// Yousef Zoumot

// main.cpp

// Coen70HW7.1 Chapter 10 Problem#8

//

// Created by Yousef Zoumot on 3/13/16.

// Copyright (c) 2016 Yousef Zoumot. All rights reserved.

//

#include <iostream>

#include <math.h>

#include <list>

using namespace std;

template <class T>

struct Node {

T value;

Node \*left;

Node \*right;

Node(T val) {

this->value = val;

}

Node(T val, Node<T> left, Node<T> right) {

this->value = val;

this->left = &left;

this->right = &right;

}

};

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

class BinaryTree {

private:

Node<T> \*root;

void addRecursive(Node<T> \*root, T val);

void printRecursive(Node<T> \*root);

int nodesCountRecursive(Node<T> \*root);

int heightRecursive(Node<T> \*root);

bool deleteValueRecursive(Node<T>\* parent, Node<T>\* current, T value);

public:

Node<T>\* getRoot(){return root;};

void add(T val);

void print();

int nodesCount();

int height();

bool deleteValue(T value);

};

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template<class T>

class LinkedList: public BinaryTree<T>{

list<T> head;

public:

LinkedList(BinaryTree<T>);

void linkedListRecursion(Node<T> \*root);

void pushBack(T);

void printLinkedList();

};

template<class T>

void LinkedList<T>:: pushBack(T val){

head.push\_back(val);

}

template<class T>

LinkedList<T>::LinkedList(BinaryTree<T> bT){

linkedListRecursion(bT.getRoot());

}

template<class T>

void LinkedList<T>:: linkedListRecursion(Node<T> \*root){

if(!root){

return;

}

linkedListRecursion(root->left);

pushBack(root->value);

linkedListRecursion(root->right);

}

template<class T>

void LinkedList<T>:: printLinkedList(){

typename list<T>:: iterator it;

for(it=head.begin(); it!= head.end(); it++){

cout<< \*it <<" ";

}

cout<<"\n";

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

void BinaryTree<T>:: addRecursive(Node<T> \*root, T val) {

if (root->value > val) {

if (!root->left) {

root->left = new Node<T>(val);

} else {

addRecursive(root->left, val);

}

} else {

if (!root->right) {

root->right = new Node<T>(val);

} else {

addRecursive(root->right, val);

}

}

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

void BinaryTree<T>:: printRecursive(Node<T> \*root) {

if (!root) return;

printRecursive(root->left);

cout<<root->value<<' ';

printRecursive(root->right);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

int BinaryTree<T>:: nodesCountRecursive(Node<T> \*root) {

if (!root) return 0;

else return 1 + nodesCountRecursive(root->left) + nodesCountRecursive(root->right);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

int BinaryTree<T>:: heightRecursive(Node<T> \*root) {

if (!root) return 0;

else return 1 + max(heightRecursive(root->left), heightRecursive(root->right));

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

bool BinaryTree<T>:: deleteValueRecursive(Node<T>\* parent, Node<T>\* current, T value) {

if (!current) return false;

if (current->value == value) {

if (current->left == NULL || current->right == NULL) {

Node<T>\* temp = current->left;

if (current->right) temp = current->right;

if (parent) {

if (parent->left == current) {

parent->left = temp;

} else {

parent->right = temp;

}

} else {

this->root = temp;

}

} else {

Node<T>\* substitute = current->right;

while (substitute->left) {

substitute = substitute->left;

}

T temp = current->value;

current->value = substitute->value;

substitute->value = temp;

return deleteValueRecursive(current, current->right, temp);

}

delete current;

return true;

}

return deleteValueRecursive(current, current->left, value) ||

deleteValueRecursive(current, current->right, value);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

void BinaryTree<T>:: add(T val) {

if (root) {

this->addRecursive(root, val);

} else {

root = new Node<T>(val);

}

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

void BinaryTree<T>:: print() {

printRecursive(this->root);

cout<<"\n";

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

int BinaryTree<T>:: nodesCount() {

return nodesCountRecursive(root);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

int BinaryTree<T>:: height() {

return heightRecursive(this->root);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

template <class T>

bool BinaryTree<T>:: deleteValue(T value) {

return this->deleteValueRecursive(NULL, this->root, value);

}

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////

int main(int argc, const char \* argv[]) {

// insert code here...

BinaryTree<int> \*bst1=new BinaryTree<int>();

bst1->add(5);

bst1->add(4);

bst1->add(7);

bst1->add(2);

bst1->add(9);

bst1->add(8);

bst1->print();

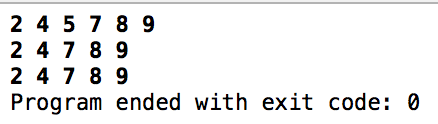
bst1->deleteValue(5);

bst1->print();

LinkedList<int> \*linkedL= new LinkedList<int>(\*bst1);

linkedL->printLinkedList();

return 0;

}

// Yousef Zoumot

// main.cpp

// Coen70HW7.2 Chapter 10 Problem 9

//

// Created by Yousef Zoumot on 3/13/16.

// Copyright (c) 2016 Yousef Zoumot. All rights reserved.

//

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#include <list>

//#define pow2(n) (1 << (n))

using namespace std;

struct avl\_Node

{

int data;

struct avl\_Node \*left;

struct avl\_Node \*right;

};

class avlTree

{

public:

avlTree()

{

root = NULL;

}

avlTree(list<int>);

int height(){return heightRecursive(root);};

void insert(int value){root=insertRecursive(root , value);};

void remove(int value){root=removeRecursive(root, value);};

void display();

void inOrder();

void preOrder();

void postOrder();

private:

avl\_Node \*root;

int heightRecursive(avl\_Node \*);

int heightDifferenceRecursive(avl\_Node \*);

avl\_Node \*rightright\_rotationRecursive(avl\_Node \*);

avl\_Node \*leftleft\_rotationRecursive(avl\_Node \*);

avl\_Node \*leftright\_rotationRecursive(avl\_Node \*);

avl\_Node \*rightleft\_rotationRecursive(avl\_Node \*);

avl\_Node\* balanceRecursive(avl\_Node \*);

avl\_Node\* insertRecursive(avl\_Node \*, int );

avl\_Node\* removeRecursive(avl\_Node \*, int );

void displayRecursive(avl\_Node \*, int);

void inOrderRecursive(avl\_Node \*);

void preOrderRecursive(avl\_Node \*);

void postOrderRecursive(avl\_Node \*);

avl\_Node\* minValueNode(avl\_Node\* node);

};

avlTree::avlTree(list<int> source){

root=NULL;

typename list<int>::iterator it;

for(it=source.begin(); it!=source.end(); it++){

insert(\*it);

}

}

void avlTree:: display(){

if(root ==NULL)

cout<<"This AVL Tree is empty"<< "\n";

else{

displayRecursive(root, 1);

}

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

}

void avlTree:: preOrder(){

preOrderRecursive(root);

}

void avlTree:: inOrder(){

inOrderRecursive(root);

}

void avlTree:: postOrder(){

postOrderRecursive(root);

}

avl\_Node\* avlTree:: minValueNode(avl\_Node\* node)

{

avl\_Node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

current = current->left;

return current;

}

//\* Height of AVL Tree

int avlTree::heightRecursive(avl\_Node \*temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = heightRecursive (temp->left);

int r\_height = heightRecursive (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

// \* Height Difference

int avlTree::heightDifferenceRecursive(avl\_Node \*temp)

{

int l\_height = heightRecursive (temp->left);

int r\_height = heightRecursive (temp->right);

int b\_factor= l\_height - r\_height;

return b\_factor;

}

// Right- Right rotationRecursive

avl\_Node \*avlTree::rightright\_rotationRecursive(avl\_Node \*parent)

{

avl\_Node \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

// Left- Left rotationRecursive

avl\_Node \*avlTree::leftleft\_rotationRecursive(avl\_Node \*parent)

{

avl\_Node \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

// Left - Right rotationRecursive

avl\_Node \*avlTree::leftright\_rotationRecursive(avl\_Node \*parent)

{

avl\_Node \*temp;

temp = parent->left;

parent->left = rightright\_rotationRecursive (temp);

return leftleft\_rotationRecursive (parent);

}

// Right- Left rotationRecursive

avl\_Node \*avlTree::rightleft\_rotationRecursive(avl\_Node \*parent)

{

avl\_Node \*temp;

temp = parent->right;

parent->right = leftleft\_rotationRecursive (temp);

return rightright\_rotationRecursive (parent);

}

// Balancing AVL Tree

avl\_Node \*avlTree::balanceRecursive(avl\_Node \*temp)

{

int bal\_factor = heightDifferenceRecursive (temp);

if (bal\_factor > 1)

{

if (heightDifferenceRecursive (temp->left) > 0)

temp = leftleft\_rotationRecursive (temp);

else

temp = leftright\_rotationRecursive (temp);

}

else if (bal\_factor < -1)

{

if (heightDifferenceRecursive (temp->right) > 0)

temp = rightleft\_rotationRecursive (temp);

else

temp = rightright\_rotationRecursive (temp);

}

return temp;

}

//insertRecursive Element into the tree

avl\_Node \*avlTree::insertRecursive(avl\_Node \*root, int value)

{

if (root == NULL)

{

root = new avl\_Node;

root->data = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->data)

{

root->left = insertRecursive(root->left, value);

root = balanceRecursive (root);

}

else if (value >= root->data)

{

root->right = insertRecursive(root->right, value);

root = balanceRecursive (root);

}

return root;

}

//removes element from tree

avl\_Node \*avlTree:: removeRecursive(avl\_Node \*root, int key)

{

// PERFORM STANDARD BST DELETE

if (root == NULL)

return root;

// If the key to be deleted is smaller than the root's key,

// then it lies in left subtree

if ( key < root->data )

root->left = removeRecursive(root->left, key);

// If the key to be deleted is greater than the root's key,

// then it lies in right subtree

else if( key > root->data )

root->right = removeRecursive(root->right, key);

// if key is same as root's key, then This is the node

// to be deleted

else

{

// node with only one child or no child

if( (root->left == NULL) || (root->right == NULL) )

{

avl\_Node\* temp = root->left ? root->left : root->right;

// No child case

if(temp == NULL)

{

temp = root;

root = NULL;

}

else // One child case

\*root = \*temp; // Copy the contents of the non-empty child

delete temp;

}

else

{

// node with two children: Get the inorder successor (smallest

// in the right subtree)

avl\_Node\* temp = minValueNode(root->right);

// Copy the inorder successor's data to this node

root->data = temp->data;

// Delete the inorder successor

root->right = removeRecursive(root->right, temp->data);

}

}

// If the tree had only one node

if (root == NULL)

return root;

// GET THE BALANCE FACTOR OF THIS NODE (to check whether

// this node became unbalanced)

int balance = heightDifferenceRecursive(root);

// If unbalanced, there are 4 cases

// Left Left Case

if (balance > 1 && heightDifferenceRecursive(root->left) >= 0)

return leftleft\_rotationRecursive(root);

// Left Right Case

if (balance > 1 && heightDifferenceRecursive(root->left) < 0)

{

return leftright\_rotationRecursive(root);

}

// Right Right Case

if (balance < -1 && heightDifferenceRecursive(root->right) <= 0)

return rightright\_rotationRecursive(root);

// Right Left Case

if (balance < -1 && heightDifferenceRecursive(root->right) > 0)

{

return rightleft\_rotationRecursive(root);

}

return root;

}

//displayRecursive AVL Tree

void avlTree::displayRecursive(avl\_Node \*ptr, int level)

{

int i;

if (ptr!=NULL)

{

displayRecursive(ptr->right, level + 1);

printf("\n");

if (ptr == root)

cout<<"Root -> ";

for (i = 0; i < level && ptr != root; i++)

cout<<" ";

cout<<ptr->data;

displayRecursive(ptr->left, level + 1);

}

}

//inOrderRecursive Traversal of AVL Tree

void avlTree::inOrderRecursive(avl\_Node \*tree)

{

if (tree == NULL)

return;

inOrderRecursive (tree->left);

cout<<tree->data<<" ";

inOrderRecursive (tree->right);

}

// preOrderRecursive Traversal of AVL Tree

void avlTree::preOrderRecursive(avl\_Node \*tree)

{

if (tree == NULL)

return;

cout<<tree->data<<" ";

preOrderRecursive (tree->left);

preOrderRecursive (tree->right);

}

// postOrderRecursive Traversal of AVL Tree

void avlTree::postOrderRecursive(avl\_Node \*tree)

{

if (tree == NULL)

return;

postOrderRecursive ( tree ->left );

postOrderRecursive ( tree ->right );

cout<<tree->data<<" ";

}

int main(int argc, const char \* argv[]) {

avlTree avlt1 = avlTree();

avlt1.insert(5);

avlt1.insert(4);

avlt1.insert(3);

avlt1.insert(2);

avlt1.insert(1);

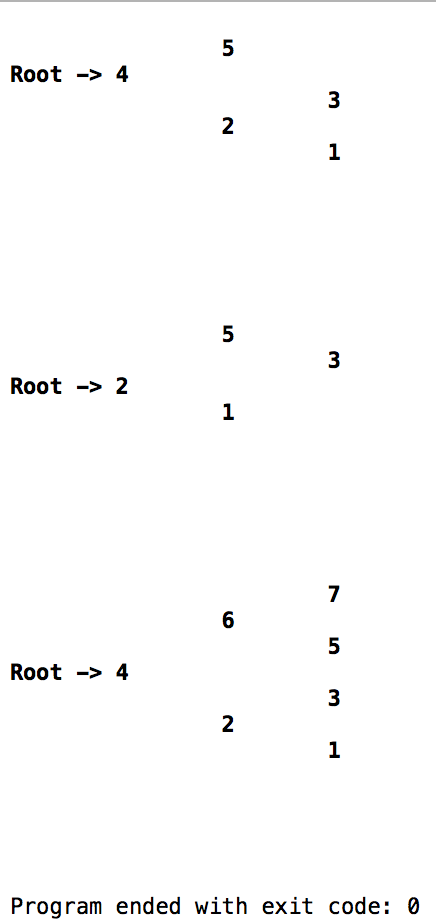
avlt1.display();

avlt1.remove(4);

avlt1.display();

list<int> myList;

myList.push\_back(1);

 myList.push\_back(2);

myList.push\_back(3);

myList.push\_back(4);

myList.push\_back(5);

myList.push\_back(6);

myList.push\_back(7);

avlTree avl2=avlTree(myList);

avl2.display();

}

// Yousef Zoumot

// main.cpp

// Coen70HW7.2 Chapter 10 Problem 12

//

// Created by Yousef Zoumot on 3/13/16.

// Copyright (c) 2016 Yousef Zoumot. All rights reserved.

//

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#include <list>

#include <vector>

#include <string>

#include <cstring>

#include "string.h"

using namespace std;

class Book

{

public:

struct BookNode

{

BookNode(string nam, string auth, int isbn, string dat);

string name;

string author;

int iSBN;

string date;

struct BookNode \*left;

struct BookNode \*right;

};

Book()

{

root = NULL;

}

Book(list<int>);

int height(){return heightRecursive(root);};

void insert(BookNode source){root=insertRecursive(root ,source);};

void display();

private:

BookNode \*root;

int heightRecursive(BookNode \*);

int heightDifferenceRecursive(BookNode \*);

BookNode \*rightright\_rotationRecursive(BookNode \*);

BookNode \*leftleft\_rotationRecursive(BookNode \*);

BookNode \*leftright\_rotationRecursive(BookNode \*);

BookNode \*rightleft\_rotationRecursive(BookNode \*);

BookNode\* balanceRecursive(BookNode \*);

BookNode\* insertRecursive(BookNode \*, BookNode source);

BookNode\* removeRecursive(BookNode \*, int );

void displayRecursive(BookNode \*, int);

BookNode\* minValueNode(BookNode\* node);

};

bool lowercase(char c1, char c2){

return tolower(c1) < tolower(c2);

}

Book::BookNode::BookNode(string nam, string auth, int isbn, string dat){

name=nam;

author=auth;

iSBN=isbn;

date=dat;

}

void Book:: display(){

if(root ==NULL)

cout<<"This AVL Tree is empty"<< "\n";

else{

displayRecursive(root, 1);

}

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

cout<<"\n";

}

void Book:: preOrder(){

preOrderRecursive(root);

}

void Book:: inOrder(){

inOrderRecursive(root);

}

void Book:: postOrder(){

postOrderRecursive(root);

}

Book::BookNode\* Book:: minValueNode(BookNode\* node)

{

BookNode\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

current = current->left;

return current;

}

//\* Height of AVL Tree

int Book::heightRecursive(BookNode \*temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = heightRecursive (temp->left);

int r\_height = heightRecursive (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

// \* Height Difference

int Book::heightDifferenceRecursive(BookNode \*temp)

{

int l\_height = heightRecursive (temp->left);

int r\_height = heightRecursive (temp->right);

int b\_factor= l\_height - r\_height;

return b\_factor;

}

// Right- Right rotationRecursive

Book::BookNode \*Book::rightright\_rotationRecursive(BookNode \*parent)

{

BookNode \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

// Left- Left rotationRecursive

Book::BookNode \*Book::leftleft\_rotationRecursive(BookNode \*parent)

{

BookNode \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

// Left - Right rotationRecursive

Book::BookNode \*Book::leftright\_rotationRecursive(BookNode \*parent)

{

BookNode \*temp;

temp = parent->left;

parent->left = rightright\_rotationRecursive (temp);

return leftleft\_rotationRecursive (parent);

}

// Right- Left rotationRecursive

Book::BookNode \*Book::rightleft\_rotationRecursive(BookNode \*parent)

{

BookNode \*temp;

temp = parent->right;

parent->right = leftleft\_rotationRecursive (temp);

return rightright\_rotationRecursive (parent);

}

// Balancing AVL Tree

Book::BookNode \*Book::balanceRecursive(BookNode \*temp)

{

int bal\_factor = heightDifferenceRecursive (temp);

if (bal\_factor > 1)

{

if (heightDifferenceRecursive (temp->left) > 0)

temp = leftleft\_rotationRecursive (temp);

else

temp = leftright\_rotationRecursive (temp);

}

else if (bal\_factor < -1)

{

if (heightDifferenceRecursive (temp->right) > 0)

temp = rightleft\_rotationRecursive (temp);

else

temp = rightright\_rotationRecursive (temp);

}

return temp;

}

//insertRecursive Element into the tree

Book::BookNode \*Book::insertRecursive(BookNode \*root, BookNode source)

{

if (root == NULL)

{

root = new BookNode(source.name, source.author, source.iSBN, source.date);

root->left = NULL;

root->right = NULL;

return root;

}

string temp1=source.author;

string temp2=root->author;

for(int i=0; i<temp1.size(); i++){

temp1[i]=tolower(temp1[i]);

}

for(int i=0; i<temp2.size(); i++){

temp2[i]=tolower(temp2[i]);

}

if( temp1 < temp2)

{

root->left = insertRecursive(root->left, source);

root = balanceRecursive (root);

}

else if (temp1 >= temp2)

{

root->right = insertRecursive(root->right, source);

root = balanceRecursive (root);

}

return root;

}

//displayRecursive AVL Tree

void Book::displayRecursive(BookNode \*ptr, int level)

{

int i;

if (ptr!=NULL)

{

displayRecursive(ptr->right, level + 1);

printf("\n");

if (ptr == root)

cout<<"Root -> ";

for (i = 0; i < level && ptr != root; i++)

cout<<" ";

cout<<ptr->author;

displayRecursive(ptr->left, level + 1);

}

}

int main(int argc, const char \* argv[]) {

Book::BookNode \*b1= new Book::BookNode("name", "author1", 1234, "3/6/2001");

Book::BookNode \*b2= new Book::BookNode("name", "author2", 123456, "3/6/2001");

Book::BookNode \*b3= new Book::BookNode("name", "author3", 12345, "3/6/2001");

Book::BookNode \*b4= new Book::BookNode("name", "author4", 12346, "3/6/2001");

Book::BookNode \*b5= new Book::BookNode("name", "author5", 12356, "3/6/2001");

Book::BookNode \*b6= new Book::BookNode("name", "author6", 12456, "3/6/2001");

Book::BookNode \*b7= new Book::BookNode("name", "author7", 13456, "3/6/2001");

Book::BookNode \*b8= new Book::BookNode("name", "author4", 23456, "3/6/2001");

Book avlt1 = Book();

avlt1.insert(\*b1);

avlt1.insert(\*b2);

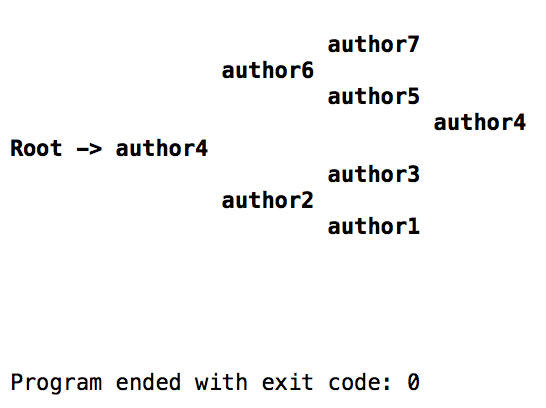
avlt1.insert(\*b3);

avlt1.insert(\*b4);

avlt1.insert(\*b5);

avlt1.insert(\*b6);

avlt1.insert(\*b7);

 avlt1.insert(\*b8);

avlt1.display();

}