Tree Data Structure

Tree is a non-linear data structure that stores the data in a hierarchal form.

Node Class:

The Node class represents a single node in the binary tree. Each node has an integer value (val) and pointers to its left and right children. The constructor initializes the node with a given value and sets both children to NULL.

Tree Class:

The Tree class represents the binary tree and contains various methods to manipulate and analyze the tree.

Tree has a pointer to the root node, which is initially set to NULL in the constructor.

deleteTree(Node* leaf):

This is a recursive function used to delete the entire binary tree starting from a given node leaf. It deletes nodes in a post-order traversal, ensuring that child nodes are deleted before their parent. This function is not used in the main function, but it's a useful utility function for memory management.

Node* parent(Node* curr, Node* p, Node* par):

The function takes three arguments:

curr: A pointer to the current node in the binary tree.

p: A pointer to the node for which you want to find the parent.

par: A pointer to the parent of the current node curr.

The function works in a recursive manner to traverse the binary tree to find the parent of the target node p. It follows these cases:

- If the curr node is NULL, it means there is no parent to be found, and the function returns NULL. This is the base case that signifies the end of a branch.
- If the curr node is equal to the target node p, it means the parent has been found, and the function returns the node par as the parent of p.
- If neither of the above cases is met, the function proceeds to search for the parent in the left and right subtrees of the current node curr.

The function uses a temporary pointer t to store the result of the recursive calls. It first attempts to find the parent in the left subtree:

It calls parent(curr->left, p, curr) to search in the left subtree and assigns the result to t.

If t is not NULL, it means the parent has been found in the left subtree, and the function returns t.

If the parent is not found in the left subtree, the function proceeds to search in the right subtree:

It calls parent(curr->right, p, curr) to search in the right subtree and assigns the result to t.

If t is not NULL, it means the parent has been found in the right subtree, and the function returns t.

int maxDepth(Node* n):

The function starts with a base case. If the current node n is NULL, it means we've reached the end of a branch in the tree. In this case, we return -1 to indicate that there are no levels or depths beyond this point.

- If the current node n is not NULL, we proceed to calculate the depth of the tree rooted at this node:
- leftDepth is calculated by recursively calling maxDepth on the left subtree of the current node (n->left). This step finds the depth of the left subtree.
- rightDepth is calculated by recursively calling maxDepth on the right subtree of the current node (n->right). This step finds the depth of the right subtree.
- The function then compares leftDepth and rightDepth to determine which one is greater. This is done to find the maximum depth of the two subtrees.
- Finally, the function returns the maximum of leftDepth and rightDepth, incremented by 1. The addition of 1 is essential because it accounts for the current node in the calculation of the depth.
- When you call t.maxDepth(t.root) in the main function, it calculates and returns the maximum depth of the entire tree rooted at the root node. This value represents the longest path from the root to a leaf node in the binary tree.

Node* sibling(Node* root, Node* curr):

The function takes two arguments:

root: A pointer to the root of the binary tree.

curr: A pointer to the node for which you want to find the sibling.

- To find the sibling, it first calls the parent function to determine the parent of the current node curr. The parent function is expected to return the parent node of the given tree (root) and the current node (curr).
- Once the parent node par is identified, the function checks whether the given node curr is the left child or the right child of its parent:
- If par->left is equal to curr, it means curr is the left child. In this case, the function returns par->right as the sibling, which is the right child.
- If par->right is equal to curr, it means curr is the right child. In this case, the function returns par->left as the sibling, which is the left child.

preOrder(Node* n):

The pre-order traversal is a depth-first traversal technique in which each node is processed in the following order:

- Visit the current node (print its value or perform any other desired operation).
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.

inOrder(Node* n):

In an in-order traversal, each node is processed in the following order:

- Recursively traverse the left subtree.
- Visit the current node (print its value or perform any other desired operation).
- Recursively traverse the right subtree.

postOrder(Node* n):

Post-order traversal processes each node in the following order:

- Recursively traverse the left subtree.
- Recursively traverse the right subtree.
- Visit the current node (print its value or perform any other desired operation).

```
#include<iostream>
using namespace std;
class Node {
	public:
		int val;
		Node *left;
		Node *right;
		Node(int val) {
			this->val=val;
			left=NULL;
			right=NULL;
		}
};
class Tree {
	public:
		Node *root;
		Tree() {
			root=NULL;
		}
		void deleteTree(Node* leaf) {
			if (leaf != NULL) {
				deleteTree(leaf->left);
				deleteTree(leaf->right);
				delete leaf;
			}
		}
```

```
Node *parent(Node *curr, Node *p, Node *par) {
        if(curr==NULL) {
                return NULL;
        } else if(curr==p) {
                return par;
        } else {
                Node *t=parent(curr->left, p, curr);
                if(t!=NULL) {
                        return t;
                } else {
                        t=parent(curr->right, p, curr);
                        if(t!=NULL) {
                                return t;
                        }
                }
        }
}
int maxDepth(Node*n) {
        if(n==NULL) {
                return -1;
        } else {
                int leftDepth=maxDepth(n->left);
                int rightDepth=maxDepth(n->right);
                if(leftDepth>rightDepth) {
                        return (leftDepth+1);
                } else {
                        return (rightDepth+1);
                }
        }
}
```

```
Node *sibling(Node *root, Node *curr) {
        // find its parent
        Node *par= parent(root, curr, root);
        if(par->left==curr) {
                return par->right;
        } else {
                return par->left;
        }
}
void preOrder(Node *n) {
        if(n == NULL)
                return;

        cout<<n->val<<"\t";
        preOrder(n->left);
        preOrder(n->right);
}

void inOrder(Node *n) {
        if(n == NULL)
                return;

        inOrder(n->left);
        cout<<n->val<<"\t";
        inOrder(n->right);
}
```

```
        void postOrder(Node *n) {
            if(n == NULL)
                return;
            postOrder(n->left);
            postOrder(n->right);
            cout<<n->val<<"\t";
        }
};
int main() {
    Tree t;
    t.root=new Node(10);
    t.root->left=new Node(9);
    t.root->right=new Node(8);
    t.root->left->left=new Node(1);
    t.root->left->right=new Node(6);
    Node *n=t.root->left->right;
    Node *par=t.parent(t.root, n, t.root);
    if(par==NULL) {
        cout<<"Not found"<<endl;
    } else {
        cout<<"Parent of: "<<n->val<<" is: "<<par->val<<endl;
    }
    Node *sib=t.sibling(t.root, n);
    if(sib==NULL) {
        cout<<"Not found"<<endl;
    } else {
        cout<<"Sibling of: "<<n->val<<" is: "<<sib->val<<endl;
    }
```

```
        int depth = t.maxDepth(t.root);
        cout<<"Depth of root is: "<<depth<<endl;
        cout<<"\nInorder traversal: "<<endl;
        t.inOrder(t.root);
                cout<<"\nPreorder traversal: "<<endl;
        t.preOrder(t.root);
                cout<<"\nPostorder traversal: "<<endl;
        t.postOrder(t.root);
}
```

Output:

Parent of: 6 is: 9

Sibling of: 6 is: 1

Depth of root is: 2

Inorder traversal:

1 9 6 10 8

Preorder traversal:

10 9 1 6 8

Postorder traversal:

1 6 9 8 10