|  |
| --- |
| FINAL PROJECT |
| DATA STRUCTURES |
| BS-SE 3A |

|  |
| --- |
| Meerab Awais, Fatima Nasir- 23F-0515,23L-0714  12-6-2024 |

**Description of the Project:**

This project is about creating a basic version of Instagram using simple data structures such as linked lists, stacks, queues, and graphs, without using any built-in data structures like those found in standard libraries. The goal is to manage users, their personal information like username, password, and city, their posts, friendships, and messages in a social network-like system.

**Its Goals and Objectives:**

This program is designed to achieve the following functionalities:

* Users can text each other or send follow requests.
* The other person is notified when something like this happens.
* The other user can easily respond to these notifications either by replying to the messages received or by accepting follow requests.
* Users can post something which will be displayed with proper current date and timing (timestamp) and with the name of the user who has posted it.
* Users can also search for their friends or any other person that is on Instagram

**Key Features and Functionalities:**

* Each user in the system will have a profile that includes Username, password, city, posts, messages, follow requests, and notifications received.
* A graph is used to represent users and their relationships. Each user is a node in the graph, and friendships are edges between nodes. When someone sends a friend request to another user, it is stored in a queue, and they get the option of accepting it or not.
* A hash table is used to verify the username and password when logging in.
* Each user has a stack of messages for each friend. The stack stores messages in such a way that the most recent message is always at the top, allowing users to easily see the latest messages.
* Real-Time Messaging: When a message is sent, it is enqueued in a queue and can be dequeued when the recipient opens their inbox.
* Each user has their own post stack, and their followers also have a stack for posts from the user they follow. Posts are displayed on both the user's and their followers' newsfeeds.
* Friend requests are managed through a queue, i.e., it follows the first-in-first-out method. The first friend request received will be managed first.
* Notifications are stored in a queue, through which they are shown in the order received.
* A BST is implemented to store users by their usernames in sorted order, which will later help in searching the user by their names as well.
* A menu will be displayed giving the user options to sign up, log in, log out, send follow requests, cancel or accept requests, make posts, view notifications, display followers and followings, send messages, search users, and display search history.

**Workflow or Process Description:**

* Each user will create an account with a username, password, and city, which will be stored in their profile.
* The system allows users to log in using their username and password, verified using the hash table.
* Users can send friend requests, which are stored in a queue, and they can either accept or decline these requests.
* Users can message each other, with each conversation being stored in a stack of messages. The most recent message will be displayed at the top of the stack.
* When a message is sent, it is added to a queue and can be dequeued when the recipient opens their inbox.
* Users can make posts, which are stored in a stack and displayed on the user's newsfeed as well as their followers' newsfeeds.
* Notifications are managed through a queue, which is displayed in the order they are received.
* A binary search tree (BST) is used to store users in alphabetical order by their usernames, enabling efficient searching.
* The system provides a menu with options for signing up, logging in, logging out, managing follow requests, viewing posts and notifications, sending messages, searching for users, and viewing search history.

**List of Data Structures:**

1. linked list
2. Graph
3. Stack
4. Queue
5. Hash Table
6. Binary search Tree (BST)

**Use of Each Data Structure:**

**Linked list** is used to implement user profile, posts, messages and friends list i.e. each user can have a linked list of their followers and following users. And it is used because unlike arrays we don’t have to resize it, we can use memory dynamically.

**Graph** is used to represent the relationship of two users by adding an edge between them. It allows efficient traversal to find friends, handle requests, and check connections.

**Stack** is used for posts and messages and since it allows last in first out method, the user can see the latest post first and the message received can be read and answered timely.

**Queue** is used for notifications and friend requests which allows notifications to be displayed in the order they were received and the oldest request gets to processed first since it follows first in first out method.

**Hash table** is used for quick lookup of users when logging in. By hashing the username or password, the system can quickly find the corresponding user without needing to search through the entire list of users and in this code it also helps in verifying credentials during login and searching users efficiently.

**BST** is used for sorting the list as each user is added or searched and it makes searching by a username fast and efficient since its searching time complexity is O(log n).

**List of Libraries:**

5 libraries are used in our code:

1. Iostream
2. String
3. Iomanip
4. Ctime
5. Windows.h

**Purpose and Usage of Each Library:**

**Iostream** is used for the input and output operations such as cin and cout or getline().

**String** is used for string functions like cin. getline() and for easy implementation of string name, message etc. without any ambiguity.

**Iomanip** is used for controlling the appearance of the data on screen/formatting the output. It is used for setting the width and alignment of the output. For example, setw() function is used in this code multiple times.

**Ctime** is used for timestamp, to get current time and date for basically displaying posts and checking the date and time of friend’s posts. It is mainly used in function getCurrentTime() .

**Windows.h** is used for system() function that is used to add colors in this program like different colours of console are shown in output. The windows.h library is used for the Sleep() function, system("Color"), and system("CLS") commands.

**Summary of Findings and Results:**

This project is a basic social network where users can create profiles, send friend requests, post updates, send messages, and view notifications. The system uses various data structures like graphs, queues, stacks, BST, and hash tables to manage and organize the data. The project focuses on a modular design in which each feature is separately implemented in separate classes or functions to support future feature expansions, ensuring that the application can grow as more features are added.

**Final Thoughts and Remarks:**

As the network grows, we can use graph traversal techniques like BFS or DFS to suggest friends and show mutual friends. This can enhance the user experience by recommending connections based on the existing network.

**Code:**

#include <iostream>

#include <string>

#include <ctime>//for timestamp

#include<iomanip> //formatting output on console

#include<windows.h>//for color

using namespace std;

//timestamp: this function will return current date and time as string

string getCurrentTime() {

time\_t now = time(0);//get the current time

char buffer[26]; //get enough space for date and time

ctime\_s(buffer, sizeof(buffer), &now); //to avoid overflow

return string(buffer); //returns the string without the newline

}

//node for posts through linked list

struct PostNode {

string content;

string timestamp;//when post was created

PostNode\* next;//to point to the next post in stack

PostNode(string text)

{

content = text;

timestamp = getCurrentTime();

next = NULL;

}

};

//posts stack for each user to manage their posts

class PostStack {

private:

PostNode\* top;

public:

PostStack()

{

top = NULL;

}

//push a new post in stack

void push(string content) {

PostNode\* newNode = new PostNode(content);

newNode->next = top;

top = newNode;

}

bool isEmpty() {

if (top == NULL)

{

return true;

}

return false;

}

//display all posts in stack

void display() {

if (!top) {

cout << "No posts available." << endl;

return;

}

PostNode\* current = top;

while (current) {

cout << "[" << current->timestamp << "] " << current->content << "\n";

current = current->next;

}

}

};

//node for messagestack

struct MessageNode {

string sender;

string content;

string timestamp;

MessageNode\* next;

MessageNode(string person, string text)

{

sender = person;

content = text;

timestamp = getCurrentTime();

next = NULL;

}

};

//stack to handle messages between users

class MessageStack {

private:

MessageNode\* top;

public:

MessageStack()

{

top = NULL;

}

//push a new message onto stack

void push(string sender, string content) {

MessageNode\* newNode = new MessageNode(sender, content);

newNode->next = top;

top = newNode;

}

//display all messages from the stack

void display(const string& currentUser, const string& withUser) {

if (!top) { // Check if the stack is empty

cout << "No messages available." << endl;

return;

}

MessageNode\* current = top;

bool foundMessages = false; // to check if there are any messages in stack that's exchanged between two users

const int consoleWidth = 79; // so that if the message recieved is long enough it can adjust its width so that it is shown to user neatly

while (current)

{

// Display messages where the sender is the current user or the specified user

if (current->sender == currentUser || current->sender == withUser) {

foundMessages = true; // Mark that at least one message was found

if (current->sender == currentUser) {

// message sent by current user aligned to left

cout << "[" << current->timestamp << "] You: " << current->content << endl;

}

else if (current->sender == withUser) {

// displays message recieved from the other user aligned to left

cout << "[" << current->timestamp << "] " << current->sender << ": " << current->content << endl;

}

}

current = current->next; // to see the next message

}

if (!foundMessages) {

cout << "No messages with " << withUser << endl;

}

}

};

//node for follow requests in queue

struct FollowRequestNode {

string requester;

FollowRequestNode\* next;

FollowRequestNode(string req)

{

requester = req;

next = NULL;

}

};

//queue to handle follow requests and notifications

class FollowRequestQueue {

private:

FollowRequestNode\* front;

FollowRequestNode\* rear;

public:

FollowRequestQueue()

{

front = NULL;

rear = NULL;

}

//add follow request to the queue

void enqueue(string requester) {

FollowRequestNode\* newNode = new FollowRequestNode(requester);

if (!rear) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

//remove follow request from front

void dequeue() {

if (!front) {

cout << "There are no follow requests" << endl;

return;

}

FollowRequestNode\* temp = front;

front = front->next;

delete temp;

if (!front) rear = NULL;

}

//display all follow requests

void display() {

if (!front) {

cout << "There are no follow requests"<<endl;

return;

}

FollowRequestNode\* current = front;

while (current) {

cout << "- " << current->requester << endl;

current = current->next;

}

}

};

// User node for the graph(contains all user data and their relationships)

struct User {

string Name; //username

string password;//user password

string City;

PostStack posts;

MessageStack messages;

FollowRequestQueue followrequests;

FollowRequestQueue notifications;

User\* next; //pointer to next user in linked list

User\* followers; // adjacency list of followers

User\* following; // Adjacency list of users this user is following

User(string name, string pass, string city)

{

Name = name;

password = pass;

City = city;

next = NULL;

followers = NULL;

following = NULL;

}

};

// BST Node for storing users to help in searching

struct BSTNode {

string username;//key

User\* userptr; //links directly to the relevant user in graph

BSTNode\* leftchild;

BSTNode\* rightchild;

BSTNode(string name, User\* user)

{

username = name;

userptr = user;

leftchild = NULL;

rightchild = NULL;

}

};

//class to manage users in sorted order

class UserBST {

private:

BSTNode\* root;

BSTNode\* insert(BSTNode\* node, string name, User\* user) {

if (node == NULL) {

return new BSTNode(name, user);

}

if (name < node->username) {

node->leftchild = insert(node->leftchild, name, user);

}

else if (name > node->username) {

node->rightchild = insert(node->rightchild, name, user);

}

return node;

}

//search for a user by username

User\* search(BSTNode\* node, string name) {

if (node == NULL) return NULL;

if (name == node->username) {

return node->userptr;

}

else if (name < node->username) {

return search(node->leftchild, name);

}

else {

return search(node->rightchild, name);

}

}

public:

UserBST()

{

root = NULL;

}

void insert(string name, User\* user) {

root = insert(root, name, user);

}

User\* search(string name) {

return search(root, name);

}

//to display all users in sorted manner

void displayInOrder(BSTNode\* node) {

if (node != NULL) {

displayInOrder(node->leftchild);

cout << node->username << endl;

displayInOrder(node->rightchild);

}

}

void displayUsers() {

displayInOrder(root);

}

};

//Queue for friend requests or notifications

struct QueueNode {

string content;

QueueNode\* next;

QueueNode(string text)

{

content = text;

next = NULL;

}

};

// Queue for follow requests or notifications

class RequestQueue {

private:

QueueNode\* front;

QueueNode\* rear;

public:

RequestQueue()

{

front = NULL;

rear = NULL;

}

void enqueue(string content) {

QueueNode\* newNode = new QueueNode(content);

if (!rear) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

void dequeue() {

if (!front) {

cout << "There are no notifications" << endl;

return;

}

QueueNode\* temp = front;

front = front->next;

delete temp;

if (!front) rear = NULL;

}

void display() {

if (!front) {

cout << "There are no notifications" << endl;

return;

}

QueueNode\* current = front;

while (current) {

cout << "- " << current->content << endl;

current = current->next;

}

}

};

struct searchHistoryNode

{

string username;

searchHistoryNode\* next;

searchHistoryNode(string user)

{

username = user;

next = NULL;

}

};

class SearchHistory

{

private:

searchHistoryNode\* head;

public:

SearchHistory()

{

head = NULL;

}

//add searched username to the history

void addSearch(string username) {

// Create a new node

searchHistoryNode\* newNode = new searchHistoryNode(username);

newNode->next = head;

head = newNode;

}

//display search history

void displayHistory() {

if (!head) {

cout << "No search history available." << endl;

return;

}

cout << "Search History:" << endl;

searchHistoryNode\* current = head;

while (current) {

cout << "- " << current->username << endl;

current = current->next;

}

}

};

//hash table size for sorting users and managing collisions using chaining method

const int tableSize = 101;

//Graph for user networks

class UserGraph {

private:

User\* head;

User\* currentUser; //pointer to the currently logged-in user

UserBST bst;

SearchHistory searchHistory; //to store search history

User\* table[tableSize];//hash table to store users

public:

UserGraph()

{

head = NULL;

currentUser = NULL;

// Initialize hash table

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

table[i] = NULL;

}

}

//function to search for a user in hash table

User\* searchUser(string name) {

int index = hashFunction(name);//calculate hash index for given user

User\* recipient = table[index];//get the linked list at the computed hash index

//if a user is currently logged in, log this search in their search history

if (currentUser) {

searchHistory.addSearch(name);

}

//return fount user else nullptr

return recipient;

}

//logout the current user

void logout() {

if (!currentUser) {

cout << "Error: No user is currently logged in." << endl;

return;

}

cout << "Goodbye, " << currentUser->Name << "!" << endl;

currentUser = NULL;

}

User\* getCurrentUser() {

return currentUser;

}

void displaySearchHistory() {

if (!currentUser) {

cout << "Error: Please login to view your search history." << endl;

return;

}

searchHistory.displayHistory(); // Call the display function of the SearchHistory class

}

void sendFollowRequest(string to) {

if (!currentUser) {

cout << "Error: Please login to send follow requests." << endl;

return;

}

User\* recipient = searchUser(to);

if (!recipient) {

cout << "Invalid recipient." << endl;

return;

}

recipient->followrequests.enqueue(currentUser->Name + " wants to follow you!");

recipient->notifications.enqueue(currentUser->Name + " sent you a follow request");

cout << "Follow request sent from " << currentUser->Name << " to " << to << endl;

}

void acceptFollowRequest(string requesterName) {

if (!currentUser) {

cout << "Error: Please login to accept follow requests." << endl;

return;

}

User\* requester = searchUser(requesterName);

if (!requester) {

cout << "Invalid requester." << endl;

return;

}

// Add to following list of the current user

User\* newFollowingNode = new User(currentUser->Name, "", "");

newFollowingNode->next = requester->following;

requester->following = newFollowingNode;

// Add to followers list of the requester

User\* newFollowerNode = new User(requesterName, "", "");

newFollowerNode->next = currentUser->followers;

currentUser->followers = newFollowerNode;

currentUser->notifications.enqueue(requesterName + " started following you.");

cout << requesterName << " is now following " << currentUser->Name << endl;

}

void displayFollowers(string name) {

User\* user = searchUser(name);

if (!user) {

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

return;

}

cout << "Followers of " << name << ":" << endl;

User\* current = user->followers;

while (current) {

cout << "- " << current->Name << endl;

current = current->next;

}

}

void displayFollowing(string name) {

User\* user = searchUser(name);

if (!user) {

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

return;

}

cout << "Following of " << name << ":" << endl;

User\* current = user->following;

while (current) {

cout << "- " << current->Name << endl;

current = current->next;

}

}

//to compute an index for a given username

int hashFunction(const string& key) {

int hash = 0;

for (char ch : key) {

hash = (hash \* 31 + ch) % tableSize;

}

return hash;

}

//insert a new user into the hash table

void insert(const string& username, const string& password, const string& ct) {

int index = hashFunction(username);//compute index at which we will put this user in table

User\* newNode = new User(username, password, ct);//create new user

if (table[index] == NULL) {

table[index] = newNode;

}

else {

//Collision handling by adding to the front of the linked list

newNode->next = table[index];

table[index] = newNode;

}

}

//verify user's username and password while loging in

bool verify(const string& username, const string& password) {

int index = hashFunction(username);

User\* current = table[index];

while (current != NULL) {

if (current->Name == username && current->password == password) {

currentUser = current;

cout << "Welcome " << username << endl;

return true;

}

current = current->next;

}

return false;

}

//this functions returns a user from hash table using their username

User\* getUser(string name) {

int index = hashFunction(name);//get hash index for the given user

User\* recipient = table[index];//get user placed at that index

return recipient;

}

//displays posts of yours and your followers

void displayFollowerPosts()

{

//is there a currently logged in user

if (currentUser == NULL)

{

cout << "Please login to view follower posts." << endl;

return;

}

User\* friendnode = currentUser->followers;//gives list of followers

bool postsexist = false;//to check if any follower has posts to display FF

cout << "Posts from your friends:" << endl;

//traverse the followers list to get their posts

while (friendnode)

{

cout << "\nChecking friend: " << friendnode->Name << endl;//outputs the current follower that is being checked

User\* friendUser = searchUser(friendnode->Name);//gives user object from the hash table of whom we've given name as input

if (friendUser)

{

cout << "Found friend: " << friendUser->Name << endl;

//check if there's a post from that friend

if (!friendUser->posts.isEmpty())

{

cout << "Posts by " << friendUser->Name << ":" << endl;

friendUser->posts.display();

postsexist = true;

}

else

{

cout << "No posts available from " << friendUser->Name << endl;

}

}

else

{

cout << "Error: Could not retrieve user " << friendnode->Name << endl;

}

friendnode = friendnode->next;

}

if (!postsexist)

{

cout << "None of your friends have posted yet." << endl;

}

}

};

void displayMenu() {

cout << "\n------------------------------------------------Micro-Instagram Menu:--------------------------------------------------" << endl;

cout << "1. Signup" << endl;

cout << "2. Login" << endl;

cout << "3. Exit" << endl;

}

void display2() {

cout << "1. Logout" << endl;

cout << "2. Send Follow Request" << endl;

cout << "3. Accept Follow Request" << endl;

cout << "4. Post Content" << endl;

cout << "5. View Your Posts" << endl;

cout << "6. Send Messages" << endl;

cout << "7. View Messages" << endl;

cout << "8. View Follow Requests" << endl;

cout << "9. View Notifications" << endl;

cout << "A. Display Followers" << endl;

cout << "B. Display Following" << endl;

cout << "C. Search Users" << endl;

cout << "D. View Search History" << endl;

cout << "E. Display Posts of Followers" << endl;

}

int main() {

UserGraph graph;

string name, password, city, namefriend, content, recipient, message, namefollow;

char choice;

string name2 = "";

int lengthpass, index;

bool flag = false;

User\* recipientuser;

//code for outer look of instagram

system("Color DF");

cout << "\n\n";

for (int i = 0; i < 8; ++i) {

cout << endl;

}

// Outer large box

cout << setw(80) << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << setw(80) << "\* \*" << endl;

cout << setw(80) << "\* \*" << endl;

cout << setw(80) << "\* \*" << endl;

// Mini box inside

cout << setw(80) << "\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*" << endl;

cout << setw(80) << "\* \* \* \*" << endl;

cout << setw(80) << "\* \* INSTAGRAM \* \*" << endl;

cout << setw(80) << "\* \* \* \*" << endl;

cout << setw(80) << "\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*" << endl;

cout << setw(80) << "\* \*" << endl;

cout << setw(80) << "\* \*" << endl;

cout << setw(80) << "\* \*" << endl;

cout << setw(80) << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\n\n";

Sleep(1000);

system("Pause");

system("CLS");

//code for loading display

system("Color DF");

system("cls");

int bar1 = 177, bar2 = 219;

cout << "\n\n\n\t\t\t LOADING...";

cout << "\n\n\n\t\t\t\t";

for (int i = 0; i < 40; i++)

cout << char(bar1);

cout << "\r";

cout << "\t\t\t\t";

for (int i = 0; i < 40; i++)

{

cout << char(bar2);

Sleep(60);

}

cout << endl << endl;

cout << "\t\t\t\t";

system("Pause");

system("CLS");

while (true)

{

do {

flag = false;

system("Color 9F");//changes text and background color of console through windows system() function

displayMenu();

cout << "Enter choice: ";

cin >> choice;

system("CLS");

switch (choice) {

case '1':

cout << "Enter username: ";

cin >> name;

name2 = name;

do {

cout << "Enter password: ";

cin >> password;

lengthpass = password.length();

if (lengthpass < 8)

{

cout << "Password Must be of 8 characters!" << endl;

}

} while (lengthpass < 8);

cout << "Enter city: ";

cin >> city;

graph.insert(name, password, city);

break;

case '2':

cout << "Enter username: ";

cin >> name;

name2 = name;

cout << "Enter password: ";

cin >> password;

flag = graph.verify(name, password);

if (!flag)

{

cout << "Invalid Username or Password!" << endl;

}

break;

case '3':

cout << "\n------------------------------------------------Exiting the Instagram--------------------------------------------------" << endl;

system("Pause");

return 0;

default:

cout << "Invalid Input!" << endl;

}

system("Pause");

system("CLS");

} while (!flag);

system("Color 17");

system("cls");

int bar1 = 177, bar2 = 219;

cout << "\n\n\n\t\t\t LOADING...";

cout << "\n\n\n\t\t\t\t";

for (int i = 0; i < 40; i++)

cout << char(bar1);

cout << "\r";

cout << "\t\t\t\t";

for (int i = 0; i < 40; i++)

{

cout << char(bar2);

Sleep(60);

}

cout << endl << endl;

cout << "\t\t\t\t";

system("Pause");

system("CLS");

User\* foundUser = NULL;

do

{

system("Color 17");

cout << "Instagram ID of: " << name2 << endl;

display2();

cout << "Enter Choice: ";

cin >> choice;

system("CLS");

switch (choice)

{

case '1':

graph.logout();

flag = false;

break;

case '2':

cout << "Enter friend's username: ";

cin >> namefollow;

graph.sendFollowRequest(namefollow);

break;

case '3':

cout << "Enter friend's username to accept: ";

cin >> namefollow;

graph.acceptFollowRequest(namefollow);

break;

case '4':

cout << "Enter post content: ";

cin.ignore();

getline(cin, content);

graph.getCurrentUser()->posts.push(content);

cout << "Post added." << endl;

break;

case '5':

graph.getCurrentUser()->posts.display();

break;

case '6': // Send Message

cout << "Enter recipient's username: ";

cin >> recipient;

cout << "Enter message: ";

cin.ignore();

getline(cin, message);

recipientuser = graph.searchUser(recipient);

if (recipientuser) {//if receipient exists message is added to their stack

recipientuser->messages.push(graph.getCurrentUser()->Name, message);

recipientuser->notifications.enqueue(name2 + " sent you a message!");//recipient is notified of your sended message thorugh notifications queue

graph.getCurrentUser()->messages.push(graph.getCurrentUser()->Name, message);//message is added tp currently logged in users's sent message stack

cout << "Message sent." << endl;

}

else {

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

}

break;

case '7': {

cout << "Enter the username of the person whose messages you want to view: ";

cin >> recipient;

graph.getCurrentUser()->messages.display(graph.getCurrentUser()->Name, recipient);

break;

}

case '8':

graph.getCurrentUser()->followrequests.display();

break;

case '9':

graph.getCurrentUser()->notifications.display();

break;

case 'A':

case 'a':

graph.displayFollowers(graph.getCurrentUser()->Name);

break;

case 'B':

case 'b':

graph.displayFollowing(graph.getCurrentUser()->Name);

break;

case 'C':

case 'c':

cout << "Enter username to search: ";

cin >> name;

foundUser = graph.searchUser(name);//The searchUser function looks through all the users in the userGraph to see if anyone matches the username we've entered

if (foundUser) {//if found it returns a pointer to that user's information

cout << "User found: " << foundUser->Name << endl << "City: " << foundUser->City << endl;

}

else {

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

}

break;

case 'D': //display search history

graph.displaySearchHistory();

break;

case 'E':

graph.displayFollowerPosts();

break;

default:

cout << "Invalid choice. Try again." << endl;

}

system("Pause");

system("CLS");

} while (choice != '1');

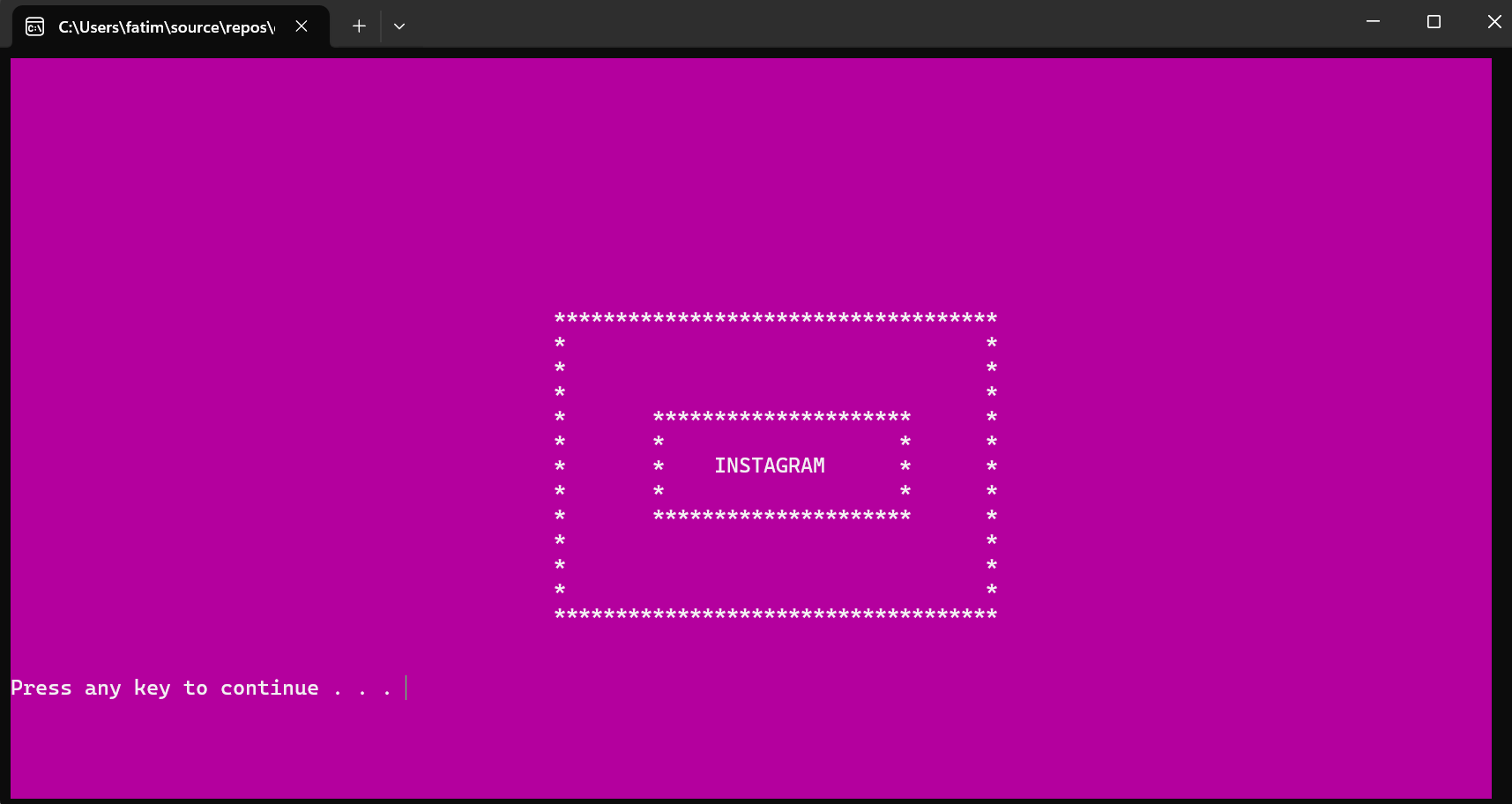
}

system("Pause");

return 0;

}

**Output:**

****

**A screenshot of a computer

Description automatically generated**

**A blue screen with white text

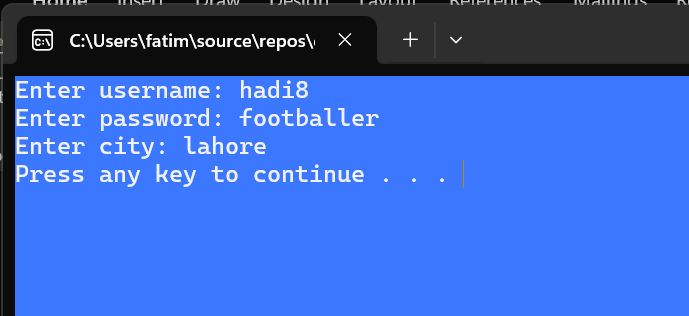
Description automatically generated**

**A screenshot of a computer

Description automatically generated**

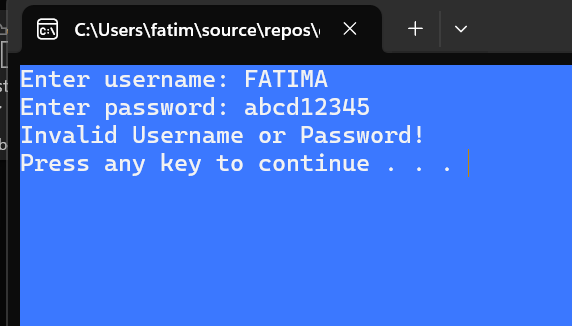
**A screenshot of a computer

Description automatically generated**

****

**A computer screen shot of a blue screen

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated**

**A screen shot of a computer

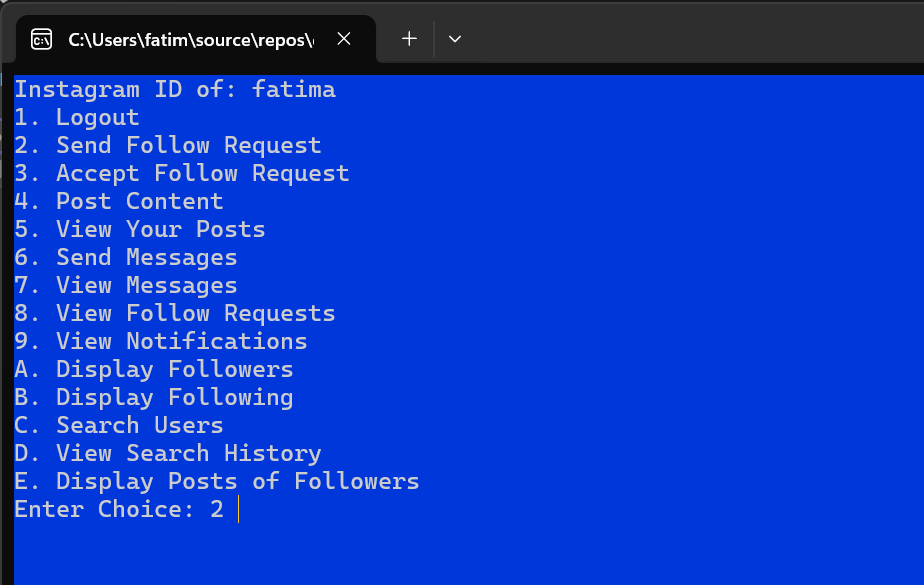
Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a computer screen

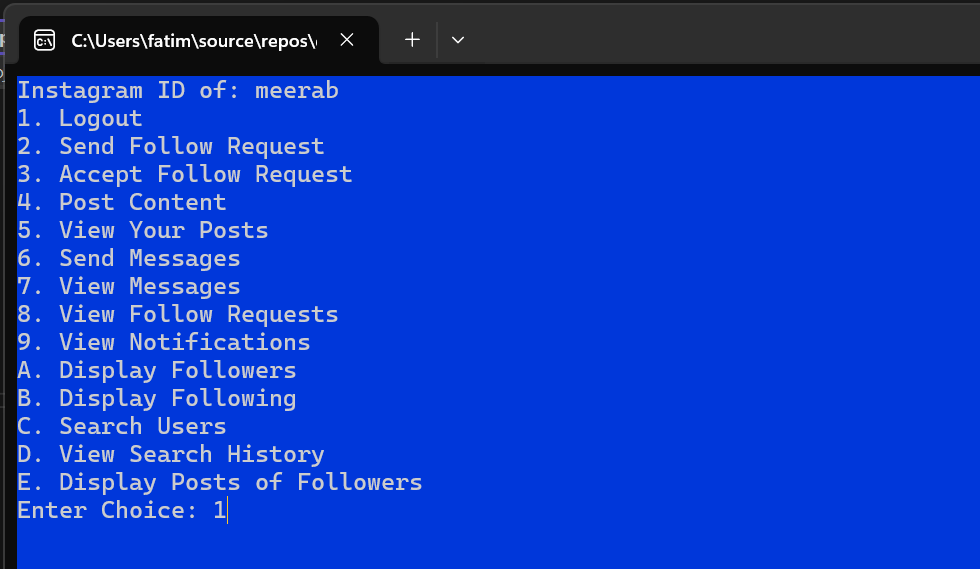
Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**



A screen shot of a computer

Description automatically generated

**A screen shot of a computer

Description automatically generated**