Non-Linear Data Structure

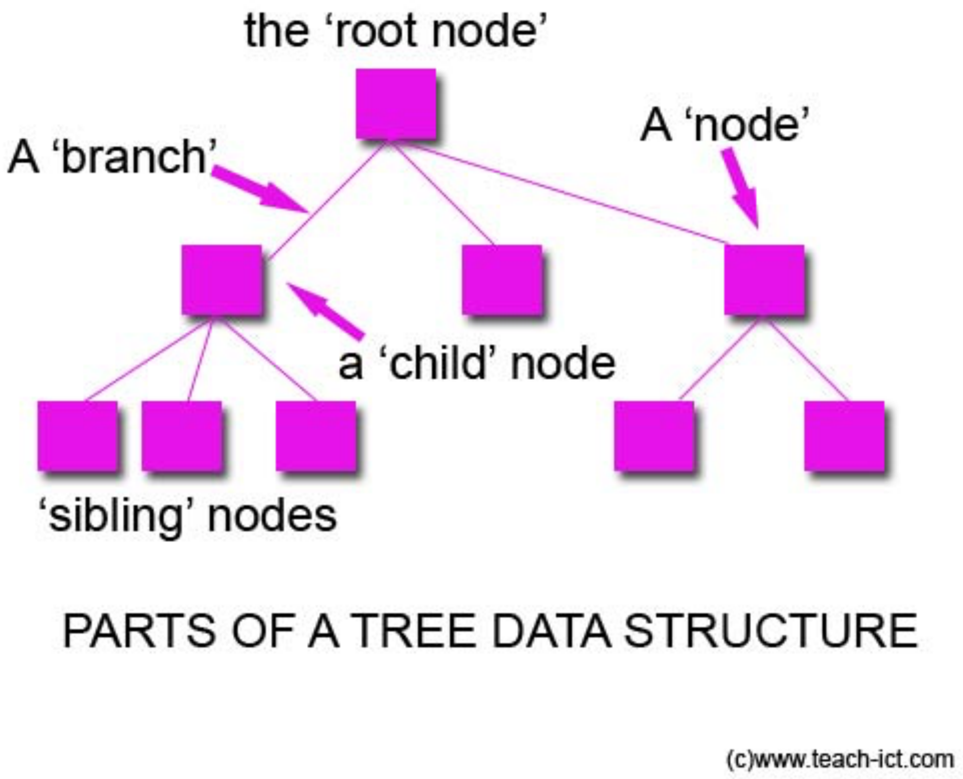
- Trees

Acknowlege: www.tutorialspoint.com

Prof. Yi Fang (yfang@nyu.edu)

Trees

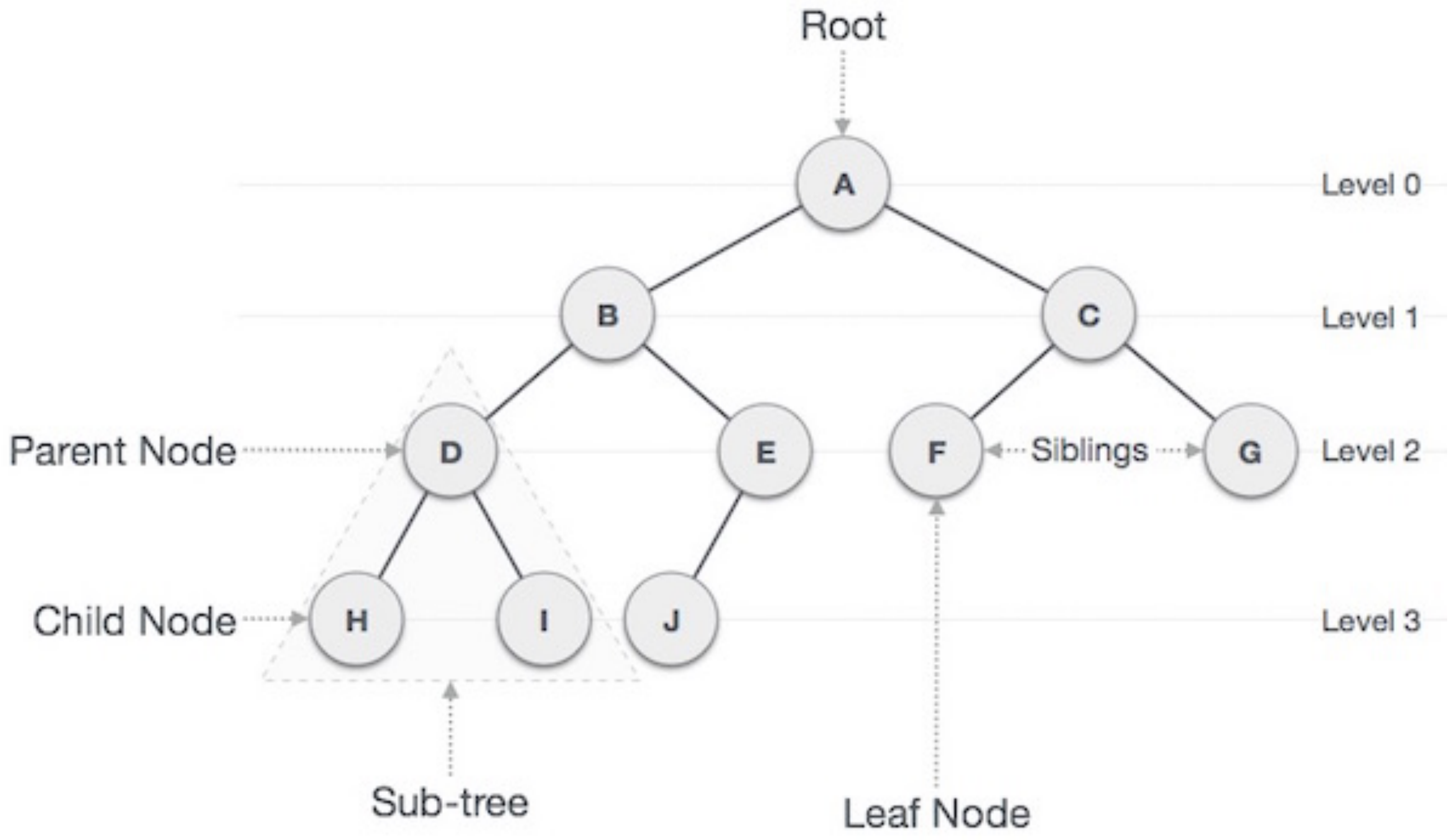
Acknowlege: www.tutorialspoint.com



A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.

Binary Tree

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

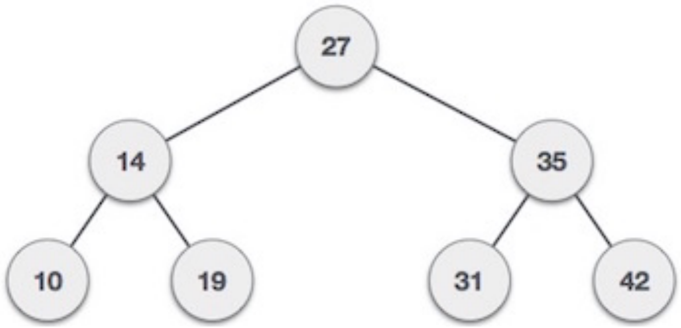

Root
A
B
C
D
E
F
G
H
I
J
Level 0
Level 1
Level 2
Level 3
Parent Node
Child Node
Siblings
Sub-tree
Leaf Node

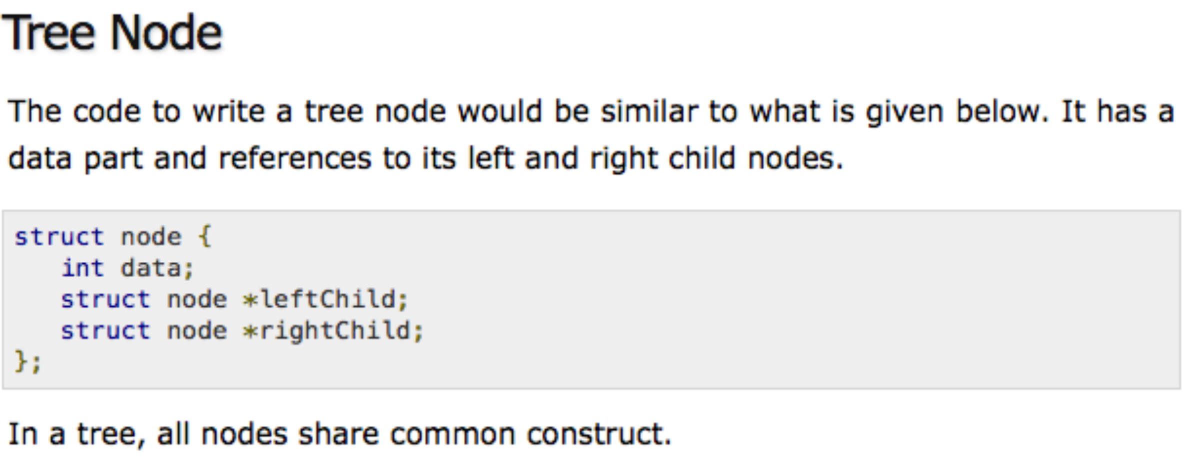
Definitions:

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.





Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

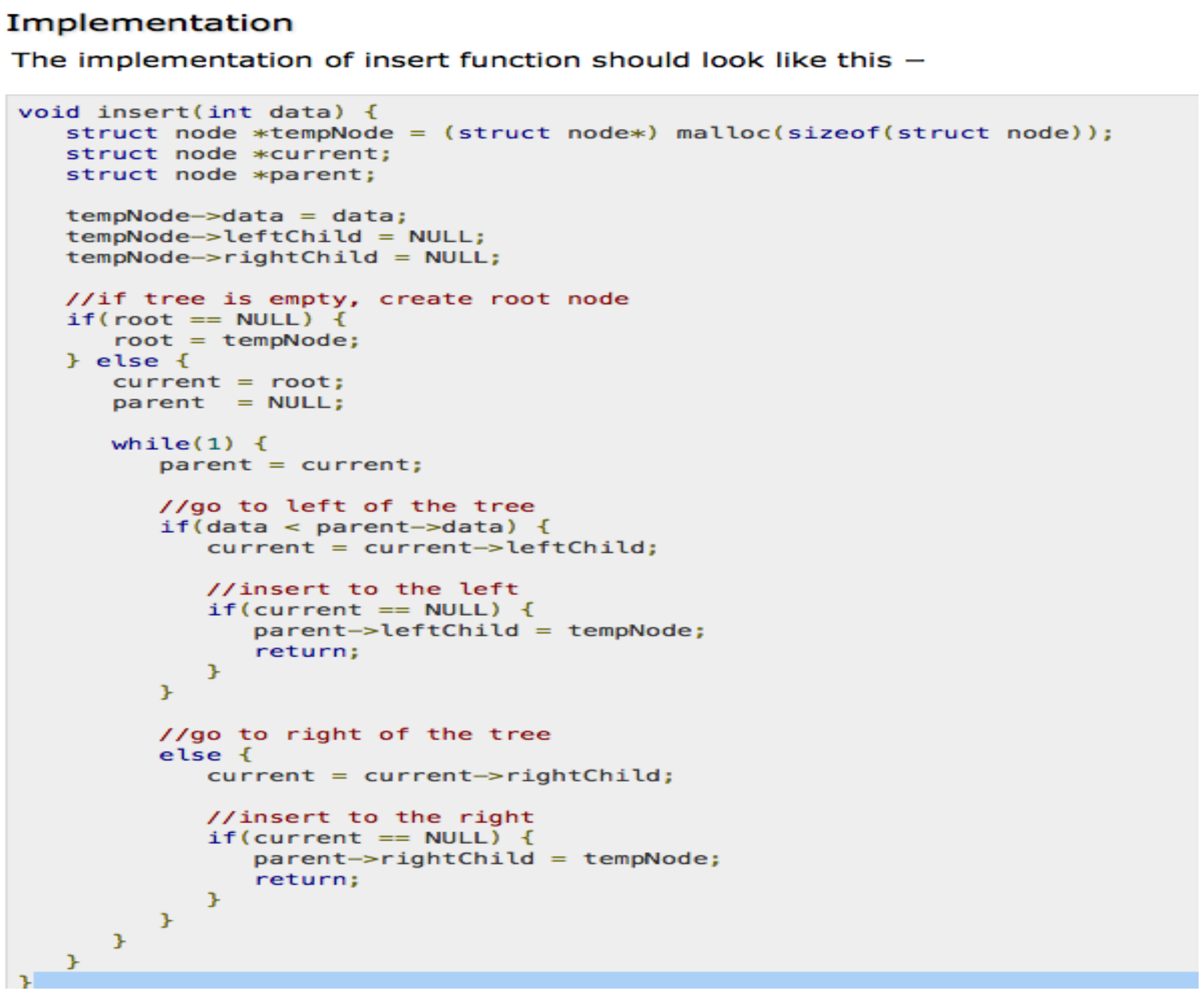
- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

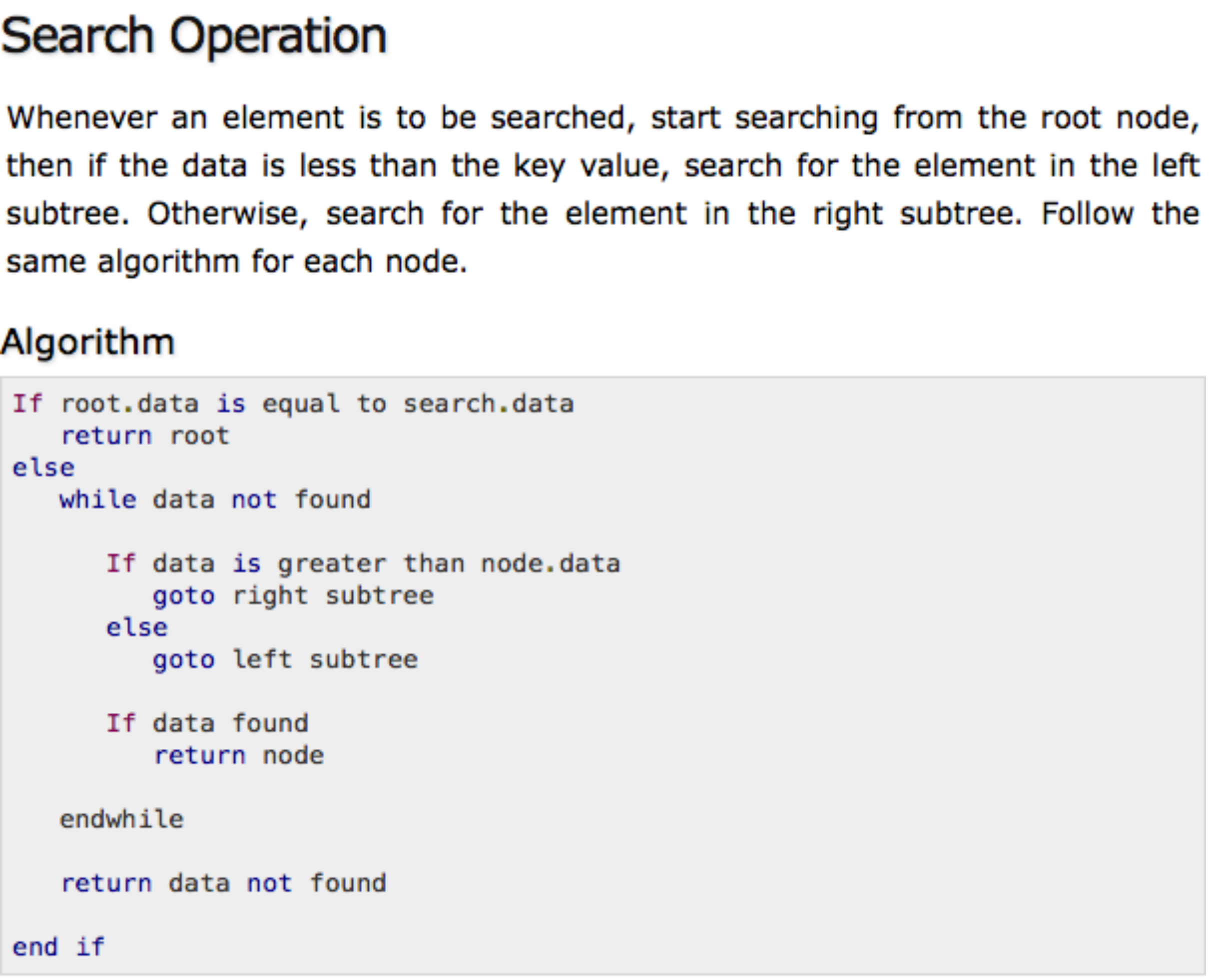
```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile

   insert data

end If
```



Implementation

The implementation of insert function should look like this –

```c
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```



Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile

   return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

Tree Traversal

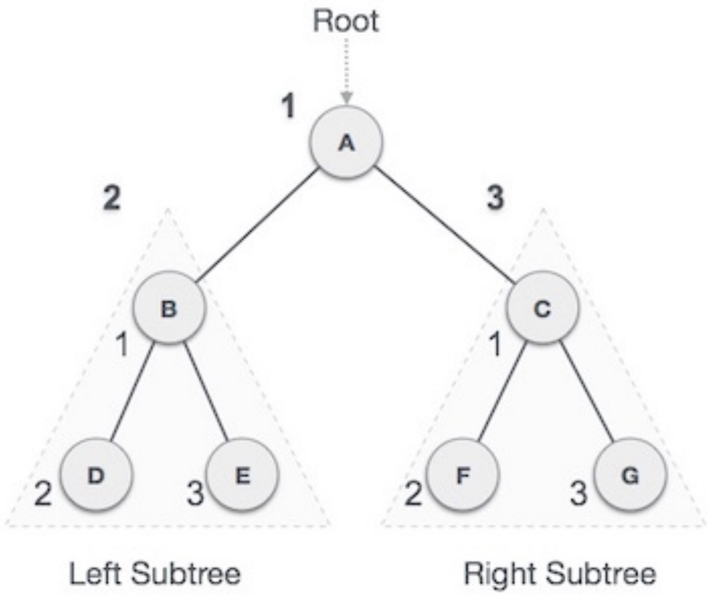
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

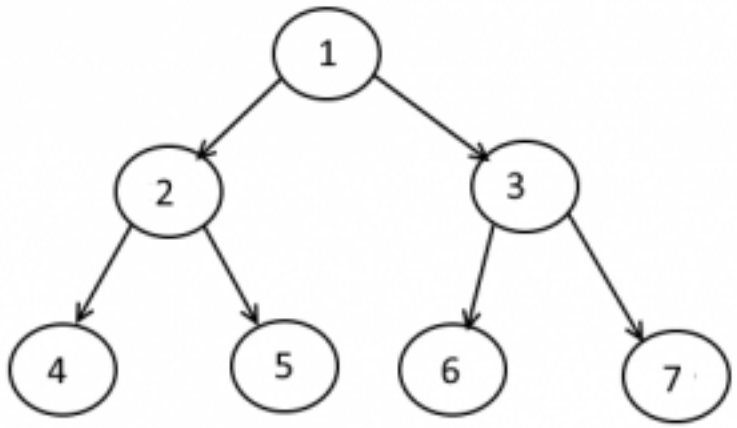


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

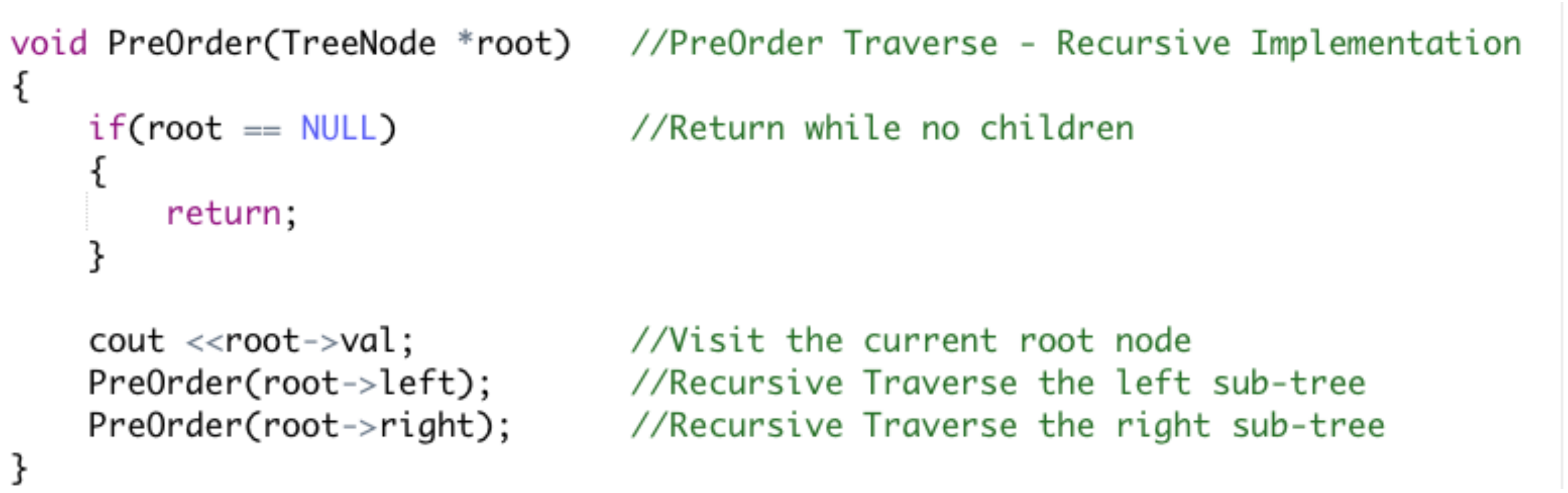
```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```



Preorder Traversal: 1 2 4 5 3 6 7

Cite: http://algorithms.tutorialhorizon.com/binary-tree-preorder-traversal-non-recursive-approach/

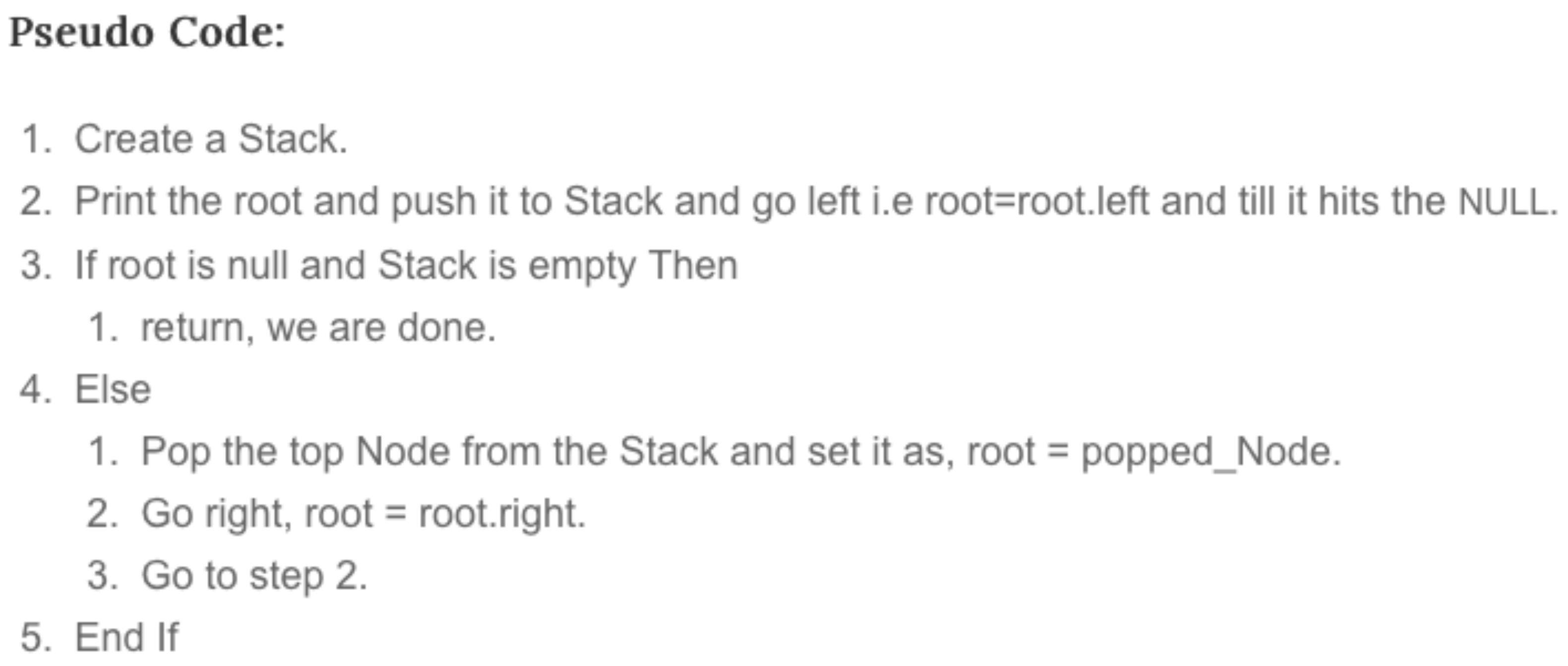
Recursive Implementation



```
void PreOrder(TreeNode *root)   //PreOrder Traverse - Recursive Implementation
{
    if(root == NULL)            //Return while no children
    {
        return;
    }

    cout <<root->val;           //Visit the current root node
    PreOrder(root->left);       //Recursive Traverse the left sub-tree
    PreOrder(root->right);      //Recursive Traverse the right sub-tree
}
```

Non Recursive Approach

Since we are not using recursion, we will use the **Stack** to store the traversal, we need to remember that preorder traversal is, first traverse the root node then left node followed by the right node.



Pseudo Code:

```
1. Create a Stack.
2. Print the root and push it to Stack and go left i.e root=root.left and till it hits the NULL.
3. If root is null and Stack is empty Then
    1. return, we are done.
4. Else
    1. Pop the top Node from the Stack and set it as, root = popped_Node.
    2. Go right, root = root.right.
    3. Go to step 2.
5. End If
```

Cite: http://algorithms.tutorialhorizon.com/binary-tree-preorder-traversal-non-recursive-approach/

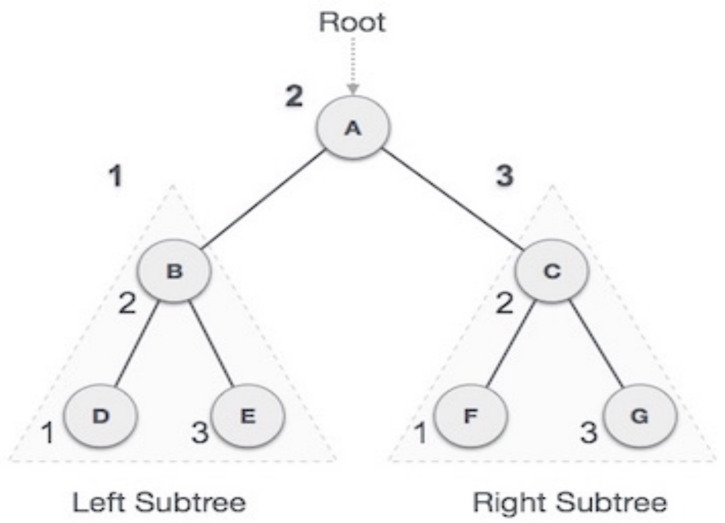
Implementation

```
public void preorderIteration(Node root) {
        Stack<Node> s = new Stack<Node>();
        while (true) {
                // First print the root node and then add left node
                while (root != null) {
                        System.out.print(root.data + " ");
                        s.push(root);
                        root = root.left;
                }
                // check if Stack is emtpy, if yes, exit from everywhere
                if (s.isEmpty()) {
                        return;
                }
                // pop the element from the stack and go right to the tree
                root = s.pop();
                root = root.right;
        }
}
```

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

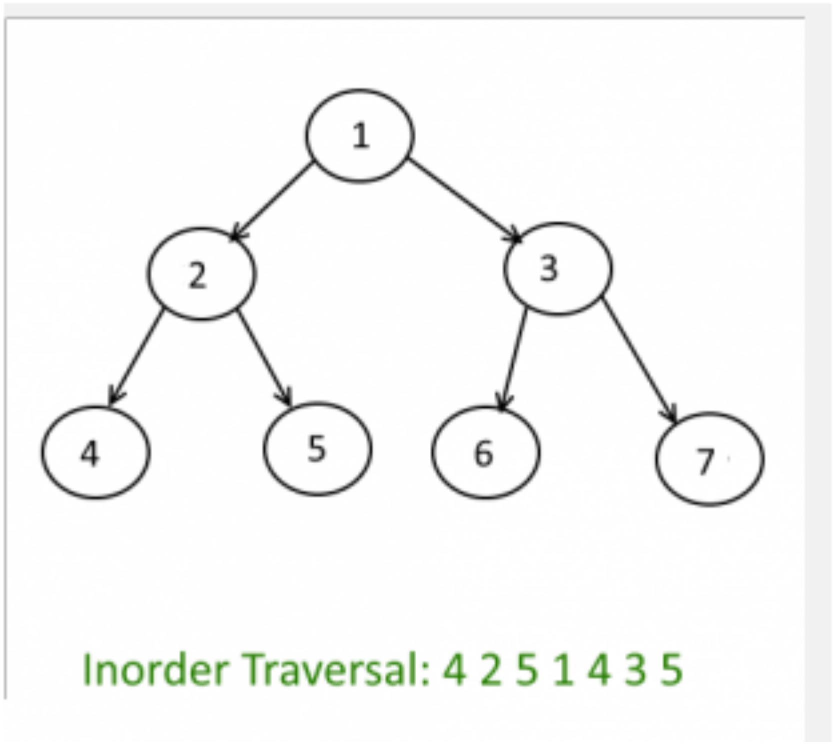


We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

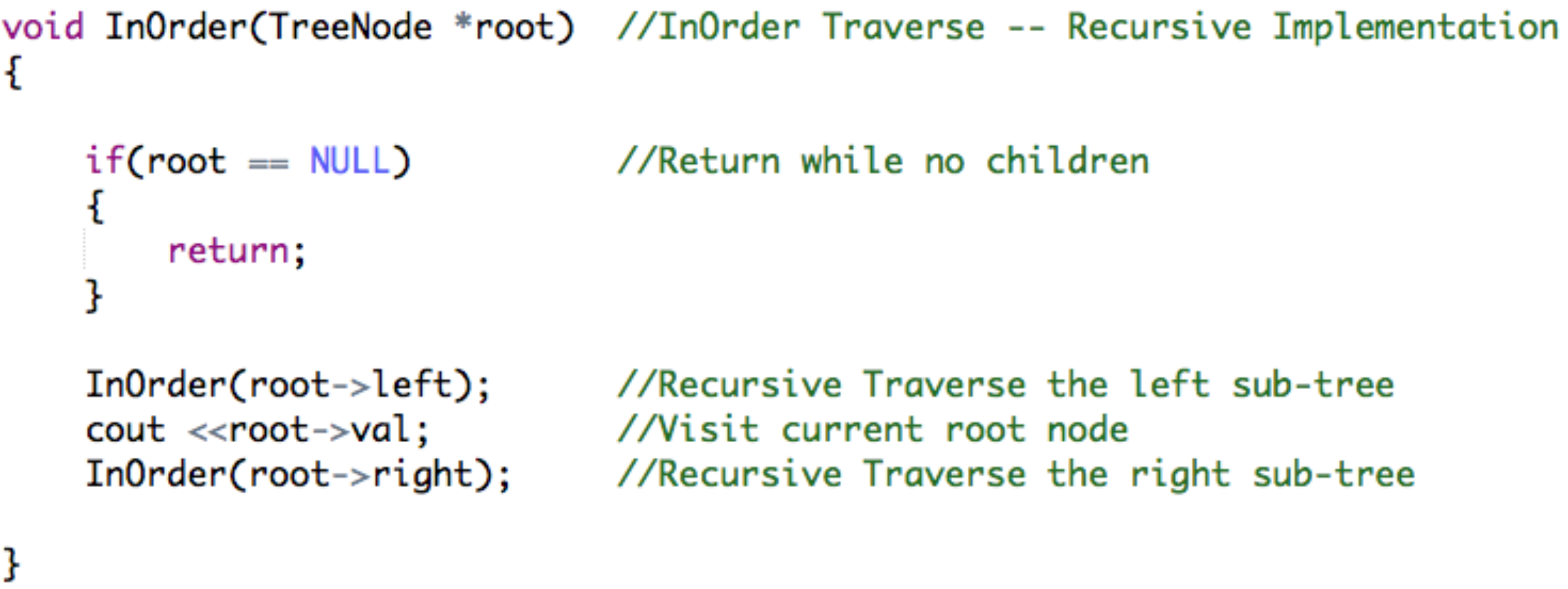
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```



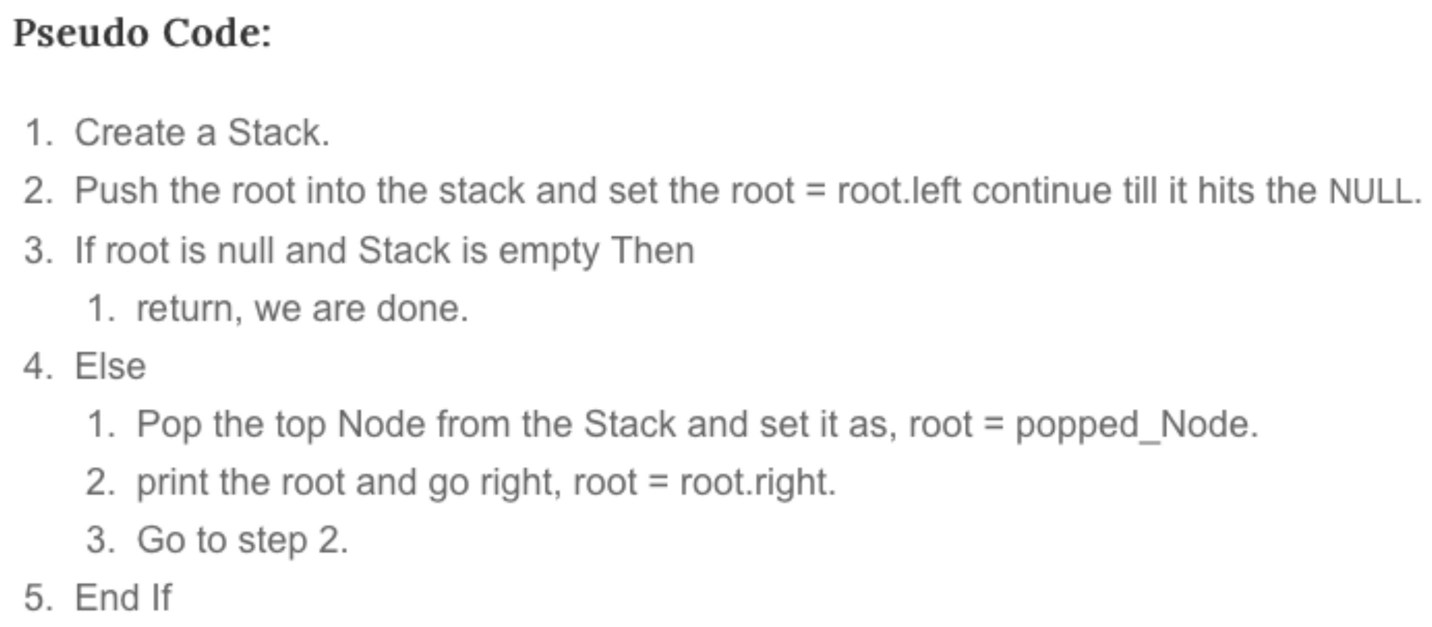
Recursive Approach



```
void InOrder(TreeNode *root)  //InOrder Traverse -- Recursive Implementation
{

    if(root == NULL)          //Return while no children
    {
        return;
    }

    InOrder(root->left);      //Recursive Traverse the left sub-tree
    cout <<root->val;         //Visit current root node
    InOrder(root->right);     //Recursive Traverse the right sub-tree

}
```

Non-Recursive Approach

Since we are not using recursion, we will use the **Stack** to store the traversal, we need to remember that inorder traversal is, first traverse the left node then root followed by the right node.



Pseudo Code:

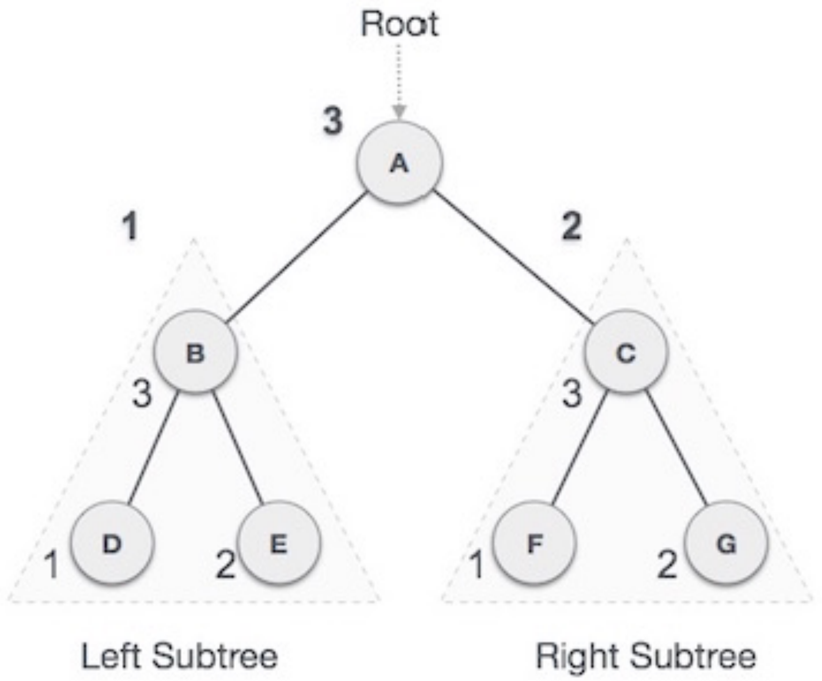
```
1. Create a Stack.
2. Push the root into the stack and set the root = root.left continue till it hits the NULL.
3. If root is null and Stack is empty Then
    1. return, we are done.
4. Else
    1. Pop the top Node from the Stack and set it as, root = popped_Node.
    2. print the root and go right, root = root.right.
    3. Go to step 2.
5. End If
```

Implementation

```
public void inorderIteration(Node root) {
        Stack<Node> s = new Stack<Node>();
        while (true) {
                // Go to the left extreme insert all the elements to stack
                while (root != null) {
                        s.push(root);
                        root = root.left;
                }
                // check if Stack is empty, if yes, exit from everywhere
                if (s.isEmpty()) {
                        return;
                }
                // pop the element from the stack , print it and add the nodes at
                // the right to the Stack
                root =s.pop();
                System.out.print(root.data + " ");
                root = root.right;
        }
}
```

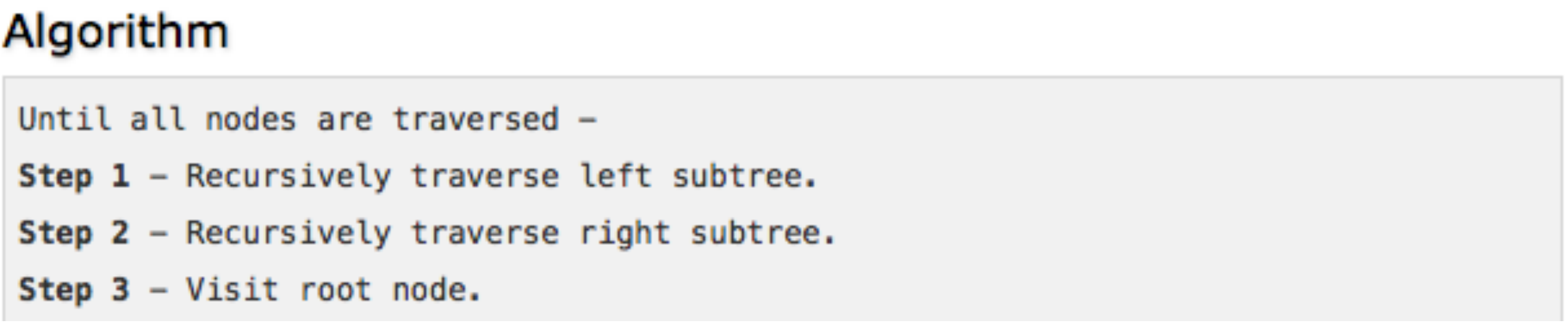
Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

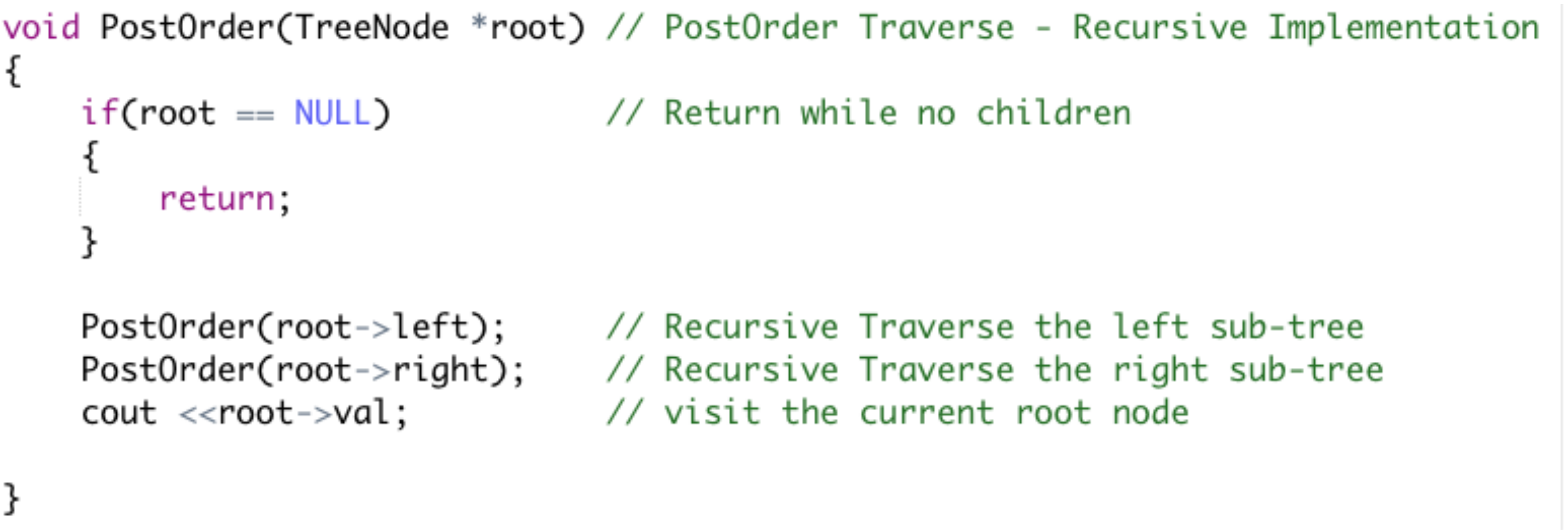
$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$



Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

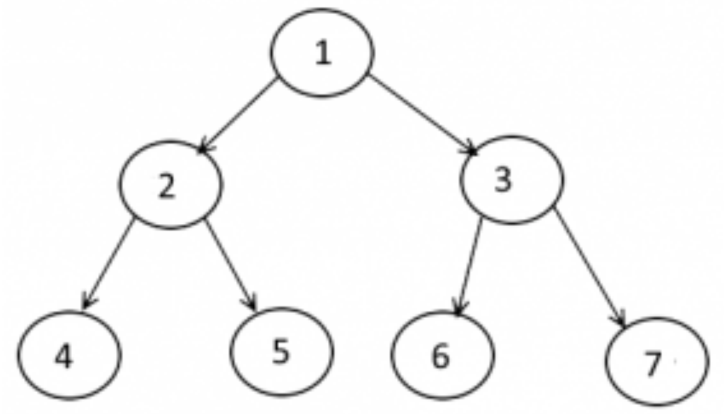
Recursive Implementation



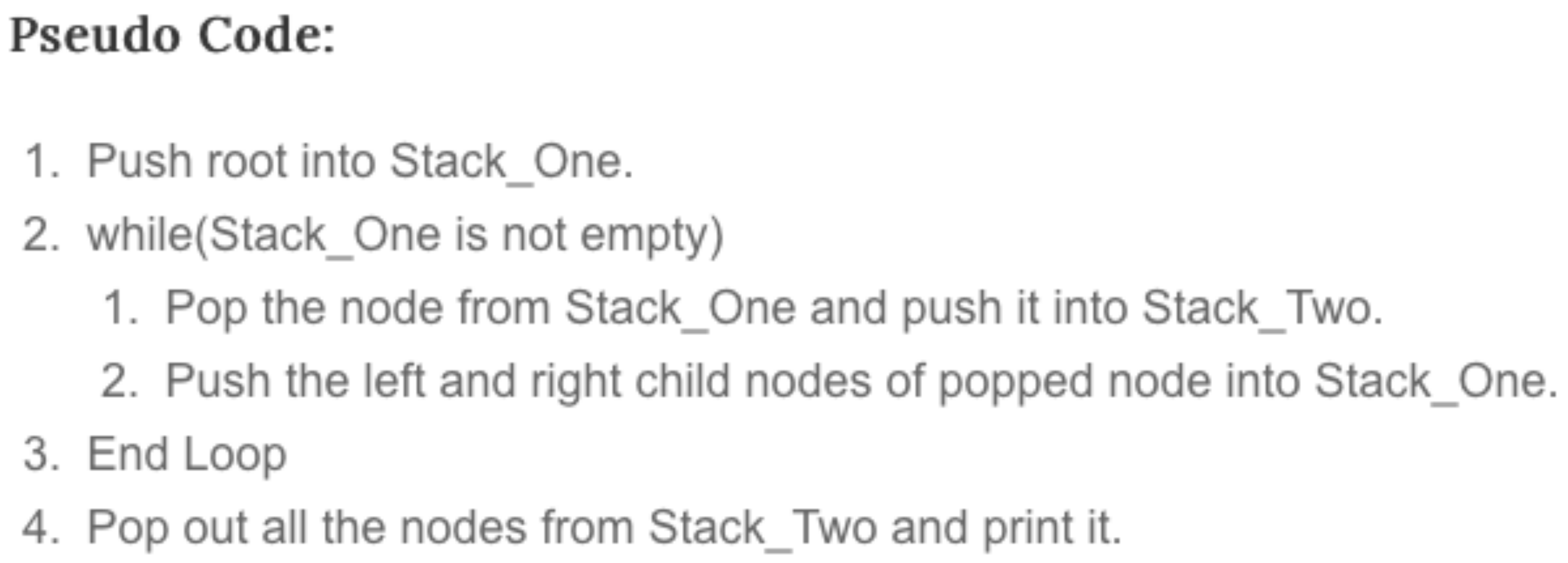
```
void PostOrder(TreeNode *root) // PostOrder Traverse - Recursive Implementation
{
    if(root == NULL)           // Return while no children
    {
        return;
    }

    PostOrder(root->left);     // Recursive Traverse the left sub-tree
    PostOrder(root->right);    // Recursive Traverse the right sub-tree
    cout <<root->val;          // visit the current root node

}
```

Non-Recursive Approach

- If you just observe here, postorder traversal is just reverse of preorder traversal (1 3 7 6 2 5 4 if we traverse the right node first and then left node.)
- So idea is follow the same technique as preorder traversal and instead of printing it push it to the another Stack so that they will come out in reverse order (LIFO).
- At the end just pop all the items from the second Stack and print it.



Postorder Traversal: 4 5 2 6 7 3 1

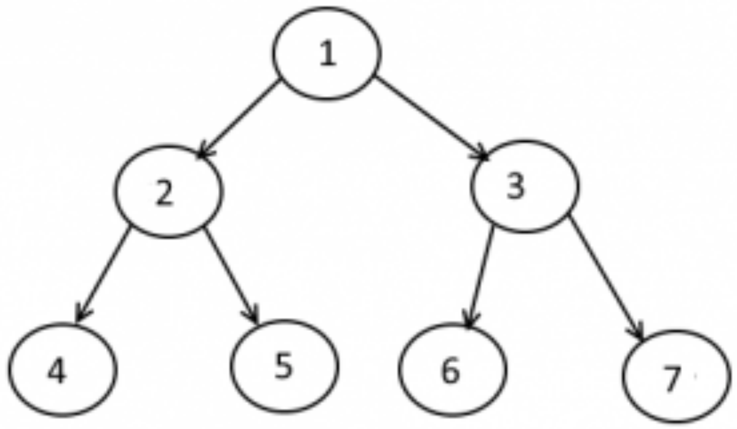


Pseudo Code:

```
1. Push root into Stack_One.
2. while(Stack_One is not empty)
    1. Pop the node from Stack_One and push it into Stack_Two.
    2. Push the left and right child nodes of popped node into Stack_One.
3. End Loop
4. Pop out all the nodes from Stack_Two and print it.
```

Implementation

```
public void preorderIteration(Node root) {
        Stack<Node> s1 = new Stack<Node>();
        Stack<Node> s2 = new Stack<Node>();
        // push the root node into first stack.
        s1.push(root);
        while (s1.isEmpty() == false) {
                // take out the root and insert into second stack.
                Node temp = s1.pop();
                s2.push(temp);
                // now we have the root, push the left and right child of root into
                // the first stack.
                if(temp.left!=null){
                        s1.push(temp.left);
                }
                if(temp.right!=null){
                        s1.push(temp.right);
                }
        }
        //once the all node are traversed, take out the nodes from second stack and print it.
        System.out.println("Preorder Traversal: ");
        while(s2.isEmpty()==false){
                System.out.print(s2.pop());
        }
}
```



Preorder Traversal: 1 2 4 5 3 6 7