Data Structure

Assignment 4

Name: Faizan Tariq

Roll no: 22F-3858

Section : 3D

TASK 1:

#include<iostream>

#include<fstream>

using namespace std;

struct Node {

int data;

Node\* next;

Node() {

data = 0;

next = NULL;

}

Node(int data) {

this->data = data;

next = NULL;

}

};

class QueueLinkedListAdt {

Node\* rear;

Node\* front;

int numItems;

public:

QueueLinkedListAdt() {

rear = NULL;

front = NULL;

numItems = 0;

}

bool isEmpty() {

return numItems == 0;

}

void enqueue(int data) {

Node\* newNode = new Node(data);

if (isEmpty()) {

front = newNode;

rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

numItems++;

}

void dequeue(int& num) {

if (isEmpty()) {

cout << "\nQueue is already empty.\n";

return;

}

numItems--;

num = front->data;

Node\* deletedNode = front;

front = front->next;

delete deletedNode;

}

void makeNull() {

int x;

while (!isEmpty()) {

dequeue(x);

}

}

~QueueLinkedListAdt() {

makeNull();

}

};

struct StackNode {

int data;

StackNode\* next;

StackNode() {

data = 0;

next = NULL;

}

StackNode(int data) {

this->data = data;

next = NULL;

}

};

class StackAdt {

StackNode\* top;

public:

StackAdt() {

top = NULL;

}

bool isEmpty() {

return top == NULL;

}

bool push(int data) {

StackNode\* newNode = new StackNode(data);

newNode->next = top;

top = newNode;

return true;

}

bool pop(int& tempData) {

if (isEmpty()) {

return false;

}

StackNode\* tempNode = top;

tempData = tempNode->data;

top = tempNode->next;

delete tempNode;

}

void displayStack() {

StackNode\* cur = top;

cout << endl;

while (cur != NULL) {

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

cur = cur->next;

}

cout << endl;

}

~StackAdt() {

int n;

while (top != NULL) {

pop(n);

}

delete top;

}

};

struct Cell {

bool isObstacle;

bool isVisited;

int cost;

char type; // ' ' for empty, 'S' for start, 'G' for goal, '\*' for path

Cell() {

isObstacle = false;

isVisited= false;

cost = 0;

type = ' ';

}

};

class RobotNavGraph {

Cell\*\* grid;

int size;

int startRow, startCol;

int goalRow, goalCol;

int cost = 0;

public:

RobotNavGraph(int S) : size(S) {

grid = new Cell\*[size];

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

grid[i] = new Cell[size];

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

grid[i][j].isObstacle = false;

grid[i][j].isVisited = false;

grid[i][j].cost = 0;

grid[i][j].type = ' ';

}

}

}

void loadFromFile() {

ifstream fin("input.txt");

if (!fin.is\_open()) {

cout << "Cannot open file.\n";

return;

}

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

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

fin >> grid[i][j].type;

if (grid[i][j].type == '1') {

grid[i][j].isObstacle = true;

}

else if (grid[i][j].type == 'S') {

startRow = i;

startCol = j;

}

else if (grid[i][j].type == 'G') {

goalRow = i;

goalCol = j;

}

}

}

fin.close();

}

void displayPath() {

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

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

cout << grid[i][j].type << "\t";

}

cout << endl;

}

}

void setGoal\_Destination() {

grid[14][0].type = 'S';

grid[0][14].type = 'G';

grid[0][14].isObstacle = true;

goalRow = 0;

goalCol = 14;

startCol = 0;

startRow = 14;

}

void bfs() {

QueueLinkedListAdt q;

q.enqueue(startRow \* size + startCol);

grid[startRow][startCol].isVisited = true;

grid[startRow][startCol].cost = 0;

bool moreTraverse = true;

while (!q.isEmpty()) {

int curPos;

q.dequeue(curPos);

int curRow = curPos / size;

int curCol = curPos % size;

for (int i = -1; i <= 1 && moreTraverse; i++) {

for (int j = -1; j <= 1 && moreTraverse; j++) {

int newRow = curRow + i;

int newCol = curCol + j;

if (newRow >= 0 && newRow < size && newCol >= 0 && newCol < size) {

if (!grid[newRow][newCol].isObstacle && !grid[newRow][newCol].isVisited) {

q.enqueue(newRow \* size + newCol);

grid[newRow][newCol].isVisited = true;

grid[newRow][newCol].type = '\*'; // Mark the path

if (i == 0 && j == 0) {

}

else if (i == 0) {

cost += 2;

cout << "\nMove UP\n";

}

else if (i == 0 || j == 0) {

cost+= 2; // Move up or right, cost 2

cout << "\nMove Right\n";

}

else{

cout << "\nMove Diagonal\n";

cost+= 3; // Move up or right, cost 2

grid[newRow][newCol].cost = grid[curRow][curCol].cost + 3; // Diagonal move, cost 3

}

// displayPath();

}

if (grid[newRow][newCol].type == 'G') {

cout << "\nGOAL REACHEDn\n";

cout << "YOUR COST WITH BFS :" <<cost<< endl;

moreTraverse = false;

break;

}

}

}

}

}

}

void dfs() {

cost = 0;

StackAdt s;

s.push(startRow \* size + startCol);

grid[startRow][startCol].isVisited = true;

grid[startRow][startCol].cost = 0;

bool moreTraverse = true;

while (!s.isEmpty() && moreTraverse) {

int curPos;

s.pop(curPos);

int curRow = curPos / size;

int curCol = curPos % size;

for (int i = -1; i <= 1 && moreTraverse; i++) {

for (int j = -1; j <= 1 && moreTraverse; j++) {

int newRow = curRow + i;

int newCol = curCol + j;

if (newRow >= 0 && newRow < size && newCol >= 0 && newCol < size) {

if (!grid[newRow][newCol].isObstacle && !grid[newRow][newCol].isVisited) {

s.push(newRow \* size + newCol);

grid[newRow][newCol].isVisited = true;

grid[newRow][newCol].type = '\*'; // Mark the path

if (i == 0 && j == 0) {

// for the same cell, no change in cost

}

else if (i == 0 || j == 0) {

cost += 2; // Move up or right, cost 2

cout << "\nMOVED UP OR RIGHT\n";

}

else {

cout << "\nMOVED DIAGONAL\n";

cost += 3; // Move diagonally, cost 3

grid[newRow][newCol].cost = grid[curRow][curCol].cost + 3; // Diagonal move, cost 3

}

// displayPath();

if (grid[newRow][newCol].type == 'G' || grid[newRow][newCol].isObstacle) {

cout << "\nGOAL REACHED\n";

cout << "YOUR COST WITH DFS: " << cost << endl;

moreTraverse = false;

break;

}

}

}

}

}

}

cout << "YOUR COST WITH DFS :" << cost<<endl;

}

};

int main() {

RobotNavGraph robot(15);

RobotNavGraph robot2(15);

robot.loadFromFile();

robot.setGoal\_Destination();

robot.displayPath();

// Perform BFS

robot.bfs();

robot.displayPath();

cout << endl;

robot2.loadFromFile();

robot2.setGoal\_Destination();

robot2.dfs();

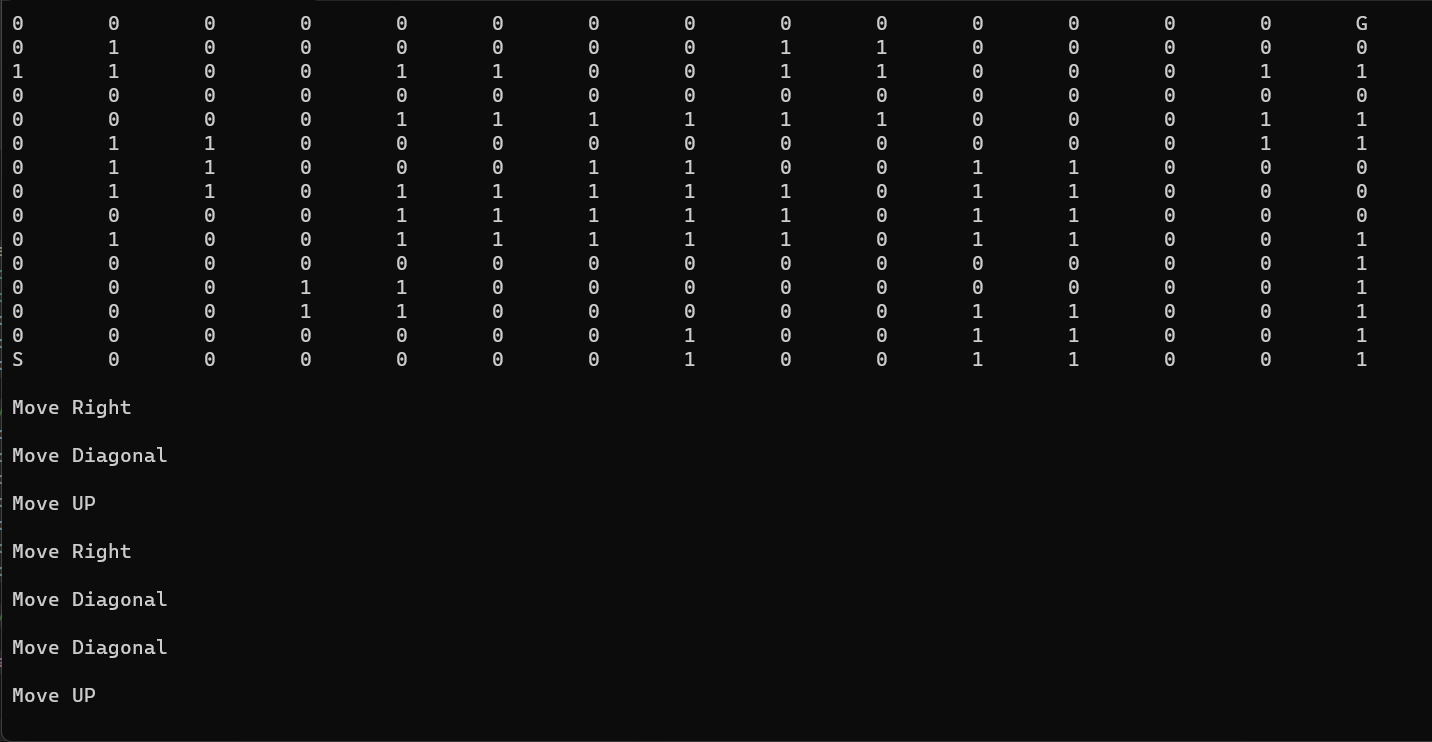
robot2.displayPath();

// Display the final grid after BFS traversal

return 0;

}

**OUTPUT:**

****

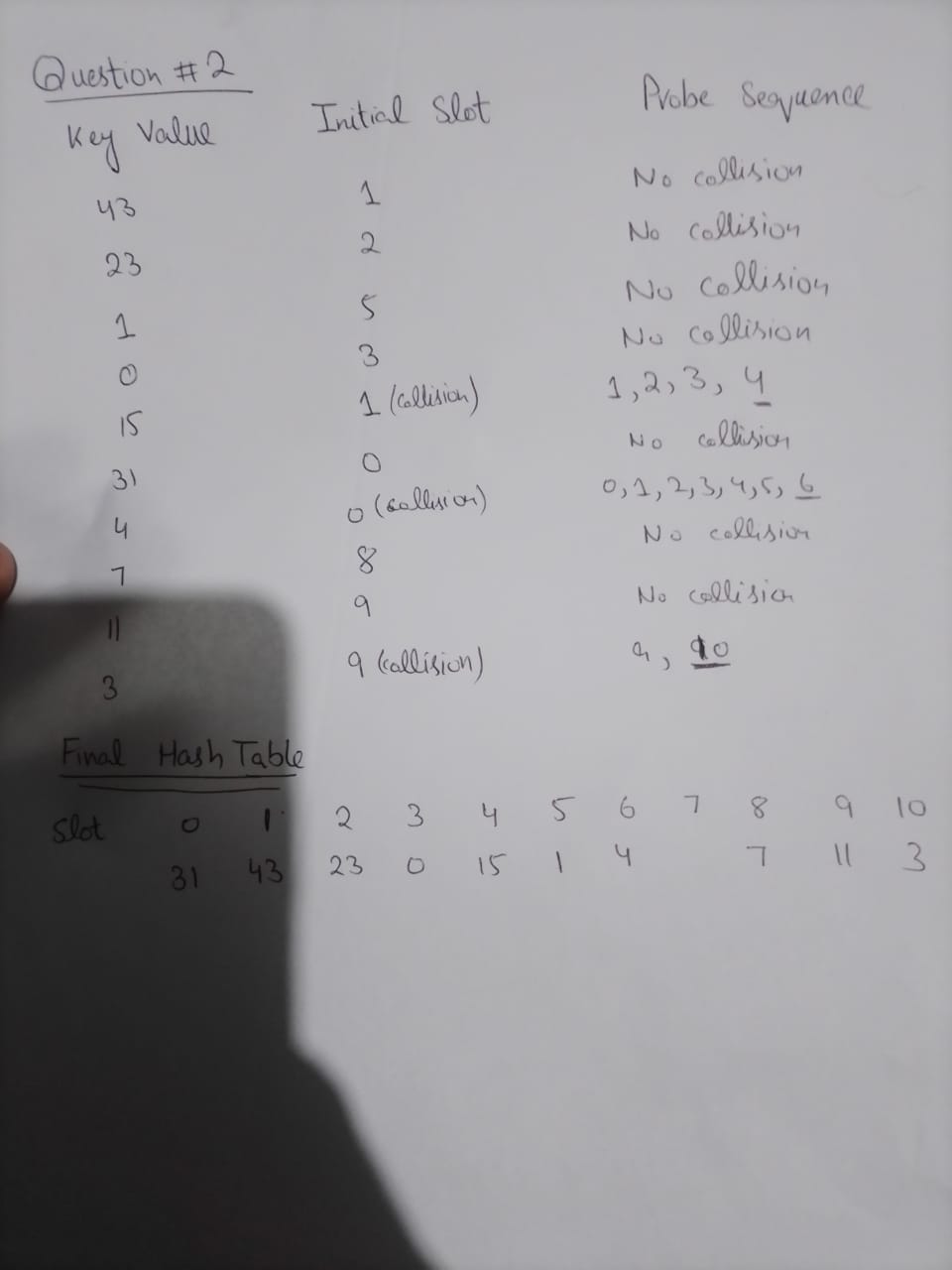
**A black background with white numbers

Description automatically generated**

**A group of white text on a black background

Description automatically generated**

**TASK 2:**



**TASK 3:**

#include<iostream>

using namespace std;

class HashClass {

int size = 5;

int\* hashTable;

int count = 0;

int probingType = 0; // 0 fir linear 1 for quadratic and 2 for double hashing

public:

HashClass(int Size , int proobingType) {

size = Size;

probingType = proobingType;

hashTable = new int[size];

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

hashTable[i] = -1;

}

};

int hashFunc(int key) {

return key % size;

}

int doubleHashing(int address) {

int hash2 = 7 - (address % 7); // A secondary hash function

return (address + hash2) % size;

}

void insert(int data) {

if (loadFactor() > 0.7) {

//cout << "REHASHED" << endl;

HashClass temp(size \* 2, probingType);

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

{

temp.insert(hashTable[i]);

}

temp.insert(data);

delete[] hashTable;

this->hashTable = temp.hashTable;

this->size = temp.size;

this->count = temp.count;

}

else if (count < size) {

++count;

if (probingType == 0) { // linear

int address = hashFunc(data);

while (hashTable[address] != -1) {

address = (address + 1) % size;

}

hashTable[address] = data;

}

else if (probingType == 1) { // quadratic

int address = hashFunc(data);

int i = 1;

while (hashTable[address] != -1) {

address = (address + i\*i) % size;

i++;

}

hashTable[address] = data;

}

else if (probingType == 2) { // double hashing

int address = doubleHashing(data);

while (hashTable[address] != -1) {

address = (address + 1) % size;

}

hashTable[address] = data;

}

}

}

int\* getAddress() {

return hashTable;

}

bool search(int data) {

int address = hashFunc(data);

while (hashTable[address] != -1) {

if (hashTable[address] == data) {

return true;

}

address = (address + 1) % size;

}

return false;

}

double loadFactor() {

return static\_cast<double>(count / size);

}

void deleteData(int data) {

int address = hashFunc(data);

while (hashTable[address] != -1) {

if (hashTable[address] == data) {

hashTable[address] = -2;

}

address = (address + 1) % size;

}

}

void displayAll() {

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

{

cout << " " << hashTable[i];

}

}

};

int main() {

HashClass linear\_table(15, 0);

HashClass quad\_table(15, 1);

HashClass double\_table(15, 2);

int keys[16] = { 17, 26, 15, 9, 11, 43, 75, 19, 35, 45, 55, 9, 10, 21, 61, 23 };

for (int i = 0; i < 16; i++)

{

linear\_table.insert(keys[i]);

quad\_table.insert(keys[i]);

double\_table.insert(keys[i]);

}

cout << "\nDATA IN LINEAR HASHTABLE :\n";

linear\_table.displayAll();

cout << "\nDATA IN QUAD HASHTABLE :\n";

quad\_table.displayAll();

cout << "\nDATA IN DOUBLE HASHTABLE :\n";

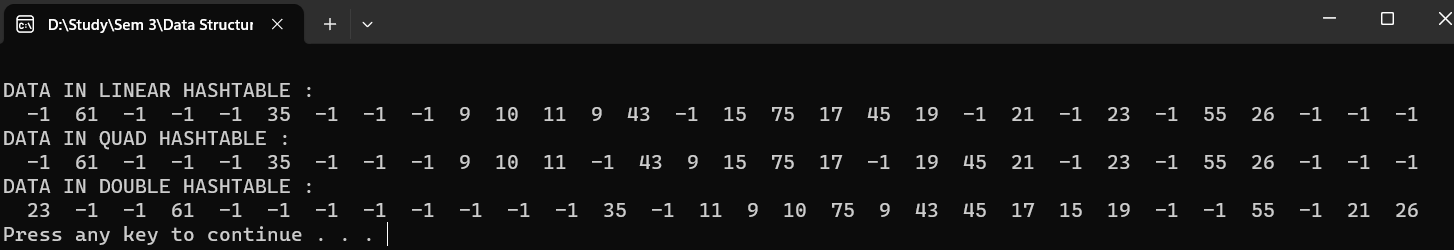
double\_table.displayAll();

cout << endl;

system("pause");

return 0;

}

****

**TASK 4:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int key) : data(key), next(NULL) {}

};

class HashTable {

static const int size = 15;

Node\* hashtable[size];

public:

HashTable() {

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

hashtable[i] = NULL;

}

}

int hashFunc(int key) {

return key % size;

}

void insert(int key) {

int index = hashFunc(key);

Node\* newNode = new Node(key);

if (hashtable[index] == NULL) {

hashtable[index] = newNode;

}

else {

newNode->next = hashtable[index];// insert at start

hashtable[index] = newNode;

}

}

bool search(int key) {

int index = hashFunc(key);

Node\* cur = hashtable[index];

while (cur != NULL) {

if (cur->data == key) {

return true;

}

cur = cur->next;

}

return false;

}

void deleteVal(int key) {

int index = hashFunc(key);

Node\* cur = hashtable[index];

Node\* prev = NULL;// for deleting 1st node

while (cur != NULL && cur->data != key) {

prev = cur;

cur = cur->next;

}

if (cur != NULL) {

if (prev == NULL) { // for deleting 1st node

hashtable[index] = cur->next;

}

else {

prev->next = cur->next;

}

delete cur;

}

}

void displayALL() {

int i = 0;

cout << "\n Data in Chainigng hashtable :\n";

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

{

Node\* temp = hashtable[i];

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

}

};

const int BUCKET\_SIZE = 3;

class BucketHashTable {

int size;

int count; // count number of items

int\*\* hashtables;

public:

BucketHashTable(int S) : size(S) {

count = 0;

hashtables = new int\* [size];

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

hashtables[i] = new int[BUCKET\_SIZE];

for (int j = 0; j < BUCKET\_SIZE; j++) {

hashtables[i][j] = -1;

}

}

}

int hashFunc(int key) {

return key % size;

}

double loadFactor() {

return static\_cast<double>(count) / size;

}

void insert(int data) {

if (loadFactor() > 0.7) {

rehash(data);

}

else {

int index = hashFunc(data) % size;

int bucketIndex = -1;

for (int i = 0; i < BUCKET\_SIZE; ++i) {

if (hashtables[index][i] == -1) {

bucketIndex = i;

break;

}

}

if (bucketIndex != -1) {

hashtables[index][bucketIndex] = data;

count++;

}

}

}

void rehash(int data) {

BucketHashTable temp(size \* 2);

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

{

for (int j = 0; j < BUCKET\_SIZE; j++) {

int tempValue = hashtables[i][j];

if (tempValue != -1) {

temp.insert(tempValue);

}

}

}

temp.insert(data);

delete[] hashtables;

this->hashtables = temp.hashtables;

this->count = temp.count;

this->size = temp.size;

}

void displayALL() {

cout << "\nData in Bucketing Hashtable :\n";

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

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

for (int j = 0; j < BUCKET\_SIZE; ++j) {

if (hashtables[i][j] != -1) {

cout << " " << hashtables[i][j];

}

}

cout << endl;

}

}

};

int main() {

HashTable table;

BucketHashTable bucket(15);

int keys[16] = { 17, 26, 15, 9, 11, 43, 75, 19, 35, 45, 55, 9, 10, 21, 61, 23 };

for (int i = 0; i < 16; i++)

{

table.insert(keys[i]);

bucket.insert(keys[i]);

}

table.displayALL();

bucket.displayALL();

system("pause");

return 0;

}

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**