Source.cpp

#include<iostream>

#include<time.h>

#include "BST\_ADT.h"

using std::cout;

using std::cin;

using std::endl;

class TestData {

public:

void setKey(int newKey) {

keyField = newKey;

}

int getKey() const {

return keyField;

} // Returns the key

private:

int keyField; // Key for the data item

};

int main() {

BSTree<TestData, int> testTree;

TestData testData;

int inputKey;

srand(time(NULL));

cout << "Inputting 7 random values" << endl;

for (int i = 0; i < 7; i++) {

inputKey = rand() %100 + 1;

testData.setKey(inputKey);

testTree.insert(testData);

}

cout << "Inputting 42" << endl;

inputKey = 42;

testData.setKey(inputKey);

testTree.insert(testData);

cout << "Tree:" << endl;

testTree.showStructure();

cout << endl;

cout << "Finding 42:" << endl;

cout << testTree.retrieve(inputKey, testData) << endl;

cout << "Choose a value to remove: ";

cin >> inputKey;

cout << endl;

testTree.remove(inputKey);

cout << "Tree after removal: " << endl;

testTree.showStructure();

cout << "Height: ";

cout << testTree.getHeight() << endl;

cout << "Count: ";

cout << testTree.getCount() << endl;

cout << "Keys: ";

testTree.writeKeys();

cout << endl << "less than 50: ";

testTree.writeLessThan(50);

return 0;

}

BSTADT.h

#ifndef BST\_ADT\_H

#define BST\_ADT\_H

#include<iostream>

#include <stdexcept>

#include <iostream>

#include <vector>

using namespace std;

template < typename DataType, class KeyType > // DataType : tree data item

class BSTree // KeyType : key field

{

public:

BSTree() {

clear();

}; // Default constructor

BSTree(const BSTree<DataType,KeyType>& other) {

clear();

copyHelper(root, other);

}; // gave up, couldnt get this to work

BSTree& operator= (const BSTree<DataType,KeyType>& other){ // couldnt get this to work either

copyHelper(root, other.root);

return \*this;

};

~BSTree() {

clear();

};

void insert(const DataType& newDataItem) {

insertHelper(root, newDataItem);

};

bool retrieve(const KeyType& searchKey, DataType& searchDataItem) const {

return retrieveHelper(root, searchKey, searchDataItem);

};

bool remove(const KeyType& deleteKey) {

return removeHelper(root, deleteKey);

};

void writeKeys() const {

writeKeysHelper(root);

}; // Output keys

void clear() {

clearHelper(root);

}; // Clear tree

bool isEmpty() const {

if (root == NULL)

return true;

else

return false;

}; // Tree is empty

void showStructure() const {

if (root == 0)

cout << "Empty tree" << endl;

else

{

cout << endl;

showHelper(root, 1);

cout << endl;

}

};

int getHeight() const {

return getHeightHelper(root, 0);

}; // Height of tree

int getCount() const {

return getCountHelper(root);

}; // Number of nodes in tree

void writeLessThan(const KeyType& searchKey) const {

vector<KeyType> temp;

writeLessThanHelper(root, searchKey);

}; // Output keys < searchKey

protected:

class BSTreeNode

{

public:

// Constructor

BSTreeNode(const DataType& newDataItem, BSTreeNode\* leftPtr, BSTreeNode\* rightPtr) {

dataItem = newDataItem;

left = leftPtr;

right = rightPtr;

};

DataType dataItem; // Binary search tree data item

BSTreeNode\* left, \* right;

};

void showHelper(BSTreeNode\* p, int level) const;

void insertHelper(BSTreeNode\*& p, const DataType& newDataItem);

bool retrieveHelper(BSTreeNode\* p, const KeyType& searchKey, DataType& searchDataItem) const;

bool removeHelper(BSTreeNode\*& p, const KeyType& deleteKey);

void writeKeysHelper(BSTreeNode\* p) const;

void clearHelper(BSTreeNode\*& p);

int getHeightHelper(BSTreeNode\* p, int level) const;

int getCountHelper(BSTreeNode\* p) const;

void writeLessThanHelper(BSTreeNode\* p, const KeyType& searchKey) const;

void copyHelper(BSTreeNode\* p, BSTreeNode\* other);

BSTreeNode\* root; // Pointer to the root node

};

template < typename DataType, typename KeyType >

void BSTree<DataType, KeyType>::showHelper(BSTreeNode\* p, int level) const

{

int j; // Loop counter

if (p != 0)

{

showHelper(p->right, level + 1); // Output right subtree

for (j = 0; j < level; j++) // Tab over to level

cout << "\t";

cout << " " << (p->dataItem).getKey(); // Output key

if ((p->left != 0) && // Output "connector"

(p->right != 0))

cout << "<";

else if (p->right != 0)

cout << "/";

else if (p->left != 0)

cout << "\\";

cout << endl;

showHelper(p->left, level + 1); // Output left subtree

}

} //feature taken from show.cpp

template < typename DataType, typename KeyType >

void BSTree<DataType, KeyType>::insertHelper(BSTreeNode\*& p, const DataType& newDataItem) {

if (p == 0) p = new BSTreeNode(newDataItem, 0, 0);

else if (newDataItem.getKey() < p->dataItem.getKey())

insertHelper(p->left, newDataItem);

else if (newDataItem.getKey() > p->dataItem.getKey())

insertHelper(p->right, newDataItem);

else

p->dataItem = newDataItem;

};//feature complete

template < typename DataType, typename KeyType >

bool BSTree<DataType, KeyType>::retrieveHelper(BSTreeNode\* p, const KeyType& searchKey, DataType& searchDataItem) const {

bool result = false; // Result returned

if (p == 0) {// Fell off the tree while searching. Item is not in tree.

result = false;

return result;

}

else if (searchKey < p->dataItem.getKey()) {// Key is smaller than current node. Search to left.

retrieveHelper(p->left, searchKey, searchDataItem);

result = true;

return result;

}

else if (searchKey > p->dataItem.getKey()) { // Key is larger than current node. Search to right.

retrieveHelper(p->right, searchKey, searchDataItem);

result = true;

return result;

}

else {

return result;

}

};//in testing

template<typename DataType, typename KeyType>

bool BSTree<DataType, KeyType>::removeHelper(BSTreeNode\*& p, const KeyType& deleteKey) {

BSTreeNode\* delPtr;

int result;

if (p == 0) {

return false;

}

else if (deleteKey < p->dataItem.getKey()) {

removeHelper(p->left, deleteKey);

}

else if (deleteKey > p->dataItem.getKey()) {

removeHelper(p->right, deleteKey);

}

else {

delPtr = p;

if (p->left == 0) {

p = p->right;

delete delPtr;

}

else if (p->right == 0) {

p = p->left;

delete delPtr;

}

else {

BSTreeNode\* temp = p->left;

while (temp->right) {

temp = temp->right;

};

p->dataItem = temp->dataItem;

removeHelper(p->left, temp->dataItem.getKey());

};

};

return true;

};

template<typename DataType, typename KeyType>

void BSTree<DataType, KeyType>::writeKeysHelper(BSTreeNode\* p) const {

if (p == 0) {

return;

}

else {

writeKeysHelper(p->left);

cout << (p->dataItem).getKey() << " ";

writeKeysHelper(p->right);

}

};

template<typename DataType, typename KeyType>

void BSTree<DataType, KeyType>::clearHelper(BSTreeNode\*& p) {

if (p == 0) {

return;

}

else {

clearHelper(p->left);

clearHelper(p->right);

delete p;

}

}

template<typename DataType, typename KeyType>

int BSTree<DataType, KeyType>::getHeightHelper(BSTreeNode\* p, int level) const {

static int maxLevel = 0;

if (p == 0) {

if (maxLevel < level)

maxLevel = level;

}

else {

getHeightHelper(p->left, level + 1);

getHeightHelper(p->right, level + 1);

}

return maxLevel;

};

template<typename DataType, typename KeyType>

int BSTree<DataType, KeyType>::getCountHelper(BSTreeNode\* p) const {

int temp;

if (p == 0)

return 0;

else {

temp = 1 + getCountHelper(p->left) + getCountHelper(p->right);

return temp;

}

};

template<typename DataType, typename KeyType>

void BSTree<DataType, KeyType>::writeLessThanHelper(BSTreeNode\* p, const KeyType& searchKey) const {

if (p == 0)

return;

else {

writeLessThanHelper(p->left, searchKey);

if ((p->dataItem).getKey() < searchKey)

cout << (p->dataItem).getKey() << ", ";

}

};

template<typename DataType, typename KeyType>

void BSTree<DataType, KeyType>::copyHelper(BSTreeNode\* p, BSTreeNode\* other) {

if(other != NULL){

p = new BSTreeNode( other->dataItem, 0, 0);

copyHelper( p->left, other->left );

copyHelper( p->right, other->right );

}

root = p;

};

#endif // !BST\_ADT\_H

