

## **Session 1**

### **Advanced Concepts of C and Introduction to data structures**

(Page 1 – 22)

- 1.1. Introduction
- 1.2. Data Types
- 1.3. Arrays
- 1.4. Handling Arrays
- 1.5. Initializing the Arrays.
- 1.6. Multidimensional Arrays
- 1.7. Initialization of Two Dimensional Array
- 1.8. Pointers
- 1.9. Advantages and disadvantages of pointers
- 1.10. Declaring and initializing pointers
- 1.11. Pointer Arithmetic
- 1.12. Array of pointers
- 1.13. Passing parameters to the functions
- 1.14. Relation between Pointers and Arrays
- 1.15. Scope Rules and Storage Classes
  - 1.15.1. Automatic Variables
  - 1.15.2. Static Variables
  - 1.15.3. External Variables
  - 1.15.4. Register Variable
- 1.16. Dynamic allocation and de-allocation of memory
  - 1.16.1. Function malloc(size)
  - 1.16.2. Function calloc(n,size)
  - 1.16.3. Function free(block)
- 1.17. Dangling pointer problem.
- 1.18. Structures
- 1.19. Enumerated Constants
- 1.20. Unions

## **Session 2**

### **Introduction to Data structures**

(Page 23– 27)

- 2.1. Introduction
- 2.2. Data Objects & Data Structures
- 2.3. Primitive Data structures
- 2.4. Non Primitive Data structures

- 2.5. Implementation of the Data Structures
- 2.6. Abstract Data type
- 2.7. Algorithm and pseudo code
- 2.8. Complexity of Algorithms

### **Session 3**

**(Page 28 – 51)**

#### **Arrays**

- 3.1. Introduction
- 3.2. Evaluation of Polynomials
  - 3.2.1. Polynomial in a single variable
  - 3.2.2. Polynomial in Two variables
- 3.3. Strings
- 3.4. String Functions
  - 3.4.1. String Length
    - 3.4.1.1. Using Arrays
    - 3.4.1.2. Using Pointers
- 3.5. String Copy
  - 3.5.1. Using Arrays
  - 3.5.2. Using Pointers
- 3.6. String Compare
  - 3.6.1. Using Arrays
- 3.7. Concatenation of S2 to the end of S1
- 3.8. Two Dimensional Arrays
- 3.9. Sparse Matrix
- 3.10. Multiplication of the sparse matrices
- 3.11. Multidimensional Arrays

### **Session 4**

**(Page 52 – 80)**

#### **Linked Lists**

- 4.1. Introduction
- 4.2. Singly Linked List [or] One way chain
  - 4.2.1. Function to add a node at the end of the linked list
  - 4.2.2. Function to add a node at the beginning of the linked list

- 4.2.3. Function to add a node after the specified node
- 4.2.4. Functions for display and count
- 4.2.5. Function to delete the specified node from the list
- 4.3. Doubly Linked Lists [or] Two-way chain
- 4.4. Applications of the linked lists
- 4.5. Linked lists and Polynomials.

## **Session 5**

**(Page 81 - 113)**

### **Stacks and Queues**

- 5.1. Introduction
- 5.2. Stacks
- 5.3. Implementation of Stacks using Arrays
  - 5.3.1. Function To Insert an element into the stack
  - 5.3.2. Function to delete an element from the stack
  - 5.3.3. Function to peep an element from the stack
  - 5.3.4. Function to change an element in the stack
- 5.4. Implementing Stacks using Linked Lists
  - 5.4.1. Function to check whether the stack is empty
  - 5.4.2. Function to check whether the stack is full
  - 5.4.3. Function To push an element into the stack
  - 5.4.4. Function To pop an element from the stack
- 5.5. Queues
- 5.6. Implementing Queues using Arrays
  - 5.6.1. Function to insert an element into the Queue
  - 5.6.2. Function to delete an element from the queue
- 5.7. Queues using linked lists
  - 5.7.1. Function to check whether the queue is empty
  - 5.7.2. Function to check whether the queue is full
  - 5.7.3. Function To insert an element into the queue
  - 5.7.4. Function To delete an element from the queue
- 5.8. Circular Queue
  - 5.8.1. Function to find the next position in the circular queue
  - 5.8.2. Function to find whether the circular queue is empty
  - 5.8.3. Function to find whether the circular queue is full
  - 5.8.4. Function to add an element to the circular queue
  - 5.8.5. Function to delete an element to the circular queue
- 5.9. Circular Queue using linked list[ Circular Lists]
  - 5.9.1. Function to check whether the circular list is empty
  - 5.9.2. Function to check whether the circular list is full

- 5.9.3. Function To insert an element into the circular list
- 5.9.4. Function To delete an element from the circular list
- 5.10. Evaluation of expressions
- 5.11. Postfix expression
- 5.12. Prefix expression

## **Session 6**

### **Graphs**

**(Page 114 – 133)**

- 6.1. Introduction
- 6.2. Adjacency Matrix and Adjacency lists
- 6.3. Breadth first search and Depth first search
- 6.4. Other Tasks For The Graphs:
  - a. To find the degree of the vertex
  - b. To find the number of edges.
  - c. To print a path from one vertex to another.
  - d. To print the multiple paths from one vertex to another.
  - e. To find the number of components in a graph.
  - f. To find the critical vertices and edges.

## **Session 7**

**(Page 134 – 181)**

### **Trees**

#### **(Part 1)**

- 7.1. Introduction
- 7.2. Rooted Tree
- 7.3. Binary Tree
- 7.4. Creation of Binary Tree
- 7.5. Breadth First Search
- 7.6. Binary Search Trees
- 7.7. Binary Search Tree Creation
- 7.8. Searching a value in a binary search tree
- 7.9. Depth First Traversal of the tree
- 7.10. Tree Traversals
  - 7.10.1. Preorder traversal of the tree
  - 7.10.2. Postorder traversal of the tree
  - 7.10.3. Inorder Traversal
- 7.11. Functions to find Predecessor, Successor, Parent,  
Brother

- 7.12. To delete a node from the tree
- 7.13. Expression trees.

## **Session 8**

**(Page 182 – 196)**

### **Trees (PART – 2)**

- 8.1 Introduction
- 8.2. Binary Threaded Tree
  - 8.2.1. Inorder Threading
    - 8.2.1.1. To create a simple binary tree and then thread it
    - 8.2.1.2. To thread the tree during creation
- 8.3. Postorder Threading
- 8.4. Preorder Treading
- 8.5. Traversal of Binary Threaded Tree
  - 8.5.1: Inorder Traversal
  - 8.5.2. Preorder Traversal
  - 8.5.3. Postorder Traversal

## **Session 9**

**(Page 197 – 225)**

### **Searching And Sorting**

- 9.1. Introduction
- 9.2. Searching Techniques
  - 9.2.1. Sequential Search
  - 9.2.2. Binary Search
- 9.3 Sorting
  - 9.3.1. Selection Sort
  - 9.3.2. Bubble sort
  - 9.3.3 Merge Sort
  - 9.3.4. Quick Sort (Partition Exchange Sort)
  - 9.3.5. Radix Sort
- 9. Heap Sort

## Advanced Concepts of C and Introduction to data structures

*During this session you will learn about:*

1. *Structures.*
2. *Arrays.*
3. *Pointers.*
4. *Advantages and disadvantages of a pointer.*
5. *Usage of pointers.*
6. *Difference between arrays and pointers.*
7. *Passing parameters – by value, by reference and by address.*
8. *Dynamic allocation and de-allocation of memory.*
9. *Dynamic arrays.*
10. *Scope and lifetime of a variable.*
11. *Unions.*
12. *Enumerations.*

### 1. Introduction

This chapter familiarizes you with the concepts of arrays, pointers and dynamic memory allocation and de-allocation techniques. We briefly discuss about types of data structures and algorithms. Let us start the discussion with data types.

### 2. Data Types

As we know that the data, which will be given to us, should be stored and again referred back. These are done with the help of variables. A particular variable's memory requirement depends on which type it belongs to. The different types in C are integers, float (Real numbers), characters, double, long, short etc. These are the available built in types.

Many a times we may come across many data members of same type that are related. Giving them different variable names and later remembering them is a tedious

process. It would be easier for us if we could give a single name as a parent name to refer to all the identifiers of the same type. A particular value is referred using an index, which indicates whether the value is first, second or tenth in that parents name.

We have decided to use the index for reference as the values occupy successive memory locations. We will simply remember one name (starting address) and then can refer to any value, using index. Such a facility is known as ARRAYS.

### 3. Arrays

*An array can be defined as the collection of the sequential memory locations, which can be referred to by a single name along with a number, known as the index, to access a particular field or data.*

When we declare an array, we will have to assign a type as well as size.

e.g.

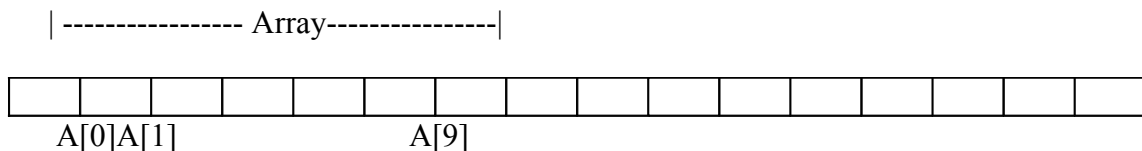
When we want to store 10 integer values, then we can use the following declaration.

```
int A[10];
```

By this declaration we are declaring A to be an array, which is supposed to contain in all 10 integer values.

When the allocation is done for array a then in all 10 locations of size 2 bytes for each integer i.e. 20 bytes will be allocated and the starting address is stored in A.

When we say `A[0]` we are referring to the first integer value in `A`.



fig(1). Array representation

Hence if we refer to the  $i^{\text{th}}$  value in array we should write  $A[i-1]$ . When we declare the array of SIZE elements, where, SIZE is given by the user, the index value changes from 0 to SIZE-1.

Here it should be remembered that the size of the array is ‘always a constant’ and not a variable. This is because the fixed amount of memory will be allocated to the array before execution of the program. This method of allocating the memory is known as ‘static allocation’ of memory.

#### **4. Handling Arrays**

Normally following procedure is followed for programming so that, by changing only one `#define` statement we can run the program for arrays of different sizes.

```
#define SIZE 10

int a[SIZE], b[SIZE];
```

Now if we want this program to run for an array of 200 elements we need to change just the `#define` statement.

#### **5. Initializing the Arrays.**

One method of initializing the array members is by using the ‘*for*’ loop. The following *for* loop initializes 10 elements with the value of their index.

```
#define SIZE 10

main()
{
    int arr[SIZE], i;

    for(i = 0; i < SIZE ; i++ )
    {
        arr[i] = i;
    }
}
```

An array can also be initialized directly as follows.

```
int arr[3] = {0,1,2};
```

An explicitly initialized array need not specify size but if specified the number of elements provided must not exceed the size. If the size is given and some elements are not explicitly initialized they are set to zero.



e.g.

```
int arr[] = {0,1,2};  
int arr1[5] = {0,1,2}; /* Initialized as {0,1,2,0,0} */  
const char a_arr3[6] = "Daniel"; /* ERROR; Daniel has 7  
elements 6 in Daniel and a \0*/
```

To copy one array to another each element has to be copied using for structure.

Any expression that evaluates into an integral value can be used as an index into array.

e.g.

```
arr[get_value()] = somevalue;
```

## 6. Multidimensional Arrays

*An array in which the elements need to be referred by two indices it is called a two-dimensional array or a “matrix” and if it is referred using more than two indices, it will be Multidimensional Array.*

e.g.

```
int arr[4][3];
```

This is a two-dimensional array with 4 as row dimension and 3 as a column dimension.

## 7. Initialization of Two Dimensional Array

Just like one-dimensional arrays, members of matrices can also be initialized in two ways – using ‘for’ loop and directly. Initialization using nested loops is shown below.

e.g.

```
int arr[10][10];  
for(int i = 1; i <= 10; i++)  
    for(int j = 1; j <= 10; j++)  
    {  
        arr[i][j] = i+j;
```

}

Now let us see how members of matrices are initialized directly.

e.g.

```
int arr[4][3] = {{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
```

The nested brackets are optional.

## 8. Pointers

The computer memory is a collection of storage cells. These locations are numbered sequentially and are called addresses.

*Pointers are addresses of memory location. Any variable, which contains an address is a pointer variable.*

Pointer variables store the address of an object, allowing for the indirect manipulation of that object. They are used in creation and management of objects that are dynamically created during program execution.

## 9. Advantages and disadvantages of pointers

Pointers are very effective when

- 12.1. The data in one function can be modified by other function by passing the address.
- 12.2. Memory has to be allocated while running the program and released back if it is not required thereafter.
- 12.3. Data can be accessed faster because of direct addressing.

The only disadvantage of pointers is, if not understood and not used properly can introduce bugs in the program.

## 10. Declaring and initializing pointers

Pointers are declared using the (\*) operator. The general format is:

```
data_type *ptrname;
```

where type can be any of the basic data type such as integer, float etc., or any of the user-defined data type. Pointer name becomes the pointer of that data type.

e.g.

```
int    *iptr;  
char   *cptr;  
float  *fptr;
```

The pointer iptr stores the address of an integer. In other words it points to an integer, cptr to a character and fptr to a float value.

Once the pointer variable is declared it can be made to point to a variable with the help of an address (reference) operator(&).

e.g.

```
int num = 1024;  
int *iptr;  
iptr = &num; // iptr points to the variable num.
```

The pointer can hold the value of 0(NULL), indicating that it points to no object at present. *Pointers can never store a non-address value.*

e.g.

```
iptr1=ival;    // invalid, ival is not address.
```

A pointer of one type cannot be assigned the address value of the object of another type.

e.g.

```
double dval, *dptr = &dval; // allowed  
iptr = &dval ;              //not allowed
```

## 11. Pointer Arithmetic

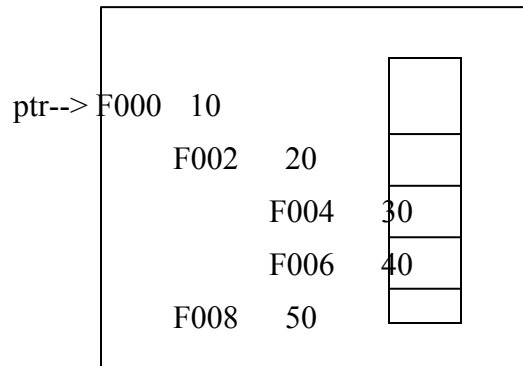
The pointer values stored in a pointer variable can be altered using arithmetic operators. You can increment or decrement pointers, you can subtract one pointer from another, you can add or subtract integers to pointers but two pointers can not be added as

it may lead to an address that is not present in the memory. No other arithmetic operations are allowed on pointers than the ones discussed here. Consider a program to demonstrate the pointer arithmetic.

e.g.

```
#include<stdio.h>

main()
{
    int a[]={10,20,30,40,50};
    int *ptr;
    int i;
    ptr=a;
    for(i=0; i<5; i++)
    {
        printf("%d",*ptr++);
    }
}
```



Output:

10 20 30 40 50

The addresses of each memory location for the array 'a' are shown starting from F002 to F008. Initial address of F000 is assigned to 'ptr'. Then by incrementing the pointer value next values are obtained. Here each increment statement increments the pointer variable by 2 bytes because the size of the integer is 2 bytes. The size of the various data types is shown below for a 16-bit machine. It may vary from system to system.

char	1byte
int	2bytes
float	4bytes
long int	4bytes
double	8bytes
short int	2bytes

## 12. Array of pointers

Consider the declaration shown below:

```
char *A[3]={“a”, “b”, “Text Book”};
```

The example declares ‘A’ as an array of character pointers. Each location in the array points to string of characters of varying length. Here A[0] points to the first character of the first string and A[1] points to the first character of the second string, both of which contain only one character. However, A[2] points to the first character of the third string, which contains 9 characters.

### 13. Passing parameters to the functions

The different ways of passing parameters into the function are:

**12.4.** Pass by value( call by value)

**12.5.** Pass by address/pointer(call by address)

In pass by value we copy the actual argument into the formal argument declared in the function definition. Therefore any changes made to the formal arguments are not returned back to the calling program.

In pass by address we use pointer variables as arguments. Pointer variables are particularly useful when passing to functions. The changes made in the called functions are reflected back to the calling function. The program uses the classic problem in programming, swapping the values of two variables.

```
void val_swap(int x, int y)           // Call by Value
{
    int t;

    t = x;
    x = y;
    y = t;
}

void add_swap(int *x, int *y) // Call by Address
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}
```

```

void main()
{
    int n1 = 25, n2 = 50;

    printf("\n Before call by Value : ");
    printf("\n n1 = %d n2 = %d",n1,n2);
    val_swap( n1, n2 );
    printf("\n After call by value : ");
    printf("\n n1 = %d n2 = %d",n1,n2);

    printf("\n Before call by Address : ");
    printf("\n n1 = %d n2 = %d",n1,n2);
    val_swap( &n1, &n2 );
    printf("\n After call by value : ");
    printf("\n n1 = %d n2 = %d",n1,n2);

}

```

Output:

```

Before call by value   : n1 = 25 n2 = 50
After call by value    : n1 = 25 n2 = 50 // x = 50, y = 25

Before call by address : n1 = 25 n2 = 50
After call by address  : n1 = 50 n2 = 25 //x = 50, y = 25

```

#### 14. Relation between Pointers and Arrays

Pointers and Arrays are related to each other. All programs written with arrays can also be written with the pointers. Consider the following:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
```

To access the value we can write,

```
arr[0] or *arr;
arr[1] or *(arr+1);
```

Since ‘\*’ is used as pointer operator and is also used to dereference the pointer variables, you have to know the difference between them thoroughly.

`*(arr+1)` means the address of `arr` is increased by 1 and then the contents are fetched.

`*arr+1` means the contents are fetched from address `arr` and one is added to the content.

Now we have understood the relation between an array and pointer. The traversal of an array can be made either through subscripting or by direct pointer manipulation.

e.g.

```
void print(int *arr_beg, int *arr_end)
{
    while(arr_beg != arr_end)
    {
        printf("%i", *arr);
        ++arr_beg;
    }
}

void main()
{
    int arr[] = {0,1,2,3,4,5,6,7,8,9}
    print(arr, arr+9);
}
```

`arr_end` initializes element past the end of the array so that we can iterate through all the elements of the array. This however works only with pointers to array containing integers.

## 15. Scope Rules and Storage Classes

Since we explained that the values in formal variables are not reflected back to the calling program, it becomes important to understand the scope and lifetime of the variables.

The storage class determines the life of a variable in terms of its duration or its scope. There are four storage classes:

- automatic
- static
- external
- register

### 1.15.1 Automatic Variables

Automatic variables are defined within the functions. They lose their value when the function terminates. It can be accessed only in that function. All variables when declared within the function are, by default, 'automatic'. However, we can explicitly declare them by using the keyword '*automatic*'.

e.g.

```
void print()
{
    auto int i =0;

    printf("\n Value of i before incrementing is %d", i);
    i = i + 10;
    printf("\n Value of i after incrementing is %d", i);
}

main()
{
    print();
    print();
    print();
}
```

Output:

```
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
```

### 1.15.2. Static Variables

Static variables have the same scope as automatic variables, but, unlike automatic variables, static variables retain their values over number of function calls. The life of a static variable starts, when the first time the function in which it is declared, is executed



and it remains in existence, till the program terminates. They are declared with the keyword *static*.

e.g.

```
void print()
{
    static int i =0;

    printf("\n Value of i before incrementing is %d", i);
    i = i + 10;
    printf("\n Value of i after incrementing is %d", i);

}

main()
{
    print();
    print();
    print();
}
```

Output:

```
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
Value of i before incrementing is : 10
Value of i after incrementing  is : 20
Value of i before incrementing is : 20
Value of i after incrementing  is : 30
```

It can be seen from the above example that the value of the variable is retained when the function is called again. It is allocated memory and is initialized only for the first time.

### **1.15.3. External Variables**

Different functions of the same program can be written in different source files and can be compiled together. The scope of a global variable is not limited to any one function, but is extended to all the functions that are defined after it is declared. However, the scope of a global variable is limited to only those functions, which are in the same file

scope. If we want to use a variable defined in another file, we can use *extern* to declare them.

e.g.

```
// FILE 1 – g is global and can be used only in main() and // // fn1();
```

```
int g = 0;
```

```
void main()
```

```
{
```

```
    :
```

```
    :
```

```
}
```

```
void fn1()
```

```
{
```

```
    :
```

```
    :
```

```
}
```

```
// FILE 2 If the variable declared in file 1 is required to be used in file 2 then it is  
to be declared as an extern.
```

```
extern int g = 0;
```

```
void fn2()
```

```
{
```

```
    :
```

```
    :
```

```
}
```

```
void fn3()
```

```
{
```

```
    :
```

```
}
```

#### 1.15.4. Register Variable

Computers have internal registers, which are used to store data temporarily, before any operation can be performed. Intermediate results of the calculations are also stored in registers. Operations can be performed on the data stored in registers more quickly than on the data stored in memory. This is because the registers are a part of the processor itself. If a particular variable is used often – for instance, the control variable in a loop, can be assigned a register, rather than a variable. This is done using the keyword *register*. However, a register is assigned by the compiler only if it is free, otherwise it is taken as automatic. Also, global variables cannot be register variables.

e.g.

```
void loopfn()
{
    register int i;

    for(i=0; i< 100; i++)
    {
        printf("%d", i);
    }
}
```

## 16. Dynamic allocation and de-allocation of memory

Memory for system defined variables and arrays are allocated at compilation time. The size of these variables cannot be varied during run time. These are called '*static data structures*'.

The disadvantage of these data structures is that they require fixed amount of storage. Once the storage is fixed if the program uses small memory out of it remaining locations are wasted. If we try to use more memory than declared overflow occurs.

If there is an unpredictable storage requirement, sequential allocation is not recommended. The process of allocating memory at run time is called '*dynamic allocation*'. Here, the required amount of memory can be obtained from free memory called 'Heap', available for the user. This free memory is stored as a list called '*Availability List*'. Getting a block of memory and returning it to the availability list, can be done by using functions like:

12.6.

12.7. malloc()

12.8. calloc()

12.9. free()

## 17. Function malloc(size)

This function is defined in the header file <stdlib.h> and <alloc.h>. This function allocates a block of 'size' bytes from the heap or availability list. On success it returns a pointer of type 'void' to the allocated memory. We must typecast it to the type we require like int, float etc. If required space does not exist it returns NULL.

Syntax:

```
ptr = (data_type*) malloc(size);
```

where

**12.10.** ptr is a pointer variable of type data\_type.

**12.11.** data\_type can be any of the basic data type, user defined or derived data type.

**12.12.** size is the number of bytes required.

e.g.

```
ptr=(int*)malloc(sizeof(int)*n);
```

allocates memory depending on the value of variable n.

```
# include<stdio.h>
# include<string.h>
# include<alloc.h>
# include<process.h>
```

```
main()
```

```
{
```

```
    char *str;
```

```
    if((str=(char*)malloc(10))==NULL) /* allocate memory for
                                     string */
```

```
    {
```

```
        printf("\n OUT OF MEMORY");
```

```
        exit(1); /* terminate the program */
```

```
    }
```

```
    strcpy(str,"Hello"); /* copy hello into str */
```

```
    printf("\n str= %s ",str); /* display str */
```

```
    free(str); /* free memory */
```

```
}
```

In the above program if memory is allocated to the str, a string hello is copied into it. Then str is displayed. When it is no longer needed, the memory occupied by it is released back to the memory heap.

### 18. Function calloc(n,size)

This function is defined in the header file <stdlib.h> and <alloc.h>. This function allocates memory from the heap or availability list. If required space does not exist for the new block or n, or size is zero it returns NULL.

Syntax:

```
ptr = (data_type*) malloc(n,size);
```

where

**12.13.** ptr is a pointer variable of type data\_type.

**12.14.** data\_type can be any of the basic data type, user defined or derived data type.

**12.15.** size is the number of bytes required.

**12.16.** n is the number of blocks to be allocated of size bytes.

and a pointer to the first byte of the allocated region is returned.

e.g.

```
# include<stdio.h>
# include<string.h>
# include<alloc.h>
# include<process.h>
```

```
main()
```

```
{
```

```
    char *str = NULL;
```

```
    str=(char*)calloc(10,sizeof(char)); /* allocate memory for
                                         string */
```

```
    if(str == NULL);
```

```
    {
```

```
        printf("\n OUT OF MEMORY");
```

```
        exit(1); /* terminate the program */
```

```
    }
```

```
    strcpy(str,"Hello"); /* copy hello into str */
```

```

printf("\n str= %s ",str);          /* display str */
free(str);                          /* free memory */
}

```

### 19. Function free(block)

This function frees allocated block of memory using malloc() or calloc(). The programmer can use this function and de-allocate the memory that is not required any more by the variable. It does not return any value.

### 17. Dangling pointer problem.

We can allocate memory to the same variable more than once. The compiler will not raise any error. But it could lead to bugs in the program. We can understand this problem with the following example.

```

#include<stdio.h>
#include<alloc.h>

main()
{
    int *a;

    a= (int*)malloc(sizeof(int));
    *a = 10;
    }
    a= (int*)malloc(sizeof(int));
    *a = 20;
    }

```

The diagram illustrates the memory state during the execution of the provided code. It shows three memory boxes represented by rectangles. The first box contains the value 10, and the second box contains the value 20. A third empty box is shown below them. Brackets and arrows indicate the state of the pointer 'a':

- For the first allocation: `a = (int*)malloc(sizeof(int));` and `*a = 10;`, a bracket groups these lines, and an arrow points from the bracket to the first box containing 10.
- For the second allocation: `a = (int*)malloc(sizeof(int));` and `*a = 20;`, a bracket groups these lines, and an arrow points from the bracket to the second box containing 20.
- The third empty box represents memory that has been freed but is still accessible via the pointer 'a', which is a dangling pointer.

In this program segment memory allocation for variable ‘a’ Is done twice. In this case the variable contains the address of the most recently allocated memory, thereby making the earlier allocated memory inaccessible. So, memory location where the value 10 is stored, is inaccessible to any of the application and is not possible to free it so that it can be reused.

To see another problem, consider the next program segment:

```

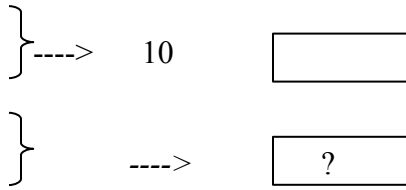
main()
{
    int *a;

```

```

a= (int*)malloc(sizeof(int));
*a = 10;
    free(a);
}

```



The diagram illustrates the state of memory. In the first row, a closing curly brace '}' is followed by a dashed arrow '---->' pointing to the number '10', which is inside a rectangular box. In the second row, another closing curly brace '}' is followed by a dashed arrow '---->' pointing to a question mark '?' inside a rectangular box.

Here, if we de-allocate the memory for the variable ‘a’ using free(a), the memory location pointed by ‘a’ is returned to the memory pool. Now since pointer ‘a’ does not contain any valid address we call it as ‘*Dangling Pointer*’. If we want to reuse this pointer we can allocate memory for it again.

## 18. Structures

A structure is a derived data type. It is a combination of logically related data items. Unlike arrays, which are a collection of similar data types, structures can contain members of different data type. The data items in the structures generally belong to the same entity, like information of an employee, players etc.

The general format of structure declaration is:

```

struct tag
{
    type member1;
    type member2;
    type member3;
    :
    :
}variables;

```

We can omit the variable declaration in the structure declaration and define it separately as follows :

```

struct tag variable;

```

e.g.

```

struct Account
{
    int accnum;
    char acctype;
}

```

```
        char name[25];
        float balance;
    };
```

We can declare structure variables as :

```
struct Account oldcust;
```

We can refer to the member variables of the structures by using a dot operator (.).

e.g.

```
newcust.balance = 100.0
printf(“%s”, oldcust.name);
```

We can initialize the members as follows :

e.g.

```
Account customer = {100, 'w', 'David', 6500.00};
```

We cannot copy one structure variable into another. If this has to be done then we have to do member-wise assignment.

We can also have nested structures as shown in the following example:

```
struct Date
{
    int dd, mm, yy;
};

struct Account
{
    int accnum;
    char acctype;
    char name[25];
    float balance;
    struct Date d1;
};
```

Now if we have to access the members of date then we have to use the following method.



```
Account c1;
```

```
c1.d1.dd=21;
```

We can pass and return structures into functions. The whole structure will get copied into formal variable.

We can also have array of structures. If we declare array to account structure it will look like,

```
Account a[10];
```

Every thing is same as that of a single element except that it requires subscript in order to know which structure we are referring to.

We can also declare pointers to structures and to access member variables we have to use the pointer operator -> instead of a dot operator.

```
Account *aptr;
```

```
printf("%s",aptr->name);
```

A structure can contain pointer to itself as one of the variables, also called self-referential structures.

e.g.

```
struct info
{
    int i, j, k;

    info *next;
};
```

In short we can list the uses of the structure as:

**12.17.** Related data items of dissimilar data types can be logically grouped under a common name.

**12.18.** Can be used to pass parameters so as to minimize the number of function arguments.

**12.19.** When more than one data has to be returned from the function these are useful.

**12.20.** Makes the program more readable.

## 19. Enumerated Constants

Enumerated constants enable the creation of new types and then define variables of these types so that their values are restricted to a set of possible values. Their syntax is:

```
enum identifier {c1,c2,...}[var_list];
```

where

**12.21.** enum is the keyword.

**12.22.** identifier is the user defined enumerated data type, which can be used to declare the variables in the program.

**12.23.** {c1,c2,...} are the names of constants and are called enumeration constants.

**12.24.** var\_list is an optional list of variables.

e.g.

```
enum Colour{RED, BLUE, GREEN, WHITE, BLACK};
```

Colour is the name of an enumerated data type. It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1 and so on.

Every enumerated constant has an integer value. If the program doesn't specify otherwise, the first constant will have the value 0, the remaining constants will count up by 1 as compared to their predecessors.

Any of the enumerated constant can be initialised to have a particular value, however, those that are not initialised will count upwards from the value of previous variables.

e.g.

```
enum Colour{RED = 100, BLUE, GREEN = 500, WHITE, BLACK = 1000};
```

The values assigned will be RED = 100, BLUE = 101, GREEN = 500, WHITE = 501, BLACK = 1000

You can define variables of type Colour, but they can hold only one of the enumerated values. In our case RED, BLUE, GREEN, WHITE, BLACK.

You can declare objects of enum types.

e.g.

```
enum Days{SUN, MON, TUE, WED, THU, FRI, SAT};
Days day;
Day = SUN;
Day = 3; // error int and day are of different types
Day = hello; // hello is not a member of Days.
```

Even though enum symbolic constants are internally considered to be of type unsigned int we cannot use them for iterations.

e.g.

```
enum Days{SUN, MON, TUE, WED, THU, FRI, SAT};
for(enum i = SUN; i<SAT; i++) //not allowed.
```

There is no support for moving backward or forward from one enumerator to another.

However whenever necessary, an enumeration is automatically promoted to arithmetic type.

e.g.

```
if( MON > 0)
{
    printf("Monday is greater");
}
```

```
int num = 2*MON;
```

## 20. Unions

A union is also like a structure, except that only one variable in the union is stored in the allocated memory at a time. It is a collection of mutually exclusive variables, which means all of its member variables share the same physical storage and only one variable is defined at a time. The size of the union is equal to the largest member variables. A union is defined as follows:

```
union tag
{
    type memvar1;
```

```

        type memvar2;
        type memvar3;
        :
        :
};

```

A union variable of this data type can be declared as follows,

```
union tag variable_name;
```

e.g.

```

union utag
{
    int num;
    char ch;
};

```

```
union tag ff;
```

The above union will have two bytes of storage allocated to it. The variable num can be accessed as ff.sum and ch is accessed as ff.ch. At any time, only one of these two variables can be referred to. Any change made to one variable affects another.

Thus unions use memory efficiently by using the same memory to store all the variables, which may be of different types, which exist at mutually exclusive times and are to be used in the program only once.

In this chapter we have studied some advanced features of C. We have seen how the flexibility of language allowed us to define a user-defined variable, is a combination of different types of variables, which belong to some entity. We also studied arrays and pointers in detail. These are very useful in the study of various data structures.

2. *Data objects and Data structures.*
3. *Classification of data structures.*
4. *Implementation of data structures.*
5. *Abstract data type.*
6. *Algorithm and pseudo codes.*
7. *Complexity of algorithm.*

## **2.1. Introduction**

Data structures, is one of the most important subject, which is required by all the software programmers. As a programmer, we would be handling the data in huge quantity. The data we are given, requires to be stored in the memory. The memory should be used efficiently.

The data will be handled on the basis of its type. It could be an integer, real, character etc. There could be even the combination of the data, result in some new type, like structures, unions etc which were covered in the previous session.

## **2.2. Data Objects & Data Structures**

*DATA OBJECT* is a set of elements, say D. This D may or may not be finite, for example, when data object is set of real numbers is infinite and when it is a set of numeric digits it is finite.

*DATA STRUCTURE* contains the data object along with the set of operations, which will be performed on them, or data structure contains the information about the manner in which these data objects are related. The data structures deal with the study of how data is organized in the memory, how efficiently it can be retrieved and manipulated.

They can be classified into

8. Primitive Data structures.
9. Non Primitive Data structures.

## **2.3. Primitive Data structures**

*These are the data structures that can be manipulated directly by machine instructions.* The integer, real, character etc., are some of primitive data structures. In C, the different primitive data structures are *int*, *float*, *char* and *double*.

## 2.4. Non Primitive Data structures

These data structures cannot be manipulated directly by machine instructions. Arrays, linked lists, trees etc., are some non-primitive data structures. These data structures can be further classified into '*linear*' and '*non-linear*' data structures.

*The data structures that show the relationship of logical adjacency between the elements are called linear data structures. Otherwise they are called non-linear data structures.*

Different linear data structures are stack, queue, linear linked lists such as singly linked list, doubly linked list etc.

Trees, graphs etc are non-linear data structures.

## 2.5. Implementation of the Data Structures

When we define the data structure, we also give the functions or the rules used for handling of the data and its logical and/or physical relation. This can be considered as conceptual handling of data for effective working. But very often we face limitations of a particular language or by the available data i.e. its type and size. When these come into picture along with the defined data structure, we may choose some other available form for handling the data along with all the restrictions imposed on it.

Consider the common example of the QUEUE. It will be every day experience that whenever we are in queue – we follow all the rules of the queue. We are aware that the first person in the queue will be the first one to leave it. We will not allow anyone to enter the queue in between the first and the last person. You cannot leave the queue and cannot search for someone.

When we think of processing the data the very first thing that comes to our mind is that, we should process the data in same sequence in which it arrives and hence for storing the data we define the data structure called QUEUE.

The rules to be followed by the queue are :

- 10 . The data can be removed from one end, called as the front.
- 11 . The new data should be always added at the other end called rear.
- 12 . It is possible that there is no data in the queue, which indicates queue empty condition.

13. It is possible that there is no space in the queue for data to be stored which indicates queue full condition.

Now if we think of actually using the concept in our program then it is necessary to store these data items. We should remember which is the first and which is the last data item. If we use different variable names for each item they will not look as if they are related.

The only method to store related items, which we all know by this time, is using an Array. The array can be used to implement queue.

In case of arrays, deletion and addition can be made at any position. For queues, we have to impose some restrictions. We will have to remember two positions indicating the first and the last positions of the queue. Whenever an item is removed the first position will change to its next.

If our first position is beyond the last, the queue will not contain any data items. The deletion of an element from the queue will be logical deletion. If we observe the array, then all the elements are physically available all the time.

The same data structure can also be implemented by another data structure known as LINKED LIST. In short we say that implementing data structure d1 using another data structure d2, is mapping from d1 to d2.

## **2.6. Abstract Data type**

A data structure is a set of domains  $D$ , a designated domain  $P$ , a set of functions  $F$  and set of axioms  $G$ . A triplet  $(D, F, G)$  denotes the data structure  $d$ .

The triplet is referred to as abstract data type (ADT). It is abstract because the axioms in the triple do not give any idea about the representation. Defining the data structure is a continuous process because at the initial stage we can design a data structure, and also we indicate as what it should do. In the later stages of the refinement we try to find the ways in which it can be done or how it can be achieved. Thus it is the total process of specification and implementation.

The idea for representing of data, relation in the data objects and the tasks to be performed will be the specification of the data structure. When we actually try to use all the concepts then we decide as how to achieve each goal, which set by each function. This will be the implementation phase.

## 2.7. Algorithm and pseudo code

Whenever we need to solve a problem it is a better approach to first write down the solution in algorithm or pseudo codes. Once the logic and data structures to be used are decided we can write algorithm or a pseudo code. Later we can implement them into a program of a particular language.

*Algorithm is a set by step solution to a problem written in English alone or with some programming language constructs.*

*Pseudo code is algorithm written in a particular programming language that will be used to implement the algorithm.*

## 2.8. Complexity of Algorithms

When a program is written, it is evaluated on many criteria, like satisfactory results, minimum code, optimum logic etc. The complexity of the algorithm, which is used, will depend on the number of statements executed. Though the execution for each statement is different, we can roughly check as how many times each statement is executed. Whenever we execute a conditional statement, we will be skipping some statements or we might repeat some statements. Hence the total number of statements executed will depend on conditional statements.

At this stage we can roughly estimate that the complexity is the number of times the condition is executed.

e.g.

```
for(i=0; i<n; i++)  
{  
    printf("%d",i);}
```

The output is from 0 to n-1. We can easily say that it has been executed n times. The condition which is checked here is  $i < n$ . It is executed n+1 times - n times when it is true and once when it is false. Hence the total number of statements executes is  $2n+1$ . Also the statement  $i=0$  is executed once and  $i++$  is executed n times.

$$\text{The total} = 1 + (n+1) + (n) + (n) = 3n + 2$$

If we ignore the constants, we say that the complexity is of the order of n. The notation used is big-O. i.e.  $O(n)$ .



## Exercises :

Find the number of times for which each statement is executed.

```
14 . i=2;

    while( i < n)
    {
        for( i = 0; i < n; i++)
        {
            k=k+j;
        }
        i=i+2;
    }

15 . for( i =0; i<n; i++)
    {
        for(j=0; j<i; j++)
        {
            printf(" Enter the num ");
            scanf("%f",&p[i][j]);
        }
    }
```

***During this session you will learn about:***

- *Some applications of Arrays.*
- *Polynomial representation for one variable and two variables.*
- *String Handling – Array of characters.*
- *Two-dimensional arrays- Matrices.*
- *Sparse Matrices.*
- *Multidimensional Arrays.*

## 3.1. Introduction

All the discussion in this book will be in context to the language C. Our main interest is to use memory effectively.

We have already learnt in the first session that when we need many data members of same type, which are related, we use arrays to represent them.

Now let us discuss some applications where arrays can be used.

## 3.2. Evaluation of Polynomials

### 3.2.1. Polynomial in a single variable

Polynomial is a expression of type :

$$P(x) = P_0 + P_1 X^1 + P_2 X^2 + \dots + P_n X^n.$$

Where n is called the degree of that polynomial.

One thing is very clear from the expression, which contain only one variable x, is that we will require storing for each item its coefficient and its power. The following structure would serve the purpose:

```
struct polyno
{
    float coef;
    int pw;};
```

Then we may proceed with the declaration of an array of structures.

```
struct polyno P[SIZE];
```

Array as a data structure will have many applications and also very important facilities. But size limitations will appear as a setback for this choice. The degree of the polynomial will decide the size of the array.

$$(\text{size of the polynomial}) = (\text{degree of polynomial}) + 1$$

But as we are going to input any polynomial, the size i.e. the degree is not known. Hence let us decide the array size bigger than the normal value polynomial say 10 or 100. We have to keep this limitation in mind when using the program.

Now also observe that the degree of each term is an integer and so is the index of the array. Now we may say that the position of the coefficient in the array itself is the degree of x with it. Then we may not even require the array of structures, but even a float array of **coef** will work.

e.g.

```
coef[0] is coefficient of x^0
coef[1] is coefficient of x^1
:
```

:  
coef[i] is coefficient of  $x^i$

The input may be taken in two ways:

1. Ask for coefficient, every time display the power of x i.e.

Please input the coefficient of  $[x^0]$ :

Please input the coefficient of  $[x^1]$ :

Here it will be totally the user's responsibility to give the coefficient as zero if a particular power is absent.

2. Ask the user to input the power and coefficient. Here the user should carefully input all values without repeating any power because we will be directly storing the coefficient at the power location. Here user doesn't have to input all the zero coefficients. It is the programmer's responsibility now to initialize the array with zeroes so that if a particular power is not entered then its coefficient is automatically zero.

Now, we can declare the array as :

```
float coef[SIZE];
```

### 3.2.2. Polynomial in Two variables

Now if we are given a polynomial having two variables per term, i.e. for example :

$$P(x, y) = 3x^3y - 4x^2y^4 + 6x - 7y^3 + 18xy$$

Here we cannot employ the previous method and so we have to go back for declaring a structure as follows :

```
struct poly2
{
    float coef;
    int powx, powy;
};
```

The following is the program, which deals with the polynomials in one variable. It is implemented using array of structures. The terms are stored in the sequence given by the user. Hence we have to scan the second polynomial to match the power when we

perform addition. Also, we are required to remember whether a particular term is added or not. We can use a flag for this purpose.

### **Program 3.2.2. Addition of polynomials in single variable**

```
# include <stdio.h>
# include <conio.h>
# define size 10

int n1, n2, n3;

struct polyx
{
    float coef; /*coefficient of each term, with sign */
    int pw, flag;
} p1[size], p2[size], p3[size];

void read_poly( struct polyx p[], int n); /* to read
                                           polynomial of size n*/

int add_poly(struct polyx p1[],int n1,
            struct polyx p2[], int n2,
            struct polyx p3[]);

int find_pos( struct polyx p[], int n, int power);

void print_poly( struct polyx p[], int n);

float pwr( float x, int n); /* to evaluate x to
                             the power n */

main()
{
    clrscr();

    printf("Enter the no of terms:");
    scanf("%d",&n1);
    printf("\n Enter the data for first polynomial\n");
    read_poly(p1,n1);

    printf("Enter the no of terms:");
    scanf("%d",&n2);
```

```

printf("\n Enter the data for second polynomial\n");
read_poly(p2,n2);

n3= add_poly(p1,n1,p2,n2,p3);

printf(" The first polynomial is...\n");
print_poly(p1,n1);

printf(" The second polynomial is...\n");
print_poly(p2,n2);

printf(" The resultant polynomial is...\n");
print_poly(p3,n3);

} /* main ends */

```

/\*\*\*\*\*\* function to read polynomial \*\*\*\*\*/

```

void read_poly(struct polyx p[], int n)
{
    int i=0;

    for( i= 0; i<n; i++)
    {
        printf(" Give power...");
        scanf("%d",&p[i].pw);
        printf("Give coeff...");
        scanf("%d", &p[i].coef);
        p[i].flag=0;
    }
}

```

/\*\*\*\*\*\* Addition of poly\*\*\*\*\*\*/

```

int add_poly(struct polyx p1[],int n1,struct polyx p2[],
            int n2,struct polyx p3[])
{
    int i=0, j=0,k=0,n3=0;

    n3 = n1;

    do

```

```

    {
        j = find_pos(p2,n2,p1[i].pw); /* having power*/
                                   /* p1[i].pw in p2*/
        if( j == -1) /* -1 indicates it is absent */
        {
            p3[k].pw = p1[i].pw;
            p3[k].coef = p1[i].coef;
            p1[i++].flag = 1;
        }
        else
        {
            p3[k].pw = p1[i].pw; /*when the term is */
                                /* present */
            p3[k].coef = p1[i].coef; /* coef added*/
            p1[i++].flag = 1; /* flag set to 1*/
            p2[j++].flag = 1; /* flag indicates that
                                /* the term has*/ /*been
                                considered */

            k++;
        }
    } while( j < n1);

j= 0;

while( j < n2)
{
    if(p2[j].flag == 0) /* search for the term in p2 */
                       /* whose flag is 0 */

    {
        /* add such a term */
        p3[k].pw = p2[j].pw;
        p3[k].coef = p2[j].coef;
        k++;
        n3++;
    }

    j++;
}
return ( n3);
}

```

/\*\* find the position of the term, which has the given power \*\*/

```

int find_pos( struct polyx p[], int n, int power)
{
    int i=0;

    while( i < n)
    {
        if( p[i].pw == power)
            return(i);
        else
            i++;
    }
    return(-1);
}

```

/\*\*\*\*\*\* prints the polynomial \*\*\*\*\*/

```

void print_poly( struct polyx p[], int n)
{
    int i=0;

    for(i=0;i<n;i++)
        if(p[i].coef != 0 )
            printf(“%2.2f[x^%d]+”, p[i].coef,p[i].pw);

    printf(“%2.2f[x^%d]\n”, p[i].coef,p[i].pw);
}

```

/\*\*\*\*\*\*to evaluate x to the power n \*\*\*\*\*/  
/\* ( considers positive as well as negative powers ) \*/

```

float pwr( float x, int n)
{
    int i=0, m;
    float p=1;

    if(n < 0 )
        m= -n;
    else
        m=n;

    while(i < m)
    {

```

```

        p=p*x;
        i++;
    }
    if( n >=0)
        return p;
    else
        return (1/p);
}

```

Output:

Enter the no of terms: 3

Enter the data for first polynomial

Give power...1

Give coef... 3.000000

Give power...4

Give coef... -4.000000

Give power...3

Give coef... 2.000000

Enter the no of terms: 2

Enter the data for second polynomial

Give power...4

Give coef... -1.000000

Give power...2

Give coef... 1.000000

The first poly is....

$3.00[x^1] + -4.00[x^4] + 2.00[x^3] + 0.00[x^0]$

The second poly is....

$-1.00[x^4] + 1.00[x^2] + 0.00[x^0]$

The resultant polynomial is.. .

$3.00[x^1] + -5.00[x^4] + 2.00[x^3] + 1.00[x^2] + 0.00[x^0]$

You can try other operations like subtraction and multiplication of two polynomials on these. You may write programs to perform these functions on polynomials with two variables.



### 3.3. Strings

*A string is an array of characters.* They can contain any ASCII character and are useful in many operations. A character occupies a single byte. Therefore a string of length N characters requires N bytes of memory. Since strings do not use bounding indexes it is important to mark their end. Whenever enter key is pressed by the user the compiler treats it as the end of string. It puts a special character ‘\0’ (NULL) at the end and uses it as the end of the string marker there onwards.

When the function `scanf()` is used for reading the string, it puts a ‘\0’ character when it receives space. Hence if a string must contain a space in it we should use the function `gets()`.

### 3.4. String Functions

Let us first consider the functions, which are required for general string operations. The string functions are available in the header file “string.h”. We can also write these ourselves to understand their working. We can write these functions using

- Array of Characters
- Pointers

#### 3.4.1. String Length

The length of the string is the number of characters in the string, which includes spaces, and all ASCII characters. As the array index starts at zero, we can say the position occupied by ‘\0’ indicates the length of that string. Let us write these functions in two different ways mentioned earlier.

##### 3.4.1.1. Using Arrays

```
int strlen1(char s[])
{
    int i=0;

    while(s[i] != '\0')
        i++;

    return(i);
}
```

Here we increment the positions till we reach the end of the string. The counter contains the size of the string.

### 3.4.1.2. Using Pointers

```
int strlen1(char *s)
{
    char *p;

    p=s;
    while(*s != '\0')
        s++;
    return(s-p);
};
```

The function is called in the same manner as earlier but in the function we accept the start address in s. This address is copied to p. The variable s is incremented till we get end of string. The difference in the last and first address will be the length of the string.

## 3.5. String Copy : Copy s2 to s1

In this function we have to copy the contents of one string into another string.

### 3.5.1. Using Arrays

```
void strcpy(char s1[], char s2[])
{
    int i=0;

    while( s2[i] != '\0')
        s1[i] = s2[i++];
    s1[i]='\0';
}
```

Till i<sup>th</sup> character is not '\0' copy the character s and put a '\0' as the end of the new string.

### 3.5.2. Using Pointers

```
void strcpy( char *s1, char *s2)
```

```

{
    while( *s2)
    {
        *s1 = *s2;
        s1 ++;
        s2 ++;
    }
    *s1 = *s2;
}

```

### 3.6. String Compare

#### 3.6.1. Using Arrays

```

void strcmp(char s1[], char s2[])
{
    int i=0;
    while( s1[i] != '\0' && s2[i] != '\0')
    {
        if(s1[i] != s2[i])
            break;
        else
            i++;
    }
    return( s1[i] - s2[i]);
}

```

The function returns zero , if the two strings are equal. When the first string is less compared to second, it returns a negative value, otherwise a positive value.

The reader can write the same function using the pointers.

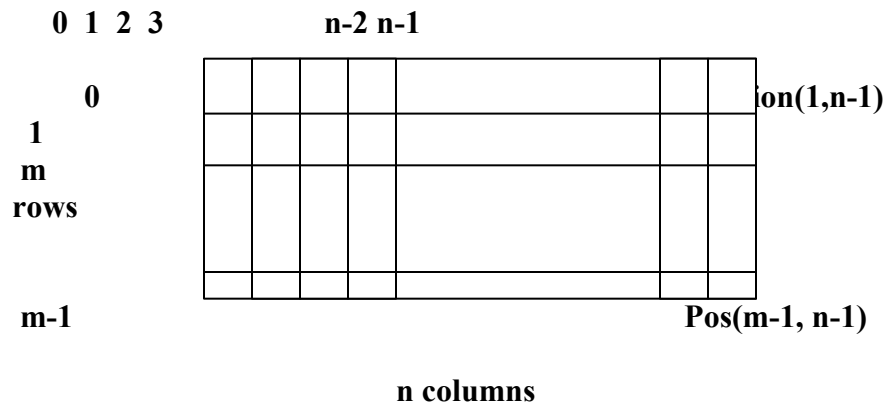
### 3.7. Concatenation of S2 to the end of S1

At the end of string one add the string two. Go till the end of the first string. From the next position copy the characters from the second string as long as there are characters in the second string and at the end close it with a '\0' character. This is left as an exercise for the student.

### 3.8. Two Dimensional Arrays

We have already seen how a two dimensional array can be represented in C, in session 1. Now we can see how it is represented in the memory.

Like we have a linear array with single index, we can also have multidimensional arrays with **n** indexes. At present we will discuss about two-dimensional array, **Matrix**. They are identified with two indices, one is row index and another is the column index. If there are **m** rows and **n** columns in the matrix we say the matrix is of the order (**m x n**). The pictorial representation is:



**fig 1. Matrix of size m x n**

While indicating the position ( $i,j$ ),  $i$  will always be row and  $j$  will always be the column number. The names  $i$  and  $j$  are not important but their position is always very important.

We should always put the condition on the number of variables i.e. row number and column number up to the maximum row and column number. This is very essential to avoid undesirable errors. As we have already seen that the address arithmetic row number will start from **0** up to (**m-1**) and column numbers will start from **0** to (**n-1**).

Diagram illustrating the structure of a 2D array (matrix) with  $m$  rows and  $n$  columns. The rows are indexed from  $0^{\text{th}}$  to  $m-1^{\text{th}}$ . The columns are indexed from  $0$  to  $n-1$ , and the total number of elements is  $mn$ . The diagram shows the first row, the second row, and the last row ( $m-1^{\text{th}}$  row). Below the array, the column indices are listed as  $0, 1, 2, \dots, n-1, n, n+1, \dots, 2n-1, \dots, (m-1)n, \dots, mn-1$ . Below the array, the row indices are listed as  $(0,0)(0,1)\dots(0,n-1)\dots(1,0)(1,1)\dots(1,n-1)\dots(m-1,0)(m-1,1)\dots(m-1,n-1)$ .

Now we can even find from the position  $(i,j)$ , the actual location of in the one-dimensional memory. When we are in the  $i^{\text{th}}$  row, we know that  $(i-1)$  rows are before it and each having  $n$  memory locations. Therefore  $(i-1)*n$  memory locations should be skipped to reach  $i^{\text{th}}$  row. It may appear to be confusing as when we refer to  $i^{\text{th}}$  row, the row number is actually  $(i-1)$  and ( as our counting begins at 0 rather than 1) hence 1<sup>st</sup> row will be row number 0 which means that the row number itself will indicate as how many rows are complete prior to it.

Row number 3 i.e. 0,1,2,3, is actually the 4<sup>th</sup> row and therefore 3 rows are completed before we reach it.

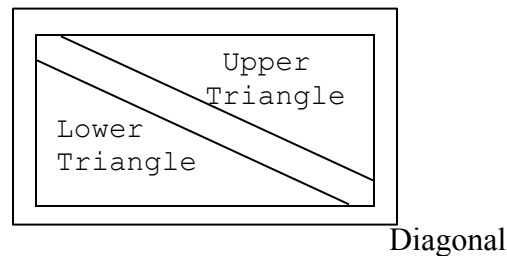
$$\begin{aligned} (0,0) &\rightarrow (0*n+0) \rightarrow 0 \\ (1,4) &\rightarrow (1*n+4) \rightarrow n+4 \\ (5,2) &\rightarrow (5*n+2) \rightarrow 5n+2 \dots \text{etc.} \\ (m-1,n-1) &\rightarrow ((m-1)*n + (n-1)) \\ &\rightarrow mn - n + n + 1 \end{aligned}$$

Now we may also be required to reverse the job, i.e. if we are given a position in one-dimensional memory, we should be able to find row number and column number. If

p is the position, then the row number will be  $(p / n + 0)$  and the column number will be  $(p \% n)$ .

Consider some basic definitions related to matrices.

2. Matrix is said to be **square** if number of rows is equal to the number of columns .  
i.e.  **$m=n$** .
3. When only the diagonal elements of the square matrix are 1, and every other element is zero, it is called as **Identity matrix**.
4. In a square matrix, position wise the elements are divided as :
  - 4.1. Diagonal elements
  - 4.2. Lower triangle, i.e. elements below the diagonal.
  - 4.3. Upper triangle, i.e. elements above the diagonal.



**Fig 3. two dimensional matrix with classifications.**

5. Reading or referring to the elements row wise. In this we will refer the elements in a particular row one by one, i.e. row number remains same in the process but the column numbers will vary. Similarly we can also refer to the elements column-wise.
6. Transpose of the matrix is changing the row elements to the column elements. i.e. the element in the  $(i,j)$  position will occupy  $(j,i)$  position in the transpose. The transpose matrix will have the size  $n \times m$ .
7. Symmetric matrix is a square matrix whose transpose is identical to the original matrix.

Some functions you may use with respect to matrices :

- Read a matrix row-wise/ column-wise.

2. When the number of rows and columns are read outside the function .

```
void readmat(int a[][SIZE], int m, int n)
{
    int i,j;

    for(i=0;i<m;i++)
        for(j=0; j<n; j++)
        {
            printf(" Enter elements for a[%d][%d]=",i,j);
            scanf("%d",&a[i][j]);
        }
}
```

3. When the number of rows and columns are read inside the function. Since these changes have to be reflected back to calling function we use pointer variables as arguments.

```
Void readmat(int a[][SIZE], int *m, int *n)
{
    int i,j;

    printf(" Enter the order of the matrix ");
    scanf("%d %d",m,n);

    for(i=0;i< *m;i++)
        for(j=0; j< *n;j++)
        {
            printf(" Enter elements for a[%d][%d]=",i,j);
            scanf("%d",&a[i][j]);
        }
}
```

```
}
```

- Print the matrix.

```
Void printmat(int a[][SIZE], int m, int n)
{
    int i,j;

    for(i=0;i<m;i++)
    {
        printf(" Enter elements for a[%d][%d]=",i,j);
        printf("%d",a[i][j]);
    }
}
```

- Find the sum of the upper triangle , diagonal and lower triangle matrix elements.

```
int lowmat(int a[][SIZE], int n)
{
    int i,j,sum1=0;

    for(i=0;i<n;i++)
        for(j=0;j<i;j++)
        {
            sum1=sum1+a[i][j];
        }
    return(sum1);
}
```



The above function considers a square matrix of size  $n$ . To find the sum of the upper triangle elements just interchange  $i$  and  $j$  in the statement  $sum1=sum1+a[i][j]$  and make it  $sum1=sum1+a[j][i]$ . To find the sum of the diagonal elements put a check condition of whether it is equal to  $j$  before finding the sum1. These two functions are left to the students.

- Checking whether a matrix is symmetric or not.

For this, the matrix has to be a square matrix and the element at every position  $(i,j)$  must be same as that of the element at the position  $(j,i)$ .

```
Int sym_mat(int a[][SIZE], int n)
{
    int i,j;

    for(i=0;i<n;i++)
        for(j=0;j<i;j++)
        {
            if(a[i][j] != a[j][i])
                return(0);
            else
                return(1);
        }
}
```

- Printing the transpose/ storing the transpose.

To interchange  $i^{\text{th}}$  row with  $j^{\text{th}}$  row, we should swap( exchange) the elements in the columns:

```
void swap_row(int a[][SIZE],int i, int j, int n)
```

```

{
    int k,temp;

    for(k=0;k<n;k++)
    {
        temp = a[i][k];
        a[i][k]= a[j][k];
        a[j][k]=temp;
    }
}

```

### 3.9. Sparse Matrix

*A matrix in which majority of the elements are zeroes they are known as Sparse matrices.*

Generally in scientific calculations, a matrix with hundreds of rows and columns may be required. The elements required could be very few and remaining positions are filled with zeroes. For example, in a 10 x 10 matrix, only 15 elements are non-zero and the remaining 85 locations contain zeroes. In such cases we can store the matrix in a different form. The representation we will be using is a one-dimensional array where each data item is required to store three things.

1. Row number
2. Column number
3. Value

Hence, we will use a structure for this case :

```

struct sparse
{
    int row, col, val;
} sp[SIZE];          /* SIZE is declared as required */

```

Here, we are declaring sp as one-dimensional array of structures. Now this array can represent a matrix having any number of rows and columns but the only restriction is that the number of nonzero elements should not exceed SIZE.

e.g.

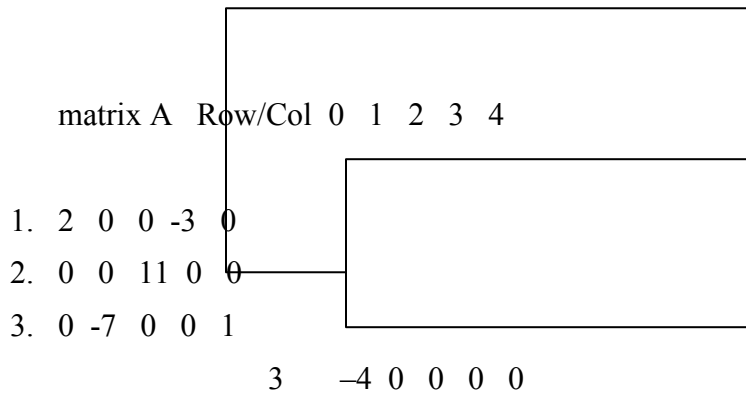


fig 4. matrix of order 4 X 5

Sparse Representation:

		0	1	2	3	4	5	6	7
row	4	0	0	1	2	2	3	3	
col	5	0	3	2	1	4	0	4	
val		7	2	-3	11	-7	1	-4	9

fig 5. sparse representation of fig 4.

Here the position 0 in the sparse matrix representation actually informs about the total number of rows , total number of columns and number of non-zero elements.

Figure 5 shows as to how a particular matrix has been stored in rows, columns and val form. Here we should preferably take the row number in ascending manner and for each row we must take the columns in ascending manner. This is not a rule but it helps in a number of applications.

The next question is how to take the input from the user. One way is to accept the full matrix and then converting it to the sparse form. The other way is to ask the user to input the row number, col number and val.

Now for the first method it is a simple matrix reading.

```
for(i=0; i<m; i++)
for(j=0; j<n ;j++)
{
    scanf("%d",&a[i][j]);

    if(a[i][j] != 0)
    {
        sp[k].row=i;
        sp[k].col=j;
        sp[k].val=a[i][j];
        k++;
    }
}
```

[OR]

```
ans = 1;
do
{
scanf("%d %d%d",&sp[k].row,&sp[k].col,&sp[k].val);
k++;
printf("\n Any more values ? ( 1= Yes / 0=No)");
scanf("%d",&ans);
}while(ans==1);
```

Both these operations will work in an identical way and will generate sp array correctly. Now once the matrix is read properly, the next job is to perform operations like addition, multiplication etc.

### Algorithm

1. To add two sparse matrices, every time we will check whether row numbers are same.
2. If not then copy the smaller row value element into the resultant and increment in that array.
3. Otherwise check the column number, if they are not same, then again copy the smallest element into the resultant and increment that array.
4. When the row numbers and the column numbers are same , we will add the elements and store them into the resultant.

The advantage of the addition of the two sparse matrices is that irrespective of their size we can add two matrices. We can understand this concept by checking the working of the following program.

```
# define <stdio.h>
```

```
# define SIZE 10
```

```
/* new data type SPR with three elements has been defined */
```

```
typedef struct sparse
{
    int row,col,val;
} SPR;
```

```
/* this function reads the matrix in normal form */
```

```
void readmat(int a[][SIZE], int m, int n)
{
    int i,j;

    for(i=0; i<m; i++)
        for(j=0; j<n ;j++)
            scanf("%d",&a[i][j]);
}
```

```
/* a function which converts the matrix into sparse form */
```

```
int make_sparse( int a[][SIZE], int m, int n, SPR s1[])
{
    int i,j,k=1;

    for(i=0; i<m; i++)
        for(j=0; j<n ;j++)
```

```

        if(a[i][j] != 0)
        {
            s1[k].row=i;
            s1[k].col=j;
            s1[k].val=a[i][j];
            k++;
        }

        s1[0].row=m; /* first position contains */
        s1[0].col=n; /* no of rows, columns and */
        s1[0].val=k; /* non-zero elements */

        return(k);
    }

```

/\* a function which displays sparse form of the matrix \*/

```

disply_spr(SPR s[], int n)
{
    int i;

    printf(" Row \t Col \t Val \n");
    printf("-----\n");

    for(i=0;i<n;i++)
        printf("%d\t%d\t%d\n",s[i].row,s[i].col,s[i].val);
}

```

```
/****** main() *****/
```

```
main()
```

```
{
```

```
int a[SIZE][SIZE], b[SIZE][SIZE], i, j, k, m, n, i1, i2, n1, n2, i3, n3;
```

```
SPR s1[SIZE][SIZE], s2[SIZE][SIZE], s3[SIZE][SIZE] ;
```

```
    i1=i2=i3=1;
```

```
    clrscr();
```

```
    printf("Enter the matrix A:\n");
```

```
    printf("Size of matrix A:");
```

```
    scanf("%d %d", &m, &n);
```

```
    printf("Please input the matrix A:\n");
```

```
    read_mat(a, m, n);
```

```
    n1= make_sparse(a, m, n, s1);
```

```
    display_spr(s1, n1);
```

```
    printf("Enter the matrix B:\n");
```

```
    printf("Size of matrix B:");
```

```
    scanf("%d %d", &m, &n);
```

```
    printf("Please input the matrix A:\n");
```

```
    read_mat(b, m, n);
```

```
    n2= make_sparse(b, m, n, s2);
```

```
    display_spr(s2, n2);
```

```
    s3[0].row = s1[0].row > s2[0].row ? s1[0].row : s2[0].row;
```

```
    s3[0].col = s1[0].col > s2[0].col ? s1[0].col : s2[0].col;
```

```
    while( i1 < n1 && i2 < n2) /* till none of the sparse */
```



```

{
/* matrix ends /

if(s1[i1].row == s2[i2].row)
{
if(s1[i1].col == s2[i2].col)
{
s3[i3].row=s1[i1].row; /* when row and col */
s3[i3].col=s1[i1].col; /* are same copy them */
s3[i3].val = s1[i1].val + s2[i2].val; /* add
                                them*/

i1++;
i2++;
i3++;
}
else if( s1[i1].col < s2[i2].col)
{
s3[i3].row=s1[i1].row;      s3[i3].col=s1[i1].col;
s3[i3].val = s1[i1].val;

i1++;
i3++;

/* when col.no of first is less than the*/
/* col.no of second copy the 1st element */

}
else
{
s3[i3].row=s2[i2].row;      s3[i3].col=s2[i2].col;
s3[i3].val = s2[i2].val;

i2++;
i3++;
}
}

```

```

        /* when col.no of first is more than the*/
        /* col.no of second copy the 2nd element */
    }
}

else if( s1[i1].row < s2[i2].row)
{
    s3[i3].row=s1[i1].row;      s3[i3].col=s1[i1].col;
    s3[i3].val = s1[i1].val;

    i1++;
    i3++;

    /* when row.no of first is less than the*/
    /* row.no of second copy the 1st element */

}
else
{
    s3[i3].row = s2[i2].row;
    s3[i3].col = s2[i2].col;
    s3[i3].val = s2[i2].val;

    i2++;
    i3++;

    /* when row.no of first is more than the*/
    /* row.no of second copy the 2nd element */

}
}
}

```

```

while( i2 < n2)

```

```

{
    s3[i3].row = s2[i2].row;
    s3[i3].col = s2[i2].col;
    s3[i3].val = s2[i2].val;

    i2++;
    i3++;

    /* when the first matrix ends */
    /* but the second does not then */
    /* copy the remaining elements of */
    /* 2nd matrix */
}

while( i1 < n1)
{
    s3[i3].row = s1[i1].row;
    s3[i3].col = s1[i1].col;
    s3[i3].val = s1[i1].val;

    i1++;
    i3++;

    /* when the second matrix ends */
    /* but the first does not then */
    /* copy the remaining elements of */
    /* first matrix */
}

n3 = i3;
s3[0].val=n3;
printf("ADDED MATRIX IS ...\n");
display_spr(s3,n3);
}

```

Output:

Enter the matrix A:

Size of matrix A: 4 5

Please input the matrix:

```
2 0 0 -3 0
0 0 4 0 -1
0 0 0 5 0
0 6 0 0 0
```

Row	Col	Val
-----	-----	-----

5	4	7
0	0	2
0	3	-3
1	2	4
1	4	-1
2	3	5
3	1	6

Enter the matrix B:

Size of matrix B: 3 4

Please input the matrix:

```
2 0 0 0
0 -3 -2 0
```

0 0 0 11

Row	Col	Val
4	3	5
0	0	2
1	1	-3
1	2	-2
2	3	11

ADDED MATRIX IS . . .

Row	Col	Val
5	4	8
0	0	4
0	3	-3
1	1	-3
1	2	2
1	4	-1
2	3	16
3	1	6

### 3.10. Multiplication of the sparse matrices:

Now let us consider a problem of multiplying two sparse matrices. It probable could be very difficult as we do not even know the size of the matrices. Secondly, the other matrix, to which the first matrix is going to be multiplied should satisfy the condition, i.e. number of columns of the first matrix must be equal to the number of rows of the second matrix. In sparse matrix we can always add a row of zeroes or a column of zeroes so that the size barrier doesn't exist. Another thing is that we should take transpose of the second matrix or arrange it column-wise so that during multiplication, instead of

row of the first into column of the second, we may have row-to-row multiplication and the result will be the same.

First we should write a function of taking the transpose of the sparse matrix. If you look at the definition of the transpose, which says that (i,j) element in the original matrix will have the position(j,i) in transpose, we get the idea that we should interchange the row number and the column number. This can be very easily done as given below:

```
for(i=0; i<k; i++)
{
    temp = sp[i].row;
    sp[i].row = sp[i].col;
    sp[i].col = temp;
}
```

But then it won't follow the sorted nature of the matrix, i.e. row wise arrangement of elements. hence, we will have to sort sp. Now, the two matrices , sp1 and sp2, are ready for multiplication and the result is to be stored in the matrix sp3.

The pseudo code is given below. You may write the program and implement it.

1. sp1 is first matrix and sp2 is the second matrix in the transpose form.
2. Let M1 and M2 be number of elements in the matrices sp1 and sp2.
3. i and j are positions of sp1 and sp2 and k for sp3.
4. i=0, j=0, k=0
5. sp3[k].row = i  
sp3[k].col = j  
sp3[k].val = 0
6. if(sp1[i].col == sp2[j].col)  
{  
sp3[k].val += sp1[i].bval \* sp2[j].val;

```

        i++;
        j++;
    }

```

7. if(sp1[i].col < sp2[j].col)
 

```

                i++;
            else
                j++;
        
```
8. if((sp1[i].row = sp3[k].row) &&
 ( sp2[j].row=sp3[k].row))
 goto step 6;

Using the above steps, the user is advised to write a program for the multiplication of the sparse matrices.

### 3.11. Multidimensional Arrays:

We can have three dimensional array or even more. The three dimensional arrays will have as usual two indexes for row number and column number and the third index for page number.

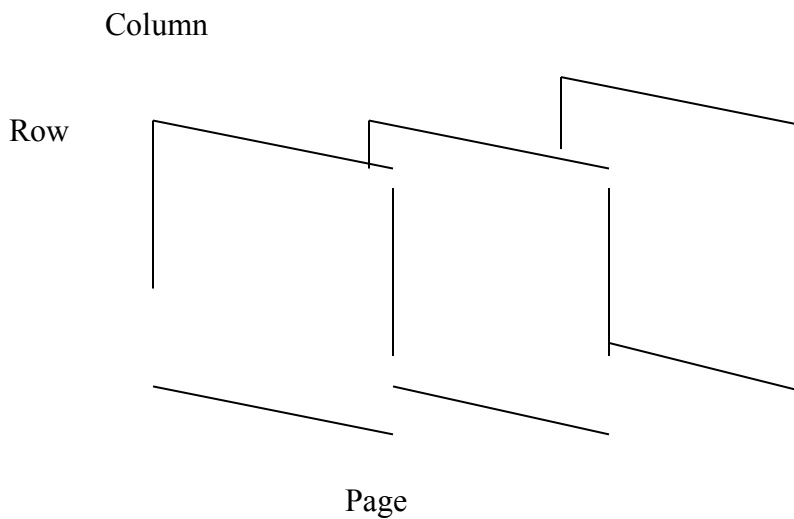


Fig 6. Multidimensional array

We normally read data page wise. This will be used only in case of specific applications where all the data under consideration is related to each other by the position specification. By three dimensional or more dimensional array, the physical interpretation is rather difficult to imagine. When we learn about other data structures and after considering the memory requirement, compared to the utilization of the memory in multidimensional arrays, we may choose some other data structure (very effective) related to the specific application.

Exercises:

1. Write a program to add an element into the array and delete an element from the array.
2. Write a program to add and multiply two matrices in normal form.
3. Write a program to multiply two matrices in Sparse representation for which the algorithm is given in the chapter.
4. WAP (write a program ) for the transpose of the sparse matrix.
5. Discuss the limitations and advantages of the array as a data structure.

## Linked Lists

***During this session you will learn about:***

5. *Linked allocation methods.*
6. *Singly linked lists.*
7. *Basic operations on a singly linked list.*
8. *Doubly linked list.*
9. *Basic operations on a doubly linked list.*
10. *Some applications of the lists.*

### 4.1. Introduction



We have seen representation of linear data structures by using sequential allocation method of storage, as in, arrays. But this is unacceptable in cases like:

4. Unpredictable storage requirements:

The exact amount of data storage required by the program varies with the amount of data being processed. This may not be available at the time we write programs but are to be determined later.

For example, linked allocations are very beneficial in case of polynomials. When we add two polynomials, and none of their degrees match, the resulting polynomial has the size equal to the sum of the two polynomials to be added. In such cases we can generate nodes (allocate memory to the data member) whenever required, if we use linked representation (dynamic memory allocation).

5. Extensive data manipulation takes place.

Frequently many operations like insertion, deletion etc, are to be performed on the linked list.

Pointers are used for the dynamic memory allocation. These pointers are always of same length regardless of which data element it is pointing to( int, float, struct etc,). This enables the manipulation of pointers to be performed in a uniform manner using simple techniques. These make us capable of representing a much more complex relationship between the elements of a data structure than a linear order method.

The use of pointers or links to refer to elements of a data structure implies that elements, which are logically adjacent, need not be physically adjacent in the memory. Just like family members dispersed, but still bound together.

## 4.2. Singly Linked List [or] One way chain

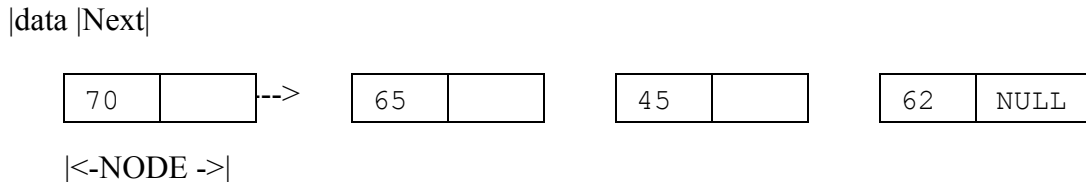
This is a list, which can may consist of an ordered set of elements that may vary in number. Each element in this linked list is called as *node*. A node in a singly linked list consists of two parts, a *information part* where the actual data is stored and a *link part*, which stores the address of the successor(next) node in the list. The order of the elements is maintained by this explicit link between them. The typical node is as shown :

Info	Link
------	------

*NODE*

**Fig 1. Structure of a Node**

Consider an example where the marks obtained by the students are stored in a linked list as shown in the figure :



**fig 2. Singly Linked List**

In figure 2, the arrows represent the links. The **data** part of each node consists of the marks obtained by a student and the **next** part is a pointer to the next node. The NULL in the last node indicates that this node is the last node in the list and has no successors at present. In the above the example the data part has a single element marks but you can have as many elements as you require, like his name, class etc.

There are several operations that we can perform on linked lists. We can see some of them now. To begin with we must define a structure for the node containing a data part and a link part. We will write a program to show how to build a linked list by adding new nodes in the beginning, at the end or in the middle of the linked list. A function display() is used to display the contents of the nodes present in the linked list and a function delete(), which can delete any node in the linked list .

```
typedef struct node
{
    int data;
    struct node *link;
}NODE;

#include <stdio.h>
#include <alloc.h> /* required for dynamic memory */
                  /* allocation */

main()
{
    NODE *p;

    P = NULL; /* empty linked list */
```

```

printf("\n No of elements in the linked list = %d",
        count(p));
append(&p,1); /* adds node at the end of the list */
append(&p,2);
append(&p,3);
append(&p,4);
append(&p,17);

clrscr();
display(p);

add_beg(&p,999);/* adds node at the beginning of the
               list */
add_beg(&p,888);
add_beg(&p,777);

display(p);

add_after(p,7,0); /* adds node after specified node */
add_after(p,2,1);
add_after(p,1,99);

disply(p);
printf("\n No of elements in the linked list = %d",
        count(p));

delete(&p,888); /* deletes the node specified */
delete(&p,1);
delete(&p,10);

disply(p);
printf("\n No of elements in the linked list = %d",
}

```

To begin with the variable **p** has been declared as pointer to a node. This pointer is a pointer to the first node in the list. No matter how many nodes get added to the list, **p** will always be the first node in the linked list. When no node exists in the linked list, **p** will be set to **NULL** to indicate that the linked list is empty. Now we will write and discuss each of these functions.

#### 4.2.1.Function to add a node at the end of the linked list

```

append( NODE **q, int num)
{
    NODE *temp, *r;

    temp = *q;

    if( *q == NULL) /*list empty, create the first node */
    {
        temp = malloc(sizeof(NODE));
        temp->data = num;
        temp->link = NULL;
        *q = temp;
    }
    else
    {
        temp = *q;

        while(temp->link != NULL ) /* goto the end of */
            temp = temp->link; /* list */

        r = malloc(sizeof(NODE));
        r->data = num; /* add node at the */
        r->link = NULL; /* end of the list */
        temp->link = r;
    }
}

```

The append() function has to deal with two situations:

- The node is being added to an empty list.
- The node is being added to the end of the linked list.

In the first case, the condition

```
if( *q == NULL )
```

gets satisfied. Hence space is allocated for the node using malloc() . Data and the link part of this node are set up using the statements :

```
temp->data = num;
temp->link = NULL;
```

Lastly p is made to point to this node, since the first node has been added to the linked list and p must always point to the first node. Note that \*q is nothing but equal to p.

In the other case, when the linked list is not empty, the condition :

```
if( *q == NULL)
```

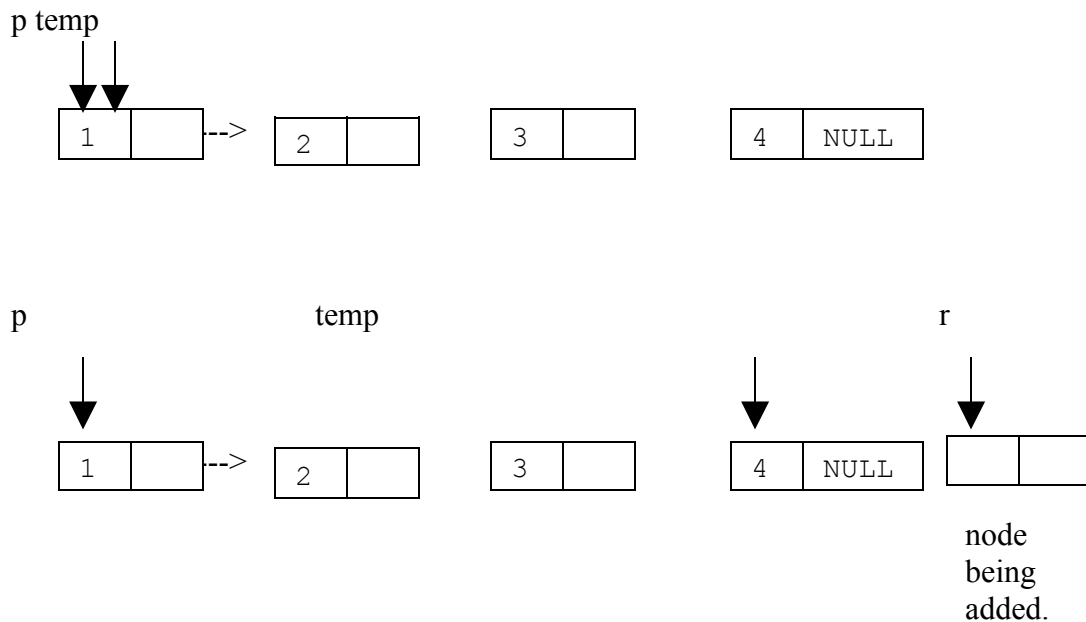
would fail, since \*q (i.e. p is non-NULL). Now temp is made to point to the first node in the linked list through the statement,

```
temp = *q;
```

Then using temp we have traversed through the entire linked list using the statements:

```
while(temp->link != NULL)
    temp=temp->link;
```

The position of the pointer before and after traversing the linked list is shown below:



**Fig 3. Node being added at the end of a SLL**

Each time through the loop the statement `temp= temp->link` makes temp point to the next node in the list. When temp reaches the last node the condition `temp->link != NULL` would fail. Once outside the loop we allocate the space for the new node through the statement

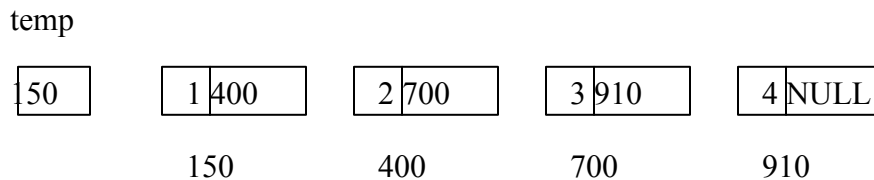
```
r = malloc(sizeof(NODE));
```

Once the space has been allocated for the new node its data part is filled with num and the link part with NULL. Note that this node is now going to be the last node in the list.

All that now remains is connecting the previous last node to this new last node. The previous node is being pointed to by temp and the new last node is by r. they are connected through the statement

```
temp->link = r;
```

There is often a confusion amongst the beginners as to how the statement `temp=temp->link` makes temp point to the next node in the linked list. Let us understand this with the help of an example. Suppose in a linked list containing 4 nodes temp is pointing to the first node. This is shown in the figure below:



**Fig 4. Actual representation of a SLL in memory**

Instead of showing the links to the next node the above diagram shows the addresses of the next node in the link part of each node. When we execute the statement `temp = temp-> link`, the right hand side yields 400. This address is now stored in temp. As a result, temp starts positioning nodes present at address 400. In effect the statement has shifted temp so that it has started positioning to the next node in the linked list.

#### **4.2.2. Function to add a node at the beginning of the linked list**

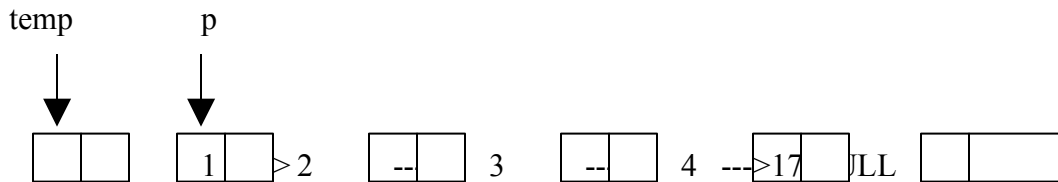
```
add_beg( NODE **q, int num)
{
    temp = malloc(sizeof(NODE));
```

```

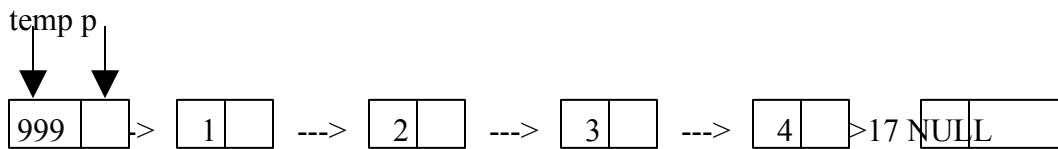
temp->data = num;
temp->link = *q;
*q = temp;
}

```

Suppose there are already 5 nodes in the list and we wish to add a new node at the beginning of this existing linked list. This situation is shown in the figure below.



### Before Addition



### After Addition

*Fig 5. Addition of a node in the beginning of a SLL*

For adding a new node at the beginning, firstly space is allocated for this node and data is stored in it through the statement

```
temp->data = num;
```

Now we need to make the link part of this node point to the existing first node. This has been achieved through the statement

```
temp->link = *q;
```

Lastly this new node must be made the first node in the list. This has been attained through the statement

```
*q = temp;
```

#### 4.2.3. Function to add a node after the specified node

```

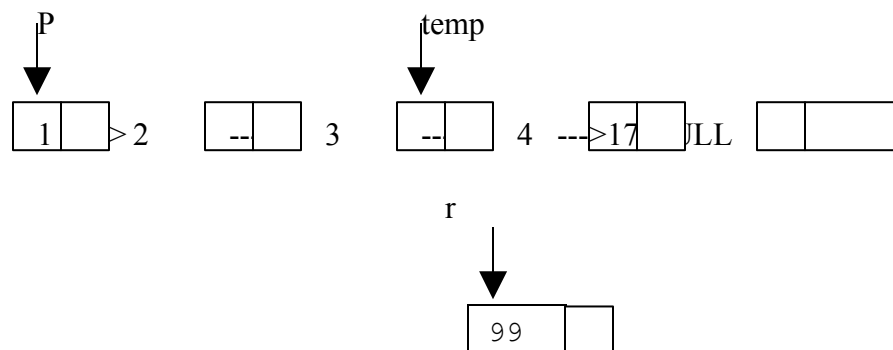
add_after(NODE *q, int loc, int num)
{
    NODE *temp, *t;
    int i;

    temp = q;
    for( i=0 ; i<loc; i++)
    {
        temp = temp->link;
        if(temp == NULL)
        {
            printf(" There are less than %d elements in
                    the list",loc);
            return;
        }
    }
    r = malloc(sizeof(NODE));
    r->data = num;
    r->link = temp->link;
    temp->link = r;
}

```

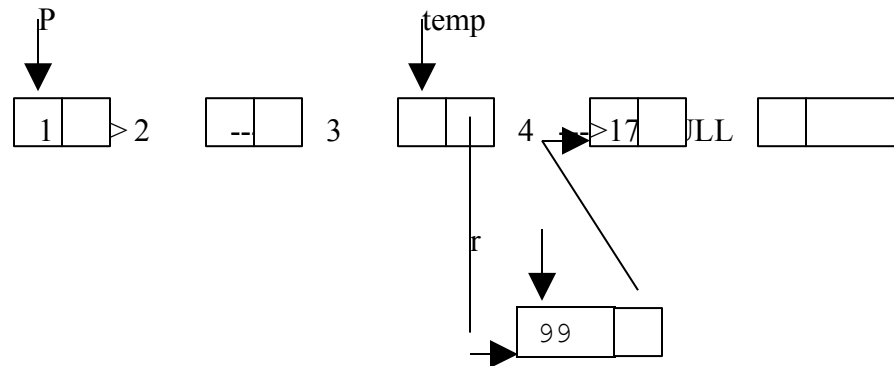
The add\_after() function permits us to add a new node after a specified number of nodes in the linked list.

To begin with, through a loop we skip the desired number of nodes after which a new node is to be added. Suppose we wish to add a new node containing data as 99 after the third node in the list. The position of pointers once the control reaches outside the **for** loop is shown below:





### Before Insertion



### After Insertion

**Fig 6. Insertion of a node in the specified position**

The space is allocated for the node to be inserted and 99 is stored in the data part of it. All that remains to be done is readjustment of links such that 99 goes in between 3 and 4. this is achieved through the statements

```
r->link = temp->link;  
temp->link = r;
```

The first statement makes link part of node containing 99 to point to the node containing 4. the second statement ensures that the link part of the node containing 3 points to the new node.

#### 4.2.4. Functions for display and count

These functions are very simple and straightforward. So no further explanation is required for them.

```
/* function to count the number of nodes in the linked list */
```

```
count(NODE *q)  
{  
    int c = 0;  
  
    while( q != NULL) /* traverse the entire list */  
    {
```

```

        q = q->link;
        c++;
    }

    return (c);
}

```

/\* function to display the contents of the linked list \*/

```

display(NODE *q)
{
    printf("\n");

    while( q != NULL) /* traverse the entire list */
    {
        printf("%d",q->data);
        q=q->link;
    }
}

```

#### 4.2.5. Function to delete the specified node from the list

```

delete(NODE **q, int num)
{
    NODE *old, *temp;

    temp = *q;

    while( temp != NULL)
    {
        if(temp->data == num)
        {
            if(temp == q) /*if it is the first node */
            {
                *q = temp->link;
                free(temp); /* release the memory */
                return;
            }
            else
            {

```

```

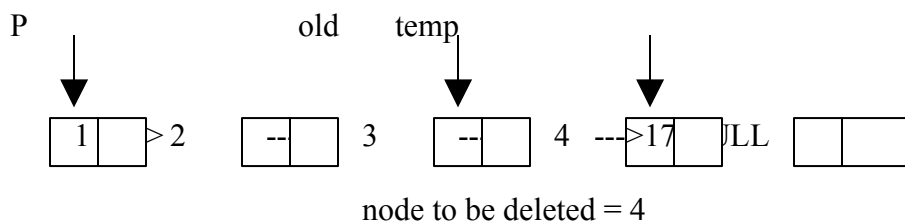
        old->link == temp->link;
        free(temp);
        return;
    }
}
else
{
    old = temp;
    temp=temp->link;
}
}

printf("\n Element %d not found",num);
}

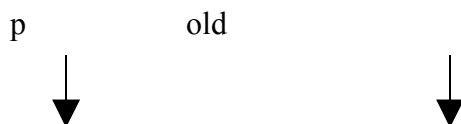
```

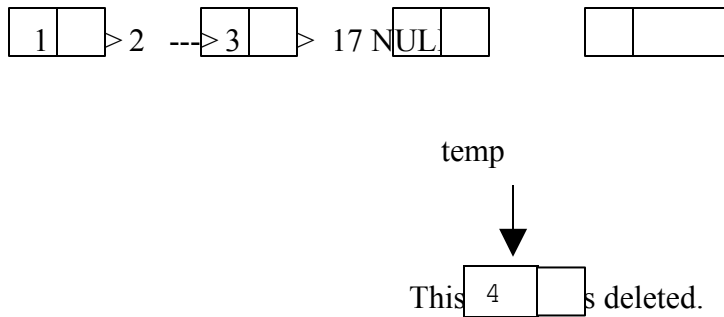
In this function through the **while** loop , we have traversed through the entire linked list, checking at each node, whether it is the node to be deleted. If so, we have checked if the node is the first node in the linked list. If it is so, we have simply shifted **p** to the next node and then deleted the earlier node.

If the node to be deleted is an intermediate node, then the position of various pointers and links before and after deletion are shown below.



**Before deletion**





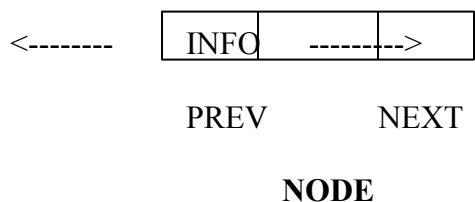
**After deletion**

**Fig 7. Deletion of a node from SLL**

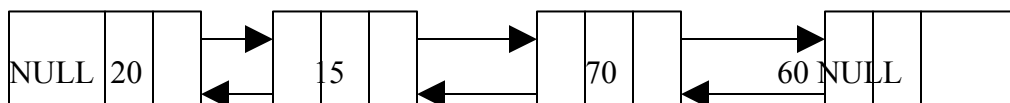
Though the above linked list depicts a list of integers, a linked list can be used for storing any similar data. For example, we can have a linked list of floats, character array, structure etc.

#### 4.3. Doubly Linked Lists [or] Two-way chain

In a singly linked list we can traverse in only one direction (forward), i.e. each node stores the address of the next node in the linked list. It has no knowledge about where the previous node lies in the memory. If we are at the 12<sup>th</sup> node(say) and if we want to reach 11<sup>th</sup> node in the linked list, we have to traverse right from the first node. This is a cumbersome process. Some applications require us to traverse in both forward and backward directions. Here we can store in each node not only the address of the next node but also the address of the previous node in the linked list. This arrangement is often known as a Doubly linked list . The node and the arrangement is shown below:



**Fig 8. Node structure of a DLL**



### Fig 9. Doubly Linked List

The left pointer of the leftmost node and the right pointer of the rightmost node are NULL indicating the end in each direction.

The following program implements the doubly linked list.

```
/* program to maintain a doubly linked list*/

#include < alloc.h >

typedef struct node
{
    int data;
    struct node *prev, *next;
}NODE;

main()
{
    NODE *p;

    p = NULL; /* empty doubly linked list */

    d_append(&p,11);
    d_append(&p,21);

    clrscr();
    display(p);
    printf("\n No of elements in the doubly linked list =
                                         %d", count(p));

    d_add_beg(&p,33);
    d_add_beg(&p,55);

    dispby(p);
    printf("\n No of elements in the doubly linked list =
                                         %d", count(p));

    d_add_after(p,1,4000);
    d_add_after(p,2,9000);
```

```

        disply(p);
        printf("\n No of elements in the linked list = %d",
               count(p));

        d_delete(&p,51);
        d_delete(&p,21);

        disply(p);
        printf("\n No of elements in the linked list = %d",
               count(p));
    }

```

/\* adds a new node at the beginning of the list\*/

```

d_add_beg( NODE **s, int num)
{
    NODE *q;

    /* create a new node */
    q = malloc(sizeof(NODE));

    /* assign data and pointers*/
    q->prev = NULL;
    q->data = num;
    q->next = *s;

    /* make the new node as head node */
    (*s)-> prev = q;
    *s = q;
}

```

/\* adds a new node at the end of the doubly linked list\*/

```

d_append( NODE **s, int num)
{
    NODE *r, *q=*s;

    if( *s == NULL) /*list empty, create the first node */
    {
        *s = malloc(sizeof(NODE));
        (*s)->data = num;
        (*s)->next = (*s)->prev = NULL;
    }
}

```

```

    }
else
{

    while(q->next != NULL ) /* goto the end of */
        q = q->next;    /* list */

    r = malloc(sizeof(NODE));
    r->data = num;        /* add node at the */
    r->next = NULL;      /* end of the list */
    r->prev = q;
    q->next = r;

}
}

```

/\* adds a new node after the specified number of nodes \*/

```

d_add_after(NODE *q, int loc, int num)
{
    NODE *temp;
    int i;

    /* skip to the desired position*/

    for( i=0 ; i<loc; i++)
    {
        q = q->next;
        if(q == NULL)
        {
            printf(" There are less than %d elements in
                    the list",loc);
            return;
        }
    }

    /* insert a new node */

    q = q->prev;
    temp = malloc(sizeof(NODE));
    temp->data = num;
    temp->prev = q;
    temp->next = q->next;

```

```

        temp->next->prev = temp;
        q->next = temp;
    }

```

/\* counts the number of nodes present in the linked list \*/

```

count(NODE *q)
{
    int c = 0;

    while( q != NULL) /* traverse the entire list */
    {
        q=q->next;
        c++;
    }
    return (c);
}

```

/\* Function to display the contents of the doubly linked list \*/  
 /\* in left to right order \*/

```

displayLR(NODE *q)
{
    printf("\n");

    while( q != NULL) /* traverse the entire list */
    {
        printf("%d",q->data);
        q=q->next;
    }
}

```

/\* Function to display the contents of the doubly linked list \*/  
 /\* in right to left order \*/

```

displayRL(NODE *q)
{
    printf("\n");
}

```



```

while( q->next != NULL) /* traverse the entire list */
{
    q=q->next;
}

while( q != NULL)
{
    printf("%d",q->data);
    q=q->prev;
}
}

```

/\* to delete the specified node from the list. \*/

```

d_delete(NODE **s, int num)
{
    NODE *q= *s;

    /* traverse the entire linked list */

    while( q != NULL)
    {
        if(q->data == num)
        {
            if(q == *s) /*if it is the first node */
            {
                *s = (*s)->next;
                (*s)-> prev = NULL;
            }
            else
            {
                /* if the node is last node */
                if( q->next == NULL)
                    q->prev->next = NULL;

                else
                /* node is intermediate */
                {
                    q->prev->next = q->next;
                    q->next->prev = q->prev;
                }
            }
        }
        q=q->next;
    }
}

```

```

        }
        free(q);
    }
    return; /* after deletion */
}
q = q->next ; /* goto next node if not found */
}
printf("\n Element %d not found",num);
}

```

As you must have realized by now any operation on linked list involves adjustments of links. Since we have explained in detail about all the functions for singly linked list , it is not necessary to give step-by-step working anymore. We can understand the working of doubly linked lists with the help of diagrams. We have shown all the possible operations in a doubly linked list along with the functions used in diagrams below:

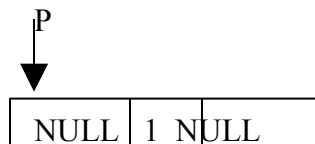
#### *Addition of new node to an empty linked list*

Case 1: Addition to an empty list

Related function : d\_append()

p = \*s = NULL

#### **Before Addition**

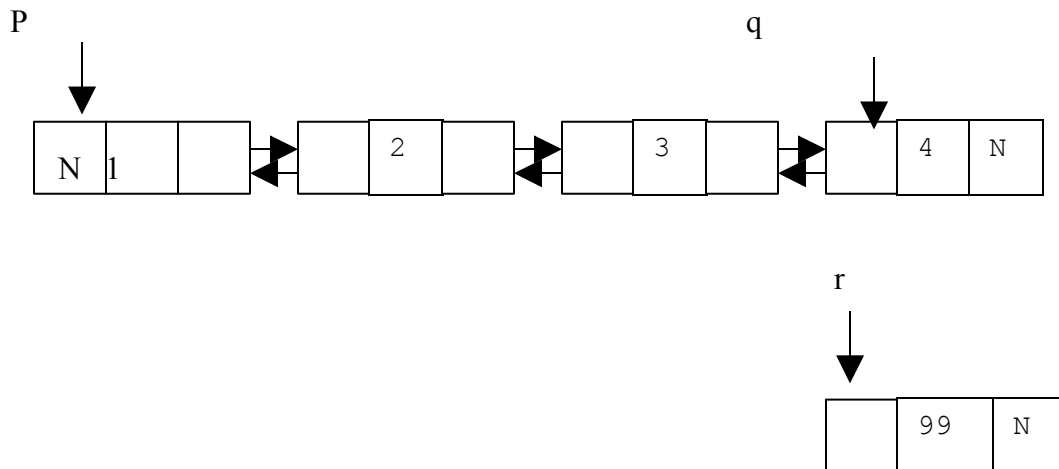


New node

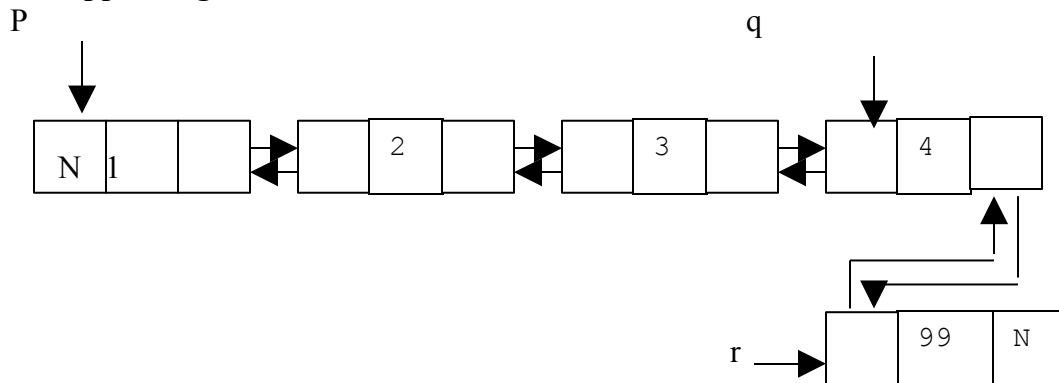
#### **After Addition**

Case 2: Addition to an existing linked list

Related function : d\_append()



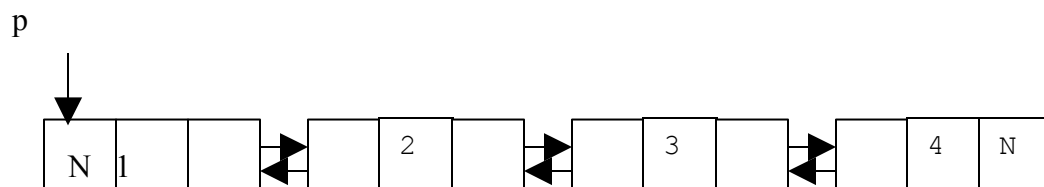
**Before Appending**

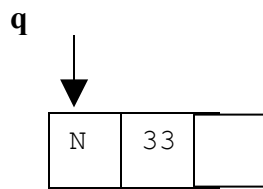


**After Appending**

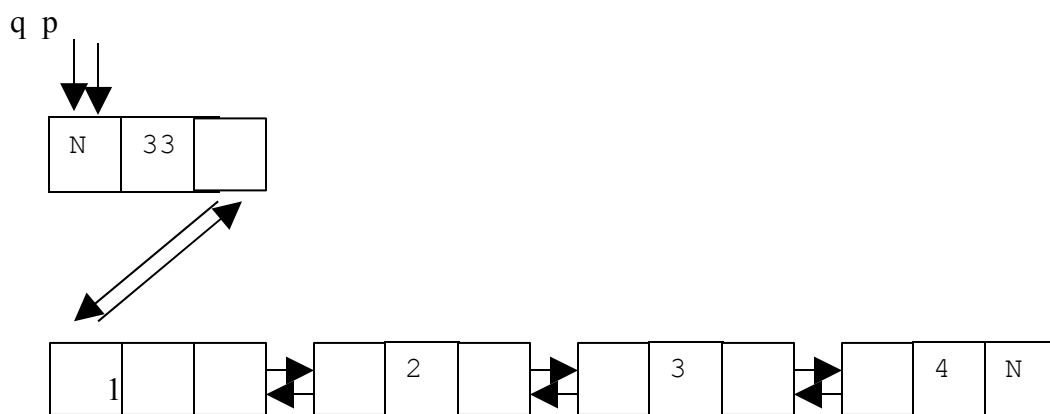
**Addition of new node at the beginning.**

Related Function : `d_add_beg()`





*Before Addition*

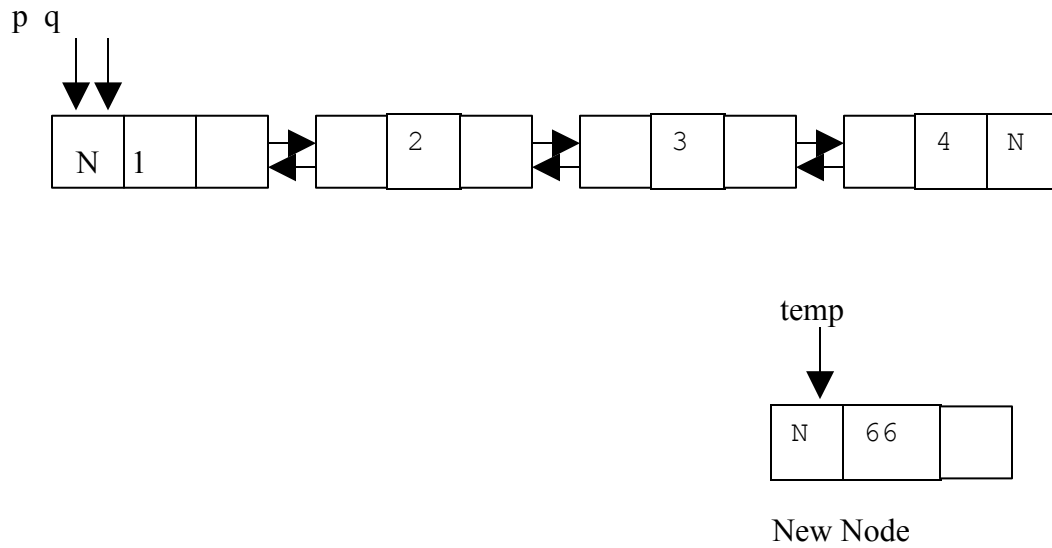


*After Addition*

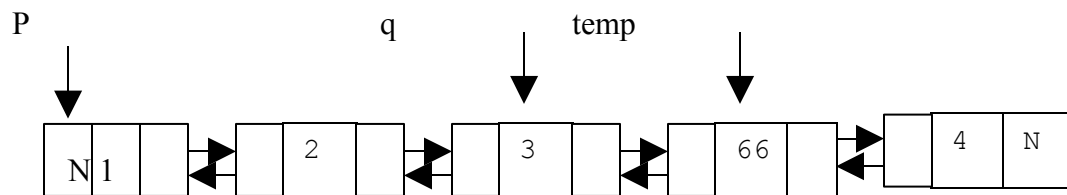
**Fig 10. Addition of nodes at various positions in the DLL**

*Insertion of a new node after a specified node*

Related function : `d_add_after()`



*Before Insertion*

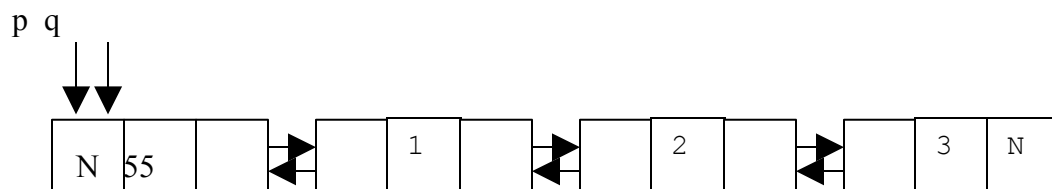


*After Insertion*

**Fig 11. Insertion of node in the DLL**

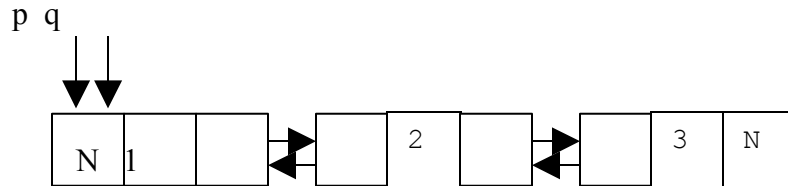
### *Deletion Of a Node*

Case 1: Deletion of first node  
 Related function : `d_delete()`



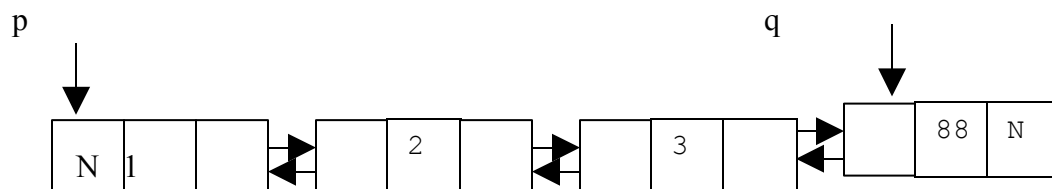
Node to be deleted : 55

## Before Deletion



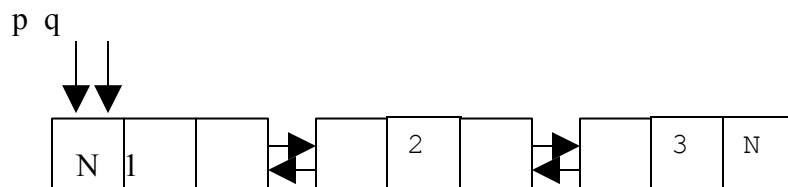
*After Deletion*

Case 2: Deletion of the last node  
Related function : d\_delete()



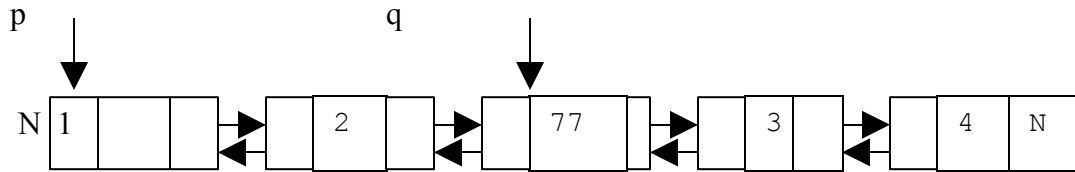
Node to be deleted : 88

## Before Deletion



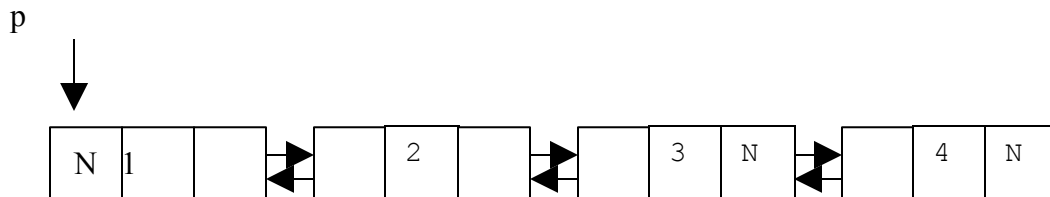
## After Deletion

Case 3: Deletion of the intermediate node  
Related function : d\_delete()



Node to be deleted : 77

### Before Deletion



### After Deletion

**Fig 11. Deletion of nodes from various positions in the DLL**

## 4.4. Applications of the linked lists

In computer science linked lists are extensively used in Data Base Management Systems Process Management, Operating Systems, Editors etc. Earlier we saw that how singly linked list and doubly linked list can be implemented using the pointers. We also saw that while using arrays vary often the list of items to be stored in an array is either too short or too big as compared to the declared size of the array. Moreover, during program execution the list cannot grow beyond the size of the declared array. Also, operations like insertions and deletions at a specified location in a list require a lot of movement of data, thereby leading to an inefficient and time-consuming algorithm.

The primary advantage of linked list over an array is that the linked list can grow or shrink in size during its lifetime. In particular, the linked list's maximum size need not be known in advance. In practical applications this often makes it possible to have several data structures share the same space, without paying particular attention to their relative size at any time.

The second advantage of providing flexibility in allowing the items to be rearranged efficiently is gained at the expense of quick access to any arbitrary item in the list. In arrays we can access any item at the same time as no traversing is required.

We are not suggesting that you should not use arrays at all. There are several applications where using arrays is more beneficial than using linked lists. We must select a particular data structure depending on the requirements.

Let us now see some more applications of the linked lists, like merging two lists and how the linked lists can be used for polynomial representations.

### **Function to Merge the two lists.**

```
Merge(NODE *p, NODE *q, NODE **s)
{
    NODE *z;

    /* If both lists are empty */
    if(p==NULL && q == NULL)
    {
        return;
    }

    /* traverse both linked lists till the end. If end of any one linked list is encountered
    then the loop is terminated */

    while( p != NULL && q != NULL)
    {
        /* if node being added in the first list */

        if ( *s == NULL)
        {
            *s = malloc(sizeof(NODE));
            z = *s;
        }
        else
        {
            z->link = malloc(sizeof(NODE));
            z = z->link;
        }

        if( p->data < q->data)
        {
            z->data = p->data;
            p = p->link;
        }
        else
```



```

    {
        if( p->data > q->data)
        {
            z->data = q->data;
            q = q->link;
        }
        else
        {
            if( p->data == q->data)
            {
                z->data = q->data;
                q = q->link;
                p = p->link;
            }
        }
    }
}

/* if end of first list has not been reached */

while( p != NULL)
{
    z->link = malloc(sizeof(NODE));
    z = z->link;
    z->data = p->data;
    p = p->link;
}

/* if end of second list has not been reached */

while( q != NULL)
{
    z->link = malloc(sizeof(NODE));
    z = z->link;
    z->data = q->data;
    q = q->link;
}

z->link = NULL;
}

```

In this program, assume that structure NODE with data and link is available. Also using add() used for singly linked list earlier we have two linked lists. Three pointers point to three linked lists. The merge function can be called to merge the two linked lists.

This merged list is pointed to by the pointer third. While merging two lists it is assumed that the lists themselves are in ascending order.

#### 4.5. Linked lists and Polynomials.

Polynomials can be maintained using a linked list. To have a polynomial like  $5x^4 - 2x^3 + 7x^2 + 10x - 8$ , each node should consist of three elements, namely coefficient, exponent and a link to the next item. While maintaining the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. If this is not the case you can also use a function to build a list, which maintains this order. Once we build a linked list to represent the polynomial we can perform operations like addition and multiplication. Consider the program given below.

```
/* program to add two polynomials */

typedef struct node
{
    float coeff;
    int exp;
    struct node *link;
} PNODE;

void p_append(PNODE **, float, int);
void p_addition(PNODE *, PNODE *, PNODE **);

main()
{
    PNODE (first, *second, *total;
    int i = 0;

    first = second = total = NULL; /* empty linked lists */

    p_append(&first, 1, 4, 5);
    p_append(&first, 1, 5, 4);
    p_append(&first, 1, 7, 2);
    p_append(&first, 1, 8, 1);
    p_append(&first, 1, 9, 0);

    clrscr();
    display_p(first);

    p_append(&second, 1, 5, 6);
    p_append(&second, 2, 5, 5);
```

```

        p_append(&second,-3,5,4);
        p_append(&second,4,5,3);
        p_append(&second,6,5,1);

        display_p(second);

        p_addition(first,second, &total)

        display_p(total);
};

```

The function to append the polynomial `p_append()` and `display_p()` are similar to our functions for singly linked list. So they are expected to be written by the user. The function to add two polynomials is given below.

```

void p_addition(PNODE *x, PNODE *y, PNODE **s)
{
    PNODE *z;

    /* if both lists are empty */

    if( x == NULL && y == NULL )
        return;

    /* traverse till one node ends */

    while( x != NULL && y != NULL )
    {
        if ( *s == NULL )
        {
            *s = malloc(sizeof(PNODE));
            z = *s;
        }
        else
        {
            z->link = malloc(sizeof(PNODE));
            z = z->link;
        }

        /* store a term of larger degree if polynomial */
        if( x->exp < y->exp )
        {
            z->coeff = y->coeff;

```

```

        z->exp = y->exp;
        y = y->link; /* goto the next node */
    }
    else
    {
        if( x->exp > y->exp)
        {
            z->coeff = x->coeff;
            x->exp = x->exp;
            x = x->link; /* goto the next node */
        }
        else
        {
            if( x->exp == y->exp)
            {
                z->coeff = x->coeff + y->coeff;
                x->exp = x->exp;
                x = x->link; /* goto the next node */
                y = y->link; /* goto the next node */
            }
        }
    }
}

```

/\*assign remaining elements of the first polynomial to the result \*/

```

while( x != NULL)
{
    if( *s == NULL)
    {
        *s = malloc(sizeof(PNODE));
        z = *s;
    }
    else
    {
        z->link = malloc(sizeof(PNODE));
        z = z->link;
    }
    z->coef = x->coef;
    z->exp = x->exp;
    x = x->link;
}

```

```
/*assign remaining elements of the second polynomial to the  
result */
```

```
while( y != NULL)  
{  
    if( *s == NULL)  
    {  
        *s = malloc(sizeof(PNODE));  
        z = *s;  
    }  
    else  
    {  
        z->link = malloc(sizeof(PNODE));  
        z = z->link;  
    }  
    z->coef = y->coef;  
    z->exp = y->exp;  
    y = y->link;  
}  
  
z->link = NULL; /* at the end of list append NULL*/  
  
}
```

In this program two polynomials are built and pointed by the pointers first and second. Next the function `p_addition()` is called to carry out the addition of these two polynomials. In this function the linked lists representing the two polynomials are traversed till the end of one of them is reached. While doing this traversal the polynomials are compared on term-by-term basis. If the exponents of the two terms being compared are equal then their coefficients are added and the result is stored in the third polynomial. If the exponents are not equal then the bigger exponent is added to the third polynomial. During the traversal if the end of one list is reached the control breaks out of the while loop. Now the remaining terms of that polynomial are simply appended to the resulting polynomial. Lastly the result is displayed.

### Exercises:

1. WAP for adding and deleting nodes from an ascending order linked list.
2. WAP to reverse a singly linked list by adjusting the links.
3. Write programs for reversing a doubly linked list (though it does not serve any purpose it will give practice to manipulate the pointers).
4. WAP to delete a node after the specified node and before a specified node using both singly and doubly linked lists.

5. WAP to break a linked list into two linked lists using both SLL and DLL.
6. Write a program to add two polynomials using a DLL.
7. WAP to multiply two polynomials for both SLL and DLL.
8. WAP to add two long integers. Each integer may contain 15 to 20 digits, which can be stored in nodes, a digit each or more depending on the users choice. Add these long integers (from least significant digit backwards) and display the resultant list.

## Session 6

### Graphs

*During this session you will learn about:*

- 1 . *Graphs.*
- 2 . *Adjacency matrix and lists.*
- 3 . *Breadth search and depth search on graphs.*
- 4 . *Some important functions and definitions regarding graphs.*

#### 6.1. Introduction

*A graph is a collection of vertices and edges,  $G = (V, E)$  where  $V$  is set of vertices and  $E$  is set of edges. An edge is defined as pair of vertices, which are adjacent to each other. When these pairs are ordered, the graph is known as *directed graph*.*

These graphs have many properties and they are very important because they actually represent many practical situations, like networks. In our current discussion we are interested on the algorithms which will be used for most of the problems related to graphs like to check connectivity, the depth first search and breadth first search, to find a path from one vertex to another, to find multiple paths from one vertex to another, to find the number of components of the graph, to find the critical vertices and edges.

The basic problem about the graph is its representation for programming.

## 6.2. Adjacency Matrix and Adjacency lists

We can use the adjacency matrix, i.e. a matrix whose rows and columns both represent the vertices to represent graphs. In such a matrix when the  $i^{\text{th}}$  row,  $j^{\text{th}}$  column element is 1, we say that there is an edge between the  $i^{\text{th}}$  and  $j^{\text{th}}$  vertex. When there is no edge the value will be zero. The other representation is to prepare the adjacency lists for each vertex.

Now we will see an example of a graph and see how an adjacency matrix can be written for it. We will also see the adjacency relations expressed in form of a linked list.

For Example:

**Fig 1 Graph**

The adjacency matrix for representing this graph is:

	V1	V2	V3	V4	V5	V6	V7	V8
V1	0	1	0	0	1	1	0	0
V2	1	0	0	1	1	0	0	0
V3	0	0	0	1	1	0	0	1

V4	0	1	1	0	0	0	0	0
V5	1	1	1	0	0	1	1	1
V6	1	0	0	0	1	0	0	0
V7	0	0	0	0	1	0	0	0
V8	0	0	1	0	1	0	0	0

**Fig 2. Adjacency Matrix representation of graph in fig 1**

Adjacency list will be:

```

V1 -> V2 -> V5 -> V6
V2 -> V1 -> V4 -> V5
V3 -> V4 -> V5 -> V8
V4 -> V2 -> V3
V5 -> V1 -> V2 -> V3 -> V6 -> V7 -> V8
V6 -> V1 -> V5
V7 -> V5
V8 -> V3 -> V5

```

**Fig 3. Adjacency List representation of graph in fig 1**

### 6.3. Breadth first search and Depth first search

Suppose the graph is represented as an adjacency matrix, and we are required to have the breadth first search of the graph. Here we will require the starting vertex from which this search will begin. First that vertex will be printed, then all the vertices, which are adjacent to it, are printed and so on.

If we have a matrix and there are  $n$  vertices. Let the starting vertex be  $j$ . Now the  $j^{\text{th}}$  vertex will be printed first, and to find all the vertices adjacent to this vertex, we must travel along  $j^{\text{th}}$  row of the matrix, and whenever we find 1 we will print the corresponding column number. Next time we will require each of the vertices printed recently so that we can travel level by level. For the same purpose we will push into a queue all the columns with the value one in the  $j^{\text{th}}$  row. Next time pop the vertex number from the queue and print it. Replace the value of  $j$ , which is currently printed. Again push all the vertices that are adjacent to this vertex into the queue and continue the above process until all the vertices are dealt with.



Remember there will be many vertices, which will be connected to more than a vertex, and therefore there are chances that we may repeat some of the vertices or there will be an infinite loop. To avoid this problem, we use what is known as visited array. It will be initially all zeroes. Whenever a vertex is pushed into the queue the corresponding position the visited array is changed to 1. Now we use another rule that we push only those vertices into the queue whose corresponding value in the visited array at that point is zero. When all the vertices are printed we will stop. Sometimes it happens that a particular vertex or a group of vertices is non reachable from the current vertex and in this case the graph is ‘*not connected*’.

*Therefore a connected graph is the one in which we can travel through all the vertices starting from a current node.*

Algorithm for the breadth first traversal in a graph:

- 5 . Initialize the adjacency matrix p to all zeroes.
- 6 . Accept the number of vertices from the user, say n.
- 7 . Initialize the visited array v to all zeroes.
- 8 . Accept the graph.
  - a. Initialize i to 0.
  - b. Accept the vertex adjacent to ith vertex, say j.
  - c. Make  $p[i][j] = p[j][i] = 1$ , as they are adjacent to each other.
  - d. If more vertices are adjacent to ith vertex then goto step (a).
  - e. Now consider the next vertex i.e. increment i and repeat from step(a).
- 9 . Accept the starting vertex say i.
- 10 . Push i to the queue, and mark it as visited, i.e.  $v[i]=1$ .
- 11 . Pop a vertex from queue, say i.
- 12 . Print i.
- 13 . Search in ith row for 1,
  - f. Initialize j to 0.
  - g. If the jth vertex is adjacent to i and not visited i.e.  $(p[i][j]==1 \ \&\& \ v[j] \neq 1)$ , push j to the queue and mark it as visited, i.e.  $v[j]=1$ .
  - h. Increment j and if the number of vertices is not over then repeat from step(b).
- 14 . If queue is not empty repeat from step 7.
- 15 . Now check whether all vertices are visited.
  - i. Initialize j=0 and flag = -1.
  - j. If jth vertex is not visited, set flag to j.
  - k. Increment j, and if number of vertices is not over, repeat from step b.

- 16 . If flag  $\neq -1$ , the graph is not a connected graph. Otherwise it is a connected graph.
- 17 . Stop.

The algorithm will give us a clear idea about connectedness of the graph. Here we are accepting the graph as the adjacency list.

The above algorithm can be changed for the linked lists as below.

- 18 . Accept the number of vertices from the user , say n.
- 19 . Create a list having n nodes. This list is called as a header list. It will be connected by the down pointer where as the adjacency list will be connected by the next pointer. Remember that these two lists will follow different structures.
- 20 . Initialize the visited array v to zeroes.
- 21 . Accept the graph.
  - l. Accept an edge say i,j.
  - m. Search in the header list for vertex i, and in the adjacency list of that vertex i, attach a node of vertex j.
  - n. Search in the header list for vertex j, and in the adjacency list of that vertex j, attach a node of vertex i.
  - o. If more edges then goto step (a).
- 22 . Accept the starting vertex say i.
- 23 . Push i to the queue, and mark it as visited, i.e.  $v[i]=1$ .
- 24 . Pop a vertex from queue, say i.
- 25 . Print i.
- 26 . Move in the adjacency list of ith vertex, and for every node say j which is not visited, push it to the queue and mark it as visited, i.e.  $v[j]=1$ .
- 27 . If queue is not empty repeat from step 7.
- 28 . Now check whether all vertices are visited.
  - a. Initialize  $j=0$  and flag = -1.
  - b. If jth vertex is not visited, set flag to j.
  - c. Increment j, and if number of vertices is not over, repeat from step b.
- 29 . If flag  $\neq -1$ , the graph is not a connected graph. Otherwise it is a connected graph.
- 30 . Stop.

*For depth first search, the algorithm is same as breadth first but in place of queues we have to use stacks.*

Consider the following graph,

**Fig 4. Graph**

The adjacency matrix will be

	0	1	2	3	4	5	6
0	0	1	0	1	0	0	0
1	1	0	1	1	1	0	0
2	0	1	0	0	1	1	0
3	1	1	0	0	0	0	1
4	0	0	1	0	0	1	1
5	0	0	1	0	1	0	1
6	0	0	0	1	1	1	0

**Fig 5. Adjacency Matrix representation of graph in fig 4**

The adjacency list will be:

X = NULL

### **Fig 6. Adjacency List representation of graph in fig 4**

The figure 5 shows how the graph is stored using the matrices and figure 6 shows it stored as adjacency list. The logic of both the algorithms remain same but the change in representation is due to the data structure that we are using. The data structure will always provide some additional features and facilities applicable to a particular problem. When we try to utilize these facilities the algorithm is bound to change.

Observe that in the above two cases when we are using the arrays, we have a simple representation but once the graph is inputted, to check the adjacency we have to check again whether a particular position contains a zero or one. This check is not required when we are using the linked list. Here the vertices which are adjacent to a particular vertex are available directly and can be used without any checks.

While creating the adjacency list, observe that we are required to check for the appropriate position in the header list and then only we can insert the node. In case of arrays we simply place the elements in the  $i$ th row and  $j$ th column, there is no check involved in the process.

In both the methods, we have used the same array visited. This is because, we are assuming that the vertices are given numbers and not names. If we use names to refer vertices then we need a linked list to store the status of vertices. Here before inserting any vertex into the queue the whole visited list has to be scanned, as there is no direct way of getting the information that whether a particular vertex is visited.

As stated earlier, observe that the header list as well as the adjacency lists have different structures. These structures are given below:

```
struct adj_node
```

```

{
    int ver;
    struct adj_node *next;
}

struct head_node
{
    int ver;
    struct head_node *down;
    struct adj_node *next;
}

```

In the creation of the lists we find that we are required to have checks for searching the vertex in the head list. Do not have the misconception as the headlist will contain all the nodes in a sorted fashion. The headlist will also get created simultaneously. Whenever a new vertex is received, it will be inserted in the head list. Here we are required to keep track of the additions as well as search and traversals in both the types of the lists.

Sometimes a combination of data structures will give us a better algorithm suitable for the current application. In fact we should have array for header nodes and not the list. By this we can reduce the initial searching time and directly go to a particular adjacency list.

The following program will read the graph, given by the user in the form of edges. A proper list is formed as shown in the previous figure. The aim of the program is to print the breadth first and depth first search of the given graph.

The logic is implemented by the use of stack or queue, which are in turn implemented using, linked lists.

This program can also be considered as a good example for handling of the multiple lists. Here we are handling the adjacency list corresponding to all the vertices.

While writing the complete program you are expected to write the following functions yourself. The functions use linked list representation for queues and stacks.

1. Function to create the queue.  
Q create();
2. Function To insert an element into the queue.  
void push1 (char data, Q head);
3. Function To delete an element from the queue.

```

        char pop1 ( Q head);

6. Function to check whether the queue is empty.
    int q_empty(Q head);

7. Function to create a stack.
    STK createst();

8. Function to check whether the stack is empty.
    int stk_empty(STK head);

9. Function To push an element into the stack.
    void push(char data, STK h);

10.Function To pop an element from the stack.
    char pop ( STK h);

/* The graph representation starts here */

typedef struct verlist
{
    char vertex;
    struct verlist *right;
}*VR;          /* for adjacency list */

typedef struct hlist
{
    int flag
    char vertex;
    struct verlist *right;
    struct hlist *down;
}*VH;          /* the list of all vertices */

VH create1()
{
    VH header;

    header = malloc(sizeof(struct hlist));
    header->down = header->right = NULL;
    return(header);
}
/* function to find the vertex c, in the header list */

```

```

VH find(char c, VH header)
{
    VH tmp;
    tmp = header;

    do
    {
        if(tmp->vertex == c)
            return tmp; /* returns pointer to the header node */
        else
            tmp= tmp->down;
    }while (tmp);

    return(tmp); /* returns NULL, when absent */
}

```

```

VR get_ver(char c2)
{
    VR new1;

    new1 = malloc(sizeof(struct verlist));
    new1->vertex=c2; /* generating a node for adj. List */
    new1->right = NULL;
    return(new1);
}

```

```

VH get_ch(char c)
{
    VH new2;

    new2 = malloc(sizeof(struct hlist));
    new2->vertex = c; /* generating a node for header List */
    new2->right = NULL;
    new2->down = NULL;
    new2->flag = 0;

    return(new2);
}

```

/\* function to display the adjacency list \*/

```

void display(CH header)
{
    VH ttmp;
    VR t;

    ttmp = header -> down;
    while(ttmp)
    {
        printf("%c ==> ", ttmp->vertex);
        t = ttmp->right;

        while(t)
        {
            printf("%c □"t->vertex);
            t = t->right);
        }
        printf("NULL\n");
        ttmp = ttmp->down;
    }
}

```

```

void search(char c1, char c2, VH header)
{
    VH ttmp, new1;
    VR new2;

    ttmp = header;

    while(ttmp->down != NULL && ttmp->down->vertex != c1)
        ttmp = ttmp->down; /*search node c1 in the header
                             list */

    if(ttmp->down->vertex == c1)
    {
        new2 = get_ver(c2); /* generate a node for c2 and add
                             to the list */
        new2->right = ttmp->down->right;
        ttmp->down->right = new2;
    }
    else
    {
        new1 = get_vh(c1); /* if header list does not contain
                             node for c1 */
    }
}

```



```

        new1->down = tmp->down;
        tmp->down = new1;
        new1->flag = 0;
        tmp=new1;
        new2=get_ver(c2);
        new2->right = tmp->right;
        tmp->right=new2;
    }
}

```

/\* function for BREADTH FIRST SEARCH \*/

```

void bfs(VH header)
{
    char c1, ans[30];
    int k=0, i=0;
    VR t;
    VH tmp;
    Q head1, hh;

    head1 = create();
    tmp = header->down;

    while(tmp)
    {
        tmp->flag = 0;
        tmp = tmp->down;
    }

    tmp = header->down;

    printf("Enter the character you want to start from\n");
    fflush();

    c1=getchar();

    do
    {
        tmp = find(c1,header);
        printf("%c\n", tmp->vertex);
        ans[i++]= tmp->vertex;
    }
}

```

```

    tmp->flag = 1;
    t = tmp->right;

    while(t)
    {
        tmp= find(t->vertex, header);

        if(tmp->flag == 0)
        {
            push1(t->vertex, head1);
            printf("\n %c vertex is pushed in the queue \n", t->vertex);
            tmp->flag = 1;
            getch();
        }
        t = t-> right;
    }
    printf(" now the queue is ... \n");
    hh= head1->next;
    printf("-----\n");

    while(hh)
    {
        printf("%c\t", hh->val);
        hh = head->next;
    }

    printf("\n ----- \n");
    getch();

    k = q_empty(head1);

    if(!k)
    {
        printf("\n Removing the character from the queue ...");
        c1 = pop1(head1);
    }
} while(!k);

ans[i] = '\0';
printf("THE BFS IS:\n");

for(i=0; ans[i] != '\0'; i++)
    printf("%c\t",ans[i]);
}

```

```
/* function for DEPTH FIRST SEARCH */
```

```
void dfs(VH header)
```

```
{  
    char c1, ans[30];  
    int k=0, i=0;  
    VR t;  
    VH tmp;  
    STK head1, hh;
```

```
    head1 = create();  
    tmp = header->down;
```

```
    while(tmp)  
    {  
        tmp->flag = 0;  
        tmp = tmp->down;  
    }
```

```
    tmp = header->down;
```

```
    printf("Enter the character you want to start from\n");  
    fflush();
```

```
    c1=getchar();
```

```
    do  
    {
```

```
        tmp = find(c1, header);  
        printf("%c\n", tmp->vertex);  
        ans[i++] = tmp->vertex;  
        tmp->flag = 1;  
        t = tmp->right;
```

```
        while(t)  
        {
```

```
            tmp = find(t->vertex, header);
```

```
            if(tmp->flag == 0)  
            {
```

```
                push(t->vertex, head1);  
                printf("\n %c vertex is pushed in the stack \n", t->vertex);
```

```

                                tmp->flag = 1;
                                getch();
                            }
t = t-> right;
}
printf(" now the stack is ...\n");
hh= head1->next;
printf("-----\n");

while(hh)
{
    printf("%c\t", hh->val);
    hh = head->next;
}

printf("\n ----- \n");
getch();

k = stk_empty(head1);

if(!k)
{
    printf("\n Removing the character from the stack ...");
    c1 = pop(head1);
}
}while(!k);

ans[i] = '\0';
printf("THE DFS IS:\n");

for(i=0; ans[i] != '\0'; i++)
    printf("%c\t",ans[i]);
}

/* MAIN */

main()
{
    VH header;
    char c1,c2;
    int choice;

    header = create1();

```

```

do
{
    printf("WHICH CHARACTERS ARE ADJACENT \n");
    fflush();
    c1 = getchar();
    fflush();
    c2 = getchar();

    search(c1,c2,header);
    search(c2,c1,header);

    printf("DO YOU WANT TO CONTINUE ? \n");
    fflush();
} while(getchar() == 'y');

display(header);

do
{
    printf("\n\n ***** menu *****\n");
    printf("\n ENTER YOUR CHOICE\n");
    printf("1:***BFS***\n");
    printf("2:***DFS***\n");
    printf("3:QUIT\n");
    printf("*****\n");

    scanf("%d", &choice);

    switch(choice)
    {
        case 1: bfs(header);
                break;

        case 2: dfs(header);
                break;

    }
} while(choice != 3);
}

```

Output :

WHICH CHARACTERS ARE ADJACENT?

a b

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

a c

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

a d

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

b c

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

b e

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

e f

DO YOU WANT TO CONTINUE?

y

WHICH CHARACTERS ARE ADJACENT?

f g

DO YOU WANT TO CONTINUE?

n

a ==> d--> c --> b --> NULL

b ==> e--> c --> a --> NULL

c ==> b--> a --> NULL

d ==> a --> NULL

e ==> f --> b --> NULL

f ==> g --> e --> NULL

g ==> f --> NULL

\*\*\*\*\* MENU \*\*\*\*\*

ENTER YOUR CHOICE

1.\*\*\*BFS\*\*\*

2.\*\*\*DFS\*\*\*

3:QUIT

\*\*\*\*\*

1

ENTER THE CHARACTER YOU WANT TO START FROM

a

a

d vertex is pushed in the queue

c vertex is pushed in the queue

b vertex is pushed in the queue

Now the queue is ...

-----

d c b

-----

Removing the character from the queue ... d

Now the queue is ...

-----

c b

-----

Removing the character from the queue ... c

Now the queue is ...

-----

b

-----

Removing the character from the queue ... b

e vertex is pushed in the queue

Now the queue is ...

-----

e

-----

Removing the character from the queue ... e

f vertex is pushed in the queue

Now the queue is ...

-----

f

-----

Removing the character from the queue ... f

g vertex is pushed in the queue

Now the queue is ...

-----

g

-----

Removing the character from the queue ... g

Now the queue is ...

-----

-----

THE BFS IS ...

a d c b e f g

\*\*\*\*\* MENU \*\*\*\*\*

ENTER YOUR CHOICE

1.\*\*\*BFS\*\*\*

2.\*\*\*DFS\*\*\*

3:QUIT

\*\*\*\*\*

2

ENTER THE CHARACTER YOU WANT TO START FROM

a

a

d vertex is pushed in the stack

c vertex is pushed in the stack

b vertex is pushed in the stack

Now the stack is ...

-----



b c d

-----

Removing the character from the stack ... b

e vertex is pushed on the stack

Now the stack is ...

-----

e c d

-----

Removing the character from the stack ... e

f vertex is pushed on the stack

Now the stack is ...

-----

f c d

-----

Removing the character from the stack ... f

g vertex is pushed in the stack

Now the stack is ...

-----

g c d

-----

Removing the character from the stack ... g

f vertex is pushed in the stack

Now the stack is ...

-----

c d

-----

Removing the character from the stack ... c

Now the stack is ...

-----

d

-----  
Removing the character from the stack ... d

Now the stack is ...  
-----  
-----

THE DFS IS ...

a b e f g c d

\*\*\*\*\* MENU \*\*\*\*\*

ENTER YOUR CHOICE

1.\*\*\*BFS\*\*\*

2.\*\*\*DFS\*\*\*

3:QUIT

\*\*\*\*\*

3

The purpose of the above program is to make the header familiar with the generation concepts for the linked lists as well as the use of stacks and queues for the operations on the graph.

The output is self-explanatory. As you read the output of the program, you will understand the total procedure or the logic to get the BFS or DFS of the graph.

#### **6.4. Other Tasks For The Graphs:**

Some other functions, which are associated with graph for solving the problems are:

##### **31 . To find the degree of the vertex**

The degree of the vertex is defined as a number of vertices which are adjacent to given vertex, in other words, it is number of 1's in the row of that

vertex in the adjacency matrix or it will be number of nodes present in the adjacency list of that vertex.

### **32 . To find the number of edges.**

By hand shaking lemma , we know that the number of edges in a graph is half of the sum of degrees of all the vertices.

### **33 . To print a path from one vertex to another.**

Here we are required to follow the above algorithm of BFS such that one of the vertices is the starting vertex for the algorithm and the process will continue till we reach the second vertex.

### **34 . To print the multiple paths from one vertex to another.**

The previous algorithm should be used in some different form so that we can get multiple paths.

### **35 . To find the number of components in a graph.**

In this case we will again use the BFS, and check the visited array, if it does not contain all the vertices marked as visited then increment the component counter by 1 and from any of the vertex which is not visited, restart the BFS. Repeat till all the vertices are visited.

### **36 . To find the critical vertices and edges.**

The vertex which when removed from the graph, leaves the graph as disconnected, will be termed as critical vertex. To find the critical vertex we should first remove each vertex and check the number of components of the remaining graph. If the graph, which is remaining, is not a connected graph, the vertex, which is removed, is a critical vertex.

Similarly removal of an edge from the graph, if increases the number of components, it will be known as critical edge. If we try to check whether a particular vertex or edge is critical, then remove the same and rerun the program for finding the number of components.

### Exercise:

37. WAP to accept the graph from the user along with weight attached to each edge. Accept a path from the user which is in the form of sequence vertices and we are required to print the weight of that path.

### Session 7

## Trees

### (Part 1)

#### During this session you will learn about

- *Trees.*
- *Rooted tree and binary trees.*
- *Binary search tree.*
- *Traversals of the trees.*
- *Preorder Traversal – recursive and iterative.*
- *Postorder Traversal – recursive and iterative.*
- *Inorder Traversal – recursive and iterative.*
- *Operations on trees.*

### 7.1. Introduction

Until now we have seen the data structures, which were basically connected linearly. But many times we are required to have two more paths from the correct 'object'. It is not the linear traversal but there will be multiple choices. The data object could be connected to more than 2 data object.

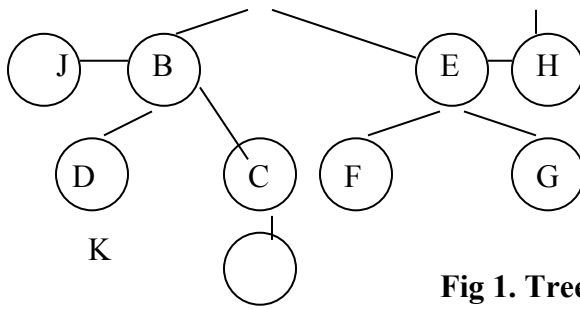
*A **Graph** is defined as a collection of vertices and edges  $GE (V, E)$ , where  $V$  is set of vertices and  $E$  is set of edge. An **Edge** is defined as pair of vertices, which are connected. If the edge has direction, the graph is 'directed graph' and edge will be ordered pair of vertices.*

It is also possible to travel from one vertex to other vertices then the graph is known as connected graph. If the path that we follow takes us back to start vertex, then we say that there exists a *cycle* or a *closed path* or *circuit*.

*A **TREE** is defined as a connected graph without a circuit.*

e.g.





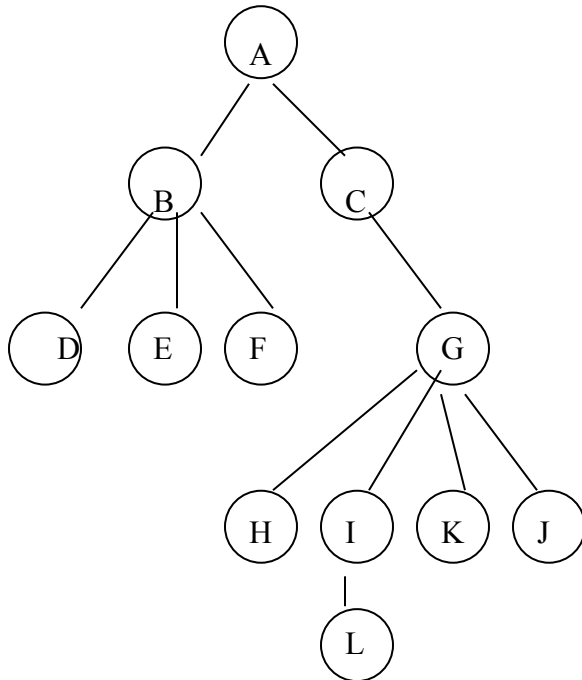
**Fig 1. Tree**

## 7.2. Rooted Tree

A tree is which one vertex is distinguished from others and called as **Root** is known as a **Rooted Tree**.

If we consider the tree to be a directed graph, then every vertex will have incoming degree as well as outgoing degree. Incoming degree is the number of nodes that consider this node as terminal node. Outgoing degree is the number of nodes consider this node as the initial node. **Degree** is the number of edges incident as a vertex. i.e. it is the sum of incoming degree and the outgoing degree. Observe that the root is a vertex having incoming degree zero.

e.g.



**Fig 2. Rooted Tree**

	In	Out
A	0	2
B	1	3
C	1	1
D	1	0
E	1	0
F	1	0
G	1	4
H	1	0
I	1	0
J	1	0
K	1	0
L	1	0

**Fig 3. Incoming and outgoing**

## vertices of tree in fig 2

Let us take a look at some other properties of trees.

1. A connected graph on vertices having (1-1) edges is known as TREE.
2. A graph in which there is a unique path between any pair of vertices is a TREE.
3. It is a minimally connected graph.

Observe that all the other vertices have the incoming degree as 1. When incoming degree is more than 1, we can say that there is more than one way to reach the vertex and it will not be a tree. Also observe that,

$$\begin{aligned}\text{sum of incoming degrees} &= \text{sum of outgoing degrees.} \\ &= \text{number of edges} \\ &= \text{number of vertices}-1.\end{aligned}$$

$$\begin{aligned}\square \text{ incoming degrees} &= 11. \\ \square \text{ outgoing degrees} &= 2 + 3 + 1 + 0 + 0 + 0 + 4 + 0 + \\ &\quad 0 + 0 + 4 + 0 \\ &= 11.\end{aligned}$$

The node with incoming degree 'zero' is a Root node. i.e. A will be the 'ROOT'. The nodes with outgoing degree 'zero' are known as 'leaf nodes' e.g., D,E,F,H,L,J.

From a node if the outgoing edges are reaching the vertices v1,v2, ..etc. then v1,v2 etc. will be children of that node e.g. B and C are children of A and L is the child of K.

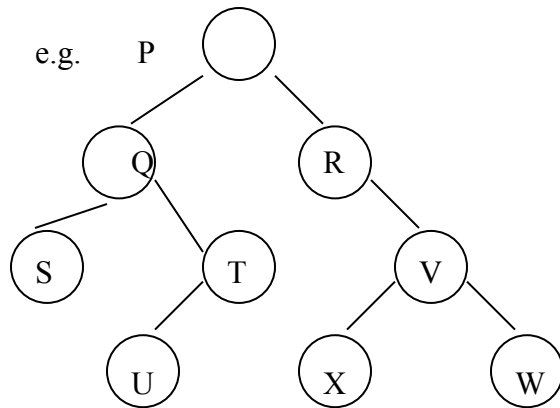
### 7.3. Binary Tree

*Binary Tree is a rooted tree in which root has maximum two children such that each of them again is a binary tree.*

The definition is a *recursive definition* as we use the words 'Binary Tree' to define the binary Tree. We also say that a '*NULL TREE*' is a Binary Tree.

In Binary Tree, the outgoing degree of each vertex can be maximum two. The children are identified as left child and right child respectively.

The tree in the previous diagram is definitely not a Binary Tree because B and G have more than two children. It is not necessary that nodes should have two children. We may have a binary tree in which none of the nodes have two children.



**Fig 4. Binary Tree**

The structure used for defining the nodes for a Binary Tree is shown below:

```

struct bin_tr
{
int data;
struct bin_tr *left,* right;
}
  
```

Here we will have data field of any type and size, and two pointers are used to point to the left child and right child respectively. This structure is again dynamic in nature and there is no limitation on number nodes, and we can go on building the tree in any form. When the tree is not required we can free all the nodes so that the memory can be utilized for some other process.

We will use a header node in case of a binary tree. It is not compulsory but it helps us to write easier algorithms. Therefore we will use this concept of header nodes in our discussions. The root of the tree will be attached to this node. It can be connected to the header's left or right. It will be the programmer's decision. Of course the header does not contain any data.

## 7.4 Creation of Binary Tree

As we have previously created the linked list, we are aware of procedure for creation of a list. During the creation we are required to create new locations or nodes, enter the data into them and set the proper links so that the path will be set to access the data. In case of linked list it was easy because at particular position of the list, we have only one way or one pointer where the new node can be attached. But in case of binary trees at any node we have two choices to attach the node. Hence while creating the tree,

we will have to ask the user as where a particular node should be attached, to the left or to the right.

Thus we can write the algorithm for creating the binary tree keeping this concept in mind. The algorithm for the creation of the tree is as follows

1. Create a node say new\_node.
- Set temp to header.
- Check whether the temp has left child if so, new\_node will be attached as left soon to temp. If temp has left child, it indicates that the tree is present, we move to the left child of temp. i.e. set temp to temp's left.
- Display the value of temp's data and ask whether the new\_node should be attached to left or right of temp. If the respective child exists then move to that child otherwise attach the new\_node as the child requested by the user.
- Repeat step 4, till new\_node gets attached.
- Ask the user whether there are more values? If yes then go to step 1.
7. Stop. Creation is over.

For this algorithm we assume that the header node has been created before we begin creation of the tree. Also we will write a function to generate a new\_node.

```
struct bin_tr * get_new_node
{
    struct bin_tr * temp;
    temp =(struct bin_tr &)malloc(size of (struct bin_tr));
    temp->left = NULL;
    temp->right = NULL;
    return (temp);
}
```

**Suppose we declare a new type for the above structure as BITR, which is a pointer type then the function could be rewritten as**

```
typedef struct bin_tr * BITR;

BITR get_new_node()
{
    BITR temp;
    temp =(BITR)malloc(sizeof(struct bin_tr ));
    temp->left = temp->right = NULL;
    return (temp);
}
```



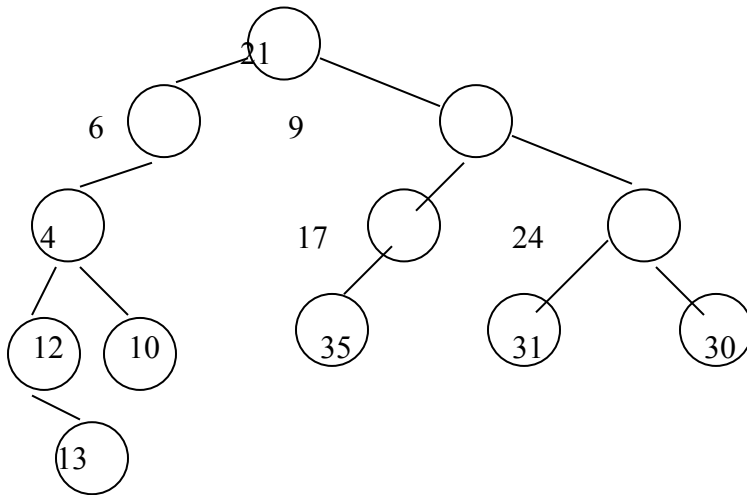
With this we will write the create function.

```
void create_tr(BITR h)
{
    BITR temp, new_node;
    int chk;
    char c;
    do
    {
        new_node = get_new_node();
        chk = 0;
        temp = h;
        scanf("%d", &new_node->data);
        if (temp->left != NULL)
            temp = temp->left;
        else
            temp->left = new_node;
        while (chk == 0)
        {
            printf ( " The current node is %d\n ", temp->chk);
            printf ( "Whether the new node should be attached
                to left or right?\n");
            c = getch();
            if (c == 'L')
                if (temp->left = NULL)
                    temp = temp->left;
                else
                {
                    temp->left = new_node;
                    chk = 1;
                }
            else
                if (temp->right = NULL)
                    temp = temp->right;
                else
                {
                    temp->right = new_node;
                    chk = 1;
                }
        }
        printf ( " Any more node? \n");
        c = getch ( );
    } while ( c == 'y');
}
```

Observe that the tree which we are creating is as given by the user. Every time for attaching a new\_node, we come from the header, asking whether to attach to left or right.

Though the trees get created successfully, while displaying them, we will have to face a lot of trouble.

If the generated tree is



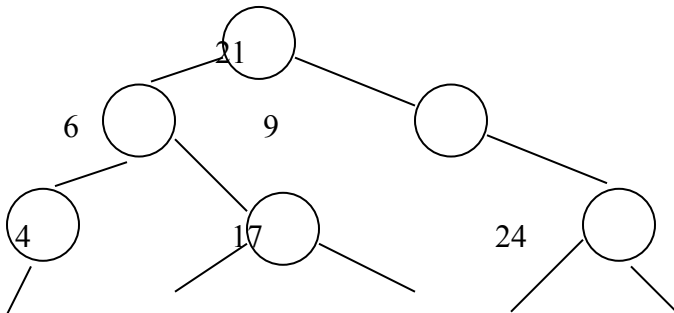
**Fig 5. Binary Tree**

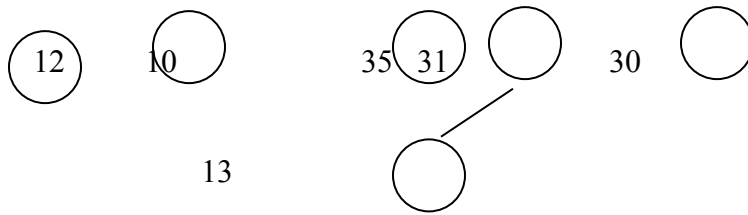
Normally we prefer to print the tree in level wise format as shown:

```

21
6    9
4    17
12   10   35   31   30
13
  
```

But this output does not give us the correct idea of the tree. The same output will be generated even for the following tree.





**Fig 6. Binary Tree**

Another way would be rather descriptive

Node	21	has	leftchild	6,	rightchild	9
Node	6	has	leftchild	4,	rightchild	NIL
Node	9	has	leftchild	17,	rightchild	24
Node	4	has	leftchild	12,	rightchild	10
Node	17	has	leftchild	35,	rightchild	NIL
Node	24	has	leftchild	31,	rightchild	30
Node	12	has	leftchild	NIL,	rightchild	3
Node	10	has	leftchild	NIL,	rightchild	NIL
	:					
	:					

Observe that the sequence of the nodes is still level wise but assertion information will give unique tree.

## 7.5 Breadth First Search

This is (level-wise printing) also known as the Breadth First Search BFS. To implement it, following algorithm will be useful,

Steps :

1. temp will point to header's left.
2. Use queue for the operation.
3. push temp in the queue.
4. pop a node from the queue say temp.
5. print the data of node temp.
6. if temp has left child, push it into the queue.
7. if temp has right child, push it into the queue.
8. repeat the process from step 4 till the queue becomes empty.
9. Stop.

Here we are using the data structure 'queue' for the particular operation on tree. To declare the structure of the queue we will have to decide the type of data, which will go into the queue? Should it be same as data type in the node of the tree? If you think the answer is 'yes' then observe that with first push operation, the data i.e. value 21 will be pushed into the queue. Then we pop the value and print it. But if we want to go to the left child then remember that the data cannot inform about the address of left child. Hence we should push the node of the tree into the queue or in other words the pointer to the node in the tree will be the data element for node in the queue.

If we are implementing the queue by array then the queue will be declared as

```
BITR que_bfs[SIZE];
```

and if we are using linked list then

```
struct que_b
{
    BITR tval;
    struct que_b *next;
}
```

With this, we should accordingly use the push and pop functions to run the program for bfs.

```
void bfs (BITR h)
{
    BITR temp;
    temp = h → left;
    push(temp);

    while(! qempty())
    {
        temp = pop();
        printf("%d/n", temp → data);

        if (temp → left != NULL)
            push (temp → left);
        if (temp → right != NULL)
            push (temp → right);
    }
}
```

This 'bfs' function will be very useful as there are many applications where 'bfs' with little additional code will give us the answers.

e.g.

To count number of nodes in the tree, we will just use bfs along with an additional counter. The counter increment statement will simply replace the printf() in above code and after the completion of loop, we may return the counter which contains the number of nodes.

```
int count_nodes (BITR n)
{
    BITR temp;
    int cnt = 0;
        :                /* rest all is same as that of bfs
        :
        :

    while(!qempty())
    {
        temp = pop( );
        cnt ++;
        :                /* same as that of bfs */
        :
    }

    return(cnt);
}
```

Suppose we want to print the nodes on each level on same line and of next level on new line. Observe the nodes will go into the queue in the following manner.

21				
6	9			
9	4			
4	17	24		
17	24	12	10	
24	12	10	35	
12	10	35	31	30
10	35	31	30	13
35	31	30	13	
31	30	13		
30	13			
3				

Observe that there is no distinction between the values dependent their levels. We should place some delimiter indication end of the level in queue, say H. Now the process will be seen as below :

21	<b>H</b>				
<b>H</b>	6	9			
6	9	<b>H</b>			
9	<b>H</b>	4			
<b>H</b>	4	17	24		
4	17	24	<b>H</b>		
17	24	<b>H</b>	12	10	
24	<b>H</b>	12	10	35	
<b>H</b>	12	10	35	31	30
12	10	35	31	30	<b>H</b>
10	35	31	30	<b>H</b>	13

Observe that whenever we pop H, the level is over and we should go to line for printing. Also this H should be pushed again into the queue. When popped and the queue is empty we should stop.

As the queue contains the pointer H, it also should be a pointer of same type but has to be treated in a special way. Let us take it as header. The function will be modified as follows:

```
void bfs_level (BITR h)
{
    BITR temp;
    temp = h □ left;
    push (temp);
    push (h);
    temp = pop();

    while (! qempty())
    {
        if ( temp != h)
        {
            printf ("%d", temp □ data);

            if ( temp □ left)
                push ( temp □ right )
            if (temp □ right);
                push (temp □ right);
        }
        else
        {
            push (temp);
            printf ("\n");
        }
    }
}
```

```

    }

    temp = pop ( );
}
}

```

Again the above function can be modified so that we can count number of nodes per levels in a tree. As we have seen before, whenever header is encountered, we understand that the level has ended. Every time when we receive the header node we will first reset the counter to zero and increment the counter till we again receive header. Print the contents of the header at that time, which represents the number of nodes in that level.

Suppose we want to search for a particular data in the tree, we can also use the breadth first search. Here we use the search, which is along the breadth of the tree. We should pass the header as well as value for searching to the function. It can return 1 to 0 or pointer or NULL dependent on presence of value.

```

int bf_search (BITR h, int val)
{
    BITR temp;
    temp = h → left;
    push (temp);

    while (!qempty())
    {
        temp = pop();
        if (temp → dat == val)
            return (1);
        if (temp → left)
            push (temp → left);
        if (temp → right)
            push (temp → right);
    }
    return ( 0 );
}

```

If we are required to count the number of leaf nodes in the tree the every time we pop a node from the queue, check whether it is a leaf node, if yes you increment count.

```

int count_leaf (BITR h)
{
    ----
    ----

    while (!qempty())

```

```

        {
            temp = pop();
            if ((!temp->left)&&(!temp->right))
                -----
        }
        return (count);
    }
}

```

In the similar way we can go for counting the nodes having only one child and nodes, which have both the children.

The program ahead gives the complete idea about creation of the tree and some other functions. The functions are modified to show a friendly display.

Here we are using the gotoxy() function by which we can place the output at the yth row and xth Column on the screen. There are in all 80 columns and 25 rows.

/\* Creation by binary tree, position defined by the user \*/

```

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

```

```

typedef struct tree
{
    int val;
    struct tree *left,* right;
}*TR;

```

TR header; /\* header for the tree is global \*/

```

typedef struct que
{
    TR node;
    struct que *next;
}*Q;

```

/\* the data in case of queue node will be pointer to the node of the tree \*/

Q headq, last;

/\* header of the queue and the last position are declared as global \*/



```

void createq()
{
    headq = malloc(sizeof(struct que));
    headq->next = NULL;
    last = headq; /* initially header itself will be the
                    end of the queue*/
}

int qempty()
{
    return(headq->next==NULL);
}

TR poppp()
{
    Q tq; /* pop function will always return the data */
    TR tn;

    tq = headq->next;
    tn = tq->node;
    headq->next = headq->next->next;
    free(tq);
    return(tn);          /* in this case the data is
                           pointer to the node of tree */
}

void push(TR n)
{
    Q newq;

    newq = malloc(sizeof(struct que));
    newq->node = n;
    newq->next = last->next;
    last->next = newq;
    last = newq;
}

TR getnode()
{
    TR r;
    r = malloc(sizeof(struct tree));
    gotoxy(20,19);
    printf("Enter the value...\n");
}

```

```

    gotoxy(20,20);
    scanf("%d", &r->val);
    gotoxy(20,19);
    printf("      ");
    gotoxy(20,20);
    printf(" ");
    r->left = r->right = NULL;
    return(r);
}

```

```

TR create()
{

```

```

    int flag, i, j, k, p, q, way;
    char c;
    TR header, temp, new;

```

```

    header = malloc(sizeof(struct tree));
    gotoxy(38,1);
    printf("HEADER ");
    header->left = NULL;

```

```

do
{
    i = 40;
    j = 2;
    k = 20;

    new = getnode();
    flag = 0;

    if (header->left == NULL)
    {
        header->left = new;
        gotoxy(i,j);
        printf("%d", new->val);
    }
    else
    {
        temp = header->left;

```

```

do
{
    gotoxy(20,19);
    printf("MOVE TO LEFT OR RIGHT (0/1)");
    gotoxy(20,20);
    scanf("%d", &way);
    gotoxy(20,19);
    printf("      ");

    if(way == 0)
    {
        gotoxy(40,22);
        printf("Moving to leftchild...");
        gotoxy(40,22);
        printf("      ");

        for( p =i; p > i-k, p--)
        {
            gotoxy(p,j+1);
            printf("%d", new->val);
            gotoxy(p,j+1);
            printf("  ");
        }

        gotoxy(i-k/2,j+1);
        printf("/");
        i = i-k;
        j = j+2;
        k = k/2;

        if(temp->left == NULL)
        {
            temp->left = new;
            printf("%d", new->val);
            flag = 1;
        }
        else
        {
            gotoxy(i,j);
            temp = temp->left;
            printf("%d", temp->val);
        }
    }
}
else

```

```

{
    for( p =i; p > i-k, p--)
    {
        gotoxy(p,j+1);
        gotoxy(40,22);
        printf("Moving to rightchild...");
        gotoxy(40,22);
        printf("      ");

        for( p =i; p > i+k, p++)
        {
            gotoxy(p,j+1);
            printf("%d", new->val);
            gotoxy(p,j+1);
            printf(" ");
        }

        gotoxy(i+k/2,j+1);
        printf("\\");
        i=i+k;
        j=j+2;
        k=k/2;

        if(temp->right==NULL)
        {
            gotoxy(i,j);
            temp->right=new;
            printf("%d", new->val);
            temp->right=new;
            flag=1;
        }

        else
        {
            gotoxy(i,j);
            temp=temp->right;
            printf("%d", temp->val);
        }
    }
} while(flag==0);
}

gotoxy(40,19);

```

```

    printf("Anymore Y/N");
    gotoxy(40,200;
    flushall();
    c = getchar();
    gotoxy(40,19);
    printf(" ");
    gotoxy(40,20);
    printf(" ");

    } while(c=='y' || c=='Y');

return(header);
}

TR leftmost(TR temp)
{
    while(temp->left!=NULL)
        temp=temp->left;
    return(temp);
}

TR rightmost(TR temp)
{
    while(temp->right!=NULL)
        temp=temp->right;
    return(temp);
}

main()
{
    TR t;
    clrscr();

    header = create();
    t = header->left;
    createq();
    push(t);
    gotoxy(10,24);

    while(!qempty())
    {
        t=poppp();
        printf("%d\t", t->val);
    }
}

```

```

        if(t->left)
            push(t->left);
        if(t->right)
            push(t->right);
    }

    printf("%d\n", t->val);
}

```

The output of the above program is not attached because the output of is a dynamic one. Using the screen coordinates, an attempt is made to show the actual creation of the tree. Here you can see how a particular tree is being built. This interactive program will be very useful who believe in the self-learning and understanding process.

In the above problem we were just creating a tree. It was a binary tree and the nodes were placed at the users wish. These nodes have the relation as parent and child but that relation purely set by the user and data in the node has just no role to play.

## 7.6. Binary Search Trees

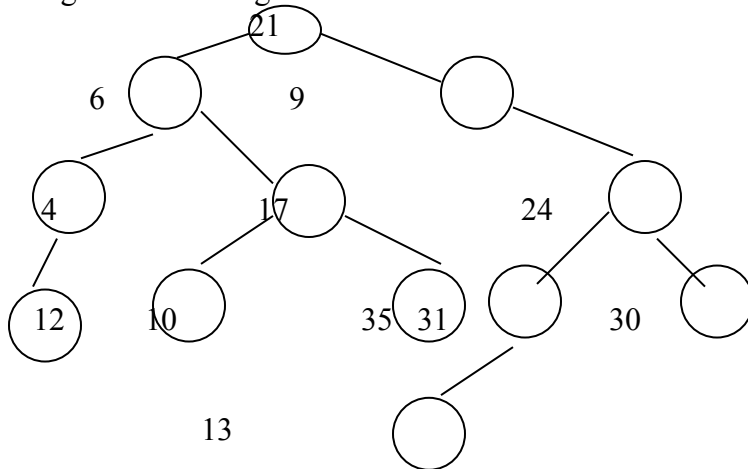
If we think of attaching the nodes in such a manner that their positions decided by the data contained in the node and also that if we are at a particular side it should be possible for us to know where a node of a particular data may be present. This is possible by using the concept of a binary search tree.

A Binary Tree in which the nodes are positioned in the left sub tree or right sub tree dependent on the value of some key, which is present in the data, is known as 'Binary Search Tree'. If the key value is less than the node value it should be placed in the left sub tree otherwise in the right sub tree.

It is known as a Search Tree, because searching for the particular value in the normal tree will be as complicated as moving to the depth of the tree. When Binary Tree has 'm' levels counting down 0,1,2,..., m, the tree can have maximum  $(2^{m+1} - 1)$  nodes. The complexity for searching in Binary Tree will be of the order of  $2^{m+1}$  where as if it is a Binary Search Tree, it will be of the order  $(m+1)$ .

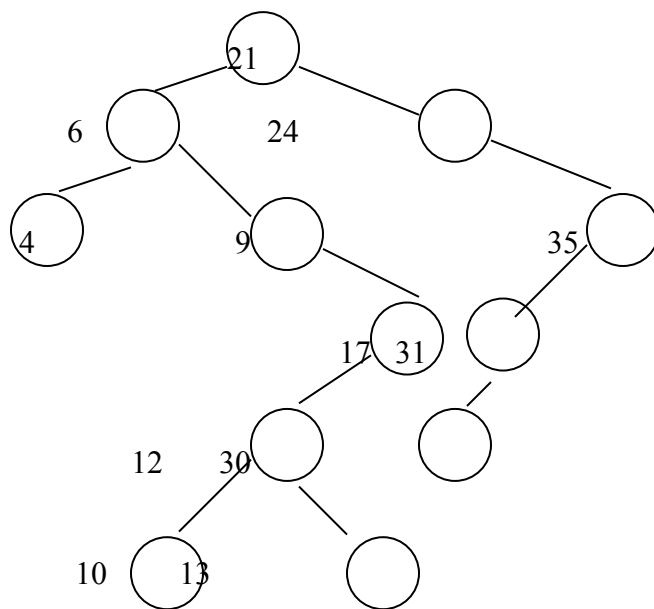
**Generally we say that once the root is created, if the value is less it should go to the left sub tree otherwise to the right sub tree. Even if we change the nature, it will still be same i.e. Binary Search Tree. Actually with the value to be searched, the current node in the tree should inform us, in some way, whether the value should be searched in the left sub tree or right sub tree of node.**

E.g. - Consider figure



**Fig 6. Binary Tree**

This as a search tree will be



**Fig 7. Binary Search Tree**

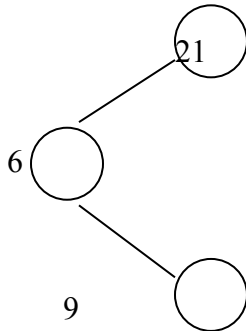
Here we do not ask the user about the position of the node in the tree but it will be placed as per the rule. The above tree is constructed from the input sequence as

21 6 9 4 17 24 12 10 35 31 30 13.

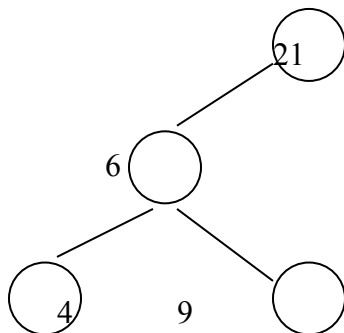
First the root : 21

Next value 6, which are, less than 21, hence it should be placed as left child.

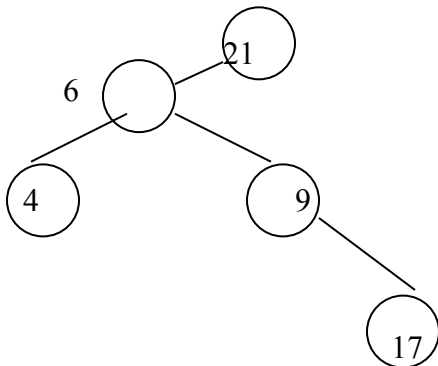
Now value 9, less than 21, hence as the left of 21. Left child exists, therefore compare with left child i.e. 6, it is greater than 6, hence will be on the right 6.



For value 4, left of 21, left of 6



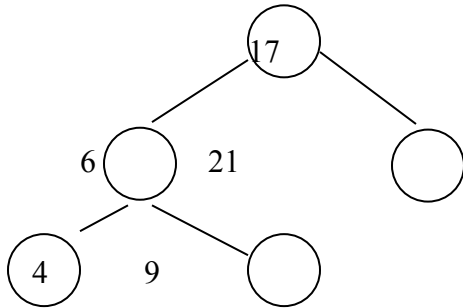
For value 17, left of 21, right of 6, right 9





Hence the sequence, in which values are received, will change the nature the tree.

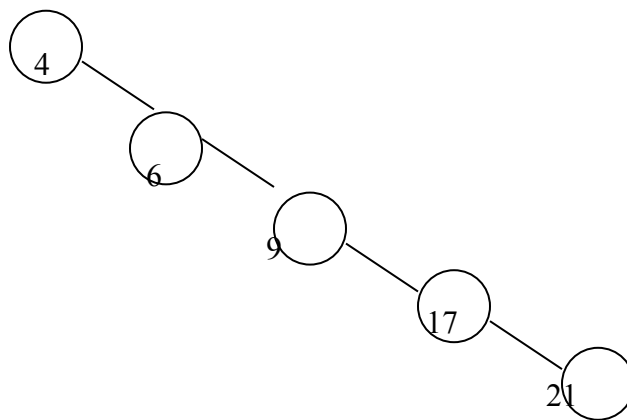
e.g. if the sequence is 17 6 9 4 21 then the tree will be



If the values are ordered say ascending then the tree will take form of

e.g.

4 6 9 17 21



### 7.7. Binary Search Tree Creation

Now let us consider the algorithm for creating a Binary Search Tree. Assuming that the first value is the root and values in left sub tree are always less than the root node whereas values in the right sub tree are greater than the root node value.

Algorithm for generating a binary search tree:

1. Create a new node.
2. Accept the value for new\_node.
3. Attach new\_node as left child of header.
4. Create a value for new\_node.
5. Accept the value for new\_node.
6. Set temp to header's left (ROOT).
7. If value of new\_node is less than temp's value then  
    if temp has left child, move temp to temp's left  
    Otherwise attach new\_node to temp's left and set the flag to 1.  
else if temp has right child, move temp to temp's right  
    otherwise attach new\_node to temp's right and set the flag to 1.
8. if flag is zero (i.e. new\_node is not yet attached) goto step 7.
9. if flag is not zero then ask the user whether there are more nodes?  
    If yes then step 4.
10. stop.

```
/* function to create a binary search tree */
void create_bst (BITR h)
{
    BITR temp, new_node;
    int flag;
    new_node = get_new_node(); /* create a new node */
    printf ("Enter the root");
    scanf ("%d", &new_node->data); /* root node */
    h->left = new_node; /* make it as the left child
                        of the header node */

    do
    {
        new_node = get_new_node ();
        printf ("Enter the value of node");
        temp = h->left;
        flag = 0;

        do
```

```

{
    if (new_node->data < temp->data) /* if val is
                                    less than the parent make it as left
                                    child */
        if (temp->left)
            temp = temp->left;
        else
        {
            temp->left = new_node;
            flag = 1;
        }
    else
        if (temp->right) /* if the new node
                        has value more than the parent make it a
                        right child */
            temp = temp->right;
        else
        {
            temp->right = new_node;
            flag = 1;
        }
    } while (flag == 0);

    printf("Any more node ? Y/N");
    } while(toupper(getch())=='Y');
}

```

The above function can be rewritten in more readable form with the help of smaller functions.

```

void create_bst ( BITR h )
{
    BITR new_node;
    h->left = get_new_node ( );
    do
    {
        new_node = get_new_node;
        insert(new_node, h->left);
        printf("Any more nodes ? Y/N");
    } while(toupper(getch()) == 'Y');
}

```

```

BITR get_new_node()
{
    BITR temp;
    temp = get_new_node();
    scanf ("%d", &temp->data);
}

void insert (BITR t, BITR h )
{
    if (t->data < h->data)
        if (h->left == 1;
            insert(t, h->left);
        else
            if (h->right)
                insert (t, h->right);
            else
            {
                h->right = t;
                return;
            }
    else
        if (h->right)
            insert (t, h->right);
        else
        {
            h->right = t;
            return;
        }
}

```

Here insert is recursive function, which follows from the recursive definitions of Binary Tree.

### 7.8. Searching a value in a binary search tree

As the name suggests, the Binary search trees have the aim of simplifying the process of search. Here the word simplify implies that we are required to have less number of comparisons to check whether a particular node value is present. Now if we are required to search for a value in the tree, we can simply modify the insert function as shown below

```
int search(BITR h, int val)    /* to search a value
```

```

                                from node h */
{
    if (h == NULL) /* if that node itself is NULL,
                    value is absent*/
        return(0);
    if (h->data == val)
        return ( 1 );
    if (val < h->data)    /* if value is less, then
                           search on LHS*/
        return (search (h->left, val) );
    else                  /* otherwise search on RHS */
        return (search (h->right, val) );
}

```

Observe that this is a recursive function. The description can be given to search for a value in the subtree with root h. If the tree is not empty then check whether the data is present at the root. If the value is less than that of value of root then search in the left subtree of the root otherwise search in the right subtree of the root.”

If the passed node is NULL, we can conclude that the given value is absent in the tree.

If value matches with h->data, we return 1 saying that value is present.

If value is less then we search in left subtree of h otherwise in the right subtree of h.

## 7.9 Depth First Traversal of the tree

There are many other ways of traveling the tree. Depth First Search is one of the ways in which the nodes can be traveled. From root we should move in the highest level in one particular direction.

e.g.

In the previous tree the DFS is

21,6,4,9,17,12,10,13,24,35,31,30.

The Algorithm to get the DFS of the tree is as follows :

1. Set temp to root.
2. Print temp's data.
3. Push temp's right (if exists) on the stack.
4. Set temp to temp's left .

5. If temp is not NULL then step 2.  
    else pop a node from the stack.
6. The process continues till the stack is not empty.
7. Stop.

Here we will be pushing the address on the stack. The stack can be implemented using array or linked list. Assuming that the following functions are available (push, pop, stack\_empty), the DFS function will be written as

```
void dfs (TR h)
{
    TR t;
    t = h → left;

    do
    {
        printf("%d", t → data);

        if (t → right)
            t = t → right;
        t = t → left;

        if (t == NULL)
            t = pop();
    } while (!stack_empty());
}
```

The above function is not a recursive one. But we can even written recursive function of the DFS.

### 7.10. Tree Traversals

**Traversal is the most common operation that can be performed on trees. In the traversal technique each node in the tree is processed or visited once, systematically one after the other. Processing may include just displaying the contents of the node or assists in some other operation. For the binary tree we also have these important traversals, which are namely**

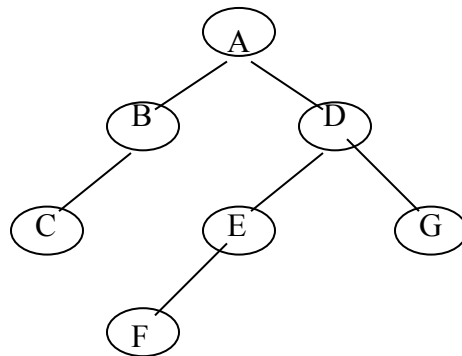
1. Preorder Traversal
2. Postorder Traversal
3. Inorder Traversal

We are very much aware of the fact that there will be left child and right child for any node in the binary tree. The sequence in which the node, its left child and right child are printed, determines the traversal.

### 7.10.1. Preorder traversal of the tree

*In the Preorder traversal, the sequence is in which the nodes are visited is root, left subtree and then right subtree. In short it is **Root-Left-Right**.*

Consider the tree:



**Fig 8. Binary Tree**

Processing order : A B C D E F G

The recursive function for the preorder traversal will be:

```
void pre_order ( TR t )
{
    if (t)
    {
        printf("%d", t->data); /* visit the node */
        pre_order (t->left); /* visit the left subtree in
                                preorder */
        pre_order (t->right); /* visit the right subtree in
                                preorder */
    }
}
```

}

The recursive functions will use the internal stack. If we observe the recursion of the function, we find that there is a recursive call with left child, hence every time left child will be printed. When the node 't' does not have left child, the control will be taken to the previous call and the next call is to the right of node 't'. Actually the Preorder traversal of the tree is same as that of DFS of the tree. The dfs function written before is the non-recursive function for Preorder traversal.

Algorithm for non-recursive function, for preorder traversal

Assume that a pointer T points the root node, S is a stack, TOP is a top index and P represents the current node in the tree.

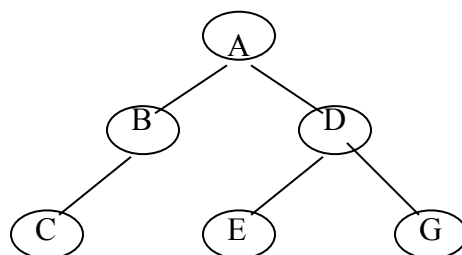
1. Initialization is done first.  
If  $T == \text{Null}$  the function prints it as a NULL tree.  
Otherwise the node is pushed into the stack.
2. Repeat step 3 and step 4 while there is still some node left in the stack .i.e.,  $\text{TOP} > 0$ .
3. Pop the address from top of the stack into the pointer P as  $P = \text{pop}(S, \text{TOP})$ .
4. Repeat while( $P \neq \text{NULL}$ )  
Print the data in the node P.  
If P has a right child call this function again and push the address of this right child into the stack.  
Otherwise store the address of the left child into P.
5. stop.

You may implement the above function and test it with various trees.

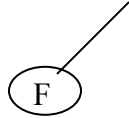
### 7.10.2. Postorder traversal of the tree

*The Postorder traversal is Left – Right – Root, i.e. traverse the left subtree, then the right subtree and then print the root.*

Consider the tree:







**Fig 9. Binary Tree**

Processing order: C B F E G D A

*/\* recursive function to traverse tree in postorder \*/*

```
void post_order ( TR t )
{
    if (t)
    {
        post_order (t->left); /* visit the left subtree in
                                postorder */
        post_order (t->right); /* visit the right subtree in
                                postorder */

        printf("%d", t->data); /* visit the node */
    }
}
```

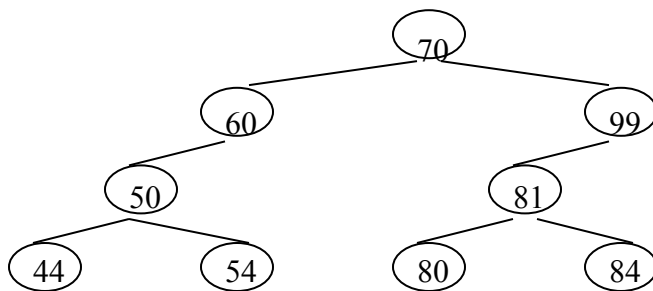
The Iterative algorithm is as follows:

Assume temp is a pointer variable that stores addresses of nodes. Flag is used to denote if the node is visited twice. Here each node will be stacked twice, once when left subtree is traversed and once when its right subtree is traversed. Only on completion of both its subtrees, which is denoted by making the flag as two, the root value will be printed.

1. Set temp to root.
2. Push temp.
3. Set temp to temp's left.
4. if temp is not NULL then step 2.
5. temp = pop()
6. if (temp's flag == 2) then
  - a. print temp's data
- else
  - b. temp's flag set to 2
  - c. push temp
  - d. temp = temp's right
  - e. step 4

7. if (stack is not empty ) then step 5.
8. Stop.

The steps are shown using figures below. Consider the tree.



**Fig 10. Binary Tree**

Initially temp will point to 70 .... Step 1

Following is the execution of the above algorithm as well as the picture shows the sequence in which the different steps in algorithm as well as their effect.

From the root all the nodes will be pushed in the stack, as we travel to the root of each node, till we get NULL.

Stack	temp	flag	Step
70	70	1	Step 2
60	60	1	Step 3
70 60	60	1	Step 2
50	50	1	Step 3
70 60 50	50	1	Step 2
44	44	1	Step 3
70 60 50 44	44	1	Step 2
	NULL		Step 3

When we get NULL, we pop a node from the stack, if its flag is 1, we change it to two and push it back to the stack. Now we travel to the right child of the pop node i.e. 44's right in this case.

70 60 50	44	1	Step 5
----------	----	---	--------

		44	2	Step 6.b
70 60 50 44	44	2		Step 6.c
		NULL		Step 6.d

As we repeat the step of pushing all the left children, again on NULL we pop a node from the stack. If it has flag 2, print it. i.e. 44 in this case.

70 60 50	44	2	Step 5
	44	2	Step 6.a 44

Now pop the next node from the stack and repeat the whole process

70 60	50	1	Step 5
	50	2	Step 6.b

flag of 50, has changed to 2 and pushed in the stack

70 60 50	50	2	Step 6.c
----------	----	---	----------

The next node will be 50's right child i.e. 54, processing repeats from here

70 60 50 54	54	1	Step 6.d
	54	1	Step 2

70 60 50 54 52	52	1	Step 3
	52	1	Step 2
	NULL		Step 3

70 60 50 54	52	1	Step 5
-------------	----	---	--------

70 60 50 54 52	52	2	Step 6.b
	52	2	Step 6.c

70 60 50 54	52	2	Step 6.d
	52	2	Step 5

As we pop the node 52, its flag is already 2, hence it should be printed

70 60 50	52	2	Step 6.a 52
	54	2	Step 5

As we pop the node 54, its flag is already 2, hence it should be printed

70 60	54	2	Step 6.a 54
	50	2	Step 5

	50	2	Step 6.a 50
70	60	1	Step 5
	60	2	Step 6.b
70 60	60	2	Step 6.c
	NULL		Step 6.d
70	60	2	Step 5
	60	2	Step 6. a60
-	70	1	Step 5

The stack is empty, but the popped node has flag 1, hence it will be again pushed in the stack with flag 2, hence the stack empty condition will not be true.

	70	2	Step 6.b
70	70	2	Step 6.c
	99	1	Step 6.d
70 99	99	1	Step 2
	81	1	Step 3
70 99 81	81	1	Step 2
	80	1	Step 3
70 99 81 80	80	1	Step 2
	NULL		Step 3
70 99 81	80	1	Step 5
	80	2	Step 6.b
70 99 81 80	80	2	Step 6.c
	NULL		Step 6.d
70 99 81	80	2	Step 5
	80	2	Step 6.a 80
70 99	81	1	Step 5
	81	2	Step 6.b
70 99 81	81	2	Step 6.c
	84	1	Step 6.d
70 99 81 84	84	1	Step 2
	NULL	2	Step 3
70 99 81	84	1	Step 5
	84	2	Step 6.b
70 99 81 84	84	2	Step 6.c
	NULL		Step 6.d
70 99 81	84	2	Step 5
	84	2	Step 6.a 84
70 99	81	2	step 5
	81	2	Step 6.a 81
70	99	1	step 5
	99	2	Step 6.b
70 99	99	2	Step 6.c
	NULL		Step 6.d

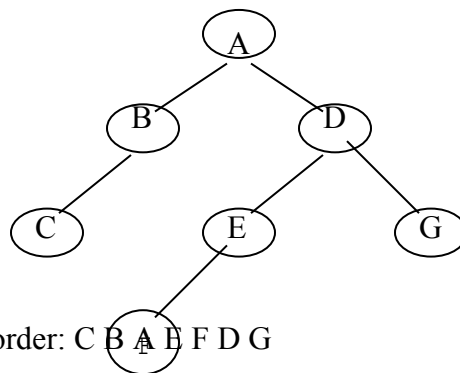
70	99	2	Step 5
	99	2	Step 6.a 99
	70	2	Step 5
	70	2	Step 6.a 70
			Step 7
			Step 8

Therefore the postorder traversal is 44, 52, 54, 50, 60, 80, 84, 81, 99, 70.

### 7.10.3. Inorder Traversal

*The inorder traversal of the tree is traversing the left subtree first, then root and then traversing the right subtree. i.e. before printing the node value, print the value on RHS.*

Consider the tree:



**Fig 11. Binary Tree**

Processing order: C B A E F D G

*/\* recursive function for tree traversal inorder \*/*

```

void in_order ( TR t )
{
  if (t)
  {
    in_order (t->left); /* visit the left subtree in
                        inorder */
    printf("%d", t->data); /* visit the node */

    in_order (t->right); /* visit the right subtree in
                        inorder */
  }
}

```

The iterative algorithm is as follows:

6. Set temp to root
7. Push temp
8. Set temp to temp's left
9. if temp is not NULL then step 2.
10. if (stack is empty) then step 10.
11. temp = pop ( )
12. print temp's data.
13. set temp to temp's right
14. step 4.
15. Stop.

e.g.

In previous tree

70 will be printed only after 60,  
 60 will be printed only after 50  
 50 will be printed only after 44  
 44 does not have left child, hence 44 is printed.  
 There is no right child for 44, hence 50 is printed

As we move to the right of 50 i.e. 54  
 54 will be printed only after 52  
 52 does not have left child, hence 52 is printed.  
 There is no right child for 52, hence 60 is printed.  
 There is no right child for 60, hence 70 is printed.  
 As we move towards the right child of 70, i.e. 90,  
 99 will be printed only after 81  
 81 will be printed only after 80  
 80 does not have left child, hence 80 is printed.  
 There is no right child for 80, hence 84 is printed.  
 As we move towards the right child 81 i.e. 84  
 84 does not have left child, hence 84 is printed.  
 There is no right child for 84, all nodes are printed.

Hence the inorder traversal is  
 44,50,52,54,60,70,80,81,84,99

Observe that the inorder traversal of binary search tree is always the ascending order of the data.

### **7.11. Functions to find Predecessor, Successor, Parent, Brother**

The following program gives some useful functions. These are also useful when we study Binary Threaded Trees in the next session.

```
typedef struct tree /* structure of the tree */
{
    int data;
    struct tree *left, *right;
}*TR;

TR head;          /* header node */

TR getnode (void)/* creates new node and stores
                                value into it */
{
    TR temp;
    int val;

    temp=malloc(sizeof(struct tree));
    temp-> left = temp-> right =NULL;
    scanf("%d", &val);
    temp->data=val;
    return(temp);
}

void display (TR head)
{
    Tr temp;
    temp=head;

    printf("\n\t\t\t\t %d", temp->data);
    if(temp->left)
    {
        printf("\b");
        display(temp-> left);
    }

    if(temp->right)
    {
        printf("\t\t")
        display(temp->right);
    }
}
```

```
void create_tree(void);                /* we have already seen this
                                        function*/
```

```
Tr search(TR root, int target) /* search for a node with
                                value target*/
```

```
{
    do
    {
        if(root->data==target) /* whether target is root */
            return(root);
        else
            if(target < root->data) /* whether it exists in left
                                    subtree*/
            {
                if(root-> left)
                    root=root->left;
                else
                    return(NULL);
            }
        else
        {
            if (root->right) /* whether it exists in right
                               subtree*/

                root=root->right;
            else
                return (NULL);
        }
    } while (1);
}
```

```
TR parent_find(int target)/* to find the parent of target */
```

```
{
    TR temp, root;
    root = head-> left;
    temp = search(root, target);

    if(temp==NULL) /* node itself is not present)
        return(temp);
    else
    {
        if(target== head-> left->data)
            return(head);
    }
```



```

else
{
    temp=head->left;
    do
    {
        if(target <temp->data)
        {
            if(target==temp->left->data)
                return(temp)
            else
                temp=temp->left;
        }
    } while(1);
}
}
}

```

```

void parent(void)
{
    TR result;
    int val;

    char ans ='Y'
    while (ans=='Y')
    {
        clrscr();
        printf("Enter node to find parent:");
        scanf ("%d", &val);
        result = parent_find(val);
        if(result==NULL)
        {
            printf("\n Node is not in the tree !\n");
            printf ("Enter node properly!\n\n");
            getch();
        }
        else
            if(result==head)

```

```

        {
            print("\n\n Node is the root. \n It has no
                    parent!");
            getch();
        }
        else
        {
            printf("\n parent is %d!", result->data);
            getch();
        }
        printf("\n\n Continue (y/n)");
        ans = toupper(getch());
    }
}

int brother_find(int target)
{
    TR root, temp;
    int flag=0;
    root = head-> left;
    temp = search(root, target);

    if(temp==NULL)/* when node itself is not present*/
        return(-1);
    else
    {
        if (target ==head-> left->data)
            return(-999);
        else
        {
            temp=head -> left;
            do
            {
                if(target<temp->left->data)
                {
                    if(target==temp->left->data)
                    {
                        flag=1;
                        if(temp->right)
                            return(temp->right->data);
                        else
                            return(0);
                    }
                }
            }
            else

```

```

        {
            if (target==temp->right->data)
            {
                flag=1;
                if(temp->left)
                    return(temp->left->data);
            }
            else
                return(0);
        }
        else
            temp=temp->right;
    }
} while(flag==0);
}
}
}

```

```

void brother(void)
{
    int val, result;
    char ans ='Y'
    while (ans=='Y')
    {
        clrscr();
        printf("\n\n\n");
        printf("Enter node to find brother :");
        scanf("%d", &val);
        result = brother_find(val);

        if(result==-1)
        {
            printf("node is not in the tree!\n");
            printf("Enter node properly!\n\n");
        }
        else
            if(result ==-999)
            {
                printf ("\n\n Node is the root. \n It has no brother!");
                getch();
            }
            else
            {
                if (result ==0)

```

```

        printf ("\n Node has no brother !\n");
    else
        printf("\n Brother is %d !", result);
    getch();
}

printf("\n\n Continue (y/n)");
ans = toupper(getch());
}
}

```

```

Tr leftmost (TR node)
{
    while(node-> left!= NULL)
        node=node-> left;
    return (node);
}

```

```

TR rightmost(TR node)
{
    while (node->right!=NULL)
        node=node->right;
    return(node);
}

```

```

int successor_find(int target)
{
    TR root, temp, lmost, p;
    root=head->left;
    temp=search(root, target);
    if(temp==NULL) /*when node itself is not present*/
        return(-1);
    else
    {
        if (temp->right)
        {
            lmost=leftmost(temp->right);
            return(lmost-> data);
        }
        else
        {
            p=parent_find(temp->data);

            while(p->left !=temp)

```

```

        {
            temp=p;
            p=parent_find(temp->data);
        }
        return (p->data);
    }
}

```

```

void successor (void)

```

```

{
    int succ, target;
    char ans = 'Y';
    clrscr();

    while(ans=='Y')
    {
        printf("\n Enter node to find successor"    );
        scanf("%d", &target);
        succ=successor_find(target);
        if(succ==-1)
        {
            printf("Node is not in the tree\n");
            printf ("Enter node properly !\n\n");
        }
        else
        if(succ==-999)
        {
            printf ("\nthis node has no successor ! \n");
            getch();
        }
        else
        {
            printf ("Successor is %d", succ);
            getch();
        }

        printf("\n\n Continue (y/n)");
        ans = toupper(getch());
    }
}

```

```

int predecessor_find(int target)
{

```

```

Tr root, temp, rmost, p;
root=head->left;
temp=search(root, target);
if(temp==NULL) /*when node itself is not present*/
    return(-1);
else
{
    if (temp->left)
    {
        rmost=rightmost(temp->left);
        return(rmost->data);
    }
    else
    {
        p=parent_find(temp->data);

        while(p->right!=temp && data!= -999)
        {
            temp=p;
            p=parent_find(temp->data);
        }
        if(p->data==-999)
            return (-999);
        else
            return(p->data);
    }
}
}

```

```

void predeccessor (void)
{
    int pred, target;
    char ans='Y';
    clrscr();

    while (ans =='Y')
    {
        printf("\n enter node ti find predeccessor:");
        scanf("%d", &target);
        pred=predeccessor_find(target);

        if(pred==-1)
        {
            printf("\n Node itself is not present. \n");
        }
    }
}

```

```

        printf("Enter node properly !\n\n");
        getch();
    }
    else
    if (pred==-999)
    {
        printf("\n this node has no predeccessor! N");
        getch();
    }
    else
    {
        printf("Predeccessor is %d" pred);
        getch();
    }
    printf("\n\n Continue (y/n)");
    ans=tpupper(getch());
}
}

main()
{
    create_tree();
    brother( );
    parent ( );
    successor ( );
    predeccessor();
}

```

### 7.12. To delete a node from the tree

Suppose for some reason we are required to delete a value from the tree, then if it is the leaf node there will not be any difficulty in deleting the value. We will have to find the parent node of that value and set the respective pointer to NULL. But if it is any other node we must replace the deleted node with another node such that the positions of new nodes satisfy the condition of the binary search tree.

We will first write a general algorithm and then a detailed pseudo code. You may develop the pseudo code into a program.

Algorithm:

1. determine the parent node of the node marked for deletion. For root no parent exists.

2. If the node being deleted has either a left or right empty subtree then append the non empty subtree to its parent and exit.
3. Obtain the inorder successor of the node to be deleted. Append right subtree of this to its grandparent. Replace the node to be deleted by its inorder successor node. Also, the successor node is appended to the parent of the node just deleted.

#### Pseudo code

Assumptions: X is info of the node to be deleted.

PARENT – address of the parent of the node to be deleted.

CUR – address of the node to be deleted.

PRED,SUCC – pointers to find inorder successor of CUR.

Q – address of the node to be attached to PARENT

D – direction from parent node to CUR

HEAD – list head

FOUND – variable indicating whether node is found or not.

1. /\* Initialize \*/
  - if HEAD->lptr != HEAD /\* no tree exists \*/
    - CUR = HEAD->lptr;
    - PARENT = HEAD
    - D = 'L'
  - else
    - printf("NODE NOT FOUND ");
    - return;
2. /\* search for the node marked for deletion \*/
  - FOUND = 'false'
  - While( !FOUND && CUR != NULL)
    - {
      - if CUR->data = X
        - FOUND = 'true'
      - else
        - if X < CUR->data /\* branch left \*/
          - {
            - PARENT = CUR
            - CUR = CUR->lptr
            - D = 'L'
          - }
        - else
          - {



```

        PARENT = CUR
        CUR = CUR->rptr
        D = 'R'
    }
    if FOUND == 'false'
        printf("NODE NOT FOUND ");
        return;
}/* end of while */

3. /* perform the indicated deletion and restructure the tree */
    if( CUR->lptr == NULL) /* empty left subtree */
        Q= CUR->rptr
    else
    {
        if(CUR->rptr == NULL) /* empty right subtree */
            Q= CUR->lptr
        else /* check right child for successor */
        {
            SUC = CUR->rptr
            if (SUC->lptr == NULL)
            {
                SUC->lptr = CUR->lptr
                Q = SUC
            }
            else /* search for successor of CUR *.
            {
                PRED = CUR->rptr
                SUC = PRED->lptr

                While( SUC->lptr != NULL)
                {
                    PRED=SUC
                    SUC = PRED->lptr
                }
                /* connect the successor */
                PRED->lptr = SUC->rptr
                SUC->lptr = CUR->lptr
                SUC->rptr = CUR->rptr
                Q = SUC
            }
        }
    }
}
/* connect parent of X to its replacement */
if D = 'L'

```

**COND1**

**COND2**

**COND3**

**COND4**

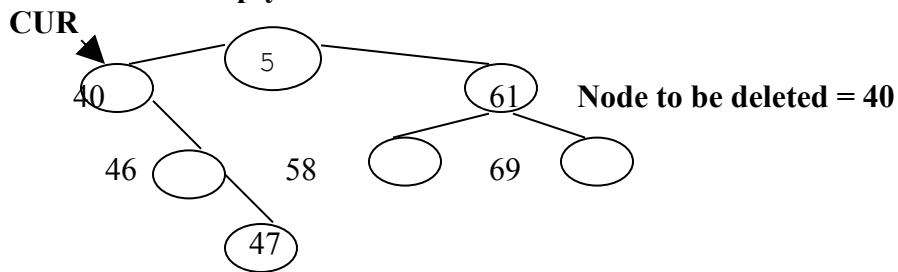
```

    PARENT->lptr = Q
else
    PARENT->rptr = Q
return

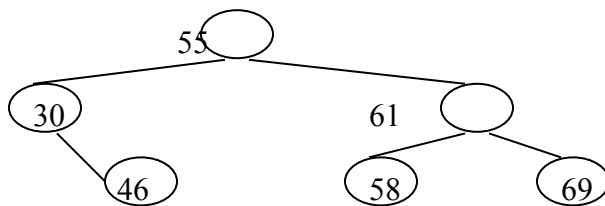
```

Now let us consider some diagrams that depict us various conditions that can occur while deleting a node from the tree.

**Condition 1: Empty left subtree.**



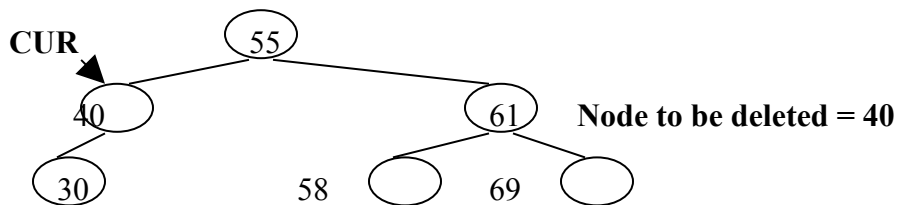
**Before Deletion**



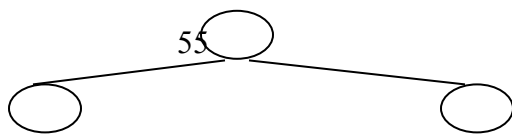
**After Deletion**

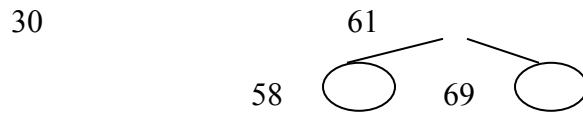
**Fig 11. tree before and after deletion of a node with no left subtree**

**Condition 2: Empty right subtree.**



**Before Deletion**

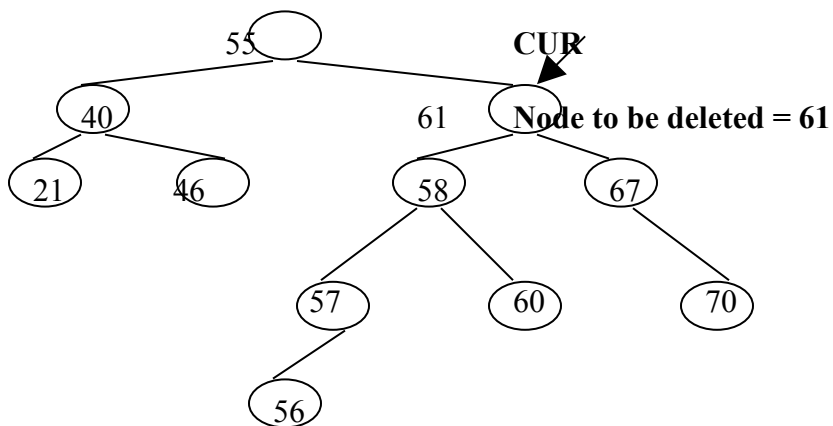




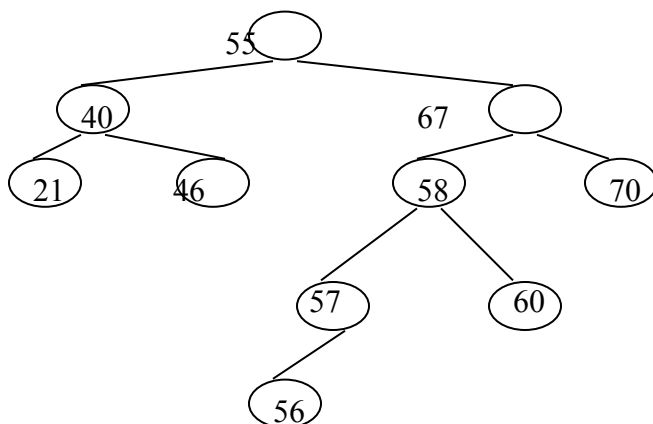
**After Deletion**

**Fig 12. tree before and after deletion of  
a node with no right subtree**

**Condition 3: With right subtree.**



**Before Deletion**



**Fig 13. tree before and after deletion of a node with right subtree**

**Condition 4: With right and left subtree**



**Fig 14. tree before and after deletion of a node with both subtree**

All the arithmetic expressions contain variables or constants, operators and parenthesis. These expressions are normally in the infix form, where the operators separate the operands. Also, there will be rules for the evaluation of the expressions and for assigning the priorities to the operators. The expression after evaluation will result in a single value.

We can evaluate an expression using the stacks and expression trees.

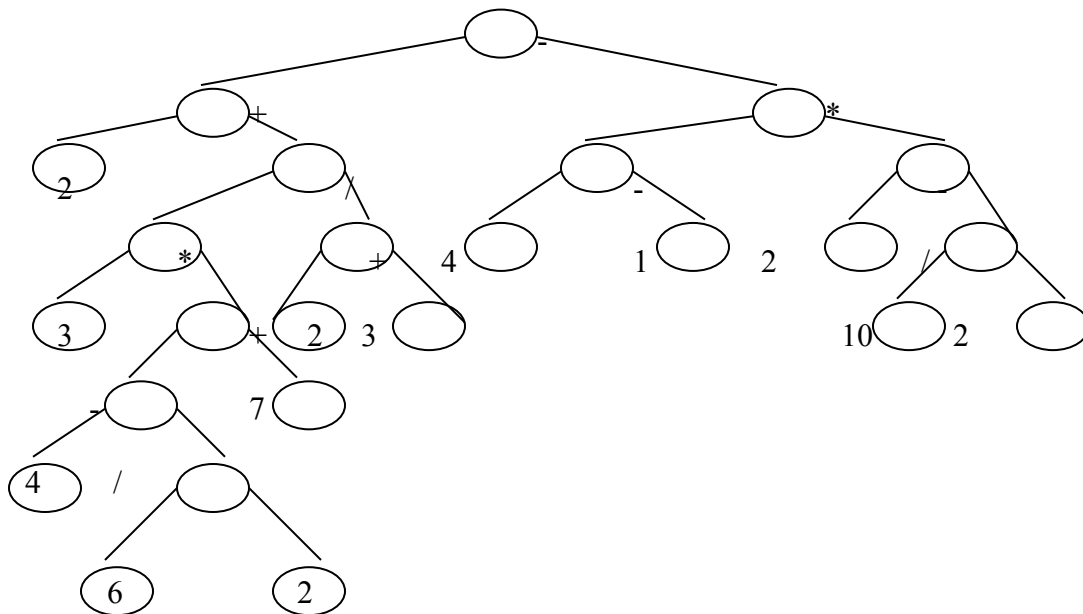
### 7.13. Expression Trees.

We have already seen how to solve expressions in form of postfix or prefix form with the help of stacks. Expressions can also be solved using tree called as expression trees.

Consider an infix expression:

$$2 + 3 * (4 - 6 / 2 + 7) / (2 + 3) - (4 - 1) * (2 - 10 / 2))$$

The expression tree for this expression is:

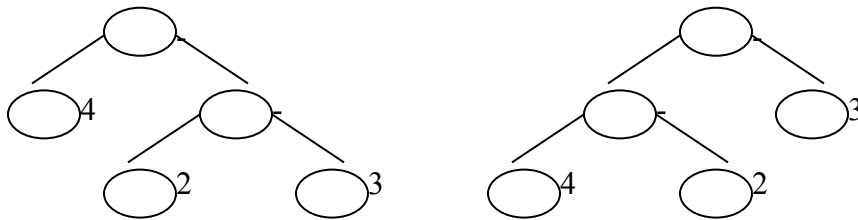


**Fig 15.Expression tree**

We can observe that an expression tree has the operators in all roots and the operands are in leaves of the respective operators. The evaluations will always being from the bottom of the tree, i.e.  $6/2$  is the first operation, which confirms with the usual procedure shown above. A wrong expression tree, would result in wrong answer.

e.g.

$4 - 2 - 3$ . The answer is  $-1$ . The expression trees could be



The difference in the trees is due to the associativity rules. In the first case,  $2-3$  will be evaluated first, resulting in  $-1$  and then it will be subtracted from  $4$  resulting in  $5$ . In the second case,  $4-2$  will be evaluated first, resulting in  $2$  and after subtracting  $3$ , we get the answer as  $-1$ . Hence we can conclude that the second tree is proper.

When it comes to solving the expression, using computer, the expression in the infix form would be slightly troublesome, or if we make certain conversions, the evaluations will be much easier. These forms are, namely, prefix and postfix expressions.

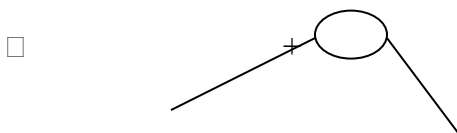
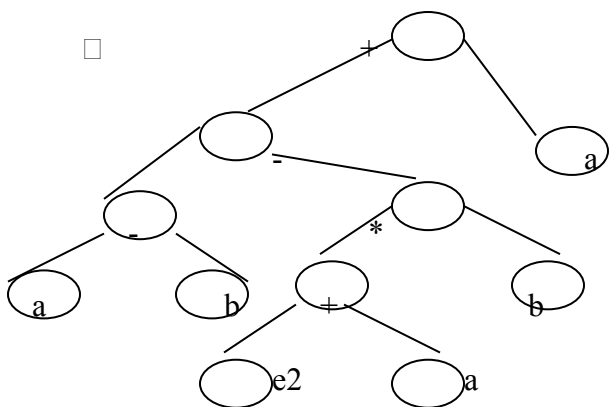
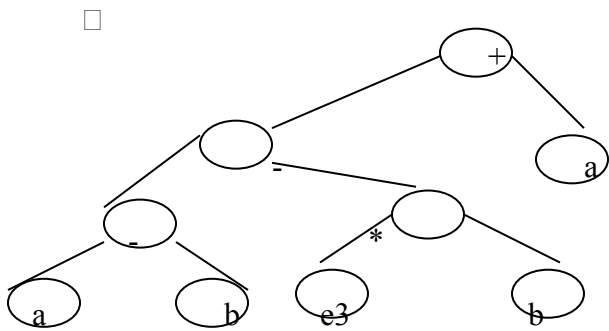
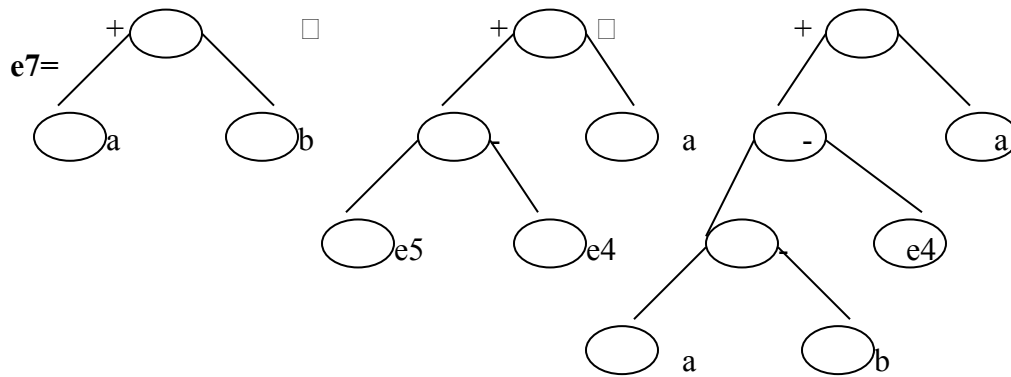
We will see another example by drawing the expression tree for the following expression

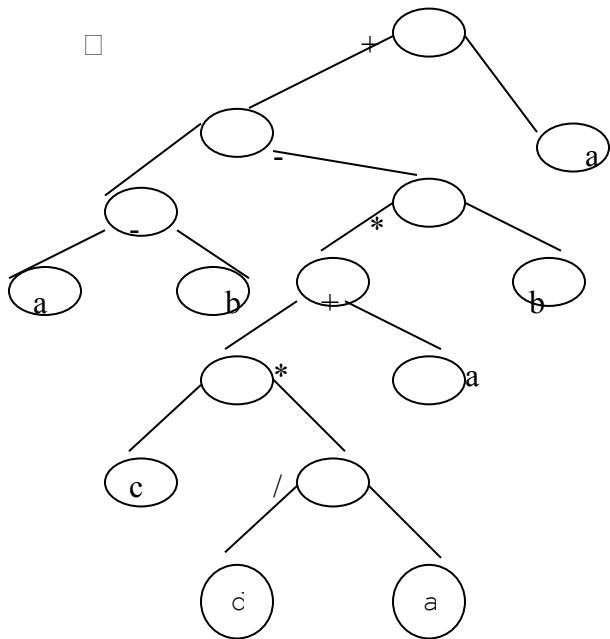
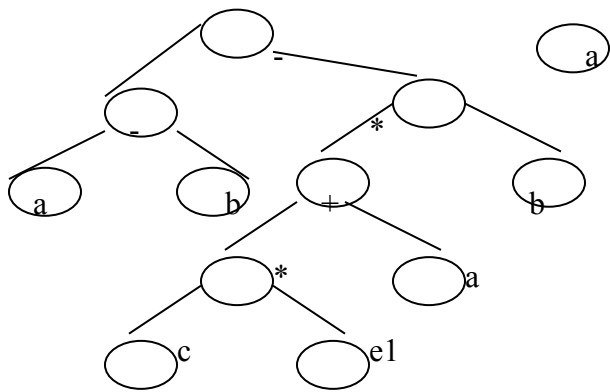
$$a - b(c * d / a + a) * b + a$$

The easiest way for evaluation as well as for the other purposes, is to write expression in fully parenthesized form. During this process, we should consider the steps for evaluation.

$$\begin{aligned}
 & a - b - (c * d / a + a) * b + a \\
 = & a - b - (c * e_1 + a) * b + a \\
 = & a - b - (e_2 + a) * b + a \\
 = & a - b - e_3 * b + a \\
 = & a - b - e_4 + a \\
 = & e_6 + a \\
 = & e_7
 \end{aligned}$$

Forming the tree will be easy from any of the above notations. Travel from bottom to root.





The other way of generating the tree is to convert the expression to the fully parenthesized form. Then every time you remove the pair of brackets you will get two operands separated by an operator, make that operator, the root and the two operands as left and right child and repeat the process.

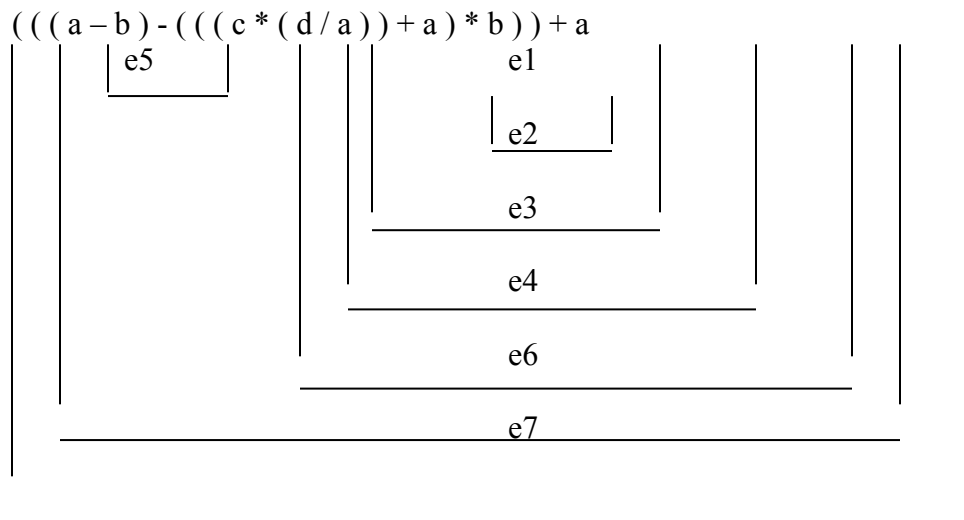
Conversion to the fully parenthesized form is shown below:

$$a - b - (c * d / a + a) * b + a$$



$$\begin{aligned}
&= a - b - (c * (d / a + a) * b + a) \\
&= a - b - ((c * (d / a + a)) * b + a) \\
&= a - b - (((c * (d / a)) + a) * b + a) \\
&= (a - b) - (((c * (d / a)) + a) * b) + a \\
&= ((a - b) - (((c * (d / a)) + a) * b)) + a \\
&= (((a - b) - (((c * (d / a)) + a) * b)) + a)
\end{aligned}$$

During evaluation we will have to solve the inner most bracket first as its priority will be the highest. Writing  $e_1, e_2, e_3$ , etc is equivalent to putting the brackets, which is as shown below.



### Exercises:

- Write a program to create a ternary tree and then to traverse it using inorder traversal.
- If a binary tree has say 'n' levels, what is the maximum number of nodes contained in it? If we are given 'n' values and we are required to generate a binary tree, what could be the minimum and the maximum length of such tree?
- Write a function to check whether the given two trees are identical.
  - Write a function to check whether the structure of the two trees is identical.

***During this session you will learn about:***

4. *Binary threaded tree.*
5. *How to create a binary threaded tree from an existing binary tree.*
6. *Threading during creation.*
7. *Various traversals and threading.*

## 8.1 Introduction

In the previous session on trees, we have seen the creation of trees and its traversal in detail. In this chapter, we will study some advance functions and operations on trees. We have defined the predecessor and successor with reference to inorder traversal. Here, we will see how to reach them without traveling from the root each time. Let us begin our discussion with binary threaded tree.

## 8.2. Binary Threaded Tree

We have defined the tree as a connected graph without a circuit, i.e. there can be no path from the vertex to itself. For binary search tree there could be maximum two children per node and the incoming degree of each node is 1 and the outgoing degree of each node is maximum 2.

Hence we declare the structure for the node in the tree as

```
struct bin_tree
{
    int dat;
    struct bin_tree *left, *right;
}
```

Now whenever we allocate the memory, it will be allocated considering the size of data as well as the memory required to store two pointers. But we find that we will have nodes in the tree, which will have either one or no children. In case of all such nodes, where it does not have either a left child or a right child, the respective pointer, left or right, will contain NULL.

If a tree contains  $n$  nodes, then as per the definition it will have  $(n - 1)$  edges. Now every node has two pointers in it. Hence there is a capacity to store  $2^{\text{nd}}$  address among  $n$  nodes. To form these we are using  $(n-1)$  pointers. Hence in a tree of  $n$  nodes there will be  $(n + 1)$  pointers, which are stored as NULL because the respective children do not exist.

As a concept as well as the definition, this is correctly implemented. But when we consider different algorithms and if we actually study the repetitive requirements of different operations on trees, we will find that if we utilize those  $(n+1)$  locations to store same specific node addresses, then the algorithms would be faster and efficient. But if you store addresses in these, then there will be  $n$  edges, violating the definition of the tree. So we decide to put some marking on these addresses or edges, indicating that these edges do not belong to the tree. We have seen in the trees, that all the nodes will point to their descendent or children. Hence the direction of all the edges is downwards. The edges, which we will be introducing, either point upwards or at the same level.

Once this is decided, then we find that we cannot decide whether a particular node is a leaf node, because there will be addresses in the positions left and right. But now we would check the markings on these edges or what we will be calling as flags, should be checked. There will be two flags per node, lflag for left edge and rflag for right edge.

When these flags have value 1, then we can say that the children are present. When lflag is 1 and rflag is 0, the node has leftson but it does not have the right son.

When lflag is 1, we can say that the node has left child and its address will be stored at the left. But when lflag is 0, in the left we will get the address of some other node in the tree that is not the child of the original node. Actually lflag = 0 has pointed out that left child is absent. These addresses, which we store in left or right links form the edges, which are not edges of the tree. These are known as '**threads**'. The tree thus formed is called as **Threaded Binary Tree**.

There must be some logic to decide as where the left thread or the right thread of a particular node is pointing at. We can store these addresses depending on a particular traversal.

### 8.2.1. Inorder Threading

**When the left thread points to the inorder predecessor and right thread to inorder successor, the threading is known as 'Inorder Threading'.**

To have a threaded binary tree, there are two way of creation,

- You can create a simple binary tree and then thread it.
- During creation only, form these threads.

Let us consider the first one.

#### 8.2.1.1. To create a simple binary tree and then thread it

We already know how to create a binary search tree. If we want to thread this tree, then we will have to perform two jobs.

- To set the flag to proper values and
- To set the proper addresses in place of NULL.

Hence again we are required to travel through all the nodes, checking for the NULL pointers. This can be done using Breadth First Search.

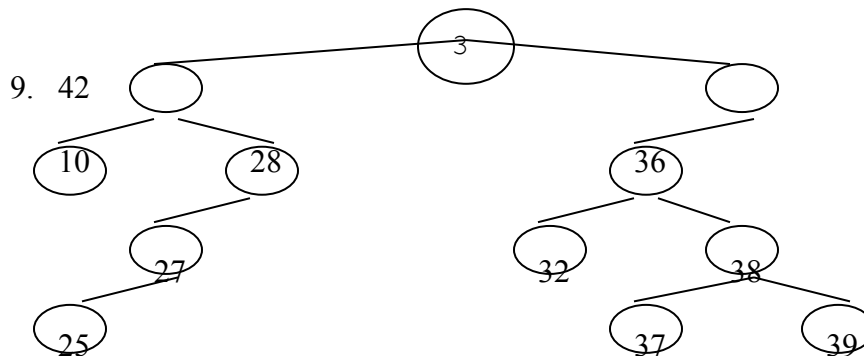
Through BFS when we get the node, perform the following steps.

1. Say the node is temp.
2. If temp has leftson, set lflag = 1  
Else temp's left to point to its predecessor and set lflag = 0
3. If temp has rightson, set r flag = 1  
Else temp's right to point to its successor and set rflag = 0.

There will be one node, i.e. the leftmost, which does not have predecessor and one node, i.e. the rightmost, which does not have the successor. At these places, we should respectively write the address of header.

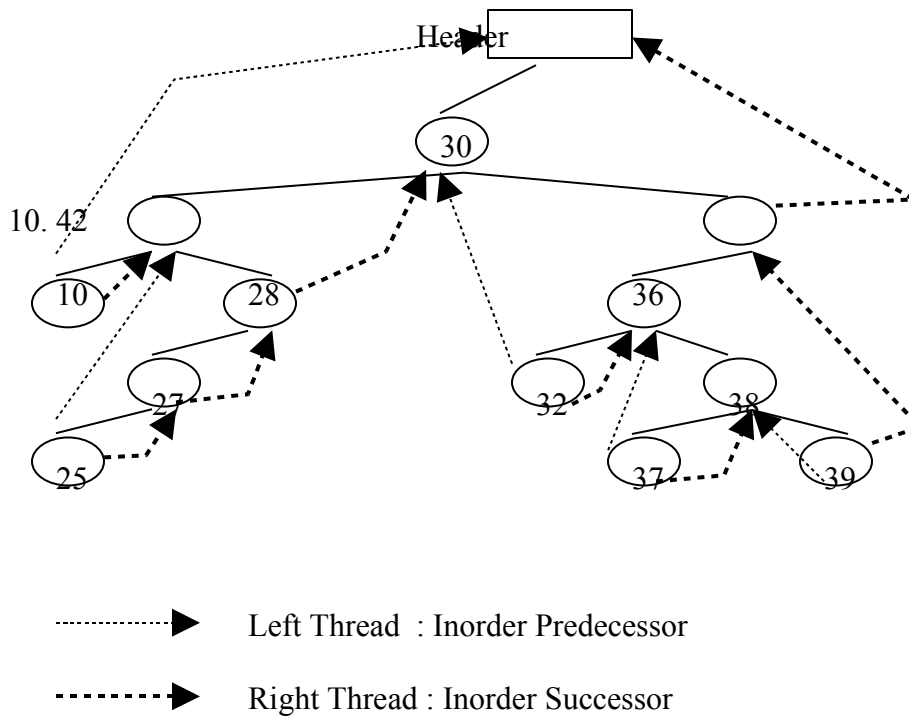
Consider the binary search tree formed by using.

8. 20 28 27 42 10 36 38 39 32 25 37, i.e.



**fig 1. Binary tree**

The corresponding threaded binary tree will be



**fig 2. Binary threaded tree for binary tree in fig 1.**

Observe that  $\text{pred}(25) = 21$ ,  $\text{succ}(25) = 27$  and the threads are indicating the same nodes. Thus this will have very high time complexity as for finding the predecessor / successor and then threading it..

To reduce the time complexity, we can thread the tree during creation only.

### 8.2.1 2. To thread the tree during creation

It follows a simple technique. Initially when the root is connected to the header, it being the only node, its left as well right threads are pointing towards the header. After that whenever we attach a node to any other node, we will simply pull the existing thread and generate a new thread where required.

Suppose a node new is to be attached to temp as leftson, then the following steps should be used.

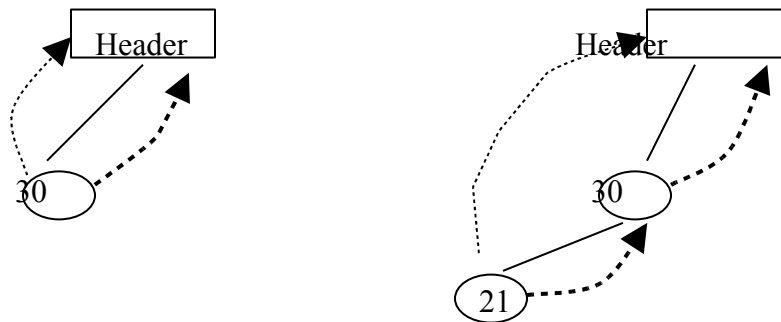
As new is attached as leftson of temp, it will be in between pred(temp) and temp. Thus the predecessor of the temp will now be predecessor of new.

Therefore      new  $\square$  left = temp  $\square$  left ;  
                     temp  $\square$  lflag = 1  
                     new  $\square$  lflag = 0  
                     temp  $\square$  left = new

Thus we copy the thread, set proper values to flags and attach new as leftson of temp. 'New' is the leaf node. Hence will even have right thread pointing to the successor. But as new is leftson of temp, temp will be successor of new.

Therefore new  $\square$  right = temp  
                     new  $\square$  rflag = 0

e.g.

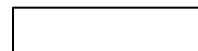
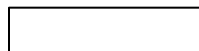


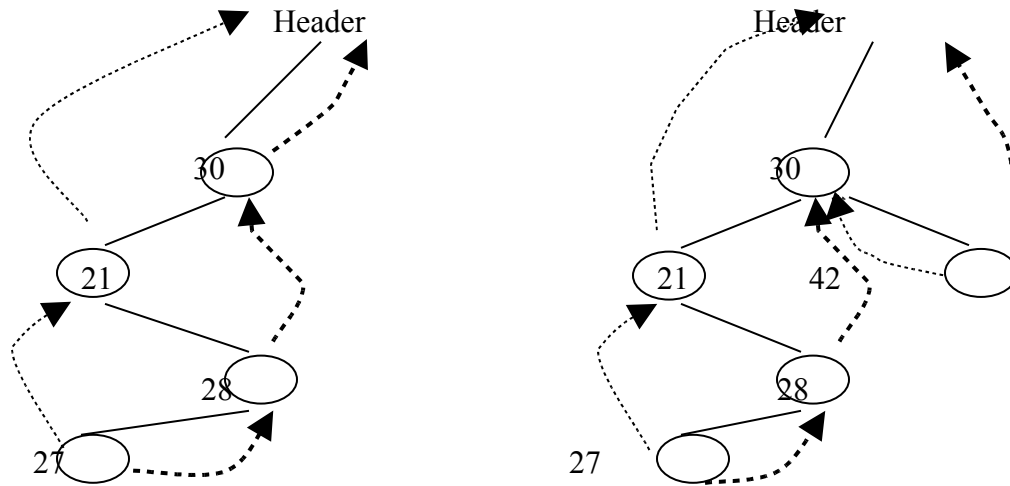
**fig 3. threading process**

Similarly when new is to be attached as rightson of temp, following steps will be used.

- new  $\square$  right = temp  $\square$  right                      (pull the thread)
- temp  $\square$  rflag = 1
- new  $\square$  rflag = 0
- temp  $\square$  right = new
- new  $\square$  left = temp                      (parent will be predecessor)
- new  $\square$  lflag = 0

Thus





left thread of 28 is  
pulled by 27

Right thread of 30 is  
pulled by 42

**fig 4. threading process**

If we make the changes in the previous create function, we can create the threaded tree.

Let us see a program and understand the working of this tree creation process.

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

typedef struct tree
{
    int dat, lflag, rflag;
    struct tree *left, *right;
} *TR;

TR que[20], s, header;

struct tree head;

int front =0, end=0,

TR pop();
void push(TR node);
int quempty();
```

```

int leftmost (TR temp);
int rightmost (TR temp);
TR getnode (int val);
void insert (TR new);
void display();

```

```

main()
{
    TR t, new;
    char c;
    int val;

    header = malloc(sizeof(struct tree));
    header->dat = 10000;
    printf("give the data ...");
    scanf("%d", &val);
    s = getnode(val);
    header->left = s;
    s -> left = header;
    s-> right = header;
    do
    {
        printf("Give the data ...");
        scanf("%d", &val);

        if (val != -99)
        {
            new = getnode(val);
            insert (new);
        }
    } while (val != -99);
    display ();
}

```

```

void insert (TR new)
{
    TR t;
    int flag = 0;
    t = sl;
    while (!flag)
    {
        if(new ->dat <t->dat)
            if (t->lflag)
                t = t -> left;
    }
}

```



```

        else
        {
            new -> left = t -> left;
            new -> right = t;
            t -> left = new ;
            t -> lflag = 1;
            flag = 1;
        }
    else
        if (t->rflag)
            t = t->right ;
        else
        {
            new -> right = t -> right;
            new -> left = t;
            t -> rflag = 1;
            t -> right = new ;
            flag = 1;
        }
    }
}

```

```

TR getnode (int val)
{
    TR p;
    p = malloc (sizeof (struct tree))
    p-> lflag = 0;
    p->rflag =0;
    p->dat=val;
    p->left = NULL;
    p->right = NULL;
    return(p);
}

```

```

void display()
{
    int i=0,p,q;
    TR t;
    t=s;
    push(t);
    push (&head);
    print (“the breadth first search is ....\n”);
    printf(“LEVEL    %d: “,i);
}

```

```

    While (!quempty())
    {
        t = pop ();
        if (t == &head)
        {
            i ++;
            printf("\n##### LEVEL IS OVER #####\n");
            getch();
            printf("\n\n LEVEL          %d:  ",i);
            printf("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            push (&head);
        }
        else
        {
            printf("value is ....% d\n", t->dat);
            if (t->lflag)
                p=rightmost(t->left);
            else
                p = t->left->dat;

            if (t->rflag)
                q=leftmost (t->right);
            else
                q=t->right ->dat;
            printf("Predecessor is %d Successor is ...%d\n",p,q);
            print("\n*****\n");
        }

        if(t->lflag)
            push(t->left);
        if(t->rflag)
            push(t->right);
    }
}

TR pop()
{
    TR n;
    n = que[front];
    front ++;
    return(n);
}

void push (TR no)

```

```

{
    que[end]=no;
    end ++;
}

int qempty()
{
    if (front == end-1)
        return(1);
    return(0);
}

int rightmost (TR temp)
{
    while (temp->rflag)
        temp=temp->right;
    return (temp->dat);
}

int leftmost (TR temp)
{
    while(temp->lflag)
        temp=temp->left;
    return(temp->dat);
}

```

## Output

Give the data .....5  
 Give the data .....4  
 Give the data .....7  
 Give the data .....9  
 Give the data .....3  
 Give the data .....34  
 Give the data .....12  
 Give the data .....-99

The breadth fist search is .....

LEVEL 0 ; value is ....5

Predecessor is .....4 Successor is .....7

\*\*\*\*\*

##### LEVEL IS OVER #####

```

LEVEL 1 :
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
value is .....4
Predecessor is .....3 Successor is .....5
*****
value is .....7
Predecessor is .....5 Successor is .....9
*****
##### LEVEL IS OVER #####

```

```

LEVEL 2 :
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Value is .....3
Predecessor is .....10000 Successor is .....4
*****
value is .....9
Predecessor is .....7 Successor is .....12
*****
##### LEVEL IS OVER #####

```

```

LEVEL 3 :
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!11
value is .....34
Predecessor is .....12 Successor is .....10000
*****
##### LEVEL IS OVER #####

```

```

LEVEL 4 :
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
value is .....12
Predecessor is .....9 Successor is .....34
*****

```

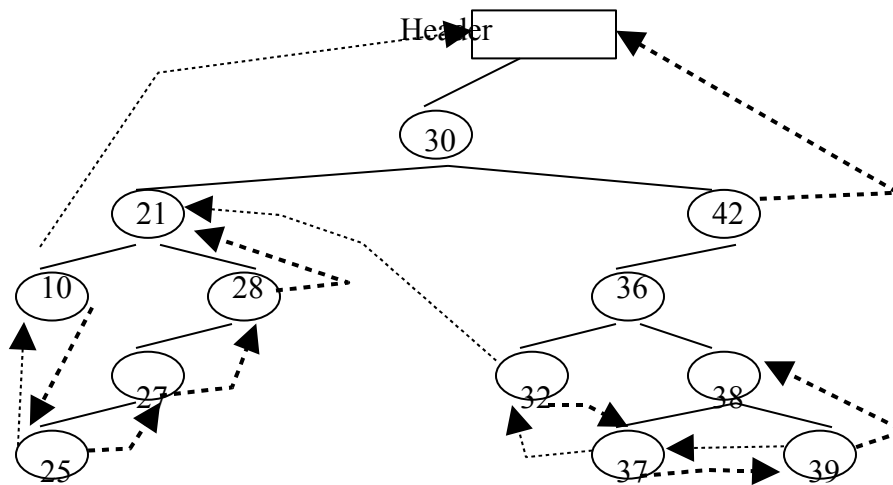
### 8.3. Postorder Threading

In the Inorder threading we have used the inorder threads, i.e. when rightson was absent we had written the inorder successor at that position and for leftson, if absent we write inorder predecessor. For postorder threading, we will have to place the postorder successor and postorder successor.

To travel it in postorder, first goto leftmost. If it does not have a rightson, then print it. Else goto its rightson and its leftmost. (Repeat). When node does not have rightson, using thread goto successor and print it. When we reach a node, which has

rightson, using thread, then print it and goto, its parent and its rightson it repeat the process. The actual traversal with reasons is shown below.

Let us see the postorder threaded tree and the steps to see how it is done next:



**fig 5. Post order Binary threaded tree**

**Temp**

**Print**

1. Goto leftmost of the root	10	
2. Check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread. Check whether temp has rightson.	25	10
2. If yes, temp is temp's right Else print temp and move to right thread.		25
2. Check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.	27	
		27
2. check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.	28	
		28
3. Now 20 is reached through right thread hence print it and move its parent if the previous node was leftson	20	20
	30	
4. Move to parents right	42	
1. Goto leftmost	32	
2. Check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.		32
<b>2. check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.</b>	37	
2. Check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.	39	
		39

2. check whether temp has rightson. If yes, temp is temp's right else print temp and move to right thread.

38

3. Now 38 is reached through right thread hence print it and move its parent if the previous node was leftson.

38

36

4. The previous node was rightson hence again goto parent and print it.

36

2. Check whether temp has rightson, if yes, temp is temp's right else print temp and move to right thread.

42

42

3. Now 30 is reached through right thread hence print it and move its parent if the previous node was leftson

30

30

4. When parent is header

Header

STOP

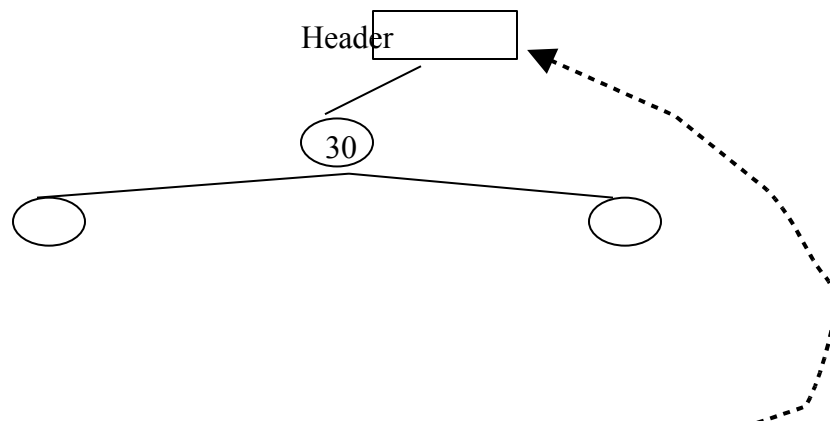
#### 8.4. Preorder Threading

On the similar lines, we can thread the tree using preorder threads. For preorder threading, we will have to place the preorder successor and preorder successor.

For preorder traversal a single technique can be used. Start from root, move to the left as long as leftson exists. When lflag is 0, move to right and repeat the process, till we reach the header.

In all of the above discussions, we find that inorder threading is very powerful as we get the traversals in better way without calling the functions, like parent.

The preorder threaded tree is:



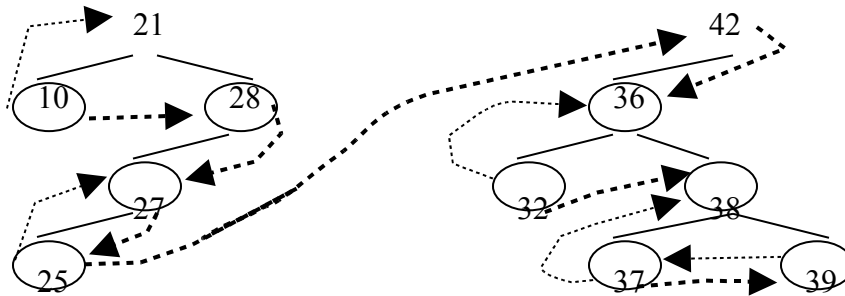


fig 6. Pre order Binary threaded tree

## 8.5. Traversal of Binary Threaded Tree

### 8.5.1: Inorder Traversal

For an inorder threaded binary tree, this traversal will be very easy. If we start from the leftmost, then using right threading we can get the inorder successor. But thread is possible only when rightson is absent. When rightson is present. Its leftmost will be inorder successor. Hence the steps can be written as follows.

11. temp = leftmost(root)
12. if (temp has rightson)
  - then temp = leftmost(rightson)
  - else temp = temp → right
13. repeat step 2 while temp != header

Here we are making use of leftmost function, which returns the address of the leftmost node of the current mode.

Thus in previous tree :

Temp = leftmost (30) = 10

Temp does not have rightson

Therefore using thread temp = temp's right = 20

Temp has rightson, i.e.28 and the leftmost of it will be 25

Therefore temp = leftmost (28) = 25

Temp does not have rightson, therefore thread.

temp=28

Temp does not have rightson, therefore thread.

temp =30

Temp has rightson, i.e. 42, whose leftmost is 32.

And so on.



In preorder threading, we find that whenever leftson exists, it will be the preorder successor and if rightson is absent, the right thread will always point to the leftson. Whereas in postorder threading, the traversals will contain parent function.

### 8.5.2. Preorder Traversal

The preorder traversal is Root – Left – Right. Here to travel left we do not require any thread, hence we can move from root to its leftmost just by using Left child. As we reach the leftmost now it is the time to use thread. The next value to be printed is the position if we move to right with thread till we get the node having rightson. We move to the rightson and again repeat the procedure.

Hence the steps will be as follows.

- g. temp = root
- h. print temp's data
- i. if (temp  $\square$  lflag is 1)
  - then temp = temp's left and goto step 2
- j. else while (temp  $\square$  rflag is 0)
  - temp = temp  $\square$  right
- k. temp = temp  $\square$  right
- l. goto step 2

In case of previous tree :

Initially temp is 30	- step 1
Print 30	- step 2
Now temp is 20	- step 3
Print 20	- step 2
Now temp is 10	- step 3
Print 10	- step 2
Now temp is 20	- step 4
Now temp is 28	- step 5
Print 28	- step 6/2
Now temp is 27	- step 3
Print 27	- step 2
Now temp is 25	- step 3
Print 25	- step 2
Now temp is 27	- step 4
Now temp is 28	- step 4
Now temp is 30	- step 4
Now temp is 42	- step 5
Print 42	- step 6/2
Now temp is 36	- step 3

Print 36	- step 2
Now temp is 32	- step 3
Print 32	- step 2
Now temp is 36	- step 4
Now temp is 38	- step 5
Print 38	- step 6/2
Now temp is 37	- step 3
Print 37	- step 2
Now temp is 38	- step 4
Now temp is 39	- step 5
Print 39	- step 6/2
Now temp is header	- STOP

### 8.5.3. Postorder Traversal

For postorder traversal, the printing sequence will be Left – Right – Root.

In this case we should first goto the leftmost of root, then if rightson exist, then goto leftmost of rightson, else using thread reach a node, which is inorder successor. Check whether it has rightson and repeat the process. Remember whenever we print a node which is by itself a rightson, then the next mode to be printed is the left thread of its leftmost and then the next node is right thread of its rightmost.