LAB 14

CSE225L

BINARY SEARCH TREE (BST)

In this lab, we will:

- Design and implement the Binary Search Tree (BST).
- Implement core **BST** operations such as insertion, searching, deletion, and tree traversal.
- Explore and understand different tree traversal methods: inorder, preorder, and postorder.
- Test the functionality of the **BST** class by performing various operations on a set of integer values.

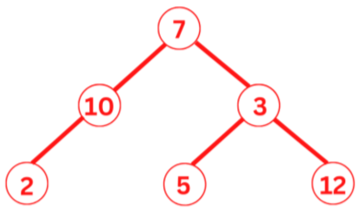
BINARY SEARCH TREE (BST)

bst.h

```
#ifndef BST_H_INCLUDED
#define BST_H_INCLUDED
#include "queuetype.h"

template <class T>
struct Node
{
    T data;
    Node* left;
    Node* right;
};

enum OrderType {PRE_ORDER, IN_ORDER, POST_ORDER};

template <class T>
class BST
{
private:
    Node<T>* root;
    QueueType<T> preQue;
    QueueType<T> inQue;
    QueueType<T> postQue;

public:
    BST();
    ~BST();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
    int Length();
    void Insert(T value);
    void Search(T& value, bool& found);
    void Delete(T value);
    void GetNext(T& value, OrderType order);
    void Reset(OrderType order);
    void Print();
};
#endif // BST_H_INCLUDED
```

✗

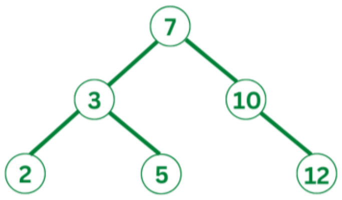
Not a binary search tree



This is just a binary tree

✓

Yes a binary search tree



all values to L (incl. L subtree) are < parent value

all values to R (incl. R subtree) are > parent value bst.cpp

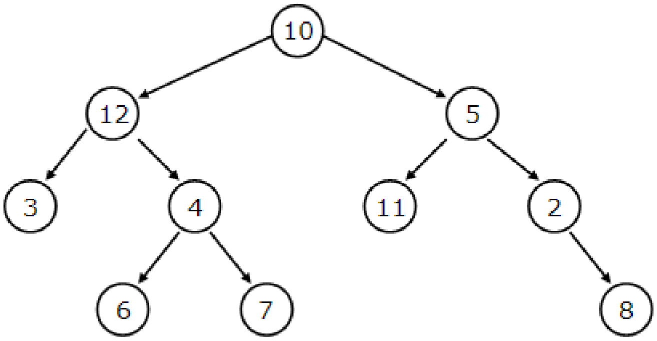
```
#include "bst.h"
#include "queuetype.cpp"
#include <iostream>
using namespace std;

template <class T>
BST<T>::BST()
{
    root = NULL;
}

template <class T>
void Destroy(Node<T>* &tree)
{
    if (tree != NULL)
    {
        Destroy(tree->left);
        Destroy(tree->right);
        delete tree;
        tree = NULL;
    }
}

template <class T>
BST<T>::~BST()
{
    Destroy(root);
}

template <class T>
void BST<T>::MakeEmpty()
{
    Destroy(root);
}

template <class T>
bool BST<T>::IsEmpty()
{
    return root == NULL;
}

template <class T>
bool BST<T>::IsFull()
{
    try
    {
        Node<T>* temp = new Node<T>;
        delete temp;
        return false;
    }
    catch(bad_alloc& exception)
    {
        return true;
    }
}
```

```
template <class T>
int CountNodes(Node<T>* tree)
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) + CountNodes(tree->right) + 1;
}

template <class T>
int BST<T>::Length()
{
    return CountNodes(root);
}

template <class T>
void SearchValue(Node<T>* tree, T &value, bool &found)
{
    if (tree == NULL)
        found = false;
    else if (value < tree->data)
        SearchValue(tree->left, value, found);
    else if (value > tree->data)
        SearchValue(tree->right, value, found);
    else
    {
        value = tree->data;
        found = true;
    }
}

template <class T>
void BST<T>::Search(T &value, bool &found)
{
    SearchValue(root, value, found);
}

template <class T>
void InsertValue(Node<T>* &tree, T value)
{
    if (tree == NULL)
    {
        tree = new Node<T>;
        tree->right = NULL;
        tree->left = NULL;
        tree->data = value;
    }
    // Otherwise, recursively Insert the value
    else if (value < tree->data)
        InsertValue(tree->left, value); // Insert in left subtree
    else
        InsertValue(tree->right, value); // Insert in right subtree
}
```

```
template <class T>
void BST<T>::Insert(T value)
{
    InsertValue(root, value);
}

template <class T>
void DeleteValue(Node<T>* &tree, T value)
{
    if (value < tree->data)
        DeleteValue(tree->left, value);
    else if (value > tree->data)
        DeleteValue(tree->right, value);
    else
        DeleteNode(tree);
}

template <class T>
void DeleteNode(Node<T>* &tree)
{
    T data;
    Node<T>* temp = tree;
    if (tree->left == NULL)
    {
        tree = tree->right;
        delete temp;
    }
    else if (tree->right == NULL)
    {
        tree = tree->left;
        delete temp;
    }
    else
    {
        GetPredecessor(tree->left, data);
        tree->data = data;
        DeleteValue(tree->left, data);
    }
}

template <class T>
void GetPredecessor(Node<T> *tree, T &data)
{
    while (tree->right != NULL)
    {
        tree = tree->right;
    }
    data = tree->data;
}

template <class T>
void BST<T>::Delete(T value)
{
    DeleteValue(root, value);
}
```

```
template <class T>
void PreOrder(Node<T> *tree, QueueType<T> &queue)
{
    if (tree != NULL)
    {
        queue.Enqueue(tree->data);
        PreOrder(tree->left, queue);
        PreOrder(tree->right, queue);
    }
}

template <class T>
void InOrder(Node<T> *tree, QueueType<T> &queue)
{
    if (tree != NULL)
    {
        InOrder(tree->left, queue);
        queue.Enqueue(tree->data);
        InOrder(tree->right, queue);
    }
}

template <class T>
void PostOrder(Node<T>* tree, QueueType<T> &queue)
{
    if (tree != NULL)
    {
        PostOrder(tree->left, queue);
        PostOrder(tree->right, queue);
        queue.Enqueue(tree->data);
    }
}

template <class T>
void BST<T>::GetNext(T &value, OrderType order)
{
    bool finished = false;

    switch (order)
    {
    case PRE_ORDER:
        preQue.Dequeue(value);
        if(preQue.IsEmpty())
            finished = true;
        break;
    case IN_ORDER:
        inQue.Dequeue(value);
        if(inQue.IsEmpty())
            finished = true;
        break;
    case POST_ORDER:
        postQue.Dequeue(value);
        if(postQue.IsEmpty())
            finished = true;
        break;
    }
}
```

```
template<class T>
void BST<T>::Reset(OrderType order)
{
    switch (order)
    {
    case PRE_ORDER:
        PreOrder(root, preQue);
        break;
    case IN_ORDER:
        InOrder(root, inQue);
        break;
    case POST_ORDER:
        PostOrder(root, postQue);
        break;
    }
}

template <class T>
void PrintTree(Node<T>* tree)
{
    if (tree != NULL)
    {
        PrintTree(tree->left);
        cout << tree->data << " ";
        PrintTree(tree->right);
    }
}

template <class T>
void BST<T>::Print()
{
    PrintTree(root);
}
```



Levelorder tree traversal
10, 12, 5, 3, 4, 11, 2, 6, 7, 8

Inorder tree traversal
3, 12, 6, 4, 7, 10, 11, 5, 2, 8

Preorder tree traversal
10, 12, 3, 4, 6, 7, 5, 11, 2, 8

Postorder tree traversal
3, 6, 7, 4, 12, 11, 8, 2, 5, 10

Inorder: Visit the left subtree, then the root node, then the right subtree
Preorder: Visit the root node first, then the left subtree, and finally the right subtree.
Postorder: Visit the left subtree first, then the right subtree, and finally the root node.

BINARY SEARCH TREE (BST)

TASKS:

Instructions:

- Create the driver file (main.cpp) and perform the following tasks.

Operation	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
Search 20 and print whether found or not		Found
Search 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

Tasks 1

Given a sequence of integers, determine the best ordering of the integers to insert them into a binary search tree. The best order is the one that will allow the binary search tree to have the minimum height.

Hint: Sort the sequence (use the inorder traversal). The middle element becomes the root. Insert it into an empty tree. Then, in the same way, recursively build the left subtree and the right subtree.

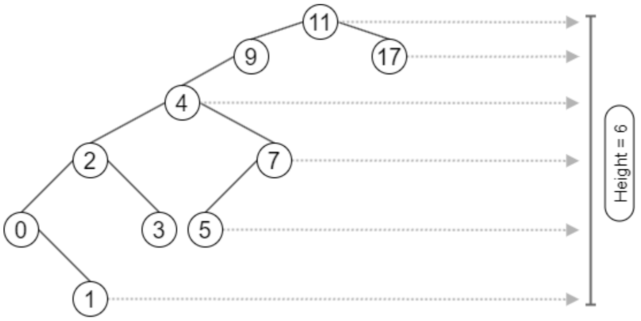
Input Values

```
11 9 4 2 7 3 17 0 5 1
```

Expected Output

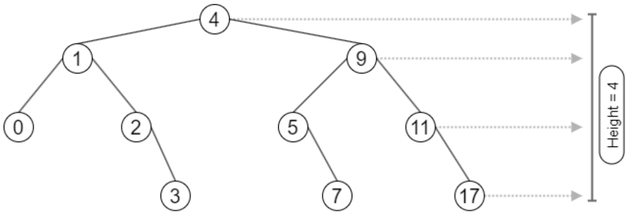
Without Optimal Ordering:

```
Inorder:  0 1 2 3 4 5 7 9 11 17
Preorder:  11 9 4 2 0 1 3 7 5 17
Postorder:  1 0 3 2 5 7 4 9 17 11
```



With Optimal Ordering:

```
Inorder:  0 1 2 3 4 5 7 9 11 17
Preorder:  4 1 0 2 3 9 5 7 11 17
Postorder:  0 3 2 1 7 5 17 11 9 4
```



Task 1 (C++ Implementation):

Add the following code to the **.h** and **.cpp** files of the **bst** class.

bst.h

```
template <class T>
class BST
{
private:
    void BuildBST(T arr[], int start, int end);

public:
    void BuildOptimalTree();
};
```

bst.cpp

```
template <class T>
void BST<T>::BuildBST(T arr[], int start, int end)
{
    if (start <= end)
    {
        int mid = (start + end) / 2;
        Insert(arr[mid]);
        BuildBST(arr, start, mid - 1);
        BuildBST(arr, mid + 1, end);
    }
}

template <class T>
void BST<T>::BuildOptimalTree()
{
    T arr[Length()];
    T value;
    int n = sizeof(arr) / sizeof(arr[0]);

    Reset(IN_ORDER);
    for(int i = 0; i < Length(); i++)
    {
        GetNext(value, IN_ORDER);
        arr[i] = value;
    }
    Destroy(root);
    BuildBST(arr, 0, n-1);
}
```