Tree Data Structure

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

In this C++ code, we're building a binary tree and demonstrating three common tree traversal algorithms: preorder, inorder, and postorder. Here we have a class Node that represent the value of a node, and a pointer to its left and right child. The binary tree is constructed by creating a root node with the character 'A'. Subsequently, child nodes are created and linked to the root node, forming the structure of the tree. For instance, 'B' is assigned as the left child of 'A', and 'C' as the right child. The 'D' node is added as the right child of 'B', and so on, building a hierarchical tree structure.

Now, let's delve into how each tree traversal algorithm works:

Preorder Traversal:

- Preorder traversal begins at the root node ('A').
- It prints the data of the current node ('A').
- Then, it recursively visits the left subtree ('B') and prints its data.
- Next, it moves to the right subtree ('C') and prints its data.
- Within each subtree, the traversal follows the same pattern: root, left, right.
- The result is a sequence of nodes visited in the order of Root-Left-Right: 'A', 'B', 'D', 'C', 'E', 'G', 'F', 'H', 'I'.

Inorder Traversal:

- Inorder traversal starts at the leftmost node ('B').
- It recursively traverses the left subtree ('B') and prints its data ('B').
- Then, it moves to the root ('A') and prints its data ('A').
- Subsequently, it proceeds to the right subtree ('D') and prints its data ('D').
- The traversal continues in this fashion: Left-Root-Right.
- The result is a sequence of nodes visited in the order of Left-Root-Right: 'B', 'D', 'A', 'G', 'E', 'C', 'H', 'F', 'I'.

Postorder Traversal:

- Postorder traversal begins at the leftmost leaf node ('D').
- It recursively traverses the left subtree ('D') and prints its data ('D').
- Then, it moves to the right subtree ('B') and prints its data ('B').
- After processing the subtrees, it returns to the root ('A') and prints its data ('A').
- Subsequently, it proceeds to the right subtree ('G') and prints its data ('G').

- The traversal continues in the pattern of Left-Right-Root.
- The result is a sequence of nodes visited in the order of Left-Right-Root: 'D', 'B', 'G', 'E', 'H', 'I', 'F', 'C', 'A'.

```
#include<iostream>
using namespace std;
class Node {
	public:
		char data;
		Node *left;
		Node *right;
		Node(char data) {
			this->data=data;
			left=NULL;
			right=NULL;
		}
};
void preOrder(Node *root) {
	if(root==NULL) {
		return;
	} else {
		cout<<root->data<<"  ";
		preOrder(root->left);
		preOrder(root->right);
	}
}
```

```
void inOrder(Node *root) {
	if(root==NULL) {
		return;
	} else {
		inOrder(root->left);
		cout<<root->data<<"  ";
		inOrder(root->right);
	}
}
void postOrder(Node *root) {
	if(root==NULL) {
		return;
	} else {
		postOrder(root->left);
		postOrder(root->right);
		cout<<root->data<<"  ";
	}
}
int main() {
	Node *root=new Node('A');
	root->left=new Node('B');
	root->right=new Node('C');
	root->left->right=new Node('D');
	root->right->left=new Node('E');
	root->right->right=new Node('F');
	root->right->left->left=new Node('G');
	root->right->right->left=new Node('H');
	root->right->right->right=new Node('I');
```

```
cout<<"Preorder traversal:  ";
        preOrder(root);
        cout<<"\nInorder traversal: ";
        inOrder(root);
        cout<<"\nPost order traversal: ";
        postOrder(root);
}
```

Output:

Preorder traversal: A B D C E G F H I

Inorder traversal: B D A G E C H F I

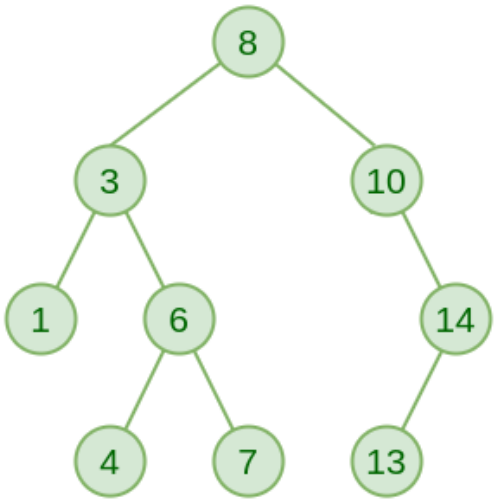
Post order traversal: D B G E H I F C A

Please note that for better visualization, run the code in Python tutor or in your notebook.

What is Binary Search Tree?

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

Node Class:

The code defines a Node class to represent individual nodes in the binary search tree. Each node contains an integer value (data) and two pointers, left and right, which point to its left and right children, respectively. In the constructor, a new node is initialized with the given data, and its children pointers are set to NULL.

Tree Class:

The Tree class represents the binary search tree. It contains a pointer root that points to the root node of the tree. Initially, the root is set to NULL.

Insertion (insert) Function:

The insert function is used to insert a new node with a given value into the binary search tree.

It takes two arguments: the current root of the subtree and the key to be inserted.

If the root is NULL, indicating an empty subtree, it creates a new node with the given key and returns it.

If the key is less than the current node's data, it recursively inserts the key into the left subtree.

If the key is greater than the current node's data, it recursively inserts the key into the right subtree.

The function returns the root of the current subtree, ensuring that the tree structure is maintained.

Search Function:

The search function is used to search for a specific key within the binary search tree.

It takes two arguments: the current root of the subtree and the key to be searched.

If the root is NULL or the current node's data matches the key, it returns the current root.

If the key is less than the current node's data, it recursively searches in the left subtree.

If the key is greater than the current node's data, it recursively searches in the right subtree.

Main Function:

In the main function, an instance of the Tree class is created (t), and the root of the tree is initially set to NULL.

Several values are inserted into the tree using the insert method.

An in-order traversal of the tree is performed using the inOrder method, which prints the values in ascending order.

A search is performed for two values: 5 and 0. The result of the search is checked, and a message is printed accordingly.

```
#include<iostream>
using namespace std;
class Node {
	public:
		int data;
		Node *left;
		Node *right;
		Node(int data) {
			this->data=data;
			left=NULL;
			right=NULL;
		}
};
class Tree {
	public:
		Node *root;
		Tree() {
			root=NULL;
		}
		~Tree() {
			destroyTree(root);
		}
		void destroyTree(Node *node) {
			if (node == NULL) {
				return;
			}
			destroyTree(node->left);
			destroyTree(node->right);
			delete node;
		}
```

```
        Node * search(Node *root, int key) {
                if(root==NULL || root->data==key) {
                        return root;
                } else if(key < root->data) {
                        return search(root->left, key);
                } else if(key > root->data) {
                        return search(root->right, key);
                }
        }

        Node * insert(Node *root, int key) {
                if(root==NULL) {
                        return new Node(key);
                } else if(key < root->data) {
                        root->left=insert(root->left, key);
                } else if(key > root->data) {
                        root->right=insert(root->right, key);
                }
        return root;
        }
        void inOrder(Node *root) {
                if(root==NULL) {
                        return;
                } else {
                        inOrder(root->left);
                        cout<<root->data<<"  ";
                        inOrder(root->right);
                }
        }

};
```

```
int main() {
	Tree t;
	Node *root=t.insert(NULL, 2);
	t.insert(root, 3);
	t.insert(root, 1);
	t.insert(root, 5);
	t.insert(root, 8);
	t.insert(root, 7);
	t.inOrder(root);
	root = t.search(root, 5);
	if(root!=NULL)
	{
		cout<<"\nValue found"<<endl;
	}
	else
	{
		cout<<"Value not found"<<endl;
	}
}
```

Output:

1 2 3 5 7 8

Value found

Now try to handle deletion in the tree with all 3 cases yourself.