Assignment

Name:Muhammad Shaheer

Roll no:20p-0480

Data Structures

Q1: Write a program to check if a given BST is AVL or not.

Code:

// AVL Tree Implementation..

#include<iostream>

using namespace std;

    class node {

    public:

    int data;

    node\* left, \*right;

    int height;

    node(int data){

        this->data=  data;

        left= right= NULL;

        height=0;

        }

    };

class AVLTree{

    private:

    //private data member and methods

    node\* root;

    node\* Insert( node\* root, int val);

    node\* Delete(node\* root,int data);

    node\* PreOrderTraversal( node\* root);

    node\* FindMax(node\* root);

        // Rotation

    node\* singleRightRotate(node\* &t);

    node\* singleLeftRotate(node\* &t);

    node\* doubleRightLeftRotate(node\* &t);

    node\* doubleLeftRightRotate(node\* &t);

    int height(node \*N);

    bool isAVL(node\* root);

    int getBalance(node \*N);

    public:

    AVLTree(){

        root= NULL;

    }

    void    Insert(int val){

       root= Insert(this->root,  val);

   }

    void    Delete(int val){

    root = Delete(this->root, val);

   }

    void PreOrderTraversal(){

        PreOrderTraversal( this->root);

    }

    bool isAVL()

    {

        if(isAVL(root))

    cout << "Given tree is AVL" << endl;

else

    cout << "Given tree is not AVL" << endl;

    }

     int height(node\* t)

    {

        return (t == NULL ? -1 : t->height);

    }

    int length(node\* r){

        int  lh=0,rh=0;

        if(r==NULL)

         return -1;

        if(r->left!=NULL){

            lh++;

            lh+=length(r->left);

        }

        if(r->right!=NULL){

            rh++;

            rh+=length(r->right);

        }

        if(rh>lh) return rh;

        else return lh;

    }

    int treeHeight(node \*t)

    {

    int static l\_height=0;

    int static r\_height=0;

    if (t == NULL)

        return -1;

    else

    {

    l\_height = treeHeight(t->left);

       r\_height = treeHeight(t->right);

        if (l\_height > r\_height)

            return (l\_height + 1);

        else

            return (r\_height + 1);

    }

    }

};

int main (){

    AVLTree tree1;

      tree1.Insert(30);

    tree1.Insert(20);

    tree1.Insert(40);

    tree1.Insert(15);

    tree1.Delete(40);

    cout<<"Pre Order Print (Root--left--Right)"<<endl;

    tree1.PreOrderTraversal();

    tree1.isAVL();

    return 0;

}

node\* AVLTree::Insert(node\* t, int val ){

 if (t==NULL)

    {

         t= new node(val);

    }

    else if (t->data== val){

        cout<<"Record already exist"<<val<<endl;

    }

    else if (val < t->data) // insert on left s

    {

        t->left = Insert(t->left , val );

         int bf= height(t->left) - height(t->right);

            if(bf == 2)

            {

                if(val < t->left->data)

                    t = singleRightRotate(t);

                else

                    t = doubleLeftRightRotate(t);

            }

    }

    else if (val > t->data) // Right side

        {

        t->right= Insert( t->right,val);

        int bf=height(t->right) - height(t->left);

            if(bf == 2)

            {

                if(val > t->right->data)

                    t = singleLeftRotate(t);

                else

                    t = doubleRightLeftRotate(t);

            }

    }

    t->height = max(height(t->left), height(t->right))+1;

    return t;

}

node\* AVLTree::doubleRightLeftRotate(node\* &t)

    {

        t->right = singleRightRotate(t->right);

        return singleLeftRotate(t);

    }

node\* AVLTree::doubleLeftRightRotate(node\* &t)

    {

        t->left = singleLeftRotate(t->left);

        return singleRightRotate(t);

    }

node\* AVLTree::singleRightRotate(node\* &t)

    {

        node\* u = t->left;

        t->left = u->right;

        u->right = t;

        t->height = max(height(t->left), height(t->right))+1;

        u->height = max(height(u->left), t->height)+1;

        return u;

    }

node\* AVLTree::singleLeftRotate(node\* &t)

    {

        node\* u = t->right;

        t->right = u->left;

        u->left = t;

        t->height = max(height(t->left), height(t->right))+1;

        u->height = max(height(u->right), t->height)+1 ;

        return u;

    }

node \* AVLTree::Delete(node\* r, int data)

{

    if(r==NULL)

     return r;

    else if(data < r->data)

        r->left = Delete(r->left, data);

    else if (data> r->data)

        r->right = Delete(r->right, data);

    else

    {

        //No child

        if(r->right == NULL && r->left == NULL)

        {

            delete r;

            r = NULL;

            return r;

        }

        //One child on left

        else if(r->right == NULL)

        {

            node\* temp = r;

            r= r->left;

            delete temp;

        }

        //One child on right

        else if(r->left == NULL)

        {

            node\* temp = r;

            r= r->right;

            delete temp;

        }

        //two child

        else

        {

            node\* temp = FindMax(r->right);

            int x= r->data;  //

            r->data = temp->data;

            temp->data= x;

            r->left = Delete(r->left, temp->data);

        }

    }

     if(r == NULL)

            return r;

        r->height = max(height(r->left), height(r->right))+1;

        int bal = height(r->left) - height(r->right);

            if(bal>1){

                if(height(r->left) >= height(r->right)){

                    return singleRightRotate(r);

                }else{

                    r->left = singleLeftRotate(r->left);

                    return singleRightRotate(r);

                }

            }else if(bal < -1){

                if(height(r->right) >= height(r->left)){

                    return singleLeftRotate(r);

                }else{

                    r->right = singleRightRotate(r->right);

                    return singleLeftRotate(r);

                }

            }

        return r;

}

node\* AVLTree::FindMax(node\* r){

    while(r->right!=NULL){

        r= r->right;

    }

    return r;

}

node\* AVLTree::PreOrderTraversal( node\* r){

     if (r == NULL)

        return NULL;

    cout << " "<< r->data << " -> ";

    PreOrderTraversal(r->left);

    PreOrderTraversal(r->right);

}

int height(node \*N)

{

    if (N == NULL)

        return 0;

    return N->height;

}

int getBalance(node \*N)

{

    if (N == NULL)

        return 0;

    return height(N->left) - height(N->right);

}

bool isAVL(node\* root){

// An empty tree is AVL

if (root == NULL)

return true;

// Get the balance factor of root node

int balance = getBalance(root);

// If the balance factor is greater than 1 or less than -1, then it is not AVL

if (balance > 1 || balance < -1)

    return false;

// Recursively check if left and right subtrees are AVL

return isAVL(root->left) && isAVL(root->right);

}

Q2: Provide a C++ implementation of AVL tree must include.

● Recursive RR

● Recursive LL

● Recursive RL

● Recursive LR

● Apply on BST Insertion

● Finding Balancing Factor

● Display Nodes

● Test Your Code

Code:

// AVL Tree Implementation..

#include<iostream>

using namespace std;

    class node {

    public:

    int data;

    node\* left, \*right;

    int height;

    node(int data){

        this->data=  data;

        left= right= NULL;

        height=0;

        }

    };

class AVLTree{

    private:

    //private data member and methods

    node\* root;

    node\* Insert( node\* root, int val);

    node\* Delete(node\* root,int data);

    node\* PreOrderTraversal( node\* root);

    node\* FindMax(node\* root);

        // Rotation

    node\* singleRightRotate(node\* &t);

    node\* singleLeftRotate(node\* &t);

    node\* doubleRightLeftRotate(node\* &t);

    node\* doubleLeftRightRotate(node\* &t);

    int balance\_fact(node \*point);

    int path(node \*point);

    int getBalance(node\* t);

    public:

    AVLTree(){

        root= NULL;

    }

    void    Insert(int val){

       root= Insert(this->root,  val);

   }

    void    Delete(int val){

    root = Delete(this->root, val);

   }

    void PreOrderTraversal(){

        PreOrderTraversal( this->root);

    }

     int height(node\* t)

    {

        return (t == NULL ? -1 : t->height);

    }

    int length(node\* r){

        int  lh=0,rh=0;

        if(r==NULL)

         return -1;

        if(r->left!=NULL){

            lh++;

            lh+=length(r->left);

        }

        if(r->right!=NULL){

            rh++;

            rh+=length(r->right);

        }

        if(rh>lh) return rh;

        else return lh;

    }

    int treeHeight(node \*t)

    {

    int static l\_height=0;

    int static r\_height=0;

    if (t == NULL)

        return -1;

    else

    {

    l\_height = treeHeight(t->left);

       r\_height = treeHeight(t->right);

        if (l\_height > r\_height)

            return (l\_height + 1);

        else

            return (r\_height + 1);

    }

    }

};

int main (){

    AVLTree tree1, tree2;

      tree1.Insert(30);

    tree1.Insert(20);

    tree1.Insert(40);

    tree1.Insert(15);

    tree1.Delete(40);

    cout<<"Displaying nodes"<<endl;

    tree1.PreOrderTraversal();

    return 0;

}

node\* AVLTree::Insert(node\* t, int val ){

 if (t==NULL)

    {

         t= new node(val);

    }

    else if (t->data== val){

        cout<<"Record already exist"<<val<<endl;

    }

    else if (val < t->data) // insert on left s

    {

        t->left = Insert(t->left , val );

         int bf= height(t->left) - height(t->right);

            if(bf == 2)

            {

                if(val < t->left->data)

                    t = singleRightRotate(t);

                else

                    t = doubleLeftRightRotate(t);

            }

    }

    else if (val > t->data) // Right side

        {

        t->right= Insert( t->right,val);

        int bf=height(t->right) - height(t->left);

            if(bf == 2)

            {

                if(val > t->right->data)

                    t = singleLeftRotate(t);

                else

                    t = doubleRightLeftRotate(t);

            }

    }

    t->height = max(height(t->left), height(t->right))+1;

    return t;

}

node\* AVLTree::doubleRightLeftRotate(node\* &t)

    {

        t->right = singleRightRotate(t->right);

        return singleLeftRotate(t);

    }

node\* AVLTree::doubleLeftRightRotate(node\* &t)

    {

        t->left = singleLeftRotate(t->left);

        return singleRightRotate(t);

    }

node\* AVLTree::singleRightRotate(node\* &t)

    {

        node\* u = t->left;

        t->left = u->right;

        u->right = t;

        t->height = max(height(t->left), height(t->right))+1;

        u->height = max(height(u->left), t->height)+1;

        return u;

    }

node\* AVLTree::singleLeftRotate(node\* &t)

    {

        node\* u = t->right;

        t->right = u->left;

        u->left = t;

        t->height = max(height(t->left), height(t->right))+1;

        u->height = max(height(u->right), t->height)+1 ;

        return u;

    }

node \* AVLTree::Delete(node\* r, int data)

{

//    node \* r= root1;

    if(r==NULL)

     return r;

    else if(data < r->data)

        r->left = Delete(r->left, data);

    else if (data> r->data)

        r->right = Delete(r->right, data);

    else

    {

        //No child

        if(r->right == NULL && r->left == NULL)

        {

            delete r;

            r = NULL;

            return r;

        }

        //One child on left

        else if(r->right == NULL)

        {

            node\* temp = r;

            r= r->left;

            delete temp;

        }

        //One child on right

        else if(r->left == NULL)

        {

            node\* temp = r;

            r= r->right;

            delete temp;

        }

        //two child

        else

        {

            node\* temp = FindMax(r->right);

            int x= r->data;  //

            r->data = temp->data;

            temp->data= x;

            r->left = Delete(r->left, temp->data);

        }

    }

     if(r == NULL)

            return r;

        r->height = max(height(r->left), height(r->right))+1;

        int bal = height(r->left) - height(r->right);

            if(bal>1){

                if(height(r->left) >= height(r->right)){

                    return singleRightRotate(r);

                }else{

                    r->left = singleLeftRotate(r->left);

                    return singleRightRotate(r);

                }

            }else if(bal < -1){

                if(height(r->right) >= height(r->left)){

                    return singleLeftRotate(r);

                }else{

                    r->right = singleRightRotate(r->right);

                    return singleLeftRotate(r);

                }

            }

        return r;

}

node\* AVLTree::FindMax(node\* r){

    while(r->right!=NULL){

        r= r->right;

    }

    return r;

}

node\* AVLTree::PreOrderTraversal( node\* r){

     if (r == NULL)

        return NULL;

    cout << " "<< r->data << " -> ";

    PreOrderTraversal(r->left);

    PreOrderTraversal(r->right);

}

int path(node \*point){

    if(point==NULL){

        return 0;

        }

    else{

            int left=path(point->left);

            int right=path(point->right);

            if(left>right){

            return left+1;

            }

            else return right+1;

        }

    }

int balance\_fact(node \*point){

        int left=0;

        int right=0;

        if(point!=NULL){

        if(point->left!=NULL){

        left=path(point->left);

        }

        if(point->right!=NULL){

        right=path(point->right);

        }

        }

        return left-right;

    }

Output:



The END---------------------------------------------------------------------------------------------------------------