****

**Department of AI&DS**

FAST – National University of Computer & Emerging Sciences

Chiniot-Faisalabad Campus.

**Data Structures Lab**

Lab Instructor

**Prof. Yousuf**

**Group Members:**

|  |  |
| --- | --- |
| **Name** | **Roll Number** |
| Abdullah Niazi | 23F-0017 |
| Muhammad Ahmad | 23F-0022 |

**Section: BSAI-3A**

**Outputs:**

A screenshot of a computer

Description automatically generated

**Main Menu**

A screenshot of a computer

Description automatically generated

**User Representation**

A screenshot of a computer

Description automatically generated

**Sign-In**

A black screen with white text

Description automatically generated

**Logged-In as a User**

A screenshot of a computer program

Description automatically generated

**Logged-In as Admin**

A black screen with white text

Description automatically generated

**Notifications**

A black and white screen with white text

Description automatically generated

**Followers/Friends**A screenshot of a computer

Description automatically generated

**Messages**A screenshot of a computer

Description automatically generated

**Accept Friend Requests**A screenshot of a video game

Description automatically generated

**Feed**A screenshot of a computer

Description automatically generated

**Change Password**

A black and white image of a road

Description automatically generated with medium confidence

**Code:**

**AdjNode.h**

#ifndef ADJNODE\_H

#define ADJNODE\_H

#include"List.h"

struct AdjNode {

userData data;

AdjNode\* next;

List\* edges;

int vertexID;

AdjNode() : vertexID(0), next(nullptr), edges(nullptr) {}

};

#endif // ADJNODE\_H

**Graph.h**

#ifndef ADJLIST\_H

#define ADJLIST\_H

#include "Structures.h"

#include"Tree.h"

#include"AdjNode.h"

#include"HashMap.h"

class baseAdjList {

public:

virtual void insertVertex(userData val) = 0;

virtual void insertAtVertex(int vertex, edgeData val) = 0;

virtual void deleteVertex(string vertex) = 0;

virtual AdjNode\* findVertex(string username) = 0;

virtual AdjNode\* findVertex(int id) = 0;

virtual AdjNode\* getFirstNode() = 0;

};

class AdjList : public baseAdjList {

private:

Bst<string>\*obj;

AdjNode\* head;

AdjNode\* current;

HashTable map;

int vertexCount;

public:

AdjList();

~AdjList();

void insertVertex(userData val);

void insertAtVertex(int vertex, edgeData val);

void deleteVertex(string vertex);

AdjNode\* findVertex(string username);

AdjNode\* findVertex(int id);

AdjNode\* getFirstNode();

void print();

void printNames();

};

#endif // ADJLIST\_H

**Graph.cpp**

#include "Graph.h"

#include "Structures.h"

AdjList::AdjList() : head(nullptr), current(nullptr), vertexCount(0) {

obj = new Bst<string>;

}

void AdjList::insertVertex(userData val) {

AdjNode\* newVertex = new AdjNode;

newVertex->data = val;

newVertex->vertexID = vertexCount++;

obj->insert(newVertex);

map.insert(newVertex->vertexID, newVertex->data.username, newVertex->data.password,newVertex);

newVertex->edges = new List;

if (head == nullptr) {

head = newVertex;

}

else {

current = head;

while (current->next != nullptr) {

current = current->next;

}

current->next = newVertex;

}

}

void AdjList::insertAtVertex(int vertex, edgeData val) {

current = head;

while (current != nullptr && current->vertexID != vertex) {

current = current->next;

}

if (current == nullptr) {

cout << "Vertex# " << vertex << " Not Found!" << endl;

return;

}

current->edges->insert\_at\_end(val);

}

void AdjList::deleteVertex(string vertex) {

if (head == nullptr) return;

if (head->data.username == vertex) {

AdjNode\* temp = head;

head = head->next;

delete temp->edges;

delete temp;

return;

}

current = head;

while (current->next != nullptr && current->next->data.username != vertex) {

current = current->next;

}

if (current->next != nullptr) {

AdjNode\* temp = current->next;

current->next = current->next->next;

delete temp->edges;

delete temp;

}

obj->deleteNode(vertex);

}

AdjNode\* AdjList::findVertex(int id) {

return map.getUser(id);

}

//AdjNode\* AdjList::findVertex(string username) {

// current = head;

// while (current != nullptr) {

// if (current->data.username == username) return current;

// current = current->next;

// }

// return nullptr;

//}

AdjNode\* AdjList::findVertex(string username) {

AdjNode\* result = obj->findNode(username);

return result;

}

AdjList::~AdjList() {

while (head != nullptr) {

AdjNode\* temp = head;

head = head->next;

delete temp->edges;

delete temp;

}

}

AdjNode\* AdjList::getFirstNode() {

return head;

}

void AdjList::print() {

current = head;

cout << "[Graph]" << endl;

while (current != nullptr) {

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

current->edges->print();

current = current->next;

}

cout << endl << "[TREE]" << endl;

cout << "[InOrder] : "; obj->printInOrder();

cout << endl;

cout << "[HASHMAP]" << endl;

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

cout << "[INDEX] : (USERNAME : PASSWORD)" << endl;

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

map.display();

}

void AdjList::printNames() {

current = head;

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

cout << "[ID] \t[USERNAME]" << endl;

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

while (current != nullptr) {

cout << "[" << current->vertexID << "] -> [" << current->data.username << "]" << endl;

current = current->next;

}

}

**Instagram.h**#include"Graph.h"

#include"UI.h"

#include <iostream>

#include <iomanip>

#include<string>

#include <ctime>

#include<fstream>

#include<conio.h>

using namespace std;

class Instagram :public UI {

private:

AdjList\* insta;

bool isUsernameUnique(const string& username);

void Return();

void changePassword(AdjNode\* obj, bool admin = true);

void viewFollowersList(AdjNode\* obj);

public:

void printAvailableUsers();

Instagram();

~Instagram();

void destroyAndCreate();

string getCurrentTimeAndDate();

void loggedIn(AdjNode\*& obj, bool admin = false);

void SignUp();

void SignIn();

void AdminMode();

void viewFeed(AdjNode\*& obj);

void viewMessages(AdjNode\*& obj);

void viewNotification(AdjNode\*& obj);

void viewFriendRequest(AdjNode\*& obj);

void sendMessages(AdjNode\*& obj);

void sendFriendRequest(AdjNode\*& obj);

private:

void approveAllRequests(AdjNode\*& obj);

void removeRequest(AdjNode\*& obj, string from);

public:

void acceptFriendRequest(AdjNode\*& obj);

private:

void ExportMessageToFile(AdjNode\*& obj, ofstream& messages);

void ExportNotificationsToFile(AdjNode\*& obj, ofstream& notifications);

void ExportPostsToFile(AdjNode\*& obj, ofstream& posts);

void ExportFriendRequestsToFile(AdjNode\*& obj, ofstream& friendRequests);

void ExportEdgesToFile(AdjNode\*& obj, ofstream& edges);

public:

void ExportToFile();

private:

void importMessagesFromFile(AdjNode\*& temp, ifstream& messages);

void importNotificationsFromFile(AdjNode\*& temp, ifstream& notifications);

void importPostsFromFile(AdjNode\*& temp, ifstream& posts);

void importFriendRequestsFromFile(AdjNode\*& temp, ifstream& friendRequests);

void importEdgesFromFile(AdjNode\*& temp, ifstream& edges);

public:

void importFromFile();

void print();

void mainMenu();

};

#pragma once

**Instagram.cpp**

#include"Instagram.h"

using namespace std;

bool Instagram::isUsernameUnique(const string& username) {

if (insta->findVertex(username) == nullptr) {

return true;

}

return false;

}

void Instagram::Return() {

printBoxed("Press Any [Key] to Return");

while (!\_kbhit()) {}

char ch = \_getch();

clear();

printBoxed("Returning to Main Menu...");

delay(1500);

}

void Instagram::changePassword(AdjNode\* obj, bool admin) {

clear();

printTitle("CHANGE PASSWORD");

string password;

if (admin == false) {

cout << "Enter Old Password : "; cin >> password;

if (obj->data.password == password) {

cout << "Enter New Password : "; cin >> password;

obj->data.password = password;

}

else {

printBoxed("Incorrect Password");

Return();

}

}

else {

cout << "Enter New Password : "; cin >> password;

obj->data.password = password;

}

clear();

printBoxed("Password Changed Successfully");

delay(1500);

}

void Instagram::viewFollowersList(AdjNode\* obj) {

clear();

printTitle("FOLLOWERS/FRIENDS");

obj->edges->print();

printSeparator('=');

Return();

}

void Instagram::printAvailableUsers() {

insta->printNames();

printSeparator('=');

}

Instagram::Instagram() {

insta = new AdjList;

}

Instagram::~Instagram() {

delete insta;

}

void Instagram::destroyAndCreate() {

delete insta;

insta = new AdjList;

}

string Instagram::getCurrentTimeAndDate() {

time\_t now = time(nullptr);

tm localTime = {};

localtime\_s(&localTime, &now);

char dateTime[20];

strftime(dateTime, sizeof(dateTime), "%Y-%m-%d %H:%M:%S", &localTime);

string str(dateTime);

return str;

}

void Instagram::loggedIn(AdjNode\*& obj, bool admin) {

if (!obj) {

printBoxed("Error : Invalid User Node.");

return;

}

char choice = ' ';

while (choice != 27) {

clear();

printTitle("Logged-In as " + obj->data.name);

printBoxed("========== MENU ==========");

printCentered("[1] View Feed");

printCentered("[2] View Messages");

printCentered("[3] Send Message");

printCentered("[4] View Friend Requests");

printCentered("[5] Send Friend Request");

printCentered("[6] View Notifications");

printCentered("[7] Accept/Reject Friend Request");

printCentered("[8] View Followers List");

printCentered("[9] Change Password");

printCentered("[Esc] Logout");

printSeparator('=');

choice = \_getch();

switch (choice) {

case '1':

clear();

viewFeed(obj);

break;

case '2':

clear();

viewMessages(obj);

break;

case '3':

clear();

sendMessages(obj);

break;

case '4':

clear();

viewFriendRequest(obj);

break;

case '5':

clear();

sendFriendRequest(obj);

break;

case '6':

clear();

viewNotification(obj);

break;

case '7':

clear();

acceptFriendRequest(obj);

break;

case '8':

viewFollowersList(obj);

break;

case '9':

changePassword(obj, admin);

break;

case 27:

clear();

printBoxed("Logged out successfully!");

break;

default:

clear();

printBoxed("Invalid choice. Please try again.");

delay(1500);

}

}

}

void Instagram::SignUp() {

printTitle("SIGN UP");

string username, password, name;

cout << "Enter Name : "; cin >> name;

cout << "Enter Username : "; cin >> username;

if (!isUsernameUnique(username)) {

printBoxed("Username already exists.Try another one.");

Return();

return;

}

cout << "Create a Password : "; cin >> password;

string timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

userData temp(username, password, name, timestamp);

insta->insertVertex(temp);

printBoxed("Signed-Up Successfully");

delay(1500);

}

void Instagram::SignIn() {

printTitle("SIGN IN");

string username;

string password;

cout << "Enter UserName : "; cin >> username;

AdjNode\* temp = insta->findVertex(username);

if (!temp) {

printBoxed("Error: Username not found.");

Return();

return;

}

if (temp->data.username == username) {

cout << "Enter Password : "; cin >> password;

if (password == temp->data.password) {

printBoxed("Logged-In Successfully");

string timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

delay(1000);

loggedIn(temp);

temp->data.lastLogin = timestamp;

}

else {

printBoxed("Error : Incorrect Password.");

Return();

return;

}

}

}

void Instagram::AdminMode() {

int id = 0;

string username = "";

cout << "Enter Password : "; cin >> username;

if (username == "MAYDAY" || username == "mayday") {

clear();

}

else {

return;

}

AdjNode\* temp = nullptr;

while (true) {

clear();

printTitle("ADMIN MODE");

cout << "Available Users:" << endl;

printAvailableUsers();

cout << "Select User by : [1] Username | [2] ID | [Esc] Main Menu" << endl;

char ch = \_getch();

if (ch == '1') {

cout << "Enter Username : "; cin >> username;

temp = insta->findVertex(username);

if (temp != nullptr) {

loggedIn(temp, true);

}

else {

clear();

printBoxed("Username not found. Try again.");

delay(1500);

clear();

}

}

else if (ch == '2') {

cout << "Enter ID : "; cin >> id;

temp = insta->findVertex(id);

if (temp != nullptr) {

loggedIn(temp, true);

}

else {

clear();

printBoxed("ID not found. Try again.");

delay(1500);

clear();

}

}

else if (ch == 27) {

clear();

printBoxed("Returning to Main Menu");

delay(1500);

return;

}

else {

clear();

printBoxed("Invalid Input. Please press [1] for Username or [2] for ID.");

delay(1500);

clear();

}

}

}

void Instagram::viewFeed(AdjNode\*& obj) {

printTitle("FEED");

if (obj->data.posts->isEmpty()) {

printBoxed("No Posts Available Currently");

Return();

return;

}

obj->data.posts->print();

printSeparator('=');

Return();

}

void Instagram::viewMessages(AdjNode\*& obj) {

printTitle("MESSAGES");

Stack<Messages>\* msg = obj->data.messages->copy();

if (msg->isEmpty()) {

printBoxed("No Messages Available Currently");

Return();

return;

}

while (msg->isEmpty() != true) {

cout << msg->pop();

}

printSeparator('=');

Return();

}

void Instagram::viewNotification(AdjNode\*& obj) {

printTitle("NOTIFICATIONS");

Queue<Notification>\* received = obj->data.notifications->copy();

if (received->isEmpty()) {

printBoxed("No Notifications Available Currently");

Return();

return;

}

while (received->isEmpty() != true) {

cout << received->dequeue();

}

printSeparator('=');

Return();

}

void Instagram::viewFriendRequest(AdjNode\*& obj) {

printTitle("FRIEND REQUESTS");

Queue<FriendRequest>\* received = obj->data.friendRequests->copy();

if (received->isEmpty()) {

printBoxed("No Friend Request Available Currently");

Return();

return;

}

while (received->isEmpty() != true) {

cout << received->dequeue();

}

printSeparator('=');

Return();

}

void Instagram::sendMessages(AdjNode\*& obj) {

printTitle("SEND A MESSAGE");

string sender;

string receiver;

string content;

string timestamp;

cout << "Enter Receiver User Name : "; cin >> receiver;

cout << "Enter Message to Send : "; cin >> content;

if (insta->findVertex(receiver) == nullptr) {

printBoxed("A User with Username " + receiver + " is Not Available");

Return();

return;

}

timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

sender = obj->data.username;

Messages msg(sender, receiver, content, timestamp);

obj->data.messages->push(msg);

Notification notification("Message\_Received", sender, receiver, timestamp, obj->data.name + "\_Sent\_a\_Message");

Notification notification2("Message\_Sent", "Instagram", sender, timestamp, "Message\_Sent\_Successfully\_to\_" + insta->findVertex(receiver)->data.name);

obj->data.notifications->enqueue(notification2);

insta->findVertex(receiver)->data.notifications->enqueue(notification);

printBoxed("Message Sent Successfully");

delay(1500);

}

void Instagram::sendFriendRequest(AdjNode\*& obj) {

printTitle("SEND A FRIEND REQUEST");

string sender;

string receiver;

string timestamp;

cout << "Enter Receiver User Name : "; cin >> receiver;

if (insta->findVertex(receiver) == nullptr) {

printBoxed("A User with Username " + receiver + " is Not Available");

Return();

return;

}

sender = obj->data.username;

timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

FriendRequest request(sender, receiver, timestamp);

insta->findVertex(receiver)->data.friendRequests->enqueue(request);

Notification notification("Friend\_Request\_Received", sender, receiver, timestamp, obj->data.name + "\_Sent\_a\_Friend\_Request");

Notification notification2("Friend\_Request\_Sent", "Instagram", obj->data.username, timestamp, "Friend\_Request\_Sent\_Successfully\_to\_" + insta->findVertex(receiver)->data.name);

obj->data.notifications->enqueue(notification2);

insta->findVertex(receiver)->data.notifications->enqueue(notification);

printBoxed("Request Sent Successfully");

delay(1500);

}

void Instagram::approveAllRequests(AdjNode\*& obj) {

while (!obj->data.friendRequests->isEmpty()) {

FriendRequest request = obj->data.friendRequests->dequeue();

string name = request.sender;

string timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

edgeData temp(obj->data.username, name, "Friends", "Active");

obj->edges->insert\_at\_end(temp);

temp = edgeData(name, obj->data.username, "Friends", "Active");

insta->findVertex(name)->edges->insert\_at\_end(temp);

Notification notification("Friend\_Request\_Accepted", "Instagram", name, timestamp, "Your\_Friend\_Request\_Was\_Accepted\_By\_" + obj->data.name);

Notification notification2("Friend\_Request\_Accepted", "Instagram", obj->data.username, timestamp, "You\_Have\_Accepted\_The\_Friend\_Request\_From\_" + insta->findVertex(name)->data.name);

obj->data.notifications->enqueue(notification2);

insta->findVertex(name)->data.notifications->enqueue(notification);

removeRequest(obj, name);

printBoxed("Request Accepted Successfully");

delay(1500);

}

}

void Instagram::removeRequest(AdjNode\*& obj, string from) {

Queue<FriendRequest>\* requests = obj->data.friendRequests->copy();

obj->data.friendRequests->makeNull();

while (!requests->isEmpty()) {

FriendRequest request = requests->dequeue();

if (request.sender != from) {

obj->data.friendRequests->enqueue(request);

}

}

}

void Instagram::acceptFriendRequest(AdjNode\*& obj) {

printTitle("ACCEPT FRIEND REQUEST");

if (obj->data.friendRequests->isEmpty()) {

printBoxed("No Friend Requests Available");

return;

}

cout << "Friend Requests : " << endl;

Queue<FriendRequest>\* received = obj->data.friendRequests->copy();

while (!received->isEmpty()) {

cout << received->dequeue();

}

printSeparator('=');

string name;

cout << "Enter The Username of The Request to Process : "; cin >> name;

if (isUsernameUnique(name) == false) {

char choice;

cout << "Do You Want To Accept The Request (Yes : Y/No : N) : "; cin >> choice;

string timestamp = getCurrentTimeAndDate();

timestamp[10] = '\_';

if (choice == 'y' || choice == 'Y') {

edgeData temp(obj->data.username, name, "Friends", "Active");

obj->edges->insert\_at\_end(temp);

temp = edgeData(name, obj->data.username, "Friends", "Active");

insta->findVertex(name)->edges->insert\_at\_end(temp);

Notification notification("Friend\_Request\_Accepted", "Instagram", name, timestamp, "Your\_Friend\_Request\_Was\_Accepted\_By\_" + obj->data.name);

Notification notification2("Friend\_Request\_Accepted", "Instagram", obj->data.username, timestamp, "You\_Have\_Accepted\_The\_Friend\_Request\_From\_" + insta->findVertex(name)->data.name);

obj->data.notifications->enqueue(notification2);

insta->findVertex(name)->data.notifications->enqueue(notification);

removeRequest(obj, name);

printBoxed("Request Accepted Successfully");

}

else if (choice == 'n' || choice == 'N') {

Notification notification("Friend\_Request\_Rejected", "Instagram", name, timestamp, "Your\_Friend\_Request\_Was\_Rejected\_By\_" + obj->data.username);

Notification notification2("Friend\_Request\_Rejected", "Instagram", obj->data.username, timestamp, "You\_Have\_Rejected\_The\_Friend\_Request\_From\_" + insta->findVertex(name)->data.name);

insta->findVertex(name)->data.notifications->enqueue(notification);

obj->data.notifications->enqueue(notification2);

removeRequest(obj, name);

printBoxed("Request Rejected Successfully");

}

}

delay(1500);

}

void Instagram::ExportMessageToFile(AdjNode\*& obj, ofstream& messages) {

Messages msg;

while (!obj->data.messages->isEmpty()) {

msg = obj->data.messages->pop();

messages << msg.sender << " " + msg.receiver << " " + msg.content << " " << msg.timestamp << endl;

}

}

void Instagram::ExportNotificationsToFile(AdjNode\*& obj, ofstream& notifications) {

Notification notification;

while (!obj->data.notifications->isEmpty()) {

notification = obj->data.notifications->dequeue();

notifications << notification.type << " " + notification.fromUser << " " + notification.toUser << " " + notification.timestamp << " " + notification.content << endl;

}

}

void Instagram::ExportPostsToFile(AdjNode\*& obj, ofstream& posts) {

Post post;

while (!obj->data.posts->isEmpty()) {

post = obj->data.posts->pop();

posts << post.user << " " + post.content << " " + post.timestamp << " " + post.postID << " " + post.mediaURL << " " + post.likeCount << " " + post.postType << endl;

}

}

void Instagram::ExportFriendRequestsToFile(AdjNode\*& obj, ofstream& friendRequests) {

Queue<FriendRequest>\* requests = obj->data.friendRequests->copy();

while (!requests->isEmpty()) {

FriendRequest request = requests->dequeue();

friendRequests << request.sender << " " << request.receiver << " " << request.timestamp << endl;

}

}

void Instagram::ExportEdgesToFile(AdjNode\*& obj, ofstream& edges) {

Edge\* edge = obj->edges->getHead();

while (edge != nullptr) {

edges << edge->data.from << " " << edge->data.to << " " << edge->data.relation << " " << edge->data.status << endl;

edge = edge->next;

}

}

void Instagram::ExportToFile() {

ofstream user;

ofstream posts;

ofstream messages;

ofstream notifications;

ofstream friendRequests;

ofstream edges;

user.open("users.txt");

messages.open("messages.txt");

notifications.open("notifications.txt");

posts.open("posts1.txt");

friendRequests.open("friendrequests.txt");

edges.open("edges.txt");

AdjNode\* temp = insta->getFirstNode();

while (temp != nullptr) {

user << temp->data.username << " " + temp->data.password << " " + temp->data.name << " " + temp->data.lastLogin << endl;

ExportMessageToFile(temp, messages);

ExportNotificationsToFile(temp, notifications);

ExportFriendRequestsToFile(temp, friendRequests);

ExportEdgesToFile(temp, edges);

temp = temp->next;

}

temp = insta->getFirstNode();

ExportPostsToFile(temp, posts);

user.close();

posts.close();

notifications.close();

messages.close();

friendRequests.close();

edges.close();

}

void Instagram::importMessagesFromFile(AdjNode\*& temp, ifstream& messages) {

string sender;

string receiver;

string content;

string timestamp;

while (messages >> sender >> receiver >> content >> timestamp) {

Messages msg(sender, receiver, content, timestamp);

if (msg.receiver == temp->data.username) {

temp->data.messages->push(msg);

}

}

}

void Instagram::importNotificationsFromFile(AdjNode\*& temp, ifstream& notifications) {

string type;

string fromUser;

string toUser;

string content;

string timestamp;

while (notifications >> type >> fromUser >> toUser >> timestamp >> content) {

Notification notification(type, fromUser, toUser, timestamp, content);

if (notification.toUser == temp->data.username) {

temp->data.notifications->enqueue(notification);

}

}

}

void Instagram::importPostsFromFile(AdjNode\*& temp, ifstream& posts) {

string User;

string content;

string timestamp;

string postID;

string mediaURL;

string postType;

int likeCount;

while (posts >> User >> content >> timestamp >> postID >> mediaURL >> likeCount >> postType) {

Post post(User, content, timestamp, postID, mediaURL, likeCount, postType);

temp->data.posts->push(post);

}

}

void Instagram::importFriendRequestsFromFile(AdjNode\*& temp, ifstream& friendRequests) {

string sender, receiver, timestamp;

while (friendRequests >> sender >> receiver >> timestamp) {

if (receiver == temp->data.username) {

FriendRequest request(sender, receiver, timestamp);

temp->data.friendRequests->enqueue(request);

}

}

}

void Instagram::importEdgesFromFile(AdjNode\*& temp, ifstream& edges) {

string from, to, relation, status;

while (edges >> from >> to >> relation >> status) {

if (from == temp->data.username) {

edgeData edge(from, to, relation, status);

temp->edges->insert\_at\_end(edge);

}

}

}

void Instagram::importFromFile() {

ifstream user;

ifstream posts;

ifstream messages;

ifstream notifications;

ifstream friendRequests;

ifstream edges;

string username;

string password;

string name;

string lastLogin;

user.open("users.txt");

while (user >> username >> password >> name >> lastLogin) {

posts.open("posts.txt");

notifications.open("notifications.txt");

messages.open("messages.txt");

friendRequests.open("friendrequests.txt");

edges.open("edges.txt");

userData obj(username, password, name, lastLogin);

insta->insertVertex(obj);

AdjNode\* temp = insta->findVertex(username);

importMessagesFromFile(temp, messages);

importNotificationsFromFile(temp, notifications);

importPostsFromFile(temp, posts);

importFriendRequestsFromFile(temp, friendRequests);

importEdgesFromFile(temp, edges);

posts.close();

notifications.close();

messages.close();

friendRequests.close();

edges.close();

}

user.close();

}

void Instagram::print() {

printTitle("GRAPH/AVL-BST/HASHMAP REPRESENTATION");

insta->print();

printSeparator('=');

Return();

}

void Instagram::mainMenu() {

char choice;

while (true) {

importFromFile();

clear();

printTitle("Welcome to Instagram Console App");

printBoxed("========== MAIN MENU ==========");

printCentered("[1] Sign Up");

printCentered("[2] Sign In");

printCentered("[3] Display Users");

printCentered("[4] Admin Mode");

printCentered("[Esc] Exit");

printSeparator('=');

choice = \_getch();

switch (choice) {

case '1':

clear();

SignUp();

break;

case '2':

clear();

SignIn();

break;

case '3':

clear();

print();

break;

case '4':

clear();

AdminMode();

break;

case 27:

clear();

printBoxed("Exiting The Application. Goodbye!");

return;

default:

clear();

printBoxed("Invalid Choice. Please Try Again.");

delay(1000);

}

ExportToFile();

destroyAndCreate();

}

}

**HashMap.h**#include <iostream>

#include <vector>

#include <string>

#include"AdjNode.h"

using namespace std;

const int TABLE\_SIZE = 10;

struct HashNode {

int id;

string username;

string password;

HashNode\* next;

AdjNode\* user;

HashNode() :id(0), username(""), password(""), next(nullptr), user(nullptr) {};

HashNode(int ID, string u,string p,AdjNode \* t) :id(ID), username(u), password(p), next(nullptr), user(t) {};

};

class HashTable {

private:

vector<HashNode\*> table;

int hashFunction(const string& key) {

int hash = 0;

for (char ch : key) {

hash += static\_cast<int>(ch);

}

return hash % TABLE\_SIZE;

}

public:

HashTable() : table(TABLE\_SIZE, nullptr) {}

void insert(int id, const string& username, const string& password, AdjNode\* adjNode = nullptr) {

int index = hashFunction(username);

HashNode\* newNode = new HashNode(id, username, password, adjNode);

if (table[index] == nullptr) {

table[index] = newNode;

}

else {

HashNode\* current = table[index];

while (current->next != nullptr) {

current = current->next;

}

current->next = newNode;

}

}

string getPassword(const string& username) {

int index = hashFunction(username);

HashNode\* current = table[index];

while (current != nullptr) {

if (current->username == username) {

return current->password;

}

current = current->next;

}

return "User not found!";

}

AdjNode\* getUser(int ID) {

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

HashNode\* current = table[i];

while (current != nullptr) {

if (current->id == ID) {

return current->user;

}

current = current->next;

}

}

return nullptr;

}

void display() {

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

cout << "Index " << i << ": ";

HashNode\* current = table[i];

while (current != nullptr) {

cout << "[" << current->id << "] -> (" << current->username << " : " << current->password << ") ";

current = current->next;

}

cout << endl;

}

}

~HashTable() {

for (auto& node : table) {

while (node != nullptr) {

HashNode\* temp = node;

node = node->next;

delete temp;

}

}

}

};

**Node.h**

#ifndef NODE\_H

#define NODE\_H

template<class T>

struct Node {

T data;

Node\* next;

};

#endif // NODE\_H

**List.h**

#ifndef LIST\_H

#define LIST\_H

#include "Structures.h"

class BaseSingly {

protected:

int count;

Edge\* head;

Edge\* current;

public:

BaseSingly() = default;

~BaseSingly() = default;

virtual void insert\_at\_front(edgeData& val) = 0;

virtual void insert\_at\_end(edgeData& val) = 0;

virtual void delete\_at\_front() = 0;

virtual void delete\_at\_end() = 0;

virtual void print() = 0;

virtual int search(edgeData& val) = 0;

};

class List : public BaseSingly {

public:

List() = default;

~List() = default;

void insert\_at\_front(edgeData& val);

void insert\_at\_end(edgeData& val);

void delete\_at\_front();

void delete\_at\_end();

void print();

int search(edgeData& val);

Edge\* getHead();

};

#endif // LIST\_H

**List.cpp**#include"List.h"

//BaseSingly::BaseSingly() {

// count = 0;

// head = nullptr;

// current = nullptr;

//}

//

//List::List() {}

//List::~List() {

// while (count > 0) {

// delete\_at\_end();

// }

//}

void List::insert\_at\_front(edgeData& val) {

Edge\* temp = new Edge;

temp->data = val;

temp->next = head;

head = temp;

count++;

}

void List::insert\_at\_end(edgeData& val) {

if (head == nullptr) {

insert\_at\_front(val);

return;

}

current = head;

while (current->next != nullptr) {

current = current->next;

}

Edge\* temp = new Edge;

temp->data = val;

current->next = temp;

count++;

}

void List::delete\_at\_front() {

if (head == nullptr) return;

current = head;

head = head->next;

delete current;

count--;

}

void List::delete\_at\_end() {

if (head == nullptr) return;

if (head->next == nullptr) {

delete\_at\_front();

return;

}

current = head;

while (current->next->next != nullptr) {

current = current->next;

}

delete current->next;

current->next = nullptr;

count--;

}

int List::search(edgeData& val) {

current = head;

int index = 1;

while (current != nullptr) {

/\* if (current->edgeData == val) {

cout << "Found at Index: " << index << endl;

return index;

}\*/

current = current->next;

index++;

}

cout << "Not Found" << endl;

return -1;

}

void List::print() {

current = head;

if (current == nullptr) {

cout << "[No Friends]";

}

while (current != nullptr) {

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

current = current->next;

}

cout << endl;

}

Edge\* List::getHead() {

return head;

}

**Stack.h**

#ifndef STACK\_H

#define STACK\_H

#include "Node.h"

#include <iostream>

using namespace std;

template<class T>

class baseStack {

protected:

Node<T>\* top;

public:

virtual bool isEmpty() = 0;

virtual void push(T val) = 0;

virtual T pop() = 0;

virtual T Top() = 0;

virtual void print() = 0;

// virtual Stack<T>\* copy() = 0;

};

template<class T>

class Stack : public baseStack<T> {

public:

Stack();

~Stack();

bool isEmpty();

void push(T val);

T pop();

T Top();

void print();

Stack<T>\* copy();

};

// Constructor definition for Stack

template<class T>

Stack<T>::Stack() {

this->top = nullptr;

}

// Destructor definition for Stack

template<class T>

Stack<T>::~Stack() {

while (!isEmpty()) {

pop();

}

}

// isEmpty definition for Stack

template<class T>

bool Stack<T>::isEmpty() {

return this->top == nullptr;

}

// push definition for Stack

template<class T>

void Stack<T>::push(T val) {

Node<T>\* temp = new Node<T>;

temp->data = val;

//cout << "Pushed Data : " << temp->data << endl;

temp->next = this->top;

this->top = temp;

}

// pop definition for Stack

template<class T>

T Stack<T>::pop() {

if (isEmpty()) {

throw std::out\_of\_range("Stack is empty.");

}

Node<T>\* temp = this->top;

T obj = this->top->data;

this->top = this->top->next;

delete temp;

return obj;

}

// Top definition for Stack

template<class T>

T Stack<T>::Top() {

if (isEmpty()) {

throw std::out\_of\_range("Stack is empty.");

}

return this->top->data;

}

// print definition for Stack

template<class T>

void Stack<T>::print() {

Node<T>\* tempNode = this->top;

while (tempNode != nullptr) {

cout << tempNode->data;

tempNode = tempNode->next;

}

cout << endl;

}

template<class T>

Stack<T>\* Stack<T>::copy() {

Stack<T>\* temp = new Stack<T>;

Stack<T> reverse;

while (!isEmpty()) {

T obj = pop();

reverse.push(obj);

}

while (!reverse.isEmpty()) {

T obj = reverse.pop();

temp->push(obj);

}

return temp;

}

#endif // STACK\_H

**Queue.h**#ifndef QUEUE\_H

#define QUEUE\_H

#include "Node.h"

#include <iostream>

using namespace std;

template<class T>

class baseQueue {

public:

virtual bool isEmpty() = 0;

virtual void enqueue(T value) = 0;

virtual T dequeue() = 0;

virtual void print() = 0;

virtual T peek() = 0;

virtual void makeNull() = 0;

//virtual Queue<T>\* copy() = 0;

};

template<class T>

class Queue : public baseQueue<T> {

private:

Node<T>\* front;

Node<T>\* rear;

int numOfItems;

public:

Queue();

~Queue();

bool isEmpty();

void enqueue(T value);

T dequeue();

T peek();

void makeNull();

void print();

Queue<T>\* copy();

int getCount();

};

template<class T>

int Queue<T>::getCount() {

return numOfItems;

}

template<class T>

Queue<T>::Queue() {

this->front = nullptr;

this->rear = nullptr;

numOfItems = 0;

}

template<class T>

Queue<T>::~Queue() {

makeNull();

}

template<class T>

bool Queue<T>::isEmpty() {

return numOfItems == 0;

}

template<class T>

void Queue<T>::enqueue(T value) {

Node<T>\* newNode = new Node<T>;

newNode->data = value;

// cout << "Enqueued : " << newNode->data << endl;

newNode->next = nullptr;

if (isEmpty()) {

this->front = newNode;

this->rear = newNode;

}

else {

this->rear->next = newNode;

this->rear = newNode;

}

numOfItems++;

}

template<class T>

T Queue<T>::dequeue() {

if (isEmpty()) {

cout << "Queue is Empty." << endl;

return T();

}

Node<T>\* temp = front;

T value = this->front->data;

this->front = this->front->next;

delete temp;

numOfItems--;

if (numOfItems == 0) {

this->rear = nullptr;

}

return value;

}

template<class T>

T Queue<T>::peek() {

if (isEmpty()) {

cout << "Queue is Empty. " << endl;

return T();

}

return this->front->data;

}

template<class T>

void Queue<T>::makeNull() {

while (!isEmpty()) {

dequeue();

}

}

template<class T>

void Queue<T>::print() {

if (isEmpty() == true) {

cout << "Empty" << endl;

return;

}

while (isEmpty() != true) {

cout << dequeue() << endl;

}

}

template<class T>

Queue<T>\* Queue<T>::copy() {

Queue<T>\* newQueue = new Queue<T>;

Queue<T> tempQueue;

while (!isEmpty()) {

T value = dequeue();

tempQueue.enqueue(value);

newQueue->enqueue(value);

}

while (!tempQueue.isEmpty()) {

enqueue(tempQueue.dequeue());

}

return newQueue;

}

#endif // QUEUE\_H

**Tree.h**#ifndef TREE\_H

#define TREE\_H

#include<iostream>

#include"AdjNode.h"

#include <algorithm>

using namespace std;

template<class T>

struct NodeTree {

T data;

AdjNode\* user;

NodeTree\* left;

NodeTree\* right;

NodeTree(T val);

};

template<class T>

class Base {

protected:

NodeTree<T>\* root;

public:

Base();

virtual void insert(AdjNode\* obj) = 0;

virtual void printInOrder() = 0;

virtual void deleteNode(T val) = 0;

virtual AdjNode\* findNode(T val) = 0;

};

template<class T>

class Bst :public Base<T> {

public:

Bst();

void insert(AdjNode\* obj);

void printInOrder();

void deleteNode(T val);

AdjNode\* findNode(T val);

private:

//T max(T a, T b);

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

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

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

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

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

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

void insertDataHelper(AdjNode\* obj,T val, NodeTree<T>\*& temp);

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

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

AdjNode\* findNodeHelper(T val,NodeTree<T>\* temp);

};

template<class T>

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

data = val;

left = nullptr;

right = nullptr;

user = nullptr;

}

template<class T>

Base<T>::Base() {

root = nullptr;

}

template<class T>

Bst<T>::Bst(){}

template<class T>

void Bst<T>::insert(AdjNode\* obj) {

insertDataHelper(obj, obj->data.username, 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>

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

// if (a > b) {

// return a;

// }

// return b;

//}

template<class T>

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

if (temp == nullptr) {

return -1;

}

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

}

template<class T>

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

int leftHeight = getHeight(temp->left);

int rightHeight = getHeight(temp->right);

return leftHeight - rightHeight;

}

template<class T>

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

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

if (ptr == nullptr) {

return;

}

temp->left = ptr->right;

ptr->right = temp;

temp = ptr;

}

template<class T>

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

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

if (ptr == nullptr) {

return;

}

temp->right = ptr->left;

ptr->left = temp;

temp = ptr;

}

template<class T>

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

if (temp->left == nullptr) {

return;

}

rotateLeft(temp->left);

rotateRight(temp);

}

template<class T>

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

if (temp->right == nullptr) {

return;

}

rotateRight(temp->right);

rotateLeft(temp);

}

template<class T>

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

if (temp == nullptr) {

temp = new NodeTree<T>(val);

temp->user = obj;

return;

}

if (val < temp->data) {

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

}

else if (val > temp->data) {

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

}

else {

return;

}

int bf = calculateBF(temp);

if (bf > 1) {

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

rotateRight(temp);

}

else {

rotateLeftRight(temp);

}

}

else if (bf < -1) {

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

rotateLeft(temp);

}

else {

rotateRightLeft(temp);

}

}

}

template<class T>

void Bst<T>::deleteNodeHelper(T val, NodeTree<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) {

NodeTree<T>\* toDelete = temp;

temp = temp->right;

delete toDelete;

}

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

NodeTree<T>\* toDelete = temp;

temp = temp->left;

delete toDelete;

}

else {

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

while (successor->left != nullptr) {

successor = successor->left;

}

temp->data = successor->data;

temp->user = successor->user;

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

}

}

if (temp == nullptr) return;

int bf = calculateBF(temp);

if (bf > 1) {

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

rotateRight(temp);

}

else {

rotateLeftRight(temp);

}

}

else if (bf < -1) {

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

rotateLeft(temp);

}

else {

rotateRightLeft(temp);

}

}

}

template<class T>

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

if (temp != nullptr) {

inOrderHelper(temp->left);

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

inOrderHelper(temp->right);

}

}

template<class T>

AdjNode\* Bst<T>::findNodeHelper(T val, NodeTree<T>\* temp) {

if (temp == nullptr) {

return nullptr; // Base case: value not found

}

if (temp->user->data.username == val) {

return temp->user; // Found the node

}

// Check left and right subtrees

AdjNode\* leftResult = findNodeHelper(val, temp->left);

if (leftResult != nullptr) {

return leftResult; // Found in the left subtree

}

return findNodeHelper(val, temp->right); // Continue searching in the right subtree

}

template<class T>

AdjNode\* Bst<T>::findNode(T val) {

return findNodeHelper(val, this->root);

}

#endif // TREE\_H

**Structures.h**#ifndef STRUCTURES\_H

#define STRUCTURES\_H

#include <iostream>

#include <string>

#include "Stack.h"

#include "Queue.h"

using namespace std;

struct Messages {

string sender;

string receiver;

string content;

string timestamp;

Messages() = default;

Messages(string s, string r, string c, string t): sender(s), receiver(r), content(c), timestamp(t) {}

friend ostream& operator<<(ostream& os, const Messages& msg) {

os << " Sender : " << msg.sender << " Receiver : " << msg.receiver << " Content : " << msg.content << " Timestamp : " << msg.timestamp << endl;

return os;

}

};

struct Notification {

string type;

string fromUser;

string toUser;

string timestamp;

string content;

Notification() = default;

Notification(string t, string f, string tu, string ti, string c): type(t), fromUser(f), toUser(tu), timestamp(ti), content(c) {}

friend ostream& operator<<(ostream& os, const Notification& notification) {

os << "Type : " << notification.type << " From : " << notification.fromUser << " To : " << notification.toUser << " Timestamp : " << notification.timestamp << " Content : " << notification.content << endl;

return os;

}

};

struct Post {

string user;

string content;

string timestamp;

string postID;

string mediaURL;

int likeCount = 0;

string postType;

Post() = default;

Post(string u, string c, string t, string pID, string mURL, int lCount, string pType): user(u), content(c), timestamp(t), postID(pID), mediaURL(mURL), likeCount(lCount), postType(pType) {}

void addLike() {

likeCount++;

}

friend ostream& operator<<(ostream& os, const Post& post) {

os << "User : " << post.user << " Timestamp : " << post.timestamp << " Content : " << post.content << " Likes : " << post.likeCount << " Type : " << post.postType << endl;

return os;

}

};

struct FriendRequest {

string sender;

string receiver;

string timestamp;

FriendRequest() = default;

FriendRequest(string s, string r, string t) : sender(s), receiver(r), timestamp(t) {}

friend ostream& operator<<(ostream& os, const FriendRequest& request) {

os << "Sender : " << request.sender << " Receiver : " << request.receiver << " Timestamp : " << request.timestamp << endl;

return os;

}

};

struct userData {

string username;

string password;

string name;

string lastLogin;

Stack<Post> \*posts;

Stack<Messages> \*messages;

Queue<Notification> \*notifications;

Queue<FriendRequest> \*friendRequests;

userData() : username(""), password(""), name(""), lastLogin("") {

posts = new Stack<Post>();

messages = new Stack<Messages>();

notifications = new Queue<Notification>();

friendRequests = new Queue<FriendRequest>();

}

userData(string u, string p, string n, string l) : username(u), password(p), name(n), lastLogin(l) {

posts = new Stack<Post>();

messages = new Stack<Messages>();

notifications = new Queue<Notification>();

friendRequests = new Queue<FriendRequest>();

}

friend ostream& operator<<(ostream& os, const userData& user) {

os << user.username +"] : ";

return os;

}

};

struct edgeData {

string from;

string to;

string relation;

string status;

edgeData() = default;

edgeData(string f, string t, string r, string s): from(f), to(t), relation(r), status(s) {}

friend ostream& operator<<(ostream& os, const edgeData& edge) {

os <<edge.to;

return os;

}

};

struct Edge {

edgeData data;

Edge\* next;

Edge() : next(nullptr) {}

Edge(edgeData e) : data(e), next(nullptr) {}

};

#endif // STRUCTURES\_H

**UI.h**#include <iostream>

#include <string>

#include <windows.h>

#include<vector>

using namespace std;

class UI {

public:

static int getConsoleWidth() {

CONSOLE\_SCREEN\_BUFFER\_INFO csbi;

if (GetConsoleScreenBufferInfo(GetStdHandle(STD\_OUTPUT\_HANDLE), &csbi)) {

return csbi.srWindow.Right - csbi.srWindow.Left + 1;

}

return 80;

}

static void printCentered(const string& text, char fill = ' ') {

int consoleWidth = getConsoleWidth();

int padding = (consoleWidth - static\_cast<int>(text.size())) / 2;

if (padding > 0) {

cout << string(padding, fill);

}

cout << text << endl;

}

static void printTitle(const string& title, char border = '=') {

int consoleWidth = getConsoleWidth();

string borderLine(consoleWidth, border);

cout << borderLine << endl;

printCentered(title, ' ');

cout << borderLine << endl;

}

static void printSeparator(char separator = '-') {

int consoleWidth = getConsoleWidth();

cout << string(consoleWidth, separator) << endl;

}

static void printOptions(const string options[], int size) {

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

cout << "[" << (i + 1) << "] " << options[i] << endl;

}

}

static void printBoxed(const string& message, char border = '\*') {

int consoleWidth = getConsoleWidth();

vector<string> lines;

size\_t start = 0, end;

while ((end = message.find('\n', start)) != string::npos) {

lines.push\_back(message.substr(start, end - start));

start = end + 1;

}

lines.push\_back(message.substr(start));

size\_t maxLineWidth = 0;

for (const string& line : lines) {

maxLineWidth = max(maxLineWidth, line.size());

}

int boxWidth = static\_cast<int>(maxLineWidth) + 4;

int padding = (consoleWidth - boxWidth) / 2;

if (padding > 0) {

cout << string(padding, ' ');

}

cout << string(boxWidth, border) << endl;

for (const string& line : lines) {

if (padding > 0) {

cout << string(padding, ' ');

}

cout << border << " " << line;

cout << string(maxLineWidth - line.size(), ' ') << " " << border << endl;

}

if (padding > 0) {

cout << string(padding, ' ');

}

cout << string(boxWidth, border) << endl;

}

void clear() {

system("cls");

}

void delay(int x) {

Sleep(x);

}

void LoadScreen() {

printTitle("INSTAGRAM CONSOLE APPLICATION");

cout << endl;

printCentered(" .:-=====++++++++=====--. ");

printCentered(" =#@@@@@@@@@@@@@@@@@@@@@@@@@@#= ");

printCentered(" =@@@@\*=--:::::::::::::::::-=\*@@@@+ ");

printCentered(" \*@@@= . -@@@% ");

printCentered(" =@@@. \*@@@: .@@@+ ");

printCentered(" #@@\* -+#%%%%#+- +@@%: +@@@ ");

printCentered(" %@@= +@@@@@%%@@@@@\*. -@@@ ");

printCentered(" @@@= -@@@%=. -#@@@= -@@@ ");

printCentered(" @@@= .@@@+ =@@@: :@@@ ");

printCentered(" @@@- \*@@% \*@@# :@@@ ");

printCentered(" @@@- \*@@% \*@@# :@@@ ");

printCentered(" @@@= :@@@= -@@@- :@@@ ");

printCentered(" @@@= -@@@#- :#@@@+ -@@@ ");

printCentered(" %@@= .\*@@@@%##%@@@@#: -@@@ ");

printCentered(" #@@\* -+#%%%%#\*-. +@@@ ");

printCentered(" =@@@. .@@@+ ");

printCentered(" #@@@- :%@@% ");

printCentered(" +@@@%\*=::................::-+%@@@\* ");

printCentered(" .+%@@@@@@@@@@@@@@@@@@@@@@@@@@%+. ");

printCentered(" :-==++++++++++++++++==-: ");

cout << endl;

printBoxed("Developed By : \n Abdullah Niazi(23F0017) \n Muhammad Ahmad(23F0022)");

Sleep(2000);

}

};

**Main.cpp**

#include"Instagram.h"

int main() {

Instagram obj;

UI ui;

ui.LoadScreen();

obj.mainMenu();

return 0;

}

**Text Files:**

**Users.txt**

****

**Posts.txt**

**A screenshot of a computer

Description automatically generated**

**Notification.txt**

**A screenshot of a computer screen

Description automatically generated**

**Messages.txt**

**A black background with white text

Description automatically generated**

**Friendrequests.txt**

**A black rectangular object with a white border

Description automatically generated**

**Edges.txt**

**A black rectangular object with a black border

Description automatically generated**