

CSE225L – Data Structures and Algorithms Lab

Lab 13

Binary Search Tree

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

binarysearchtree.h

```
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED
#include "quetype.h"
template <class ItemType>
struct TreeNode
{
    ItemType info;
    TreeNode* left;
    TreeNode* right;
};
enum OrderType {PRE_ORDER, IN_ORDER,
POST_ORDER};
template <class ItemType>
class TreeType
{
public:
    TreeType();
    ~TreeType();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
    int LengthIs();
    void RetrieveItem(ItemType& item,
bool& found);
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void ResetTree(OrderType order);
    void GetNextItem(ItemType& item,
OrderType order, bool& finished);
    void Print();
private:
    TreeNode<ItemType>* root;
    QueType<ItemType> preQue;
    QueType<ItemType> inQue;
    QueType<ItemType> postQue;
};
#endif // BINARYSEARCHTREE_H_INCLUDED
```

binarysearchtree.cpp

```
#include "binarysearchtree.h"
#include "quetype.cpp"
#include <iostream>
using namespace std;
template <class ItemType>
TreeType<ItemType>::~TreeType()
{
    root = NULL;
}
template <class ItemType>
void Destroy(TreeNode<ItemType>*& tree)
{
    if (tree != NULL)
    {
        Destroy(tree->left);
        Destroy(tree->right);
        delete tree;
        tree = NULL;
    }
}
template <class ItemType>
TreeType<ItemType>::~~TreeType()
{
    Destroy(root);
}
template <class ItemType>
void TreeType<ItemType>::MakeEmpty()
{
    Destroy(root);
}
```

```
template <class ItemType>
bool TreeType<ItemType>::IsEmpty()
{
    return root == NULL;
}
template <class ItemType>
bool TreeType<ItemType>::IsFull()
{
    TreeNode<ItemType>* location;
    try
    {
        location = new TreeNode<ItemType>;
        delete location;
        return false;
    }
    catch(bad_alloc& exception)
    {
        return true;
    }
}
template <class ItemType>
int CountNodes(TreeNode<ItemType>* tree)
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) +
CountNodes(tree->right) + 1;
}
template <class ItemType>
int TreeType<ItemType>::LengthIs()
{
    return CountNodes(root);
}
template <class ItemType>
void Retrieve(TreeNode<ItemType>* tree, ItemType&
item, bool& found)
{
    if (tree == NULL)
        found = false;
    else if (item < tree->info)
        Retrieve(tree->left, item, found);
    else if (item > tree->info)
        Retrieve(tree->right, item, found);
    else
    {
        item = tree->info;
        found = true;
    }
}
template <class ItemType>
void TreeType<ItemType>::RetrieveItem(ItemType&
item, bool& found)
{
    Retrieve(root, item, found);
}
```

```

template <class ItemType>
void Insert(TreeNode<ItemType>*& tree,
ItemType item)
{
    if (tree == NULL)
    {
        tree = new TreeNode<ItemType>;
        tree->right = NULL;
        tree->left = NULL;
        tree->info = item;
    }
    else if (item < tree->info)
        Insert(tree->left, item);
    else
        Insert(tree->right, item);
}
template <class ItemType>
void TreeType<ItemType>::InsertItem(ItemType
item)
{
    Insert(root, item);
}
template <class ItemType>
void Delete(TreeNode<ItemType>*& tree,
ItemType item)
{
    if (item < tree->info)
        Delete(tree->left, item);
    else if (item > tree->info)
        Delete(tree->right, item);
    else
        DeleteNode(tree);
}
template <class ItemType>
void DeleteNode(TreeNode<ItemType>*& tree)
{
    ItemType data;
    TreeNode<ItemType>* tempPtr;

    tempPtr = tree;
    if (tree->left == NULL)
    {
        tree = tree->right;
        delete tempPtr;
    }
    else if (tree->right == NULL)
    {
        tree = tree->left;
        delete tempPtr;
    }
    else
    {
        GetPredecessor(tree->left, data);
        tree->info = data;
        Delete(tree->left, data);
    }
}
template <class ItemType>
void GetPredecessor(TreeNode<ItemType>*&
tree, ItemType& data)
{
    while (tree->right != NULL)
        tree = tree->right;
    data = tree->info;
}
template <class ItemType>
void TreeType<ItemType>::DeleteItem(ItemType
item)
{
    Delete(root, item);
}

```

```

template <class ItemType>
void PreOrder(TreeNode<ItemType>* tree,
QueueType<ItemType>& Que)
{
    if (tree != NULL)
    {
        Que.Enqueue(tree->info);
        PreOrder(tree->left, Que);
        PreOrder(tree->right, Que);
    }
}
template <class ItemType>
void InOrder(TreeNode<ItemType>* tree,
QueueType<ItemType>& Que)
{
    if (tree != NULL)
    {
        InOrder(tree->left, Que);
        Que.Enqueue(tree->info);
        InOrder(tree->right, Que);
    }
}
template <class ItemType>
void PostOrder(TreeNode<ItemType>* tree,
QueueType<ItemType>& Que)
{
    if (tree != NULL)
    {
        PostOrder(tree->left, Que);
        PostOrder(tree->right, Que);
        Que.Enqueue(tree->info);
    }
}
template <class ItemType>
void TreeType<ItemType>::ResetTree(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 ItemType>
void TreeType<ItemType>::GetNextItem(ItemType&
item, OrderType order, bool& finished)
{
    finished = false;
    switch (order)
    {
        case PRE_ORDER:
            preQue.Dequeue(item);
            if(preQue.IsEmpty())
                finished = true;
            break;
        case IN_ORDER:
            inQue.Dequeue(item);
            if(inQue.IsEmpty())
                finished = true;
            break;
        case POST_ORDER:
            postQue.Dequeue(item);
            if(postQue.IsEmpty())
                finished = true;
            break;
    }
}

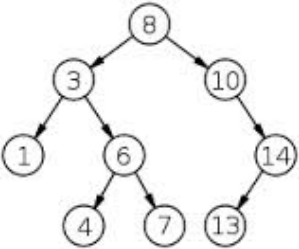
```

```

template <class ItemType>
void PrintTree(TreeNode<ItemType>* tree)
{
    if (tree != NULL)
    {
        PrintTree(tree->left);
        cout << tree->info << " ";
        PrintTree(tree->right);
    }
}
template <class ItemType>
void TreeType<ItemType>::Print()
{
    PrintTree(root);
}

```

Now generate the **Driver file (main.cpp)** where you perform the following tasks:

Operation to Be Tested and Description of Action	Input Values	Expected Output
<ul style="list-style-type: none"> Create a tree object 		
<ul style="list-style-type: none"> Print if the tree is empty or not 		Tree is empty
<ul style="list-style-type: none"> Insert ten items 	4 9 2 7 3 11 17 0 5 1	
<ul style="list-style-type: none"> Print if the tree is empty or not 		Tree is not empty
<ul style="list-style-type: none"> Print the length of the tree 		10
<ul style="list-style-type: none"> Retrieve 9 and print whether found or not 		Item is found
<ul style="list-style-type: none"> Retrieve 13 and print whether found or not 		Item is not found
<ul style="list-style-type: none"> Print the elements in the tree (inorder) 		0 1 2 3 4 5 7 9 11 17
<ul style="list-style-type: none"> Print the elements in the tree (preorder) 		4 2 0 1 3 9 7 5 11 17
<ul style="list-style-type: none"> Print the elements in the tree (postorder) 		1 0 3 2 5 7 17 11 9 4
<ul style="list-style-type: none"> Make the tree empty 		
<ul style="list-style-type: none"> Build the following tree inserting the elements, one by one 		
<ul style="list-style-type: none"> Add a member function to the TreeType class which returns the minimum element in the tree. <pre>int findMin();</pre>		1
<ul style="list-style-type: none"> Add a function to the TreeType class which returns the number of leaves in the tree. <pre>int numLeaves();</pre>		4