**/\*WAP to implement AVL Tree.\*/**

**#include<iostream>**

**#include <algorithm>**

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

**using namespace std;**

**/\***

**\* Node Declaration**

**\*/**

**struct avl\_node**

**{**

**int data;**

**struct avl\_node \*left;**

**struct avl\_node \*right;**

**}\*root;**

**/\***

**\* Class Declaration**

**\*/**

**class avlTree**

**{**

**public:**

**int height(avl\_node \*);**

**int diff(avl\_node \*);**

**avl\_node \*rr\_rotation(avl\_node \*);**

**avl\_node \*ll\_rotation(avl\_node \*);**

**avl\_node \*lr\_rotation(avl\_node \*);**

**avl\_node \*rl\_rotation(avl\_node \*);**

**avl\_node\* balance(avl\_node \*);**

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

**void display(avl\_node \*, int);**

**void inorder(avl\_node \*);**

**void preorder(avl\_node \*);**

**void postorder(avl\_node \*);**

**avl\_node\* remove(avl\_node\* t, int x);**

**avl\_node\* findMin(avl\_node\*);**

**avl\_node\* findMax(avl\_node\*);**

**avlTree()**

**{**

**root = NULL;**

**}**

**};**

**/\***

**\* Main Contains Menu**

**\*/**

**int main()**

**{**

**int choice, item;**

**avlTree avl;**

**while (1)**

**{**

**cout << "\n---------------------" << endl;**

**cout << "AVL Tree Implementation" << endl;**

**cout << "\n---------------------" << endl;**

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

**cout << "2.Delete Element into the tree" << endl;**

**cout << "3.Display Balanced AVL Tree" << endl;**

**cout << "4.InOrder traversal" << endl;**

**cout << "5.PreOrder traversal" << endl;**

**cout << "6.PostOrder traversal" << endl;**

**cout << "7.Exit" << endl;**

**cout << "Enter your Choice: ";**

**cin >> choice;**

**switch (choice)**

**{**

**case 1:**

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

**cin >> item;**

**root = avl.insert(root, item);**

**break;**

**case 2:**

**cout << "Enter value to be deleted: ";**

**cin >> item;**

**root = avl.remove(root, item);**

**break;**

**case 3:**

**if (root == NULL)**

**{**

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

**continue;**

**}**

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

**avl.display(root, 1);**

**break;**

**case 4:**

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

**avl.inorder(root);**

**cout << endl;**

**break;**

**case 5:**

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

**avl.preorder(root);**

**cout << endl;**

**break;**

**case 6:**

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

**avl.postorder(root);**

**cout << endl;**

**break;**

**case 7:**

**exit(1);**

**break;**

**default:**

**cout << "Wrong Choice" << endl;**

**}**

**}**

**return 0;**

**}**

**/\***

**\* Height of AVL Tree**

**\*/**

**int avlTree::height(avl\_node \*temp)**

**{**

**int h = 0;**

**if (temp != NULL)**

**{**

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

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

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

**h = max\_height + 1;**

**}**

**return h;**

**}**

**/\***

**\* Height Difference**

**\*/**

**int avlTree::diff(avl\_node \*temp)**

**{**

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

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

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

**return b\_factor;**

**}**

**/\***

**\* Right- Right Rotation**

**\*/**

**avl\_node \*avlTree::rr\_rotation(avl\_node \*parent)**

**{**

**avl\_node \*temp;**

**temp = parent->right;**

**parent->right = temp->left;**

**temp->left = parent;**

**return temp;**

**}**

**/\***

**\* Left- Left Rotation**

**\*/**

**avl\_node \*avlTree::ll\_rotation(avl\_node \*parent)**

**{**

**avl\_node \*temp;**

**temp = parent->left;**

**parent->left = temp->right;**

**temp->right = parent;**

**return temp;**

**}**

**/\***

**\* Left - Right Rotation**

**\*/**

**avl\_node \*avlTree::lr\_rotation(avl\_node \*parent)**

**{**

**avl\_node \*temp;**

**temp = parent->left;**

**parent->left = rr\_rotation(temp);**

**return ll\_rotation(parent);**

**}**

**/\***

**\* Right- Left Rotation**

**\*/**

**avl\_node \*avlTree::rl\_rotation(avl\_node \*parent)**

**{**

**avl\_node \*temp;**

**temp = parent->right;**

**parent->right = ll\_rotation(temp);**

**return rr\_rotation(parent);**

**}**

**/\***

**\* Balancing AVL Tree**

**\*/**

**avl\_node \*avlTree::balance(avl\_node \*temp)**

**{**

**int bal\_factor = diff(temp);**

**if (bal\_factor > 1)**

**{**

**if (diff(temp->left) > 0)**

**temp = ll\_rotation(temp);**

**else**

**temp = lr\_rotation(temp);**

**}**

**else if (bal\_factor < -1)**

**{**

**if (diff(temp->right) > 0)**

**temp = rl\_rotation(temp);**

**else**

**temp = rr\_rotation(temp);**

**}**

**return temp;**

**}**

**/\***

**\* Insert Element into the tree**

**\*/**

**avl\_node \*avlTree::insert(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 = insert(root->left, value);**

**root = balance(root);**

**}**

**else if (value >= root->data)**

**{**

**root->right = insert(root->right, value);**

**root = balance(root);**

**}**

**return root;**

**}**

**/\***

**\* Display AVL Tree**

**\*/**

**void avlTree::display(avl\_node \*ptr, int level)**

**{**

**int i;**

**if (ptr != NULL)**

**{**

**display(ptr->right, level + 1);**

**printf("\n");**

**if (ptr == root)**

**cout << "Root -> ";**

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

**cout << " ";**

**cout << ptr->data;**

**display(ptr->left, level + 1);**

**}**

**}**

**/\***

**\* Inorder Traversal of AVL Tree**

**\*/**

**void avlTree::inorder(avl\_node \*tree)**

**{**

**if (tree == NULL)**

**return;**

**inorder(tree->left);**

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

**inorder(tree->right);**

**}**

**/\***

**\* Preorder Traversal of AVL Tree**

**\*/**

**void avlTree::preorder(avl\_node \*tree)**

**{**

**if (tree == NULL)**

**return;**

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

**preorder(tree->left);**

**preorder(tree->right);**

**}**

**avl\_node\* avlTree::findMin(avl\_node\* t)**

**{**

**if (t == NULL) return NULL;**

**else if (t->left == NULL) return t; // if element traverse on max left then return**

**else return findMin(t->left); // or recursively traverse max left**

**}**

**avl\_node\* avlTree:: findMax(avl\_node\* t)**

**{**

**if (t == NULL) return NULL;**

**else if (t->right == NULL) return t;**

**else return findMax(t->right);**

**}**

**/\***

**\* Postorder Traversal of AVL Tree**

**\*/**

**void avlTree::postorder(avl\_node \*tree)**

**{**

**if (tree == NULL)**

**return;**

**postorder(tree->left);**

**postorder(tree->right);**

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

**}**

**avl\_node\* avlTree:: remove(avl\_node\* t, int x)**

**{**

**avl\_node\* temp;**

**// element not found**

**if (t == NULL) return NULL;**

**// searching element**

**else if (x < t->data) t->left = remove(t->left, x);**

**else if (x >t->data) t->right = remove(t->right, x);**

**// element found**

**// element has 2 children**

**else if (t->left && t->right)**

**{**

**temp = findMin(t->right);**

**t->data = temp->data;**

**t->right = remove(t->right, t->data);**

**}**

**// if element has 1 or 0 child**

**else**

**{**

**temp = t;**

**if (t->left == NULL) t = t->right;**

**else if (t->right == NULL) t = t->left;**

**delete temp;**

**}**

**if (t == NULL) return t;**

**// check balanced)**

**t = balance(t);**

**}**