****

**SUBMITTED BY:**

Umair Afzal 21F-9612

**DATA STRUCTURES**

**LAB PROJECT**

**PROJECT:**

**CODE:**

#include <iostream>

#include <string>

#include <windows.h>

using namespace std;

class CommentNode {

public:

string comment;

CommentNode\* parent;

CommentNode\* left;

CommentNode\* right;

CommentNode\* replyRoot; // Pointer to the root of the reply tree

bool visited; // Boolean flag to mark if the comment has been visited

CommentNode(string commentText, CommentNode\* parentComment) {

comment = commentText;

parent = parentComment;

left = NULL;

right = NULL;

replyRoot = NULL; // Initialize replyRoot to NULL

visited = false; // Initialize visited flag to false

}

// Function to add a reply to the current comment

void addReply(string replyText) {

CommentNode\* newReply = new CommentNode(replyText, this);

if (replyRoot == NULL) {

replyRoot = newReply;

}

else {

CommentNode\* temp = replyRoot;

while (temp->right != NULL) {

temp = temp->right;

}

temp->right = newReply;

}

}

// Function to display all replies for the current comment

void displayReplies(int depth = 1) {

if (visited) {

// The current comment has been visited before (circular reference)

return;

}

visited = true;

CommentNode\* currentReply = replyRoot;

while (currentReply != NULL) {

cout << string(4 \* depth, ' ') << "Reply " << depth << ": " << currentReply->comment << endl;

currentReply->displayReplies(depth + 1);

currentReply = currentReply->right;

}

}

};

class PostNode {

public:

string post;

PostNode\* left;

PostNode\* right;

CommentNode\* commentRoot;

PostNode(string postText) {

post = postText;

left = NULL;

right = NULL;

commentRoot = NULL;

}

void addComment(string commentText) {

CommentNode\* newComment = new CommentNode(commentText, NULL);

if (commentRoot == NULL) {

commentRoot = newComment;

}

else {

CommentNode\* temp = commentRoot;

while (temp->right != NULL) {

temp = temp->right;

}

temp->right = newComment;

}

}

// Function to display all comments and their replies for the current post

void displayComments() {

CommentNode\* currentComment = commentRoot;

int commentIndex = 1;

while (currentComment != NULL) {

cout << "Comment " << commentIndex << ": " << currentComment->comment << endl;

currentComment->displayReplies(); // Display replies for the current comment

commentIndex++;

currentComment = currentComment->right;

}

}

};

class NewsFeed {

private:

struct Post {

int postIndex;

int userID;

string postText;

Post() {} // Default constructor

Post(int idx, int id, const string& text) : postIndex(idx), userID(id), postText(text) {}

};

struct ComparePosts {

bool operator()(const Post& p1, const Post& p2) {

return p1.postIndex > p2.postIndex;

}

};

Post\* minHeap;

int heapSize;

int capacity;

public:

NewsFeed(int maxPosts) : heapSize(0), capacity(maxPosts) {

minHeap = new Post[capacity];

}

~NewsFeed() {

delete[] minHeap;

}

void heapify(int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < heapSize && ComparePosts()(minHeap[left], minHeap[smallest]))

smallest = left;

if (right < heapSize && ComparePosts()(minHeap[right], minHeap[smallest]))

smallest = right;

if (smallest != idx) {

Post temp = minHeap[idx];

minHeap[idx] = minHeap[smallest];

minHeap[smallest] = temp;

heapify(smallest);

}

}

void addPostToFeed(int postIndex, int userID, const string& postText) {

if (heapSize == capacity)

return;

minHeap[heapSize++] = Post(postIndex, userID, postText);

int idx = heapSize - 1;

while (idx > 0 && ComparePosts()(minHeap[idx], minHeap[(idx - 1) / 2])) {

// Store the parent index

int parentIdx = (idx - 1) / 2;

// If the current element is smaller than its parent, swap them

if (ComparePosts()(minHeap[idx], minHeap[parentIdx])) {

// Swap the elements by direct assignment

Post temp = minHeap[idx];

minHeap[idx] = minHeap[parentIdx];

minHeap[parentIdx] = temp;

// Update the index for the next iteration

idx = parentIdx;

}

else {

break; // Stop the loop if the heap property is satisfied

}

}

}

void updateNewsFeed() {

for (int i = heapSize / 2 - 1; i >= 0; i--)

heapify(i);

}

void displayNewsFeed() {

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

cout << "User " << minHeap[i].userID << " - Post " << minHeap[i].postIndex << ": " << minHeap[i].postText << endl;

}

}

};

class Notification {

public:

enum Type { FRIEND\_REQUEST, COMMENT, LIKE };

Type type;

string details;

Notification(Type notificationType, const string& notificationDetails)

: type(notificationType), details(notificationDetails) {}

};

class UserProfile {

private:

struct NotificationNode {

Notification notification;

NotificationNode\* next;

NotificationNode(const Notification& notif) : notification(notif), next(nullptr) {}

};

NotificationNode\* notificationQueue;

public:

int userID;

string name;

string profileDetails;

int maxFriends;

int numFriends;

UserProfile\*\* friends;

PostNode\* postRoot;

NewsFeed newsFeed;

UserProfile(string userName, string details, int maxFriends) :newsFeed(100), notificationQueue(NULL) {

static int idCounter = 0;

userID = ++idCounter;

name = userName;

profileDetails = details;

this->maxFriends = maxFriends;

friends = new UserProfile \* [maxFriends];

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

friends[i] = NULL;

}

numFriends = 0; // Initialize numFriends to zero

postRoot = NULL;

}

~UserProfile() {

delete[] friends;

while (notificationQueue != NULL) {

NotificationNode\* temp = notificationQueue;

notificationQueue = notificationQueue->next;

delete temp;

}

}

void addFriend(UserProfile\* friendProfile) {

if (numFriends < maxFriends) {

friends[numFriends++] = friendProfile;

}

}

void addPost(string postText) {

PostNode\* newPost = new PostNode(postText);

if (postRoot == NULL) {

postRoot = newPost;

}

else {

newPost->right = postRoot;

postRoot = newPost;

}

}

void addNotification(Notification::Type notificationType, const string& notificationDetails) {

Notification notification(notificationType, notificationDetails);

NotificationNode\* newNotificationNode = new NotificationNode(notification);

newNotificationNode->next = notificationQueue;

notificationQueue = newNotificationNode;

}

void displayNotifications() {

cout << "Notifications for User " << userID << ":" << endl;

NotificationNode\* tempNode = notificationQueue;

while (tempNode != NULL) {

cout << " - " << tempNode->notification.details << endl;

tempNode = tempNode->next;

}

}

};

class node {

public:

UserProfile\* profile;

node\* Next;

node(UserProfile\* userProfile) {

profile = userProfile;

Next = NULL;

}

};

class list {

public:

node\* Head, \* Tail;

list() {

Head = NULL;

Tail = NULL;

}

void Insert\_Beg(UserProfile\* userProfile) {

node\* newnode = new node(userProfile);

if (Head == NULL) {

Head = newnode;

Tail = Head;

}

else {

newnode->Next = Head;

Head = newnode;

}

}

void Delete(int userID) {

if (Head == NULL) {

cout << "User profile list is empty." << endl;

return;

}

if (Head->profile->userID == userID) {

node\* temp = Head;

Head = Head->Next;

delete temp;

temp = NULL;

cout << "User profile deleted successfully." << endl;

return;

}

node\* prev = Head;

node\* curr = Head->Next;

while (curr != NULL) {

if (curr->profile->userID == userID) {

prev->Next = curr->Next;

delete curr;

curr = NULL;

cout << "User profile deleted successfully." << endl;

return;

}

prev = curr;

curr = curr->Next;

}

cout << "User with UserID: " << userID << " not found" << endl;

}

UserProfile\* search(int userID) {

node\* current = Head;

while (current != NULL) {

if (current->profile->userID == userID) {

return current->profile;

}

current = current->Next;

}

return NULL;

}

void Display() {

if (Head == NULL) {

cout << "User profile list is empty." << endl;

return;

}

node\* current = Head;

while (current != NULL) {

cout << "UserID: " << current->profile->userID << ", Name: " << current->profile->name << ", Details: " << current->profile->profileDetails << endl;

current = current->Next;

}

cout << endl;

}

};

int hashFunction(int userID) {

return userID % 100; // Assuming 100 buckets for the hash table

}

class UserProfileManager {

public:

list userProfilesHashTable[100];

int idCounter;

UserProfileManager() {

idCounter = 0;

}

void AddUserProfile(UserProfile\* userProfile) {

userProfile->userID = ++idCounter;

int index = hashFunction(userProfile->userID);

userProfilesHashTable[index].Insert\_Beg(userProfile);

}

void DeleteUserProfile(int userID) {

int index = hashFunction(userID);

userProfilesHashTable[index].Delete(userID);

}

void SearchUserProfile(int userID) {

int index = hashFunction(userID);

UserProfile\* userProfile = userProfilesHashTable[index].search(userID);

if (userProfile == NULL) {

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

}

else {

cout << "User Profile Found:" << endl;

cout << "User ID: " << userProfile->userID << ", Name: " << userProfile->name << ", Details: " << userProfile->profileDetails << endl;

}

}

void DisplayProfile(int userID) {

int index = hashFunction(userID);

cout << "ID " << index << ":" << endl;

UserProfile\* userProfile = userProfilesHashTable[index].search(userID);

if (userProfile == NULL) {

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

}

else {

cout << "User ID: " << userProfile->userID << ", Name: " << userProfile->name << ", Details: " << userProfile->profileDetails << endl;

// Display user's posts

PostNode\* currentPost = userProfile->postRoot;

int postIndex = 1;

while (currentPost != NULL) {

cout << "Post " << postIndex << ": " << currentPost->post << endl;

// Display comments for the post

CommentNode\* currentComment = currentPost->commentRoot;

int commentIndex = 1;

while (currentComment != NULL) {

cout << " Comment " << commentIndex << ": " << currentComment->comment << endl;

currentComment->displayReplies(); // Display replies for the current comment

commentIndex++;

currentComment = currentComment->right;

}

postIndex++;

currentPost = currentPost->right;

}

}

}

};

struct AdjListNode {

int vertex;

AdjListNode\* next; // creating vertex and pointer to move

};

struct AdjList {

AdjListNode\* head; // initializing pointer

};

class GraphStructure {

int numVertices;

AdjList\* adjLists; // initializing pointer for graph structure

public:

bool\* visited; // for checking if visited

GraphStructure(int vertices) {

numVertices = vertices;

adjLists = new AdjList[vertices]; // passing

visited = new bool[vertices];

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

adjLists[i].head = NULL;

visited[i] = false; // for checking visited

}

}

void addEdgeToGraph(int vertex, int edge) { // adding edges to graphs

AdjListNode\* node = new AdjListNode;

node->vertex = edge;

node->next = adjLists[vertex].head;

adjLists[vertex].head = node;

}

};

class Graph {

private:

int maxUsers;

UserProfile\*\* users;

GraphStructure graphStructure; // Using the provided graph structure

public:

Graph(int maxSize) : graphStructure(maxSize) {

maxUsers = maxSize;

users = new UserProfile \* [maxSize];

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

users[i] = NULL;

}

}

~Graph() {

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

delete users[i];

}

delete[] users;

}

void addEdge(int userID1, int userID2) {

if (userID1 < 1 || userID1 > maxUsers || userID2 < 1 || userID2 > maxUsers) {

cout << "Invalid user ID. Please enter valid user IDs." << endl;

return;

}

if (users[userID1 - 1] == NULL) {

users[userID1 - 1] = new UserProfile("User" + to\_string(userID1), "", maxUsers - 1);

}

if (users[userID2 - 1] == NULL) {

users[userID2 - 1] = new UserProfile("User" + to\_string(userID2), "", maxUsers - 1);

}

users[userID1 - 1]->addFriend(users[userID2 - 1]);

users[userID2 - 1]->addFriend(users[userID1 - 1]);

// Adding edges to the graph structure

graphStructure.addEdgeToGraph(userID1 - 1, userID2 - 1);

graphStructure.addEdgeToGraph(userID2 - 1, userID1 - 1);

}

void displayFriendNetwork(int userID) {

if (userID < 1 || userID > maxUsers || users[userID - 1] == NULL) {

cout << "User not found in the network." << endl;

return;

}

cout << "Friend Network of User " << userID << ":" << endl;

UserProfile\* user = users[userID - 1];

for (int i = 0; i < user->numFriends; i++) {

UserProfile\* friendProfile = user->friends[i];

if (friendProfile != NULL) {

cout << " Name: " << friendProfile->name << endl;

}

}

}

UserProfile\*\* getUsers() {

return users;

}

};

int main() {

system("color 8F");

UserProfileManager manager;

Graph friendNetwork(100);

NewsFeed newsFeed(100);

int choice;

while (true) {

cout << "//////////////////////////////////SOCIAL NETWORK STIMULATION////////////////////////////////////" << endl;

cout << "///////////////////////////////////////////Menu:////////////////////////////////////////////////" << endl;

cout << "| 1. Add User Profile |" << endl;

cout << "| 2. Delete User Profile |" << endl;

cout << "| 3. Search User Profile |" << endl;

cout << "| 4. Display User Profile |" << endl;

cout << "| 5. Add Friendship |" << endl;

cout << "| 6. Display Friend Network |" << endl;

cout << "| 7. Add Post |" << endl;

cout << "| 8. Add Comment |" << endl;

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

cout << "| 10. Exit |" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string name;

string details;

cout << "Enter Name: ";

cin.ignore(); // Ignore the newline character left by the previous cin

getline(cin, name);

cout << "Enter profile details: ";

getline(cin, details);

UserProfile\* profile = new UserProfile(name, details, 99); // 99 is the maximum number of friends

manager.AddUserProfile(profile);

cout << "User Profile added successfully!" << endl;

break;

}

case 2: {

int userID;

cout << "Enter User ID to delete: ";

cin >> userID;

manager.DeleteUserProfile(userID);

break;

}

case 3: {

int userID;

cout << "Enter User ID to search: ";

cin >> userID;

manager.SearchUserProfile(userID);

break;

}

case 4: {

int userID;

cout << "Enter User ID to display: ";

cin >> userID;

manager.DisplayProfile(userID);

// Update and display the news feed

newsFeed.updateNewsFeed();

cout << "News Feed:" << endl;

newsFeed.displayNewsFeed();

break;

}

case 5: {

int userID1, userID2;

cout << "Enter User ID of the first user: ";

cin >> userID1;

cout << "Enter User ID of the second user: ";

cin >> userID2;

friendNetwork.addEdge(userID1, userID2);

break;

}

case 6: {

int userID;

cout << "Enter User ID to display friend network: ";

cin >> userID;

friendNetwork.displayFriendNetwork(userID);

break;

}

case 7: {

int userID;

string postText;

cout << "Enter User ID to add a post: ";

cin >> userID;

cout << "Enter the post text: ";

cin.ignore();

getline(cin, postText);

// Add the post to the news feed

newsFeed.addPostToFeed(manager.idCounter, userID, postText);

if (userID < 1 || userID > 100) {

cout << "Invalid user ID. Please enter a valid user ID." << endl;

}

else {

UserProfile\* userProfile = manager.userProfilesHashTable[hashFunction(userID)].search(userID);

if (userProfile == nullptr) {

cout << "User profile not found. Please add the user profile first." << endl;

}

else {

userProfile->addPost(postText);

cout << "Post added successfully!" << endl;

string notificationDetails = "New post added: " + postText;

userProfile->addNotification(Notification::COMMENT, notificationDetails);

}

}

// Update and display the news feed

newsFeed.updateNewsFeed();

cout << "News Feed:" << endl;

newsFeed.displayNewsFeed();

break;

}

case 8: {

int userID, postIndex, commentIndex;

string commentText, replyText;

cout << "Enter User ID to add a comment: ";

cin >> userID;

cout << "Enter the post index (starting from 1): ";

cin >> postIndex;

cout << "Enter the comment index (starting from 1): ";

cin >> commentIndex;

cout << "Enter the comment text: ";

cin.ignore();

getline(cin, commentText);

if (userID < 1 || userID > 100) {

cout << "Invalid user ID. Please enter a valid user ID." << endl;

}

else if (manager.userProfilesHashTable[hashFunction(userID)].search(userID) == nullptr) {

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

}

else {

UserProfile\* userProfile = manager.userProfilesHashTable[hashFunction(userID)].search(userID);

if (userProfile->postRoot == nullptr) {

cout << "User has no posts. Please add a post first." << endl;

}

else {

PostNode\* currentPost = userProfile->postRoot;

int currentIndex = 1;

while (currentPost != nullptr) {

if (currentIndex == postIndex) {

currentPost->addComment(commentText);

cout << "Comment added successfully!" << endl;

// Now, ask if the user wants to add a reply to the comment

cout << "Do you want to add a reply to this comment? (Y/N): ";

char choice;

cin >> choice;

cin.ignore();

while (toupper(choice) == 'Y') {

cout << "Enter the reply text: ";

getline(cin, replyText);

CommentNode\* currentComment = currentPost->commentRoot;

int commentIdx = 1;

while (currentComment != nullptr) {

if (commentIdx == commentIndex) {

currentComment->addReply(replyText);

cout << "Reply added successfully!" << endl;

// Add a notification for the reply

string notificationDetails = "New reply added: " + replyText;

userProfile->addNotification(Notification::COMMENT, notificationDetails);

break;

}

commentIdx++;

currentComment = currentComment->right;

}

if (currentComment == nullptr) {

cout << "Comment index out of range." << endl;

break;

}

cout << "Do you want to add another reply? (Y/N): ";

cin >> choice;

cin.ignore();

}

break;

}

currentIndex++;

currentPost = currentPost->right;

}

if (currentPost == nullptr) {

cout << "Post index out of range." << endl;

}

}

}

break;

}

case 9: {

int userID;

cout << "Enter User ID to view notifications: ";

cin >> userID;

UserProfile\* userProfile = manager.userProfilesHashTable[hashFunction(userID)].search(userID);

if (userProfile == NULL) {

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

}

else {

userProfile->displayNotifications();

}

break;

}

case 10: {

cout << "Exiting the program..." << endl;

return 0;

}

default:

{

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

break;

}

}

}

}

**OUTPUT:**













