**Waleed Afzal**

**23p-0566**

**Assignment 3**

**3D**

**Ques1:**

#include<iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

class BST {

Node\* root; // Root node of the BST

public:

BST() {

root = NULL;

}

void InsertNode(int num) {

Node\* newNode = new Node(num);

if (root == NULL) {

root = newNode;

}

else { // Tree is not empty

Node\* temp = root; // create a pointer to traverse the tree

while (true) {

if (num < temp->data) { // Left subtree

if (temp->left != NULL) {

temp = temp->left;

}

else {

temp->left = newNode;

return;

}

}

else if (num > temp->data) { // Right subtree

if (temp->right != NULL) {

temp = temp->right;

}

else {

temp->right = newNode;

return;

}

}

else {

cout << "Duplicate value found in tree.\n";

break;

}

}

}

}

bool FindValue(int val) {

Node\* temp = root;

while (temp != NULL) {

if (temp->data == val) {

return true;

}

else if (val < temp->data) {

temp = temp->left;

}

else {

temp = temp->right;

}

}

return false;

}

void inorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

}

void PostorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

PostorderTraversal(root->left);

PostorderTraversal(root->right);

cout << root->data << " ";

}

}

void PreorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

cout << root->data << " ";

PreorderTraversal(root->left);

PreorderTraversal(root->right);

}

}

void DisplayInOrder() {

cout << "\nDisplaying InOrder: ";

inorderTraversal(root);

}

void DisplayPreOrder() {

cout << "\nDisplaying PreOrder: ";

PreorderTraversal(root);

}

void DisplayPostOrder() {

cout << "\nDisplaying PostOrder: ";

PostorderTraversal(root);

}

};

int main() {

int value = 0;

BST T1;

cout << "Enter Values to Store (-1 To Exit): " << endl;

while (value != -1) {

cin >> value;

if (value != -1) {

T1.InsertNode(value); // Insert only if value is not -1

}

}

cout << "Enter any Value to Find in Tree: ";

cin >> value;

if (T1.FindValue(value) == true) {

cout << "\nValue Found\n";

}

else {

cout << "\nValue Not Found!\n";

}

T1.DisplayInOrder();

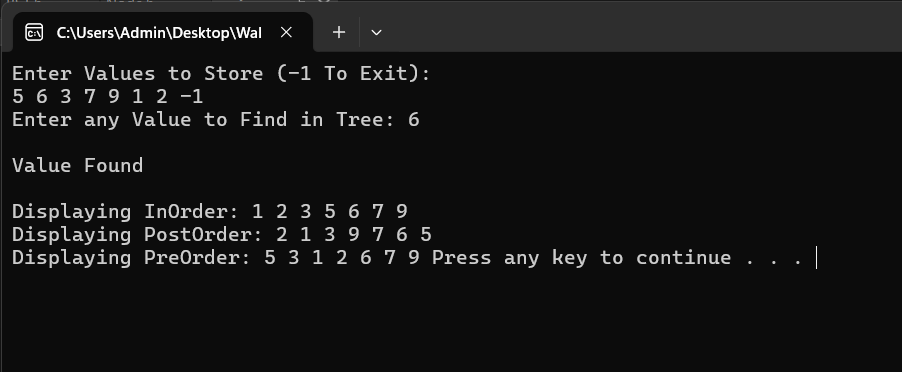
T1.DisplayPostOrder();

T1.DisplayPreOrder();

system("pause");

return 0;

}

****

**Q2:**

#include<iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

class BST {

Node\* root; // Root node of the BST

public:

BST() {

root = NULL;

}

void InsertNode(int num) {

Node\* newNode = new Node(num);

if (root == NULL) {

root = newNode;

}

else { // Tree is not empty

Node\* temp = root; // create a pointer to traverse the tree

while (true) {

if (num < temp->data) { // Left subtree

if (temp->left != NULL) {

temp = temp->left;

}

else {

temp->left = newNode;

return;

}

}

else if (num > temp->data) { // Right subtree

if (temp->right != NULL) {

temp = temp->right;

}

else {

temp->right = newNode;

return;

}

}

else {

cout << "Duplicate value found in tree.\n";

break;

}

}

}

}

bool FindValue(int val) {

Node\* temp = root;

while (temp != NULL) {

if (temp->data == val) {

return true;

}

else if (val < temp->data) {

temp = temp->left;

}

else {

temp = temp->right;

}

}

return false;

}

void inorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

}

void PostorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

PostorderTraversal(root->left);

PostorderTraversal(root->right);

cout << root->data << " ";

}

}

void PreorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

cout << root->data << " ";

PreorderTraversal(root->left);

PreorderTraversal(root->right);

}

}

void DisplayInOrder() {

cout << "\nDisplaying InOrder: ";

inorderTraversal(root);

}

void DisplayPreOrder() {

cout << "\nDisplaying PreOrder: ";

PreorderTraversal(root);

}

void DisplayPostOrder() {

cout << "\nDisplaying PostOrder: ";

PostorderTraversal(root);

}

void callArraytoBst(int\* arr, int size) {

root = arrayToBst(arr, 0, size - 1);

}

Node\* arrayToBst(int\* arr, int left, int right) {

if (left > right) {

return NULL;

}

int mid = (left + right) / 2;

Node\* newNode = new Node(arr[mid]);

newNode->left = arrayToBst(arr, left, mid - 1);

newNode->right = arrayToBst(arr, mid + 1, right);

return newNode;

}

};

int main() {

BST T1;

int\* arr;

int size = 0;

cout << "Enter Size of Array: " << endl;

cin >> size;

arr = new int[size];

cout << "\nEnter Values of Elements\n";

for (int i = 0;i < size;i++) {

cout << "Element " << i + 1 << " : ";

cin >> arr[i];

}

cout << "Converting Array to Binary Search Tree: \n";

T1.callArraytoBst(arr, size);

cout << "Converted Array to Binary Tree\n";

T1.DisplayInOrder();

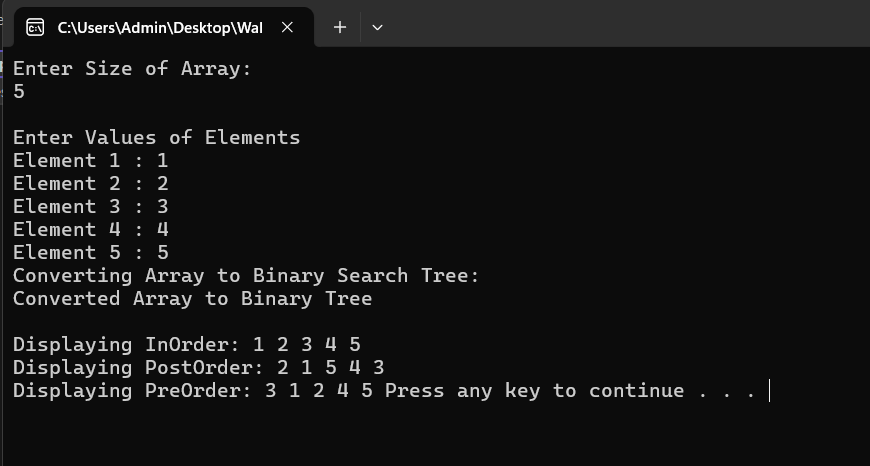
T1.DisplayPostOrder();

T1.DisplayPreOrder();

system("pause");

return 0;

}

****

**Q3:**

#include<iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

class BST {

Node\* root; // Root node of the BST

public:

BST() {

root = NULL;

}

void InsertNode(int num) {

Node\* newNode = new Node(num);

if (root == NULL) {

root = newNode;

}

else { // Tree is not empty

Node\* temp = root; // create a pointer to traverse the tree

while (true) {

if (num < temp->data) { // Left subtree

if (temp->left != NULL) {

temp = temp->left;

}

else {

temp->left = newNode;

return;

}

}

else if (num > temp->data) { // Right subtree

if (temp->right != NULL) {

temp = temp->right;

}

else {

temp->right = newNode;

return;

}

}

else {

cout << "Duplicate value found in tree.\n";

break;

}

}

}

}

bool FindValue(int val) {

Node\* temp = root;

while (temp != NULL) {

if (temp->data == val) {

return true;

}

else if (val < temp->data) {

temp = temp->left;

}

else {

temp = temp->right;

}

}

return false;

}

int callMaxSum() {

return maxPathsum(root);

}

int sum = 0; //global variable for using in function

int maxPathsum(Node\* root) {

if (root == NULL) { //base case

return 0;

}

int ans = -277000; //minimum value for comparing

int one = root->data; //store root node data

int two = maxPathsum(root->left) + root->data; //store root's left data + root's data

int three = maxPathsum(root->right) + root->data; ////store root's right data + root's data

int four = maxPathsum(root->left) + maxPathsum(root->right) + root->data; //store root's left data + root's right + root's data

int temp = max(max(one, two), max(three, four)); //calculate the maximum of 4 of the cases above

ans = max(ans, temp); //store the maximum one in ans

return ans;

}

void inorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

}

void PostorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

PostorderTraversal(root->left);

PostorderTraversal(root->right);

cout << root->data << " ";

}

}

void PreorderTraversal(Node\* root) {

Node\* temp = root;

if (root == NULL) {

return;

}

else {

cout << root->data << " ";

PreorderTraversal(root->left);

PreorderTraversal(root->right);

}

}

void DisplayInOrder() {

cout << "\nDisplaying InOrder: ";

inorderTraversal(root);

}

void DisplayPreOrder() {

cout << "\nDisplaying PreOrder: ";

PreorderTraversal(root);

}

void DisplayPostOrder() {

cout << "\nDisplaying PostOrder: ";

PostorderTraversal(root);

}

};

int main() {

int value = 0;

BST T1;

cout << "Enter Values to Store (-1 To Exit): " << endl;

while (value != -1) {

cin >> value;

if (value != -1) {

T1.InsertNode(value); // Insert only if value is not -1

}

}

T1.DisplayInOrder();

T1.DisplayPostOrder();

T1.DisplayPreOrder();

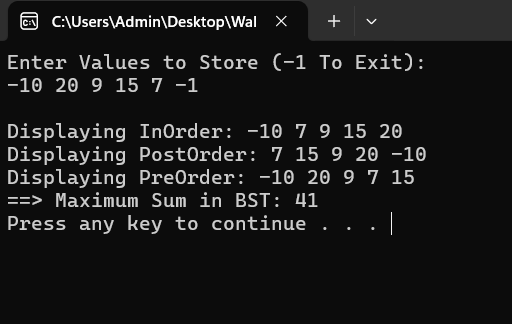
cout << "\n==> Maximum Sum in BST: ";

cout << T1.callMaxSum() << endl;

system("pause");

return 0;

}

****

**Q4:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = right = NULL;

}

};

class BST {

Node\* root; // root node of the BST

public:

Node\* head = NULL; // head pointer

Node\* prev = NULL; // previous pointer

BST() {

root = NULL;

}

void ConvertToDLL(Node\* root) { //converting bst to Doubly Linked list

if (root == NULL) {

return;

}

// Traversing left subtree

ConvertToDLL(root->left);

// Process current node

if (prev == nullptr) { //if its first node

head = root;

}

else {

//if its not first node

prev->right = root;

root->left = prev;

}

prev = root; //move prev pointer to point to newnode

// now traverse right subtree

ConvertToDLL(root->right);

}

// Function to convert BST to circular doubly linked list

Node\* bstToCDLL() {

if (root == NULL) {

return NULL;

}

// Convert BST to doubly linked list

ConvertToDLL(root);

// Make the doubly linked list circular

head->left = prev; // Linking head's left to the last node

prev->right = head; // Link last node's right to head

return head; //return root node which is head

}

void InsertNode(int num) {

Node\* newNode = new Node(num);

if (root == NULL) {

root = newNode;

}

else {

Node\* temp = root; // Pointer to traverse the tree

while (true) {

if (num < temp->data) { // Left subtree

if (temp->left != NULL) {

temp = temp->left;

}

else {

temp->left = newNode;

return;

}

}

else if (num > temp->data) { // Right subtree

if (temp->right != NULL) {

temp = temp->right;

}

else {

temp->right = newNode;

return;

}

}

else {

cout << "Duplicate value found in tree.\n";

break;

}

}

}

}

void inorderTraversal(Node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

void DisplayInOrder() {

cout << "\nDisplaying InOrder: ";

inorderTraversal(root);

}

// Print circularl doubly linked list

void DisplayCDLL(Node\* head) {

cout << "\nDisplaying Circular Doubly Linked List: ";

Node\* temp = head;

if (temp == NULL) return;

do {

cout << temp->data << " ";

temp = temp->right;

} while (temp != head);

cout << endl;

}

};

int main() {

int value = 0;

BST T1;

cout << "Enter Values to Store (-1 To Exit): " << endl;

while (value != -1) {

cin >> value;

if (value != -1) {

T1.InsertNode(value); // Insert only if value is not -1

}

}

T1.DisplayInOrder();

// Convert BST to circular DLL and displayingg

Node\* CDLLHead = T1.bstToCDLL(); //passing root for displaying the converted bst

T1.DisplayCDLL(CDLLHead);

return 0;

}

**A screenshot of a computer

Description automatically generated**

**Q5:**

#include<iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

class BST {

Node\* root; // Root node of the BST

int Ans; // Stores the result for kth smallest element search

public:

BST() {

root = NULL;

Ans = -1; // Initialize the answer to -1

}

void InsertNode(int num) {

Node\* newNode = new Node(num);

if (root == NULL) {

root = newNode;

}

else { // Tree is not empty

Node\* temp = root; // create a pointer to traverse the tree

while (true) {

if (num < temp->data) { // Left subtree

if (temp->left != NULL) {

temp = temp->left;

}

else {

temp->left = newNode;

return;

}

}

else if (num > temp->data) { // Right subtree

if (temp->right != NULL) {

temp = temp->right;

}

else {

temp->right = newNode;

return;

}

}

else {

cout << "Duplicate value found in tree.\n";

break;

}

}

}

}

void SearchKth(Node\* node, int& K) {

// Base case: if the current node is null then return

if (node==NULL) {

return;

};

//recusrively search the left part of the tree (inorder)

SearchKth(node->left, K);

K--; //after returning from left part subtract 1 from K

if (K == 0){ //check if k==0 means we found the smallest elements

Ans = node->data; //storing that node value in variable to return

return;

}

//if not found in the left subtree go to right sub tree recursively

SearchKth(node->right, K);

}

// callee function for SearchKth with parameter of element to find

int kthSmallest(int k) {

int K = k;

Ans = -1; // Reset Ans for a fresh search

SearchKth(root, K); //calling the function

return Ans; //returning the answer to main

}

void inorderTraversal(Node\* root) {

if (root == NULL) {

return;

}

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

void postorderTraversal(Node\* root) {

if (root == NULL) {

return;

}

postorderTraversal(root->left);

postorderTraversal(root->right);

cout << root->data << " ";

}

void preorderTraversal(Node\* root) {

if (root == NULL) {

return;

}

cout << root->data << " ";

preorderTraversal(root->left);

preorderTraversal(root->right);

}

void DisplayInOrder() {

cout << "\nDisplaying InOrder: ";

inorderTraversal(root);

cout << endl;

}

void DisplayPreOrder() {

cout << "\nDisplaying PreOrder: ";

preorderTraversal(root);

cout << endl;

}

void DisplayPostOrder() {

cout << "\nDisplaying PostOrder: ";

postorderTraversal(root);

cout << endl;

}

};

int main() {

int value = 0;

BST T1;

cout << "Enter Values to Store (-1 To Exit): " << endl;

while (value != -1) {

cin >> value;

if (value != -1) {

T1.InsertNode(value); // Insert only if value is not -1

}

}

T1.DisplayInOrder();

T1.DisplayPreOrder();

T1.DisplayPostOrder();

int k;

cout << "Enter k to find kth smallest element: ";

cin >> k;

int result = T1.kthSmallest(k);

if (result != -1) {

cout << "The " << k << "th smallest element is: " << result << endl;

}

else {

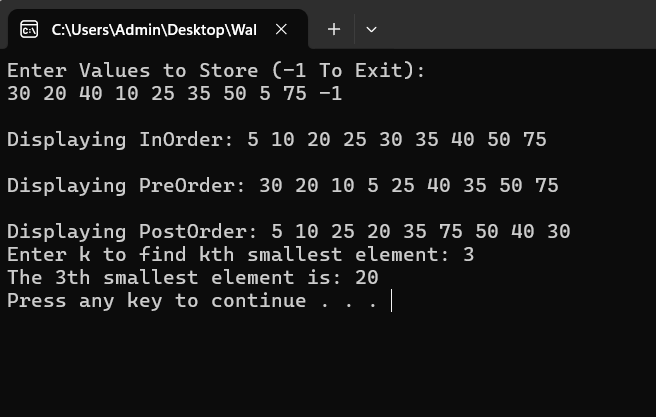
cout << "The tree has fewer than " << k << " elements.\n";

}

system("pause");

return 0;

}

****

**Q6:**

#include<iostream>

using namespace std;

class Node {

public:

int data;

int height;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

height = 1; // height of new node is 1 ; default

}

};

class BST {

Node\* root;

public:

BST() {

root = NULL;

}

int getHeight(Node\* node) {

if (node == NULL) {

return 0;

}

return node->height;

}

int getBalance(Node\* node) {

if (node == NULL) return 0;

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

}

int maxCheck(int a, int b) {

if (a > b) {

return a;

}

return b;

}

Node\* rightRotation(Node\* root) {

Node\* child = root->left;

Node\* childRight = child->right;

child->right = root;

root->left = childRight;

// update the height again

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

child->height = 1 + maxCheck(getHeight(child->left), getHeight(child->right));

return child;

}

Node\* leftRotation(Node\* root) {

Node\* child = root->right;

Node\* childLeft = child->left;

child->left = root;

root->right = childLeft;

// update the height again

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

child->height = 1 + maxCheck(getHeight(child->left), getHeight(child->right));

return child;

}

Node\* InsertNode(Node\* root, int num) {

if (root == NULL)

return new Node(num);

if (num < root->data)

root->left = InsertNode(root->left, num);

else if (num > root->data)

root->right = InsertNode(root->right, num);

else {

cout << "Duplicate value found in tree.\n";

return root;

}

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

int balance = getBalance(root);

// Left Left Case

if (balance > 1 && num < root->left->data)

return rightRotation(root);

// Right Right Case

if (balance < -1 && num > root->right->data)

return leftRotation(root);

// Left Right Case

if (balance > 1 && num > root->left->data) {

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

return rightRotation(root);

}

// Right Left Case

if (balance < -1 && num < root->right->data) {

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

return leftRotation(root);

}

return root;

}

void InsertNode(int num) {

root = InsertNode(root, num);

}

void inorderTraversal(Node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

void DisplayInOrder() {

cout << "Displaying InOrder: ";

inorderTraversal(root);

cout << endl;

}

};

int main() {

int value = 0;

BST T1;

cout << "Enter Values to Store (-1 To Exit): " << endl;

while (value != -1) {

cin >> value;

if (value != -1) {

T1.InsertNode(value);

}

}

T1.DisplayInOrder();

return 0;

}

**A screenshot of a computer

Description automatically generated**



**Q7:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int d) : data(d), left(NULL), right(NULL) {}

};

class AVL {

Node\* root;

int getHeight(Node\* node) {

if (node == NULL)

return -1;

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

if (leftHeight > rightHeight) {

return leftHeight + 1;

}

else {

return rightHeight + 1;

}

}

int getBalance(Node\* node) {

if (node == NULL)

return 0;

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

}

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

return x;

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

return y;

}

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

if (node == NULL) {

node = new Node(key);

return node;

}

if (key < node->data){

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

}

else if (key > node->data){

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

}

else

return node; // Duplicate keys

int balance = getBalance(node);

// Left Left Case

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

return rotateRight(node);

}

// Right Right Case

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

return rotateLeft(node);

}

// Left Right Case

if (balance > 1 && key > node->left->data) {

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

return rotateRight(node);

}

// Right Left Case

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

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

return rotateLeft(node);

}

return node;

}

void display(Node\* node) {

if (node != NULL) { // Inorder

display(node->left);

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

display(node->right);

}

}

Node\* join(Node\* root1, Node\* root2) {

if (root1 == NULL){

return root2;

}

if (root2 == NULL){

return root1;

}

root2 = insert(root2, root1->data);

root2 = join(root1->left, root2);

root2 = join(root1->right, root2);

return root2;

}

public:

AVL() : root(NULL) {}

void insert(int key) {

root = insert(root, key);

}

void display() {

cout << "AVL Tree (Inorder): ";

display(root);

cout << endl;

}

Node\* getRoot() {

return root;

}

void join(AVL& other) {

root = join(this->root, other.root);

}

};

int main() {

AVL T1, T2;

int valuesT1[] = { 1,2,4,5,3,6,7 };

int const n1 = 6;

for (int i = 0; i < n1; i++) {

T1.insert(valuesT1[i]);

}

int valuesT2[] = { 4,5,2,6,7,3,1 };

int const n2 = 5;

for (int i = 0; i < n2; i++) {

T2.insert(valuesT2[i]);

}

cout << "T1: ";

T1.display();

cout << "T2: ";

T2.display();

// Join T1 and T2

T1.join(T2);

cout << "Joined AVL Tree: ";

T1.display();

system("pause");

return 0;

}

**A screenshot of a computer

Description automatically generated**

**Q9:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

int search(int postorder[], int key, int start, int end) {

for (int i = start; i <= end; i++) {

if (postorder[i] == key) {

return i;

}

}

return -1;

}

Node\* constructing(int preorder[], int postorder[], int& preIdx, int start, int end) {

if (start > end) {

return NULL;

}

if (start == end) {

return new Node(preorder[preIdx++]);

}

Node\* root = new Node(preorder[preIdx++]);

int idx2 = search(postorder, preorder[preIdx], start, end);

root->left = constructing(preorder, postorder, preIdx, start, idx2);

root->right = constructing(preorder, postorder, preIdx, idx2 + 1, end - 1);

return root;

}

Node\* constructFromPrePost(int preorder[], int postorder[], int size) {

int preIdx = 0;

return constructing(preorder, postorder, preIdx, 0, size - 1);

}

void inorderTraversal(Node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

int main() {

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

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

int size = sizeof(preorder) / sizeof(preorder[0]);

Node\* root = constructFromPrePost(preorder, postorder, size);

cout << "Inorder Traversal of the constructed tree: ";

inorderTraversal(root); // Expected output: 4 2 5 1 6 3 7

cout << endl;

return 0;

}

**A screenshot of a computer

Description automatically generated**

**Q8:**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class SongNode {**

**public:**

**string songName;**

**SongNode\* next;**

**SongNode(string name) : songName(name), next(nullptr) {}**

**};**

**class Node {**

**public:**

**string songName;**

**int frequency;**

**int height;**

**SongNode\* songList; // To store songs with the same frequency**

**Node\* left;**

**Node\* right;**

**Node(string name, int freq) : songName(name), frequency(freq), height(1), left(nullptr), right(nullptr), songList(nullptr) {}**

**};**

**int FindMax(int a, int b) {**

**if (a > b) {**

**return a;**

**} else {**

**return b;**

**}**

**}**

**int getHeight(Node\* node) {**

**if (node == nullptr) {**

**return 0;**

**}**

**return node->height;**

**}**

**int getBalance(Node\* node) {**

**if (node == nullptr) {**

**return 0;**

**}**

**return getHeight(node->left) - getHeight(node->right);**

**}**

**Node\* rightRotate(Node\* y) { //right rotate**

**Node\* x = y->left;**

**Node\* T2 = x->right;**

**x->right = y;**

**y->left = T2;**

**// Update heights**

**y->height = FindMax(getHeight(y->left), getHeight(y->right)) + 1;**

**x->height = FindMax(getHeight(x->left), getHeight(x->right)) + 1;**

**return x; // New root**

**}**

**// Left rotate**

**Node\* leftRotate(Node\* x) {**

**Node\* y = x->right;**

**Node\* T2 = y->left;**

**y->left = x;**

**x->right = T2;**

**// Update heights**

**x->height = FindMax(getHeight(x->left), getHeight(x->right)) + 1;**

**y->height = FindMax(getHeight(y->left), getHeight(y->right)) + 1;**

**return y; // New root**

**}**

**Node\* insertSong(Node\* node, const string& songName, int frequency) {**

**if (node == nullptr) {**

**return new Node(songName, frequency);**

**}**

**if (songName < node->songName) {**

**node->left = insertSong(node->left, songName, frequency);**

**} else if (songName > node->songName) {**

**node->right = insertSong(node->right, songName, frequency);**

**} else {**

**// If song already exists, add it to the linked list of songs with the same frequency**

**SongNode\* newSong = new SongNode(songName);**

**newSong->next = node->songList;**

**node->songList = newSong;**

**return node;**

**}**

**node->height = 1 + FindMax(getHeight(node->left), getHeight(node->right));**

**int balance = getBalance(node);**

**// Balance the node if needed**

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

**return rightRotate(node);**

**}**

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

**return leftRotate(node);**

**}**

**if (balance > 1 && songName > node->left->songName) {**

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

**return rightRotate(node);**

**}**

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

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

**return leftRotate(node);**

**}**

**return node;**

**}**

**// Play a song (increasing the frequency)**

**Node\* playSong(Node\* node, const string& songName) {**

**if (node == nullptr) {**

**return nullptr;**

**}**

**if (songName < node->songName) {**

**node->left = playSong(node->left, songName);**

**} else if (songName > node->songName) {**

**node->right = playSong(node->right, songName);**

**} else {**

**// Song found, increase its frequency**

**node->frequency++;**

**// Rebalance the tree**

**int balance = getBalance(node);**

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

**return rightRotate(node);**

**}**

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

**return leftRotate(node);**

**}**

**if (balance > 1 && songName > node->left->songName) {**

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

**return rightRotate(node);**

**}**

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

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

**return leftRotate(node);**

**}**

**}**

**return node;**

**}**

**// Inorder traversal to display songs**

**void inorderTraversal(Node\* root) {**

**if (root == nullptr) {**

**return;**

**}**

**inorderTraversal(root->left);**

**cout << "Song: " << root->songName << ", Frequency: " << root->frequency << endl;**

**// Display songs in the linked list with the same frequency**

**SongNode\* temp = root->songList;**

**while (temp != nullptr) {**

**cout << "Song: " << temp->songName << ", Frequency: " << root->frequency << endl;**

**temp = temp->next;**

**}**

**inorderTraversal(root->right);**

**}**

**int main() {**

**Node\* root = nullptr;**

**// Initial songs and frequencies**

**string songs[] = {"Song\_A", "Song\_B", "Song\_C", "Song\_D", "Song\_E", "Song\_F", "Song\_G", "Song\_H", "Song\_I", "Song\_J", "Song\_K"};**

**int frequencies[] = {1, 5, 9, 2, 4, 6, 8, 3, 7, 9, 5};**

**// Insert initial songs into the AVL tree**

**for (int i = 0; i < 11; i++) {**

**root = insertSong(root, songs[i], frequencies[i]);**

**}**

**int choice;**

**do { //menu driven program**

**cout << "\nMenu:" << endl;**

**cout << "1. Add Song" << endl;**

**cout << "2. Play Song" << endl;**

**cout << "3. Display Songs" << endl;**

**cout << "4. Exit" << endl;**

**cout << "Enter your choice: ";**

**cin >> choice;**

**string songName;**

**switch (choice) {**

**case 1:**

**cout << "Enter song name: ";**

**cin >> songName;**

**cout << "Enter frequency: ";**

**int frequency;**

**cin >> frequency;**

**root = insertSong(root, songName, frequency);**

**break;**

**case 2:**

**cout << "Enter song name to play: ";**

**cin >> songName;**

**root = playSong(root, songName);**

**break;**

**case 3:**

**cout << "Songs in the tree:" << endl;**

**inorderTraversal(root);**

**break;**

**case 4:**

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

**break;**

**default:**

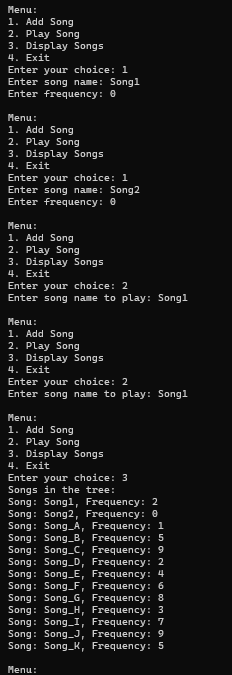
**cout << "Invalid choice!" << endl;**

**}**

**} while (choice != 4);**

**return 0;**

**}**

****