**Name: Talha Abdullah Bangyal**

**Roll number: 22F-3194**

**Section: BAI-3A**

**Lab: 11**

**----------------------------------------------------------------------------**

**Task 1**

**Problem**

#include <iostream>

#include <string>

using namespace std;

struct Node {

string word;

Node\* left;

Node\* right;

int height;

Node(const string& w) : word(w), left(nullptr), right(nullptr), height(1) {}

};

class AVLTree {

private:

Node\* root;

int max(int a, int b) {

if (a > b) {

return a;

}

else {

return b;

}

}

int height(Node\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int balanceFactor(Node\* node) {

if (node == nullptr) {

return 0;

}

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

}

void updateHeight(Node\* node) {

if (node) {

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

}

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* temp = x->right;

x->right = y;

y->left = temp;

updateHeight(y);

updateHeight(x);

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* temp = y->left;

y->left = x;

x->right = temp;

updateHeight(x);

updateHeight(y);

return y;

}

Node\* insert(Node\* node, const string& word) {

if (!node)

return new Node(word);

if (word < node->word) {

node->left = insert(node->left, word);

}

else if (word > node->word) {

node->right = insert(node->right, word);

}

else {

return node;

}

updateHeight(node);

int balance = balanceFactor(node);

if (balance > 1 && word < node->left->word) {

return rightRotate(node);

}

if (balance < -1 && word > node->right->word) {

return leftRotate(node);

}

if (balance > 1 && word > node->left->word) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && word < node->right->word) {

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, const string& word) {

if (!root) return root;

if (word < root->word) {

root->left = deleteNode(root->left, word);

}

else if (word > root->word) {

root->right = deleteNode(root->right, word);

}

else {

if (!root->left) {

Node\* temp = root->right;

delete root;

return temp;

}

else if (!root->right) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* temp = minValueNode(root->right);

root->word = temp->word;

root->right = deleteNode(root->right, temp->word);

}

if (!root) return root;

updateHeight(root);

int balance = balanceFactor(root);

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

return rightRotate(root);

}

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

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

return rightRotate(root);

}

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

return leftRotate(root);

}

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

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

return leftRotate(root);

}

return root;

}

void inOrder(Node\* node) {

if (node) {

inOrder(node->left);

cout << node->word << " ";

inOrder(node->right);

}

}

bool search(Node\* node, const string& word) {

if (!node)

return false;

if (word == node->word)

return true;

if (word < node->word)

return search(node->left, word);

return search(node->right, word);

}

public:

AVLTree() : root(nullptr) {}

void insert(const string& word) {

root = insert(root, word);

}

void deleteWord(const string& word) {

root = deleteNode(root, word);

}

bool search(const string& word) {

return search(root, word);

}

void display() {

inOrder(root);

cout << endl;

}

};

void displayMenu() {

cout << "--- AVL Tree Dictionary ---" << endl;

cout << "1. Insert a word" << endl;

cout << "2. Search for a word" << endl;

cout << "3. Delete a word" << endl;

cout << "4. Display sorted dictionary" << endl;

cout << "0. Exit" << endl;

cout << "Choose an option (0-4): ";

}

int main() {

AVLTree tree;

string word;

int opt;

string words[] = { "apple", "banana", "cat", "dog", "elephant", "lion", "hippo", "zebra", "cherry", "lemon", "blueberry" };

for (const auto& w : words) {

tree.insert(w);

}

while (true) {

displayMenu();

cin >> opt;

switch (opt) {

case 1:

cout << "Write a word to insert: ";

cin >> word;

tree.insert(word);

cout << word << " is inserted." << endl;

break;

case 2:

cout << "Write a word to search: ";

cin >> word;

if (tree.search(word)) {

cout << word << " is found in dictionary." << endl;

}

else {

cout << word << " is not found in dictionary." << endl;

}

break;

case 3:

cout << "Write a word to delete: ";

cin >> word;

tree.deleteWord(word);

cout << word << " is deleted." << endl;

break;

case 4:

cout << "Sorted dictionary (in-order): ";

tree.display();

break;

case 0:

cout << "Exiting the program." << endl;

return 0;

default:

cout << "Invalid option. Please try again." << endl;

break;

}

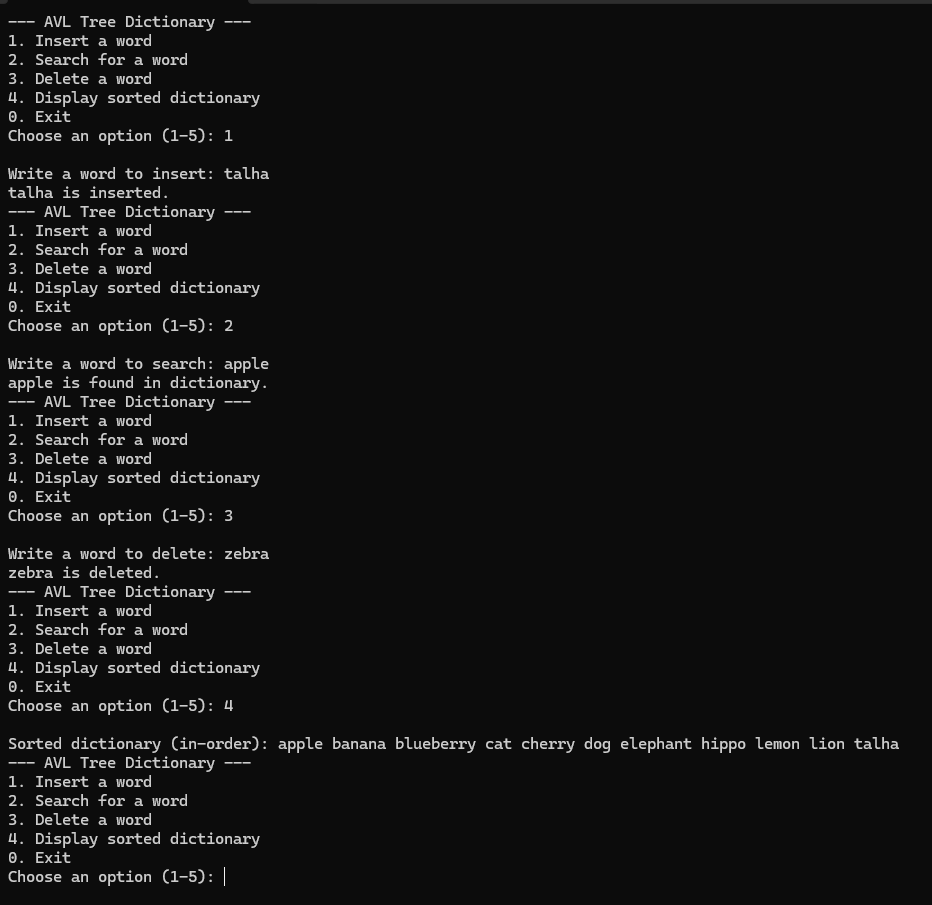
}

system("pause");

return 0;

}

**Screenshot**



**Task 2**

**Problem**

#include <iostream>

using namespace std;

struct Node {

int value;

Node\* left;

Node\* right;

int height;

Node(int val) : value(val), left(nullptr), right(nullptr), height(1) {}

};

int max(int a, int b) {

if (a > b) {

return a;

}

else {

return b;

}

}

int height(Node\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int balanceFactor(Node\* node) {

if (node == nullptr) {

return 0;

}

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

}

void updateHeight(Node\* node) {

if (node) {

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

}

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* temp = x->right;

x->right = y;

y->left = temp;

updateHeight(y);

updateHeight(x);

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* temp = y->left;

y->left = x;

x->right = temp;

updateHeight(x);

updateHeight(y);

return y;

}

Node\* insert(Node\* node, int value) {

if (!node) return new Node(value);

if (value < node->value) {

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

}

else if (value > node->value) {

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

}

else {

return node;

}

updateHeight(node);

int balance = balanceFactor(node);

if (balance > 1 && value < node->left->value) {

return rightRotate(node);

}

if (balance < -1 && value > node->right->value) {

return leftRotate(node);

}

if (balance > 1 && value > node->left->value) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && value < node->right->value) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

Node\* BSTtoAVL(Node\* bstRoot) {

if (!bstRoot) return nullptr;

Node\* avlRoot = new Node(bstRoot->value);

avlRoot->left = BSTtoAVL(bstRoot->left);

avlRoot->right = BSTtoAVL(bstRoot->right);

updateHeight(avlRoot);

int balance = balanceFactor(avlRoot);

if (balance > 1 && balanceFactor(avlRoot->left) >= 0) {

return rightRotate(avlRoot);

}

if (balance < -1 && balanceFactor(avlRoot->right) <= 0) {

return leftRotate(avlRoot);

}

if (balance > 1 && balanceFactor(avlRoot->left) < 0) {

avlRoot->left = leftRotate(avlRoot->left);

return rightRotate(avlRoot);

}

if (balance < -1 && balanceFactor(avlRoot->right) > 0) {

avlRoot->right = rightRotate(avlRoot->right);

return leftRotate(avlRoot);

}

return avlRoot;

}

bool hasPathSum(Node\* root, int sum, int X) {

if (!root)

return false;

sum += root->value;

if (!root->left && !root->right) {

return sum > X;

}

return hasPathSum(root->left, sum, X) || hasPathSum(root->right, sum, X);

}

Node\* swapSubtrees(Node\* root) {

if (!root) return nullptr;

swap(root->left, root->right);

updateHeight(root);

int balance = balanceFactor(root);

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

return rightRotate(root);

}

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

return leftRotate(root);

}

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

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

return rightRotate(root);

}

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

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

return leftRotate(root);

}

return root;

}

void printInOrder(Node\* node) {

if (!node) return;

printInOrder(node->left);

cout << node->value << " ";

printInOrder(node->right);

}

void displayMenu() {

cout << "--- BST-AVL Tree ---" << endl;

cout << "1. Insert " << endl;

cout << "2. Convert BST-AVL" << endl;

cout << "3. Display the BST-AVL" << endl;

cout << "4. Sum Target" << endl;

cout << "5. Swap sub Tree" << endl;

cout << "0. Exit" << endl;

}

int main() {

Node\* bstRoot = nullptr;

Node\* avlRoot = nullptr;

int choice, value;

while (true) {

displayMenu();

cout << "Choose an option (0-5): ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

bstRoot = insert(bstRoot, value);

break;

case 2:

avlRoot = BSTtoAVL(bstRoot);

cout << "Converted BST to AVL." << endl;

break;

case 3:

cout << "AVL tree in-order: ";

printInOrder(avlRoot);

cout << endl;

break;

case 4:

int X;

cout << "Enter target sum: ";

cin >> X;

if (hasPathSum(avlRoot, 0, X))

cout << "Yes" << endl;

else

cout << "No" << endl;

break;

case 5:

avlRoot = swapSubtrees(avlRoot);

cout << "Swapped subtrees." << endl;

break;

case 0:

cout << "Exiting..." << endl;

return 0;

default:

cout << "Invalid option. Please try again." << endl;

}

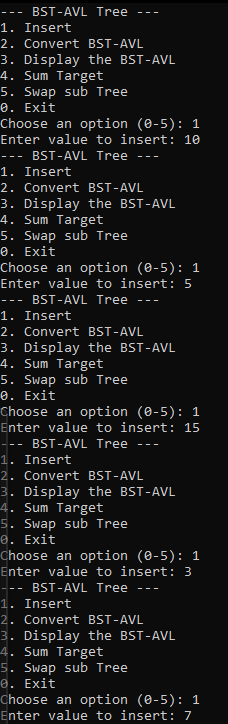
}

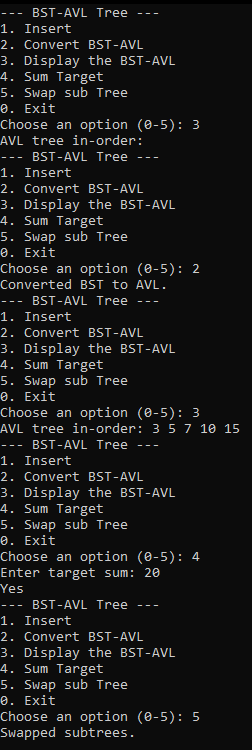
system("pause");

return 0;

}

**Screenshot**





A screen shot of a computer

Description automatically generated

**Task 3**

**Problem**

#include <iostream>

#include <string>

using namespace std;

struct Node {

string username;

int timestamp;

Node\* left;

Node\* right;

int height;

Node(string user, int time) : username(user), timestamp(time), left(nullptr), right(nullptr), height(1) {}

};

int max(int a, int b) {

if (a > b) {

return a;

}

else {

return b;

}

}

int height(Node\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int balanceFactor(Node\* node) {

if (node == nullptr) {

return 0;

}

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

}

void updateHeight(Node\* node) {

if (node)

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

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* temp = x->right;

x->right = y;

y->left = temp;

updateHeight(y);

updateHeight(x);

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* temp = y->left;

y->left = x;

x->right = temp;

updateHeight(x);

updateHeight(y);

return y;

}

Node\* insertOrUpdate(Node\* node, string username, int timestamp) {

if (!node)

return new Node(username, timestamp);

if (username < node->username)

node->left = insertOrUpdate(node->left, username, timestamp);

else if (username > node->username)

node->right = insertOrUpdate(node->right, username, timestamp);

else {

node->timestamp = timestamp;

return node;

}

updateHeight(node);

int balance = balanceFactor(node);

if (balance > 1 && username < node->left->username)

return rightRotate(node);

if (balance < -1 && username > node->right->username)

return leftRotate(node);

if (balance > 1 && username > node->left->username) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && username < node->right->username) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void rangeQuery(Node\* node, int startTime, int endTime) {

if (!node) return;

if (node->timestamp >= startTime)

rangeQuery(node->left, startTime, endTime);

if (node->timestamp >= startTime && node->timestamp <= endTime)

cout << node->username << " ";

if (node->timestamp <= endTime)

rangeQuery(node->right, startTime, endTime);

}

void printInOrder(Node\* node) {

if (!node) return;

printInOrder(node->left);

cout << node->username << " (" << node->timestamp << ") ";

printInOrder(node->right);

}

void displayMenu() {

cout << "--- Instagram login count menu ---" << endl;

cout << "1. Insert/Update User" << endl;

cout << "2. Query Users by Time Range" << endl;

cout << "3. Display All Users" << endl;

cout << "0. Exit" << endl;

}

int main() {

Node\* root = nullptr;

int choice, timestamp, startTime, endTime;

string username;

while (true) {

displayMenu();

cout << "Choose an option (0-3): ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter username: ";

cin >> username;

cout << "Enter timestamp: ";

cin >> timestamp;

root = insertOrUpdate(root, username, timestamp);

break;

case 2:

cout << "Enter start time: ";

cin >> startTime;

cout << "Enter end time: ";

cin >> endTime;

cout << "Users who logged in between " << startTime << " and " << endTime << ":\n";

rangeQuery(root, startTime, endTime);

cout << endl;

break;

case 3:

cout << "All users (in-order): ";

printInOrder(root);

cout << endl;

break;

case 0:

cout << "Exiting..." << endl;

return 0;

default:

cout << "Invalid option. Please try again." << endl;

}

}

system("pause");

return 0;

}

**Screenshot**

A screenshot of a computer program

Description automatically generated

**Task 4**

**Problem**

#include <iostream>

using namespace std;

struct Node {

int value;

Node\* left;

Node\* right;

int height;

Node(int val) : value(val), left(nullptr), right(nullptr), height(1) {}

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(Node\* node) {

return node ? node->height : 0;

}

int balanceFactor(Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

void updateHeight(Node\* node) {

if (node) {

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

}

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

Node\* insert(Node\* node, int value) {

if (!node) return new Node(value);

if (value < node->value)

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

else if (value > node->value)

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

else

return node;

updateHeight(node);

int balance = balanceFactor(node);

if (balance > 1 && value < node->left->value)

return rightRotate(node);

if (balance < -1 && value > node->right->value)

return leftRotate(node);

if (balance > 1 && value > node->left->value) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && value < node->right->value) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

Node\* findMax(Node\* node) {

while (node->right) {

node = node->right;

}

return node;

}

Node\* join(Node\* T1, Node\* T2) {

if (!T1) return T2;

if (!T2) return T1;

Node\* maxNode = findMax(T1);

maxNode->right = T2;

updateHeight(maxNode);

int balance = balanceFactor(T1);

if (balance > 1 && balanceFactor(T1->left) >= 0) {

return rightRotate(T1);

}

if (balance < -1 && balanceFactor(T1->right) <= 0) {

return leftRotate(T1);

}

if (balance > 1 && balanceFactor(T1->left) < 0) {

T1->left = leftRotate(T1->left);

return rightRotate(T1);

}

if (balance < -1 && balanceFactor(T1->right) > 0) {

T1->right = rightRotate(T1->right);

return leftRotate(T1);

}

return T1;

}

void printInOrder(Node\* node) {

if (!node) return;

printInOrder(node->left);

cout << node->value << " ";

printInOrder(node->right);

}

void displayMenu() {

cout << "--- AVL Tree Operations ---" << endl;

cout << "1. Insert into T1" << endl;

cout << "2. Insert into T2" << endl;

cout << "3. Join T1 and T2" << endl;

cout << "4. Display joined AVL tree (in-order)" << endl;

cout << "0. Exit" << endl;

}

int main() {

Node\* T1 = nullptr;

Node\* T2 = nullptr;

Node\* joinedTree = nullptr;

int choice, value;

while (true) {

displayMenu();

cout << "Choose an option (0-4): ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert into T1: ";

cin >> value;

T1 = insert(T1, value);

cout << "Inserted " << value << " into T1." << endl;

break;

case 2:

cout << "Enter value to insert into T2: ";

cin >> value;

T2 = insert(T2, value);

cout << "Inserted " << value << " into T2." << endl;

break;

case 3:

joinedTree = join(T1, T2);

cout << "Joined T1 and T2 into a single AVL tree." << endl;

break;

case 4:

if (joinedTree) {

cout << "In-order traversal of the joined AVL tree: ";

printInOrder(joinedTree);

cout << endl;

}

else {

cout << "The AVL tree has not been joined yet." << endl;

}

break;

case 0:

cout << "Exiting..." << endl;

return 0;

default:

cout << "Invalid option. Please try again." << endl;

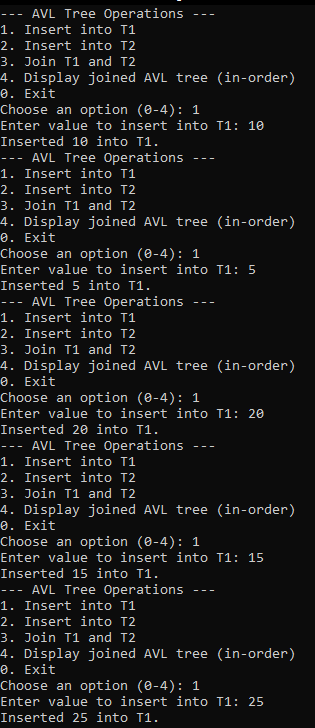
}

}

return 0;

}

**Screenshot**



A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated