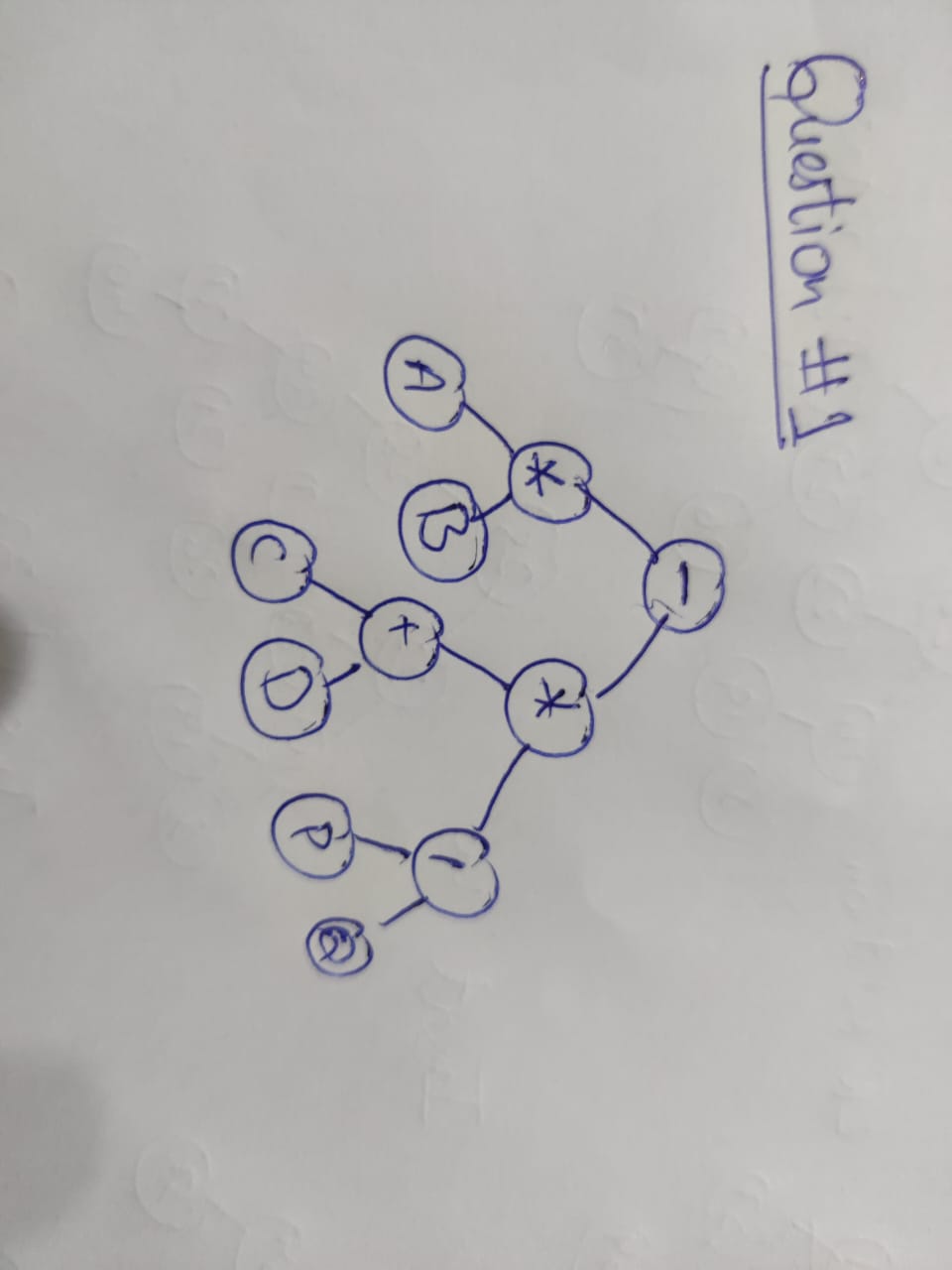
**DATA STRUCTURE**

**ASSIGNMENT 3**

**Name:**  Faizan Tariq

**Roll No:** 22F-3858

TASK 1:



TASK 2:

#include<iostream>

#include<string>

using namespace std;

struct Node {

int value;

Node\* left, \* right;

Node() {

left = NULL;

right = NULL;

value = 0;

} Node(int v) {

left = NULL;

right = NULL;

value = v;

}

};

class BST {

public:

Node\* root;

BST() {

root = NULL;

}

void insertElement(Node\*& node, int num)

{

if (node == nullptr)

node = new Node(num);

else if (num < node->value)

insertElement(node->left, num);

else if (num > node->value)

insertElement(node->right, num);

else

cout << "Duplicated values present." << endl;

}

void insert(int num)

{

insertElement(root, num);

}

void convertToGreater() {

int sum = 0;

addAllRightNodeValuesToCurrent(root, sum);

}

void addAllRightNodeValuesToCurrent(Node\* Root , int & sum) { // inorder traversal

if (!Root) {

return;

}

addAllRightNodeValuesToCurrent(Root->right, sum);

Root->value += sum;

sum = Root->value;

addAllRightNodeValuesToCurrent(Root->left, sum);

}

void inOrder(Node\* Root) {

if (!Root) {

return;

}

inOrder(Root->left);

cout << " " << Root->value;

inOrder(Root->right);

}

void displayTree() {

inOrder(root);

}

Node\* getRootNode() {

return root;

}

};

int main() {

BST tree;

tree.insert(1);

tree.insert(4);

tree.insert(2);

tree.insert(0);

tree.insert(3);

tree.insert(6);

tree.insert(5);

tree.insert(7);

tree.insert(8);

tree.displayTree();

cout << endl;

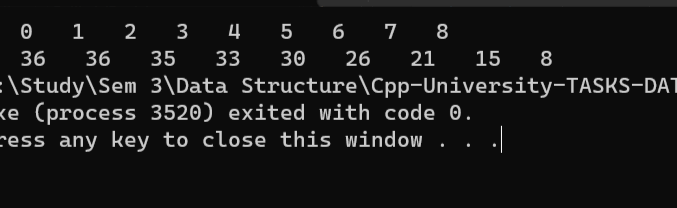
tree.convertToGreater();

tree.displayTree();

return 0;

}

OUTPUT:



TASK 3:

#include<iostream>

#include<string>

using namespace std;

int counter = 0;// to calculate number of leaf nodes

struct Node {

char data;

Node\* left, \* right;

Node() {

left = NULL;

right = NULL;

data = ' ';

}

};

class BST {

Node\* root;

public:

BST() {

root = NULL;

}

void insert(char data) {

Node\* newNode = new Node;

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

if (!root)

root = newNode; // if the tree is not created

else {

Node\* cur = root;

while (true) {

if (data< cur->data) {

if (cur->left) {

cur = cur->left;

}

else {

cur->left = newNode;

return;

}

}

else if (data > cur->data) {

if (cur->right) {

cur = cur->right;

}

else {

cur->right = newNode;

return;

}

}

else {

cur->left = newNode; // for the same score, store on the left

break;

}

}

}

}

void deleteNode(char data, Node\*& newNode) {

if (newNode == NULL) {

cout << data << " Not found\n";

}

else if (data < newNode->data) {

deleteNode(data, newNode->left);

}

else if (data > newNode->data) {

deleteNode(data, newNode->right);

}

else {

makeDelete(newNode);

}

}

void makeDelete(Node\*& newNode) {

Node\* deleteNode;

if (newNode->right == NULL) {

deleteNode = newNode;

newNode = newNode->left;

delete deleteNode;

}

else if (newNode->left == NULL) {

deleteNode = newNode;

newNode = newNode->right;

delete deleteNode;

}

else {

deleteNode = newNode->right;

while (deleteNode->left) {

deleteNode = deleteNode->left;

}

deleteNode->left = newNode->left;

deleteNode = newNode;

newNode = newNode->right;

delete deleteNode;

}

}

void inOrder(Node\* cur){

if (!cur) {

return;

}

inOrder(cur->left);

cout << cur->data<<" ";

inOrder(cur->right);

}

void display() {

inOrder(root);

}

void removeChar(char ch) {

deleteNode(ch, root);

}

};

int main() {

BST tree;

tree.insert('L');

tree.insert('P');

tree.insert('D');

tree.insert('C');

tree.insert('H');

tree.insert('F');

tree.insert('J');

tree.insert('A');

tree.display();

cout << "\nDELETING J\n";

tree.removeChar('J');

tree.display();

cout << "\nDELETING C\n";

tree.removeChar('C');

tree.display();

cout << "\nDELETING L\n";

tree.removeChar('L');

tree.display();

cout << "\nDELETING D\n";

tree.removeChar('D');

tree.display();

cout << "\nDELETING A\n";

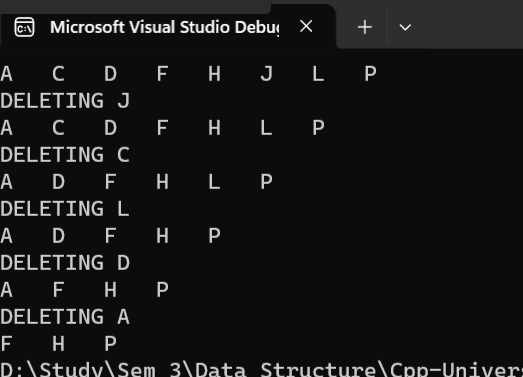
tree.removeChar('A');

tree.display();

return 0;

}

OUTPUT:



TASK 4:

#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

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

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* 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), y->height) + 1;

return x;

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* 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), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

K1->right = rotateLeft(K1->right);

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

K3->left = rotateRight(K3->left);

return rotateLeft(K3);

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == nullptr) {

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

} //Else X is in the tree already; we'll do nothing

if (x == T->data) {

cout << "Already Exists.\n";

return T;

}

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

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

bool find(int value, AvlNode\* startNode) {

while (startNode) {

if (value < startNode->data) {

startNode = startNode->left;

}

else if (value > startNode->data) {

startNode = startNode->right;

}

else {

cout << "found";

return true;

}

}

return false;

}

AvlNode\* deleteNode(int num, AvlNode\* nodePtr) {

if (nodePtr == nullptr) {

cout << "Num not found!\n";

return nullptr;

}

if (num < nodePtr->data) {

nodePtr->left = deleteNode(num, nodePtr->left);

}

else if (num > nodePtr->data) {

nodePtr->right = deleteNode(num, nodePtr->right);

}

else {

AvlNode\* tempNode = nodePtr;

if (nodePtr->left == nullptr) {

nodePtr = nodePtr->right;

}

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

nodePtr = nodePtr->left;

}

else {

tempNode = nodePtr->right;

while (tempNode->left) {

tempNode = tempNode->left;

}

tempNode->left = nodePtr->left;

nodePtr = nodePtr->right;

}

delete tempNode;

}

if (nodePtr != nullptr) {

nodePtr->height = 1 + max(getHeight(nodePtr->left), getHeight(nodePtr->right)); // calculates height of inserted node

int balance = getBalanceFactor(nodePtr);

if (balance > 1) {

if (getBalanceFactor(nodePtr->left) >= 0) {

return rotateRight(nodePtr); // Right-Right rotation

}

else {

return doubleRotateWithLeft(nodePtr); // Right-Left rotation

}

}

if (balance < -1) {

if (getBalanceFactor(nodePtr->right) <= 0) {

return rotateLeft(nodePtr); // Left-Left rotation

}

else {

return doubleRotateWithRight(nodePtr); // Left-Right rotation

}

}

}

return nodePtr;

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void insert(int x) {

root = insert(x, root);

}

void remove(int num) {

root = deleteNode(num, root);

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

AvlNode\* getRoot() {

return root;

}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

void isAVL\_subTREE() {

int minV, maxV;

cout << "Enter Min value :";

cin >> minV;

cout << "Enter max value :";

cin >> maxV;

cout << "IS SUB-TREE IS AVL ?";

if (isAVLSubtree(root, minV, maxV)) {

cout << "YES";

}

else {

cout << "NO";

}

}

bool isAVLSubtree(AvlNode\* root, int minValue, int maxValue) {

if (root == nullptr)

return true;

if (root->data < minValue || root->data > maxValue)

return false;

bool leftBalanced = isAVLSubtree(root->left, minValue, root->data);

bool rightBalanced = isAVLSubtree(root->right, root->data, maxValue);

if (!leftBalanced || !rightBalanced)

return false;

return true;

}

};

int main() {

AvlTree avl1;

avl1.insert(10);

avl1.insert(9);

avl1.insert(8);

avl1.insert(7);

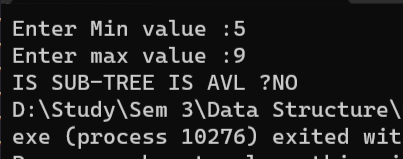
avl1.insert(6);

avl1.insert(5);

avl1.isAVL\_subTREE();

return 0;

}

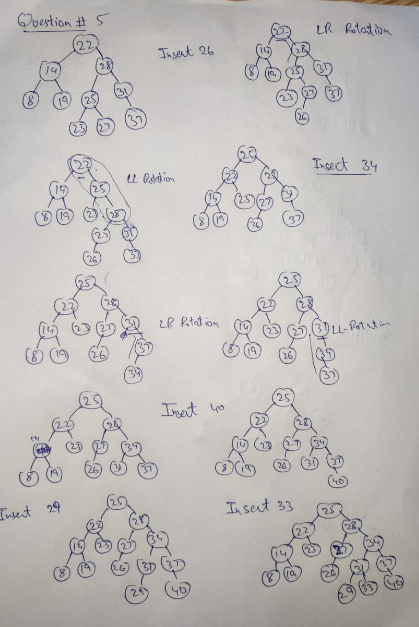
**OUTPUT:  
**

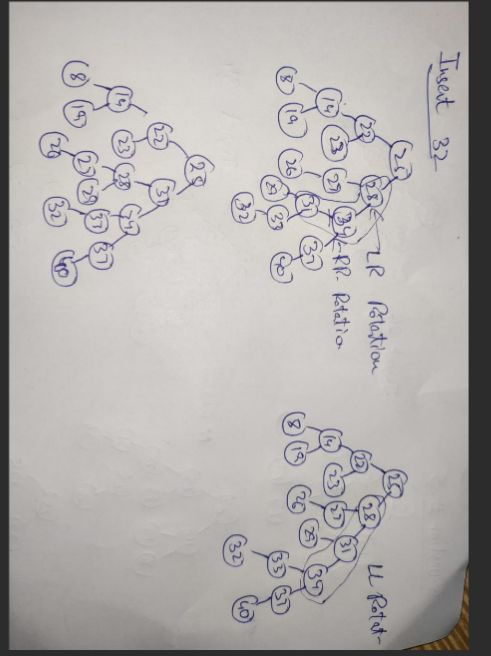
**A screen shot of a computer

Description automatically generated  
INPUT:  
A screen shot of a computer program

Description automatically generated**

**TASK 5:**

****

****

**TASK 6:**

#include <iostream>

using namespace std;

struct AvlNode {

int StudentId;

string department;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value, string Dep) : StudentId(value), left(NULL), right(NULL), height(0), department(Dep) {}

};

class AvlDataBase {

private:

AvlNode\* root;

public:

AvlDataBase() : root(NULL) {}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

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

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* 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), y->height) + 1;

return x;

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* 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), x->height) + 1;

return y;

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->StudentId << " " <<p->department <<" ";

inorderDisplay(p->right);

}

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

void addStudent(int id, string dep) {

root = insert(id, dep, root);

}

AvlNode\* insert(int id, string dep, AvlNode\* T) {

if (T == nullptr) {

return new AvlNode(id, dep);

}

if (id < T->StudentId) {

T->left = insert(id, dep, T->left);

}

else if (id > T->StudentId) {

T->right = insert(id,dep, T->right);

}

else if (id == T->StudentId) {

cout << "Student Already exists.\n";

return T;

}

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

int balance = getBalanceFactor(T);

if (balance > 1) {

if (id < T->left->StudentId) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (id > T->right->StudentId) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

string setDepartment(int studentId) {

if (studentId >= 1000 && studentId <= 1999) {

return "CS";

}

else if (studentId >= 2000 && studentId <= 2999) {

return "SE";

}

else if (studentId >= 3000 && studentId <= 3999) {

return "EE";

}

else if (studentId >= 4000 && studentId <= 4999) {

return "S&H";

}

else if (studentId >= 5000 && studentId <= 5999) {

return "FSM";

}

else return "INVALID ID";

}

void editStudent(int id) {

if (find(id, root)) {

cout << "\nStudent updated\n";

}

else {

cout << "\nStudent not found in database.\n";

}

}

bool find(int id, AvlNode\* startNode) {

while (startNode) {

if (id < startNode->StudentId) {

startNode = startNode->left;

}

else if (id > startNode->StudentId) {

startNode = startNode->right;

}

else {

string tempString = " ";

int tempId = 0;

cout << "Enter new id of student :";

cin >> tempId;

tempString = setDepartment(tempId);

startNode->StudentId = tempId;

startNode->department = tempString;

return true;

}

}

return false;

}

};

int main() {

AvlDataBase database;

int choice = 0;

int tempStudentId = 0;

string tempStudentDepartment = " ";

bool end = false;

while (!end) {

cout << "\n1. Add a student.\n2.Update Student.\n3.Display InOrder.\n";

cin >> choice;

switch (choice) {

case 1:

cin >> tempStudentId;

tempStudentDepartment = database.setDepartment(tempStudentId);

database.addStudent(tempStudentId, tempStudentDepartment);

break;

case 2:

cin >> tempStudentId;

database.editStudent(tempStudentId);

break;

case 3:

database.displayInOrder();

break;

default:

end = true;

break;

}

}

return 0;

}

**OUTPUT:**

**A screenshot of a computer program

Description automatically generated**

**TASK 7:**

#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

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

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* 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), y->height) + 1;

return x; /\* New root \*/

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* 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), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == nullptr) {

/\* Create and return a new node \*/

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

} /\* Else X is in the tree already; we'll do nothing \*/

if (x == T->data) {

cout << "Already Exists.\n";

return T;

}

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

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

bool find(int value, AvlNode\* startNode) {

while (startNode) {

if (value < startNode->data) {

startNode = startNode->left;

}

else if (value > startNode->data) {

startNode = startNode->right;

}

else {

cout << "found";

return true;

}

}

return false;

}

AvlNode\* deleteNode(int num, AvlNode\* nodePtr) {

if (nodePtr == nullptr) {

cout << "Num not found!\n";

return nullptr;

}

if (num < nodePtr->data) {

nodePtr->left = deleteNode(num, nodePtr->left);

}

else if (num > nodePtr->data) {

nodePtr->right = deleteNode(num, nodePtr->right);

}

else {

AvlNode\* tempNode = nodePtr;

if (nodePtr->left == nullptr) {

nodePtr = nodePtr->right;

}

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

nodePtr = nodePtr->left;

}

else {

tempNode = nodePtr->right;

while (tempNode->left) {

tempNode = tempNode->left;

}

tempNode->left = nodePtr->left;

nodePtr = nodePtr->right;

}

delete tempNode;

}

if (nodePtr != nullptr) {

nodePtr->height = 1 + max(getHeight(nodePtr->left), getHeight(nodePtr->right)); // calculates height of inserted node

int balance = getBalanceFactor(nodePtr);

if (balance > 1) {

if (getBalanceFactor(nodePtr->left) >= 0) {

return rotateRight(nodePtr); // Right-Right rotation

}

else {

return doubleRotateWithLeft(nodePtr); // Right-Left rotation

}

}

if (balance < -1) {

if (getBalanceFactor(nodePtr->right) <= 0) {

return rotateLeft(nodePtr); // Left-Left rotation

}

else {

return doubleRotateWithRight(nodePtr); // Left-Right rotation

}

}

}

return nodePtr;

}

void getKthLargest(AvlNode\* Root, int k) {

int cur = 0;

AvlNode\* kthNode = getkthNode(Root, k, cur);

if (!kthNode) {

cout << "Nodes in tree are less than k"<<endl;

}

else {

cout << "Your kth largest element is :" << kthNode->data << endl;

}

}

AvlNode\* getkthNode(AvlNode\* Root, int k, int cur) {

if (Root == NULL || cur >= k) {

return NULL;

}

AvlNode\* rightNode = getkthNode(Root->right, k, cur);

if (rightNode) {

return rightNode;

}

if (++cur == k) {

return Root;

}

return getkthNode(Root->left, k, cur);

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void insert(int x) {

root = insert(x, root);

}

void remove(int num) {

root = deleteNode(num, root);

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

AvlNode\* getRoot() {

return root;

}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

void getKLargest(int k) {

getKthLargest(root, k);

}

};

int main() {

AvlTree avl1;

avl1.insert(10);

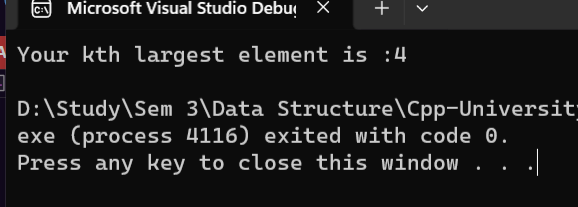
avl1.insert(4);

avl1.getKLargest(2);

return 0;

}

OUTPUT:

****

TASK 8:

#include <iostream>

#include <string>

using namespace std;

struct AvlNode {

string songName;

int frequency;

AvlNode\* left;

AvlNode\* next;

AvlNode\* right;

int height;

AvlNode(int songFreq, string Sname) :next(NULL), left(NULL), right(NULL), height(0), songName(Sname), frequency(songFreq) {

}

};

class AVLTree {

private:

AvlNode\* root;

int height(AvlNode\* N) {

if (N == NULL)

return 0;

return N->height;

}

AvlNode\* rightRotate(AvlNode\* y) {

AvlNode\* x = y->left;

AvlNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

AvlNode\* leftRotate(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

int getBalance(AvlNode\* N) {

if (N == NULL)

return 0;

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

}

AvlNode\* insert(AvlNode\* node, string songName, int frequency) {

if (node == NULL) {

return new AvlNode(frequency, songName);

}

if (songName == node->songName) {

AvlNode\* newListNode = new AvlNode(frequency, songName);

newListNode->next = node->next;// insert at start

node->next = newListNode;

return node; // No need to further insert in the tree since it's a duplicate

}

if (frequency < node->frequency) {

node->left = insert(node->left, songName, frequency);

}

else {

node->right = insert(node->right, songName, frequency);

}

// Balance the tree after insertion

int balance = getBalance(node);

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

return rightRotate(node);

}

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

return leftRotate(node);

}

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

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

return rightRotate(node);

}

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

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

return leftRotate(node);

}

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

return node;

}

void inOrderdisplay(AvlNode\* node) {

if (node == NULL) {

return;

}

inOrderdisplay(node->left);

cout << "Song Name: " << node->songName << " | Frequency: " << node->frequency << endl;

AvlNode\* current = node->next;

while (current != NULL) {

cout << "Song Name: " << current->songName << " | Frequency: " << current->frequency << endl;

current = current->next;

}

inOrderdisplay(node->right);

}

public:

AVLTree() : root(NULL) {}

void addSong(string songName, int freq) {

root = insert(root, songName, freq);

}

void menu() {

cout << "\n1. Add Song.\n2. Play Song.\n3.Quit\n";

}

void display() {

inOrderdisplay(root);

}

AvlNode\* findNode(AvlNode\* cur, int songFreq) {

while (cur) {

if (songFreq < cur->frequency) {

cur = cur->left;

}

else if (songFreq > cur->frequency){

cur = cur->right;

}

else {

return cur;

}

}

return NULL;

}

AvlNode\* minValueNode(AvlNode\* node) {

AvlNode\* current = node;

while (current->left != NULL) {

current = current->left;

}

return current;

}

AvlNode\* deleteNode(AvlNode\* cur, string songName, int frequency) {

if (cur == NULL) {

return cur;

}

if (songName < cur->songName) {

cur->left = deleteNode(cur->left, songName, frequency);

}

else if (songName > cur->songName) {

cur->right = deleteNode(cur->right, songName, frequency);

}

else {

if (cur->next != NULL) {

// If the node has a linked list, just remove the specific song

AvlNode\* temp = cur->next;

AvlNode\* prev = NULL;

while (temp != NULL && temp->songName != songName) {

prev = temp;

temp = temp->next;

}

if (temp != NULL) {

if (prev == NULL) {

cur->next = temp->next;

delete temp;

}

else {

prev->next = temp->next;

delete temp;

}

}

}

else {

if ((cur->left == NULL) || (cur->right == NULL)) {

AvlNode\* temp; // initially NULL

if (root->left != NULL) {

temp = root->left;

}

else {

temp = root->right;

}

if (temp == NULL) { // deletion for two childrens

temp = root;

root = NULL;

}

else {

\*root = \*temp;

}

delete temp;

}

else {

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

root->songName = temp->songName;

root->frequency = temp->frequency;

root->right = deleteNode(root->right, temp->songName, temp->frequency);

}

}

}

if (root == NULL) {

return root;

}

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

int balance = getBalance(root);

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

return rightRotate(root);

}

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

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

return rightRotate(root);

}

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

return leftRotate(root);

}

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

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

return leftRotate(root);

}

return root;

}

void play(string SongName, int freq) {

if (playSong(root, SongName, freq)) {

cout << "\nSong Played.\n";

}

else {

cout << "\nSong not found.\n";

}

}

bool playSong(AvlNode\* cur, string SongName, int freq) {

AvlNode\* NodeList = findNode(cur, freq); // return a node having same frequncy

if (NodeList == NULL) { // if NodeList is null return false

return false;

}

// now check if list have songName

bool hasSong = false;

int count = 0; // for counting number of nodes

while (NodeList->next != NULL) {

count++;

if (NodeList->songName == SongName) {

hasSong = true;

break;

}

NodeList = NodeList->next;

}

if (hasSong) {

// now we found song its time to delete from its old place and insert having frequency added by 1

if (count == 0) { // means there is only 1 node , we need to delete from tree not from linked list

int tempfreq = NodeList->frequency;

string tempSname = NodeList->songName;

deleteNode(NodeList, tempSname, tempfreq);

addSong(tempSname, tempfreq);

}

else { // delete from linked list

if (NodeList->next == NULL) {// if NodeList is end of linked List node

int tempfreq = NodeList->frequency;

string tempSname = NodeList->songName;

delete NodeList;

addSong(tempSname, ++tempfreq); // increment frequency by 1

}

else { // if NodeList is in middle

AvlNode\* deletedNode = NodeList;

int tempfreq = NodeList->frequency;

string tempSname = NodeList->songName;

NodeList = NodeList->next;

delete deletedNode;

addSong(tempSname, ++tempfreq); // increment frequency by 1

}

}

return true;

}

else {

return false;

}

}

};

int main() {

AVLTree playList;

int choice = 0;

int tempFrequence = 0;

string tempSongName= "";

while (choice >= 0) {

playList.menu();

cin >> choice;

switch (choice)

{

case 1:

cout << "\nEnter Song name :";

cin >> tempSongName;

cout << "\nEnter Song Frequency :";

cin >> tempFrequence;

playList.addSong(tempSongName, tempFrequence);

cout << endl;

playList.display();

break;

case 2:

cout << "\nEnter Song name to play:";

cin >> tempSongName;

cout << "\nEnter Song Frequency to play :";

cin >> tempFrequence;

playList.play(tempSongName, tempFrequence);

cout << endl;

playList.display();

break;

default:

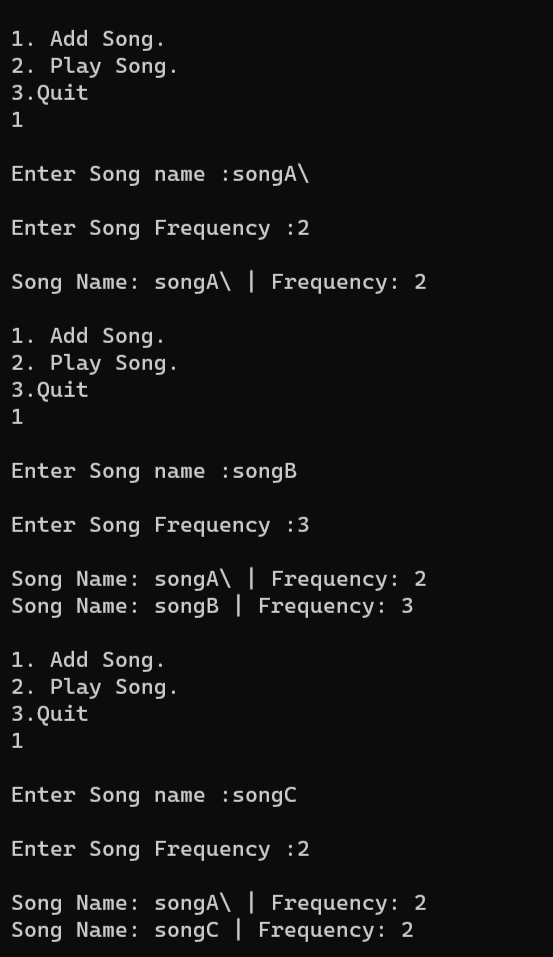
return 0;

}

}

return 0;

}

OUTPUT:  


A screenshot of a computer program

Description automatically generated

TASK 9:

#include<iostream>

using namespace std;

struct Building {

string name;

int condition;

Building() {

name = "";

condition = 0;

}

Building(string Name, int c) {

name = Name;

condition = c;

}

};

class Binary\_heap {

Building\* arrayOfpatients;

int size;

int totalPatients;

public:

Binary\_heap(int size) {

this->size = size;

totalPatients = 0;

arrayOfpatients = new Building[size];

}

string getHighestPriorityBuilding() { // returns building with heighes priority

return arrayOfpatients[1].name;

}

void insert\_min\_heap(int condition, string name) {

if (totalPatients == size - 1) {

cout << "Heap is full.\n";

return;

}

else {

++totalPatients;

arrayOfpatients[totalPatients].condition = condition;

arrayOfpatients[totalPatients].name= name;

perculate\_up(totalPatients);

}

}

void perculate\_up(int index) { //

while (index > 1 && arrayOfpatients[index / 2].condition <= arrayOfpatients[index].condition) {

swap(arrayOfpatients[index / 2].condition, arrayOfpatients[index].condition);

swap(arrayOfpatients[index / 2].name, arrayOfpatients[index].name);

index = index / 2;

}

}

void perculate\_down(int index) {

int smallerValue = index;

while (1) {

if (index \* 2 < size && arrayOfpatients[index \* 2].condition > arrayOfpatients[smallerValue].condition) {

smallerValue = index \* 2;

}

if (index \* 2 + 1 < size && arrayOfpatients[index \* 2 + 1].condition > arrayOfpatients[smallerValue].condition) {

smallerValue = index \* 2 + 1;

}

if (smallerValue == index) {

break;

}

swap(arrayOfpatients[smallerValue].condition, arrayOfpatients[index].condition);

swap(arrayOfpatients[smallerValue].name, arrayOfpatients[index].name);

index = smallerValue;

}

}

void allotbed() {

cout << endl;

cout << arrayOfpatients[1].name << " alloted bed having \"" << arrayOfpatients[1].condition<<"\" condition";

arrayOfpatients[1] = arrayOfpatients[totalPatients--];

perculate\_down(1);

}

};

int main() {

Binary\_heap binary(100);

binary.insert\_min\_heap(20,"Patient1");

binary.insert\_min\_heap(45, "Patient2");

cout << "\nGET MINI :" << binary.getHighestPriorityBuilding();

binary.insert\_min\_heap(5,"Patient3");

cout << "\nGET MINI :" << binary.getHighestPriorityBuilding();

binary.allotbed();

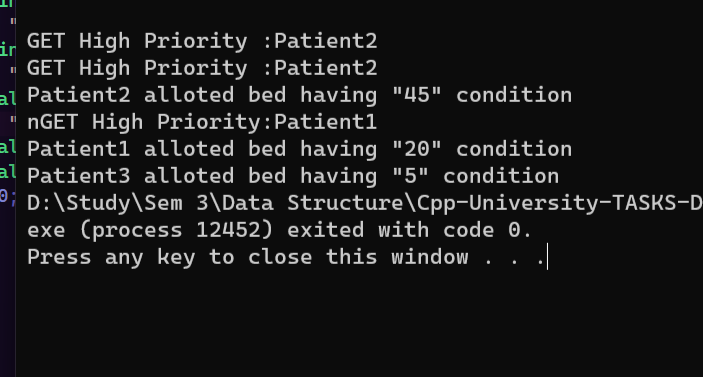
cout << "\nGET MINI :" << binary.getHighestPriorityBuilding();

binary.allotbed();

binary.allotbed();

return 0;

}

OUTPUT:  
****