Chapter 5

Binary Trees

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

In this chapter in particular, we will explain special type of trees known as binary trees, which are easy to maintain in the computer.

5.1. TREES:

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 subtree. 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 5.1.1 shows a tree and a non-tree.

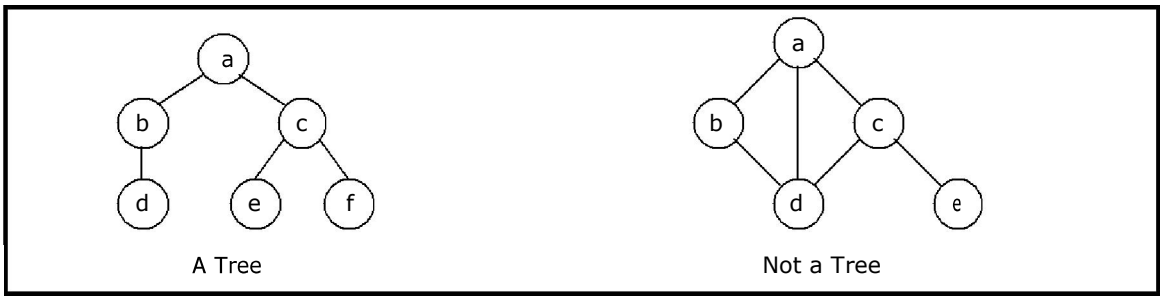


Figure 5.1.1 A Tree and a not a tree

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.

5.2. 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 figure 5.2.1.

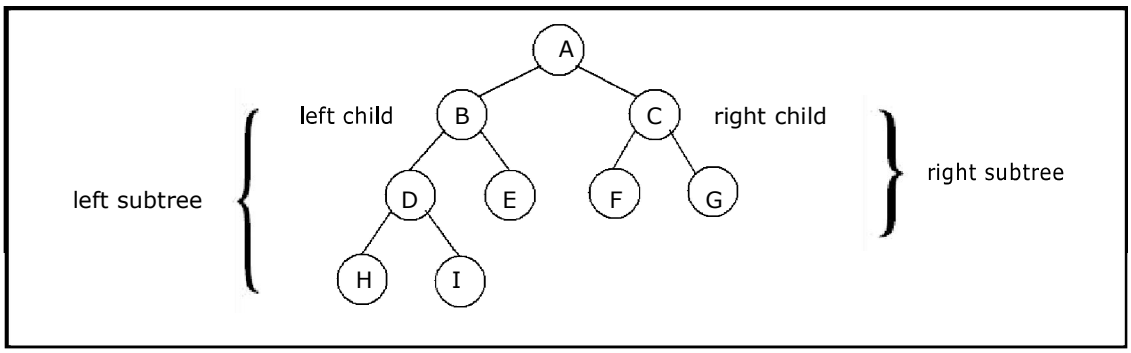


Figure 5.2.1. Binary Tree

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:

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*.

Path

A sequence of nodes n_1, n_2, . . ., n_k, such that n_i is the parent of n_{i+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.

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.

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.

Subtree

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

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* 2^n.

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

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.

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 5.2.2.

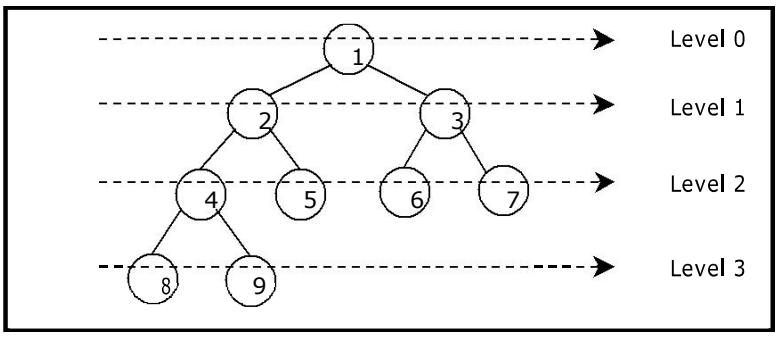


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

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 = 2^h
    b. Maximum number of nodes = $2^{h+1} - 1$
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 2^l 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 subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. 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 no children.

A full binary tree with height *h* has $2^{h+1} - 1$ nodes. A full binary tree of height h is 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 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (*tn*) = $\sum_{l=0}^{h} 2^l = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

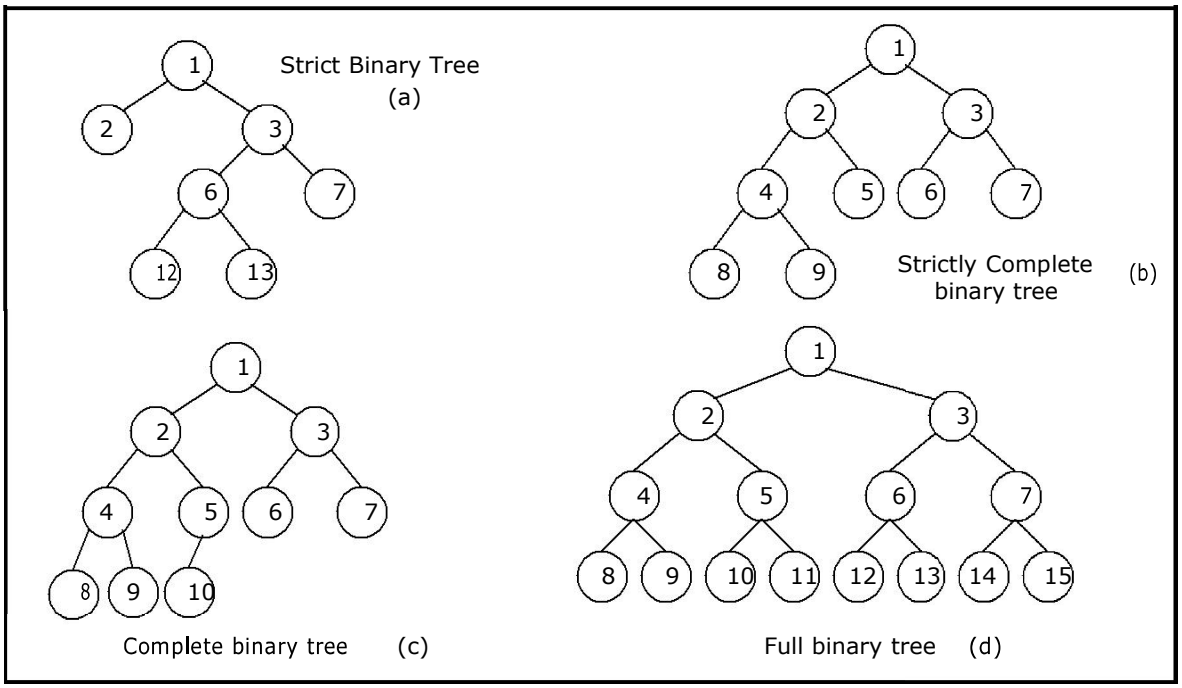


Figure 5.2.3. Examples of binary trees

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 5.2.4 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. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

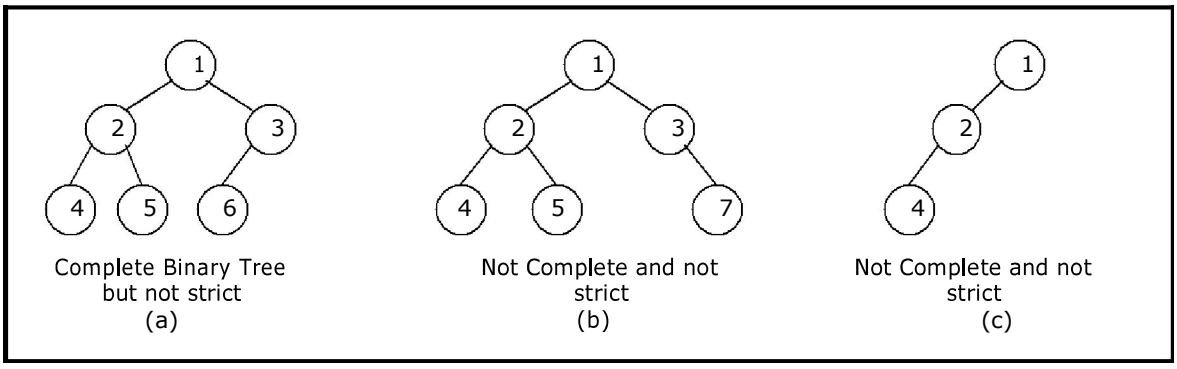


Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

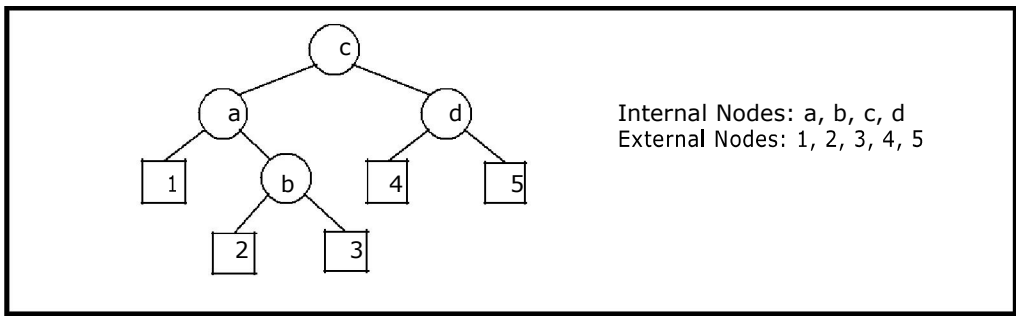


Figure 5.2.6. Internal and external nodes

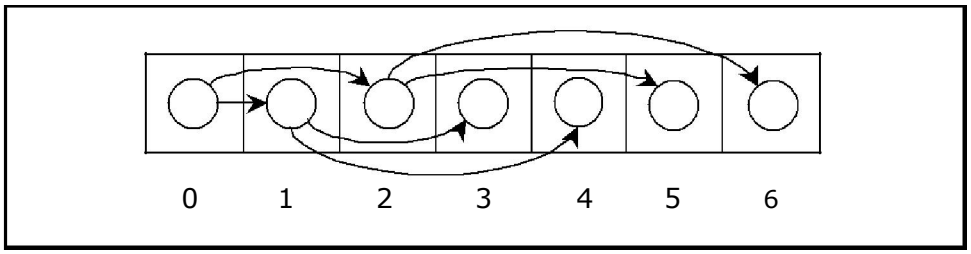
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((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 2^h - *n* for a tree of height *h* with *n* nodes.



Linked Representation (Pointer based):

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 the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary 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 as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild 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:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.
- 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 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

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

typedef struct binarytree node;

node *root = NULL;
```

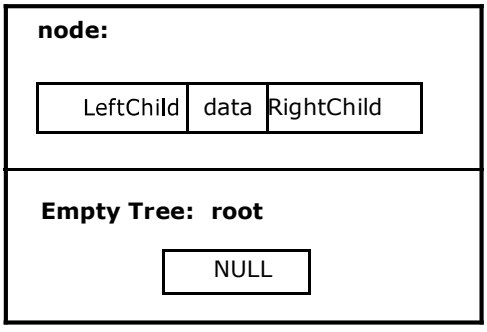


Figure 5.2.6. Structure definition, node representation and empty tree

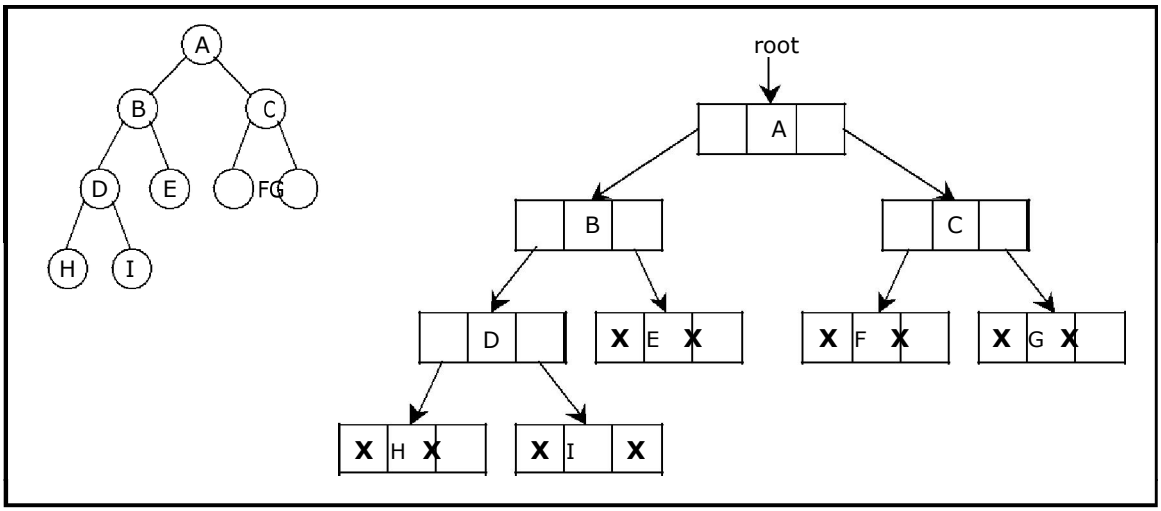


Figure 5.2.7. Linked representation for the binary tree

5.3. 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 at which a root node is visited.

5.3.1. Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing 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:

```
void inorder(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);
        }
}
```

Postorder 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:

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

Level order Traversal:

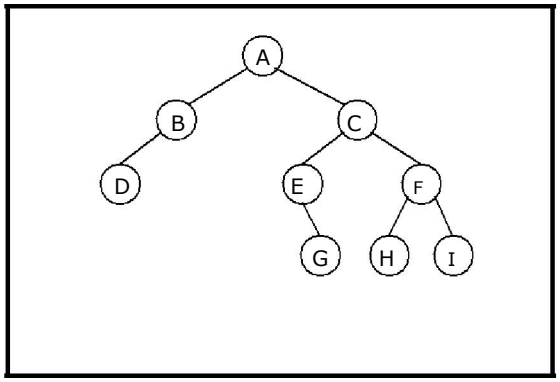
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth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

The algorithm for level order traversal is as follows:

```
void levelorder()
{
        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.



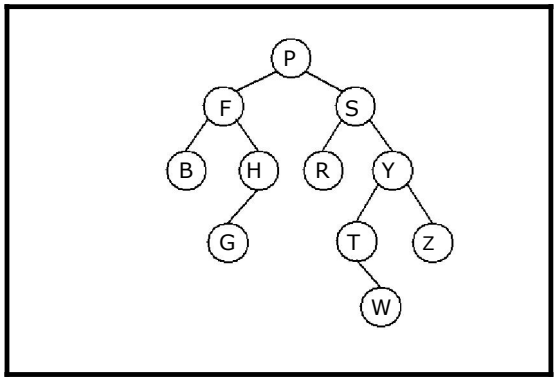
Bina ry T re e

- Preo rder traversal yields: A, B, D, C , E, G , F , H, I
- Posto rder traversal yields: D, B, G , E, H, I, F , C , A
- Ino rder traversal yields: D, B, A, E, G , C , H, F , I
- Level o rder traversal yields: A, B, C , D, E, F , G , H, I

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Example 2:

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



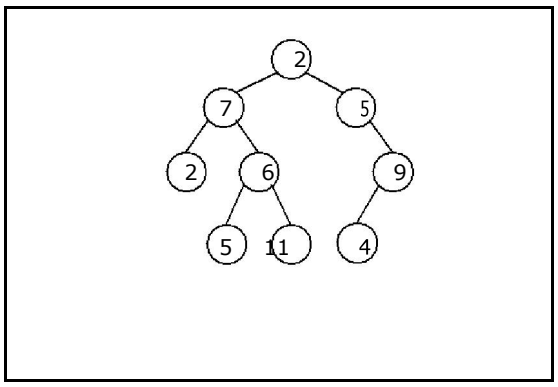
Bina ry T re e

- Preo rder traversal yields: P, F , B, H, G , S, R, Y, T, W, Z
- Posto rder traversal yields: B, G , H, F , R, W, T, Z, Y, S, P
- Ino rder traversal yields: B, F , G , H, P, R, S, T, W, Y, Z
- Level o rder traversal yields: P, F , S, B, H, R, Y, G , T, Z, W

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Example 3:

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



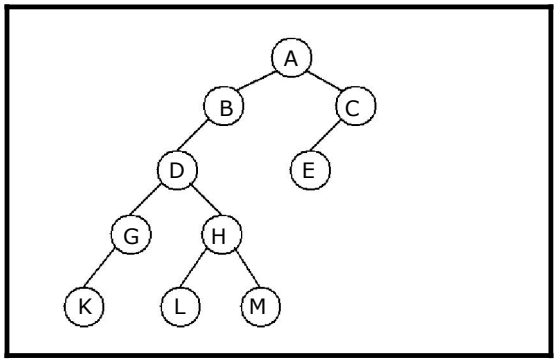
Bina ry T re e

- Preo rder traversal yields:
  2, 7 , 2 , 6 , 5 , 11 , 5 , 9 , 4
- Posto rder travarsal yields:
  2, 5 , 11 , 6 , 7 , 4 , 9 , 5 , 2
- Ino rder travarsal yields:
  2, 7 , 5 , 6 , 11 , 2 , 5 , 4 , 9
- Level o rder traversal yields:
  2, 7 , 5 , 2 , 6 , 9 , 5 , 11 , 4

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Example 4:

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



Bina ry T re e

- Preo rder traversal yields: A, B,
  D, G , K, H, L, M , C , E
- Posto rder travarsal yields: K,
  G , L, M , H, D, B, E, C , A
- Ino rder travarsal yields: K, G ,
  D, L, H, M , B, A, E, C
- Level o rder traversal yields: A,
  B, C , D, E, G , H, K, L, M

Pre, P o st, Inorder a nd lev e l order T rav ers ing

5.3.2. Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

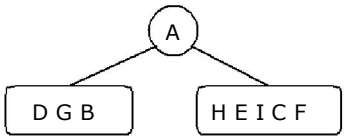
*From Preorder sequence **A** B D G C E H I F, the root is: A*

From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

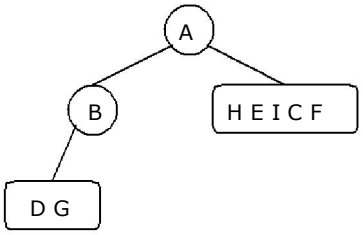


To find the root, left and right sub trees for D G B:

*From the preorder sequence **B** D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

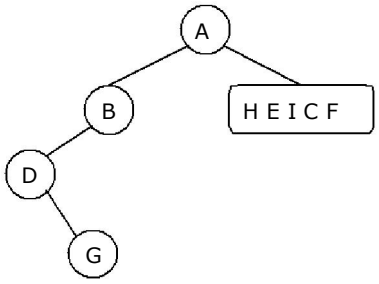


To find the root, left and right sub trees for D G:

From the preorder sequence **D** *G, the root of the tree is: D*

From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

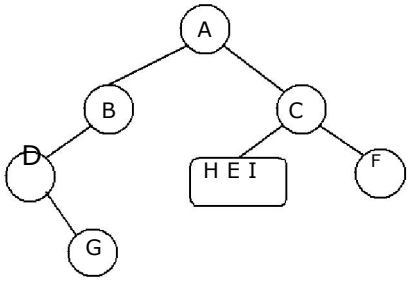


To find the root, left and right sub trees for H E I C F:

*From the preorder sequence **C** E H I F, the root of the left sub tree is: C*

From the inorder sequence *H E I **C** F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

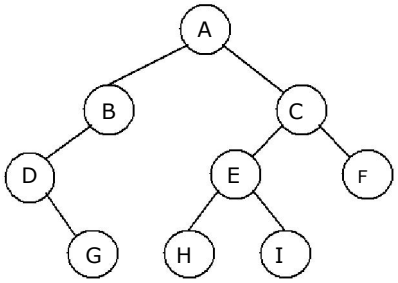


To find the root, left and right sub trees for H E I:

*From the preorder sequence **E** H I, the root of the tree is: E*

From the inorder sequence *H **E** I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

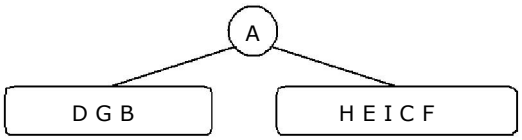
Solution:

From Postorder sequence G D B H I E F C **A**, *the root is: A*

From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B
Right sub tree is: H E I C F

The Binary tree upto this point looks like:

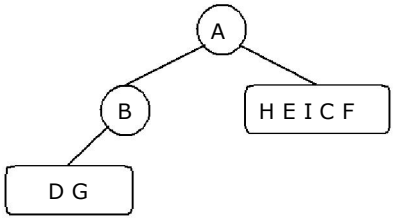


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

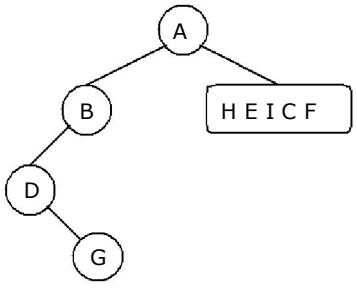


To find the root, left and right sub trees for D G:

*From the postorder sequence G **D**, the root of the tree is: D*

From the inorder sequence **D** *G*, we can find that is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:

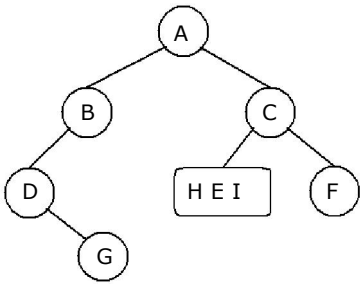


To find the root, left and right sub trees for H E I C F:

*From the postorder sequence H I E F **C**, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

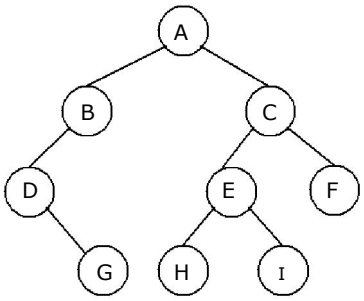


To find the root, left and right sub trees for H E I:

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

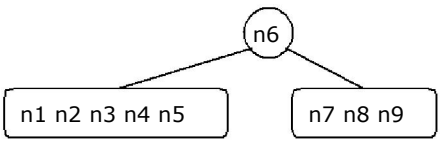
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

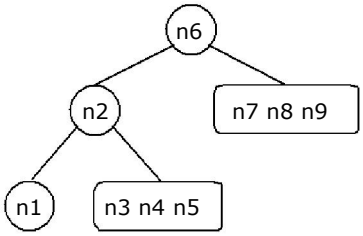
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

*From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

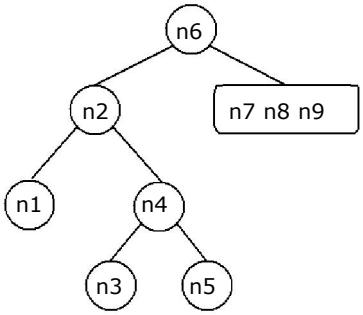


To find the root, left and right sub trees for n3 n4 n5:

*From the preorder sequence **n4** n3 n5, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

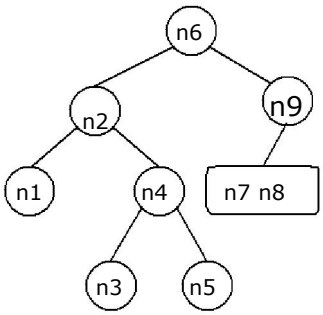


To find the root, left and right sub trees for n7 n8 n9:

*From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

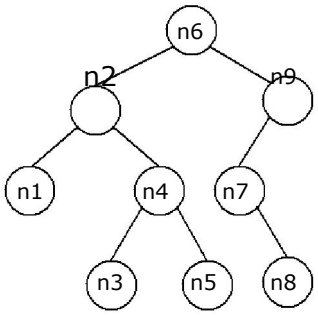


To find the root, left and right sub trees for n7 n8:

*From the preorder sequence **n7** n8, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

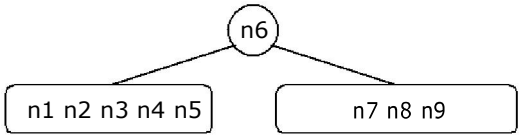
Solution:

From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5
Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:

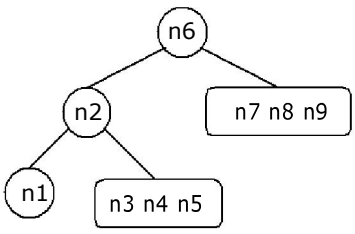


To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:

From the postorder sequence n1 n3 n5 n4 **n2**, *the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

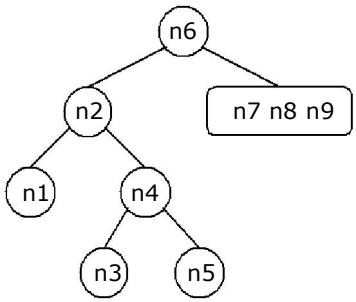
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence n3 n5 **n4***, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

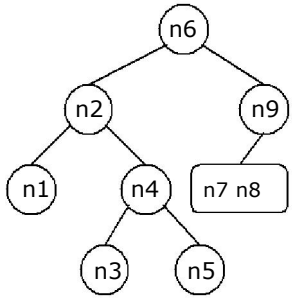


To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence n8 n7 **n9***, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

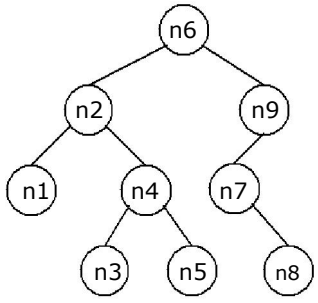
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

From the postorder sequence n8 **n7***, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



5.3.3. Binary Tree Creation and Traversal Using Arrays:

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

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

struct tree
{
        struct tree* lchild;
        char data[10];
        struct tree* rchild;
};

typedef struct tree node;
int ctr;
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;
}

void create_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);
}
```

```
void inorder(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);
        }
}

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




void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        printf("%3s",tree[j]->data);
        }
}



void print_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]));
}
```

5.3.4. Binary Tree Creation and Traversal Using Pointers:

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;
};

typedef struct tree node;
node *Q[50];
int node_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;
}
```

```
void create_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);
                }
        }
}

void make_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 ..");
}

void inorder(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);
        }
}

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

void print_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)));
}

void print_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(" ");
        printf("%s", root->data);
        print_tree(root->lchild,line+1);
    }
}

void level_order(node *Q[],int ctr)
{
    int i;
    for( i = 0; i < ctr ; i++)
    {
        if( Q[i] != NULL )
            printf("%5s",Q[i]->data);
    }
}

int menu()
{
    int ch;
    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);
    return ch;
}

void main()
{
    int i,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);
}
```

5.3.5. Non Recursive Traversal Algorithms:

At first glance, it appears that we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Algorithm inorder()

```
{
        stack[1] = 0
        vertex = root
top:    while(vertex ≠ 0)
        {
                push the vertex into the stack
                vertex = leftson(vertex)
```

```
        }

        pop the element from the stack and make it as vertex

        while(vertex ≠ 0)
        {
                print the vertex node
                if(rightson(vertex) ≠ 0)
                {
                        vertex = rightson(vertex)
                        goto top
                }
                pop the element from the stack and made it as vertex
        }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

```
Algorithm preorder( )
{
        stack[1] = 0
        vertex = root.
        while(vertex ≠ 0)
        {
                print vertex node
                if(rightson(vertex) ≠ 0)
                        push the right son of vertex into the stack.
                if(leftson(vertex) ≠ 0)
                        vertex = leftson(vertex)
                else
                        pop the element from the stack and made it as vertex
        }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

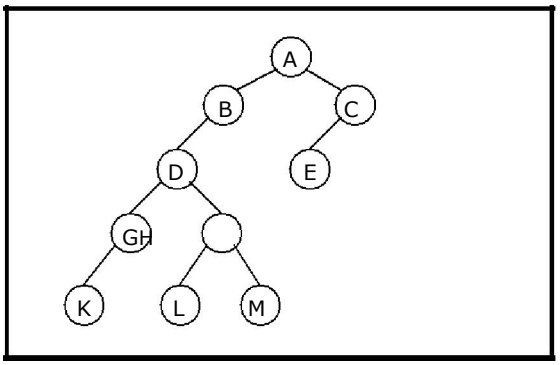
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

```
Algorithm postorder( )
{
        stack[1] = 0
        vertex = root

 top: while(vertex ≠ 0)
        {
                push vertex onto stack
                if(rightson(vertex) ≠ 0)
                        push – (vertex) onto stack
                vertex = leftson(vertex)
        }
        pop from stack and make it as vertex
        while(vertex > 0)
        {
                print the vertex node
                pop from stack and make it as vertex
        }
        if(vertex < 0)
        {
                vertex = - (vertex)
                goto top
        }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Bina ry T re e

- Preo rder traversal yields: A, B, D, G , K, H, L, M , C , E
- Posto rder travarsal yields: K, G , L, M , H, D, B, E, C , A
- Ino rder travarsal yields: K, G , D, L, H, M , B, A, E, C

Pre, P o st a nd Inorder T rav ers ing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A –C B D –H G K		PUSH the left most path of A with a -ve for right sons
	0 A –C B D –H	K G	POP all +ve nodes K and G
H	0 A –C B D	K G	Pop H

	0 A –C B D H –M L	K G	PUSH the left most path of H with a -ve for right sons
L	0 A –C B D H –M	K G L	POP all +ve nodes L
M	0 A –C B D H	K G L	Pop M
	0 A –C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A –C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0	K G L M H D B E C A	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

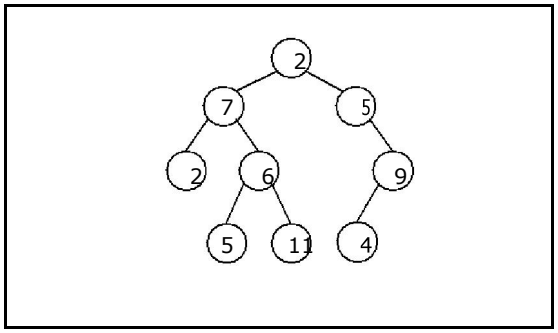
1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Bina ry T re e

- Preo rder traversal yields:
  2, 7 , 2 , 6 , 5 , 11 , 5 , 9 , 4
- Posto rder travarsal yields:
  2, 5 , 11 , 6 , 7 , 4 , 9 , 5 , 2
- Ino rder travarsal yields:
  2, 7 , 5 , 6 , 11 , 2 , 5 , 4 , 9

Pre, P o st a nd In order T rav ers ing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
6	0 2	2 7 5 6	
11	0 2 11	2 7 5 6	
11	0 2	2 7 5 6 11	
2	0	2 7 5 6 11 2	
5	0 5	2 7 5 6 11 2	
5	0	2 7 5 6 11 2 5	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
9	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 –5 7 –6 2		
2	0 2 –5 7 –6	2	
6	0 2 –5 7	2	
	0 2 –5 7 6 –11 5	2	
5	0 2 –5 7 6 –11	2 5	
11	0 2 –5 7 6 11	2 5	
	0 2 –5	2 5 11 6 7	
5	0 2 5 –9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5 11	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11 5	
9	0	2 7 2 6 5 11 5 9 4	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

5.4. Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 5.4.1 shows some more expression trees that represent arithmetic expressions given in infix form.

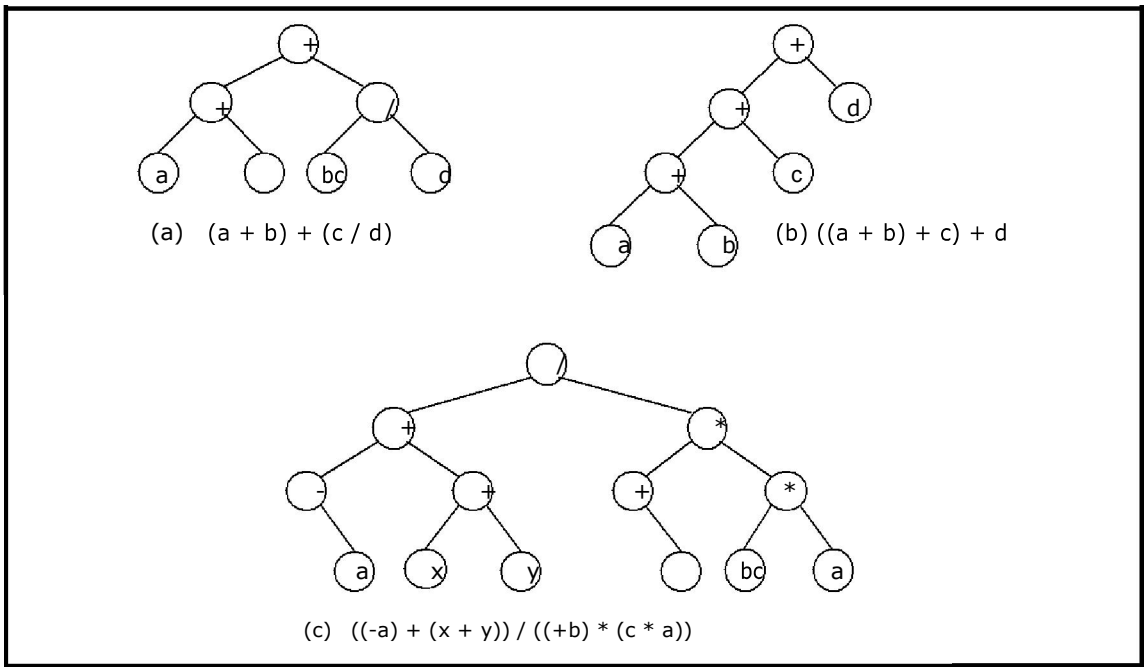


Figure 5.4.1 Expression Trees

An expression tree can be generated for the infix and postfix expressions.

An algorithm to convert a postfix expression into an expression tree is as follows:

1. Read the expression one symbol at a time.

2. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.

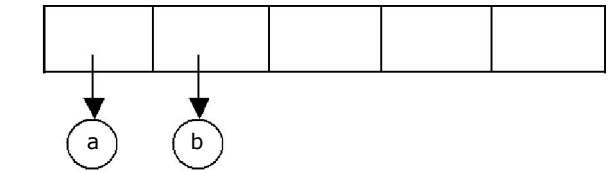
3. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

Example 1:

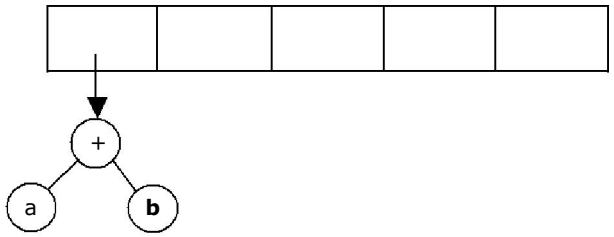
Construct an expression tree for the postfix expression: a b + c d e + * *

Solution:

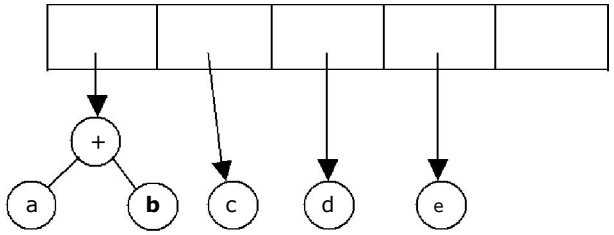
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



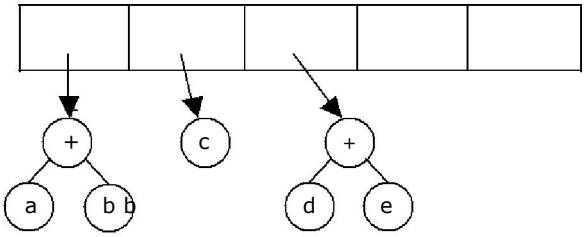
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



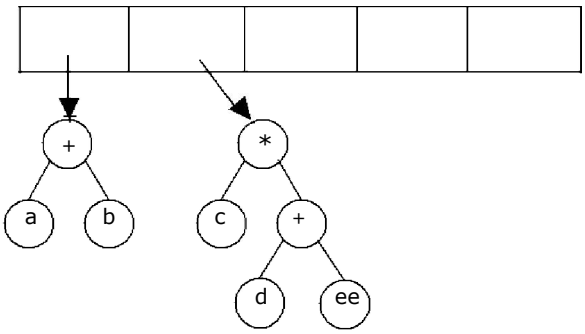
Next, c, d, and e are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



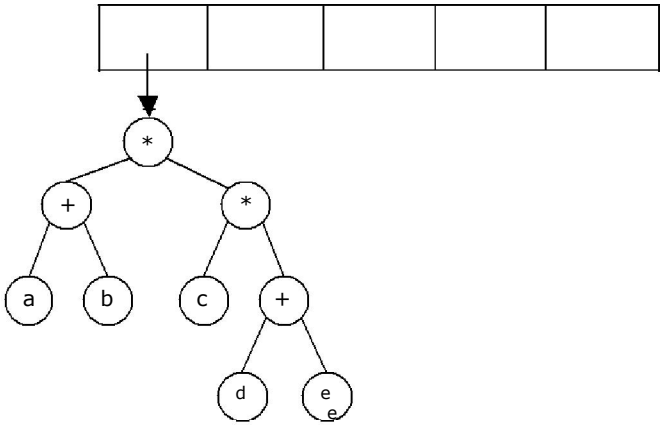
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: a + b * c * d + e

Preorder form of the expression: * + a b * c + d e

Postorder form of the expression: a b + c d e + * *

Example 2:

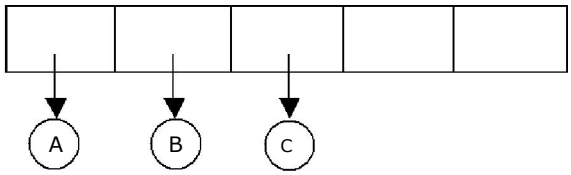
Construct an expression tree for the arithmetic expression:

(A + B * C) – ((D * E + F) / G)

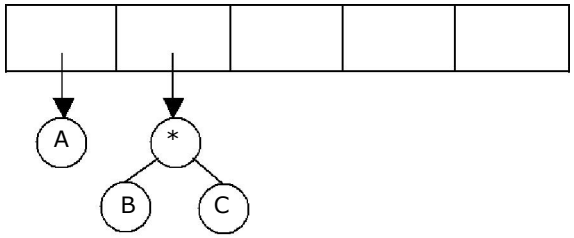
Solution:

First convert the infix expression into postfix notation. Postfix notation of the arithmetic expression is: A B C * + D E * F + G / -

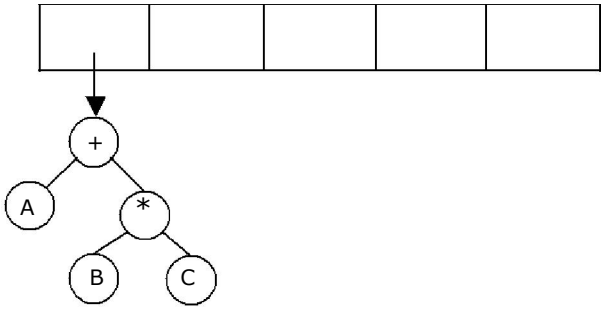
The first three symbols are operands, so we create one-node trees and pointers to three nodes pushed onto the stack.



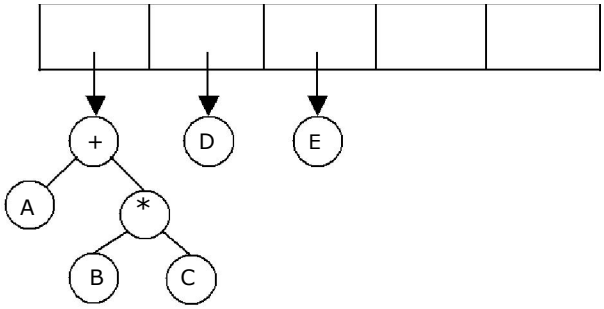
Next, a '*' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



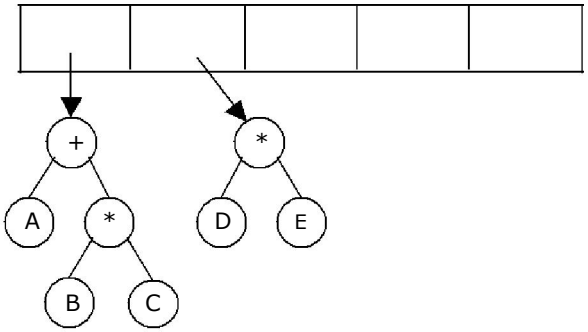
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



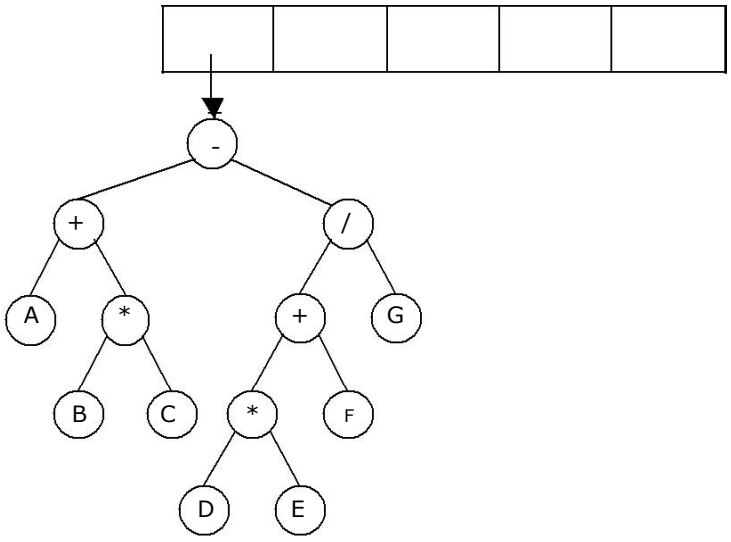
Next, D and E are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Proceeding similar to the previous steps, finally, when the last symbol is read, the expression tree is as follows:



5.4.1. Converting expressions with expression trees:

Let us convert the following expressions from one type to another. These can be as follows:

1. Postfix to infix
2. Postfix to prefix
3. Prefix to infix
4. Prefix to postfix

1. Postfix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

A. Create the expression tree from the postfix expression
B. Run inorder traversal on the tree.

2. Postfix to Prefix:

The following algorithm works for the expressions to convert postfix to prefix:

A. Create the expression tree from the postfix expression
B. Run preorder traversal on the tree.

3. Prefix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

A. Create the expression tree from the prefix expression
B. Run inorder traversal on the tree.

4. Prefix to postfix:

The following algorithm works for the expressions to convert postfix to prefix:

A. Create the expression tree from the prefix expression
B. Run postorder traversal on the tree.

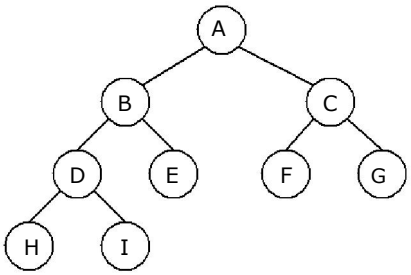
5.5. Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are 2n total links, there are n+1 null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

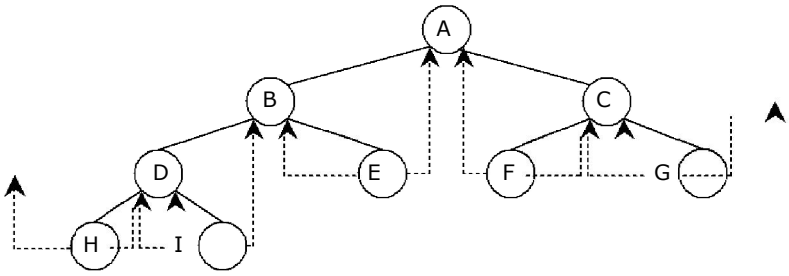
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the RCHILD(p) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:



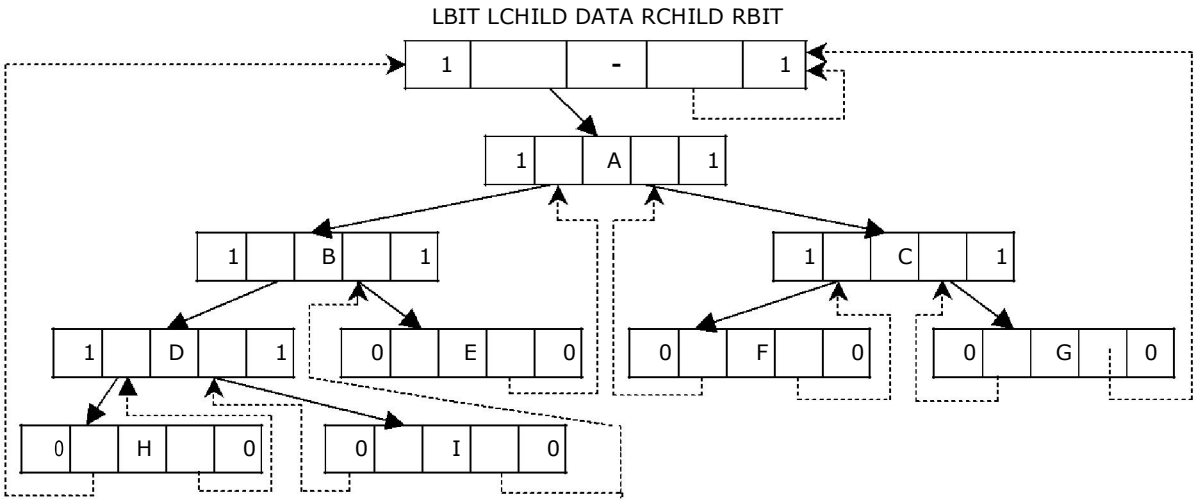
The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

LBIT(P) = 1 if LCHILD(P) is a normal pointer
LBIT(P) = 0 if LCHILD(P) is a Thread

RBIT(P) = 1 if RCHILD(P) is a normal pointer
RBIT(P) = 0 if RCHILD(P) is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



5.6. Binary Search 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 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

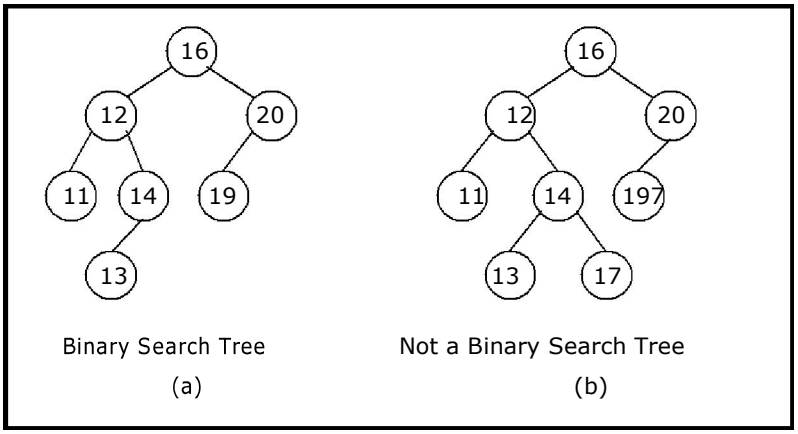


Figure 5.2.5. Examples of binary search trees

5.7. AVL Tree

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

AVL Tree Data structure

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

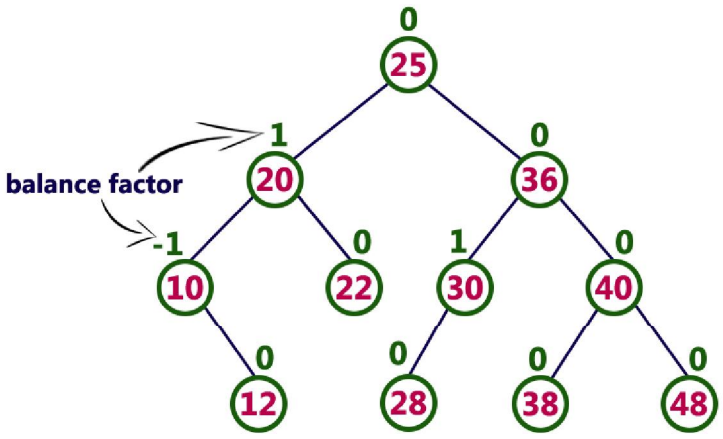
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = height Of LeftSubtree – height Of RightSubtree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

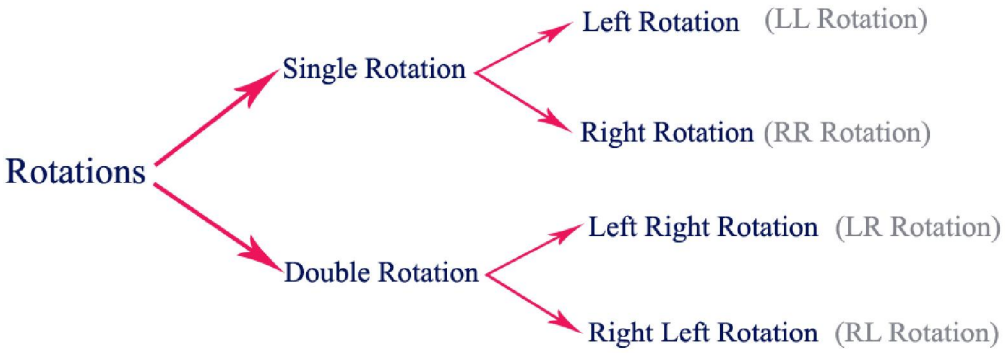
AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

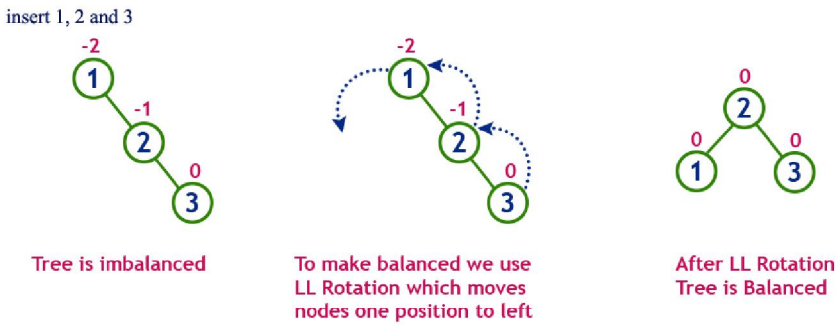
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.



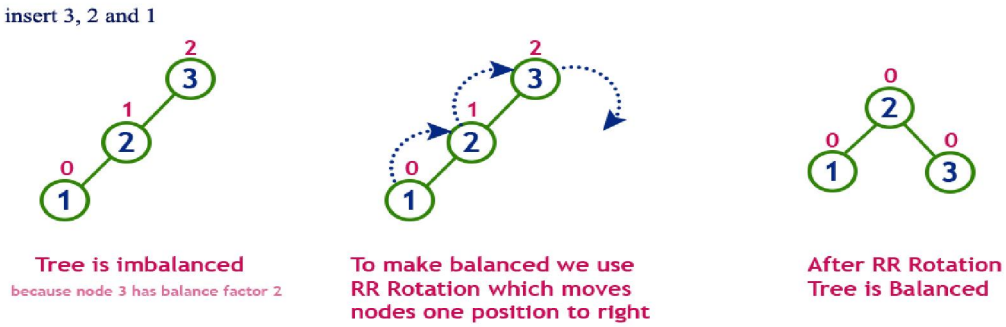
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



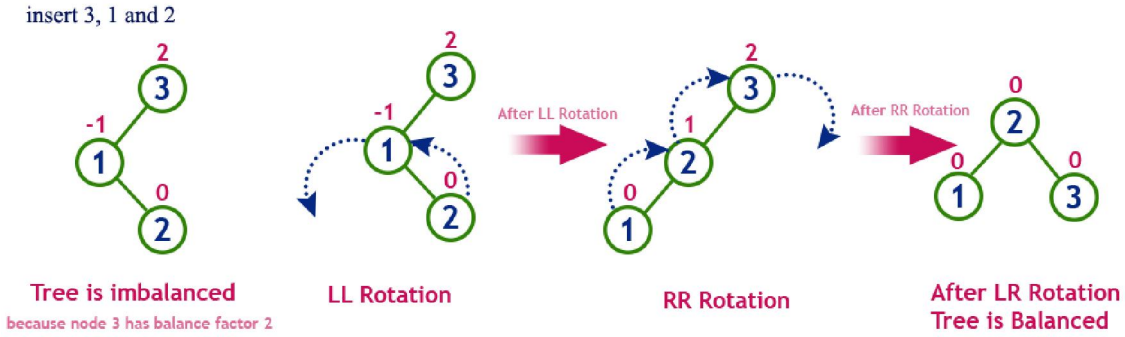
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



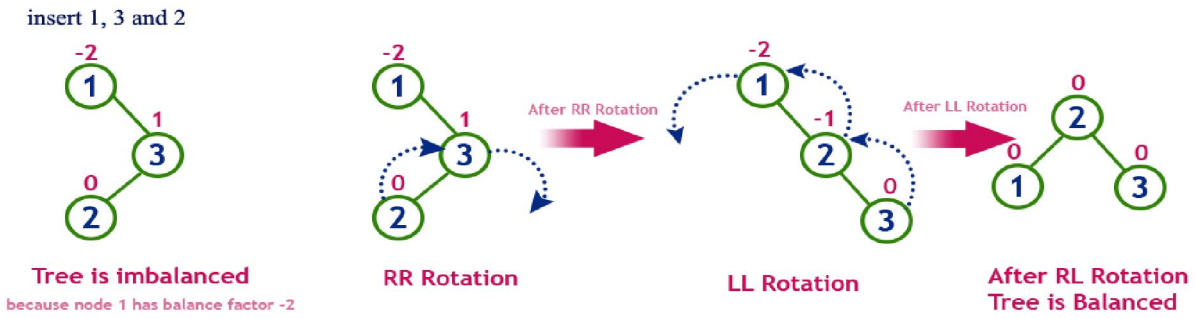
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

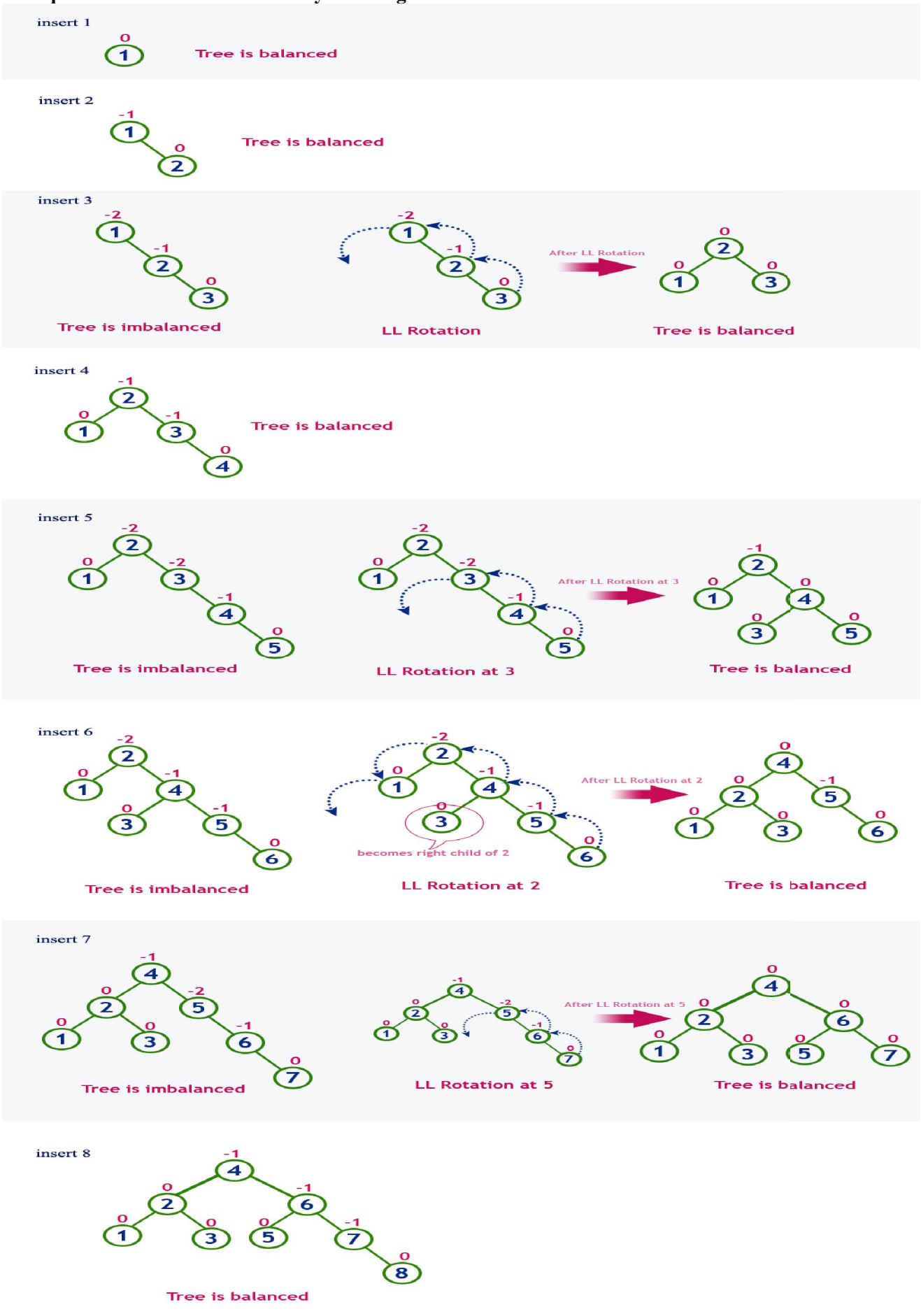
- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2 - After insertion, check the **Balance Factor** of every node.
Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.



5.8. Search and Traversal Techniques for m-ary trees:

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 of every vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

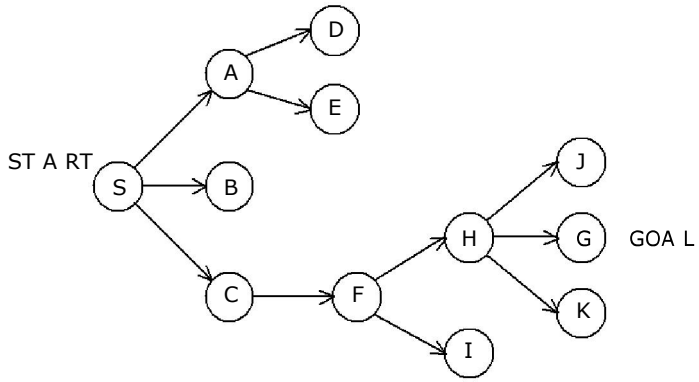
1. Depth first search or traversal.
2. Breadth first search or traversal.

5.8.1. Depth first search:

In Depth first search, we begin with root as a start state, then some successor of the start 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 a stack 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 predecessor formally, 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 second successor, 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 we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back 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 in order 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 can travel around a cycle in the graph forever.

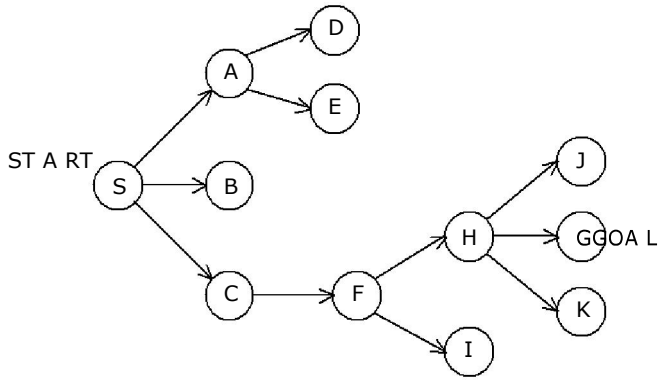
   To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

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

5.8.2. 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 the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are 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 then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order 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.

5.9. Sparse Matrices:

A sparse matrix is a two–dimensional array having the value of majority elements as null. The density of the matrix is the number of non-zero elements divided by the total number of matrix elements. The matrices with very low density are often good for use of the sparse format. For example,

$$A = \begin{matrix} 0 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{matrix}$$

As far as the storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise technique such that only non-null elements will be stored. The matrix A produces:

S =(3, 1) 1
(2, 2) 2
(3, 2) 3
(4, 3) 4
(1, 4) 5

The printed output lists the non-zero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.
In large number of applications, sparse matrices are involved. One approach is to use the linked list.

The program to represent sparse matrix:

```
/*      Check whether the given matrix is sparse matrix or not, if so then print in
        alternative form for storage.          */

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

main()
{
        int matrix[20][20], m, n, total_elements, total_zeros = 0, i, j;
        clrscr();
        printf("\n Enter Number of rows and columns: ");
        scanf("%d %d",&m, &n); total_elements = m *
        n;
        printf("\n Enter data for sparse matrix: ");
        for(i = 0; i < m ; i++)
        {
                for( j = 0; j < n ; j++)
                {
                        scanf("%d", &matrix[i][j]);
                        if( matrix[i][j] == 0)
                        {
                                total_zeros++;
                        }
                }
        }
        if(total_zeros > total_elements/2 )
        {
                printf("\n Given Matrix is Sparse Matrix..");
                printf("\n The Representaion of Sparse Matrix is: \n");
                printf("\n Row \t Col \t Value "); for(i = 0; i < m ;
                i++)
                {
                        for( j = 0; j < n ; j++)
                        {
                                if( matrix[i][j] != 0)
                                {
                                        printf("\n %d \t %d \t %d",i,j,matrix[i][j]);
                                }
                        }
                }
        }
        else
                printf("\n Given Matrix is Not a Sparse Matrix..");
}
```

EXCERCISES

1. How many different binary trees can be made from three nodes that contain the key value 1, 2, and 3?

2. a. Draw all the possible binary trees that have four leaves and all the nonleaf nodes have no children.
   b. Show what would be printed by each of the following.
   An inorder traversal of the tree
   A postorder traversal of the tree
   A preorder traversal of the tree

3. a. Draw the binary search tree whose elements are inserted in the following order:
   50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

   b. What is the height of the tree?
   c. What nodes are on level?
   d. Which levels have the maximum number of nodes that they could contain?
   e. What is the maximum height of a binary search tree containing these nodes? Draw such a tree?
   f. What is the minimum height of a binary search tree containing these nodes? Draw such a tree?

   g. Show how the tree would look after the deletion of 29, 59 and 47?

   h. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9 and 10 (in that order).

4. Write a "C" function to determine the height of a binary tree.

5. Write a "C" function to count the number of leaf nodes in a binary tree.

6. Write a "C" function to swap a binary tree.

7. Write a "C" function to compute the maximum number of nodes in any level of a binary tree. The maximum number of nodes in any level of a binary tree is also called the width of the tree.

8. Construct two binary trees so that their postorder traversal sequences are the same.

9. Write a "C" function to compute the internal path length of a binary tree.

10. Write a "C" function to compute the external path length of a binary tree.

11. Prove that every node in a tree except the root node has a unique parent.

12. Write a "C" function to reconstruct a binary tree from its preorder and inorder traversal sequences.

13. Prove that the inorder and postorder traversal sequences of a binary tree uniquely characterize the binary tree. Write a "C" function to reconstruct a binary tree from its postorder and inorder traversal sequences.

14. Build the binary tree from the given traversal techniques:

A. Inorder: g d h b e i a f j c
Preorder: a b d g h e i c f j

B. Inorder: g d h b e i a f j c
Postorder: g h d i e b j f c a

C. Inorder: g d h b e i a f j c
Level order: a b c d e f g h i j

15. Build the binary tree from the given traversal techniques:

A. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

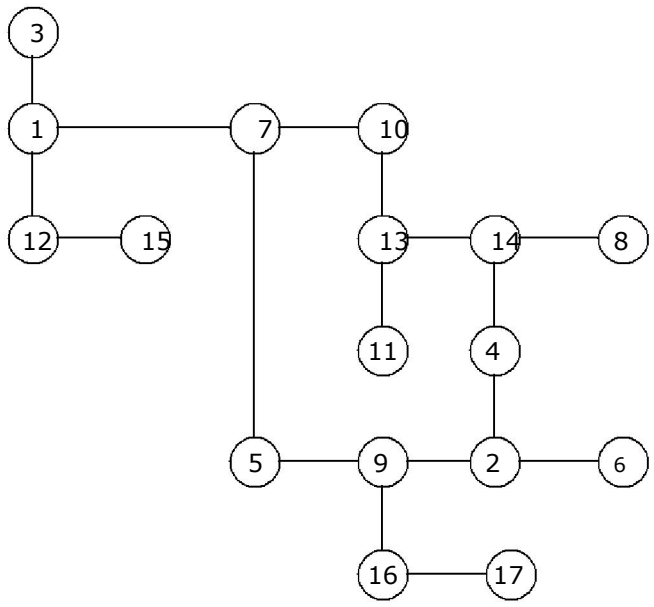
B. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

C. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Level order: n6 n2 n9 n1 n4 n7 n3 n5 n8

16. Build the binary tree for the given inorder and preorder traversals:

Inorder: E A C K F H D B G
Preorder: F A E K C D H G B

17. Convert the following general tree represented as a binary tree:



Multiple Choice Questions

1. The node that has no children is referred as: [ ]
   A. Parent node
   B. Root node
   C. Leaf node
   D. Sibblings

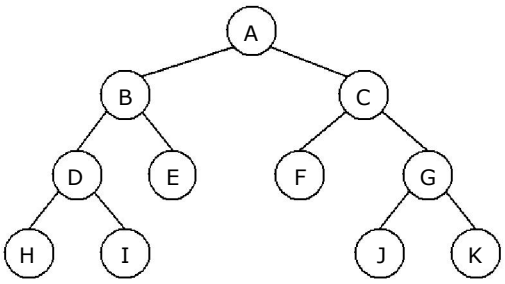
2. A binary tree in which all the leaves are on the same level is called as: [ ]
   A. Complete binary tree
   B. Full binary tree
   C. Strictly binary tree
   D. Binary search tree

3. How can the graphs be represented? [ ]
   A. Adjacency matrix
   B. Adjacency list
   C. Incidence matrix
   D. All of the above

4. The children of a same parent node are called as: [ ]
   A. adjacent node
   B. non-leaf node
   C. Sibblings
   D. leaf node

5. A tree with n vertices, consists of_____ edges. [ ]
   A. n – 1
   B. n - 2
   C. n
   D. log n

6. The maximum number of nodes at any level is: [ ]
   A. n
   B. 2^n
   C. n + 1
   D. 2n

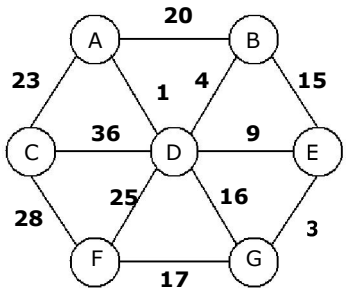


FI GURE 1

7. For the Binary tree shown in fig. 1, the in-order traversal sequence is: [ ]
   A. A B C D E F G H I J K
   B. H I D E B F J K G C A
   C. H D I B E A F C J G K
   D. A B D H I E C F G J K

8. For the Binary tree shown in fig. 1, the pre-order traversal sequence is: [ ]
   A. A B C D E F G H I J K
   B. H I D E B F J K G C A
   C. H D I B E A F C J G K
   D. A B D H I E C F G J K

9. For the Binary tree shown in fig. 1, the post-order traversal sequence is: [ ]
   A. A B C D E F G H I J K
   B. H I D E B F J K G C A
   C. H D I B E A F C J G K
   D. A B D H I E C F G J K



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 2 and its adjacency list

10. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 2 shown above: [ ]
    A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
    B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
    C. both A and B
    D. none of the above

11. For the figure 2 shown above, the cost of the minimal spanning tree is: [ ]
    A. 57 C. 48
    B. 68 D. 32

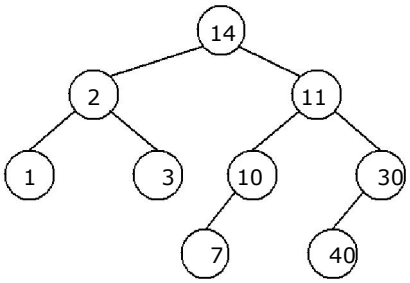


FIGURE 3

12. For the figure 3, how many leaves does it have? [ ]
    A. 2 C. 6
    B. 4 D. 8

13. For the figure 3, how many of the nodes have at least one sibling? [ ]
    A. 5 C. 7
    B. 6 D. 8

14. For the figure 3, How many descendants does the root have? [ ]
    A. 0 C. 4
    B. 2 D. 8

15. For the figure 3, what is the depth of the tree? [ ]
    A. 2 C. 4
    B. 3 D. 8

16. For the figure 3, which statement is correct? [ ]
    A. The tree is neither complete nor full.
    B. The tree is complete but not full.
    C. The tree is full but not complete.
    D. The tree is both full and complete.

17. There is a tree in the box at the top of this section. What is the order of nodes visited using a pre-order traversal? [ ]
    A. 1 2 3 7 10 11 14 30 40
    B. 1 2 3 14 7 10 11 40 30
    C. 1 3 2 7 10 40 30 11 14
    D. 14 2 1 3 11 10 7 30 40

18. There is a tree in the box at the top of this section. What is the order of nodes visited using an in-order traversal? [ ]
    A. 1 2 3 7 10 11 14 30 40
    B. 1 2 3 14 7 10 11 40 30
    C. 1 3 2 7 10 40 30 11 14
    D. 14 2 1 3 11 10 7 30 40

19. There is a tree in the box at the top of this section. What is the order of nodes visited using a post-order traversal? [ ]
    A. 1 2 3 7 10 11 14 30 40
    B. 1 2 3 14 7 10 11 40 30
    C. 1 3 2 7 10 40 30 11 14
    D. 14 2 1 3 11 10 7 30 40

20. What is the minimum number of nodes in a full binary tree with depth 3? [ ]
    A. 3
    B. 4
    C. 8
    D. 15

21. Select the one true statement. [C
    A. Every binary tree is either complete or full.
    B. Every complete binary tree is also a full binary tree.
    C. Every full binary tree is also a complete binary tree.
    D. No binary tree is both complete and full.

22. Suppose T is a binary tree with 14 nodes. What is the minimum possible depth of T?
    A. 0
    B. 3
    C. 4
    D. 5

23. Select the one FALSE statement about binary trees: [ ]
    A. Every binary tree has at least one node.
    B. Every non-empty tree has exactly one root node.
    C. Every node has at most two children.
    D. Every non-root node has exactly one parent.

24. Consider the node of a complete binary tree whose value is stored in data[i] for an array implementation. If this node has a right child, where will the right child's value be stored? [ ]
    A. data[i+1]
    B. data[i+2]
    C. data[2*i + 1]
    D. data[2*i + 2]

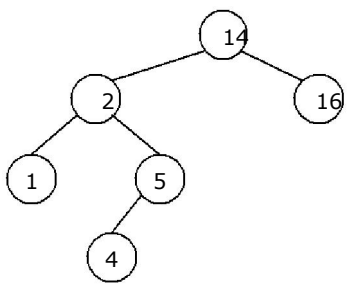
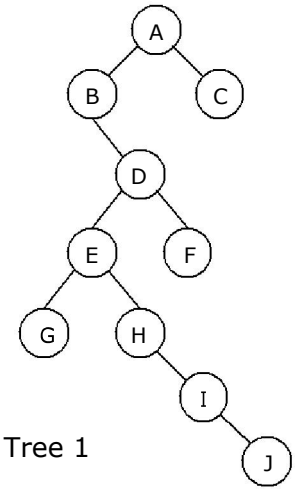


Figure 4

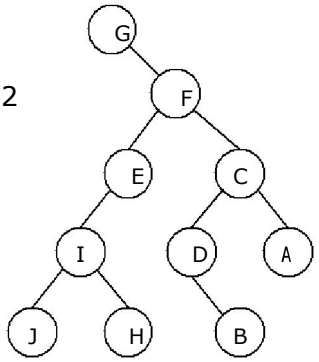
25. For the binary search tree shown in figure 4, Suppose we remove the root, [ ] replacing it with something from the left subtree. What will be the new root?

A. 1
B. 2
C. 4
D. 5
E. 16



Tree 1

Tree 2

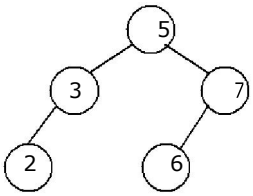


26. Which traversals of tree 1 and tree 2, will produce the same sequence of node names? [ ]

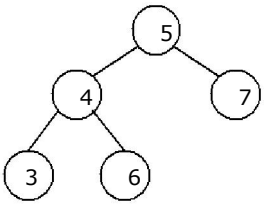
A. Preorder, Postorder
B. Postorder, Postorder
C. Postorder, Inorder
D. Inorder, Inorder

27. Which among the following is not a binary search tree? [ ]

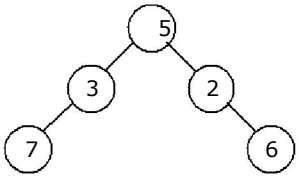
A.



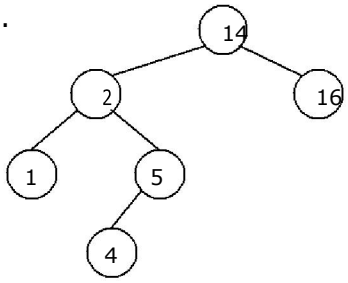
C.

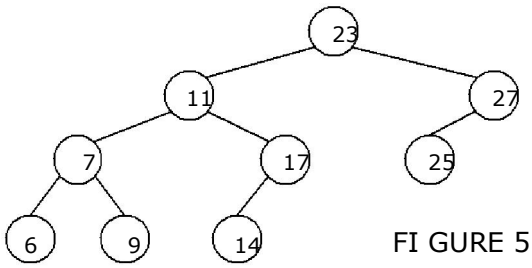


B.



D.





FI GURE 5

28. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what node will be at the root? [ ]
    A. 11 C. 27
    B. 25 D. 14

29. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what parent → child pair does not occur in the tree? [ ]
    A. 25 → 27 C. 11 → 7
    B. 27 → 11 D. 7 → 9

30. The number of nodes in a complete binary tree of depth d is: [ ]
    A. 2d C. 2^k
    B. $2^k - 1$ D. none of the above

31. The depth of a complete binary tree with n nodes is: [ ]
    A. log n C. $\log_2 n + 1$
    B. n^2 D. 2n

32. If the inorder and preorder traversal of a binary tree are D, B, F, E, G, H, A, C and A, B, D, E, F, G, H, C respectively then, the postorder traversal of that tree is: [ ]
    A. D, F, H, G, E, B, C, A C. F, H, D, G, E, B, C, A
    B. D, F, G, A, B, C, H, E D. D, F, H, G, E, B, C, A

33. The data structure used by level order traversal of binary tree is: [ ]
    A. Queue C. linked list
    B. Stack D. none of the above