//AVl Tree implementation

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

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

using namespace std;

struct avl {

int d;

struct avl \*l;

struct avl \*r;

}\*r;

class avl\_tree {

public:

int height(avl \*);

int difference(avl \*);

avl \*rr\_rotat(avl \*);

avl \*ll\_rotat(avl \*);

avl \*lr\_rotat(avl\*);

avl \*rl\_rotat(avl \*);

avl \* balance(avl \*);

avl \* insert(avl\*, int);

void show(avl\*, int);

void inorder(avl \*);

void preorder(avl \*);

void postorder(avl\*);

avl\_tree() {

r = NULL;

}

};

int avl\_tree::height(avl \*t) {

int h = 0;

if (t != NULL) {

int l\_height = height(t->l);

int r\_height = height(t->r);

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

h = max\_height + 1;

}

return h;

}

int avl\_tree::difference(avl \*t) {

int l\_height = height(t->l);

int r\_height = height(t->r);

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

return b\_factor;

}

avl \*avl\_tree::rr\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = t->l;

t->l = parent;

cout<<"Right-Right Rotation";

return t;

}

avl \*avl\_tree::ll\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = t->r;

t->r = parent;

cout<<"Left-Left Rotation";

return t;

}

avl \*avl\_tree::lr\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = rr\_rotat(t);

cout<<"Left-Right Rotation";

return ll\_rotat(parent);

}

avl \*avl\_tree::rl\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = ll\_rotat(t);

cout<<"Right-Left Rotation";

return rr\_rotat(parent);

}

avl \*avl\_tree::balance(avl \*t) {

int bal\_factor = difference(t);

if (bal\_factor > 1) {

if (difference(t->l) > 0)

t = ll\_rotat(t);

else

t = lr\_rotat(t);

} else if (bal\_factor < -1) {

if (difference(t->r) > 0)

t = rl\_rotat(t);

else

t = rr\_rotat(t);

}

return t;

}

avl \*avl\_tree::insert(avl \*r, int v) {

if (r == NULL) {

r = new avl;

r->d = v;

r->l = NULL;

r->r = NULL;

return r;

} else if (v< r->d) {

r->l = insert(r->l, v);

r = balance(r);

} else if (v >= r->d) {

r->r = insert(r->r, v);

r = balance(r);

} return r;

}

void avl\_tree::show(avl \*p, int l) {

int i;

if (p != NULL) {

show(p->r, l+ 1);

cout<<" ";

if (p == r)

cout << "Root -> ";

for (i = 0; i < l&& p != r; i++)

cout << " ";

cout << p->d;

show(p->l, l + 1);

}

}

void avl\_tree::inorder(avl \*t) {

if (t == NULL)

return;

inorder(t->l);

cout << t->d << " ";

inorder(t->r);

}

void avl\_tree::preorder(avl \*t) {

if (t == NULL)

return;

cout << t->d << " ";

preorder(t->l);

preorder(t->r);

}

void avl\_tree::postorder(avl \*t) {

if (t == NULL)

return;

postorder(t ->l);

postorder(t ->r);

cout << t->d << " ";

}

int main() {

int c, i;

avl\_tree avl;

while (1) {

cout << "1.Insert Element into the tree" << endl;

cout << "2.show Balanced AVL Tree" << endl;

cout << "3.InOrder traversal" << endl;

cout << "4.PreOrder traversal" << endl;

cout << "5.PostOrder traversal" << endl;

cout << "6.Exit" << endl;

cout << "Enter your Choice: ";

cin >> c;

switch (c) {

case 1:

cout << "Enter value to be inserted: ";

cin >> i;

r = avl.insert(r, i);

break;

case 2:

if (r == NULL) {

cout << "Tree is Empty" << endl;

continue;

}

cout << "Balanced AVL Tree:" << endl;

avl.show(r, 1);

cout<<endl;

break;

case 3:

cout << "Inorder Traversal:" << endl;

avl.inorder(r);

cout << endl;

break;

case 4:

cout << "Preorder Traversal:" << endl;

avl.preorder(r);

cout << endl;

break;

case 5:

cout << "Postorder Traversal:" << endl;

avl.postorder(r);

cout << endl;

break;

case 6:

exit(1);

break;

default:

cout << "Wrong Choice" << endl;

}

}

return 0;

}