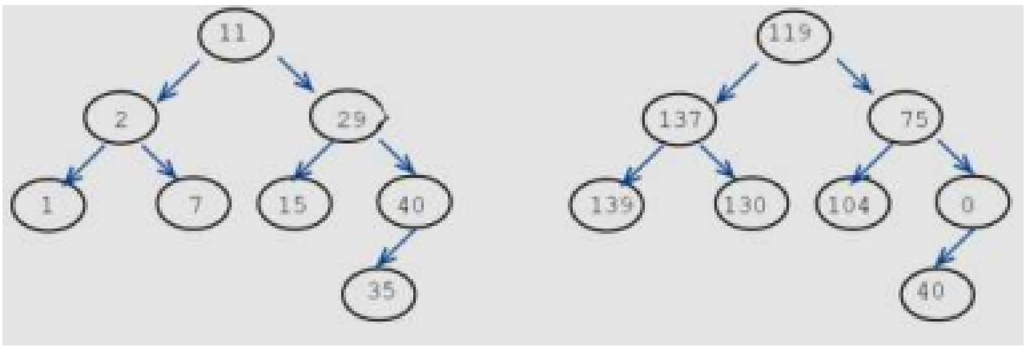
Week 8-LAB A

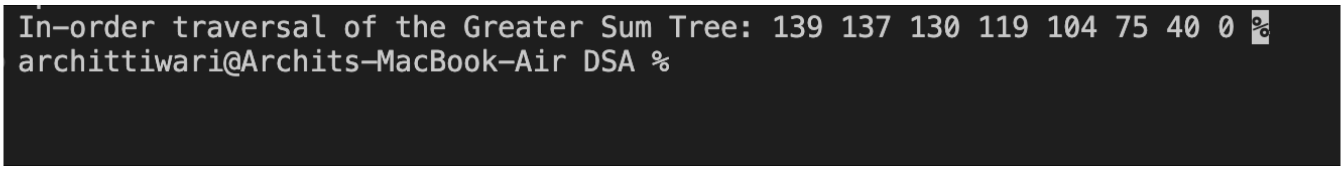
Q1. Given a BST, transform it into greater sum tree where each node contains sum of all nodes greater than that node.



Ans:

```
#include <iostream>

using namespace std;


class Node {

public:

    int data;

    Node* left;

    Node* right;


    Node(int value) {

        data = value;

        left = right = nullptr;

    }

};
```

```
class BST {
private:
    Node* root;
    int sum;

    void transformUtil(Node* node) {
        if (node == nullptr) {
            return;
        }
        transformUtil(node->right);
        sum += node->data;
        node->data = sum - node->data;
        transformUtil(node->left);
    }

public:
    BST() : root(nullptr), sum(0) {}

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

    Node* insertRec(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {

            node->left = insertRec(node->left, value);

        } else {

            node->right = insertRec(node->right, value);

        }

        return node;

    }


    void transform() {

        sum = 0;

        transformUtil(root);

    }


    void inOrderTraversal(Node* node) {

        if (node) {

            inOrderTraversal(node->left);

            cout << node->data << " ";

            inOrderTraversal(node->right);

        }

    }


    Node* getRoot() {

        return root;

    }

};


int main() {
    BST tree;

    tree.insert(11);

    tree.insert(2);

    tree.insert(29);

    tree.insert(1);

    tree.insert(7);

    tree.insert(15);

    tree.insert(40);

    tree.insert(35);


    tree.transform();


    cout << "In-order traversal of the Greater Sum Tree: ";

    tree.inOrderTraversal(tree.getRoot());


    return 0;

}
```

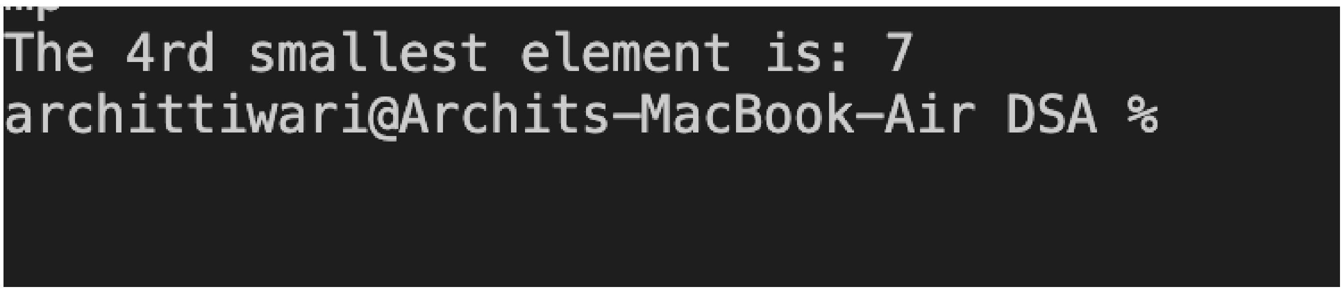


```
In-order traversal of the Greater Sum Tree: 139 137 130 119 104 75 40 0 %
archittiwari@Archits-MacBook-Air DSA %
```

Q2.WAP to find the kth smallest element in a BST.

Ans:

```
#include <iostream>

using namespace std;

struct TreeNode {

    int val;
    TreeNode *left;

    TreeNode *right;

    TreeNode(int x) {

        val = x;

        left = nullptr;

        right = nullptr;

    }

};

class Solution {

public:

    int kthSmallest(TreeNode* root, int k) {

        int count = 0;

        return inOrderTraversal(root, k, count);

    }

private:

    int inOrderTraversal(TreeNode* node, int k, int& count) {

        if (!node) return -1;

        int left = inOrderTraversal(node->left, k, count);

        if (left != -1) return left;

        count++;

        if (count == k) return node->val;

        return inOrderTraversal(node->right, k, count);

    }

};

void insert(TreeNode*& root, int val) {

    if (!root) {

        root = new TreeNode(val);
    } else if (val < root->val) {
        insert(root->left, val);
    } else {
        insert(root->right, val);
    }
}
int main() {
    TreeNode* root = nullptr;
    insert(root, 1);
    insert(root, 2);
    insert(root, 3);
    insert(root, 8);
    insert(root, 9);
    insert(root, 10);
    insert(root, 7);
    int k = 4;
    Solution sol;
    int result = sol.kthSmallest(root, k);

    if (result != -1) {
        cout << "The " << k << "rd smallest element is: " << result << endl;
    } else {
        cout << "Element not found." << endl;
    }
    return 0;
}
```



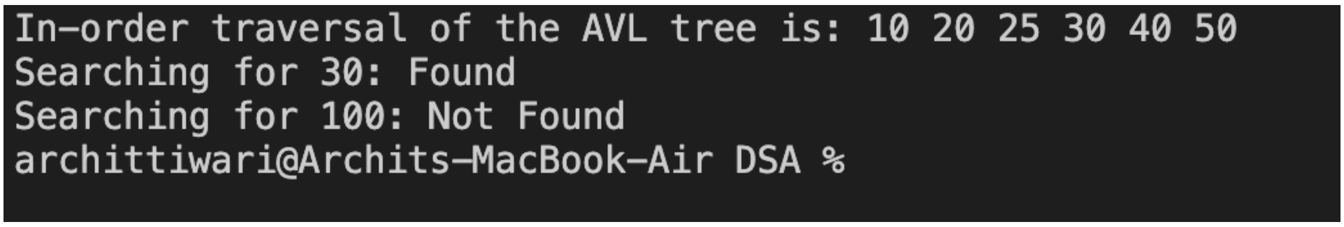
```
The 4rd smallest element is: 7
archittiwari@Archits-MacBook-Air DSA %
```

Q3. Implement an AVL Tree that supports insertion and search operations efficiently while maintaining its balanced property.

Ans:

```
#include <iostream>

class Node {
public:
    int data;
    Node* left;
    Node* right;
    int height;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    Node* root;

    int getHeight(Node* node) {
        return (node == nullptr) ? 0 : node->height;
    }

    int getBalance(Node* node) {
        return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);
    }
```

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;
    return x;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;
    return y;
}

Node* insert(Node* node, int value) {
    if (node == nullptr) {
        return new Node(value);
    }
    if (value < node->data) {
        node->left = insert(node->left, value);
    } else if (value > node->data) {
        node->right = insert(node->right, value);
    } else {
        return node;
    }

    node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));
    int balance = getBalance(node);

    if (balance > 1 && value < node->left->data) {
        return rightRotate(node);
        }
        if (balance < -1 && value > node->right->data) {
            return leftRotate(node);
        }
        if (balance > 1 && value > node->left->data) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
        if (balance < -1 && value < node->right->data) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        return node;
    }

    bool search(Node* node, int value) {
        if (node == nullptr) {
            return false;
        }
        if (value == node->data) {
            return true;
        }
        return value < node->data ? search(node->left, value) : search(node->right, value);
    }

public:
    AVLTree() {
        root = nullptr;
    }

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

    bool search(int value) {
        return search(root, value);
    }
    void inOrder(Node* node) {
        if (node != nullptr) {
            inOrder(node->left);
            std::cout << node->data << " ";
            inOrder(node->right);
        }
    }

    void inOrderTraversal() {
        inOrder(root);
        std::cout << std::endl;
    }
};

int main() {
    AVLTree tree;
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(40);
    tree.insert(50);
    tree.insert(25);
    std::cout << "In-order traversal of the AVL tree is: ";
    tree.inOrderTraversal();
    std::cout << "Searching for 30: " << (tree.search(30) ? "Found" : "Not Found") <<
        std::endl;
    std::cout << "Searching for 100: " << (tree.search(100) ? "Found" : "Not Found") <<
        std::endl;
    return 0;
}
```



```
In-order traversal of the AVL tree is: 10 20 25 30 40 50
Searching for 30: Found
Searching for 100: Not Found
archittiwari@Archits-MacBook-Air DSA %
```

Q4. WAP to implement AVL tree for the following elements:

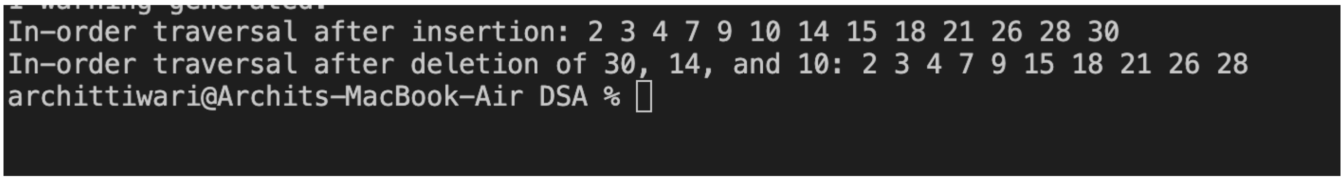
a. 21,26,30,9,4,14,28,18,15,10,2,3,7

b. Also, WAP to delete 30,14,10 nodes.

Ans:

```
#include <iostream>

class Node {
public:
    int data;
    Node* left;
    Node* right;
    int height;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    Node* root;

    int getHeight(Node* node) {
        return (node == nullptr) ? 0 : node->height;
    }

    int getBalance(Node* node) {
        return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);
    }

    Node* rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;
        x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;
        return x;
    }

    Node* leftRotate(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;
        y->left = x;
        x->right = T2;
        x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;
        y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;
        return y;
    }
```

```
Node* insert(Node* node, int value) {
    if (node == nullptr) {
        return new Node(value);
    }
    if (value < node->data) {
        node->left = insert(node->left, value);
    } else if (value > node->data) {
        node->right = insert(node->right, value);
    } else {
        return node;
    }

    node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));
    int balance = getBalance(node);

    if (balance > 1 && value < node->left->data) {
        return rightRotate(node);
    }
    if (balance < -1 && value > node->right->data) {
        return leftRotate(node);
    }
    if (balance > 1 && value > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && value < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
}

Node* deleteNode(Node* root, int value) {
    if (root == nullptr) {
        return root;
    }
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        if ((root->left == nullptr) || (root->right == nullptr)) {
            Node* temp = root->left ? root->left : root->right;
            if (temp == nullptr) {
                temp = root;
                root = nullptr;
                } else {
                    *root = *temp;
                }
                delete temp;
            } else {
                Node* temp = minValueNode(root->right);
                root->data = temp->data;
                root->right = deleteNode(root->right, temp->data);
            }
        }

        if (root == nullptr) {
            return root;
        }

        root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));
        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;
    }

    void inOrder(Node* node) {
        if (node != nullptr) {
            inOrder(node->left);
            std::cout << node->data << " ";
            inOrder(node->right);
        }
    }

public:
    AVLTree() {
        root = nullptr;
    }

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

    void deleteNode(int value) {
        root = deleteNode(root, value);
    }

    void inOrderTraversal() {
        inOrder(root);
        std::cout << std::endl;
    }
};

int main() {
    AVLTree tree;

    int elements[] = {21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7};
    for (int value : elements) {
        tree.insert(value);
    }

    std::cout << "In-order traversal after insertion: ";
    tree.inOrderTraversal();

    tree.deleteNode(30);
    tree.deleteNode(14);
    tree.deleteNode(10);

    std::cout << "In-order traversal after deletion of 30, 14, and 10: ";
    tree.inOrderTraversal();

    return 0;
}
```



```
In-order traversal after insertion: 2 3 4 7 9 10 14 15 18 21 26 28 30
In-order traversal after deletion of 30, 14, and 10: 2 3 4 7 9 15 18 21 26 28
archittiwari@Archits-MacBook-Air DSA % 
```

Q5. Develop functionality within an AVL Tree to handle deletions efficiently while preserving the tree's balanced nature. Also show the left and right children of each parent node.

Input 1:
AVL Insert: 40, 20, 60, 10, 30, 50, 70, 5, 15, 25, 35, 45, 55, 65, 75
AVL Delete: 5
Output 1:
10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75

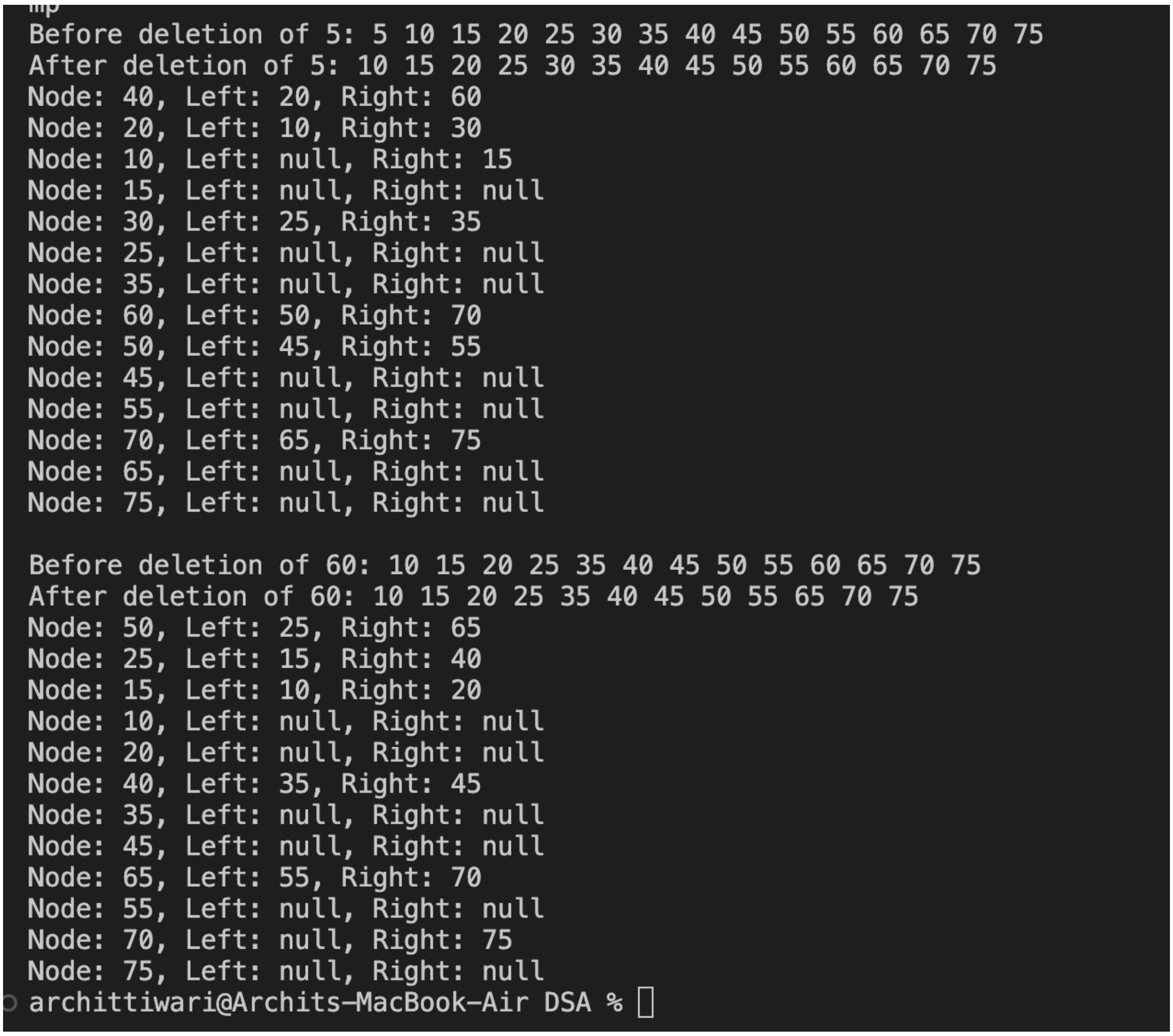
Input 2:
AVL Insert: 10, 15, 20, 25, 35, 40, 45, 50, 55, 60, 65, 70, 75
AVL Delete: 60

Output 2:
10, 15, 20, 25, 35, 40, 45, 50, 55, 65, 70, 75

Ans:

```
#include <iostream>
using namespace std;
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    int height;
    TreeNode(int val) {
        key = val;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};
class AVLTree {
public:
    TreeNode* root;
    AVLTree() {
        root = nullptr;
    }
    int height(TreeNode* node) {
        return node ? node->height : 0;
    }
    int getBalance(TreeNode* node) {
        return node ? height(node->left) - height(node->right) : 0;
    }
    TreeNode* rightRotate(TreeNode* y) {
        TreeNode* x = y->left;
        TreeNode* T2 = x->right;
        y->left = T2;
        x->right = y;
        y->height = max(height(y->left), height(y->right)) + 1;
        x->height = max(height(x->left), height(x->right)) + 1;
        return x;
    }
    TreeNode* leftRotate(TreeNode* x) {
        TreeNode* y = x->right;
        TreeNode* T2 = y->left;
        x->right = T2;
        y->left = x;
        x->height = max(height(x->left), height(x->right)) + 1;
        y->height = max(height(y->left), height(y->right)) + 1;
        return y;
    }
    TreeNode* insert(TreeNode* node, int key) {
        if (node == nullptr) {
            return new TreeNode(key);
        }
        if (key < node->key) {
            node->left = insert(node->left, key);
        } else if (key > node->key) {
```

```
        node->right = insert(node->right, key);
    } else {
        return node;  // Duplicates are not allowed
    }
    node->height = max(height(node->left), height(node->right)) + 1;
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key) {
        return rightRotate(node);
    }
    if (balance < -1 && key > node->right->key) {
        return leftRotate(node);
    }
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}
TreeNode* deleteNode(TreeNode* root, int key) {
    if (root == nullptr) {
        return root;
    }
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        if ((root->left == nullptr) || (root->right == nullptr)) {
            TreeNode* temp = root->left ? root->left : root->right;
            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else {
                *root = *temp;
            }
            delete temp;
        } else {
            TreeNode* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
    if (root == nullptr) {
        return root;
        }
        root->height = max(height(root->left), height(root->right)) + 1;
        int balance = getBalance(root);
        if (balance > 1 && getBalance(root->left) >= 0) {
            return rightRotate(root);
        }
        if (balance > 1 && getBalance(root->left) < 0) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }
        if (balance < -1 && getBalance(root->right) <= 0) {
            return leftRotate(root);
        }
        if (balance < -1 && getBalance(root->right) > 0) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }
        return root;
    }
    void insert(int key) {
        root = insert(root, key);
    }
    void deleteNode(int key) {
        root = deleteNode(root, key);
    }
    void inOrder(TreeNode* node) {
        if (node) {
            inOrder(node->left);
            cout << node->key << " ";
            inOrder(node->right);
        }
    }
    void display(TreeNode* node) {
        if (node) {
            cout << "Node: " << node->key
                 << ", Left: " << (node->left ? to_string(node->left->key) : "null")
                 << ", Right: " << (node->right ? to_string(node->right->key) : "null") << endl;
            display(node->left);
            display(node->right);
        }
    }
    void printInOrder() {
        inOrder(root);
        cout << endl;
    }
    void displayTree() {
        display(root);
    }
};
int main() {
    AVLTree tree;
    int insertElements1[] = {40, 20, 60, 10, 30, 50, 70, 5, 15, 25, 35, 45, 55, 65, 75};
    int deleteElement1 = 5;
    for (int i = 0; i < sizeof(insertElements1) / sizeof(insertElements1[0]); ++i) {
        tree.insert(insertElements1[i]);
    }
    cout << "Before deletion of " << deleteElement1 << ": ";
    tree.printInOrder();
    tree.deleteNode(deleteElement1);
    cout << "After deletion of " << deleteElement1 << ": ";
    tree.printInOrder();
    tree.displayTree();
    cout << endl;
    AVLTree tree2;
    int insertElements2[] = {10, 15, 20, 25, 35, 40, 45, 50, 55, 60, 65, 70, 75};
    int deleteElement2 = 60;
    for (int i = 0; i < sizeof(insertElements2) / sizeof(insertElements2[0]); ++i) {
        tree2.insert(insertElements2[i]);
    }
    cout << "Before deletion of " << deleteElement2 << ": ";
    tree2.printInOrder();
    tree2.deleteNode(deleteElement2);
    cout << "After deletion of " << deleteElement2 << ": ";
    tree2.printInOrder();
    tree2.displayTree();
    return 0;
}
```



```
mp
Before deletion of 5: 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
After deletion of 5: 10 15 20 25 30 35 40 45 50 55 60 65 70 75
Node: 40, Left: 20, Right: 60
Node: 20, Left: 10, Right: 30
Node: 10, Left: null, Right: 15
Node: 15, Left: null, Right: null
Node: 30, Left: 25, Right: 35
Node: 25, Left: null, Right: null
Node: 35, Left: null, Right: null
Node: 60, Left: 50, Right: 70
Node: 50, Left: 45, Right: 55
Node: 45, Left: null, Right: null
Node: 55, Left: null, Right: null
Node: 70, Left: 65, Right: 75
Node: 65, Left: null, Right: null
Node: 75, Left: null, Right: null

Before deletion of 60: 10 15 20 25 35 40 45 50 55 60 65 70 75
After deletion of 60: 10 15 20 25 35 40 45 50 55 65 70 75
Node: 50, Left: 25, Right: 65
Node: 25, Left: 15, Right: 40
Node: 15, Left: 10, Right: 20
Node: 10, Left: null, Right: null
Node: 20, Left: null, Right: null
Node: 40, Left: 35, Right: 45
Node: 35, Left: null, Right: null
Node: 45, Left: null, Right: null
Node: 65, Left: 55, Right: 70
Node: 55, Left: null, Right: null
Node: 70, Left: null, Right: 75
Node: 75, Left: null, Right: null
archittiwari@Archits-MacBook-Air DSA %
```

Q6. Problem Statement: You are given a set of key-value pairs to insert into an AVL tree. After the insertions, you need to perform a range query that partially overlaps with the existing keys. The test should validate that the tree accurately includes only the keys that fall within the specified range, even if they are close to the boundaries.

Insert: Insert a key-value pair into the tree.

Range Query: Given two keys, low and high, return the sum of values for all keys within this range (inclusive).

Insertions:

Insert (10, 100)
Insert (20, 200)
Insert (30, 300)
Insert (40, 400)
Insert (50, 500)

Range Query:

Perform a range query from 25 to 45.

Expected Output:

The expected output for the range query should be the sum of values of the keys 30 and 40, which are the ones falling within the 25 to 45 range. Thus, the output should be 300 + 400 = 700.

Insert:

```
10, 100
20, 200
30, 300
```

Range

```
10 30
```

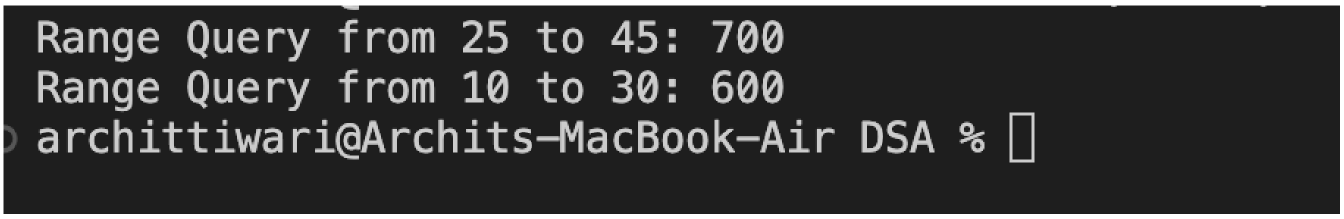
Output:

```
600
```

Ans:

```
#include <iostream>
using namespace std;

class Node {
public:
    int key;
    int value;
    Node* left;
    Node* right;
    int height;

    Node(int k, int v) {
        key = k;
        value = v;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    Node* root;

    int getHeight(Node* node) {
        return (node == nullptr) ? 0 : node->height;
    }

    int getBalance(Node* node) {
        return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);
    }

    Node* rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
        x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
        return x;
```

```
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    return y;
}

Node* insert(Node* node, int key, int value) {
    if (node == nullptr) {
        return new Node(key, value);
    }
    if (key < node->key) {
        node->left = insert(node->left, key, value);
    } else if (key > node->key) {
        node->right = insert(node->right, key, value);
    } else {
        node->value = value;  // Update value if key already exists
        return node;
    }

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key) {
        return rightRotate(node);
    }
    if (balance < -1 && key > node->right->key) {
        return leftRotate(node);
    }
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

int rangeQuery(Node* node, int low, int high) {
    if (node == nullptr) {
        return 0;
    }
    if (node->key < low) {
        return rangeQuery(node->right, low, high);
    }
    if (node->key > high) {
        return rangeQuery(node->left, low, high);
        }
        return node->value + rangeQuery(node->left, low, high) + rangeQuery(node->right, low,
high);
    }

public:
    AVLTree() {
        root = nullptr;
    }

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

    int rangeQuery(int low, int high) {
        return rangeQuery(root, low, high);
    }
};

int main() {
    AVLTree tree;

    tree.insert(10, 100);
    tree.insert(20, 200);
    tree.insert(30, 300);
    tree.insert(40, 400);
    tree.insert(50, 500);

    int low = 25, high = 45;
    cout << "Range Query from " << low << " to " << high << ": " << tree.rangeQuery(low, high)
<< endl;

    AVLTree tree2;
    tree2.insert(10, 100);
    tree2.insert(20, 200);
    tree2.insert(30, 300);

    low = 10, high = 30;
    cout << "Range Query from " << low << " to " << high << ": " << tree2.rangeQuery(low,
high) << endl;

    return 0;
}
```



```
Range Query from 25 to 45: 700
Range Query from 10 to 30: 600
archittiwari@Archits-MacBook-Air DSA %
```