```
#include <iostream>
using namespace std;

class Node{
    int data;
    Node *left, *right;
public:
    Node(int n){
        data=n;
        left=NULL;
        right=NULL;
    }
    friend class Tree;
    friend class Stack;
    friend void Copy(Node*,Node*);
};

class Stack{
    Node* st[25];
    int top;
public:
    Stack(){
        top=-1;
    }
```

```
    void Push(Node* x){
        st[++top]=x;
    }
    void Pop(){
        top--;
    }
    Node* getTop(){
        return st[top];
    }
    bool isEmpty(){
        if(top==-1)
            return true;
        else
            return false;
    }
    friend class Node;
};


class Tree{
    Node *root;
    Node* MakeTree();
public:
    Tree(){
```

```
        root=NULL;
    }
    void Create();
    void PostOrder(Node *temp);
    void PreOrder(Node *temp);
    void InOrder(Node *temp);
    void NonRecursiveInOrder(Node *temp);
    void NonRecursivePostOrder(Node *temp);
    void NonRecursivePreOrder(Node *temp);
    void Display();

    void HCall();
    int Height(Node *temp);
    void LICall();
    int LeafCount(Node *temp);
    int InternalCount(Node *temp);
    void MCall();
    void Mirror(Node *temp);
    void operator=(Tree temp);
    void ECall();
    void Erase(Node *temp);

    friend class Stack;
};
```

```
//function to call creation function
void Tree :: Create(){
    root=MakeTree();
}
//allocates memory for each node according to
user input, left-left-left first, then right-right-right
Node* Tree :: MakeTree(){
    int n;
    char ch;
    cout<<"Enter data: ";
    cin>>n;
    Node* temp = new Node(n);

    cout<<"Do you want to insert left child for
"<<temp->data<<" ? Enter (y/n)";
    cin>>ch;
    if(ch=='y'|| ch=='Y')
        temp->left=MakeTree();

    cout<<"Do you want to insert right child for
"<<temp->data<<" ? Enter (y/n)";
    cin>>ch;
    if(ch=='y'|| ch=='Y')
```

```
        temp->right=MakeTree();

    return temp;
}

//traverses tree in order and prints it, non-
recursively
void Tree :: NonRecursiveInOrder(Node *temp){
    Stack S;
    Node *current=temp;
    while(current!=NULL || S.isEmpty()==false){
        while(current!=NULL){
            S.Push(current);
            current=current->left; //go left as much as
possible first
        }
        current=S.getTop();
        cout<<current->data<<"\t";
        S.Pop();

        current=current->right;
        //if no more left children available, go back
one node and go to its right child
    }
```

```
}

//traverses tree pre order and prints it, non-
recursively
void Tree :: NonRecursivePreOrder(Node *temp){
    Node *current=temp;
    Stack S;

    if(temp==NULL)
        return;

    S.Push(current);
    while(S.isEmpty()==false){

        current=S.getTop();
        cout<<current->data<<"\t";
        S.Pop();

        if(current->right!=NULL)
            S.Push(current->right);
        if(current->left!=NULL)
            S.Push(current->left);

    }
```

```
}
//traverses tree in post order and prints it, non-recursively
void Tree :: NonRecursivePostOrder(Node *temp)
{
    Stack S;

    if(temp==NULL)
        return;

    do{
        while(temp){
            if(temp->right)
                S.Push(temp->right);
            S.Push(temp);
            temp=temp->left;
        }
        temp=S.getTop();
        S.Pop();

        if(temp->right && S.getTop()==temp->right){
            S.Pop();
            S.Push(temp);
            temp=temp->right;
```

```
        }
        else{
            cout<<temp->data<<"\t";
            temp=NULL;
        }

    }while(!S.isEmpty());
}
//traverses tree in post-order and prints it,
recursively
void Tree :: PostOrder(Node *temp){
    if(temp==NULL)
        return;
    PostOrder(temp->left);
    PostOrder(temp->right);
    cout<<temp->data<<"\t";
}
//traverses tree in order and prints it, recursively
void Tree :: InOrder(Node *temp){
    if(temp==NULL)
        return;
    InOrder(temp->left);
    cout<<temp->data<<"\t";
    InOrder(temp->right);
```

```
}
//traverses tree in pre-order and prints it,
recursively
void Tree :: PreOrder(Node *temp){
    if(temp==NULL)
        return;
    cout<<temp->data<<"\t";
    PreOrder(temp->left);
    PreOrder(temp->right);
}

//driver function to call all traversals, recursive
or non recursive
void Tree :: Display(){
    int x;
    do{
        cout<<"\nEnter: \n1. Recursive \n2. Non-
Recursive \n3. Exit \n";
        cin>>x;
        switch(x){
            case 1: cout<<"InOrder: ";
                    InOrder(root);
                    cout<<endl;
                    cout<<"PostOrder:: ";
```

```
                PostOrder(root);
                cout<<endl;
                cout<<"PreOrder: ";
                PreOrder(root);
                cout<<endl;
               break;
        case 2: cout<<"InOrder: ";
               NonRecursiveInOrder(root);
               cout<<endl;
               cout<<"PostOrder: ";
               NonRecursivePostOrder(root);
               cout<<endl;
               cout<<"PreOrder: ";
               NonRecursivePreOrder(root);
               cout<<endl;
               break;
        case 3: break;
        default: cout<<"Invalid input!\n";
      }
   }while(x!=3);
}

//function to call function that finds height of the
tree
```

```
void Tree :: HCall(){
    cout<<"Height is: "<<Height(root)<<endl;
}
//Find height of the tree (recursive)
int Tree :: Height(Node *temp){
    int L=0,R=0;
    if(temp==NULL)
        return 0;
    L=Height(temp->left);
    R=Height(temp->right);
    return(L>R?L+1:R+1);
}

//function to call function that counts leaf and
internal nodes of user-entered tree
void Tree :: LICall(){
    cout<<"Number of leaves:
"<<LeafCount(root)<<"\n";
    cout<<"Number of internal nodes:
"<<InternalCount(root)<<endl;
}
//counts leaf nodes recursively (nodes with left
and right children NULL)
int Tree :: LeafCount(Node *temp){
```

```
    if(temp==NULL)
        return 0; //terminating
    else if(temp->right==NULL&&temp-
>left==NULL)
        return 1;//checks if leaf node
    else
        return (LeafCount(temp->left)
+LeafCount(temp->right));//returns number of
leaf node children of current node
}
//counts nodes in the tree that are neither the
root or leaf nodes (internal) recursively
int Tree :: InternalCount(Node *temp){
    if(temp==NULL ||(temp->right==NULL && temp-
>left==NULL))
        //checks that current node is neither a root
(condition 1) nor a leaf (condition 2)
        return 0;
    else
        return (1+InternalCount(temp->left)
+InternalCount(temp->right));
}

//function to call mirroring function
```

```
void Tree :: MCall(){
    Mirror(root);
    Display();
}
void Tree :: Mirror(Node *temp){
    Node *n;
    if(temp==NULL)
        return;
    else{
        Mirror(temp->left);
        Mirror(temp->right);

        //swapping left and right children of each
node to mirror
        n=temp->left;
        temp->left=temp->right;//mirroring here
        temp->right=n;
    }

}

//overloading = to copy a Tree object into
another Tree object
void Tree :: operator=(Tree tree){
```

```
    if(tree.root==NULL)
        return;
    root=tree.root;
    Copy(tree.root->left, root->left);
    Copy(tree.root->right, root->right);
}
void Copy(Node *temp, Node *ctemp){
    if(temp==NULL)
        return;
    ctemp=new Node(temp->data);
    Copy(temp->left,ctemp->left);
    Copy(temp->right,ctemp->right);
}

//function to call Erase function
void Tree :: ECall(){
    Erase(root);
}
//deallocates space node by node
void Tree :: Erase(Node *temp){
    if(temp==NULL)
        return;
    else{
        Erase(temp->left); //go to left subtree, erase
```

```
first
        Erase(temp->right);//then go to right
subtree, erase
        cout<<"Deleted: "<<temp->data<<" ";
        delete temp;
        temp=NULL; //to prevent dangling pointer
    }
}

//driver function to test all above functions
int main(){
    int x;
    Tree T,TCopy;
    do{
        cout<<"\nEnter: "
            <<"\n1. Make Tree "
            <<"\n2. Traversal "
            <<"\n3. Height "
            <<"\n4. Mirror "
            <<"\n5. Copy "
            <<"\n6. Number of leaves, internal nodes"
            <<"\n7. Erase "
            <<"\n8. Exit \n";
        cin>>x;
```

```
        switch(x){
            case 1: T.Create();
                    break;
            case 2: T.Display();
                    break;
            case 3: T.HCall();
                    break;
            case 4: T.MCall();
                    break;
            case 5:    TCopy = T;
                    cout<<"Successfully copied!\n";
                    TCopy.Display();
                    break;
            case 6:    TCopy.LICall();
                    break;
            case 7:    T.ECall();
                    break;
            case 8: break;
            default: cout<<"Enter number between 1 through 8 only! \n";
        }
    }while(x!=8);
    return 0;
}
```