

22F-3104

DSA Assignment no 4

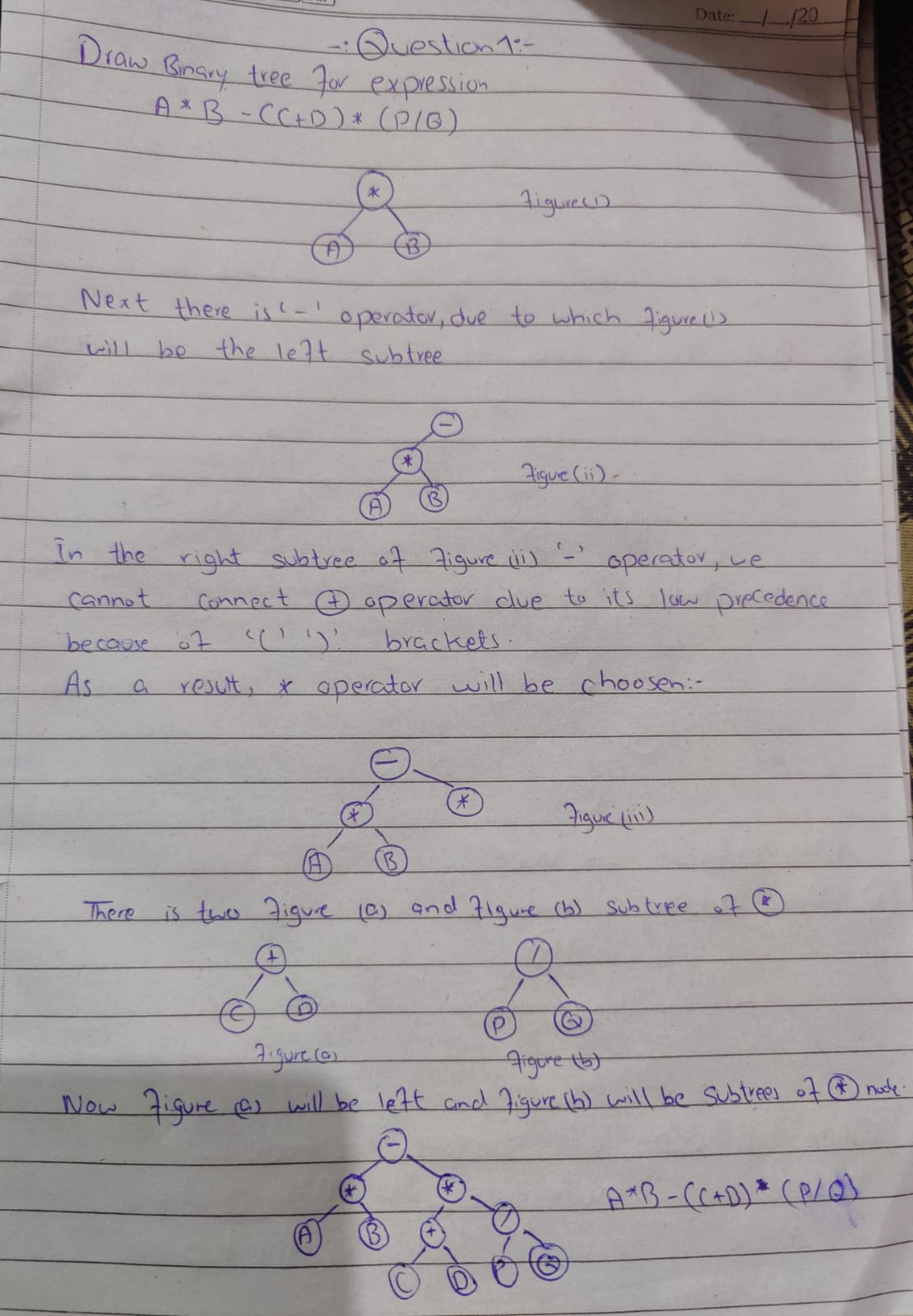


November 23, 2023

fast cfd campus

**Task no 1:**

**Screen Shot:**



**Task no 2:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

Node\* insert(Node\* root, int data) {

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

}

else if (data >= root->data) {

root->right = insert(root->right, data);

}

return root;

}

void display(Node\* root) {

if (root == nullptr) {

return;

}

display(root->left);

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

display(root->right);

}

};

Node\* GreaterTree(Node\* root, int& sum) {

if (root == nullptr) {

return nullptr;

}

GreaterTree(root->right, sum);

sum = sum + root->data;

root->data = sum;

GreaterTree(root->left, sum);

return root;

}

int main()

{

Node\* current = nullptr;

int num\_of\_nodes, sum = 0;

cout << "Enter the number of elements in the tree: ";

cin >> num\_of\_nodes;

int\* arr = new int[num\_of\_nodes];

for (int i = 0; i < num\_of\_nodes; i++) {

cout << "Enter the element " << i + 1 << ": ";

cin >> arr[i];

current = current->insert(current, arr[i]);

}

Node\* greaterTree = GreaterTree(current, sum);

cout << "\nGreater tree is: ";

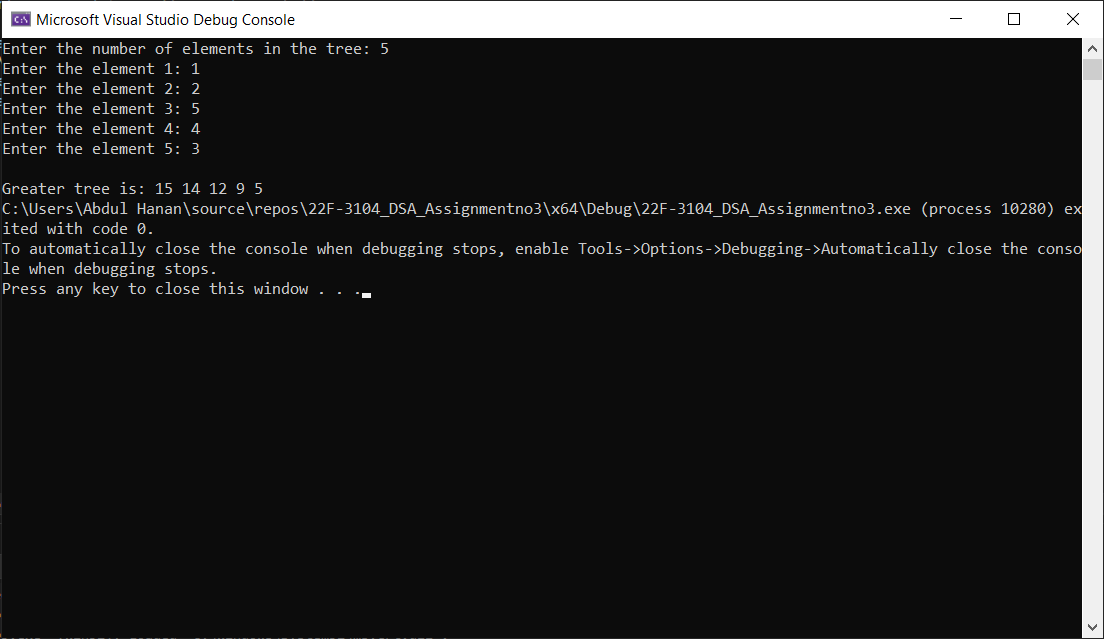
greaterTree->display(greaterTree);

delete[] arr;

return 0;

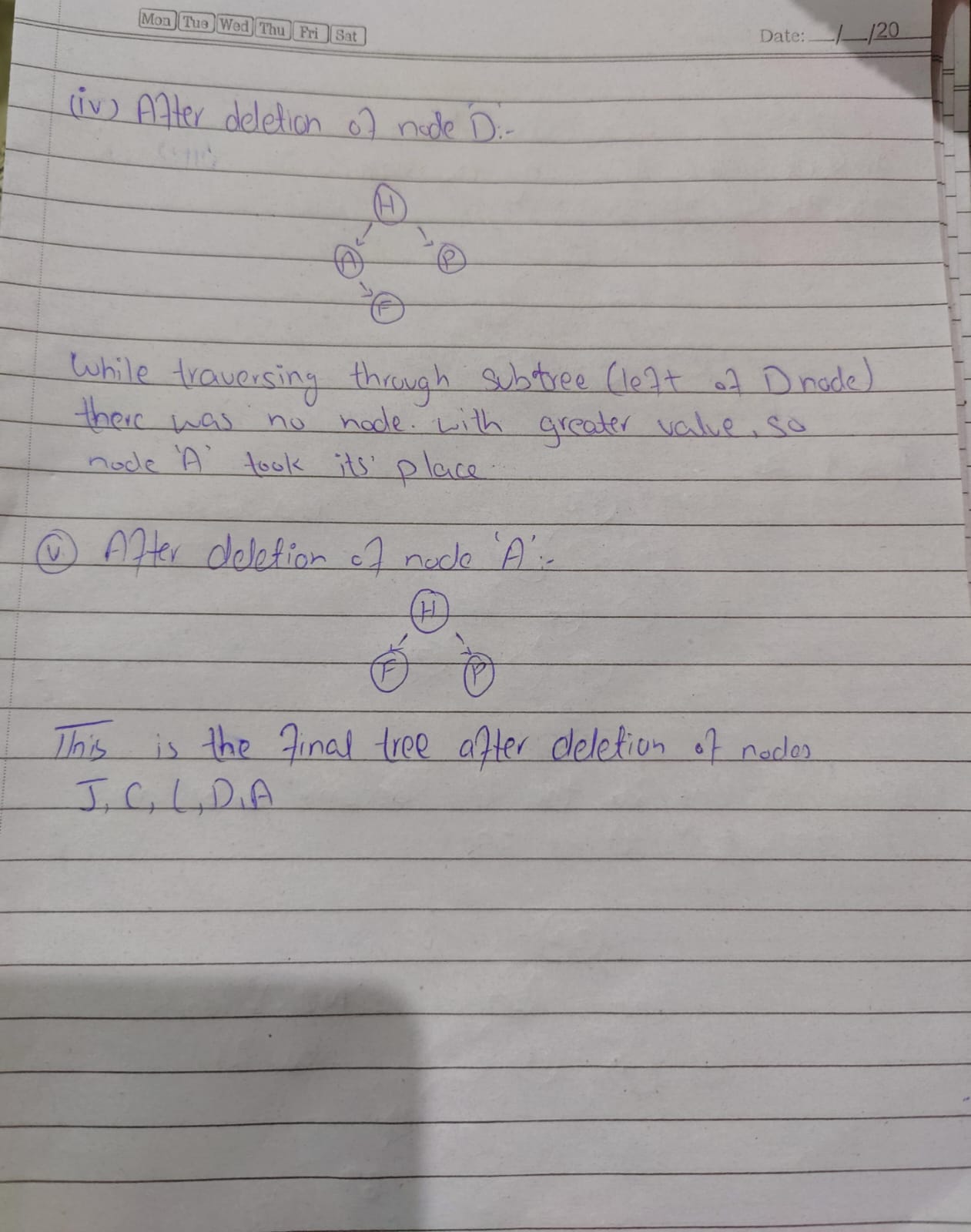
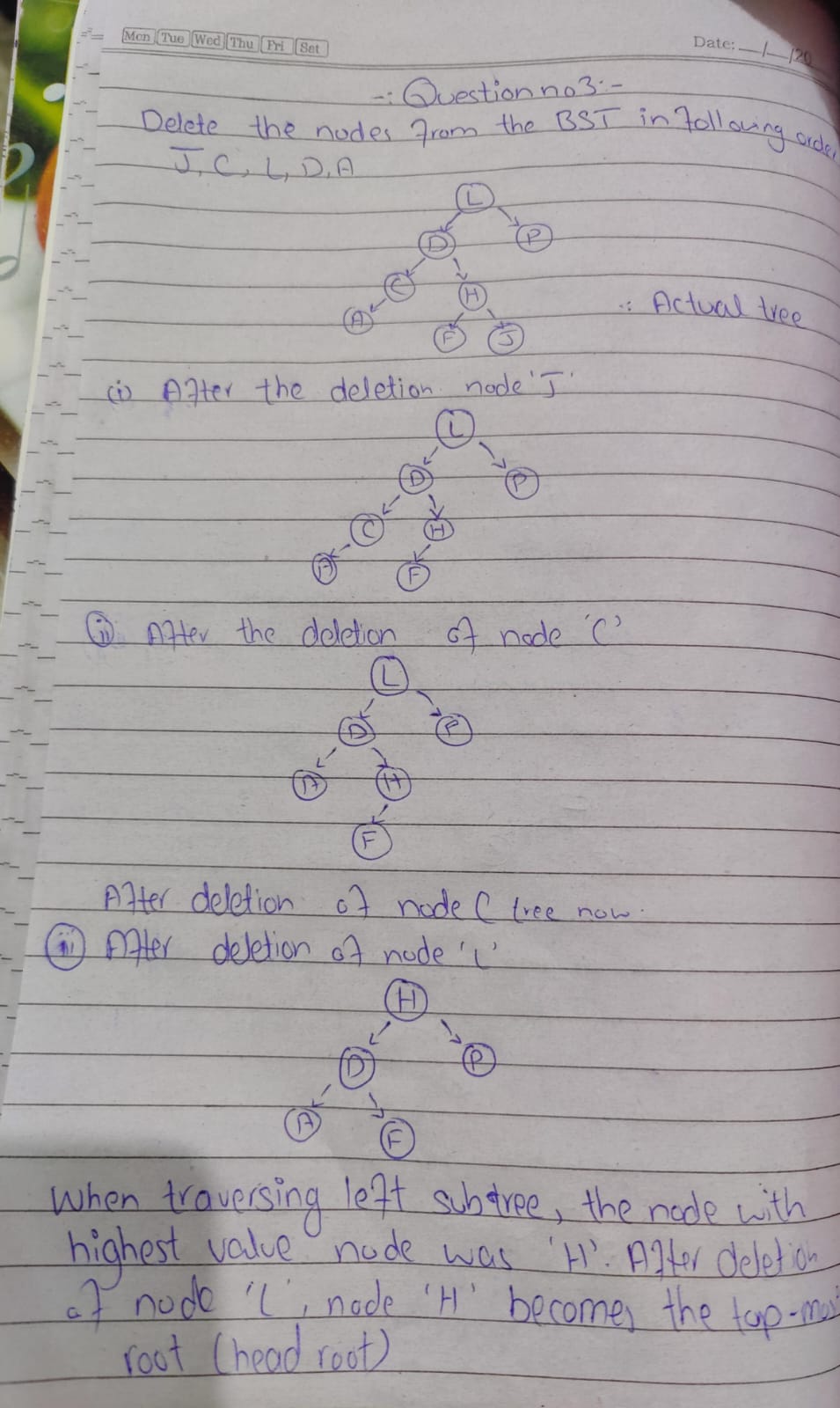
}

**Screen Shot:**

****

**Task no 3:**

**Screen Shot:**



**Task no 4:**

#include<iostream>

using namespace std;

struct avlnode {

avlnode\* right;

avlnode\* left;

int num, height;

};

class avltree {

avlnode\* root;

bool check;

public:

avltree() {

root = nullptr;

check = false;

}

avlnode\* getroot() {

return root;

}

int height(avlnode\* temp) {

if (temp == nullptr) {

return -1;

}

else {

return temp->height;

}

}

avlnode\* llrotation(avlnode\* parent) {

avlnode\* temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

avlnode\* RRrotation(avlnode\* parent) {

avlnode\* temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

avlnode\* LRrotation(avlnode\* parent) {

avlnode\* temp;

temp = parent->left;

parent->left = RRrotation(temp);

return llrotation(parent);

}

avlnode\* RLrotation(avlnode\* parent) {

avlnode\* temp;

temp = parent->right;

parent->right = llrotation(temp);

return RRrotation(parent);

}

avlnode\* Insert(int X,avlnode\* T) {

if (T == nullptr) {

T = new avlnode;

T->num = X;

T->left = T->right = nullptr;

}

else if (X < T->num) {

T->left = Insert(X, T->left);

if (height(T->left) - height(T->right)>= 2)

if (X < T->left->num)

T = llrotation(T);

else

T = LRrotation(T);

}

else if (X > T->num) {

T->right = Insert(X,T->right);

if (height(T->right) - height(T->left) >= 2)

if (X > T->right->num)

T = RRrotation(T);

else

T = RLrotation(T);

}

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

return T;

}

void inordertraversal(avlnode\* temp) {

if (temp != nullptr) {

inordertraversal(temp->left);

cout <<"Num="<< temp->num << endl;

inordertraversal(temp->right);

}

}

bool isavl(avlnode\*temp) {

if (temp == nullptr) {

return true;

}

else {

int balance = balanceFactor(temp);

if (balance < -1 || balance > 1) {

return false;

}

return isavl(temp->left) && isavl(temp->right);

}

}

int balanceFactor(avlnode\* node) {

if (node == nullptr) {

return 0;

}

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

}

bool rootedornot(avlnode\* node, int x, int y) {

if (node == nullptr) {

return true;

}

if (node->num >= x && node->num <= y) {

return true;

}

else {

return false;

}

}

};

int main() {

bool avlcheck,check1,check2;

avltree a;

avlnode \*tempnode = a.getroot();

int sel,num,num1, num2;

while (true) {

cout << "1.inserting a newnode\n";

cout << "2.check rootness and avl or not \n";

cout << "3.printall nodes\n";

cin >> sel;

if (sel == 1) {

cout << "enter num ";

cin >> num;

tempnode= a.Insert(num,tempnode);

}

else if (sel == 2) {

cout << "enter 2 values";

cin >> num1 >> num2;

check1=a.rootedornot(tempnode->right, num1, num2);

if (check1 == false) {

check2 = a.rootedornot(tempnode->left, num1, num2);

if (check2 == false) {

cout << "no root exist";

}

else {

avlcheck = a.isavl(tempnode->left);

if (avlcheck == true) {

cout << "The avl tree is rooted between 2 values and the tree is avl";

}

else {

cout << "The avl tree is rooted between 2 values but he tree is not avl";

}

}

}

else {

avlcheck = a.isavl(tempnode->right);

if (avlcheck == true) {

cout << "The avl tree is rooted between 2 values and the tree is avl";

}

}

}

else if (sel == 3) {

a.inordertraversal(tempnode);

}

else {

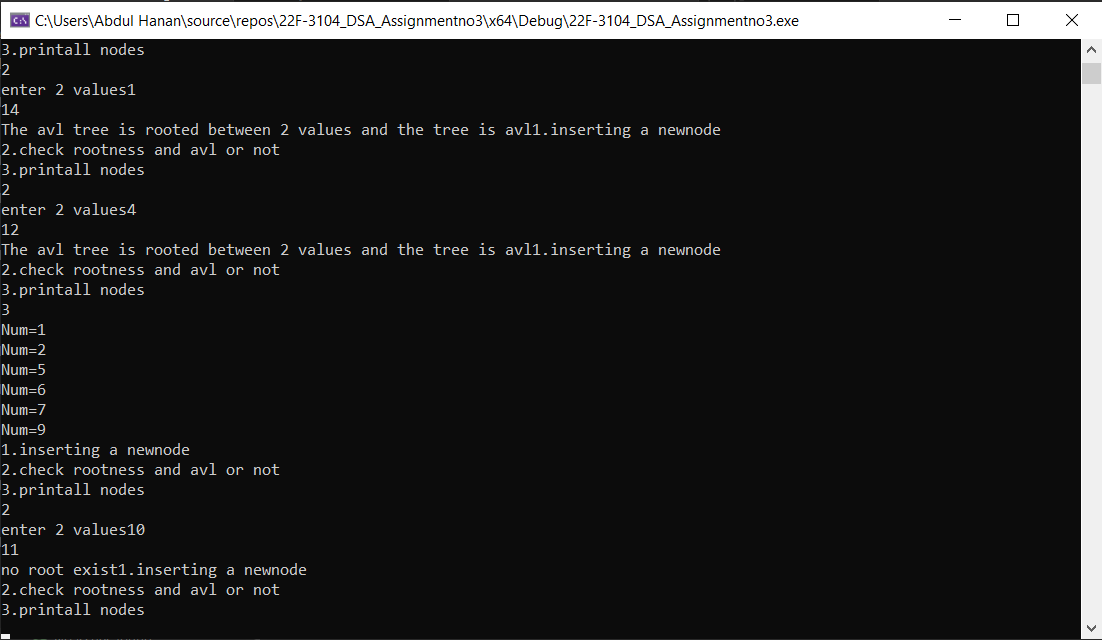
exit(1);

}

}

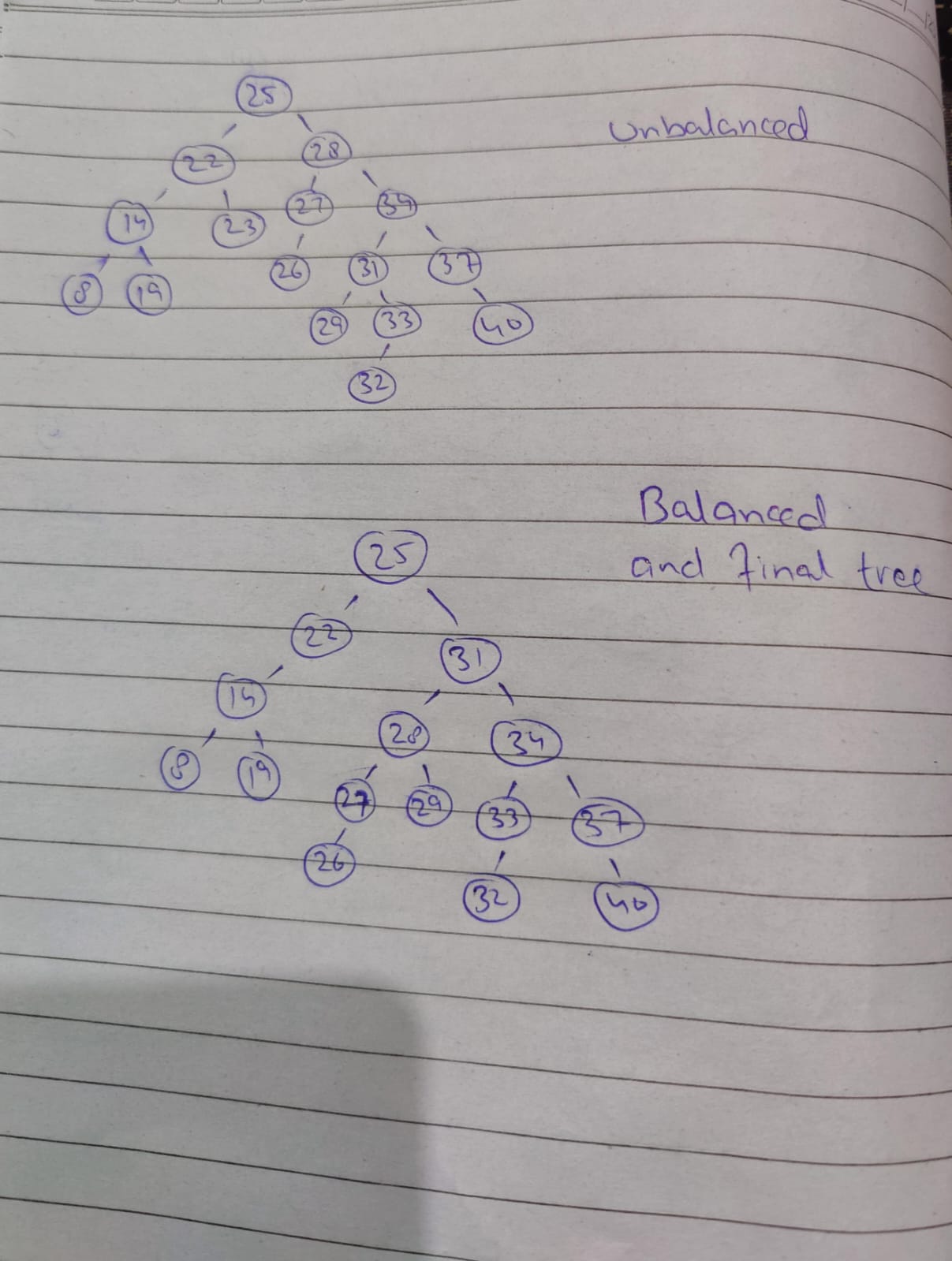
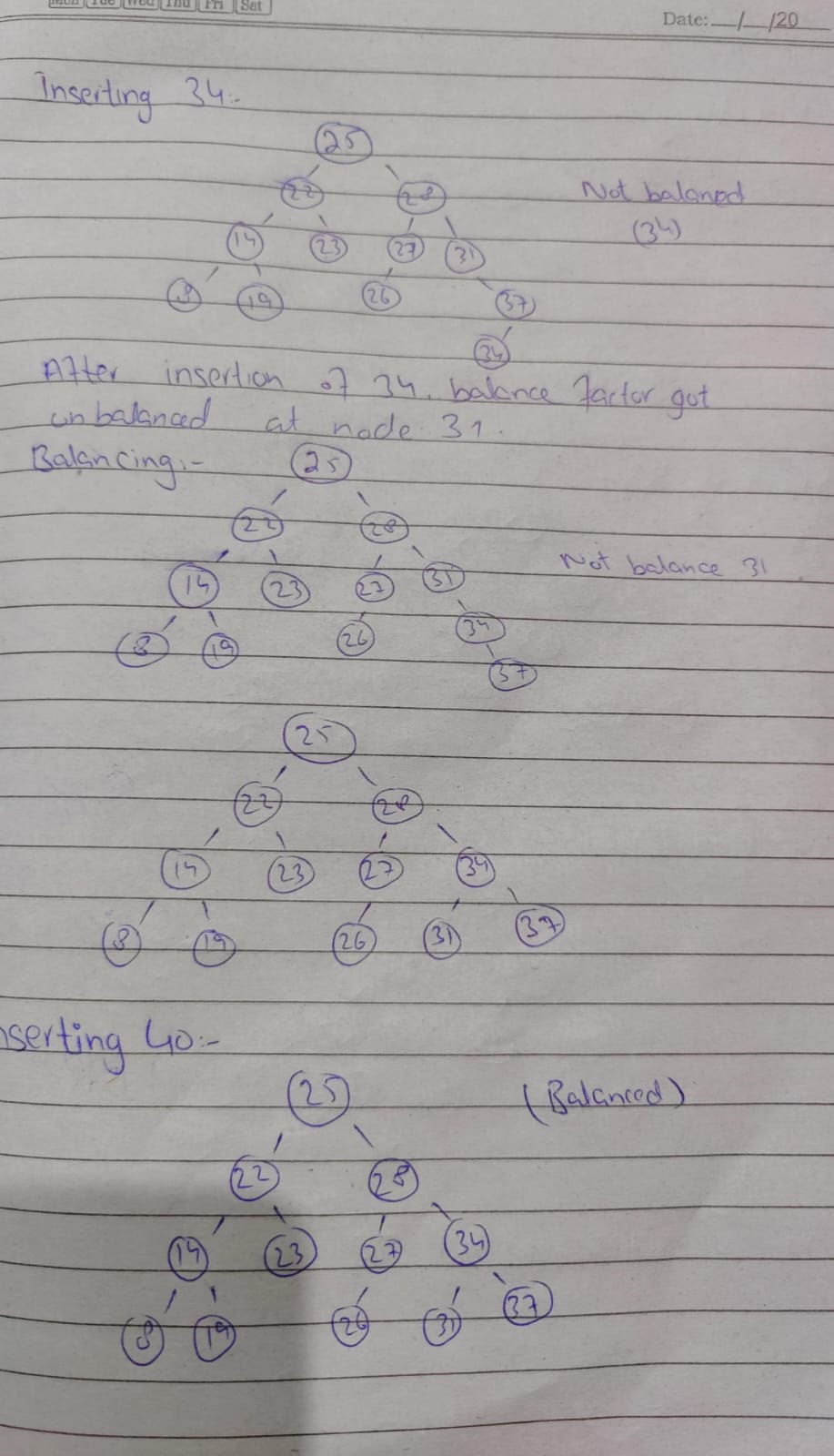
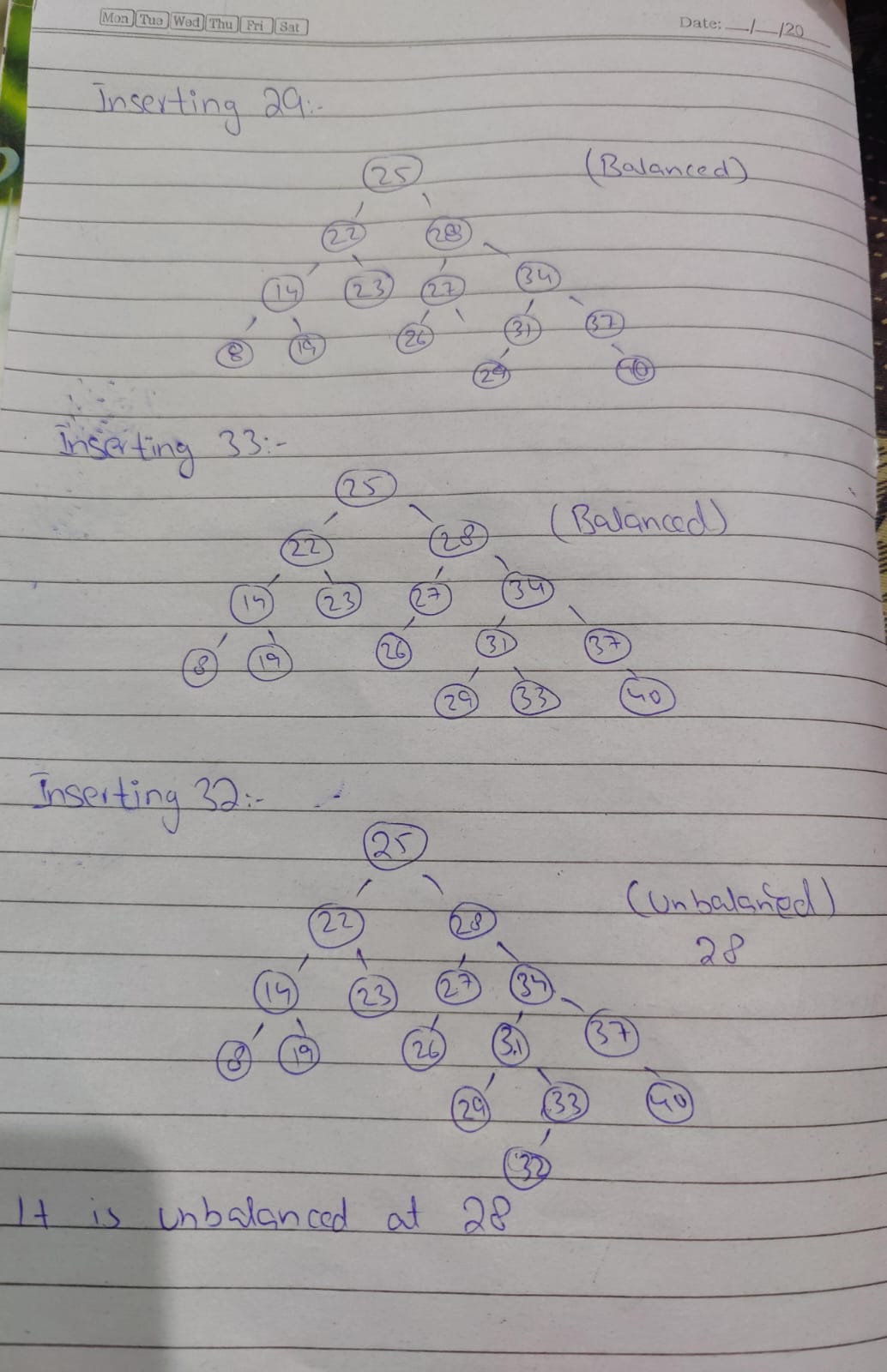
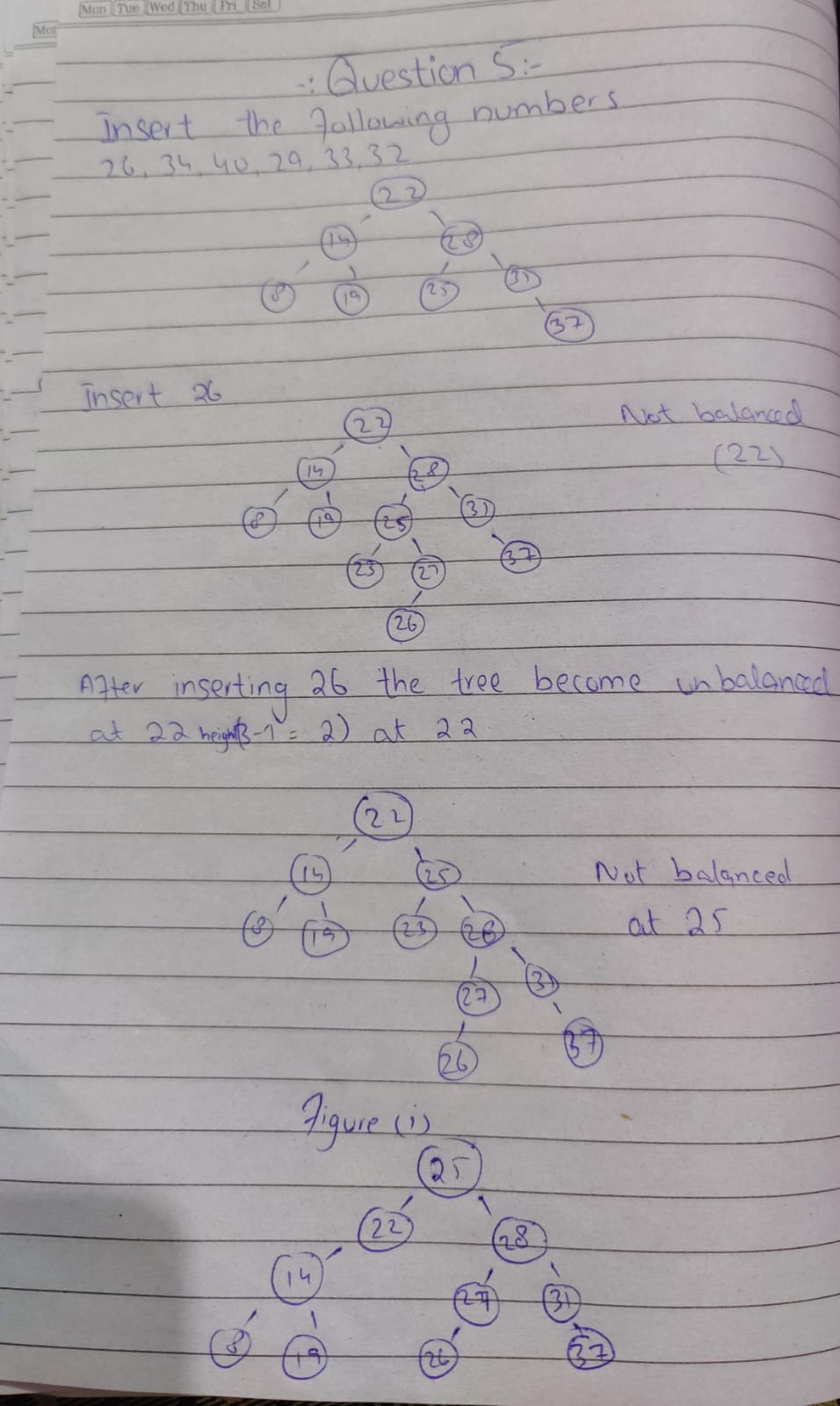
}

**Screen Shot:**

****

**Task no 5:**

**Screen Shot:**



**Task no 6:**

#include <iostream>

#include <string>

using namespace std;

struct StudentNode

{

string studentID, studentName, studentDepartment;

int height;

StudentNode\* left;

StudentNode\* right;

};

class StudentEnrollmentSystem

{

public:

StudentNode\* root;

StudentEnrollmentSystem()

{

root = NULL;

}

int getHeight(StudentNode\* node)

{

if (node == NULL)

{

return 0;

}

else

{

return node->height;

}

}

int getBalanceFactor(StudentNode\* node)

{

if (node == NULL)

{

return 0;

}

else

{

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

int heightDifference = leftHeight - rightHeight;

return heightDifference;

}

}

void updateHeight(StudentNode\* node)

{

if (node == NULL)

{

return;

}

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

int maxSubtreeHeight = max(leftHeight, rightHeight);

int updatedHeight = maxSubtreeHeight + 1;

node->height = updatedHeight;

}

StudentNode\* rotateRight(StudentNode\* y)

{

StudentNode\* x = y->left;

StudentNode\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

StudentNode\* rotateLeft(StudentNode\* x)

{

StudentNode\* y = x->right;

StudentNode\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

StudentNode\* insert(StudentNode\* node, string studentID, string studentName, string studentDepartment)

{

if (node == NULL)

{

StudentNode\* newNode = new StudentNode;

newNode->studentID = studentID;

newNode->studentName = studentName;

newNode->studentDepartment = studentDepartment;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

if (studentID < node->studentID)

{

node->left = insert(node->left, studentID, studentName, studentDepartment);

}

else if (studentID > node->studentID)

{

node->right = insert(node->right, studentID, studentName, studentDepartment);

}

else

return node; // Duplicate student IDs not allowed

updateHeight(node);

int balanceFactor = getBalanceFactor(node);

// Left Left case

if (balanceFactor > 1 && studentID < node->left->studentID)

{

return rotateRight(node);

}

// Right Right case

if (balanceFactor < -1 && studentID > node->right->studentID)

{

return rotateLeft(node);

}

// Left Right case

if (balanceFactor > 1 && studentID > node->left->studentID)

{

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

return rotateRight(node);

}

// Right Left case

if (balanceFactor < -1 && studentID < node->right->studentID)

{

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

return rotateLeft(node);

}

return node;

}

StudentNode\* search(StudentNode\* node, const string& studentID)

{

if (node == nullptr || node->studentID == studentID)

{

return node;

}

if (studentID < node->studentID)

{

return search(node->left, studentID);

}

else

return search(node->right, studentID);

}

void updateDepartment(string studentID, string newDepartment)

{

StudentNode\* studentNode = search(root, studentID);

if (studentNode == nullptr)

{

cout << "Student not found." << endl;

return;

}

studentNode->studentDepartment = newDepartment;

studentNode->studentID[0] = newDepartment[0];

cout << "Department updated successfully." << endl;

}

void inorder(StudentNode\* node)

{

if (node == nullptr)

{

return;

}

inorder(node->left);

cout << "Student ID: " << node->studentID << endl;

cout << "Student Name: " << node->studentName << endl;

cout << "Student Department: " << node->studentDepartment << endl;

cout << endl;

inorder(node->right);

}

};

void displayMenu() {

cout << "============================" << endl;

cout << "Student Enrollment System" << endl;

cout << "============================" << endl;

cout << "1. Add a student" << endl;

cout << "2. Update a student's department" << endl;

cout << "3. Display all students" << endl;

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

cout << "=============================" << endl;

cout << "Enter your choice: ";

}

int main()

{

StudentEnrollmentSystem obj;

while (true)

{

displayMenu();

int choice;

cin >> choice;

if (choice == 1)

{

string studentID, studentName, studentDepartment;

cout << "Enter student ID: ";

cin >> studentID;

cout << "Enter student name: ";

cin.ignore();

getline(cin, studentName);

cout << "Enter student department: ";

getline(cin, studentDepartment);

obj.root = obj.insert(obj.root, studentID, studentName, studentDepartment);

cout << "Student added successfully." << endl;

}

else if (choice == 2)

{

string studentID, newDepartment;

cout << "Enter student ID: ";

cin >> studentID;

cout << "Enter new department: ";

cin >> newDepartment;

obj.updateDepartment(studentID, newDepartment);

}

else if (choice == 3)

{

cout << "List of all students:" << endl;

obj.inorder(obj.root);

}

else if (choice == 4)

{

cout << "Exiting program (^\_^)|" << endl;

break;

}

else

{

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

}

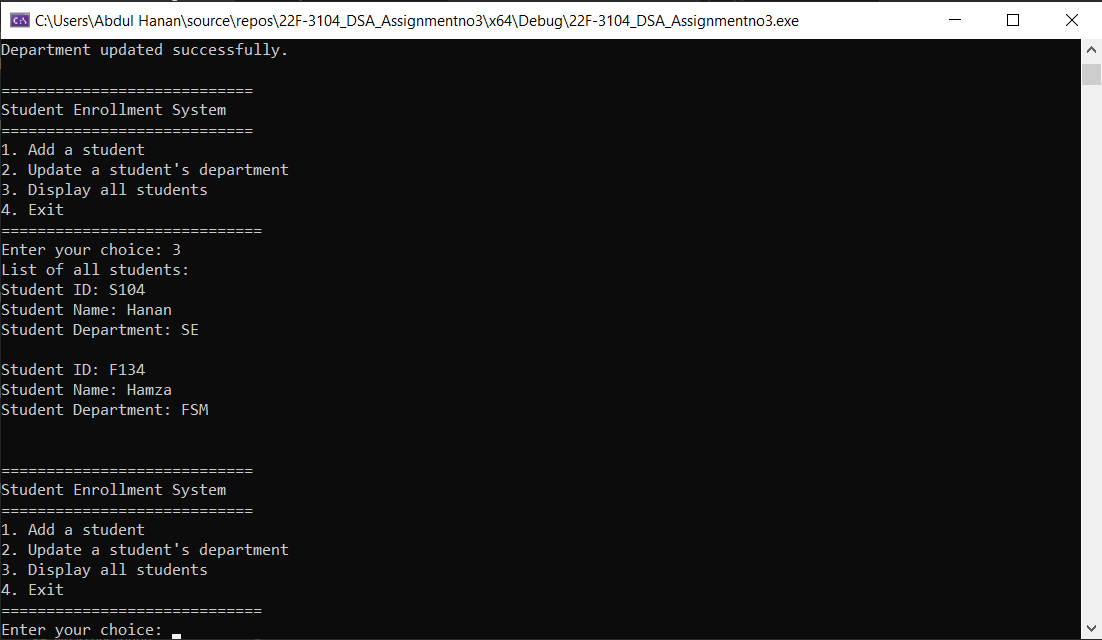
cout << endl;

}

return 0;

}

**Screen Shot:**

****

**Task no 7:**

#include <iostream>

using namespace std;

struct Node

{

int data, height;

Node\* left;

Node\* right;

};

class AVLTree

{

private:

Node\* root;

public:

AVLTree()

{

root = nullptr;

}

int max(int a, int b)

{

if (a > b)

{

return a;

}

else

{

return b;

}

}

int getHeight(Node\* node)

{

if (node == nullptr)

{

return 0;

}

else

{

return node->height;

}

}

int getBalance(Node\* node)

{

if (node == nullptr)

{

return 0;

}

else

{

int leftSubtreeHeight = getHeight(node->left);

int rightSubtreeHeight = getHeight(node->right);

int balanceFactor = leftSubtreeHeight - rightSubtreeHeight;

return balanceFactor;

}

}

Node\* rotateRight(Node\* y)

{

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

int leftYHeight = getHeight(y->left);

int rightYHeight = getHeight(y->right);

int yMaxSubtreeHeight = max(leftYHeight, rightYHeight);

int updatedYHeight = yMaxSubtreeHeight + 1;

y->height = updatedYHeight;

int leftXHeight = getHeight(x->left);

int rightXHeight = getHeight(x->right);

int xMaxSubtreeHeight = max(leftXHeight, rightXHeight);

int updatedXHeight = xMaxSubtreeHeight + 1;

x->height = updatedXHeight;

return x;

}

Node\* rotateLeft(Node\* x)

{

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

int xLeftHeight = getHeight(x->left);

int xRightHeight = getHeight(x->right);

int xMaxSubtreeHeight = max(xLeftHeight, xRightHeight);

int updatedXHeight = xMaxSubtreeHeight + 1;

x->height = updatedXHeight;

int yLeftHeight = getHeight(y->left);

int yRightHeight = getHeight(y->right);

int yMaxSubtreeHeight = max(yLeftHeight, yRightHeight);

int updatedYHeight = yMaxSubtreeHeight + 1;

y->height = updatedYHeight;

return y;

}

Node\* insert(Node\* node, int data)

{

if (node == nullptr)

{

Node\* newNode = new Node;

newNode->data = data;

newNode->left = nullptr;

newNode->right = nullptr;

newNode->height = 1;

return newNode;

}

if (data < node->data)

{

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

}

else if (data > node->data)

{

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

}

else

{

return node;

}

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

int maxSubtreeHeight = max(leftHeight, rightHeight);

int updatedHeight = 1 + maxSubtreeHeight;

node->height = updatedHeight;

int balance = getBalance(node);

// Left Left Case

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

{

return rotateRight(node);

}

// Right Right Case

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

{

return rotateLeft(node);

}

// Left Right Case

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

{

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

return rotateRight(node);

}

// Right Left Case

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

{

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

return rotateLeft(node);

}

return node;

}

Node\* minValueNode(Node\* node)

{

Node\* current = node;

while (current->left != nullptr)

{

current = current->left;

}

return current;

}

Node\* deleteNode(Node\* root, int k)

{

if (root == nullptr)

{

return root;

}

if (k < root->data)

{

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

}

else if (k > root->data)

{

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

}

else

{

if (root->left == nullptr || root->right == nullptr)

{

Node\* temp;

if (root->left)

{

temp = root->left;

}

else

{

temp = root->right;

}

if (temp == nullptr)

{

temp = root;

root = nullptr;

}

else

\*root = \*temp;

delete temp;

}

else

{

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

root->data = temp->data;

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

}

}

if (root == nullptr)

{

return root;

}

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

int maxSubtreeHeight = (leftHeight > rightHeight) ? leftHeight : rightHeight;

int updatedHeight = 1 + maxSubtreeHeight;

root->height = updatedHeight;

int balance = getBalance(root);

// Left Left Case

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

{

return rotateRight(root);

}

// Left Right Case

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

{

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

return rotateRight(root);

}

// Right Right Case

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

{

return rotateLeft(root);

}

// Right Left Case

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

{

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

return rotateLeft(root);

}

return root;

}

void display(Node\* node)

{

if (node == nullptr)

{

return;

}

display(node->left);

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

display(node->right);

}

void constructAndDisplay()

{

int n;

cout << "Enter the number of elements: ";

cin >> n;

cout << "Enter the elements: ";

for (int i = 0; i < n; i++)

{

int data;

cin >> data;

root = insert(root, data);

}

cout << "AVL Tree after construction: ";

display(root);

cout << endl;

}

Node\* kthLargestNode(Node\* node, int& k)

{

if (node == nullptr)

{

return nullptr;

}

Node\* right = kthLargestNode(node->right, k);

if (right != nullptr)

{

return right;

}

k--;

if (k == 0)

{

return node;

}

return kthLargestNode(node->left, k);

}

Node\* deleteKthLargest(Node\* root, int k)

{

int count = k;

Node\* kthLargest = kthLargestNode(root, count);

if (kthLargest == nullptr)

{

return root;

}

root = deleteNode(root, kthLargest->data);

return root;

}

void deleteAndDisplay()

{

int k;

cout << "Enter the value of k: ";

cin >> k;

root = deleteKthLargest(root, k);

cout << "AVL Tree after deletion of the kth largest value node: ";

display(root);

cout << endl;

}

};

int main()

{

AVLTree a;

int choice;

while (true)

{

cout << "Menu:\n";

cout << "1. Construct AVL Tree\n";

cout << "2. Delete kth largest value node\n";

cout << "3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

if (choice == 1)

{

a.constructAndDisplay();

}

else if (choice == 2)

{

a.deleteAndDisplay();

}

else if (choice == 3)

{

cout << "Exiting...\n";

break;

}

else

{

cout << "Invalid choice. Please enter a valid option.\n";

}

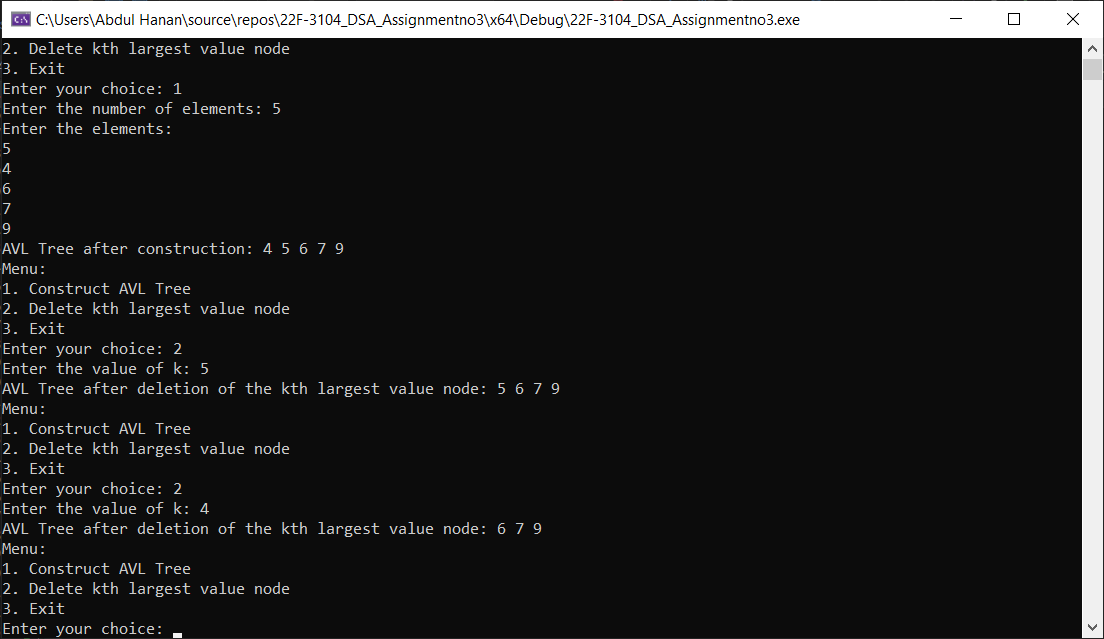
}

system("pause>0");

return 0;

}

**Screen Shot:**

****

**Task no 8:**

#include <iostream>

#include <string>

using namespace std;

struct SongNode {

string song\_name;

int frequency;

SongNode\* left, \* right;

SongNode\* next;

SongNode(string song\_name, int freq) {

this->song\_name = song\_name;

this->frequency = freq;

left = nullptr; right = nullptr; next = nullptr;

}

};

class AVLTree {

private:

SongNode\* root\_node;

SongNode\* insert(SongNode\* node, string song, int freq) { //insertion function

if (node == nullptr) //check for empty tree

return new SongNode(song, freq);

if (freq < node->frequency) //simple BST insertion

node->left = insert(node->left, song, freq);

else if (freq > node->frequency)

node->right = insert(node->right, song, freq);

else if (freq == node->frequency) {

SongNode\* current = node;

while (current->next != nullptr) {

current = current->next;

}

current->next = new SongNode(song, freq);

return node;

}

node->frequency = 1 + max\_num(height(node->left), height(node->right)); //height updation

int balance = balance\_factor(node); //for checking whether tree are balanced or not

if (balance > 1 && freq < node->left->frequency) //left-left case

return rightRotate(node);

if (balance < -1 && freq > node->right->frequency) //right-right case

return leftRotate(node);

if (balance > 1 && freq > node->left->frequency) { // left-right case

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

return rightRotate(node);

}

if (balance < -1 && freq < node->right->frequency) { // right-left case

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

return leftRotate(node);

}

return node;

}

int balance\_factor(SongNode\* node) {

if (node == nullptr) { return 0; }

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

}

int height(SongNode\* node) { //getter for height

if (node == nullptr) {

return 0;

}

return max\_num(height(node->left), height(node->right)) + 1;

}

int max\_num(int num1, int num2) { //for getting maximum value

if (num1 > num2) {

return num1;

}

else {

return num2;

}

}

SongNode\* rightRotate(SongNode\* y) { //right rotation

SongNode\* x = y->left; //left of subtree

SongNode\* T2 = x->right; //right of y subtree

x->right = y; //swapping values

y->left = T2; //rotation of subtree y

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

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

return x;

}

SongNode\* leftRotate(SongNode\* x) { //left rotation

SongNode\* y = x->right; //right of subtree

SongNode\* T2 = y->left; //left of x subtree

y->left = x; //right of subtree

x->right = T2; //rotation of x subtree

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

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

return y;

}

SongNode\* findSong(SongNode\* node, string song\_name) {

if (node == nullptr) {

return nullptr;

}

SongNode\* current\_song = node;

while (current\_song != nullptr) {

if (current\_song->song\_name == song\_name) {

return current\_song;

}

current\_song = current\_song->next;

}

SongNode\* left = findSong(node->left, song\_name);

if (left != nullptr) { return left; }

SongNode\* right = findSong(node->right, song\_name);

if (right != nullptr) { return right; }

return nullptr;

}

SongNode\* playSong(SongNode\* node, string song\_name) {

if (node == nullptr) {

return nullptr;

}

SongNode\* current\_song\_node = node;

while (current\_song\_node != nullptr) {

if (current\_song\_node->song\_name == song\_name) {

current\_song\_node->frequency++;

cout << "\nSong '" << current\_song\_node->song\_name << "' has been played.\n";

cout << "New frequency: " << current\_song\_node->frequency << endl;

return current\_song\_node;

}

current\_song\_node = current\_song\_node->next;

}

SongNode\* left = playSong(node->left, song\_name);

if (left != nullptr) { return left; }

SongNode\* right = playSong(node->right, song\_name);

if (right != nullptr) { return right; }

return nullptr;

}

void inorderTraversal(SongNode\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

SongNode\* current = root;

while (current != nullptr) {

cout << "\nSong name: " << current->song\_name << endl;

cout << "Frequency: " << current->frequency << endl;

current = current->next;

}

inorderTraversal(root->right);

}

}

public:

AVLTree() {

root\_node = nullptr;

}

void play\_song(string song\_name) {

playSong(root\_node, song\_name);

}

void insert\_song(string song\_name, int freq) {

root\_node = insert(root\_node, song\_name, freq);

}

void display\_plalist() {

cout << "\nPlaylist: ";

inorderTraversal(root\_node);

cout << endl;

}

};

int main()

{

int choice, frequency;

string song\_name;

AVLTree playlist;

playlist.insert\_song("Song\_A", 1);

playlist.insert\_song("Song\_B", 5);

playlist.insert\_song("Song\_C", 9);

playlist.insert\_song("Song\_D", 2);

playlist.insert\_song("Song\_E", 4);

playlist.insert\_song("Song\_F", 6);

playlist.insert\_song("Song\_G", 8);

playlist.insert\_song("Song\_H", 3);

playlist.insert\_song("Song\_I", 7);

playlist.insert\_song("Song\_J", 9);

playlist.insert\_song("Song\_K", 5);

while (true) {

cout << "1. Insert songs in the playlist\n";

cout << "2. Display songs\n";

cout << "3. Play songs\n";

cout << "0. Exit\n";

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

if (choice == 1) {

cout << "Enter the name: "; cin >> song\_name;

cout << "Enter the frequency of your song: "; cin >> frequency;

playlist.insert\_song(song\_name, frequency);

}

else if (choice == 2) {

playlist.display\_plalist();

}

else if (choice == 3) {

cout << "Enter the song to play: "; cin.ignore(); getline(cin, song\_name);

playlist.play\_song(song\_name);

}

else if (choice == 0) {

cout << "Thanks for using our program. Good Bye.\n";

break;

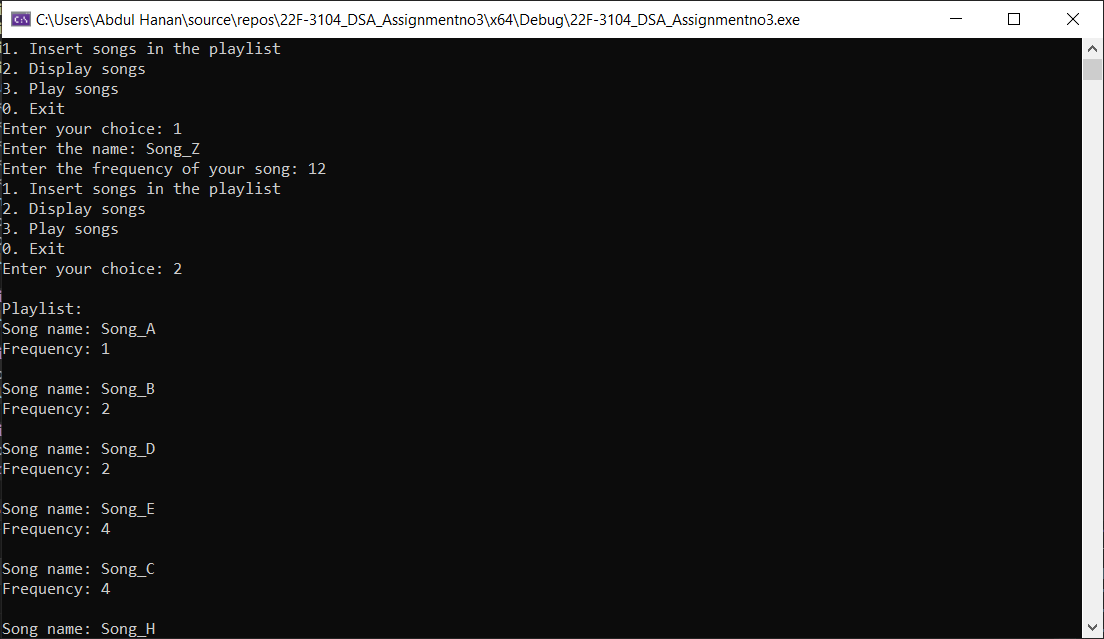
}

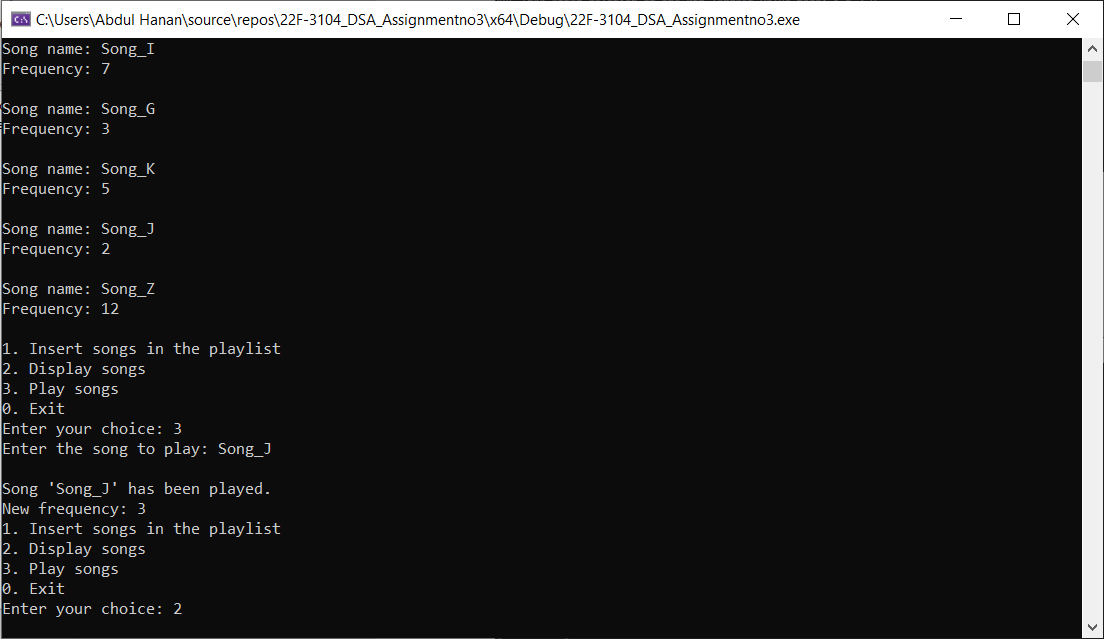
}

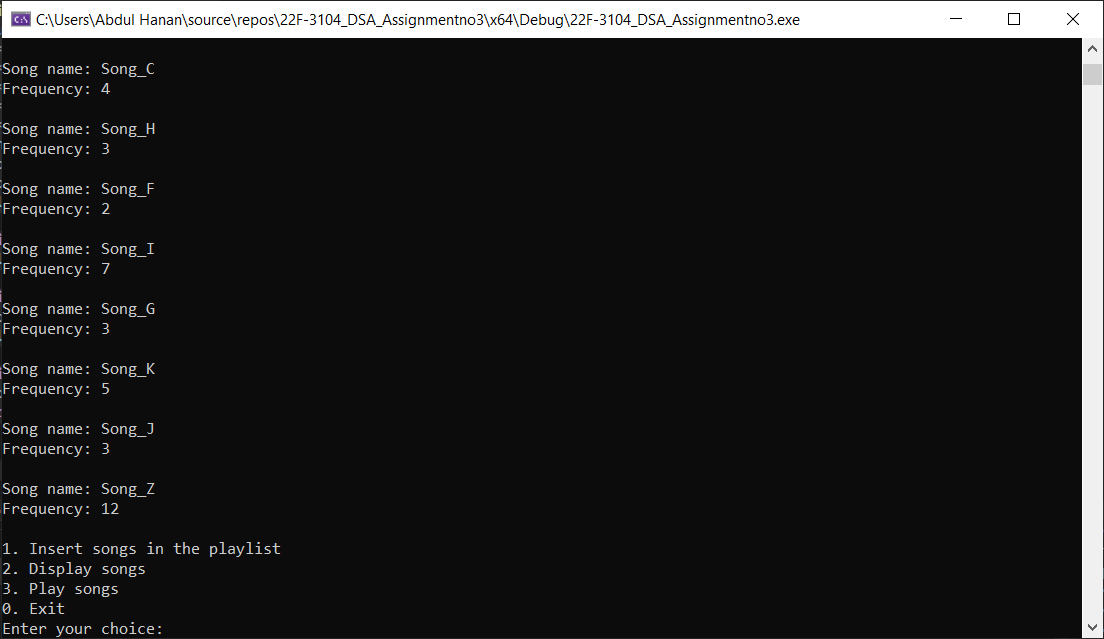
return 0;

}

**Screen Shot:**

****

****

****

**Task no 9:**

#include <iostream>

using namespace std;

struct Patient

{

string name;

int condition; // more higher value more severe condition.

int waitingRoom; // new field to store the waiting room number

Patient() { }

Patient(string name, int condition, int waitingRoom)

{

this->name = name;

this->condition = condition;

this->waitingRoom = waitingRoom;

}

};

class MaxHeap

{

private:

Patient\* heapArray;

int capacity;

int size;

public:

MaxHeap() { }

MaxHeap(int capacity)

{

this->capacity = capacity;

this->heapArray = new Patient[capacity];

this->size = 0;

}

void swap(Patient& a, Patient& b)

{

Patient temp = a;

a = b;

b = temp;

}

void heapify(int index)

{

int largest = index;

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

if (leftChild < size && heapArray[leftChild].condition > heapArray[largest].condition)

{

largest = leftChild;

}

if (rightChild < size && heapArray[rightChild].condition > heapArray[largest].condition)

{

largest = rightChild;

}

if (largest != index)

{

swap(heapArray[index], heapArray[largest]);

heapify(largest);

}

}

void insert(Patient patient, int waitingRoom)

{

if (size >= capacity)

{

cout << "Heap is full. Cannot add more patients." << endl;

return;

}

int currentIndex = size;

heapArray[size++] = Patient(patient.name, patient.condition, waitingRoom);

while (currentIndex != 0 && heapArray[(currentIndex - 1) / 2].condition < heapArray[currentIndex].condition)

{

swap(heapArray[currentIndex], heapArray[(currentIndex - 1) / 2]);

currentIndex = (currentIndex - 1) / 2;

}

}

Patient extractMax()

{

if (size <= 0)

{

cout << "No patients in the waiting system." << endl;

return Patient("", -1, -1);

}

Patient maxPatient = heapArray[0];

heapArray[0] = heapArray[--size];

heapify(0);

return maxPatient;

}

int getSize()

{

return size;

}

void display()

{

cout << "Current State of the Waiting System:" << endl;

cout << "----------------------------------" << endl;

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

{

cout << "Patient: " << heapArray[i].name << ", Condition: " << heapArray[i].condition

<< ", Waiting Room: " << heapArray[i].waitingRoom << endl;

}

cout << "----------------------------------" << endl;

}

};

int main()

{

int capacity, numWaitingRooms;

cout << "Enter the capacity of the beds: ";

cin >> capacity;

cout << "Enter the number of waiting rooms: ";

cin >> numWaitingRooms;

MaxHeap maxHeap(capacity); // Assume hospital can handle 'capacity' patients

int choice;

while (true)

{

cout << "Hospital Patient Waiting System Menu:" << endl;

cout << "1. Add Patient" << endl;

cout << "2. Allot Bed to Patient" << endl;

cout << "3. Number of Patients in Waiting System" << endl;

cout << "4. Display Waiting System" << endl;

cout << "5. Exit" << endl;

cin >> choice;

if (choice == 1)

{

string name;

int condition, waitingRoom;

cout << "Enter patient name: ";

cin >> name;

cout << "Enter patient condition: ";

cin >> condition;

cout << "Enter patient waiting room: ";

cin >> waitingRoom;

maxHeap.insert(Patient(name, condition, waitingRoom), waitingRoom);

cout << "Patient added to the waiting system." << endl;

}

else if (choice == 2)

{

Patient patient = maxHeap.extractMax();

if (patient.condition != -1)

{

cout << "Bed allotted to patient: " << patient.name << " with condition: " << patient.condition << " in waiting room: " << patient.waitingRoom << endl;

}

}

else if (choice == 3)

{

int numPatients = maxHeap.getSize();

cout << "Number of patients in the waiting system: " << numPatients << endl;

}

else if (choice == 4)

{

maxHeap.display();

}

else if (choice == 5)

{

cout << "Exiting program. Goodbye!" << endl;

break;

}

else

{

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

}

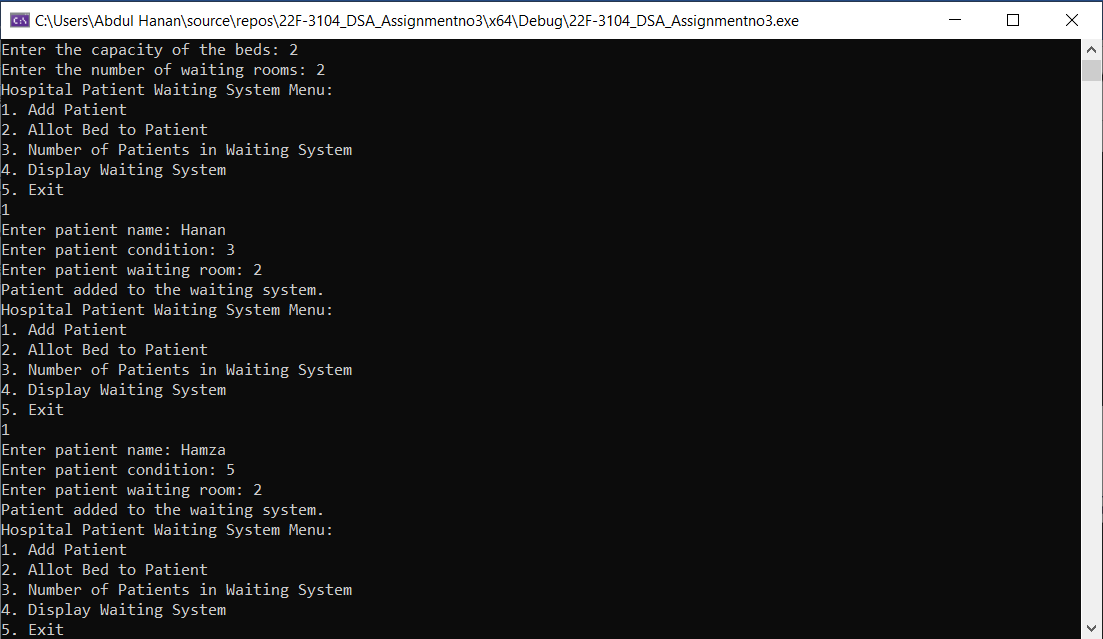
}

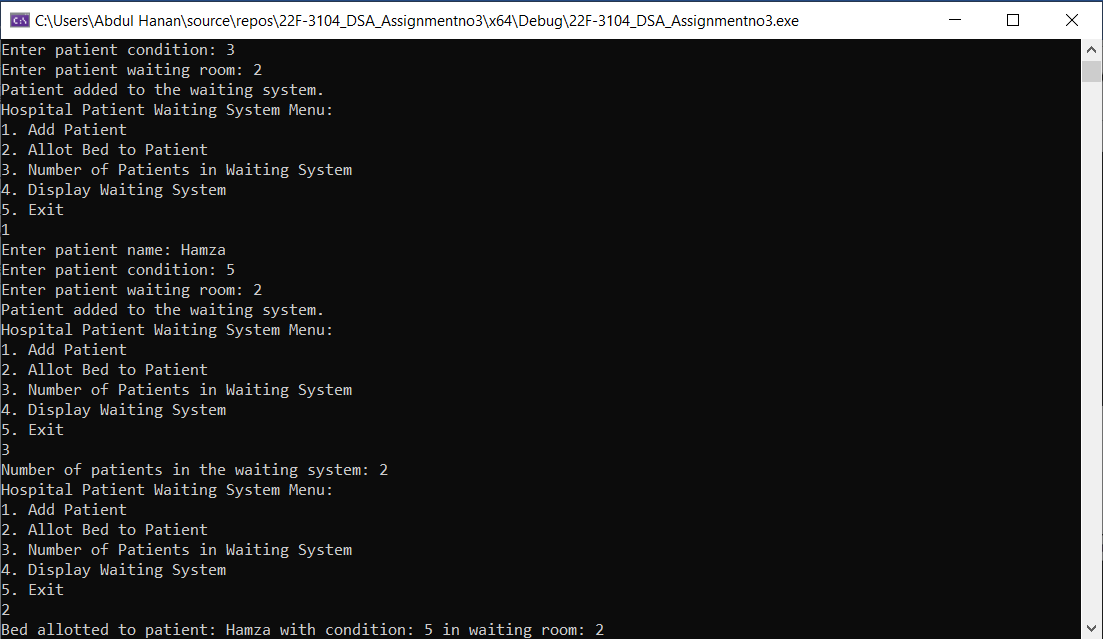
system("pause>0");

return 0;

}

**Screen Shot:**

****

****

