CSE225L – Data Structures and Algorithms Lab
Lab 13
Binary Search Tree

In today's lab we will design and implement the Binary Search Tree.

bst.h

```cpp
#ifndef BST_H
#define BST_H

struct Node {
    int data;
    Node* left;
    Node* right;
};

enum TraversalType {
    IN_ORDER,
    PRE_ORDER,
    POST_ORDER
};

class BST {
private:
    Node* root;
    Node* Insert(Node* root, int data);
    bool Search(Node* root, int data);
    Node* Delete(Node* root, int data);
    void Print(Node* root, TraversalType type);
    Node* findMin(Node* root);
    void deleteTree(Node* root);

public:
    BST();
    ~BST();
    void Insert(int data);
    bool Search(int data);
    void Delete(int data);
    void MakeEmpty();
    void Print(TraversalType type);
};

#endif // BST_H
```

bst.cpp

```cpp
#include "bst.h"
#include <iostream>
using namespace std;

BST::BST()
{
    root = NULL;
}

BST::~BST()
{
    deleteTree(root);
}

void BST::Insert(int data)
{
```

```
    root = Insert(root, data);
}

bool BST::Search(int data)
{
    return Search(root, data);
}

void BST::Delete(int data)
{
    root = Delete(root, data);
}

void BST::Print(TraversalType type)
{
    cout << "BST (";
    switch (type)
    {
        case IN_ORDER:    cout << "In-order): ";    break;
        case PRE_ORDER:   cout << "Pre-order): ";   break;
        case POST_ORDER:  cout << "Post-order): ";  break;
    }
    Print(root, type);
    cout << endl;
}

Node* BST::Insert(Node* root, int data)
{
    if (root == NULL)
    {
        Node* newNode = new Node();
        newNode->data = data;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }

    // Otherwise, recursively Insert the value
    if (data < root->data)
    {
        root->left = Insert(root->left, data);  // Insert in left subtree
    }
    else
    {
        root->right = Insert(root->right, data); // Insert in right subtree
    }
    return root;
}

bool BST::Search(Node* root, int data)
{
    if (root == NULL)
    {
        return false;  // Base case: not found
    }
    if (data == root->data)
    {
        return true;  // Found the node
    }
    if (data < root->data)
    {
        return Search(root->left, data);   // Search in the left subtree
    }
    else
    {
        return Search(root->right, data);  // Search in the right subtree
    }
}

Node* BST::Delete(Node* root, int data)
```

```cpp
{
    if (root == NULL)
    {
        return root;  // Node to delete not found
    }

    // Recurse down the tree to find the node to delete
    if (data < root->data)
    {
        root->left = Delete(root->left, data);
    }
    else if (data > root->data)
    {
        root->right = Delete(root->right, data);
    }
    else
    {
        // Node found: now handle deletion
        // Case 1: Node has no child (leaf node)
        if (root->left == NULL && root->right == NULL)
        {
            delete root;
            return NULL;
        }
        // Case 2: Node has one child
        else if (root->left == NULL)
        {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == NULL)
        {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Case 3: Node has two children
        else
        {
            // Find the Print successor (smallest in the right subtree)
            Node* temp = findMin(root->right);
            root->data = temp->data;  // Copy the Print successor's data to root
            root->right = Delete(root->right, temp->data);  // Delete the Print successor
        }
    }
    return root;
}

void BST::Print(Node* root, TraversalType type)
{
    if (root == NULL)
    {
        return;
    }

    switch (type)
    {
        case IN_ORDER:
            Print(root->left, type);          // Traverse left subtree
            cout << root->data << " ";        // Visit node
            Print(root->right, type);         // Traverse right subtree
            break;

        case PRE_ORDER:
            cout << root->data << " ";        // Visit node
            Print(root->left, type);          // Traverse left subtree
            Print(root->right, type);         // Traverse right subtree
            break;
```

```
        case POST_ORDER:
            Print(root->left, type);          // Traverse left subtree
            Print(root->right, type);         // Traverse right subtree
            cout << root->data << " ";        // Visit node
            break;
    }
}

void BST::deleteTree(Node* root)
{
    if (root != NULL)
    {
        deleteTree(root->left);  // Delete left subtree
        deleteTree(root->right); // Delete right subtree
        delete root;             // Delete the node
    }
}

Node* BST::findMin(Node* root)
{
    while (root && root->left != NULL)
    {
        root = root->left;        // Keep going left to find the minimum
    }
    return root;
}
```

Generate the **driver file (main.cpp)** where you perform the following tasks. Note that you cannot make any change to the header file or the source file.

Operation to Be Tested and Description of Action	Input Values	Expected Output
Create a tree object		
Insert seven items	50 30 20 40 70 60 80	
Print the elements in the tree (inorder)		20 30 40 50 60 70 80
Print the elements in the tree (preorder)		50 30 20 40 70 60 80
Print the elements in the tree (postorder)		20 40 30 60 80 70 50
Retrieve 20 and print whether found or not		Found
Retrieve 15 and print whether found or not		Not Found
Delete 20 from the tree		
Print the elements in the tree (inorder)		30 40 50 60 70 80
Print the elements in the tree (preorder)		50 30 40 70 60 80
Print the elements in the tree (postorder)		40 30 60 80 70 50