**Binary Tree Traversal**

The item insertion, deletion, and lookup operations require that the binary tree be traversed. Thus, the most common operation performed on a binary tree is to traverse the binary tree, or visit each node of the binary tree. A ***traversal*** of a tree *T* is a systematic way of accessing, or “visiting,” all the nodes of *T*. In this section, we present a basic traversal scheme for trees, called preorder traversal. In the next section, we study another basic traversal scheme, called postorder traversal.

As you can see from the diagram of a binary tree, the traversal must start at the root node because there is a pointer to the root node. For each node, we have two choices:

• Visit the node first.

• Visit the subtrees first.

These choices lead to three different traversals of a binary tree—Inorder, preorder, and

postorder.

Inorder Traversal

In an inorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.

2. Visit the node.

3. Traverse the right subtree.

Preorder Traversal

In a preorder traversal, the binary tree is traversed as follows:

1. Visit the node.

2. Traverse the left subtree.

3. Traverse the right subtree.

Postorder Traversal

In a postorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.

2. Traverse the right subtree.

3. Visit the node.

Definition: The height of a binary tree is the number of nodes on the longest path from the root to a leaf.

Suppose that a pointer p to the root node of a binary tree is given. We next describe the C++ function height to find the height of the binary tree. The pointer to the root node is passed as a parameter to the function height.

If the binary tree is empty, the height is 0. Suppose that the binary tree is nonempty. To find the height of the binary tree, we first find the height of the left subtree and the height of the right subtree. We then take the maximum of these two heights and add 1 to find the height of the binary tree.

**Algorithm** depth(*T*, *p*):

**if** *p*.isRoot() **then**

**return** 0

**else**

**return** 1+depth(*T*, *p*.parent())

***Full binary tree*** A binary tree is a *full binary tree* if it contains the maximum possible

number of nodes in all levels. Figure shows a full binary tree of height 2.

In a full binary tree, each node has two children or no child at all.





***Left skewed binary tree*** If the right subtree is missing in every node of a tree, we call it

a *left skewed tree*

***Strictly binary tree*** If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a *strictly binary tree*.



***Complete binary tree*** A binary tree is said to be a *complete binary tree* if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the

last level appear as far left as possible.



Binary Tree Traversal

The item insertion, deletion, and lookup operations require that the binary tree be

traversed.

A ***traversal*** of a tree *T* is a systematic way of accessing, or “visiting,” all the nodes of *T*.

Thus, the most common operation performed on a binary tree is to traverse

the binary tree, or visit each node of the binary tree. As you can see from the diagram of a

binary tree, the traversal must start at the root node because there is a pointer to the root

node. For each node, we have two choices:

• Visit the node first.

• Visit the subtrees first.

There are various traversal methods. For a systematic traversal,

it is better to visit each node (starting from the root) and its

two subtrees in the same way. In other words, when traversing,

we need to treat each node and its subtree in the same fashion. If

we let L, D, and R stand for moving left, data, and moving right,

respectively (following Fig. ), when at a node, then there are six possible

combinations—LDR, LRD, DLR, DRL, RDL, and RLD.





Consider the binary tree shown in Fig.. This tree represents a binary tree.

Let us see the result of each of the six traversals.

LDR: 5 6 3 8 4

LRD: 5 6 3 8 4

DLR: 5 6 3 8 4

DRL: 4 8 3 6 5

RDL: 4 8 3 6 5

RLD: 4 8 3 6 5

notice that DLR and RLD, LDR and RDL, and LRD and DRL are mirror

symmetric. If we adopt the convention that traversing is done left before right, only then,

the three traversals, that is, LDR, LRD, and DLR, are fundamental. These are called as *inorder*, *postorder*, and *preorder* traversals because there is a natural correspondence

between these traversals producing the infix, postfix, and prefix forms of an arithmetic

expression, respectively.

Inorder Traversal

In an inorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.

2. Visit the node.

3. Traverse the right subtree.

Preorder Traversal

In a preorder traversal, the binary tree is traversed as follows:

1. Visit the node.

2. Traverse the left subtree.

3. Traverse the right subtree.

Postorder Traversal

In a postorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.

2. Traverse the right subtree.

3. Visit the node.



A pointer to the binary tree in Figure 11-5 is stored in the pointer variable root (which

points to the node with info A). Therefore, we start the traversal at A.

1. Traverse the left subtree of A; that is, traverse LA

 {B, D}.

2. Visit A.

3. Traverse the right subtree of A; that is, traverse RA

{C}.

**Operations on binary tree** The basic operations on a binary tree can be as listed as

follows:

1. Creation—Creating an empty binary tree to which the ‘root’ points

2. Traversal—Visiting all the nodes in a binary tree

3. Deletion—Deleting a node from a non-empty binary tree

4. Insertion—Inserting a node into an existing (may be empty) binary tree

5. Merge—Merging two binary trees

6. Copy—Copying a binary tree

7. Compare—Comparing two binary trees

8. Finding a replica or mirror of a binary tree

**Traversal Examples: Formation of Binary Tree from its Traversals**

Sometimes, we need to construct a binary tree if its traversals are known. From a single

traversal, a unique binary tree cannot be constructed. However, if two traversals are known, then the corresponding tree can be drawn uniquely. Let us examine these possibilities

and then chalk out the algorithm

The basic principle for formulation is as follows:

1. 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.

2. Once the root node is identified, all the nodes in all left and right subtrees of the root

node can be identified.

3. Same techniques can be applied repeatedly to form the subtrees.

Construct a binary tree using the following two traversals:

Inorder : *D B H E A I F J C G*

Preorder: *A B D E H C F I J G*

class TreeNode

{

public:

char Data;

TreeNode \*Lchild;

TreeNode \*Rchild;

};

class BinaryTree

{

private:

TreeNode \*Root;

public:

BinaryTree() {Root = Null;} // constructor

// int BTree\_Equal(BinaryTree, BinaryTree);

TreeNode \*GetNode();

void InsertNode(TreeNode\*);

void DeleteNode(TreeNode\*);

void Postorder(TreeNode\*);

void Inorder(TreeNode\*);

void Preorder(TreeNode\*);

TreeNode \*TreeCopy();

void Mirror();

int TreeHeight(TreeNode\*);

int CountLeaf(TreeNode\*);

int CountNode(TreeNode\*);

void BFS\_Tree();

void DFS\_Tree();

TreeNode \*Create\_Btree\_InandPre\_Traversal(char

preorder[max], char inorder[max]);

void Po storder\_Non\_Recursive(void);

void Inorder\_Non\_Recursive();

void Preorder\_Non\_Recursive();

int BTree\_Equal( BinaryTree, BinaryTree);

TreeNode \*TreeCopy(TreeNode\*);

void Mirror(TreeNode\*);

};