heap. Also each time we have to maintain heap property (i.e. parent dominance - parent being greater). Let us understand this delete operation with the help of some examples.

Perform deletion on following heap.





Continuing in this fashion we can perform deletion operation on given heap.

Key Point *Always root's key has to be deleted from a heap.*

Analysis

The basic operation in deletion algorithm is the key comparison that should be made to "heapify" the tree after the swap has been made. Each time, after deletion the size of

the tree is decreased by 1. It requires less number of key comparisons than twice the heap's height. Therefore time complexity of deletion of **O(log n)**.

Example 5.13.4 *Consider the heap in the following Fig. 4.5.6.*

i) Apply deleet operation to the heap. Repair heap after deletion.

ii) Insert 38 into the heap. Repair heap after insertion.

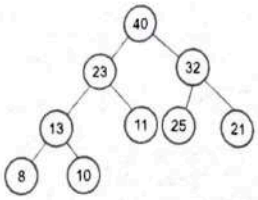
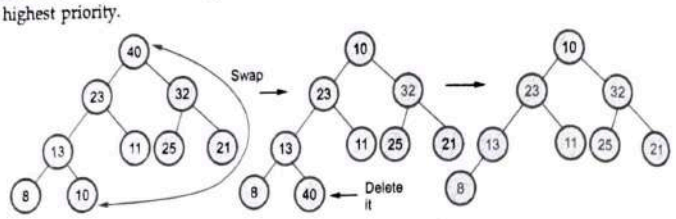
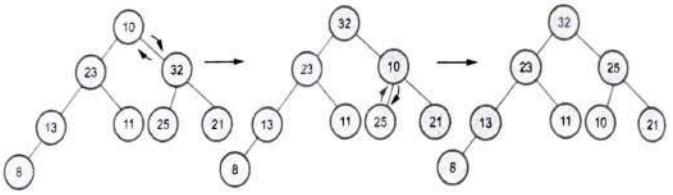


Fig. 5.13.6

Solution : **I)** For delete operation we will choose node 40 because this node is of highest priority.

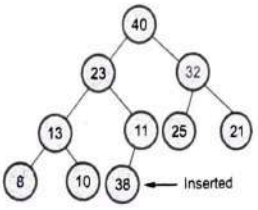


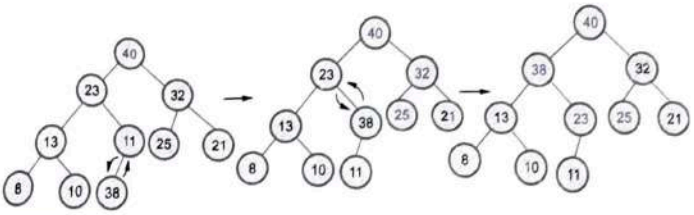
After deleting the node 40, we need to heapify the tree



II) Insert 38 : Consider the given heap.

After inserting 38, we need to heapify the tree. That means we have to maintain the heap property i.e. **parent node must be greater than the child nodes.**



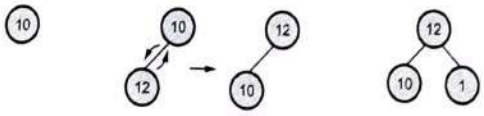


Example 5.13.5 *Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13 and 2 at a time, into initially empty binary heap. After creating such heap delete the element 8 from heap, how do you repair the heap? Then insert the element in the heap and show the final result (insertion should be at other than lead node).*

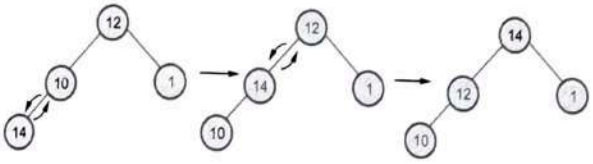
Solution : We will create max heap for the set

10 12 1 14 6 5 8 15 3 9 7 4 11 13 2

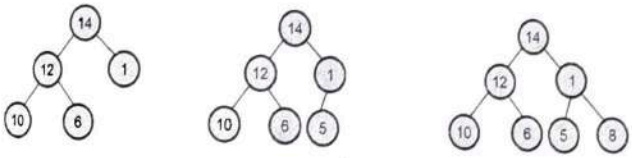
Step 1 : Insert 10. **Step 2 :** Insert 12. **Step 3 :** Insert 1.



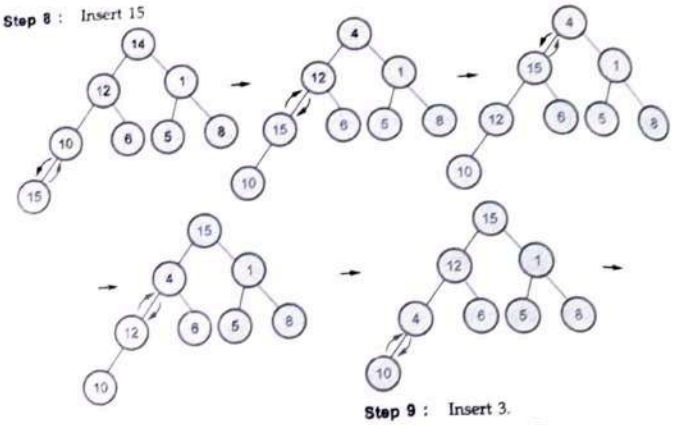
Step 4 : Insert 14.



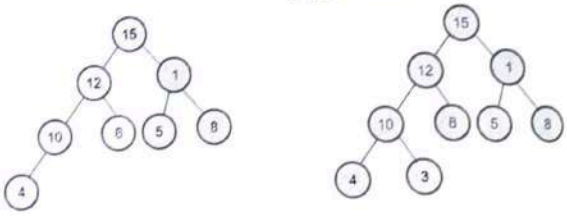
Step 5 : Insert 6. **Step 6 :** Insert 5. **Step 7 :** Insert 8.



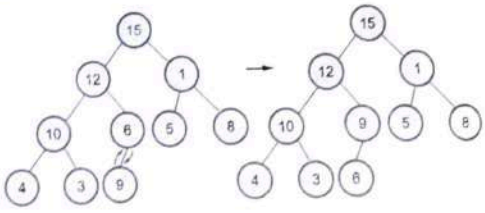
Step 8 : Insert 15



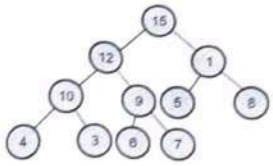
Step 9 : Insert 3.



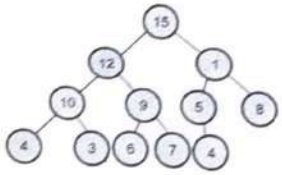
Step 10 : Insert 9.



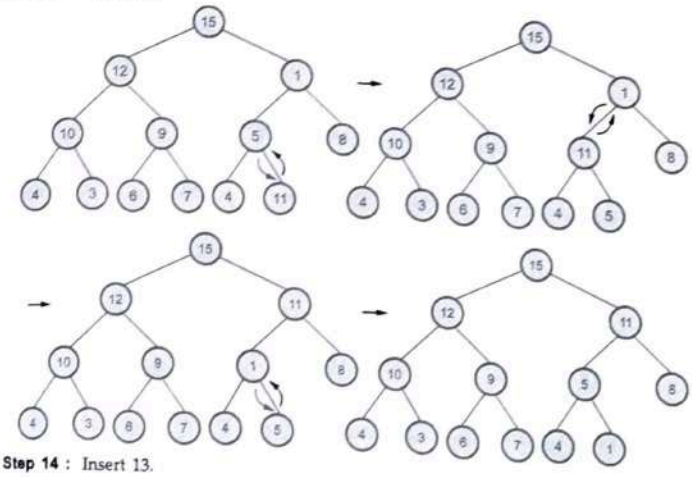
Step 11 : Insert 7.



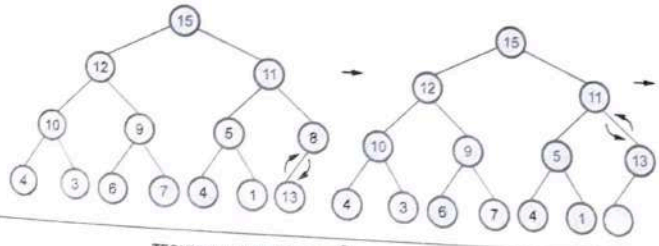
Step 12 : Insert 4.

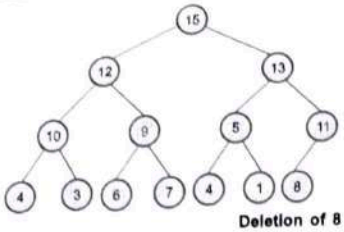


Step 13 : Insert 11.



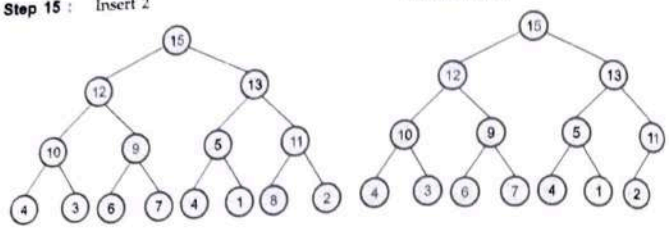
Step 14 : Insert 13.





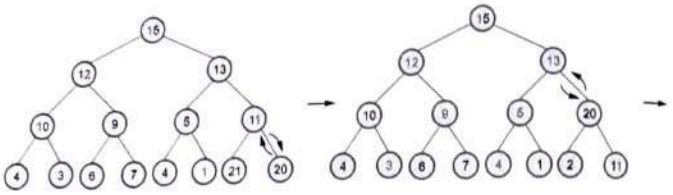
Step 15 : Insert 2

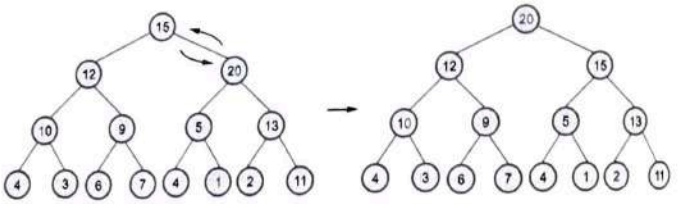
Deletion of 8



Insertion of node

If we insert node 20 then,

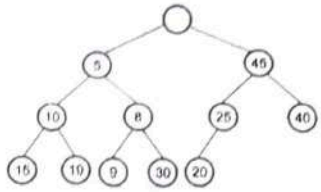




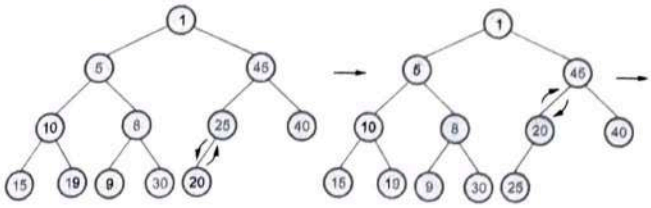
Example 5.13.6 *Consider the following D Heap :*

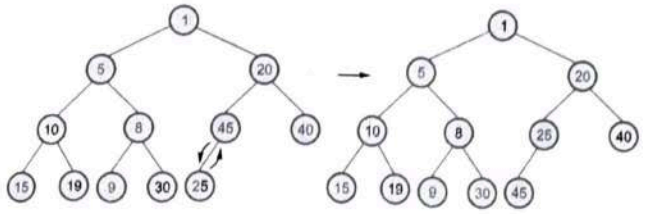
1) Insert 18 and repair heap after insertion.

2) Delete min element and repair heap after deletion

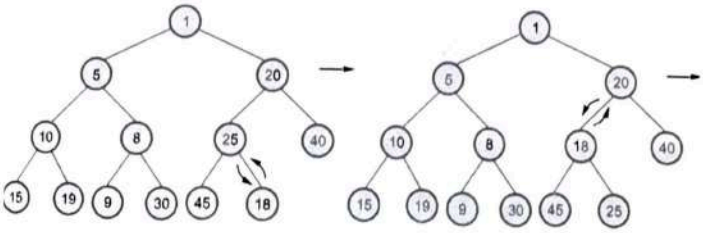


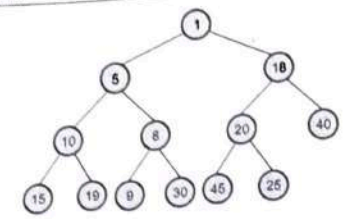
Solution : Assume the data at root node is 1.



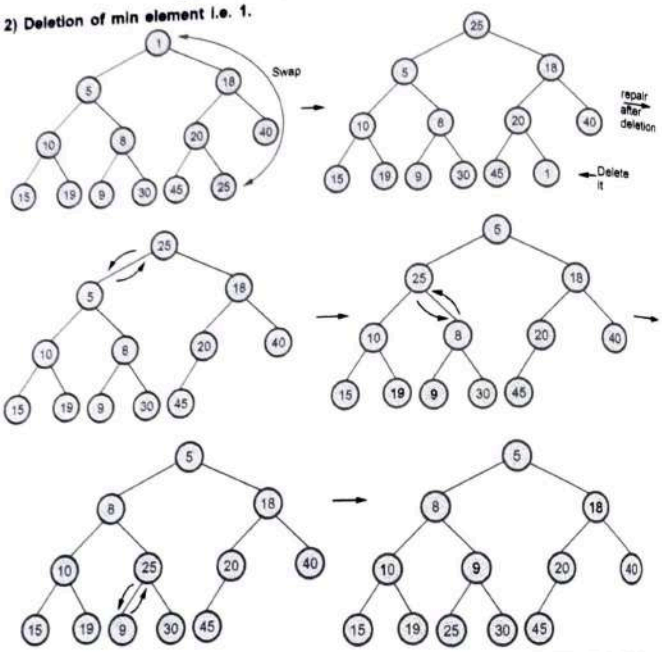


1) Insert 18





2) Deletion of min element i.e. 1.

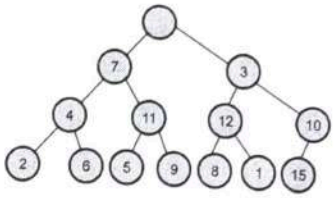


Example 5.13.7 *Define double ended heap. Construct a DEAP for given data 7, 3, 4, 11, 12, 10, 2, 6, 5, 9, 8, 1, 15. After deleting max element from DEAP, what will be the resultant DEAP ?*

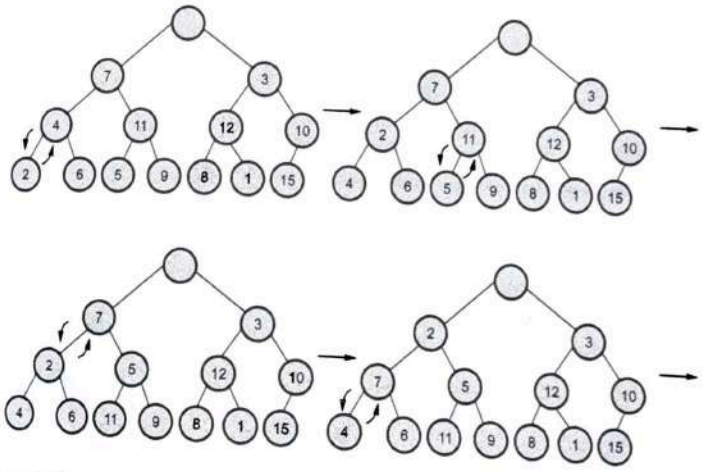
Solution : DEAP is a complete binary tree data structure which is either empty or satisfies following properties -

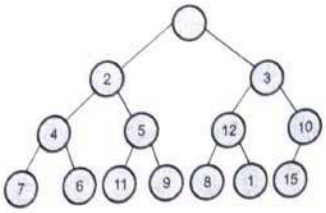
1) The root contains no element.
2) The left subtree is min-heap data structure.
3) The right subtree is max heap data structure.
4) If right subtree is not empty then i is any node in left subtree and j is corresponding node in right subtree. If such a corresponding j does not exist then j is the node in right subtree which corresponds to parent of i.

DEAP construction

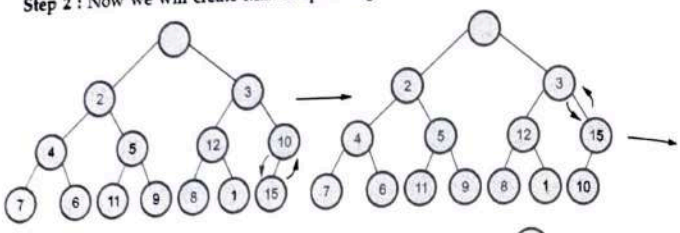


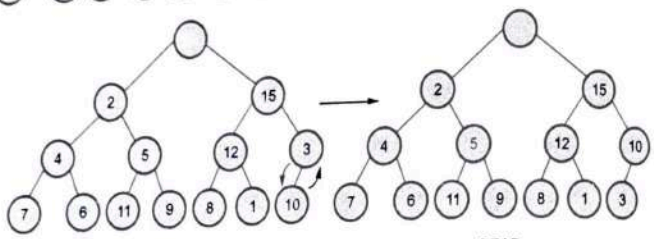
Step 1 : Create Min heap for left sub-branch.





Step 2 : Now we will create Max heap for right subbranch.



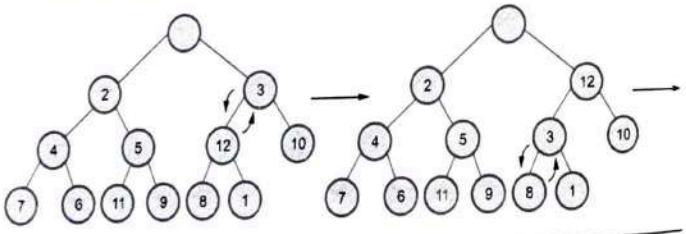


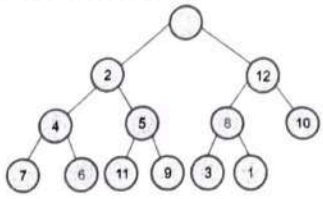
DEAP

Deletion of maximum element from DEAP

The element 15 is deleted.

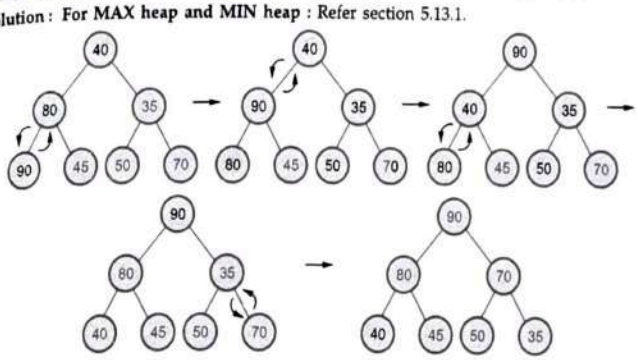
The last element is placed at the position of 15.





Example 5.13.8 *Explain in brief MAX heap and MIN heap. Draw MAX heap for given list of elements (40, 80, 35, 90, 45, 50, 70).*

Solution : For MAX heap and MIN heap : Refer section 5.13.1.



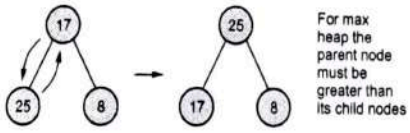
MAX HEAP

Example 5.13.9 *Create a max heap with the following elements :*

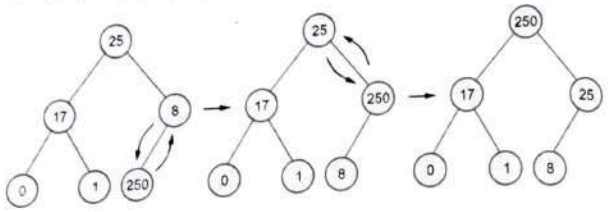
17 25 8 0 1 250 1008 65 48 101 **SPPU : Dec.-15, Marks 4**

Solution : We will create heap by inserting the values in a tree to create a complete or almost complete binary tree.

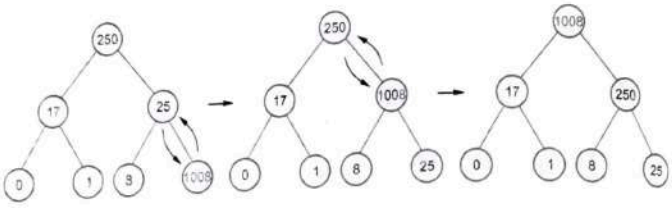
Step 1 :



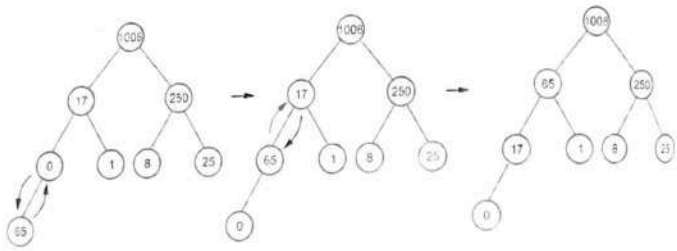
Step 2 : ~~17~~ ~~25~~ ~~8~~ ~~0~~ ~~1~~ ~~250~~ 1008 65 48 101



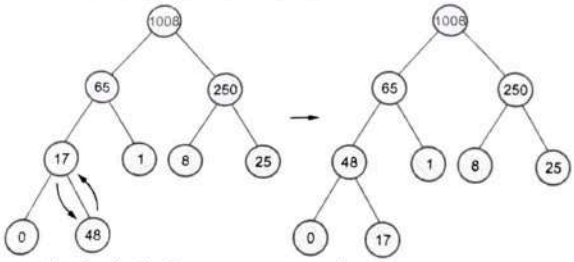
Step 3 : ~~17~~ ~~25~~ ~~8~~ ~~0~~ ~~1~~ ~~250~~ ~~1008~~ 65 48 101



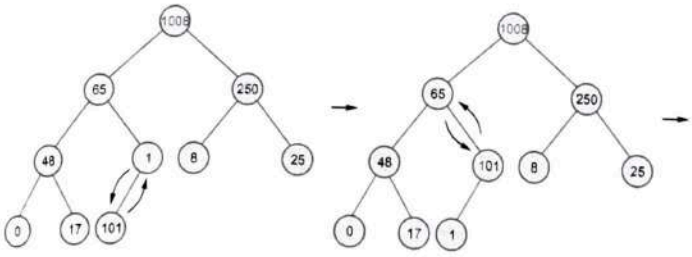
Step 4 : ~~17~~ ~~25~~ ~~8~~ ~~0~~ ~~1~~ ~~250~~ ~~1008~~ ~~65~~ 48 101

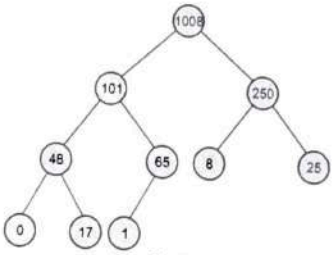


Step 5 : ~~17~~ ~~25~~ ~~8~~ ~~0~~ ~~1~~ ~~250~~ ~~1008~~ ~~65~~ ~~48~~ 101



Step 6 : ~~17~~ ~~25~~ ~~8~~ ~~0~~ ~~1~~ ~~250~~ ~~1008~~ ~~65~~ ~~48~~ ~~101~~





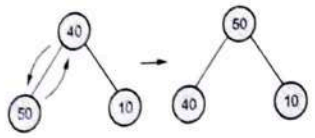
Max heap

Example 5.13.10 *Show stepwise construction of maxheap for the data :*

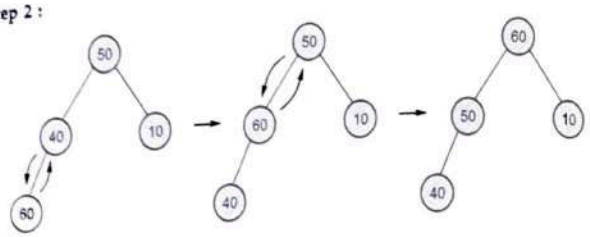
40, 50, 10, 60, 20, 30, 70 **SPPU : May-16, Marks 5**

Solution : Max heap is a complete or almost complete binary tree having parent node greater than its child nodes.

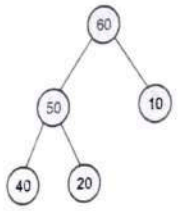
Step 1 :



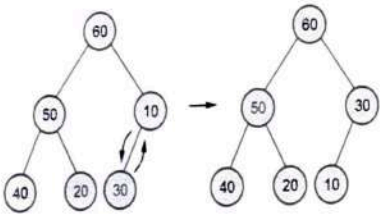
Step 2 :



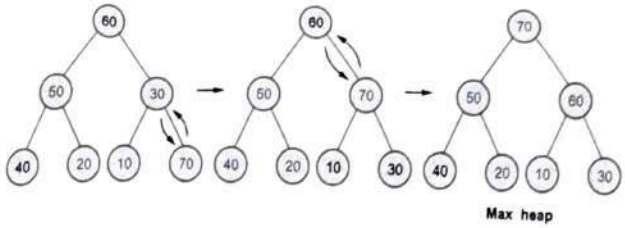
Step 3 :



Step 4 :



Step 5 :



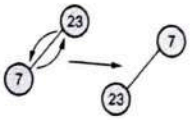
Example 5.13.11 *Construct heap out of following data read from the keyboard :*

23, 7, 92, 6, 12, 14, 40, 44, 20, 21 **SPPU : May-19, Marks 8**

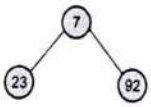
Solution : We will build a min-heap structure.

Step 1 : Insert 23

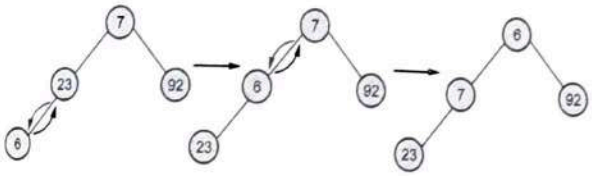
Step 2 : Insert 7



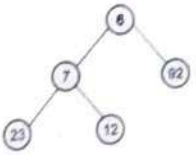
Step 3 : Insert 92



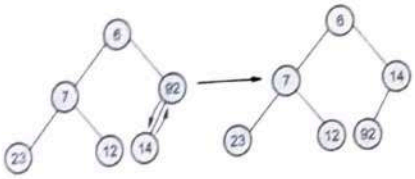
Step 4 : Insert 6



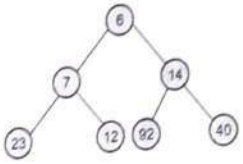
Step 5 : Insert 12



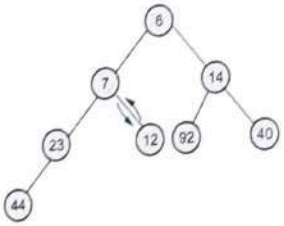
Step 6 : Insert 14



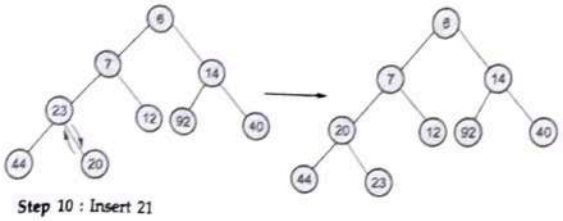
Step 7 : Insert 40



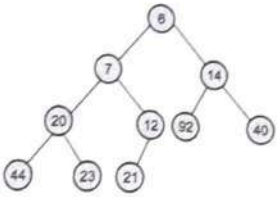
Step 8 : Insert 44



Step 9 : Insert 20



Step 10 : Insert 21

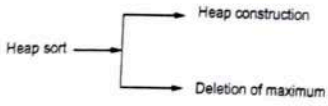


This is required heap structure.

5.14 Heap Sort

SPPU : May-11,13,14, Dec.-11,14,19, Marks 12

Heap sort is a sorting method discovered by J. W. J. Williams. It works in two stages.



1. **Heap construction :** First construct a heap for given numbers.
2. **Deletion of maximum key :** Delete root key always for (n – 1) times to remaining heap. Hence we will get the elements in decreasing order. For an array implementation of heap, delete the element from heap and put the deleted element in the last position in array. Thus after deleting all the elements one by one, if we collect these deleted elements in an array starting from last index of array.

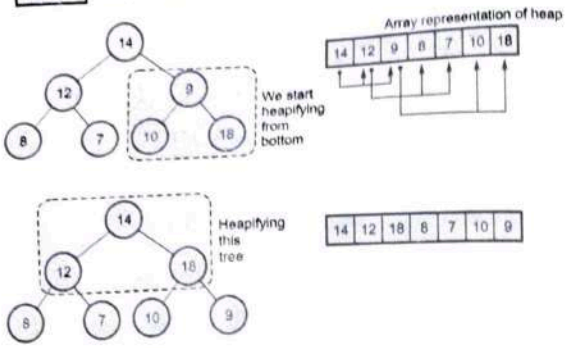
We get a list of elements in a ascending order.

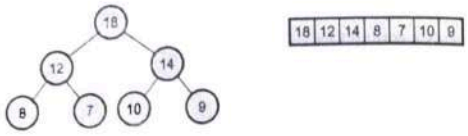
Let us understand this technique with the help of some examples.

Sort the following elements using heap sort.

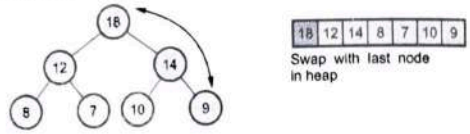
14, 12, 9, 8, 7, 10, 18.

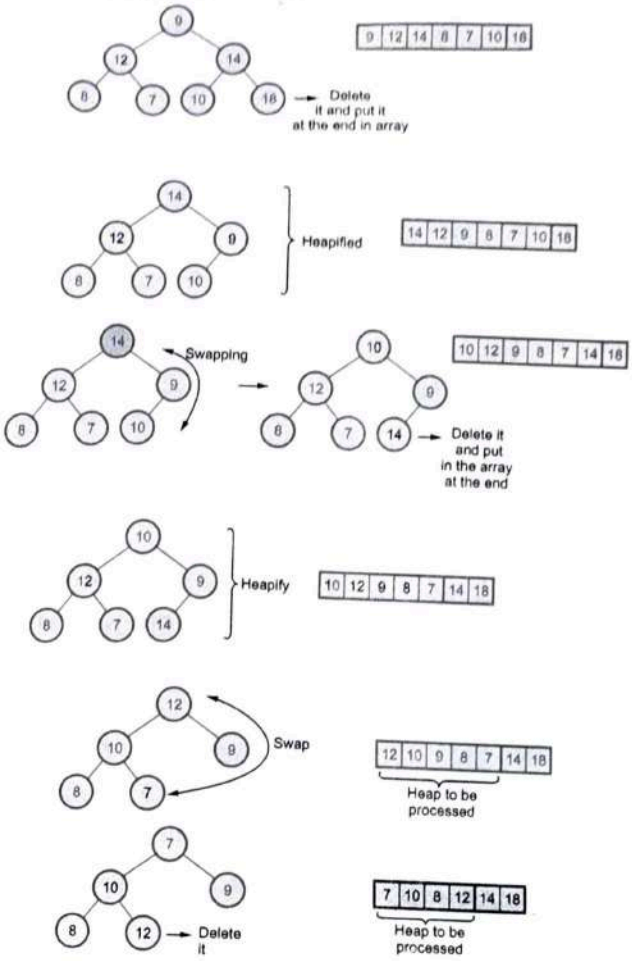
Stage I : Heap construction

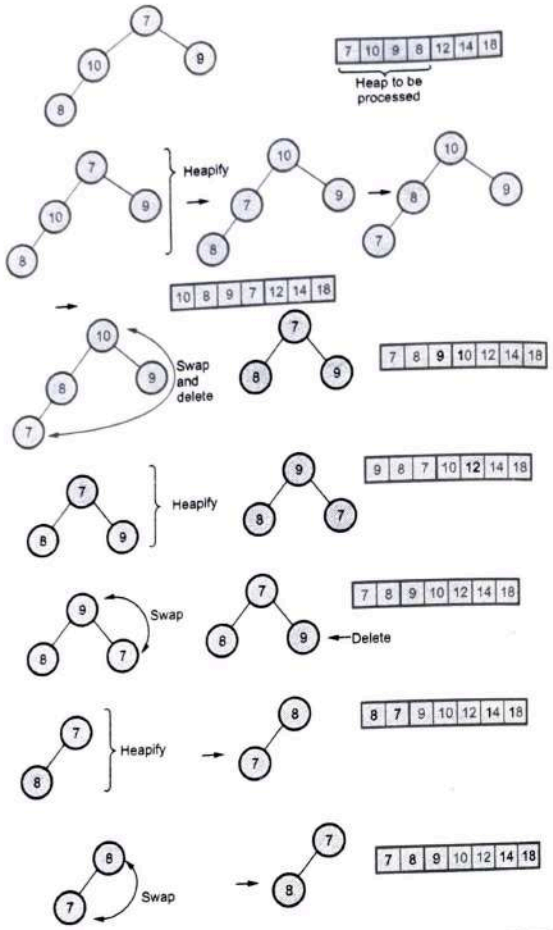


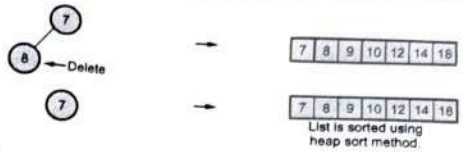


Stage II : Maximum deletion









Pseudo Code

```
void Heap::delet(int item)
{
 int item,temp;
 if(size==0)
  cout<<"Heap is empty";
else
{
  //remove the last element and reheapify
 item=H[size--];
//item is placed at root
temp=1;
child=2
while(child<=size)
{
  if(child<size && H[child]< H[child+1]
   child++;
 if(item>=H[child])
  break;
 H[temp]=H[child]
 temp=child;
 child=child*2;
}
//place the largest item at root
H[temp]=item;
}
}
```

Analysis

As this algorithm works in two stages, we get the running time.

$$\text{Running time for heap sort} = \text{Running time required by heap construction} + \text{Running time required by deletion of root key}$$

$$C(n) = C_1(n) + C_2(n) \quad \text{... (5.14.1)}$$

As the heap construction requires O(n) time,

$$C_1(n) = O(n) \quad \text{... (5.14.2)}$$

Now let us compute time required by second stage (Maximum deletion stage).

Let, number of key comparisons needed for eliminating root keys from heap are for size from n to 2. Then we can establish following relation.

$$C_2(n) \le 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \ldots + 2\lfloor \log_2 1 \rfloor$$

$$\le \underbrace{2\sum_{i=1}^{n-1} \log_2 i}_{2(n-1)\log_2 (n-1)}$$

Because a formula is

$$\sum_{i=1}^{n} \log_2 i = n \log_2 n$$

$$\therefore \sum_{i=1}^{n-1} \log_2 i = (n-1)\log_2 (n-2)$$

But

$$2(n-1)\log_2 (n-1) = (2n-2)\log_2 (n-1)$$

$$= 2n\log_2 (n-1) - 2\log_2 (n-1)$$

$\therefore$ $C_2(n) \le 2n\log_2 n$ we get,

$$C_2(n) = O(n \log_2 n) \quad \text{... (5.14.3)}$$

Now, put equations (5.14.2) and (5.14.3) in equation (5.14.1), we get,

$$C(n) = O(n) + O(n\log_2 n)$$

$$C(n) = O(n\log_2 n)$$

Time complexity of heap sort is Θ **(n log n)** in **both worst and average case.**

Features of heap sort

1. The time complexity of heap sort is Θ (n log n).
2. This is an in-place sorting algorithm. That means it does not require extra storage space while sorting the elements.
3. For random input it works slower than quick sort.
4. Heap sort is not a stable sorting method.
5. The space complexity of heap sort is O(1). As it does not require any extra storage to sort.

5.14.1 Implementation

```
#include<iostream>
#define MAX 10
using namespace std;
class Heap
{
    private:
        int arr[MAX];
        int n;
    public:
        Heap();
        void insert(int num);
        void makeheap();
        void heapsort();
        void display();

};
Heap::Heap()
{
    n=0;
    for(int i=0;i<MAX;i++)
        arr[i]=0;

}

void Heap::insert(int num)
{
    if(n<MAX)
    {
        arr[n]=num;
        n++;
    }
    else
        cout<<"\n Array is full";
}

void Heap::makeheap()
{
    for(int i=1;i<n;i++)
    {
        int val=arr[i];
        int j=i;
        int f=(j-1)/2;
        while(j>0&&arr[f]<val) //creating a MAX heap
        {
            arr[j]=arr[f];
            j=f;
```

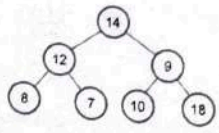
```
                f=(j-1)/2;
            }
            arr[j]=val;
        }
}
void Heap::heapsort()
{
    for(int i=n-1;i>0;i--)
    {
        int temp=arr[i];
        arr[i]=arr[0];
        int k=0;
        int j;
        if(i==1)
            j=-1;
        else
            j=1;
        if(i>2&&arr[2]>arr[1])
            j=2;
        while(j>=0&& temp <arr[j])
        {
            arr[k]=arr[j];
            k=j;
            j=2*k+1;
            if(j+1<=i-1&&arr[j]<arr[j+1])
                j++;
            if(j>i-1)
                j=-1;
        }
        arr[k]=temp;
    }
}
void Heap::display()
{
    for(int i=0;i<n;i++)
        cout<<"\n"<<arr[i];
    cout<<"\n";
}
int main()
{
    Heap obj;

    obj.insert(14);
    obj.insert(12);
    obj.insert(9);
    obj.insert(8);
    obj.insert(7);
```

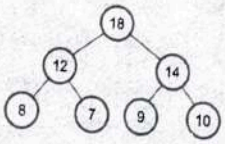


```
    obj.insert(10);
    obj.insert(18);
    cout<<"\n The Elements are ..."<<endl;
    obj.display();
    obj.makeheap();
    cout<<"\n Heapified"<<endl;
    obj.display();
    obj.heapsort();
    return 0;
}
```

Output

```
The Elements are ...

14
12
9
8
7
10
18

Heapified

18
12
14
8
7
9
10

Elements sorted by Heap sort...
7
8
9
10
12
14
18
```





Example 5.14.1 *State algorithm to sort elements of a given array in ascending order using heap sort. Sort the following numbers using heap sort : 48, 0, –1, 82, 108, 72, 54.*

SPPU : May-11, Marks 8

Solution : Algorithm :

Step 1 : Construct the min heap from given set of elements.

Step 2 : Swap the root key with the last node key of the heap.

Step 3 : Delete the last node and store its key in priority queue.

Step 4 : Heapify the remaining heap structure.

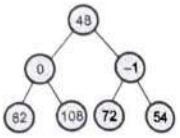
Step 5 : Repeat step 2 to 4 until the last remaining node.

Step 6 : Delete the last node and store it in priority queue.

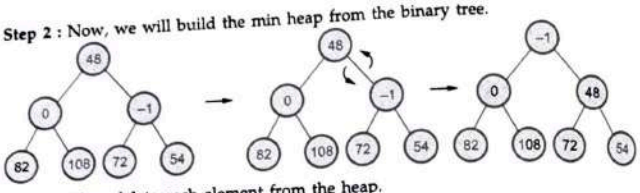
Step 7 : Print all the elements of the priority queue from front to rear. This will display the elements in sorted order.

Consider 48, 0, – 1, 82, 108, 72, 54.

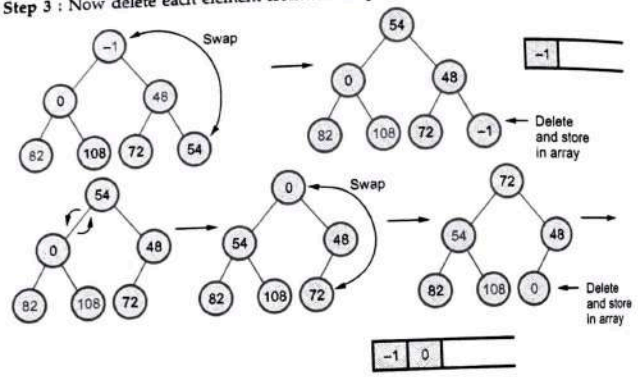
Step 1 : We will create a complete binary tree for given elements.

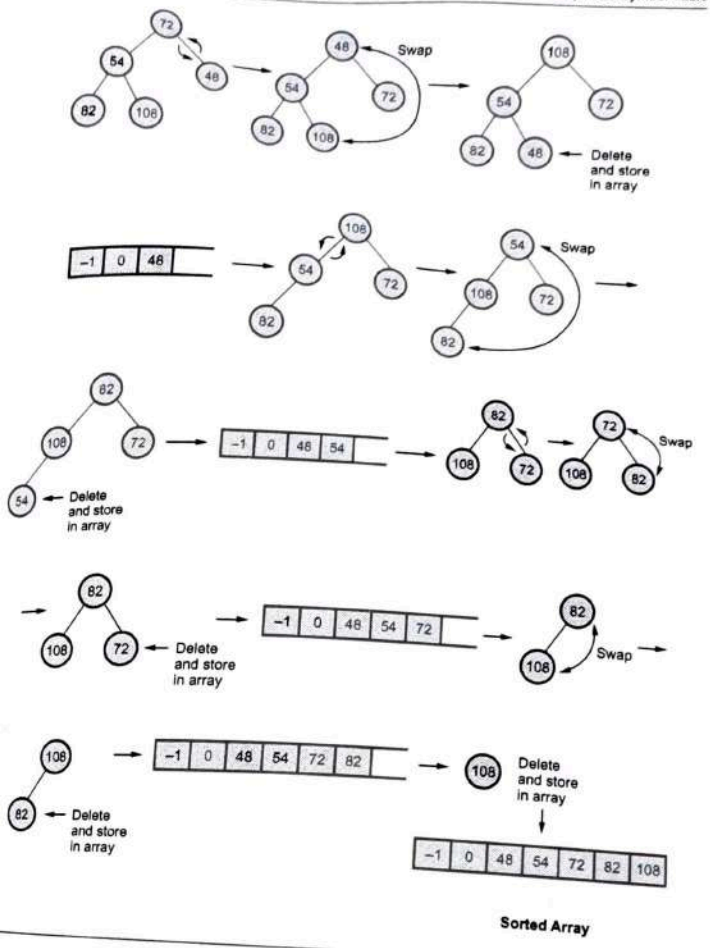


Step 2 : Now, we will build the min heap from the binary tree.



Step 3 : Now delete each element from the heap.





Example 5.14.2 *Sort the following numbers in ascending order using heap sort:* **SPPU : May-14, Marks 10**

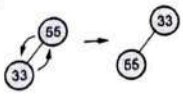
55 33 11 77 44 22 66 88 99

Solution :

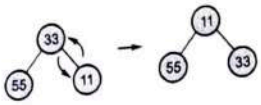
Creation of Min heap

Step 1 :

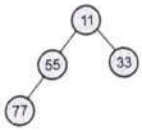
Step 2 :



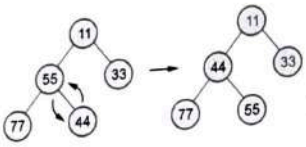
Step 3 :



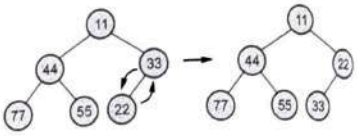
Step 4 :



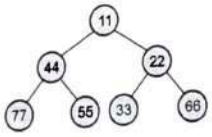
Step 5 :



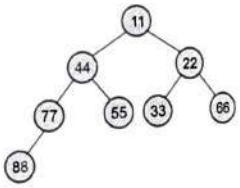
Step 6 :



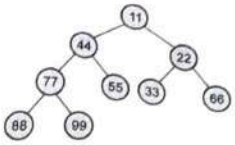
Step 7 :



Step 8 :



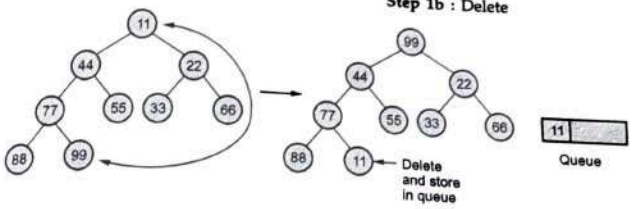
Step 9 :



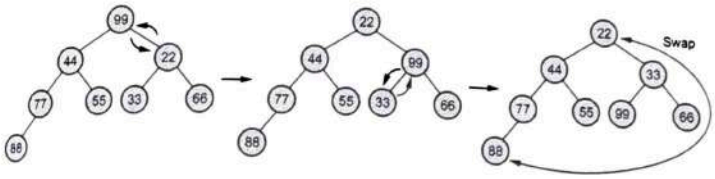
Stage II : Deletion of root Node

Step 1a : Swap

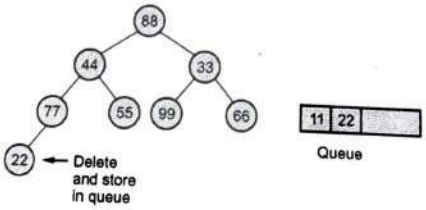
Step 1b : Delete



Step 2a : Heapify and swap



Step 2b : Delete



Continuing in this manner, we will get sorted list as

11	22	33	44	55	66	77	88	99

Queue

Example 5.14.3 *Sort the following number using heap sort and show the sorting stepwise :*
44, 66, 33, 88, 77, 55, 22. **SPPU : Dec.-14, Marks 6**

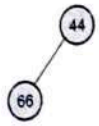
Solution :

Stage I : Heap construction

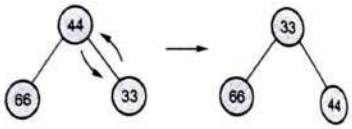
We will construct min-heap.

Step 1 :

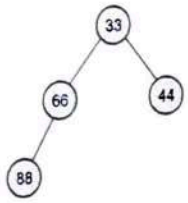
Step 2 :



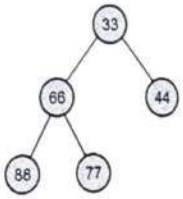
Step 3 :



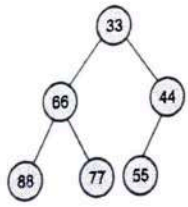
Step 4 :



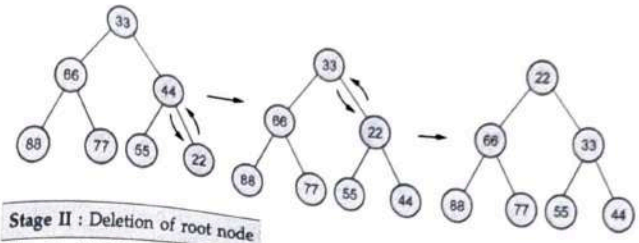
Step 5 :



Step 6 :

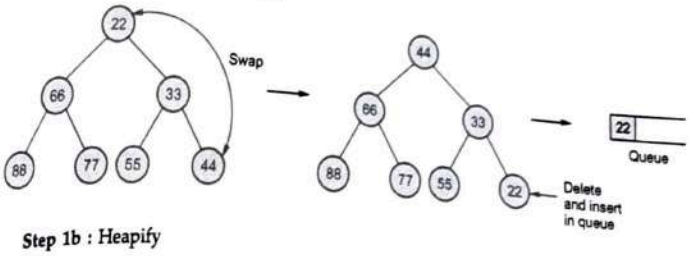


Step 7 :

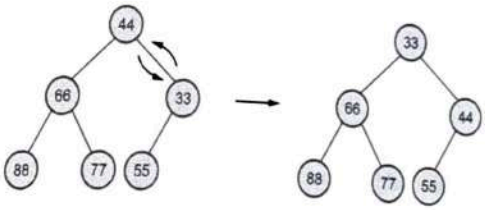


Stage II : Deletion of root node

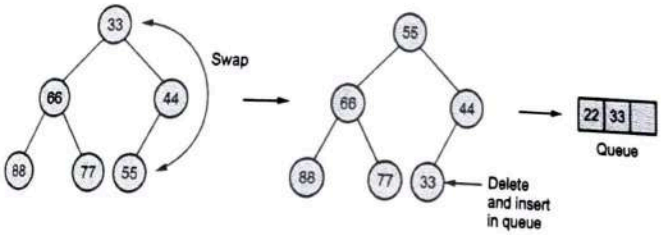
Step 1a : Deletion of root node.



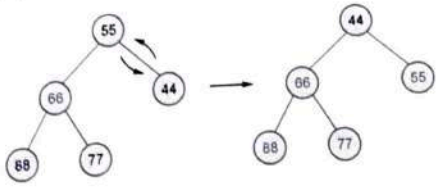
Step 1b : Heapify



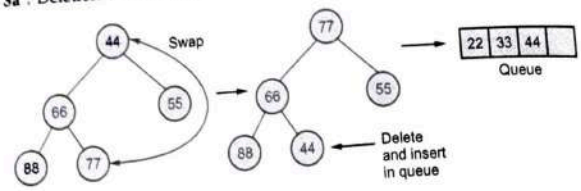
Step 2a : Deletion of root node



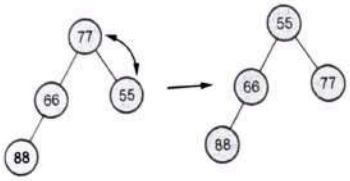
Step 2b : Heapify



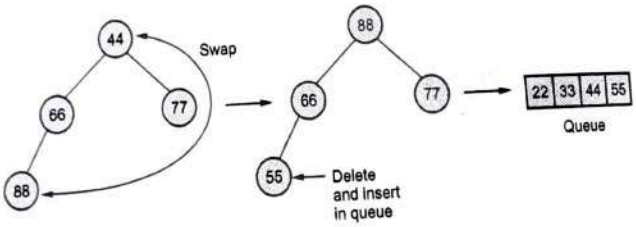
Step 3a : Deletion of root node



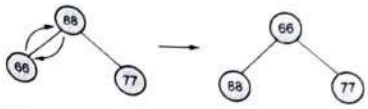
Step 3b : Heapify



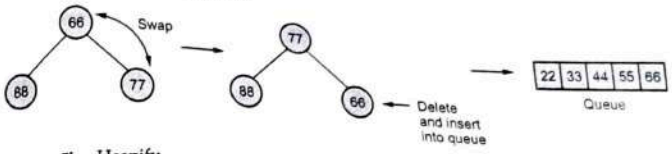
Step 4a : Deletion of root node



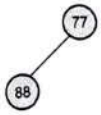
Step 4b : Heapify



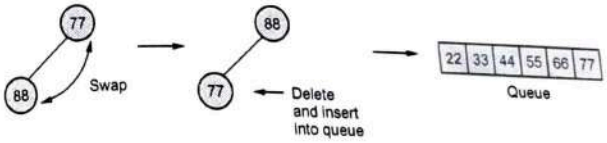
Step 5a : Deletion of root node



Step 5b : Heapify



Step 6a : Deletion of root node



Step 6b : Heapify

As there is only one node remaining, delete it and insert in queue. Finally, we w get sorted elements in queue as,

22	33	44	55	66	77	88

Example 5.14.4 *Which data structures supports to perform sorting using heap data structure ? Explain their use in detail for the following list to sort it in ascending order :*

1, 12, 9, 5, 6, 10

SPPU : Dec.-19, Marks 8

Solution : The priority queue is used to perform sorting using heap data structure.

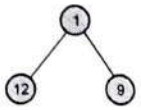
In the following example we will create a min heap structure. We will each time delete a root node from this heap structure and place it in priority queue to form a sorted list of elements.

Stage I : Heapify

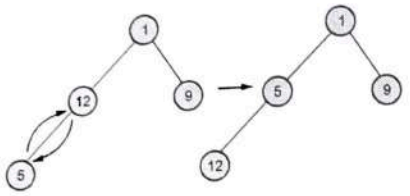
Step 1 :

Step 2 :

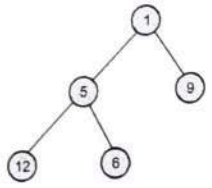
Step 3 :



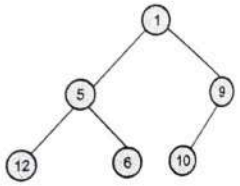
Step 4 :



Step 5 :

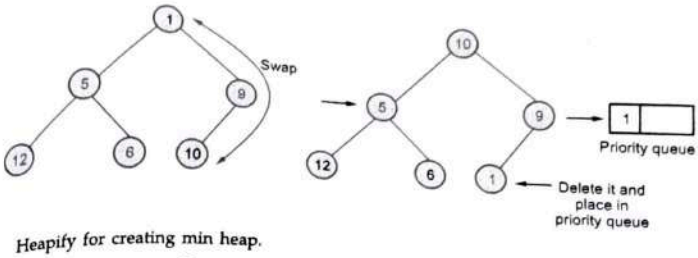


Step 6 :

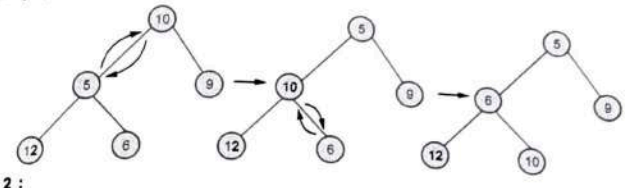


Stage II : Deletion of root

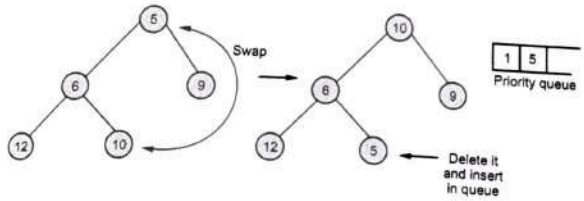
Step 1 :



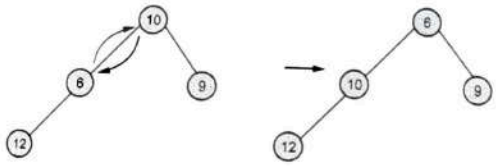
Heapify for creating min heap.



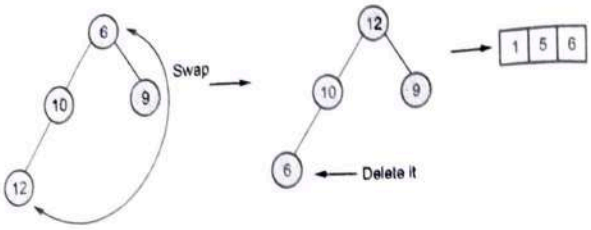
Step 2 :



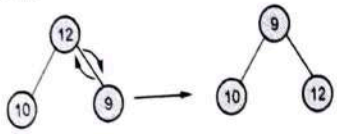
Heapify



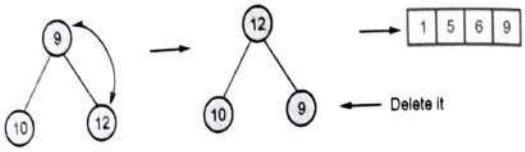
Step 3 :



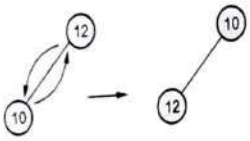
Heapify



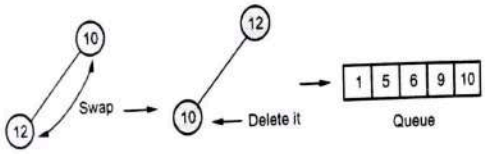
Step 4 :



Heapify



Step 5 :



Step 6 : Finally delete node 12 and insert it into priority queue. Thus we get a list in ascending order in a priority queues as follows,

1	5	6	9	10	12

Review Questions

1. *Define heap. With examples describe different types of heaps. Make a heap out of the following data read from the keyboard : 23, 7, 92, 6, 12, 14, 40, 44, 20, 21. Sort the heap in ascending order.* **SPPU : Dec.-11, Marks 12**
2. *What is heap ? Sort the following numbers in ascending order using heap sort.* **SPPU : May-13, Marks 8**

5.15 Applications of Heap

Various applications of heap are

1. Heap sort method is based on heap data structure. This is one of the best sorting methods.
2. Heap is used in graph algorithms such as finding minimal spanning tree using Prim's algorithm and Dijkstras shortest path algorithm.
3. Heap allows access to min or max element in constant time. Thus heap is used in selection algorithms.

Review Question

1. *Enlist the applications of heap data structure.*

❑❑❑