**classBst.h**

**Code:**

#include<iostream>

#include<string>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val);

};

class Base {

protected:

Node\* root;

public:

Base();

virtual void insert(int val) = 0;

virtual void printInOrder() = 0;

virtual void printPostOrder() = 0;

virtual void printPreOrder() = 0;

virtual bool find(int val) = 0;

virtual void insertFromArray(int arr[], int size) = 0;

virtual void pathWithHighestSum() = 0;

virtual void convertToDoubly() = 0;

virtual int getKthElement(int k) = 0;

virtual void constructFromPrePost(int preorder[], int postorder[], int size) = 0;

};

class Bst :public Base {

public:

Bst();

void insert(int val);

void printInOrder();

void printPostOrder();

void printPreOrder();

void insertFromArray(int arr[], int size);

bool find(int val);

void pathWithHighestSum();

void convertToDoubly();

int getKthElement(int k);

void constructFromPrePost(int preorder[], int postorder[], int size);

private:

void insertDataHelper(int val, Node\*& temp);

void inOrderHelper(Node\* temp);

void postOrderHelper(Node\* temp);

void preOrderHelper(Node\* temp);

bool findNodeHelper(int val, Node\* temp);

int pathWithHighestSumHelper(Node\* temp, int& maxSum, string& str);

void convertToDoublyHelper(Node\* temp, Node\*& head, Node\*& tail);

int getKthElementHelper(int k, Node\* temp, int& flag);

Node\* buildTree(int preorder[], int postorder[], int& preIndex, int postStart, int postEnd, int size);

};

**methodsBst.cpp**

**Code:**

#include"classBST.h"

Node::Node(int val) {

data = val;

left = nullptr;

right = nullptr;

}

Base::Base() {

root = nullptr;

}

Bst::Bst():Base() {}

void Bst::printInOrder() {

inOrderHelper(root);

cout << endl;

}

void Bst::insert(int val) {

insertDataHelper(val, root);

}

void Bst::printPostOrder() {

postOrderHelper(root);

cout << endl;

}

void Bst::printPreOrder() {

preOrderHelper(root);

cout << endl;

}

void Bst::insertFromArray(int arr[], int size) {

int flag = size;

for (int i = 0; i < (size / 2) + 1; i++) {

insertDataHelper(arr[i], root);

if (flag != i) {

insertDataHelper(arr[--flag], root);

}

}

}

bool Bst::find(int val) {

return findNodeHelper(val, root);

}

void Bst::pathWithHighestSum() {

string str = "";

int maxSum = 0;

pathWithHighestSumHelper(root, maxSum, str);

cout << "Highest Sum : " << maxSum << endl;

cout << "Path : " << str << endl;

}

void Bst::convertToDoubly() {

Node\* head = nullptr;

Node\* tail = nullptr;

convertToDoublyHelper(root, head, tail);

if (head != nullptr && tail != nullptr) {

head->left = tail;

tail->right = head;

}

else {

cout << "The Tree is Empty" << endl;

return;

}

Node\* current = head;

cout << "Doubly List: ";

do {

cout << current->data << " <-> ";

current = current->right;

} while (current != head);

cout << "(Back to Head)" << endl;

}

int Bst::getKthElement(int k) {

int flag = 0;

return getKthElementHelper(k, root, flag);

}

void Bst::constructFromPrePost(int preorder[], int postorder[], int size) {

int preIndex = 0;

root = buildTree(preorder, postorder, preIndex, 0, size - 1, size);

}

void Bst::insertDataHelper(int val, Node\*& temp) {

if (temp == nullptr) {

temp = new Node(val);

return;

}

else {

if (val < temp->data) {

insertDataHelper(val, temp->left);

}

else if (val > temp->data) {

insertDataHelper(val, temp->right);

}

else if (val == temp->data) {

return;

}

}

}

void Bst::inOrderHelper(Node\* temp) {

if (temp != nullptr) {

inOrderHelper(temp->left);

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

inOrderHelper(temp->right);

}

}

void Bst::postOrderHelper(Node\* temp) {

if (temp != nullptr) {

postOrderHelper(temp->left);

postOrderHelper(temp->right);

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

}

}

void Bst::preOrderHelper(Node\* temp) {

if (temp != nullptr) {

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

preOrderHelper(temp->left);

preOrderHelper(temp->right);

}

}

bool Bst::findNodeHelper(int val, Node\* temp) {

if (temp != nullptr) {

if (val == temp->data) {

return true;

}

else if (val < temp->data) {

return findNodeHelper(val, temp->left);

}

else if (val > temp->data) {

return findNodeHelper(val, temp->right);

}

}

return false;

}

int Bst::pathWithHighestSumHelper(Node\* temp, int& maxSum, string& str) {

if (temp == nullptr) {

return 0;

}

int leftSum = max(0, pathWithHighestSumHelper(temp->left, maxSum, str));

int rightSum = max(0, pathWithHighestSumHelper(temp->right, maxSum, str));

char ch = temp->data + '0';

if (str.empty() == true) {

str = str + ch;

}

else {

str = str + "->" + ch;

}

int currentPathSum = temp->data + leftSum + rightSum;

maxSum = max(maxSum, currentPathSum);

return temp->data + max(leftSum, rightSum);

}

void Bst::convertToDoublyHelper(Node\* temp, Node\*& head, Node\*& tail) {

if (temp == nullptr) {

return;

}

convertToDoublyHelper(temp->left, head, tail);

if (head == nullptr) {

head = temp;

}

else {

tail->right = temp;

temp->left = tail;

}

tail = temp;

convertToDoublyHelper(temp->right, head, tail);

}

int Bst::getKthElementHelper(int k, Node\* temp, int& flag) {

if (!temp) {

return -1;

}

int left = getKthElementHelper(k, temp->left, flag);

if (left != -1) {

return left;

}

flag++;

if (flag == k) {

return temp->data;

}

return getKthElementHelper(k, temp->right, flag);

}

Node\* Bst::buildTree(int preorder[], int postorder[], int& preIndex, int postStart, int postEnd, int size) {

if (preIndex >= size || postStart > postEnd) {

return nullptr;

}

Node\* temp = new Node(preorder[preIndex++]);

if (postStart == postEnd || preIndex >= size) {

return temp;

}

int postindex;

for (postindex = postStart; postindex <= postEnd; postindex++) {

if (postorder[postindex] == preorder[preIndex]) {

break;

}

}

if (postindex <= postEnd) {

temp->left = buildTree(preorder, postorder, preIndex, postStart, postindex, size);

temp->right = buildTree(preorder, postorder, preIndex, postindex + 1, postEnd - 1, size);

}

return temp;

}

**mainBST.cpp**

**Code:**

#include"classBST.h"

int main() {

Bst obj;

for (int i = 1; i <= 10; i++) {

cout << "Input : " << i << endl;

obj.insert(i);

}

cout << "InOrder : ";

obj.printInOrder();

cout << "PostOrder : ";

obj.printPostOrder();

cout << "PreOrder : ";

obj.printPreOrder();

cout << "Find(5) : " << obj.find(5) << endl;

cout << "k(2) : " << obj.getKthElement(2) << endl;

cout << "Path With Highest Sum" << endl;

obj.pathWithHighestSum();

cout << endl;

int arr[] = { 90,80,70,60,50,40,30,20,10 };

cout << "Input Array : ";

for (int i = 0; i < 9; i++) {

cout << arr[i] << " ";

}

cout << endl;

obj.insertFromArray(arr, 9);

cout << "InOrder : ";

obj.printInOrder();

obj.convertToDoubly();

cout << endl;

return 0;

}

**Question#1**

**Code:**

virtual void insert(int val) = 0;

virtual void printInOrder() = 0;

virtual void printPostOrder() = 0;

virtual void printPreOrder() = 0;

virtual bool find(int val) = 0;

void printInOrder();

void printPostOrder();

void printPreOrder();

bool find(int val);

void insertDataHelper(int val, Node\*& temp);

void inOrderHelper(Node\* temp);

void postOrderHelper(Node\* temp);

void preOrderHelper(Node\* temp);

void Bst::printInOrder() {

inOrderHelper(root);

cout << endl;

}

void Bst::insert(int val) {

insertDataHelper(val, root);

}

void Bst::printPostOrder() {

postOrderHelper(root);

cout << endl;

}

void Bst::printPreOrder() {

preOrderHelper(root);

cout << endl;

}

bool Bst::find(int val) {

return findNodeHelper(val, root);

}

void Bst::insertDataHelper(int val, Node\*& temp) {

if (temp == nullptr) {

temp = new Node(val);

return;

}

else {

if (val < temp->data) {

insertDataHelper(val, temp->left);

}

else if (val > temp->data) {

insertDataHelper(val, temp->right);

}

else if (val == temp->data) {

return;

}

}

}

void Bst::inOrderHelper(Node\* temp) {

if (temp != nullptr) {

inOrderHelper(temp->left);

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

inOrderHelper(temp->right);

}

}

void Bst::postOrderHelper(Node\* temp) {

if (temp != nullptr) {

postOrderHelper(temp->left);

postOrderHelper(temp->right);

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

}

}

void Bst::preOrderHelper(Node\* temp) {

if (temp != nullptr) {

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

preOrderHelper(temp->left);

preOrderHelper(temp->right);

}

}

bool Bst::findNodeHelper(int val, Node\* temp) {

if (temp != nullptr) {

if (val == temp->data) {

return true;

}

else if (val < temp->data) {

return findNodeHelper(val, temp->left);

}

else if (val > temp->data) {

return findNodeHelper(val, temp->right);

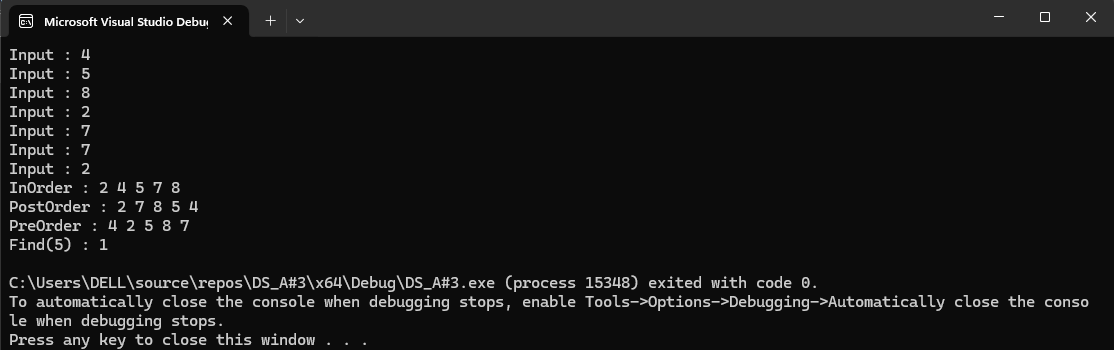
}

}

return false;

}

**Output:**

****

**Question#2**

**Code:**

virtual void insertFromArray(int arr[], int size) = 0;

void insertFromArray(int arr[], int size);

void Bst::insertFromArray(int arr[], int size) {

int flag = size;

for (int i = 0; i < (size / 2) + 1; i++) {

insertDataHelper(arr[i], root);

if (flag != i) {

insertDataHelper(arr[--flag], root);

}

}

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Question#3**

**Code:**

void pathWithHighestSum();

void pathWithHighestSum();

int pathWithHighestSumHelper(Node\* temp, int& maxSum, string& str);

void Bst::pathWithHighestSum() {

string str = "";

int maxSum = 0;

pathWithHighestSumHelper(root, maxSum, str);

cout << "Highest Sum : " << maxSum << endl;

cout << "Path : " << str << endl;

}

int Bst::pathWithHighestSumHelper(Node\* temp, int& maxSum, string& str) {

if (temp == nullptr) {

return 0;

}

int leftSum = max(0, pathWithHighestSumHelper(temp->left, maxSum, str));

int rightSum = max(0, pathWithHighestSumHelper(temp->right, maxSum, str));

char ch = temp->data + '0';

if (str.empty() == true) {

str = str + ch;

}

else {

str = str + "->" + ch;

}

int currentPathSum = temp->data + leftSum + rightSum;

maxSum = max(maxSum, currentPathSum);

return temp->data + max(leftSum, rightSum);

}

**Output:**

**A screen shot of a computer

Description automatically generated**

**Question#4**

**Code:**

virtual void convertToDoubly() = 0;

void convertToDoubly();

void convertToDoublyHelper(Node\* temp, Node\*& head, Node\*& tail);

void Bst::convertToDoubly() {

Node\* head = nullptr;

Node\* tail = nullptr;

convertToDoublyHelper(root, head, tail);

if (head != nullptr && tail != nullptr) {

head->left = tail;

tail->right = head;

}

else {

cout << "The Tree is Empty" << endl;

return;

}

Node\* current = head;

cout << "Doubly List: ";

do {

cout << current->data << " <-> ";

current = current->right;

} while (current != head);

cout << "(Back to Head)" << endl;

}

void Bst::convertToDoublyHelper(Node\* temp, Node\*& head, Node\*& tail) {

if (temp == nullptr) {

return;

}

convertToDoublyHelper(temp->left, head, tail);

if (head == nullptr) {

head = temp;

}

else {

tail->right = temp;

temp->left = tail;

}

tail = temp;

convertToDoublyHelper(temp->right, head, tail);

}

**Output:**

**A screenshot of a computer program

Description automatically generated**

**Question#5**

**Code:**

virtual int getKthElement(int k) = 0;

int getKthElement(int k);

int getKthElementHelper(int k, Node\* temp, int& flag);

int Bst::getKthElement(int k) {

int flag = 0;

return getKthElementHelper(k, root, flag);

}

int Bst::getKthElementHelper(int k, Node\* temp, int& flag) {

if (!temp) {

return -1;

}

int left = getKthElementHelper(k, temp->left, flag);

if (left != -1) {

return left;

}

flag++;

if (flag == k) {

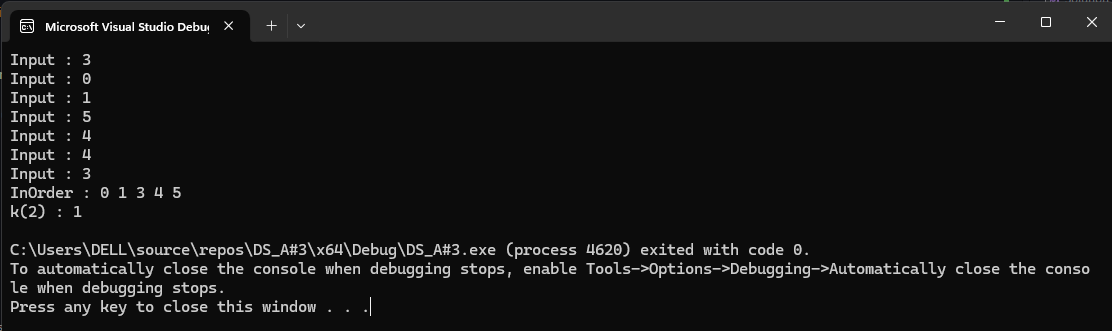
return temp->data;

}

return getKthElementHelper(k, temp->right, flag);

}

**Output:**

****

**Question#6,7**

**classAvl.h**

**Code:**

#include<iostream>

using namespace std;

template<class T>

struct Node {

T data;

Node\* left;

Node\* right;

Node(T val);

};

template<class T>

class Base {

protected:

Node<T>\* root;

public:

Base();

virtual void insert(T val) = 0;

virtual void printInOrder() = 0;

virtual void deleteNode(T val) = 0;

//virtual void merge(Bst<T>& obj) = 0;

};

template<class T>

class Bst :public Base<T> {

public:

Bst();

void insert(T val);

void printInOrder();

void deleteNode(T val);

// void merge(Bst<T>obj);

private:

T max(T a, T b);

int getHeight(Node<T>\* temp);

int calculateBF(Node<T>\* temp);

void rotateRight(Node<T>\*& temp);

void rotateLeft(Node<T>\*& temp);

void rotateLeftRight(Node<T>\*& temp);

void rotateRightLeft(Node<T>\*& temp);

void insertDataHelper(T val, Node<T>\*& temp);

void deleteNodeHelper(T val, Node<T>\*& temp);

void inOrderHelper(Node<T>\* temp);

//void mergeHelper(Node<T>\* temp);

};

**methodsAvl.cpp**

**Code:**

#include"classAvl.h"

template<class T>

Node<T>::Node(T val) {

data = val;

left = nullptr;

right = nullptr;

}

template<class T>

Base<T>::Base() {

root = nullptr;

}

template<class T>

Bst<T>::Bst():Base() {}

template<class T>

void Bst<T>::insert(T val) {

insertDataHelper(val, this->root);

}

template<class T>

void Bst<T>::printInOrder() {

inOrderHelper(this->root);

cout << endl;

}

template<class T>

void Bst<T>::deleteNode(T val) {

deleteNodeHelper(val, this->root);

}

//template<class T>

//void Bst<T>::merge(Bst<T> obj) {

// mergeHelper(obj.root);

//}

template<class T>

T Bst<T>::max(T a, T b) {

if (a > b) {

return a;

}

return b;

}

template<class T>

int Bst<T>::getHeight(Node<T>\* temp) {

if (temp == nullptr) {

return -1;

}

return 1 + max(getHeight(temp->left), getHeight(temp->right));

}

template<class T>

int Bst<T>::calculateBF(Node<T>\* temp) {

int leftHeight = getHeight(temp->left);

int rightHeight = getHeight(temp->right);

return leftHeight - rightHeight;

}

template<class T>

void Bst<T>::rotateRight(Node<T>\*& temp) {

Node<T>\* ptr = temp->right;

temp->right = ptr->left;

ptr->left = temp;

temp = ptr;

}

template<class T>

void Bst<T>::rotateLeft(Node<T>\*& temp) {

Node<T>\* ptr = temp->left;

temp->left = ptr->right;

ptr->right = temp;

temp = ptr;

}

template<class T>

void Bst<T>::rotateLeftRight(Node<T>\*& temp) {

Node<T>\* ptr = temp->right;

Node<T>\* child = ptr->left;

ptr->left = child->right;

child->right = ptr;

temp->right = child;

rotateRight(temp);

}

template<class T>

void Bst<T>::rotateRightLeft(Node<T>\*& temp) {

Node<T>\* ptr = temp->left;

Node<T>\* child = ptr->right;

ptr->right = child->left;

child->left = ptr;

temp->left = child;

rotateLeft(temp);

}

template<class T>

void Bst<T>::insertDataHelper(T val, Node<T>\*& temp) {

if (temp == nullptr) {

temp = new Node<T>(val);

return;

}

else {

if (val < temp->data) {

insertDataHelper(val, temp->left);

}

else if (val > temp->data) {

insertDataHelper(val, temp->right);

}

else if (val == temp->data) {

return;

}

}

int bf = calculateBF(temp);

if (bf > 1) {

if (val < temp->left->data) {

cout << "Rotating Left" << endl;

rotateLeft(temp);

}

else {

cout << "Rotating Left Then Right" << endl;

rotateLeftRight(temp);

}

}

else if (bf < -1) {

if (val > temp->right->data) {

cout << "Rotating Right" << endl;

rotateRight(temp);

}

else {

cout << "Rotating Right Then Left" << endl;

rotateRightLeft(temp);

}

}

}

template<class T>

void Bst<T>::deleteNodeHelper(T val, Node<T>\*& temp) {

if (temp == nullptr) {

return;

}

if (val < temp->data) {

deleteNodeHelper(val, temp->left);

}

else if (val > temp->data) {

deleteNodeHelper(val, temp->right);

}

else {

if (temp->left == nullptr && temp->right == nullptr) {

delete temp;

temp = nullptr;

}

else if (temp->left == nullptr) {

Node<T>\* toDelete = temp;

temp = temp->right;

delete toDelete;

}

else if (temp->right == nullptr) {

Node<T>\* toDelete = temp;

temp = temp->left;

delete toDelete;

}

else {

Node<T>\* successor = temp->right;

while (successor->left != nullptr) {

successor = successor->left;

}

temp->data = successor->data;

deleteNodeHelper(successor->data, temp->right);

}

}

if (temp == nullptr) return;

int bf = calculateBF(temp);

if (bf > 1) {

if (calculateBF(temp->left) >= 0) {

cout << "Rotating Left" << endl;

rotateLeft(temp);

}

else {

cout << "Rotating Left Then Right" << endl;

rotateLeftRight(temp);

}

}

else if (bf < -1) {

if (calculateBF(temp->right) <= 0) {

cout << "Rotating Right" << endl;

rotateRight(temp);

}

else {

cout << "Rotating Right Then Left" << endl;

rotateRightLeft(temp);

}

}

}

template<class T>

void Bst<T>::inOrderHelper(Node<T>\*temp) {

if (temp != nullptr) {

inOrderHelper(temp->left);

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

inOrderHelper(temp->right);

}

}

//template<class T>

//void Bst<T>::mergeHelper(Node<T>\* temp) {

// if (temp == nullptr) {

// return;

// }

// mergeHelper(temp->left);

// this->root = insertDataHelper(this->root, temp->data);

// mergeHelper(temp->right);

//}

**mainAvl.cpp**

**Code:**

#include"classAvl.h"

int main() {

Bst<int> obj;

for (int i = 1; i <= 10; i++) {

cout << "Input : " << i << endl;

obj.insert(i);

}

cout << "InOrder : ";

obj.printInOrder();

cout << "Delete(5) : ";

obj.deleteNode(5);

obj.printInOrder();

cout << endl;

cout << "Delete(2) : ";

obj.deleteNode(2);

obj.printInOrder();

cout << endl;

cout << "Delete(4) : ";

obj.deleteNode(4);

obj.printInOrder();

cout << endl;

cout << "Delete(1) : ";

obj.deleteNode(1);

obj.printInOrder();

cout << endl;

return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Question#8**

**Code:**

#include <iostream>

#include <string>

using namespace std;

struct Node {

int freq;

string songName;

Node\* left;

Node\* right;

Node\* next;

Node(int val, string song) : freq(val), songName(song), left(nullptr), right(nullptr), next(nullptr) {}

};

class Base {

protected:

Node\* root;

public:

Base() {

root = nullptr;

}

virtual void insert(int val, string song) = 0;

virtual void playSong(string song) = 0;

virtual void printInOrder() = 0;

};

class AVL : public Base {

public:

AVL() : Base() {}

void insert(int val, string song) {

insertDataHelper(val, song, this->root);

}

void playSong(string song) {

playSongHelper(song, this->root);

}

void printInOrder() {

inOrderHelper(this->root);

cout << endl;

}

private:

int max(int a, int b) {

return (a > b) ? a : b;

}

int getHeight(Node\* temp) {

if (temp == nullptr) {

return -1;

}

return 1 + max(getHeight(temp->left), getHeight(temp->right));

}

int calculateBF(Node\* temp) {

return (temp == nullptr) ? 0 : getHeight(temp->left) - getHeight(temp->right);

}

void rotateRight(Node\*& temp) {

Node\* ptr = temp->left;

temp->left = ptr->right;

ptr->right = temp;

temp = ptr;

}

void rotateLeft(Node\*& temp) {

Node\* ptr = temp->right;

temp->right = ptr->left;

ptr->left = temp;

temp = ptr;

}

void rotateLeftRight(Node\*& temp) {

rotateLeft(temp->left);

rotateRight(temp);

}

void rotateRightLeft(Node\*& temp) {

rotateRight(temp->right);

rotateLeft(temp);

}

void insertDataHelper(int val, string song, Node\*& temp) {

if (temp == nullptr) {

temp = new Node(val, song);

return;

}

else {

if (val < temp->freq) {

insertDataHelper(val, song, temp->left);

}

else if (val > temp->freq) {

insertDataHelper(val, song, temp->right);

}

else {

Node\* newNode = new Node(val, song);

newNode->next = temp->next;

temp->next = newNode;

return;

}

}

int bf = calculateBF(temp);

if (bf > 1) {

if (val < temp->left->freq) {

rotateRight(temp);

}

else {

rotateLeftRight(temp);

}

}

else if (bf < -1) {

if (val > temp->right->freq) {

rotateLeft(temp);

}

else {

rotateRightLeft(temp);

}

}

}

void playSongHelper(string song, Node\*& temp) {

if (temp == nullptr) {

return;

}

if (temp->songName == song) {

temp->freq++;

return;

}

else if (song < temp->songName) {

playSongHelper(song, temp->left);

}

else {

playSongHelper(song, temp->right);

}

}

void inOrderHelper(Node\* temp) {

if (temp != nullptr) {

inOrderHelper(temp->left);

cout << "Song Name : " << temp->songName << " Frequency : " << temp->freq << endl;

if (temp->next != nullptr) {

inOrderHelper(temp->next);

}

inOrderHelper(temp->right);

}

}

};

int main() {

AVL obj;

obj.insert(1, "Song\_A");

obj.insert(5, "Song\_B");

obj.insert(9, "Song\_C");

obj.insert(2, "Song\_D");

obj.insert(4, "Song\_E");

obj.insert(6, "Song\_F");

obj.insert(8, "Song\_G");

obj.insert(3, "Song\_H");

obj.insert(7, "Song\_I");

obj.insert(9, "Song\_J");

obj.insert(5, "Song\_K");

int choice;

string song;

bool flag = true;

while (flag) {

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

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

cout << "3. Print Songs in Order" << endl;

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

cout << "Enter your choice : ";cin >> choice;

switch (choice) {

case 1:

cout << "Enter Song Name : ";cin >> song;

int freq;

cout << "Enter Frequency : "; cin >> freq;

obj.insert(freq, song);

break;

case 2:

cout << "Enter Song Name to Play : ";cin >> song;

obj.playSong(song);

break;

case 3:

cout << "InOrder : " << endl;

obj.printInOrder();

break;

case 4:

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

flag = false;

break;

default:

cout << "Invalid Input" << endl;

}

}

return 0;

}

**Output:**

****

**Question#9**

**Code:**

virtual void constructFromPrePost(int preorder[], int postorder[], int size) = 0;

void constructFromPrePost(int preorder[], int postorder[], int size);

Node\* buildTree(int preorder[], int postorder[], int& preIndex, int postStart, int postEnd, int size);

void Bst::constructFromPrePost(int preorder[], int postorder[], int size) {

int preIndex = 0;

root = buildTree(preorder, postorder, preIndex, 0, size - 1, size);

}

Node\* Bst::buildTree(int preorder[], int postorder[], int& preIndex, int postStart, int postEnd, int size) {

if (preIndex >= size || postStart > postEnd) {

return nullptr;

}

Node\* temp = new Node(preorder[preIndex++]);

if (postStart == postEnd || preIndex >= size) {

return temp;

}

int postindex;

for (postindex = postStart; postindex <= postEnd; postindex++) {

if (postorder[postindex] == preorder[preIndex]) {

break;

}

}

if (postindex <= postEnd) {

temp->left = buildTree(preorder, postorder, preIndex, postStart, postindex, size);

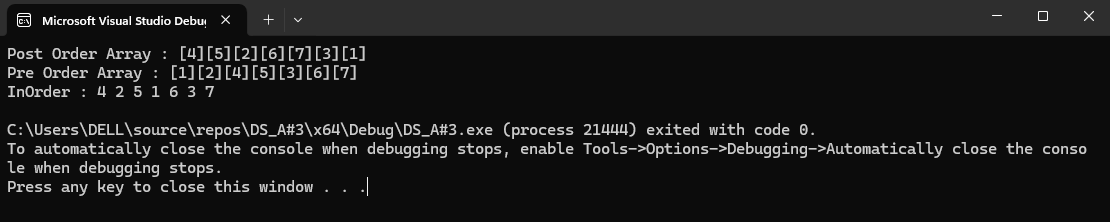
temp->right = buildTree(preorder, postorder, preIndex, postindex + 1, postEnd - 1, size);

}

return temp;

}

**Output:**

****