Data Structures

UNIT 3

INTRODUCTION

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a sub-tree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure below shows a tree and a non-tree.

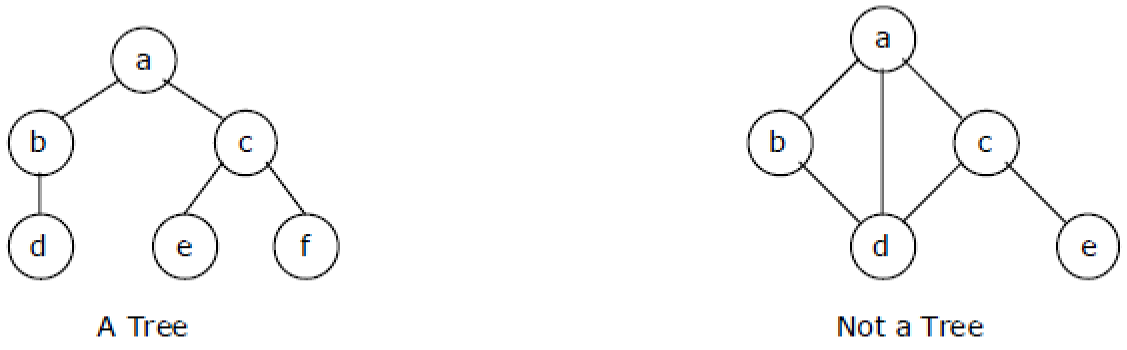


Fig : A Tree and a Non Tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the commonest form of tree data structure.

BINARY TREE

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree. A tree with no nodes is called as a null tree. A binary tree is shown in below Figure,

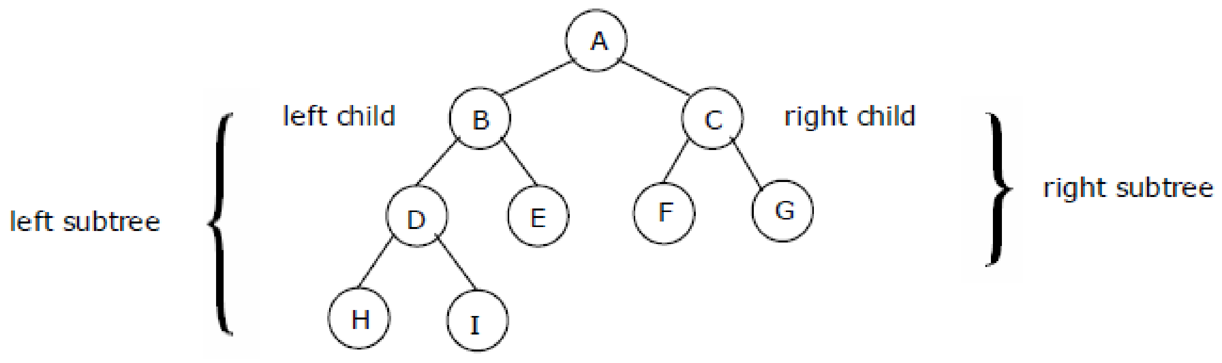


Fig : Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

TREE TERMINOLOGY

(a)LEAF NODE

A node with no children is called a leaf (or external node). A node which is not a leaf is called an internal node.

(b)PATH

A sequence of nodes n1, n2, . . .,nk, such that ni is the parent of ni + 1 for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself. For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

(c)SIBLINGS

The children of the same parent are called siblings. For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

(d)ANCESTOR AND DESCENDENT

If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

(e)SUBTREE

Any node of a tree, with all of its descendants is a subtree.

(f) LEVEL

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. The maximum number of nodes at any level is 2n.

(g) HEIGHT

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3. Depth

(h) DEPTH

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

(i) ASSIGNING LEVEL NUMBERS AND NUMBERING OF NODES FOR A BINARY TREE:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure,

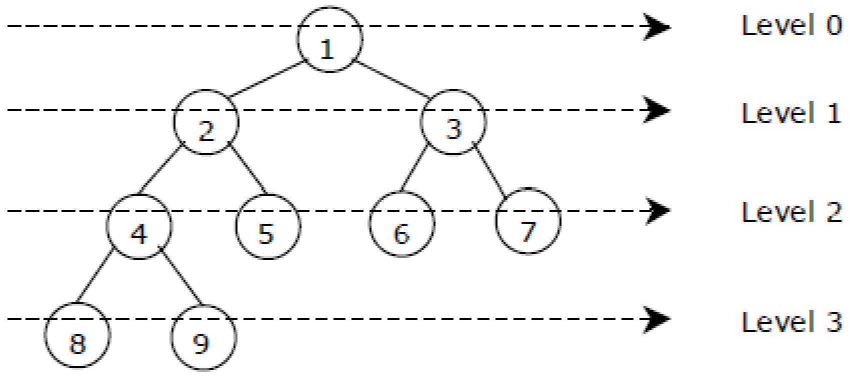


Fig : Level by level numbering of binary tree

PROPERTIES OF BINARY TREE:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

   a. Maximum number of leaves = 2h

   b. Maximum number of nodes = 2h + 1 - 1

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l

   node at level l.

4. The total number of edges in a full binary tree with n node is n - 1.

STRICTLY BINARY TREE:

If every non-leaf node in a binary tree has nonempty left and right sub-trees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictlybinary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

FULL BINARY TREE:

A full binary tree of height h has all its leaves at level h. Alternatively; All non-leaf nodes of a full binary tree have two children, and the leaf nodes have nochildren.A full binary tree with height h has 2h + 1 -1 nodes. A full binary tree of height his a strictly binary tree all of whose leaves are at level h. Figure 5.2.3(d)illustrates the full binary tree containing 15 nodes and of height 3.A full binary tree of height h contains 2h leaves and, 2h - 1 non-leaf nodes.

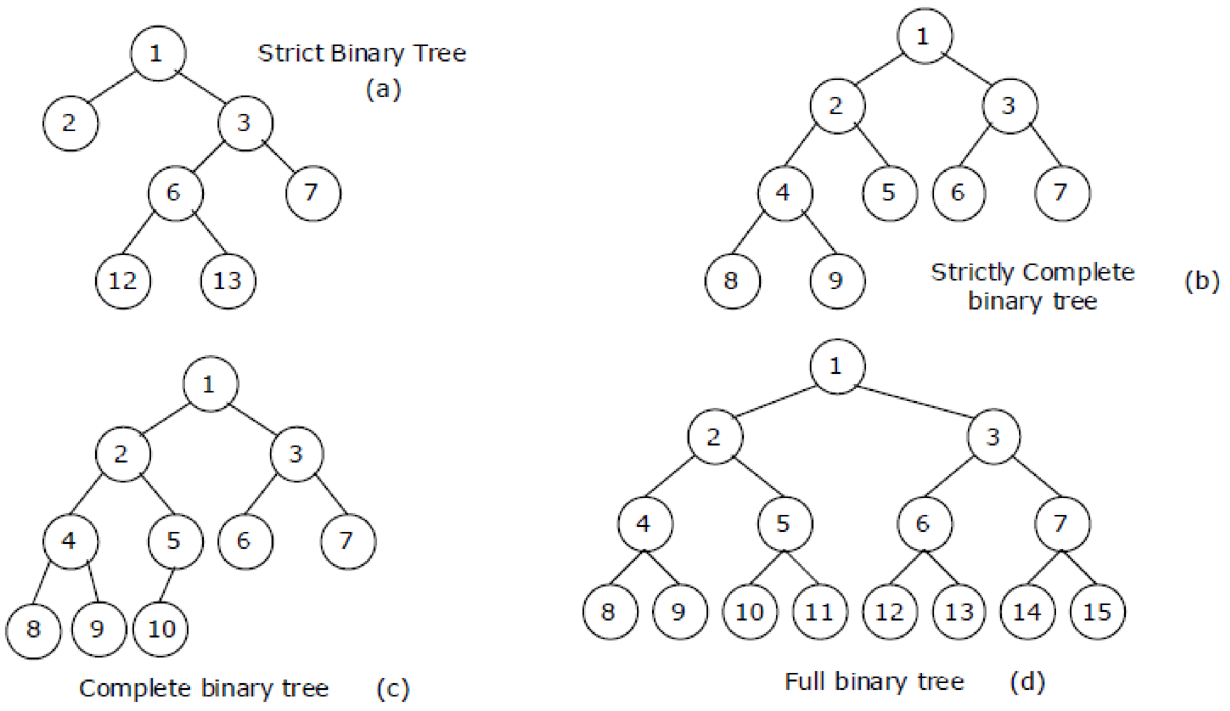


Fig : Examples of Binary Tree

COMPLETE BINARY TREE

A binary tree with n nodes is said to be complete if it contains all the first n nodes of the above numbering scheme. Figure shows examples of complete and incomplete binary trees. A complete binary tree of height h looks like a full binary tree down to level h-1,and the level h is filled from left to right.

A complete binary tree with n leaves that is not strictly binary has 2n nodes. Forexample, the tree of Figure c is a complete binary tree having 5 leaves and 10 nodes.

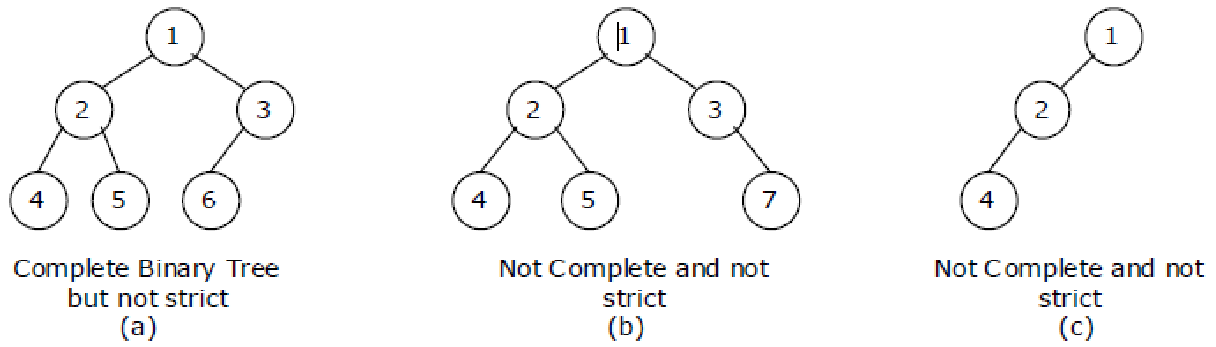


Fig : Examples of the complete and in-complete binary tree

INTERNAL AND EXTERNAL NODES:

We define two terms: Internal nodes and external nodes. An internal node is a treenode having at least one–key and possibly some children. It is sometimes convenientto have other types of nodes, called an external node, and pretend that all null childlinks point to such a node. An external node doesn't exist, but serves as a conceptualplace holder for nodes to be inserted. We draw internal nodes using circles, with letters as labels. External nodes are denotedby squares. The square node version is sometimes called an extended binary tree. Abinary tree with n internal nodes has n+ 1 external node. Figure 5.2.6 shows a sampletree illustrating both internal and external nodes.

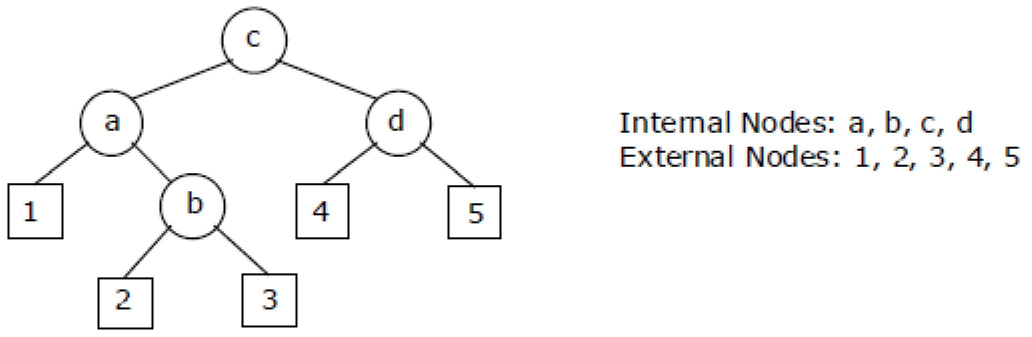


Fig: Internal and External nodes

BINARY SERACH TREE

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.

2. The keys in the left subtree are smaller than the key in the root.

3. The keys in the right subtree are larger than the key in the root.

4. The left and right subtrees are also binary search trees.

Figure (a) is a binary search tree, whereas figure (b) is not a binary searchtree.

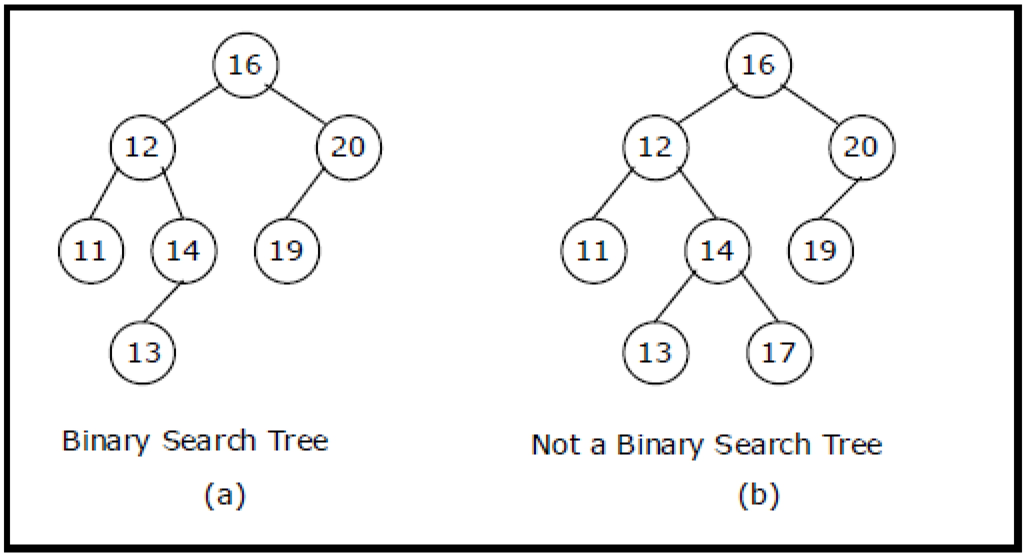


Fig: Example for Binary Search Tee

GENERAL TREE (M-ARY TREE):

If in a tree, the outdegree of every node is less than or equal to m, the tree is calledgeneral tree. The general tree is also called as an m-ary tree. If the outdegree of everynode is exactly equal to m or zero then the tree is called a full or complete m-ary tree.For m = 2, the trees are called binary and full binary trees.

SEARCH AND TRAVERSING TECHNIQUES:

Search involves visiting nodes in a tree in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination ofevery vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

1. Depth first search traversal.

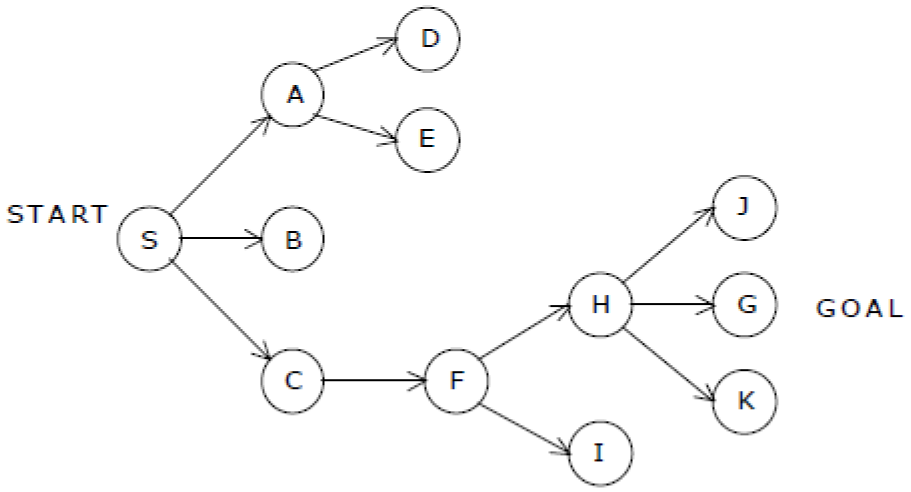
2. Breadth first search traversal.

1. DEPTH FIRST SEARCH:

In Depth first search, we begin with root as a start state, then some successor of thestart state, then some successor of that state, then some successor of that and so on,trying to reach a goal state. One simple way to implement depth first search is to use astack data structure consisting of root node as a start state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it “backs up” that means it goes to the immediately previous state or predecessorformally, the state whose successor was ‘S’ originally.

To illustrate this let us consider the tree shown below.



Suppose S is the start and G is the only goal state. Depth first search will first visit S,then A, then D. But D has no successors, so we must back up to A and try its secondsuccessor, E. But this doesn't have any successors either, so we back up to A again.But now we have tried all the successors of A and haven't found the goal state G so wemust back to 'S'. Now 'S' has a second successor, B. But B has no successors, so weback up to S again and choose its third successor C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state.So the solution path to the goal is S, C, F, H and G and the states considered were inorder S, A, D, E, B, C, F, H, J, G.

DISADVANTAGES :

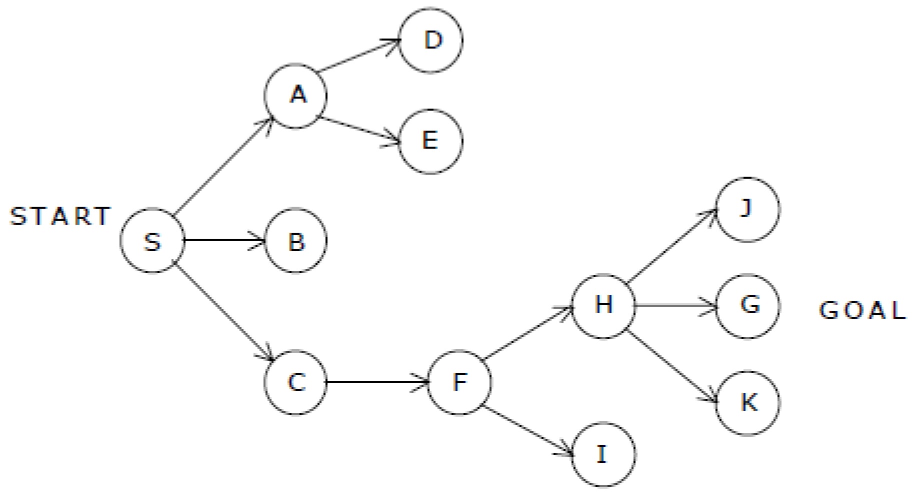
1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search cantravel around a cycle in the graph forever.To eliminate this keep a list of states previously visited, and never permitsearch to return to any of them.

2. We cannot come up with shortest solution to the problem.

2. BREADTH FIRST SEARCH:

Breadth-first search starts at root node S and "discovers" which vertices are reachable from S. Breadth-first search discovers vertices in increasing order of distance. Breadth first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all theimmediate successors of these, and so on until we find a goal node. Suppose S is thestart state and G is the goal state. In the figure, start state S is at level 0; A, B and Care at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4.

So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and thenstop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states inorder of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state.

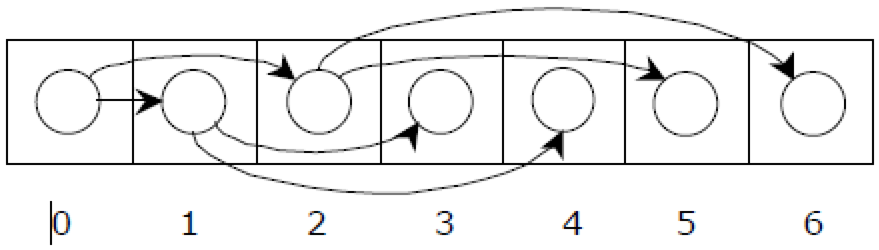
BINARY TREE REPRESENTATION

Binary Tree can be represented using arrays and linked list.

(a) ARRAY BASED REPRESENTATION

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, thismethod wastes no space. In this compact arrangement, if a node has an index i, itschildren are found at indices 2i+1 and 2i+2, while its parent (if any) is found at indexfloor((i-1)/2) (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory,expensive to grow and wastes space proportional to 2h - n for a tree of height h with nnodes.



(b) LINKED REPRESENTATION

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from themiddle of the tree, as it requires the moment of potentially many nodes to reflect thechange in level number of this node. To overcome this difficulty we represent thebinary tree in linked representation.

In linked representation each node in a binary has three fields, the left child field denoted as LeftChild, data field denoted as data and the right child field denoted asRightChild. If any sub-tree is empty then the corresponding pointer's LeftChild andRightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

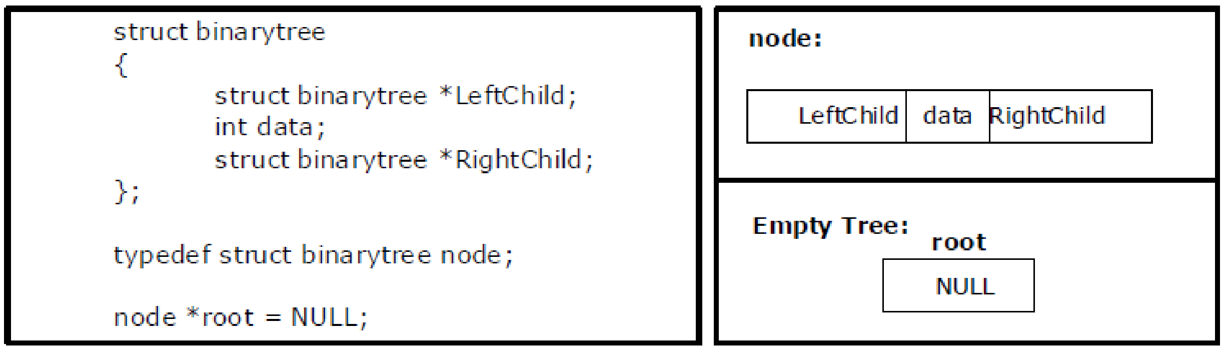
The advantage of using linked representation of binary tree is that:

i. Insertion and deletion involve no data movement and no movement of nodesexcept the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

i. Given a node structure, it is difficult to determine its parent node.
ii. Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6and the linked representation of binary tree using this node structure is given in figure5.2.7.

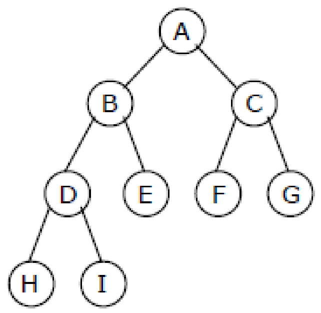


```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};

typedef struct binarytree node;

node *root = NULL;
```

Fig: Structure definition, node representation and empty tree.



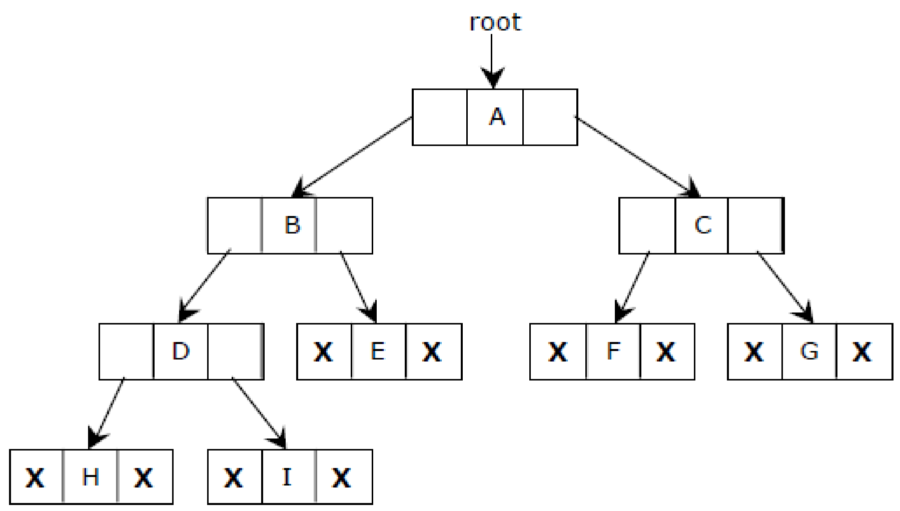


Fig : Linked Representation for the binary tree

BINARY TREE TRAVERSAL TECHNIQUES

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. Postorder
4. Level order

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time atwhich a root node is visited.

INORDER TRAVERSAL

In the case of inorder traversal, the root of each subtree is visited after its left subtreehas been traversed but before the traversal of its right subtree begins. The steps fortraversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
voidinorder(node *root)

{

        if(root != NULL)

        {

                inorder(root->lchild);

                print root -> data;

                inorder(root->rchild);

        }

}
```

PREORDER TRAVERSAL:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)

{

        if( root != NULL )

        {

                print root -> data;

                preorder (root ->lchild);

                preorder (root ->rchild);

        }

}
```

POST ORDER TRAVERSAL:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
voidpostorder(node *root)
{
        if( root != NULL )
        {
                postorder (root ->lchild);
                postorder (root ->rchild);
                print (root -> data);
        }
}
```

LEVEL ORDER TRAVERSAL:

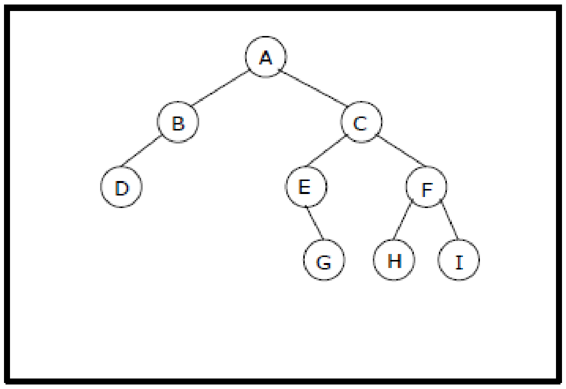
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure.So, it is not possible to develop a recursive procedure to traverse the binary tree inlevel order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
voidlevelorder()
{
        int j;
        for(j = 0; j <ctr; j++)
        {
                if(tree[j] != NULL)
                print tree[j] -> data;
        }
}
```

EXAMPLE 1:

Traverse the following binary tree in pre, post, inorder and level order.

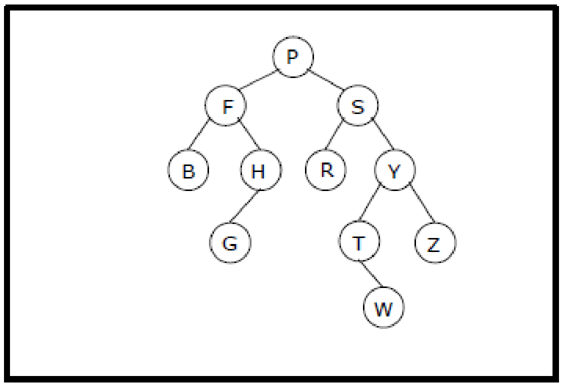


Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

EXAMPLE 2:

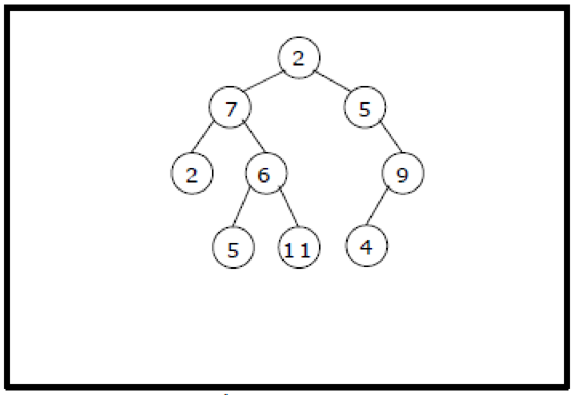


Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

EXAMPLE 3:



Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

BINARY TREE CREATION AND TRAVERSING UISNG ARRAYS:

```
/* The Program performs the following operations*/

1. Creates a complete Binary Tree

2. Inorder traversal

3. Preorder traversal

4. Postorder traversal

5. Level order traversal

6. Prints leaf nodes

7. Finds height of the tree created


# include <stdio.h>

# include <stdlib.h>

struct tree

{

        struct tree* lchild;

        char data[10];

        struct tree* rchild;

};

typedefstruct tree node;

intctr;

node *tree[100];

node* getnode()

{

        node *temp ;

        temp = (node*) malloc(sizeof(node));

        printf("\n Enter Data: ");

        scanf("%s",temp->data);

        temp->lchild = NULL;
        temp->rchild = NULL;

        return temp;

}

voidcreate_fbinarytree()

{

        int j, i=0;

        printf("\n How many nodes you want: ");

        scanf("%d",&ctr);

        tree[0] = getnode();

        j = ctr;

        j--;

        do

        {

                if( j > 0 ) /* left child */

                {

                        tree[ i * 2 + 1 ] = getnode();

                        tree[i]->lchild = tree[i * 2 + 1];

                        j--;

                }

                if( j > 0 ) /* right child */

                {

                        tree[i * 2 + 2] = getnode();

                        j--;

                        tree[i]->rchild = tree[i * 2 + 2];

                }

                i++;

        } while( j > 0);
```

```
}
voidinorder(node *root)
{
        if( root != NULL )
        {
                inorder(root->lchild);
                printf("%3s",root->data);
                inorder(root->rchild);
        }
}
void preorder(node *root)
{
        if( root != NULL )
        {
                printf("%3s",root->data);
                preorder(root->lchild);
                preorder(root->rchild);
        }
}
voidpostorder(node *root)
{
        if( root != NULL )
        {
                postorder(root->lchild);
                postorder(root->rchild);
                printf("%3s",root->data);
        }
```

```
}
voidlevelorder()
{
        int j;
        for(j = 0; j <ctr; j++)
        {
                if(tree[j] != NULL)
                        printf("%3s",tree[j]->data);
        }
}
voidprint_leaf(node *root)
{
        if(root != NULL)
        {
                if(root->lchild == NULL && root->rchild == NULL)
                printf("%3s ",root->data);
                print_leaf(root->lchild);
                print_leaf(root->rchild);
        }
}
int height(node *root)
{
        if(root == NULL)
        {
                return 0;
        }
        if(root->lchild == NULL && root->rchild == NULL)
```

```
                return 0;
        else
                return (1 + max(height(root->lchild), height(root->rchild)));
}
void main()
{
        int i;
        create_fbinarytree();
        printf("\n Inorder Traversal: ");
        inorder(tree[0]);
        printf("\n Preorder Traversal: ");
        preorder(tree[0]);
        printf("\n Postorder Traversal: ");
        postorder(tree[0]);
        printf("\n Level Order Traversal: ");
        levelorder();
        printf("\n Leaf Nodes: ");
        print_leaf(tree[0]);
        printf("\n Height of Tree: %d ", height(tree[0]));
}
```

/* BINARY TREE CREATION AND TRAVERSING USING LINKED LIST

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal

6. Prints leaf nodes

7. Finds height of the tree created

8. Deletes last node

9. Finds height of the tree created

```
# include <stdio.h>

# include <stdlib.h>

struct tree

{

        struct tree* lchild;

        char data[10];

        struct tree* rchild;

};

typedefstruct tree node;

node *Q[50];

intnode_ctr;

node* getnode()

{

        node *temp ;

        temp = (node*) malloc(sizeof(node));

        printf("\n Enter Data: ");

        fflush(stdin);

        scanf("%s",temp->data);

        temp->lchild = NULL;

        temp->rchild = NULL;

        return temp;

}

voidcreate_binarytree(node *root)
```

```
{
	char option;
	node_ctr = 1;
	if( root != NULL )
	{
		printf("\n Node %s has Left SubTree(Y/N)",root->data);
		fflush(stdin);
		scanf("%c",&option);
		if( option=='Y' || option == 'y')
		{
			root->lchild = getnode();
			node_ctr++;
			create_binarytree(root->lchild);
		}
		else
		{
			root->lchild = NULL;
			create_binarytree(root->lchild);
		}
		printf("\n Node %s has Right SubTree(Y/N) ",root->data);
		fflush(stdin);
		scanf("%c",&option);
		if( option=='Y' || option == 'y')
		{
			root->rchild = getnode();
			node_ctr++;
			create_binarytree(root->rchild);
```

```
                }
                else
                {
                        root->rchild = NULL;
                        create_binarytree(root->rchild);
                }
        }
}
voidmake_Queue(node *root,int parent)
{
        if(root != NULL)
        {
                node_ctr++;
                Q[parent] = root;
                make_Queue(root->lchild,parent*2+1);
                make_Queue(root->rchild,parent*2+2);
        }
}
delete_node(node *root, int parent)
{
        int index = 0;
        if(root == NULL)
                printf("\n Empty TREE ");
        else
        {
                node_ctr = 0;
                make_Queue(root,0);
```

```
                    index = node_ctr-1;
                    Q[index] = NULL;
                    parent = (index-1) /2;
                    if( 2* parent + 1 == index )
                              Q[parent]->lchild = NULL;
                    else
                              Q[parent]->rchild = NULL;
          }
          printf("\n Node Deleted ..");
}
voidinorder(node *root)
{
          if(root != NULL)
          {
                    inorder(root->lchild);
                    printf("%3s",root->data);
                    inorder(root->rchild);
          }
}
void preorder(node *root)
{
          if( root != NULL )
          {
                    printf("%3s",root->data);
                    preorder(root->lchild);
                    preorder(root->rchild);
          }
```

```
}
voidpostorder(node *root)
{
        if( root != NULL )
        {
                postorder(root->lchild);
                postorder(root->rchild);
                printf("%3s", root->data);
        }
}
voidprint_leaf(node *root)
{
        if(root != NULL)
        {
                if(root->lchild == NULL && root->rchild == NULL)
                        printf("%3s ",root->data);
                print_leaf(root->lchild);
                print_leaf(root->rchild);
        }
}
int height(node *root)
{
        if(root == NULL)
                return -1;
        else
                return (1 + max(height(root->lchild), height(root->rchild)));
}
```

```
voidprint_tree(node *root, int line)
{
        int i;
        if(root != NULL)
        {
                print_tree(root->rchild,line+1);
                printf("\n");
                for(i=0;i<line;i++)
                        printf("%s", root->data);
                print_tree(root->lchild,line+1);
        }
}
voidlevel_order(node *Q[],intctr)
{
        int i;
        for( i = 0; i <ctr ; i++)
        {
                if( Q[i] != NULL )
                        printf("%5s",Q[i]->data);
        }
}
int menu()
{
        intch;
        clrscr();
        printf("\n 1. Create Binary Tree ");
        printf("\n 2. Inorder Traversal ");
        printf("\n 3. Preorder Traversal ");

        printf("\n 4. Postorder Traversal ");

        printf("\n 5. Level Order Traversal");

        printf("\n 6. Leaf Node ");

        printf("\n 7. Print Height of Tree ");

        printf("\n 8. Print Binary Tree ");

        printf("\n 9. Delete a node ");

        printf("\n 10. Quit ");

        printf("\n Enter Your choice: ");

        scanf("%d", &ch);

        returnch;

}

void main()

{

        inti,ch;

        node *root = NULL;

        do

        {

                ch = menu();

                switch(ch)

                {

                case 1 :

                        if( root == NULL )

                        {

                                root = getnode();

                                create_binarytree(root);

                        }
                        else
                        {
                                printf("\n Tree is already Created ..");
                        }
                        break;
                case 2 :
                        printf("\n Inorder Traversal: ");
                        inorder(root);
                        break;
                case 3 :
                        printf("\n Preorder Traversal: ");
                        preorder(root);
                        break;
                case 4 :
                        printf("\n Postorder Traversal: ");
                        postorder(root);
                        break;
                case 5:
                        printf("\n Level Order Traversal ..");
                        make_Queue(root,0);
                        level_order(Q,node_ctr);
                        break;
                case 6 :
                        printf("\n Leaf Nodes: ");
                        print_leaf(root);
                        break;
                case 7 :
```

```
                printf("\n Height of Tree: %d ", height(root));

                break;

            case 8 :

                printf("\n Print Tree \n");

                print_tree(root, 0);

                break;

            case 9 :

                delete_node(root,0);

                break;

            case 10 :

                exit(0);

            }

            getch();

        }while(1);

}
```