```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

}
```

TABLE OF CONTENTS

Depth First Traversals (Recursive)

1. PreOrder

In [ ]:

```
void preOrder(TreeNode* root){

    if(root){
        cout<<root->val<<" ";
        preOrder(root->left);
        preOrder(root->right);
    }

}
```

2. Inorder

In [ ]:

```
void inOrder(TreeNode* root){

    if(root){
        preOrder(root->left);
        cout<<root->val<<" ";
        preOrder(root->right);
    }

}
```

3. PreOrder

In [ ]:

```
void preOrder(TreeNode* root){

    if(root){
        preOrder(root->left);
        preOrder(root->right);
        cout<<root->val<<" ";
    }

}
```

Depth First Traversals (Iterative)

1. PreOrder

In [ ]:

```
vector<int> preOrder(TreeNode* root){

    vector<int> traversal;

    stack<TreeNode*> stk;

    while(root || !stk.empty()){

        while(root){
            traversal.push_back(root->val);
            stk.push(root);
            root = root->left;
        }

        root = stk.top()->right;
        stk.pop();

    }

    return traversal;
}
```

2. Inorder

In [ ]:

```
vector<int> inOrder(TreeNode* root){

    vector<int> traversal;

    stack<TreeNode*> stk;

    while(root || !stk.empty()){

        while(root){
            stk.push(root);
            root = root->left;
        }

        root = stk.top();
        stk.pop();
        traversal.push_back(root->val);
        root = root->right;
    }

    return traversal;
}
```

Another way

In [ ]:

```
vector<int> inOrder(TreeNode* root){

    vector<int> traversal;

    stack<TreeNode*> stk;

    while(root || !stk.empty()){

        if(root){
            stk.push(root);
            root = root->left;
        }
        else
        {
            root = stk.top();
            stk.pop();
            traversal.push_back(root->val);
            root = root->right;
        }
    }

    return traversal;
}
```

3. PostOrder

In [ ]:

```
vector<int> postOrder(TreeNode* root){

    vector<int> traversal;

    stack<TreeNode*> stk;

    while(root || !stk.empty()){

        if(root){
            traversal.push_back(root->val);    //reverse the process
            stk.push(root);
            root = root->right;
        }
        else
        {
            root = stk.top();
            stk.pop();
            root = root->left;
        }

    }

    reverse(traversal.begin(),traversal.end());
    return traversal;
}
```

Another way

In [ ]:

```
vector<int> postOrder(TreeNode *root){

    vector<int> traversal;

    stack<TreeNode*> stk;

    do{

        while(root){

            stk.push(root);
            root = root->left;

        }

        while(root==NULL && !stk.empty()){

            root = stk.top();

            if(root->right == NULL || root->right==prev){

                stk.pop();
                trav.push_back(root->val);
                prev = root;
                root = NULL;
            }
            else
                root = root->right;
        }

    }while(!stk.empty());

    return traversal;

}
```

Breadth First Traversal / Level Order Traversal

In [ ]:

```
vector<vector<int>> breadthFirst(TreeNode* root){

    vector<vector<int>> traversal;

    queue<TreeNode*> q;

    if(root){
        q.push(root->val);
        q.push(NULL);
        traversal.push_back({});
    }

    while(!q.empty()){

        TreeNode *node = q.front();
        q.pop();

        if(q==NULL){

            if(!q.empty()){

                q.push(NULL);
                traversal.push_back({});
            }

        }
        else {

            traversal.back().push_back(node->val);

            if(node->left)
                q.push(node->left);

            if(node->right)
                qpush(node->right);
        }
    }

    return traversal;
}
```

Insert In A Binary Tree

In [ ]:

```
TreeNode* insert(TreeNode* root, int val){

    queue<TreeNode*> q;

    if(root)
        q.push(root);
    else
        root = new TreeNode(val);

    while(!q.empty()){

        TreeNode* root = q.front();
        q.pop();

        if(root->left)
            q.push(root->left);
        else
        {

            root->left = new TreeNode(val);
            break;
        }

        if(root->right)
            q.push(root->right);
        else
        {

            root->right = new TreeNode(val);
            break;

        }

    }

    return root;

}
```

Delete Whole Binary Tree

In [ ]:

```
void deleteBTree(TreeNode* root){

    if(root){

        deleteBTree(root->left);
        deleteBTree(root->left);
        free(root);

    }

}
```

Height / Depth Of Tree

In [ ]:

```
int height(TreeNode* root){

    if(root==NULL)
        return 0;

    return 1 + max(height(root->left),height(root->right));

}
```

Deeepest Node Of Binary Tree

In [ ]:

```
TreeNode* findDeepestNode(TreeNode *root){

    queue<TreeNode*> q;

    if(root)
        q.push(root);

    while(!q.empty()){

        root = q.front();
        q.pop();

        if(root->left)
            q.push(root->left);

        if(root->right)
            q.push(root->right);


    }

    return root;

}
```

Diameter of a Binary Tree

In [ ]:

```
int diameter = 1;

int height(TreeNode* root){

    if(root==NULL) return 0;

    int left = height(root->left);
    int right = height(root->right);

    diameter = max(left+right+1,diameter);

    return 1 + max(left,right);
}

int findDiameter(TreeNode* root){
    diameter = 1;
    height(root);
    return diameter-1;
}
```

All Paths Of Binary Tree

In [ ]:

```
void pathsOfBinaryTree(TreeNode *root,vector<int> currPath, vector<vector<int>> &paths){

    if(root==NULL)
        return;

    if(!root->left && !root->right){
        currPath.push_back(root->val);
        paths.push_back(currPath);
    }
    else{

        currPath.push_back(root->val);
        pathsOfBinaryTree(root->left,currPath,paths);
        pathsOfBinaryTree(root->right,currPath,paths);
    }

}
```

Path Sum

In [ ]:

```
bool pathSum(TreeNode *root, int sum){

    if(root==NULL)
        return false;

    if(!root->left && !root->right && sum==root->val) return true;

    return pathSum(root->left,sum-root->val) || pathSum(root->right,sum-root->val);
}
```

LCA Of Binary Tree

In [ ]:

```
TreeNode* lca(TreeNode* root, TreeNode* p, TreeNode *q){

    if(root==NULL)
        return NULL;
    if(root == p || root == q)
        return root;

    TreeNode* left = lca(root->left,p,q);
    TreeNode* right = lca(root->right,p,q);
    if(left && right)
        return root;

    return left?left:right;

}
```

Construct a binary tree from Preorder and Inorder

In [ ]:

```
TreeNode* constructBinaryTree(vector<int> inorder, vector<int> preOrder){

    int n = preOrder.size();

    if(n==0) return NULL;

    stack<TreeNode> stk;
    TreeNode* root = new TreeNode(preOrder[0]);
    stk.push(root);

    int inOrderIndex = 0;

    for(int preOrderIndex = 1; preOrderIndex < n; preOrderIndex++){

        TreeNode *curr = stk.top();

        if(curr->val != inorder[inOrderindex]){

            curr->left = new TreeNode(preOrder[preOrderIndex]);
            stk.push(curr);

        }
        else
        {
                while(!stk.empty() && stk.top()->val == inorder[inOrderIndex]){
                    curr = stk.top();
                    stk.pop();
                    inOrderIndex++;
                }

                if(inOrderindex < n){
                    curr->right = new TreeNode(preOrder[preOrderIndex]);
                    stk.push(curr->right);
                }
        }

    }

    return root;

}
```

Construct a binary tree from Postorder and Inorder

In [ ]:

```
TreeNode* constructBinaryTree(vector<int> inOrder, vector<int> postOrder){

    int n = postOrder.size();

    if(n==0) return NULL;

    stack<TreeNode*> stk;

    TreeNode* root = new TreeNode(preOrder[n-1]);
    int inOrderIndex = n-1;

    stk.push(root);

    for(int postOrderIndex = n-2; postOrderIndex >= 0; postOrderIndex--){

        TreeNode *curr = stk.top();

        if(curr->val != inOrder[inOrderIndex]){

            curr->right = new TreeNode(postOrder[postOrderIndex]);
            stk.push(curr->right);
        }
        else{

            while(!stk.empty() && stk.top()->val == inOrder[inOrderIndex]){
                curr = stk.top();
                stk.pop();
                inOrderIndex--;
            }

            if(inOrderIndex >= 0){
                curr->left = new TreeNode(postOrder[postOrderIndex]);
                stk.push(curr->left);
            }
        }

    }

    return root;
}
```

Inorder Threaded Binary Trees

```
struct TreeNode{
    int val;
    bool rightChild;
    bool leftChild;
    TreeNode *left, *right;
};
```

- If **rightchild = false** then right points to inorder successor else if **rightChild = true** then right points to right child of the node.
- If **lefttchild = false** then left points to inorder predecessor else if **leftChild = true** then left points to left child of the node.

```
TreeNode(){
  rightChild = leftChild = false;
  left = right = NULL;
}
```

1. Finding Inorder Successor In Inorder Threaded Binary Tree

In [ ]:

```
TreeNode* findSuccessor(TreeNode *node){

    if(node->rightChild == false)
        return node->right;

    #If rightChild is true then successor would be the the left most node of right subtree
    curr = node->right;

    while(curr->leftChild)
        curr = curr->left;

    return curr;

}
```

Inorder Traversal In Inorder Threaded Binary Tree

In [ ]:

```
void inOrder(TreeNode* dummy){

    #We can start with dummy node and the dummy node always have a rightChild which is itself and and leftChild is
    #so dummy->left = root of the tree. Initially right & left are pointing to itself

    TreeNode *root = findSuccessor(dummy);

    #Returns dummy if the tree is empty
    if(root != dummy){

        root = findSuccessor(leftMost);
        cout<<root->data<<" ";
        #Also the rightMost node's successor is dummy.

    }

}
```

2. Finding PreOrder Successor In Inorder Threaded Binary Tree

In [ ]:

```
TreeNode* findPreOrderSuccessor(TreeNode *node){

    #If the left child of node exists then that is the successor
    if(node->leftChild == true)
        return node->left;

    #Else find in the right subtree
    while(node->rightChild==false)
        node = node->right;

    return node->right;

}
```

PreOrder Traversal In Inorder Threaded Binary Tree

In [ ]:

```
void preOrder(TreeNode* dummy){

    #We can start with dummy node and the dummy node always have a rightChild which is itself and and leftChild is
    #so dummy->left = root of the tree. Initially right & left are pointing to itself

    TreeNode *root = findPreOrderSuccessor(dummy);

    #Returns dummy if the tree is empty
    if(root != dummy){

        root = findPreOrderSuccessor(leftMost);
        cout<<root->data<<" ";
        #Also the rightMost node's successor is dummy.

    }

}
```

3. Insert Right In Inorder Threaded Binary Tree

Insert Q to the right of P

In [ ]:

```
void insertRight(TreeNode *P, TreeNode *Q){

    Q->right = P->right;
    Q->rightChild = P->rightChild;

    Q->left = P;
    Q->leftChild = false;

    P->right = Q;
    P->rightChild = true;

    if(Q->rightChild == true){

        #Traverse to the leftMost node in the subtree and change iits predecessor

        TreeNode *curr = Q->right;

        while(curr->leftChild)
            curr = curr->left;

        curr->left = Q;

    }

}
```

Depth first Traversal (Threaded)

1. PreOrder Traversal

In [ ]:

```
vector<int> preOrder(TreeNode *root){

    vector<int> res;
    TreeNode *curr = root;

    while(curr){

        if(curr->left==NULL){
            res.push_back(curr->val);
            curr = curr->right;
        }
        else{

            TreeNode* temp = curr->left;

            while(temp->right && temp->right!=curr)
                temp = temp->right;

            if(temp->right==NULL){
                res.push_back(curr->val);
                temp->right = curr;
                curr = curr->left;
            }
            else{
                temp->right = NULL;
                curr = curr->right;
            }
        }

    }

    return res;
}
```

2. Inorder Traversal

In [ ]:

```
vector<int> inorder(TreeNode *root){

    vector<int> res;

    TreeNode *curr = root;

    while(curr){

        if(curr->left==NULL){

            res.push_back(curr->val);
            curr = curr->right;

        }
        else {

            TreeNode *temp = curr->left;
            while(temp->right && temp->right != curr)
                temp = temp->right;

            if(temp->right==NULL){
                temp->right = curr;
                curr = curr->left;
            }
            else {
                temp->right = NULL;
                res.push_back(curr->val);
                curr = curr->right;
            }
        }
    }

    return res;
}
```

3. PostOrder Traversal

In [ ]:

```
vector<int> postOrder(TreeNode* root){

    vector<int> res;

    TreeNode* curr = root;

    while(curr){

        if(curr->right == NULL){
            res.push_back(curr->val);
            curr = curr->left;
        }
        else
        {

            TreeNode *temp = curr->right;

            while(temp->left && temp->left != curr)
                temp = temp->left;

            if(temp->left==NULL){
                res.push_back(curr->val);
                temp->left = curr;
                curr = curr->right;
            }
            else{
                temp->left = NULL;
                curr = curr->left;
            }

        }

    }

    reverse(res.begin(),res.end());
    return res;
}
```

In [ ]: