North South University

Department of Electrical and Computer Engineering

CSE 225L.13 (Data Structures and Algorithms Lab)

Lab 18: Binary Search Tree

Instructor: Syed Shahir Ahmed Rakin, Arfana Rahman

Objective:

- Learn about how Binary Search Trees work

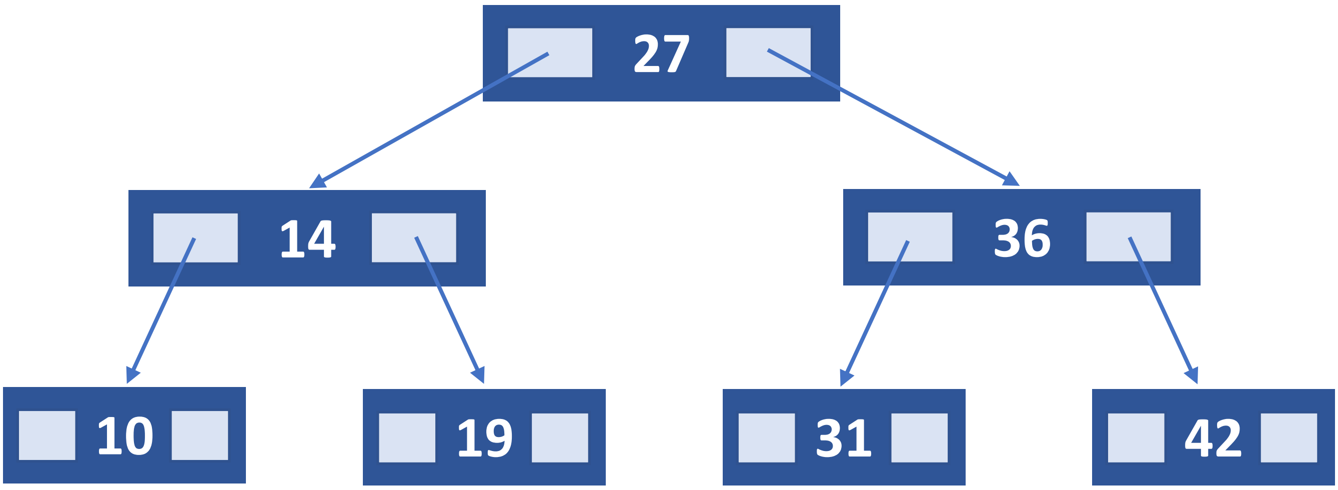
What is a Binary Search Tree:

A binary search tree (BST) is a tree in which ALL of the nodes involved follow the properties given below:

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

A binary search tree is a collection of nodes arranged in a way where they maintain the properties mentioned above. Each node in BST has a key and an associated value.

The binary search tree looks something like this:



You can see that the root is the uppermost node in the figure; the left sub-tree of the root has a key value less than the key value of the root itself. (14 < 27)

In contrast, the right sub-tree of the root has a key value greater than the key value of the root, which is another one of the BST properties. (35 > 27)

Binary Search Trees use nodes reminiscent of Linked Lists, except for one significant change: two pointers instead of one. These two pointers point to the left and right nodes, respectively. The info part of the node remained unchanged.

Prototype of Binary Search Trees:

The header and source file of the Binary Search Tree are given as follows.

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 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  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 PreOrder(TreeNode<ItemType>* tree,
              QueType<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,
             QueType<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,
QueType<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);
}
```

Tasks:

Operation to Be Tested and Description of Action	Input Values	Expected Output
• Create a tree object		
• Print if the Tree is empty or not		Tree is empty
• Insert ten items	4 9 2 7 3 11 17 0 5 1	
• Print if the Tree is empty or not		Tree is not empty.
• Print the length of the Tree		10
• Retrieve 9 and print whether found or not		Item is found
• Retrieve 13 and print whether found or not		Item is not found.
• Print the elements in the Tree (inorder)		0 1 2 3 4 5 7 9 11 17
• Print the elements in the Tree (preorder)		4 2 0 1 3 9 7 5 11 17
• Print the elements in the Tree (postorder)		1 0 3 2 5 7 17 11 9 4
• Make the Tree empty.		
• 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 is the root; insert it into an empty tree. In the same way, recursively build the left subtree and then the right subtree.	11 9 4 2 7 3 17 0 5 1	4 1 0 2 3 9 5 7 11 17