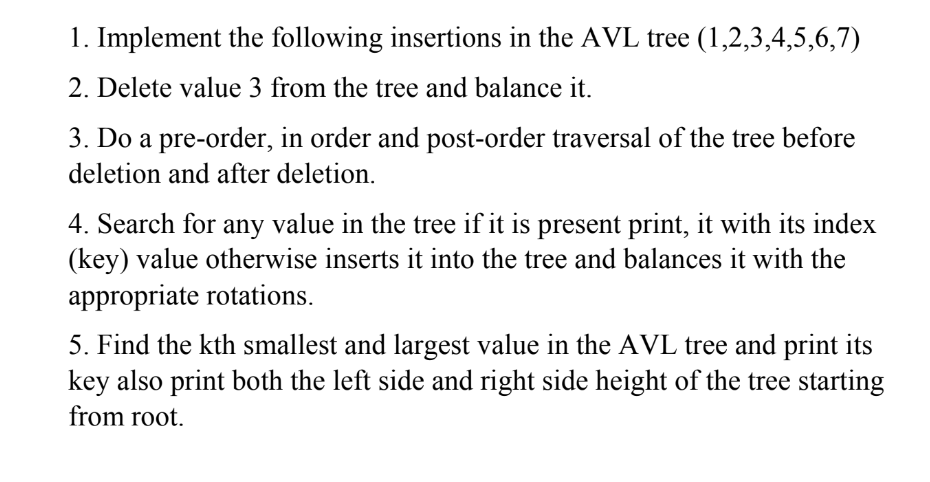
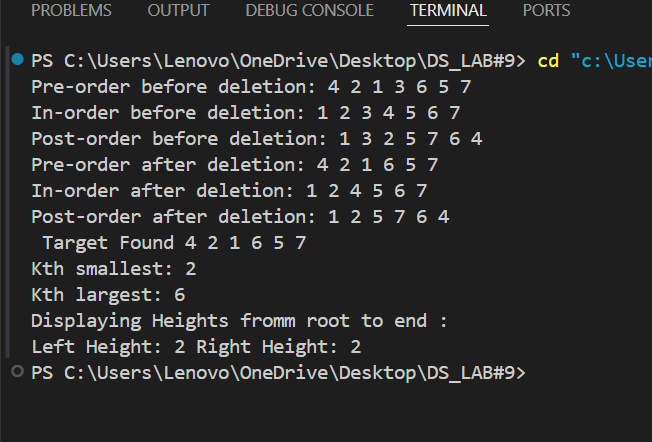
**K230607 DS\_LAB # 9 Nov 11,2024**

****



#include <iostream>

using namespace std;

class Node

{

public:

    int val;

    Node \*left;

    Node \*right;

    int height;

    Node(int k)

    {

        val = k;

        left = nullptr;

        right = nullptr;

        height = 1;

    }

    Node()

    {

        left = nullptr;

        right = nullptr;

    }

};

int height(Node \*node)

{

    if (node == nullptr)

        return 0;

    return node->height;

}

int getBalance(Node \*node)

{

    if (node == nullptr)

        return 0;

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

}

Node \*RightRotate(Node \*root)

{

    Node \*x = root->left;

    Node \*y = x->right;

    x->right = root;

    root->left = y;

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

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

    return x;

}

Node \*LeftRotate(Node \*root)

{

    Node \*x = root->right;

    Node \*y = x->left;

    x->left = root;

    root->right = y;

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

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

    return x;

}

Node \*balanceNode(Node \*root)

{

    int balance = getBalance(root);

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

        return RightRotate(root);

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

        return LeftRotate(root);

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

    {

        root->left = LeftRotate(root->left);

        return RightRotate(root);

    }

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

    {

        root->right = RightRotate(root->right);

        return LeftRotate(root);

    }

    return root;

}

Node \*Insertion(Node \*root, int key)

{

    if (!root)

        return new Node(key);

    if (key < root->val)

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

    else if (key > root->val)

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

    else

        return root;

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

    return balanceNode(root);

}

int minValNode(Node \*node)

{

    Node \*current = node;

    while (current->left != nullptr)

        current = current->left;

    return current->val;

}

Node \*Deletion(Node \*root, int key)

{

    if (root == nullptr)

        return root;

    if (key < root->val)

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

    else if (key > root->val)

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

    else

    {

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

        {

            delete root;

            return NULL;

        }

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

        {

            Node \*temp = root->right;

            delete root;

            return temp;

        }

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

        {

            Node \*temp = root->left;

            delete root;

            return temp;

        }

        else if (root->left && root->right)

        {

            int mini = minValNode(root->right);

            root->val = mini;

            root->right = Deletion(root->right, mini);

            return root;

        }

    }

    if (root == nullptr)

        return root;

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

    return balanceNode(root);

}

void Preorder\_Traversal(Node \*root)

{

    if (root)

    {

        cout << root->val << " ";

        Preorder\_Traversal(root->left);

        Preorder\_Traversal(root->right);

    }

}

void Inorder\_Traversal(Node \*root)

{

    if (root)

    {

        Inorder\_Traversal(root->left);

        cout << root->val << " ";

        Inorder\_Traversal(root->right);

    }

}

void Post\_Order(Node \*root)

{

    if (root)

    {

        Post\_Order(root->left);

        Post\_Order(root->right);

        cout << root->val << " ";

    }

}

bool Searching(Node \*root, int key)

{

    if (root == nullptr)

        return false;

    if (root->val == key)

        return true;

    if (key < root->val)

        return Searching(root->left, key);

    else

        return Searching(root->right, key);

}

Node \*FindKthSmallest(Node \*root, int &k)

{

    if (!root)

        return nullptr;

    Node \*left = FindKthSmallest(root->left, k);

    if (left)

        return left;

    if (--k == 0)

        return root;

    return FindKthSmallest(root->right, k);

}

Node \*FindKthLargest(Node \*root, int &k)

{

    if (!root)

        return nullptr;

    Node \*right = FindKthLargest(root->right, k);

    if (right)

        return right;

    if (--k == 0)

        return root;

    return FindKthLargest(root->left, k);

}

void Smallest\_Largest(Node \*root, int k)

{

    int kSmallest = k;

    Node \*smallest = FindKthSmallest(root, kSmallest);

    int kLargest = k;

    Node \*largest = FindKthLargest(root, kLargest);

    if (smallest)

        cout << "Kth smallest: " << smallest->val << endl;

    else

        cout << "Kth smallest not found" << endl;

    if (largest)

        cout << "Kth largest: " << largest->val << endl;

    else

        cout << "Kth largest not found" << endl;

}

void Display\_Heights(Node \*root)

{

    if (root)

    {

        cout << "Left Height: " << height(root->left) << " " << "Right Height: " << height(root->right) << endl;

    }

}

int main()

{

    Node \*root = nullptr;

    int values[] = {1, 2, 3, 4, 5, 6, 7};

    for (int i = 0; i < 7; i++)

        root = Insertion(root, values[i]);

    cout << "Pre-order before deletion: ";

    Preorder\_Traversal(root);

    cout << endl;

    cout << "In-order before deletion: ";

    Inorder\_Traversal(root);

    cout << endl;

    cout << "Post-order before deletion: ";

    Post\_Order(root);

    cout << endl;

    root = Deletion(root, 3);

    cout << "Pre-order after deletion: ";

    Preorder\_Traversal(root);

    cout << endl;

    cout << "In-order after deletion: ";

    Inorder\_Traversal(root);

    cout << endl;

    cout << "Post-order after deletion: ";

    Post\_Order(root);

    cout << endl;

    int key = 4;

    if (!Searching(root, key))

    {

        root = Insertion(root, key);

    }

    else

    {

        cout << " Target Found ";

        Preorder\_Traversal(root);

        cout << endl;

    }

    int k = 2;

    Smallest\_Largest(root, k);

    cout << "Displaying Heights fromm root to end : " << endl;

    Display\_Heights(root);

}