15B17CI371 – Data Structures Lab
ODD 2024
Week 8-LAB A
Practice Lab

1.

```
#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:
    void transformToGST(TreeNode* root) {
        int cumulative_sum = 0;
        reverseInOrderTraversal(root, cumulative_sum);
    }

private:
    void reverseInOrderTraversal(TreeNode* node, int& cumulative_sum) {
        if (!node) return;

        reverseInOrderTraversal(node->right, cumulative_sum);
        cumulative_sum += node->val;
        node->val = cumulative_sum - node->val;
        reverseInOrderTraversal(node->left, cumulative_sum);
    }
};

void printInOrder(TreeNode* node) {
    if (node) {
```

```
        printInOrder(node->left);
        cout << node->val << " ";
        printInOrder(node->right);
    }
}

int main() {
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    root->right->left = new TreeNode(5);
    root->right->right = new TreeNode(7);

    Solution().transformToGST(root);
    printInOrder(root);

    return 0;
}
```

2.

```
#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;
```

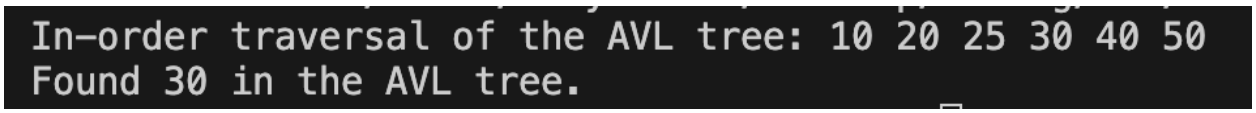
```
        return inOrderTraversal(root, k, count);
    }
private:
    int inOrderTraversal(TreeNode* node, int k, int& count) {
        if (!node) return -1;
        int left = inOrderTraversal(node->left, k, count);
        if (left != -1) return left;
        count++;
        if (count == k) return 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, 5);
    insert(root, 3);
    insert(root, 7);
    insert(root, 2);
    insert(root, 4);
    insert(root, 6);
    insert(root, 8);
    int k = 3;
    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;
}
```

3.

```
#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) {
        if (node == nullptr) {
            return 0;
        }
        return node->height;
    }
    int getBalance(TreeNode* node) {
        if (node == nullptr) {
            return 0;
        }
        return height(node->left) - height(node->right);
    }
    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;
    }
    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* search(TreeNode* node, int key) {
    if (node == nullptr || node->key == key) {
        return node;
    }
    if (key < node->key) {
        return search(node->left, key);
    }
    return search(node->right, key);
}
void insert(int key) {
    root = insert(root, key);
}
TreeNode* search(int key) {
    return search(root, key);
```

```
    }
    void inOrder(TreeNode* node) {
        if (node) {
            inOrder(node->left);
            cout << node->key << " ";
            inOrder(node->right);
        }
    }
    void printInOrder() {
        inOrder(root);
        cout << endl;
    }
};
int main() {
    AVLTree tree;
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(40);
    tree.insert(50);
    tree.insert(25);
    cout << "In-order traversal of the AVL tree: ";
    tree.printInOrder();
    int key = 30;
    TreeNode* result = tree.search(key);
    if (result) {
        cout << "Found " << key << " in the AVL tree." << endl;
    } else {
        cout << key << " not found in the AVL tree." << endl;
    }
    return 0;
}
```



4.

```
#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;
        }
```

```
    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->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 rangeQuery(TreeNode* node, int low, int high) {
    if (node) {
        if (low < node->key) {
            rangeQuery(node->left, low, high);
        }
        if (low <= node->key && high >= node->key) {
            cout << node->key << " ";
        }
        if (high > node->key) {
            rangeQuery(node->right, low, high);
        }
    }
}
```

```
    void printInOrder() {
        inOrder(root);
        cout << endl;
    }
    void rangeQuery(int low, int high) {
        rangeQuery(root, low, high);
        cout << endl;
    }
};
int main() {
    AVLTree tree;
    int elements[] = {21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7};
    int size = sizeof(elements) / sizeof(elements[0]);
    for (int i = 0; i < size; ++i) {
        tree.insert(elements[i]);
    }
    cout << "In-order traversal after insertion: ";
    tree.printInOrder();
    tree.deleteNode(30);
    tree.deleteNode(14);
    tree.deleteNode(10);
    cout << "In-order traversal after deletion: ";
    tree.printInOrder();
    cout << "Range query between 20 and 30: ";
    tree.rangeQuery(20, 30);
    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: 2 3 4 7 9 15 18 21 26 28
Range query between 20 and 30: 21 26 28
```

5.

```
#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;
}
```

```
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
```

6.

```
#include <iostream>
using namespace std;
struct TreeNode {
    int key;
    int value;
    TreeNode* left;
    TreeNode* right;
    int height;
    TreeNode(int k, int v) {
        key = k;
        value = v;
        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, int value) {
    if (node == nullptr) {
        return new TreeNode(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;
        return node;
    }
    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;
    }
    int rangeQuery(TreeNode* 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);
    }
    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;
    int sum = tree.rangeQuery(low, high);
    cout << "Sum of values in range [" << low << ", " << high << "] = " << sum << endl;
    tree.insert(10, 100);
    tree.insert(20, 200);
    tree.insert(30, 300);
    low = 10;
    high = 30;
    sum = tree.rangeQuery(low, high);
    cout << "Sum of values in range [" << low << ", " << high << "] = " << sum << endl;
    return 0;
}
```

```
Sum of values in range [25, 45] = 700
Sum of values in range [10, 30] = 600
```