CSE 330 LABORATORY -- Week 6, Spring 2018

Instructor: Kerstin Voigt

In this lab, you will be implemeting the ADT called Binary<u>Search</u>Tree. This data stucture will provide us with an alternative to vectors and linked lists when it comes to storing a list of data values.

Exercise 1: Implement the BinarySearchTree ADT in a file `BinarySearchTree.h` exactly as shown below. *<u>As always, make an effort to copy mindfully, trying to understand the purpose of each line of code as you go along.</u>*

```
// BinarySearchTree.h
// afater Mark A. Weiss, Chapter 4

// KV replaced exceptions with assert statements;
// we are writing <typename C> to indicate that the template type must be
// "comparable", i.e., have defined <, > and == operators;

#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <cassert>
#include <algorithm>
using namespace std;


template <typename C>
class BinarySearchTree
{
  public:
    BinarySearchTree( ) : root{ nullptr }
    {
    }

    BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
    {
        root = clone( rhs.root );
    }

    BinarySearchTree( BinarySearchTree && rhs ) : root{ rhs.root }
    {
        rhs.root = nullptr;
    }


    ~BinarySearchTree( )
    {
        makeEmpty();
    }
```

```
BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    BinarySearchTree copy = rhs;
    std::swap( *this, copy );
    return *this;

}


BinarySearchTree & operator=( BinarySearchTree && rhs )
{
    std::swap( root, rhs.root );
    return *this;
}


const C & findMin( ) const
{
  assert(!isEmpty());
    return findMin( root )->element;
}


const C & findMax( ) const
{

  assert(!isEmpty());
  return findMax( root )->element;
}

bool contains( const C & x ) const
{
    return contains( x, root );
}


bool isEmpty( ) const
{
    return root == nullptr;
}


void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

void makeEmpty( )
{
    makeEmpty( root );
}
```

```
    void insert( const C & x )
    {
        insert( x, root );
    }


    void insert( C && x )
    {
        insert( std::move( x ), root );
    }

    void remove( const C & x )
    {
        remove( x, root );
    }


private:

    struct BinaryNode
    {
        C element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const C & theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ theElement }, left{ lt }, right{ rt } { }

        BinaryNode( C && theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
    };

    BinaryNode *root;



    // Internal method to insert into a subtree.
    // x is the item to insert.
    // t is the node that roots the subtree.
    // Set the new root of the subtree.

    void insert( const C & x, BinaryNode * & t )
    {
        if( t == nullptr )
            t = new BinaryNode{ x, nullptr, nullptr };
        else if( x < t->element )
            insert( x, t->left );
        else if( t->element < x )
            insert( x, t->right );
        else
            ;  // Duplicate; do nothing
    }
```

```
// Internal method to insert into a subtree.
// x is the item to insert.
// t is the node that roots the subtree.
// Set the new root of the subtree.

void insert( C && x, BinaryNode * & t )
{
    if( t == nullptr )
        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
    else if( x < t->element )
        insert( std::move( x ), t->left );
    else if( t->element < x )
        insert( std::move( x ), t->right );
    else
        ;  // Duplicate; do nothing
}


// Internal method to remove from a subtree.
// x is the item to remove.
// t is the node that roots the subtree.
// Set the new root of the subtree.

void remove( const C & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}

// Internal method to find the smallest item in a subtree t.
// Return node containing the smallest item.

BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}
```

```
// Internal method to find the largest item in a subtree t.
// Return node containing the largest item.
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}

// Internal method to test if an item is in a subtree.
// x is item to search for.
// t is the node that roots the subtree.

bool contains( const C & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}

void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}

void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}

BinaryNode * clone( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    else
        return new BinaryNode{ t->element,
                               clone( t->left ), clone( t->right ) };
}
```

```
};
#endif
```

Exercise 2: Program your own file BinarySearchTreeMain.cpp in which your main() function will test the new data structure. Declare an instance of BinarySearchTree (short: BST) suitable to hold integer values. Then enter a random sequence of 25 integer values into the data structure (your values should NOT be in sorted order).

Use the print_Tree () member function in order to print out the values of the BST structure – <u>What do you notice?</u>

Next, remove 5 values from your BST and save them in a vector (use your own Vector.h or STL <vector>). Print out the reduced BST.

Exercise 3: Next add the following member function do your BinarySearchTree class:

Under public:

```
void printInternal()
{
   print_Internal(root,0);
}
```

Under private:

```
void printInternal(BinaryNode* t, int offset)
{
   if (t == nullptr)
       return;

   for(int i = 1; i ≤ offset; i++)
       cout ≪ “..”
   cout ≪ t->element ≪ endl;
   printInternal(t->left, offset + 1);
   printInternal(t->right, offset + 1);
}
```

Go back to your program BinarySearchTreeMain.cpp and change `printTree` to `printInternal.` Compile and run your program, and see what you get.

Next, insert the 5 value that have been removed back into the BST. Print the new BST with printInternal. <u>How does this printed BST compare with the BST that the program printed befre the removal of 5 values?</u> Same? Different? Explanation?

Credit for this lab (Tuesday Group): (1) Sign up on the signup sheet. (2) Email to Sarthak by Thursday 5/10, 11:59pm, a copy/scan/photo of the completed worksheet “Algorithm Complexities of BinarySearchTree Functions”.

Name:___________________________

CSE 330 – Spring 2018 – Lab6 Worksheet

“Algorithm Complexities of BinarySearchTree Functions”

Indicate worst case complexity:

BinarySearchTree Member Fct	One of: O(1), O(logN), O(N), O(NlogN), $O(N^2)$
void insert(x)	
void remove(x)	
bool contains(x)	
C findMin()	
C findMax()	
bool isEmpty()	
void makeEmpty()	
void printTree()	

(8 lab points for this table)

For some reason, our ADT BinarySearchTree does not have a ‘int BinarySearchTree::size()’ member function. We can easily add one. Write your implemetation of the size() function by hand below. Recursion will make this easy.

```
template <typename C>
int BinarySearchTree<T>::size()
{
```

}

(2 lab points for this function)