TREE

The tree structure is mainly used to represent data containing a hierarchical relationship between elements, e.g. records, family trees and official relationships among workers.

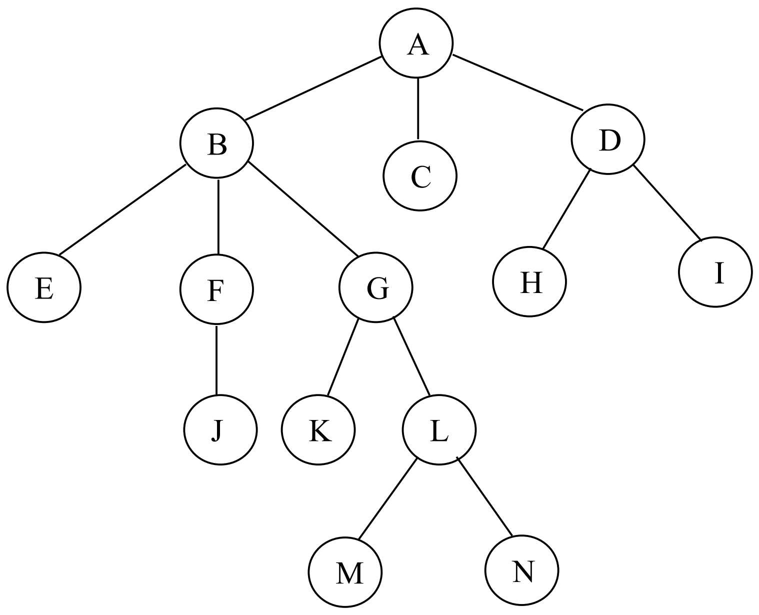
DEFINITION OF GENERAL TREE:

As general ***tree*** (or simply tree) is defined to be a non-empty finite set T of elements called nodes such that:

(i) T contains a distinguishable R called the root of T (all other elements have parent but this element does not have any predecessor).

(ii) The remaining elements of T form an ordered collection of zero or more disjoint trees: T_1, T_2,…, T_n which are called ***subtrees*** of the root R.

The following figure shows a general tree:



In the above diagram 'A' is the root of the tree. C, E, H, I, J, K, M and N are called leaf nodes.

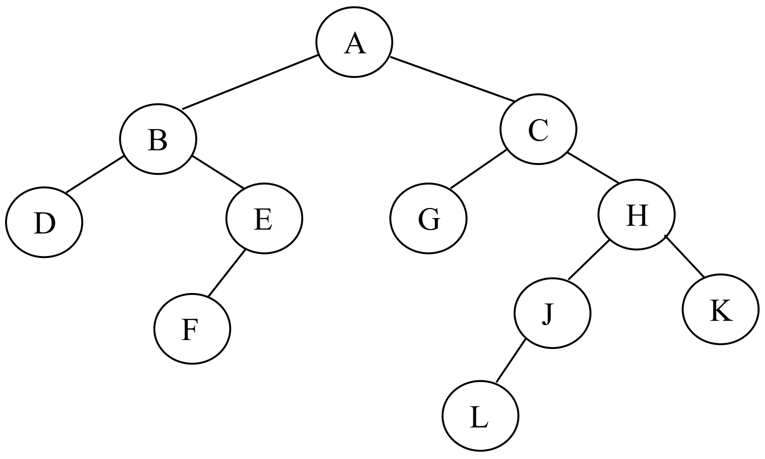
DEFINITION OF BINARY TREE: -

A binary tree is a finite set of elements, which is partitioned into three disjoined subsets.

(i) The first subset contains a single element called the ***root*** of the tree.

(ii) The other two subsets are themselves form an ordered pair of disjoint binary trees called the ***left and right subtrees*** of the root which can be empty.

The following figure on the next page shows a binary tree:



In the figure given above: -

(i) The tree T is consisted of 11 nodes represents by alphabets in capital letter.
(ii) The ***root*** of T is a node 'A' at the top of the diagram.
(iii) A left-downward slanted line from a node 'N' indicates a ***left successor*** of 'N', and a right-downward slanted line from 'N' indicates a ***right successor*** of 'N'. According to the above diagram 'B' is a left successor and 'C' is a right successor of the node 'A'.
(iv) Any node 'N' in a binary tree T has either 0,1 or 2 successors.
(v) The nodes with no successor are called terminal nodes or leaf node e.g. D, F, G, L and K are leaf nodes.
(vi) Suppose 'N' is a node in T with left successor S_1 and right successor S_2. Then 'N' is called the parent of S_1 and S_2. S_1 is called the ***left child*** of 'N' and S_2 is called the ***right child*** of 'N'. Furthermore, S_1 and S_2 are said to be ***siblings***.
(vii) Parent of a node is called its predecessor or ***ancestor***. Similarly, child of a particular node is called its ***descendent***.
(viii) The line down from a node 'N' of T to a successor is called an edge, and a sequence of connective edges is called a ***path***. A terminal node is called a ***leaf*** and a path ending in a leaf is called a ***branch***.
(ix) Each node is a binary tree T is assigned a level number, as follows: the root R of the tree T is assigned a level **0** (zero) and every other node is assigned a level number which is ***1 more than the level number of its parent***. Furthermore, those nodes with the same level number are said to belong to the ***same generation***.
(x) The ***depth (or height)*** of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T. The depth of the above binary tree is 5. But according to *"Tanenbaum"* the ***depth*** of a binary tree is the maximum level of any leaf in the tree. This is equal to the length of the longest path from the root to any leaf. Hence depth of the above binary there are 4 according to the last definition.
(xi) Binary tree T and T′ are said to be ***similar*** if they have the same structure or, in other words, if they have the same shape. The trees are said to be ***copies***

if they are similar as well as they have the same contents at corresponding nodes.

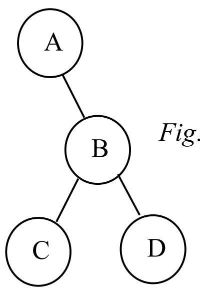


Fig: a

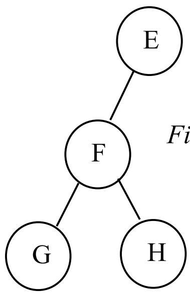


Fig: b

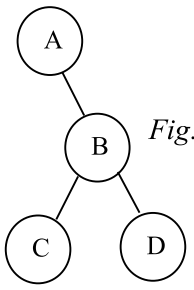


Fig: c

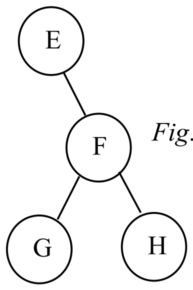


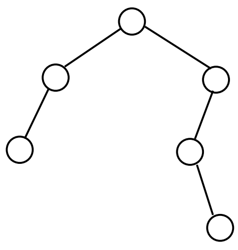
Fig: d

(a), (c) and (d) are ***similar*** & (a) and (c) are ***copies***. (b) is neither similar has a copy of the tree (d). (b) & (d) are ***mirror image*** to each other.

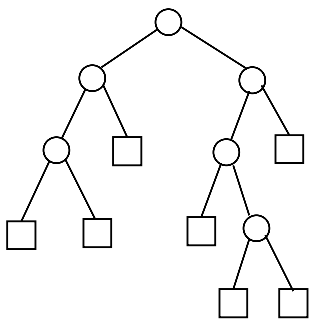
STRICTLY BINARY TREE (2- tree/Extended Binary tree): -

A binary tree is said to be the ***strictly binary tree*** or ***extended binary tree*** or ***2-tree*** if each node of the tree, say N has either **0** or **2** children. In such a tree, the nodes with 2 children are called ***internal nodes***, and the nodes with **0** child are called ***external nodes***.

Q: *Why a strictly binary tree is called an extended binary tree?*
A: The term "extended binary tree" comes from the following operation. Consider the following binary tree T.



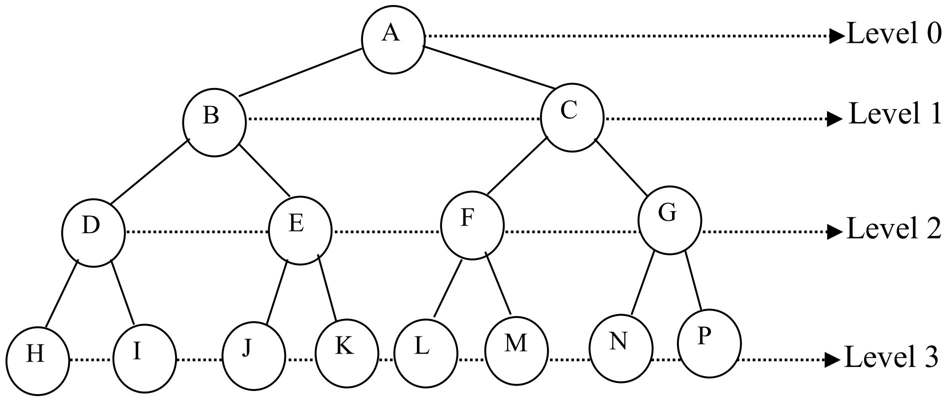
Then T may be ***"converted" into a 2-tree*** by replacing each empty subtree by a new node as follows.



In the above the new tree is a 2-tree. Furthermore, the nodes in the original tree T are now the internal nodes in the extended tree, and the new nodes are the external nodes in the extended tree.

<u>COMPLETE BINARY TREE</u>: -

A binary tree is said to be complete if all its leaves, except possibly the last, have the maximum number of possible nodes, and if all nodes at the last level appear as far left as possible.



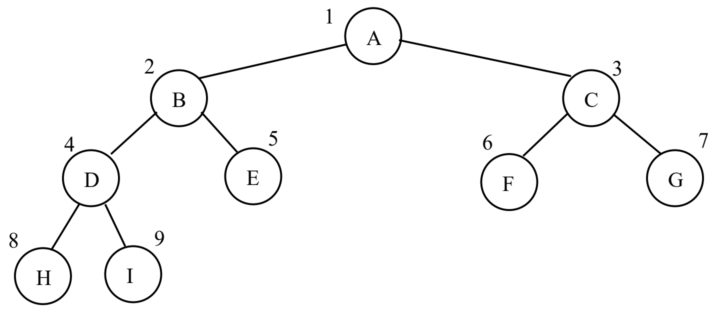
The above tree is a complete binary tree. Their nodes are numbered from 1 through 15, from left to right, generation by generation.

<u>ALMOST COMPLETE BINARY TREE</u> OR <u>NEARLY CVOMPLETE BINARY TREE</u>: -

A binary tree of depth ***d*** is ***almost complete binary thee*** if:

1. Any node ***nd*** at level less than ***d-1*** has two sons.
2. For any node ***nd*** in the tree with a right descendent at level ***d***, ***nd*** must have a left son and every left descendent of ***nd*** is either a leaf at level ***d***

The following tree is a strictly binary tree as well as almost complete binary tree.



Specifically, the left and right children of a node with number K are, respectively, 2*K and 2*K+1 e.g. left and right children of node numbered 4 are respectively 2x4=8 and 2x4+1=9.

Height of the complete binary tree (h) = (max. level+1) for the above binary tree height = 3+1= 4.

Total number of nodes in the complete binary tree = $2^{h}-1 = 2^{4}-1=15$.

Number of nodes in a particular level **l** is **2^l** e.g. Number of nodes in the level 2 is $2^2=4$.

Total number of nodes of a complete binary tree '**n**'= (2 * number of leaf nodes – 1). For the above tree n = 2*8 - 1 = 16 - 1 = 15.

We know, no. of nodes in a complete binary tree **$n = 2^h-1$.** Where h is the height of the tree i.e. **$2^h = n+1$** or **$h = \log_2 (2n+1)$** e.g. in case of a binary tree with n=15,

$$h = \log_2 (15+1) = \log_2 16 = \log_2 2^4 = 4 \log 2^2 = 4.$$

Q: If a nearly complete binary tree contains 12 nodes. Calculate its height.

A: We know, **$h =\log_2(n+1)$**

Hence height of the given binary tree = $\log_2 (12+1) =\log_2 13$. [$2^3=8$ and $2^4=16$] $\log_2 16 = \log_2 2^4 = 4 \log_2 2 = 4$.

Height of the nearly complete binary tree is 4.

REPRESENTING BINARY TREE IN MEMORY: -

We can represent a binary tree in memory by any of the following two methods:

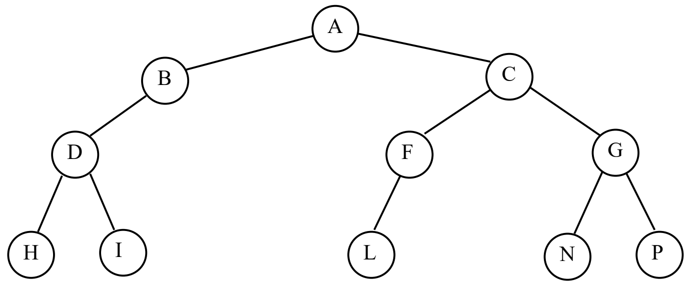
(i) Sequential representation (Using array)

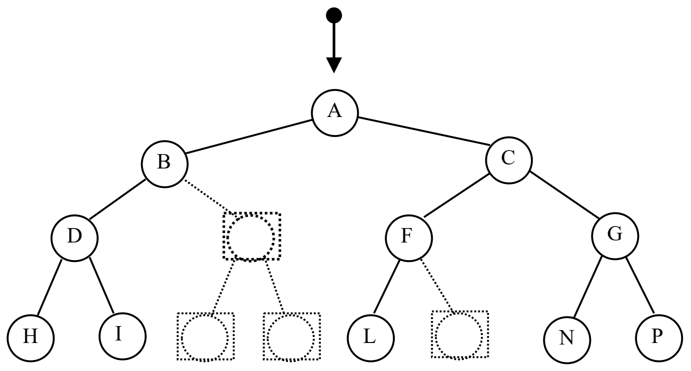
(ii) Linked representation (Using linked list)

(i) SEQUENCIAL REPRESENTATION OF BINARY TREE IN MEMORY:-

The sequential representation of a binary tree requires an array whose indices begin from 1 to N. If a node A is positioned at K-th index of that array, then its left child is positioned at (2K)-th and the right child is positioned at (2K+1)-th index of the array.

Let us consider the following binary tree given on the next page:





The sequential representation will be as follows:

	TREE
1	A
2	B
3	C
4	D
5	NULL
6	F
7	G
8	H
9	I
10	NULL
11	NULL
12	L
13	NULL
14	N
15	P

(ii) LINKED REPRESENTATION OF BINARY TREE IN MEMORY:-

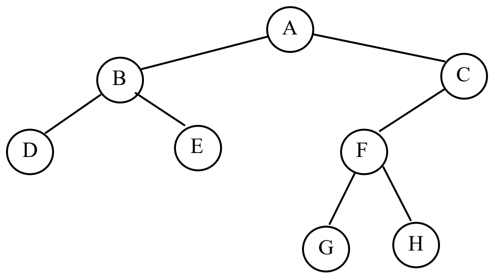
Linked representation requires a structure for a node of the binary tree in the form

LEFT	INFO	RIGHT

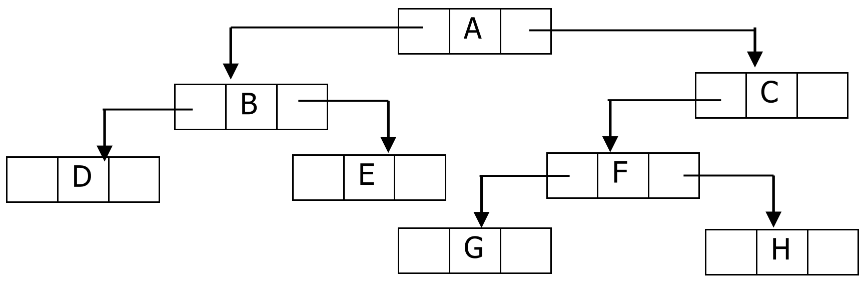
Which can be represented in 'C' Language as,

```
struct node {
   struct node *left;
   int info;
   struct node * right;
 };
```

Let us consider the following binary tree:



The schematic diagram of the above tree is as follows:



Thus the memory representation of the above tree is:

tree $\boxed{1} \rightarrow$ 1

AVAIL $\boxed{9} \rightarrow$ 9

	LEFT	*INFO*	*RIGHT*
1	2	A	3
2	4	B	5
3	6	C	NULL
4	NULL	D	NULL
5	NULL	E	NULL
6	7	F	8
7	NULL	G	NULL
8	NULL	H	NULL
9			10
10			11
11			12
12			13
13			14
14			15
15			NULL

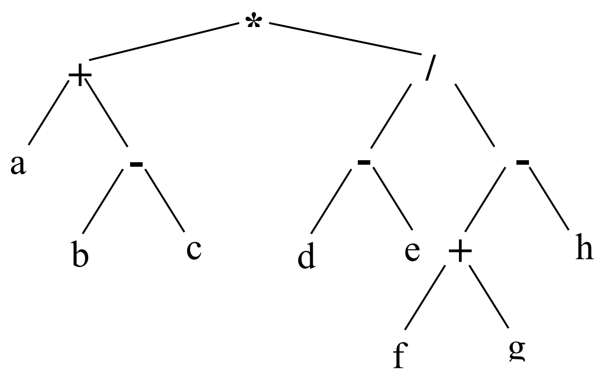
EXPRESSION - TREE

Q : Let us consider the following algebraic expression E:

[a + (b – c)] * [(d – e) / (f + g – h)]

Draw a corresponding binary tree of the above expression.

N. B. *In the binary tree representation of an algebraic expression the **internal** nodes represent the **operators** whereas the **leaf** nodes represent **operands.***



H. W.

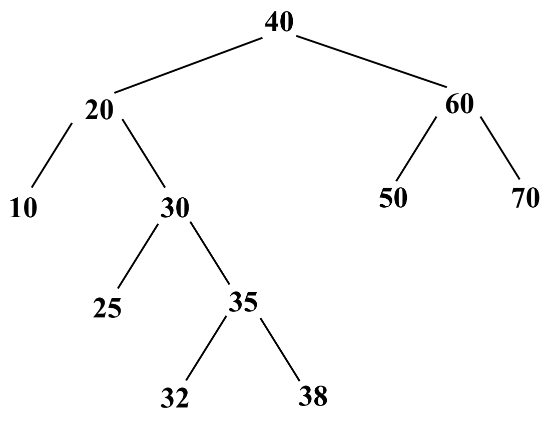
1. a + b * c $ d - (e – f / g) * h
2. a * b $ (c / h – f) $ d + k
3. a * (b – c) / ((d + e) * (f – g + h))

BINARY SEARCH TREE

Definition: Suppose T is a binary tree. Then T is called a **binary search tree** if each node N of T has the following property:

The value at N is greater than every value in the left subtree of N and less than every value in the right subtree of N.

Q: Construct a binary search tree from the following node values:
40, 20, 60, 10, 30, 50, 70, 35, 25, 20, 38, 32



H. W.

1. 60, 80, 40, 30, 10, 70, 90, 100, 95, 30, 36
2. E, D, P, B, C, F, G, H, T, X, W, Z, Y
3. JAN, FEB, MAR **...** NOV, DEC

Algorithms for traversal of a binary tree:

1. **PREORDER TRAVERSAL**
    a. Visit the node.
    b. Traverse the left subtree according to preorder traversal.
    c. Traverse the right subtree according to preorder traversal.

2. **INORDER TRAVERSAL**
    a. Traverse the left subtree according to inorder traversal.
    b. Visit the node.
    c. Traverse the right subtree according to inorder traversal.

3. **PREORDER TRAVERSAL**
    a. Traverse the left subtree according to postorder traversal.
    b. Traverse the right subtree according to postorder traversal.
    c. Visit the node.

```
/*
 Write a program to perform the following operations on a binary search
   tree
   1.  Insert a node.
   2.  Display preorder traversal.
   3.  Display inorder traversal.
   4.  Display postorder traversal.
   5.  Search for a particular value.
   6.  Delete a particular node (Binary Search Tree Deletion).
*/

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

struct node
{
        int info;
        struct node *left, *right;
};
typedef struct node ND;

void insert(ND **, int);
void preorder(ND *); /* recursive version */
void inorder(ND *); /* recursive version */
void postorder(ND *); /* recursive version */
ND* search(ND *, int);
void bst_delete(ND **, int);

int main(void) {
        ND * tree;
        int i, x;

        tree=NULL;
        do {
                printf("\n1: Insert a node");
```

```
		printf("\n2: Display preorder traversal ");
		printf("\n3: Display inorder traversal");
		printf("\n4: Display postorder traversal");
		printf("\n5: Search for a particular node");
		printf("\n6: Delete a particular node");
		printf("\n0: To exit");
		printf("\n\nEnter your choice: ");
		scanf("%d",&i);
		switch(i) {
			case 1:printf("\nGive The Node Value: ");
				scanf("%d",&x);
				insert(&tree, x);
				break;
			case 2:preorder(tree);
				break;
			case 3:inorder(tree);
				break;
			case 4:postorder(tree);
				break;
			case 5:printf("\nGive a particular node value: ");
				scanf("%d", &x);
				if( search( tree, x) ) {
					printf("\nSearch Successful");
				} else {
					printf("\nSearch Unsuccessful");
				}
				break;
			case 6:printf("\nGive a particular node value to delete: ");
				scanf("%d", &x);
				bst_delete(&tree, x);
				break;
			case 0:printf("\nThe End");
				break;
			default:printf("\nInvalid Choice");
		}
	}while(i);
	return 0;
}

void insert(ND **ptr, int item) {
	ND *temp, *current;
	temp=(ND *)malloc(sizeof(ND));
	if(!temp) {
		printf("\n Insufficient Memory");
		return;
	}
	temp->info=item;
	temp->left=temp->right=NULL;
	if(!*ptr) {
		*ptr=temp;
		return;
	}
	current=*ptr;
	while(current->info!=item) {
		if(current->info>item) {
			if(current->left!=NULL) {
				current=current->left;
			} else {
				current->left=temp;
				return;
			}
		} else {
			if(current->right!=NULL) {
				current=current->right;
			} else {
				current->right=temp;
				return;
			}
		}
	}
	free(temp);/*For Duplicate Node Value.*/
}

void preorder(ND *p) {
	if(!p) {
		return;
	}
	printf("\t%d",p->info);
	preorder(p->left);
	preorder(p->right);
}

void inorder(ND *p) {
	if(!p) {
		return;
	}
	inorder(p->left);
	printf("\t%d",p->info);
	inorder(p->right);
}

void postorder(ND *p) {
	if(!p) {
		return;
	}
	postorder(p->left);
	postorder(p->right);
	printf("\t%d",p->info);
}
```

```
ND* search(ND *p, int item) {
        while(p) {

                if(p->info==item) {
                        break;
                }
                if(p->info>item) {
                        p=p->left;
                } else {
                        p=p->right;
                }
        }
        return p;
}

void bst_delete(ND **ptr, int item) {
        ND *prev, *current, *s, *fs;
        prev=current=*ptr;
        while(current!=NULL) {
                if(current->info==item) {
                        break;
                }
                prev=current;
                if(current->info>item) {
                        current=current->left;
                } else {
                        current=current->right;
                }
        }
        if(current==NULL) {
                printf("\nVoid Deletion");
                return;
        }
        if(current->left==NULL && current->right==NULL) { /* Leaf Node */
                if(current==prev) { /* Tree has only ROOT & that ROOT to be deleted
*/
                        *ptr=NULL;
                } else {
                        if(prev->left==current) {
                                prev->left=NULL;
                        } else {
                                prev->right=NULL;
                        }
                }
                free(current);
        } else {
                if(current->left!=NULL  &&  current->right!=NULL)  {  /*  If  the  node
possess both cheldren */
                        fs=s=current->right;
                        while(s->left!=NULL) {
                                fs=s;
                                s=s->left;
                        }
                        current->info=s->info;
                        if(fs==s) {
                                current->right=s->right;
                        } else {
                                fs->left=s->right;
                        }
                        free(s);
                } else { /* If the node has only one child */
                        if(prev==current) { /* For root with only one child */
                                if(current->left!=NULL) {
                                        *ptr=current->left;
                                } else {
                                        *ptr=current->right;
                                }
                        } else { /* Other than ROOT & having only one child */
                                if(prev->left==current) {
                                        if(current->left!=NULL) {
                                                prev->left=current->left;
                                        } else {
                                                prev->left=current->right;
                                        }
                                } else {
                                        if(current->right!=NULL) {
                                                prev->right=current->right;
                                        } else {
                                                prev->right=current->left;
                                        }
                                }
                        }
                        free(current);
                }
        }
}
```

**************************END OF PART 1****************************