ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT077-3-2 DSTR

DATA STRUCTURES

HAND OUT DATE: 14 MARCH - 2023

HAND IN DATE: 31 MAY – 2023

Name	TP Number
MOHAMED KHAIRY MOHAMED ABDELRAOUF	TP066168 (Team Leader)

Table of Contents

Parts done by MOHMED KHAIRY

Introduction

The project consists of a university feedback management system implemented in C++. It allows users to register, login, search for universities, add them to favorites, provide feedback, and view feedback given by other users. The system also includes an admin user who has additional privileges such as managing customer details, deleting inactive customers, reading and replying to feedback, and generating reports.

The project utilizes the following data structures:

1. Dynamic Array: Dynamic arrays are used to store and manage collections of universities, feedbacks, favorite universities, and users. Dynamic arrays allow for efficient resizing and manipulation of the data as the size of the collection changes.

2. Linked List: Linked lists are used to store and manage the list of feedbacks for each university. Each university has a linked list of feedback nodes, allowing for efficient insertion and deletion of feedbacks without the need for resizing the entire array.

The project implements the following sorting algorithms:

1. Bubble Sort: Bubble sort algorithm is used to sort universities by name and rank. Bubble sort compares adjacent elements and swaps them if they are in the wrong order until the entire list is sorted. It has a time complexity of O(n^2).

2. Insertion Sort: Insertion sort algorithm is used to sort universities by name and rank. Insertion sort iterates through the list, comparing each element with the previous ones and inserting it at the correct position in the sorted part of the list. It has a time complexity of O(n^2).

The project also utilizes the following search algorithms:

1. Linear Search: Linear search algorithm is used to search for universities by name. It checks each element of the list sequentially until a match is found or the end of the list is reached. It has a time complexity of O(n).

2. Binary Search: Binary search algorithm is used to search for universities by rank. Binary search divides the sorted list into halves and compares the target value with the middle element, reducing the search range in each iteration. It has a time complexity of O(log n).

The project is divided into multiple header (.h) and cpp (.cpp) files for improved code organization, reusability, and easier maintenance. The header files contain class declarations, function prototypes, and necessary includes. The cpp files contain the implementation of the declared classes and functions.

The implementation includes several classes:

- **BaseUser:** Represents the base class for all user types and includes common user properties and methods.

- **RegisteredUser:** Inherits from BaseUser and adds functionality specific to registered users, such as adding feedback and managing favorite universities.

- **AdminUser:** Inherits from BaseUser and represents the admin user with additional privileges.

- University: Represents a university with properties such as name, rank, location, feedbacks, and methods to add and retrieve feedback.

- **UserInterface:** Manages the user interface and user interactions, including login, registration, and menu navigation.

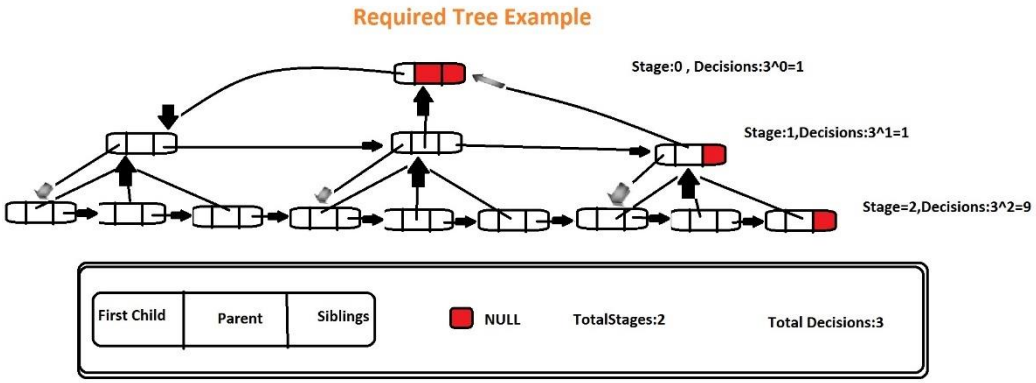
- **DataSeeder:** Loads university data from a file and provides access to the loaded universities.

- **SortAndSearch:** Contains sorting and searching algorithms used for universities.

- Main: The entry point of the program that initializes the user interface and starts the application.

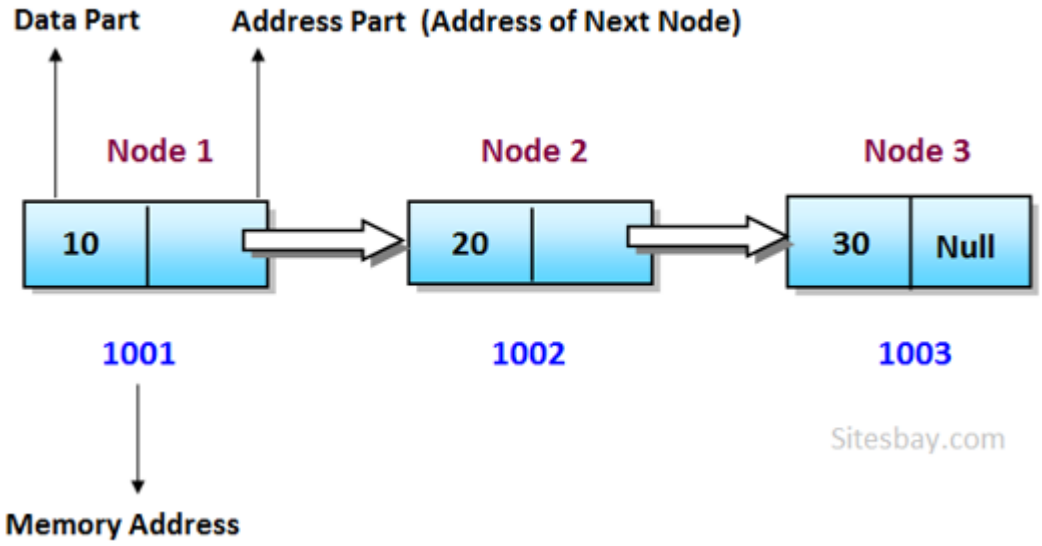
1. Dynamic Array:

A dynamic array is a data structure that can dynamically grow or shrink its size at runtime. It allows for efficient insertion and deletion of elements and provides random access to its elements. Unlike static arrays, dynamic arrays are not fixed in size and can adjust their capacity as needed. This flexibility makes dynamic arrays suitable for situations where the number of elements may change over time. Dynamic arrays are important because they provide efficient memory allocation and deallocation, allowing for optimal memory usage and reduced wastage.



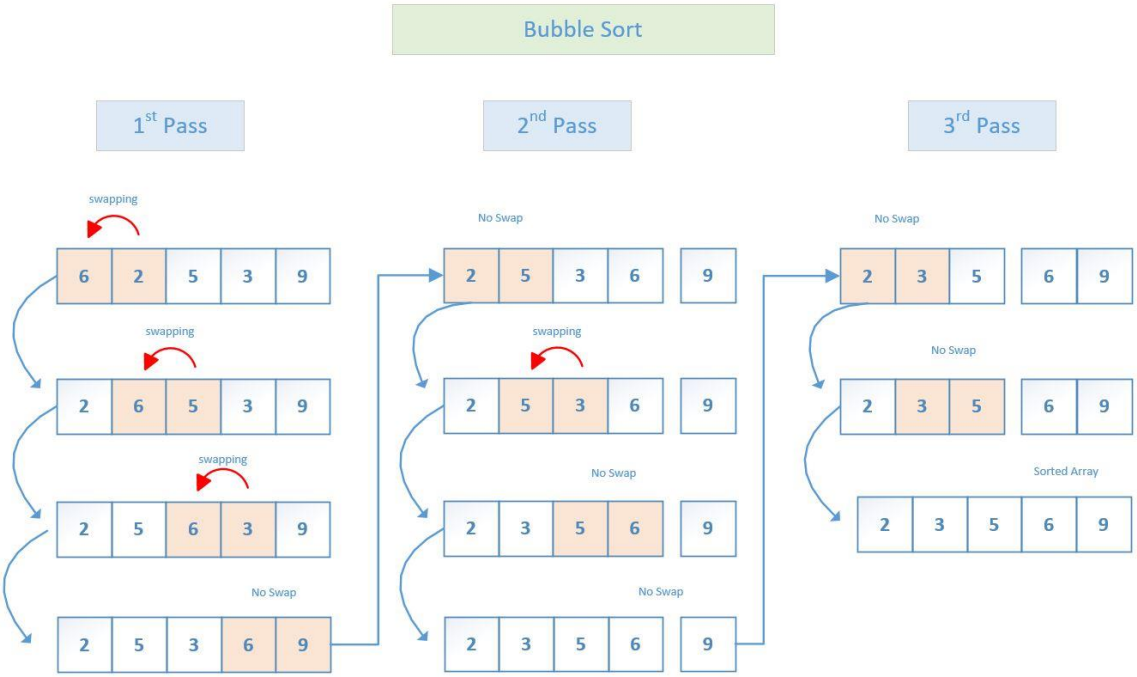
2. Linked List:

A linked list is a data structure that consists of a sequence of nodes, where each node contains a value and a reference to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation. They allow for efficient insertion and deletion of elements at any position in the list. However, random access to elements is slower in linked lists compared to arrays. Linked lists are important because they offer dynamic memory allocation, efficient insertion and deletion operations, and are especially useful when the size of the data may change frequently.



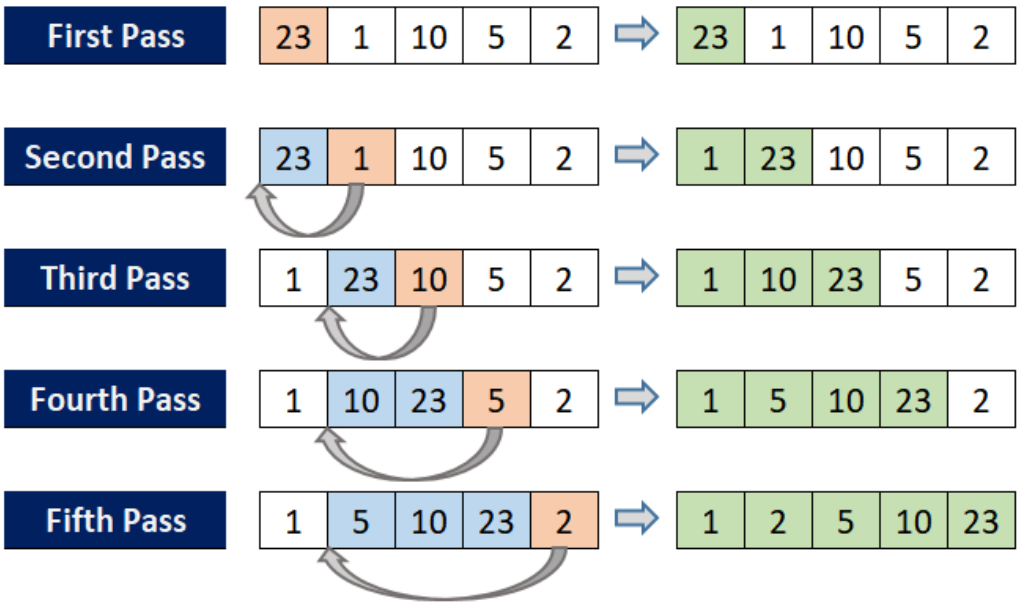
3. Bubble Sort:

Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. It iterates through the list multiple times, gradually moving the largest or smallest elements to their correct positions. Bubble sort has a time complexity of O(n^2) and is not suitable for large datasets. However, it is easy to understand and implement. Bubble sort is important for educational purposes or for small datasets where simplicity is valued over efficiency.



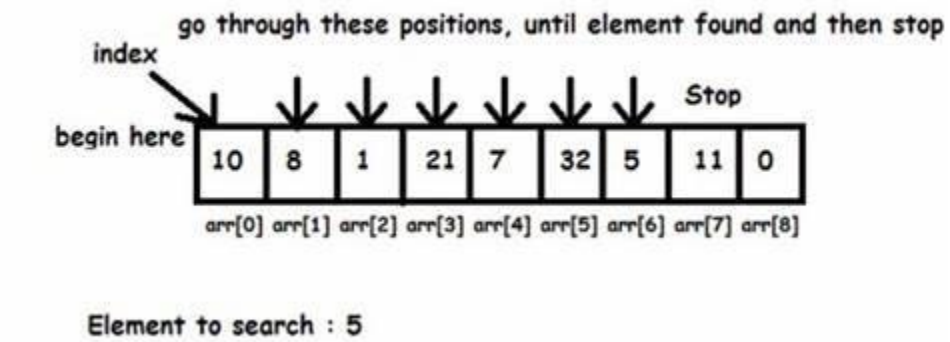
4. Insertion Sort:

Insertion sort is a sorting algorithm that builds the final sorted array one element at a time. It iterates through the input array, comparing each element with the elements before it and inserting it at the correct position in the sorted part of the array. Insertion sort has a time complexity of O(n^2) but performs well on small or partially sorted arrays. It is stable, meaning it preserves the relative order of elements with equal values. Insertion sort is important when dealing with small datasets or when the input is already partially sorted.



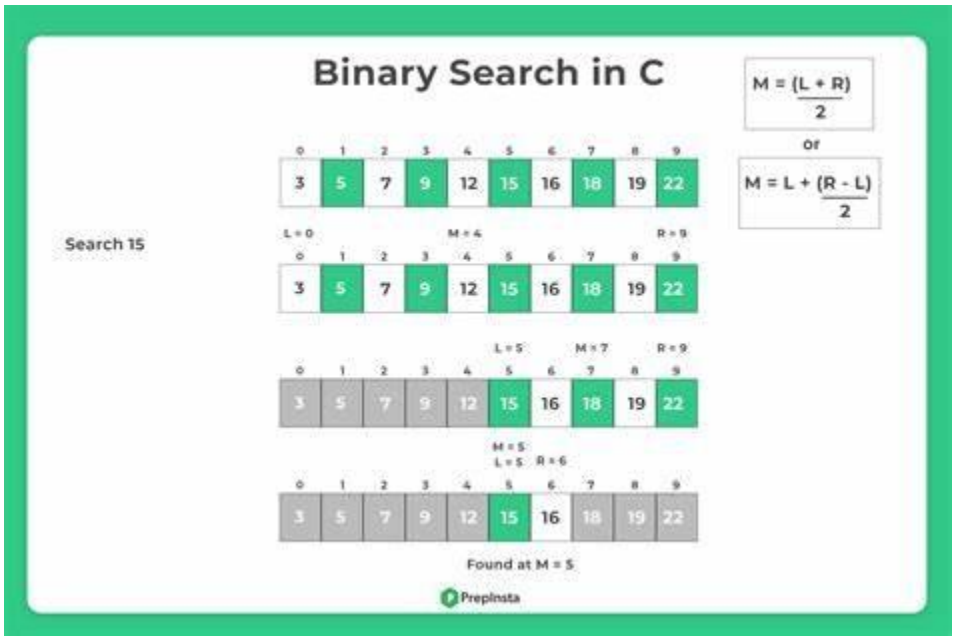
5. Linear Search:

Linear search is a simple search algorithm that sequentially checks each element in a list until a match is found or the end of the list is reached. It has a time complexity of O(n), where n is the size of the list. Linear search is straightforward to implement and works well for small lists or unsorted data. However, it is not efficient for large datasets, as it requires checking each element in the list.



6. Binary Search:

Binary search is a search algorithm that works on sorted lists or arrays. It repeatedly divides the search interval in half and compares the target value with the middle element. By eliminating half of the remaining elements in each iteration, binary search achieves a time complexity of O(log n), where n is the size of the list. Binary search is more efficient than linear search for large sorted datasets, but it requires the list to be sorted beforehand.



Main.cpp

```cpp
//main.cpp
// Include necessary header files
#include "BaseUser.h"
#include "RegisteredUser.h"
#include "AdminUser.h"
#include "University.h"
#include "UserInterface.h"
#include "DataSeeder.h"
#include "SortAndSearch.h"
#include "UserManager.h"

#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <string>
#include <chrono>
#include <ctime>
#include <string>

using namespace std::chrono;
using namespace std;


int main()
{
    // Load the data from the CSV file
    DataSeeder::loadData();

    // Check if data is loaded correctly
    if (DataSeeder::getUniversities().empty()) {
        std::cerr << "Error: No university data loaded.\n";
        return -1;  // Terminate the program if no data is loaded
    }

    UserManager userManager;  // Create a UserManager object
    UserInterface ui(userManager);  // Create a UserInterface object

    while (true)
    {
        ui.mainMenu();  // Display the main menu

        int choice;
        std::cin >> choice;
        std::cin.ignore();

        switch (choice)
        {
        case 1:
            displayAllUniversities();  // Display information about all universities
            break;
        case 2:
            SortMenu();  // Display the sorting menu
            break;
            break;
        case 3:
            SearchMenu();  // Display the search menu
            break;
        case 4:
            ui.createUserAccount();  // Create a user account
            break;
        case 5:
            ui.loginUser();  // Perform user login
            break;
        case 6:
            ui.loginAdmin();  // Perform admin login
            break;
        case 0:
            return 0;  // Terminate the program
        default:
            std::cout << "Invalid option. Please try again.\n";
        }
    }

    return 0;
}
```

Explanation:

The code represents the main function of a program that manages user accounts and universities. Here is a breakdown of its functionality:

The program starts by including necessary header files and defining a function for converting a time structure to a formatted string. Then, it enters the main function. The program loads data from a CSV file using the `loadData` function from the `DataSeeder` class. If no universities are loaded, it displays an error message and terminates the program.

Next, a `UserManager` object is created to handle user account management, and a `UserInterface` object is created, which provides a user interface for interacting with the program. The program enters a while loop that continues until explicitly terminated.

Inside the loop, the main menu is displayed using the `mainMenu` function of the `UserInterface` object. The user is prompted to enter a choice, and a switch statement is used to execute different actions based on the choice. The actions include displaying all universities, sorting, searching, creating user accounts, logging in users or admins, and terminating the program.

The loop continues, and the main menu is displayed again, allowing the user to perform multiple operations. This loop continues until the user chooses to terminate the program. Finally, the `main` function returns 0, indicating successful program execution.

Overall, this program provides a menu-based interface for managing user accounts and universities, allowing users to perform various operations such as viewing university information, sorting and searching data, creating accounts, and logging in as users or admins.

UserInterface.h

```cpp
//UserInterface.h

#ifndef USERINTERFACE_H
#define USERINTERFACE_H

#include <unordered_map>
#include <vector>
#include <string>
#include "University.h"

class UserManager; // Forward declaration of the UserManager class

class UserInterface {
public:
    void mainMenu();
    void logoutUser();
    UserInterface(UserManager& userManager); // Constructor
    ~UserInterface(); // Destructor
    void createUserAccount();
    void loginUser();
    void modifyPassword();
    void loginAdmin();
    void displayRegisteredMenu(); // Updated function declaration

private:
    UserManager& userManager; // Reference to the UserManager object
    BaseUser* loggedInUser;
};

#endif

```

Explanation

The `UserInterface` class, defined in the `UserInterface.h` header file, encapsulates the functionality of the user interface for the program. It provides various member functions that allow users to interact with the program and perform specific tasks. These tasks include creating user accounts, logging in as a user or an admin, modifying passwords, and displaying menus.

The class maintains a reference to a `UserManager` object, which is responsible for managing user accounts. This allows the `UserInterface` class to delegate tasks related to user account management to the `UserManager`. Additionally, the class has a private data member `loggedInUser`, which is a pointer to a `BaseUser` object representing the currently logged-in user.

The `UserInterface` class acts as an intermediary between the user and the program's functionality. It provides a user-friendly interface through which users can navigate menus and perform actions based on their choices. By encapsulating the user interface logic within this class, the rest of the program can focus on its core functionalities, such as loading data, performing searches, and managing universities. The `UserInterface` class plays a crucial role in ensuring a smooth and intuitive user experience while interacting with the program.

UserInterface.cpp

```cpp

//UserInterface.cpp
#include "UserInterface.h"
#include <iostream>
#include "AdminUser.h"
#include "SortAndSearch.h"
#include "BaseUser.h"
#include "RegisteredUser.h"
#include "UserManager.h"
#include "University.h"

// Display the main menu
void UserInterface::mainMenu() {
    std::cout << "\n\n";
    std::cout << "==============================================\n";
    std::cout << "=====  Welcome to Hamada QSRanking Program =====\n";
    std::cout << "==============================================\n";
    std::cout << "\n";

    std::cout << "================== Main Menu =================\n";
    std::cout << "  1. Display all universities' information\n";
    std::cout << "  2. Sort universities\n";
    std::cout << "  3. Search university details\n";
    std::cout << "  4. Register as customer\n";
    std::cout << "  5. Login as registered customer\n";
    std::cout << "  6. Login as Admin\n";
    std::cout << "  0. Exit program\n";

    std::cout << "\n";
    std::cout << "==============================================\n";
    std::cout << "Enter your choice: ";
}

// Log out the currently logged-in user and display the main menu
void UserInterface::logoutUser() {
    loggedInUser = nullptr;
    mainMenu();
}

// Constructor of UserInterface class, takes a reference to a UserManager object as a parameter
UserInterface::UserInterface(UserManager& userManager) : userManager(userManager), loggedInUser(nullptr) {}

// Destructor of UserInterface class
UserInterface::~UserInterface() {}

// Create a user account
void UserInterface::createUserAccount() {
    std::string username, password;
    std::cout << "Enter username: ";
    std::cin >> username;
    std::cout << "Enter password: ";
    std::cin >> password;

```

```
    if (userManager.findUser(username)) {
        std::cout << "Username already exists. Please choose a different username.\n";
    }
    else {
        BaseUser* newUser = new RegisteredUser(username, password);
        userManager.addUser(newUser);
        std::cout << "User account created successfully.\n";
    }
}

// Perform user login
void UserInterface::loginUser() {
    if (loggedInUser) {
        std::cout << "You are already logged in as " << loggedInUser->getUsername() << ".\n";
        return;
    }

    std::string username, password;
    std::cout << "Enter username: ";
    std::cin >> username;
    std::cout << "Enter password: ";
    std::cin >> password;

    BaseUser* user = userManager.findUser(username);
    if (user && user->authenticate(password)) {
        loggedInUser = user;
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            registeredUser->setLoggedIn(true);
            std::cout << "Login successful.\n";
            registeredUser->displayRegisteredMenu(*this, userManager);
        }
    }
    else {
        std::cout << "Invalid username or password. Login failed.\n";
    }
}

// Modify the password of the logged-in user
void UserInterface::modifyPassword() {
    if (!loggedInUser) {
        std::cout << "You must be logged in to modify your password.\n";
        return;
    }

    std::string newPassword;
    std::cout << "Enter new password: ";
    std::cin >> newPassword;
    userManager.modifyPassword(loggedInUser, newPassword);
    std::cout << "Password modified successfully.\n";
}

```

```
105
106  // Perform admin login
107 void UserInterface::loginAdmin() {
108     std::cout << "Enter your login details:\n";
109     std::string username, password;
110     std::cout << "Username: ";
111     std::cin >> username;
112     std::cout << "Password: ";
113     std::cin >> password;
114
115     if (username == "admin" && password == "123") {
116         // Admin login successful
117         std::cout << "Login successful.\n";
118
119         AdminUser admin("admin", "123"); // Create an instance of the AdminUser class
120         admin.setLoggedIn(true); // Set the admin as logged in
121
122         std::cout << "Admin Menu:\n";
123
124         // Pass the userManager object to the displayAdminMenu function
125         admin.displayAdminMenu(*this, userManager);
126     }
127     else {
128         std::cout << "Invalid username or password.\n";
129     }
130 }
131
132  // Display the registered user menu
133 void UserInterface::displayRegisteredMenu() {
134     if (loggedInUser) {
135         RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(loggedInUser);
136         if (registeredUser) {
137             registeredUser->displayRegisteredMenu(*this, userManager);  // Pass the userManager object to displayRegisteredMenu
138         }
139     }
140     else {
141         std::cout << "You must be logged in as a registered user to access the registered menu.\n";
142     }
143 }
144
```

Explanation

The `UserInterface` class in the code serves as the user interface component of a program. It encapsulates various functions that enable user interaction and control flow. Here's an explanation of its functionality:

The class includes a `mainMenu()` function that displays the main menu options to the user. This menu presents choices such as displaying university information, sorting universities, searching for details, registering as a customer, logging in as a registered customer, and logging in as an admin. The `logoutUser()` function allows the user to log out and returns them to the main menu.

The `UserInterface` class also includes functions for creating user accounts, performing user login, modifying passwords, and performing admin login. The `createUserAccount()` function prompts the user to enter a username and password, checks if the username already exists, and creates a new user account if it doesn't. The `loginUser()` function handles the process of user login, authenticating the provided username and password and granting access if successful. The `modifyPassword()` function allows a logged-in user to change their password. The `loginAdmin()` function handles the login process for the admin, checking if the provided username and password match the predefined admin credentials.

Additionally, there is a `displayRegisteredMenu()` function that is likely called from the `RegisteredUser` class. It checks if a user is logged in as a registered user and then displays a menu specific to registered users.

Overall, the `UserInterface` class provides the necessary functions and control flow to facilitate user interaction and manage user accounts within the program. It allows users to navigate menus, perform actions such as logging in, registering, and modifying passwords, and ensures the appropriate menus are displayed based on the user's role.

University.h

```
//University.h
#ifndef UNIVERSITY_H
#define UNIVERSITY_H

#include <string>
#include <vector>
#include <fstream>
#include <sstream>
#include <chrono> // For measuring execution time
#include <algorithm>
#include <iostream>
#include "BaseUser.h"

// Structure representing a University
struct University {
    int rank;
    std::string name;
    std::string locationCode;
    std::string location;
    double arScore;
    int arRank;
    double erScore;
    int erRank;
    double fsrScore;
    int fsrRank;
    double cpfScore;
    int cpfRank;
    double ifrScore;
    std::string ifrRank;
    double isrScore;
    int isrRank;
    double irnScore;
    int irnRank;
    double gerScore;
    int gerRank;
    double scoreScaled;

    std::string getName() const;  // Function declaration to get the name of the University
    static University* searchUniversityByNameBinary(const std::string& name);  // Static function declaration to search for a University by name using binary search
    static University* searchUniversityByNameLinear(const std::string& name);  // Static function declaration to search for a University by name using linear search
    bool operator==(const University& other) const;  // Overloaded equality operator to compare Universities
    void addFeedback(const std::string& feedback);  // Function declaration to add feedback to the University
    std::vector<std::string> getFeedbacks() const;  // Function declaration to get the feedbacks of the University

    void addReply(const std::string& reply);  // Function declaration to add a reply to the University

private:
    std::vector<std::string> feedbacks;  // Private member variable to store feedbacks
};

void displayAllUniversities();  // Function declaration to display all Universities

void sortUniversities();  // Function declaration to sort Universities
void SearchMenu();  // Function declaration for the search menu
```

```
void displayAllUniversities();  // Function declaration to display all Universities

void sortUniversities();  // Function declaration to sort Universities
void SearchMenu();  // Function declaration for the search menu

University* searchUniversityByRankLinear(int rank);  // Function declaration to search for a University by rank using linear search

University* searchUniversityByRankBinary(int rank);  // Function declaration to search for a University by rank using binary search

#endif
```

Explanation

The provided code defines the `University` struct and declares several related functions in the `University.h` header file. Here's an explanation of this part:

The `University` struct represents a university and includes various attributes such as rank, name, location, scores in different categories, and feedbacks. It also includes member functions to perform operations on university objects, such as retrieving the name, searching for universities by name using binary or linear search, comparing equality between universities, adding feedbacks, and retrieving feedbacks.

The header file also declares several functions outside the `University` struct. The `displayAllUniversities()` function is responsible for displaying information about all the universities. The `sortUniversities()` function is used to sort the universities based on a certain criterion. The `SearchMenu()` function likely displays a menu for searching universities. Additionally, the header file includes declarations for functions to search for universities by rank using linear or binary search.

Overall, the `University.h` header file provides the necessary data structure and function declarations for working with university data. It allows for storing and manipulating information about universities, including searching, sorting, and displaying university data. The struct and functions defined in this header file play a crucial role in managing and interacting with university information within the program.

University.cpp

UserInterface.h | UserInterface.cpp | Main.cpp | University.cpp | University.h*

QSRANKING PROJECT | (Global Scope)

```cpp
//University.cpp
#include "University.h"
#include <iostream>
#include "DataSeeder.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <algorithm>
#include "UserInterface.h"
#include "UserManager.h"

std::string University::getName() const {
    return name;
}

// Function to sort universities
void sortUniversities() {
    std::vector<University>& universities = DataSeeder::getUniversities();

    std::sort(universities.begin(), universities.end(), [](const University& a, const University& b) {
        return a.name < b.name;
        });
}

// Function to display information about all universities
void displayAllUniversities() {
    const std::vector<University>& universities = DataSeeder::getUniversities();
    for (const University& university : universities) {
        // Display various attributes of the university
        std::cout << "Name: " << university.getName() << std::endl;
        std::cout << "Location: " << university.location << std::endl;
        std::cout << "Rank: " << university.rank << std::endl;
        // ... Display other attributes
        std::cout << "-------------------------" << std::endl;
    }
}

// Function to search for a university by name using binary search
University* University::searchUniversityByNameBinary(const std::string& name) {
    std::vector<University>& universities = DataSeeder::getUniversities();
    sortUniversities();
```

```cpp

// Function to search for a university by name using binary search
University* University::searchUniversityByNameBinary(const std::string& name) {
    std::vector<University>& universities = DataSeeder::getUniversities();
    sortUniversities();

    int low = 0;
    int high = universities.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (universities[mid].name == name) {
            return &universities[mid];
        }
        if (universities[mid].name < name) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return nullptr; // University not found
}

// Function to search for a university by rank using linear search
University* searchUniversityByRankLinear(int rank) {
    std::vector<University>& universities = DataSeeder::getUniversities();
    for (int i = 0; i < universities.size(); ++i) {
        if (universities[i].rank == rank) {
            return &universities[i];
        }
    }
    return nullptr; // University not found
}

// Function to search for a university by name using linear search
University* University::searchUniversityByNameLinear(const std::string& name) {
    std::vector<University>& universities = DataSeeder::getUniversities();
    for (int i = 0; i < universities.size(); ++i) {
        if (universities[i].name == name) {
            return &universities[i];
        }
    }
    return nullptr; // University not found
}

```

```cpp
}

// Function to search for a university by rank using binary search
University* searchUniversityByRankBinary(int rank) {
    std::vector<University>& universities = DataSeeder::getUniversities();

    int low = 0;
    int high = universities.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (universities[mid].rank == rank) {
            return &universities[mid];
        }
        if (universities[mid].rank < rank) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return nullptr; // University not found
}

// Function to display the search menu
void SearchMenu() {
    int searchRank;   // Rank to be searched
    std::string searchName; // Name to be searched

    // Display the search menu options
    std::cout << "\n\n";
    std::cout << "========================\n";
    std::cout << "=====  Search Menu =====\n";
    std::cout << "========================\n";
    std::cout << "\n";
    // ... Display menu options

    // Prompt the user for their choice
    std::cout << "\n";
    std::cout << "========================\n";
    std::cout << "Enter your choice: ";

    // Create necessary objects for user management
    UserManager userManager;
    UserInterface ui(userManager);

```

```
    while (true) {
        int choice;
        std::cin >> choice;

        // Validate the choice
        if (choice < 0 || choice > 4) {
            std::cout << "Invalid option. Please try again.\n";
            continue; // This will skip the rest of the loop and go back to the beginning
        }

        // Process the chosen option
        switch (choice) {
        case 1:
            // Perform binary search for a university by rank
            std::cout << "Enter Rank to Search: ";
            std::cin >> searchRank;

            {
                // Measure execution time
                auto start = std::chrono::steady_clock::now();
                University* result = searchUniversityByRankBinary(searchRank);
                auto end = std::chrono::steady_clock::now();

                // Display the search result
                if (result) {
                    std::cout << "University found:\n";
                    std::cout << "Name: " << result->name << std::endl;
                    std::cout << "Location: " << result->location << std::endl;
                    std::cout << "Rank: " << result->rank << std::endl;
                }
                else {
                    std::cout << "University not found.\n";
                }

                // Display the execution time
                std::cout << "Execution Time: " << std::chrono::duration<double, std::milli>(end - start).count() << " ms\n";
                SearchMenu();
            }
            break;
            // ... Process other search options
        case 0:
            ui.mainMenu();
            return;
        }

        SearchMenu();  // Recursive call to display the search menu again
        return;  // Exit the function after the recursive call
    }
}
```

```
// Function to add feedback to a university
void University::addFeedback(const std::string& feedback) {
    feedbacks.push_back(feedback);
}

// Function to get the feedbacks of a university
std::vector<std::string> University::getFeedbacks() const {
    return feedbacks;
}

// Overloaded equality operator to compare universities
bool University::operator==(const University& other) const {
    // Compare the universities based on some criteria (e.g., name)
    return name == other.name;
}

// Function to add a reply to a university
void University::addReply(const std::string& reply) {
    feedbacks.push_back(reply);
}
```

The provided code is the implementation of the functions declared in the `University.h` header file. Here's an explanation of this part:

The `University.cpp` file starts by including necessary dependencies and header files. It defines member functions of the `University` struct that were declared in the header file. These functions include `getName()`, which simply returns the name of the university, and various search functions that allow searching for universities by name or rank using binary or linear search algorithms.

The file also includes the implementation of the `sortUniversities()` function, which sorts the universities based on their names using the `std::sort` algorithm. The `displayAllUniversities()` function

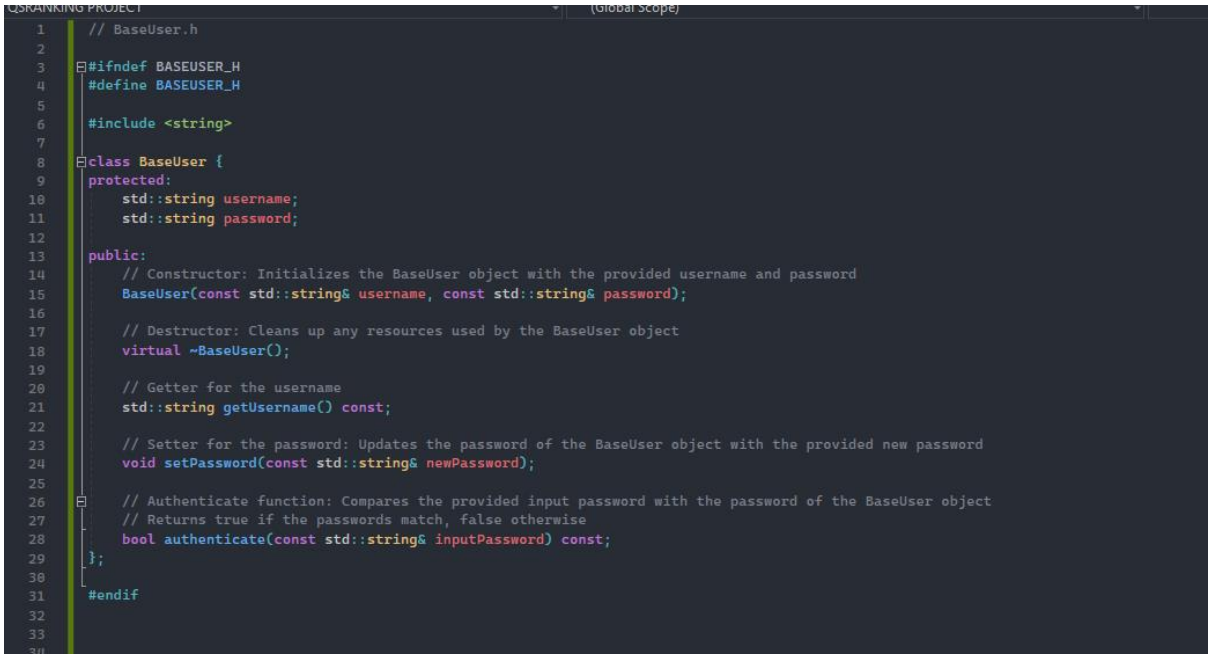
is implemented to display information about all universities, iterating over the vector of universities and printing their attributes.

Additionally, the file defines the `SearchMenu()` function, which displays a search menu and handles the user's choice. It includes options to search universities by rank or name using binary or linear search algorithms. The chosen algorithm is executed, and the result is displayed along with the execution time. The `SearchMenu()` function also allows the user to return to the main menu.

Lastly, the file implements functions related to feedback handling in the `University` struct. These functions include `addFeedback()`, which adds feedback to a university, `getFeedbacks()`, which retrieves all the feedbacks of a university, and `addReply()`, which adds a reply to a university.

Overall, the `University.cpp` file provides the implementation of the functions declared in the `University.h` header file, allowing for operations such as searching, sorting, and displaying university data, as well as handling feedback and replies.

BaseUser.h



```
// BaseUser.h

#ifndef BASEUSER_H
#define BASEUSER_H

#include <string>

class BaseUser {
protected:
    std::string username;
    std::string password;

public:
    // Constructor: Initializes the BaseUser object with the provided username and password
    BaseUser(const std::string& username, const std::string& password);

    // Destructor: Cleans up any resources used by the BaseUser object
    virtual ~BaseUser();

    // Getter for the username
    std::string getUsername() const;

    // Setter for the password: Updates the password of the BaseUser object with the provided new password
    void setPassword(const std::string& newPassword);

    // Authenticate function: Compares the provided input password with the password of the BaseUser object
    // Returns true if the passwords match, false otherwise
    bool authenticate(const std::string& inputPassword) const;
};

#endif
```

Explanation

The provided code defines the `BaseUser` class in the `BaseUser.h` header file. Here's an explanation of this part:

The `BaseUser` class serves as a base class for user objects. It has two protected member variables, `username` and `password`, which store the username and password associated with a user.

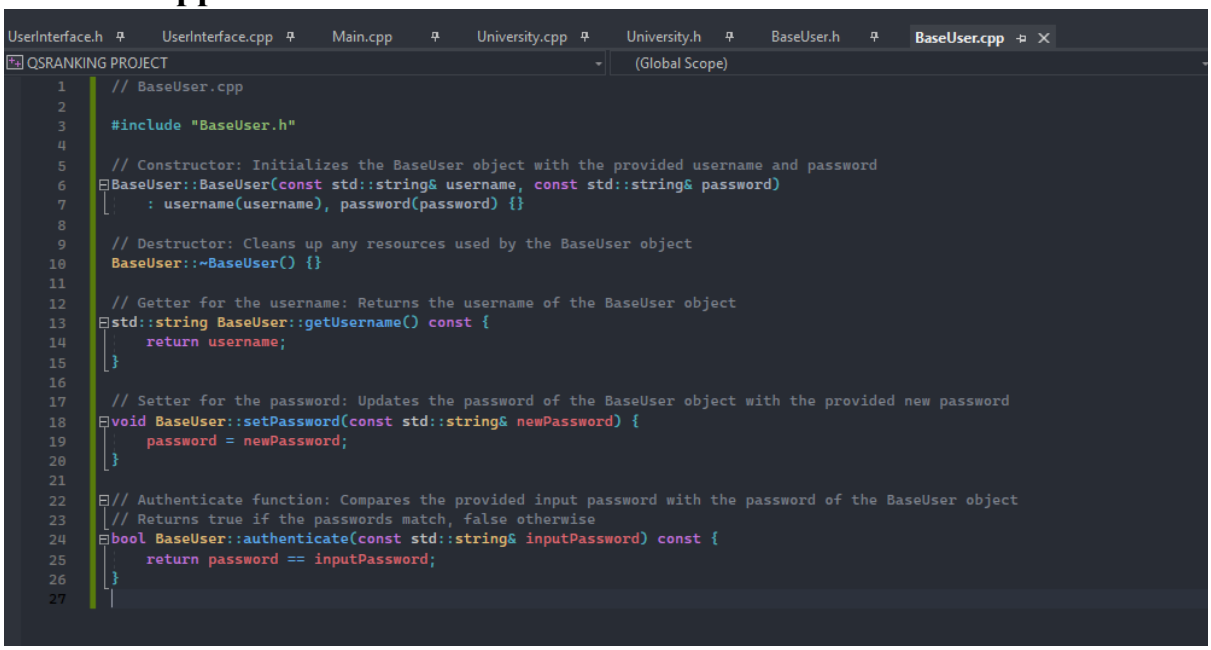
The constructor `BaseUser(const std::string& username, const std::string& password)` initializes a `BaseUser` object with the provided username and password. It sets the `username` and `password` member variables to the provided values.

The `BaseUser` class also provides several member functions. The `getUsername()` function returns the username of the `BaseUser` object. The `setPassword(const std::string& newPassword)` function allows updating the password of the `BaseUser` object with a new password by assigning the provided value to the `password` member variable.

The `authenticate(const std::string& inputPassword)` function compares the provided input password with the stored password of the `BaseUser` object. It returns `true` if the passwords match, indicating successful authentication, and `false` otherwise.

Overall, the `BaseUser` class provides a basic structure for representing a user with a username and password. It enables setting and retrieving the username, updating the password, and authenticating user credentials. Derived classes can inherit from this base class and extend its functionality to cater to specific user types or requirements.

BaseUser.cpp



Explanation

This code represents the implementation of the `BaseUser` class in the `BaseUser.cpp` file. Here's an explanation of this part:

The `BaseUser` class serves as a base class for user objects. It defines member functions to initialize, access, and modify the username and password associated with a user.

The constructor `BaseUser::BaseUser(const std::string& username, const std::string& password)` initializes a `BaseUser` object with the provided username and password. It uses member initialization syntax to assign the values of the provided username and password to the `username` and `password` member variables, respectively.

The destructor `BaseUser::~BaseUser()` is defined to clean up any resources used by the `BaseUser` object. In this case, the destructor is empty, indicating that no specific cleanup actions are required.

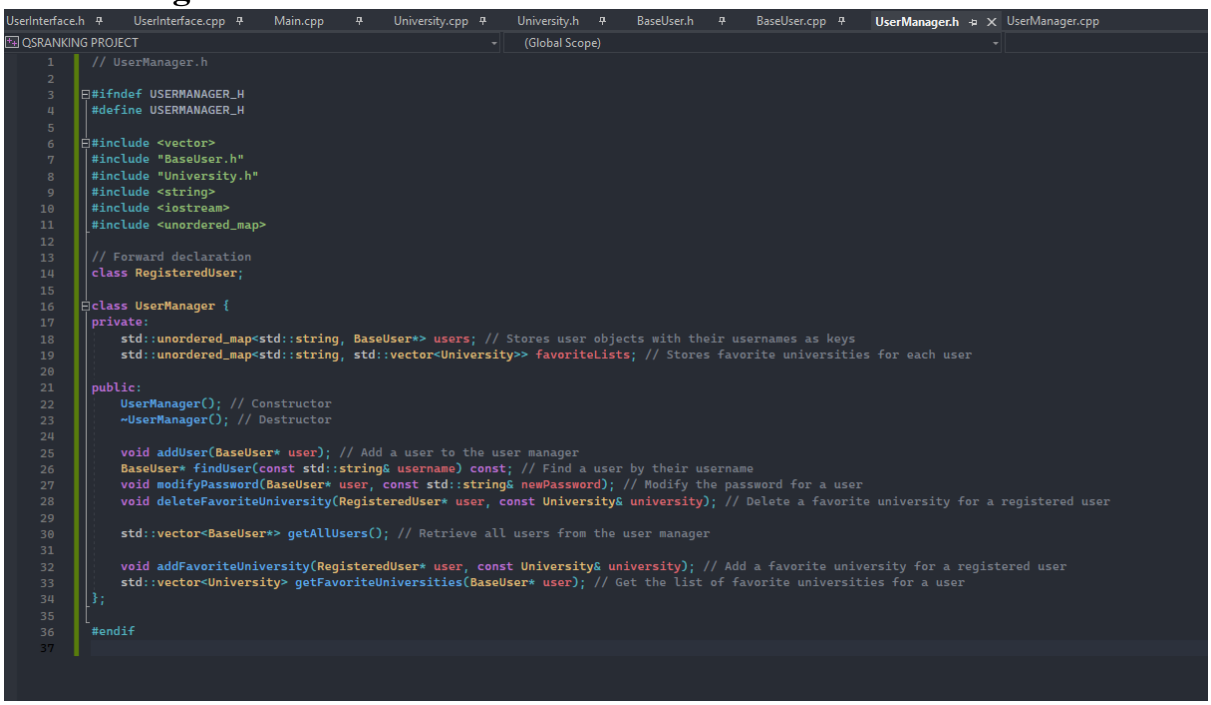
The `getUsername()` function returns the username of the `BaseUser` object. It simply returns the value stored in the `username` member variable.

The `setPassword(const std::string& newPassword)` function allows updating the password of the `BaseUser` object. It takes a new password as input and assigns it to the `password` member variable.

The `authenticate(const std::string& inputPassword)` function compares the provided input password with the stored password of the `BaseUser` object. It checks if the two passwords are equal and returns `true` if they match, indicating successful authentication. Otherwise, it returns `false`.

Overall, this code implements the basic functionality for managing user authentication and password-related operations in the `BaseUser` class. It provides methods to access and modify the username and password, as well as a function to verify the entered password during the authentication process. Derived classes can inherit from this base class and extend its functionality to cater to specific user types or requirements.

UserManager.h



```h
// UserManager.h

#ifndef USERMANAGER_H
#define USERMANAGER_H

#include <vector>
#include "BaseUser.h"
#include "University.h"
#include <string>
#include <iostream>
#include <unordered_map>

// Forward declaration
class RegisteredUser;

class UserManager {
private:
    std::unordered_map<std::string, BaseUser*> users; // Stores user objects with their usernames as keys
    std::unordered_map<std::string, std::vector<University>> favoriteLists; // Stores favorite universities for each user

public:
    UserManager(); // Constructor
    ~UserManager(); // Destructor

    void addUser(BaseUser* user); // Add a user to the user manager
    BaseUser* findUser(const std::string& username) const; // Find a user by their username
    void modifyPassword(BaseUser* user, const std::string& newPassword); // Modify the password for a user
    void deleteFavoriteUniversity(RegisteredUser* user, const University& university); // Delete a favorite university for a registered user

    std::vector<BaseUser*> getAllUsers(); // Retrieve all users from the user manager

    void addFavoriteUniversity(RegisteredUser* user, const University& university); // Add a favorite university for a registered user
    std::vector<University> getFavoriteUniversities(BaseUser* user); // Get the list of favorite universities for a user
};

#endif
```

Explanation

This code snippet the implementation of the `UserManager` class in the `UserManager.h` header file. Here's an explanation of this part:

The `UserManager` class serves as a manager for user objects and provides functions to perform user-related operations. It has two private member variables: `users`, which is an unordered map that stores user objects with their usernames as keys, and `favoriteLists`, which is an unordered map that stores favorite universities for each user.

The constructor `UserManager()` initializes a `UserManager` object. The destructor `~UserManager()` is responsible for cleaning up any resources used by the `UserManager` object.

The `addUser(BaseUser* user)` function adds a user to the user manager by inserting the user object into the `users` unordered map with the username as the key.

The `findUser(const std::string& username) const` function searches for a user by their username in the `users` unordered map and returns the corresponding user object if found.

The `modifyPassword(BaseUser* user, const std::string& newPassword)` function modifies the password for a user by updating the password in the corresponding user object stored in the `users` unordered map.

The `deleteFavoriteUniversity(RegisteredUser* user, const University& university)` function removes a favorite university from the list of favorites for a registered user by deleting it from the vector of favorite universities stored in the `favoriteLists` unordered map.

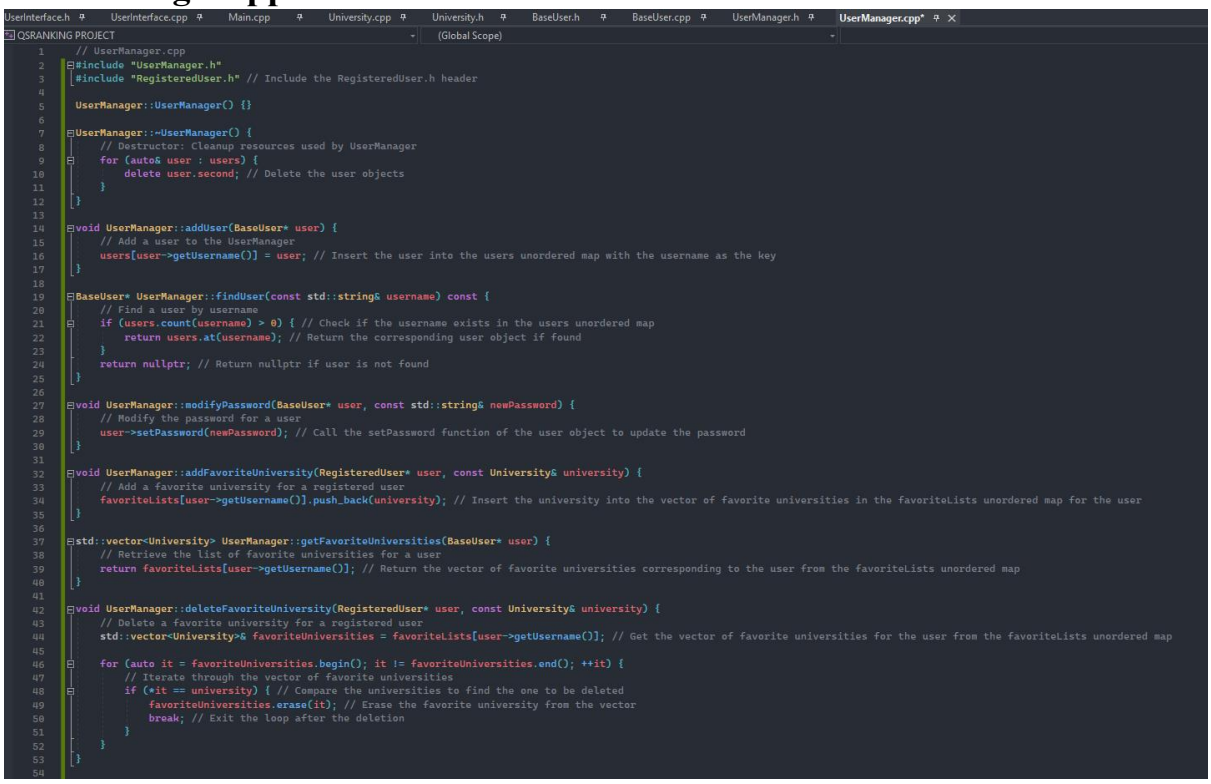
The `getAllUsers()` function retrieves all users from the user manager by returning a vector of `BaseUser` pointers containing all the user objects stored in the `users` unordered map.

The `addFavoriteUniversity(RegisteredUser* user, const University& university)` function adds a favorite university for a registered user by inserting it into the vector of favorite universities in the `favoriteLists` unordered map.

The `getFavoriteUniversities(BaseUser* user)` function retrieves the list of favorite universities for a user by returning the corresponding vector of favorite universities from the `favoriteLists` unordered map.

Overall, the `UserManager` class provides the functionality to manage user objects, including adding users, finding users, modifying passwords, managing favorite universities, and retrieving user information. It acts as a central component for user-related operations in the system.

UserManager.cpp



```cpp
// UserManager.cpp
#include "UserManager.h"
#include "RegisteredUser.h" // Include the RegisteredUser.h header

UserManager::UserManager() {}

UserManager::~UserManager() {
    // Destructor: Cleanup resources used by UserManager
    for (auto& user : users) {
        delete user.second; // Delete the user objects
    }
}

void UserManager::addUser(BaseUser* user) {
    // Add a user to the UserManager
    users[user->getUsername()] = user; // Insert the user into the users unordered map with the username as the key
}

BaseUser* UserManager::findUser(const std::string& username) const {
    // Find a user by username
    if (users.count(username) > 0) { // Check if the username exists in the users unordered map
        return users.at(username); // Return the corresponding user object if found
    }
    return nullptr; // Return nullptr if user is not found
}

void UserManager::modifyPassword(BaseUser* user, const std::string& newPassword) {
    // Modify the password for a user
    user->setPassword(newPassword); // Call the setPassword function of the user object to update the password
}

void UserManager::addFavoriteUniversity(RegisteredUser* user, const University& university) {
    // Add a favorite university for a registered user
    favoriteLists[user->getUsername()].push_back(university); // Insert the university into the vector of favorite universities in the favoriteLists unordered map for the user
}

std::vector<University> UserManager::getFavoriteUniversities(BaseUser* user) {
    // Retrieve the list of favorite universities for a user
    return favoriteLists[user->getUsername()]; // Return the vector of favorite universities corresponding to the user from the favoriteLists unordered map
}

void UserManager::deleteFavoriteUniversity(RegisteredUser* user, const University& university) {
    // Delete a favorite university for a registered user
    std::vector<University>& favoriteUniversities = favoriteLists[user->getUsername()]; // Get the vector of favorite universities for the user from the favoriteLists unordered map

    for (auto it = favoriteUniversities.begin(); it != favoriteUniversities.end(); ++it) {
        // Iterate through the vector of favorite universities
        if (*it == university) { // Compare the universities to find the one to be deleted
            favoriteUniversities.erase(it); // Erase the favorite university from the vector
            break; // Exit the loop after the deletion
        }
    }
}
```

```cpp
std::vector<BaseUser*> UserManager::getAllUsers() {
    // Retrieve all users from the user manager
    std::vector<BaseUser*> allUsers;
    for (const auto& user : users) {
        allUsers.push_back(user.second); // Add each user object to the vector
    }
    return allUsers; // Return the vector of all users
}
```

Explanation

This part of the code focuses on the implementation of the `UserManager` class. Here's a breakdown of its functionality:

The `UserManager` class is responsible for managing user accounts and favorite universities. It utilizes an unordered map called `users` to store user objects, with the username as the key. It also uses another unordered map called `favoriteLists` to store the favorite universities for each registered user.

The class provides several member functions. The `addUser` function takes a `BaseUser` pointer as input and adds the user to the `users` unordered map using the username as the key. The `findUser` function searches for a user by their username in the `users` unordered map and returns the corresponding user object if found.

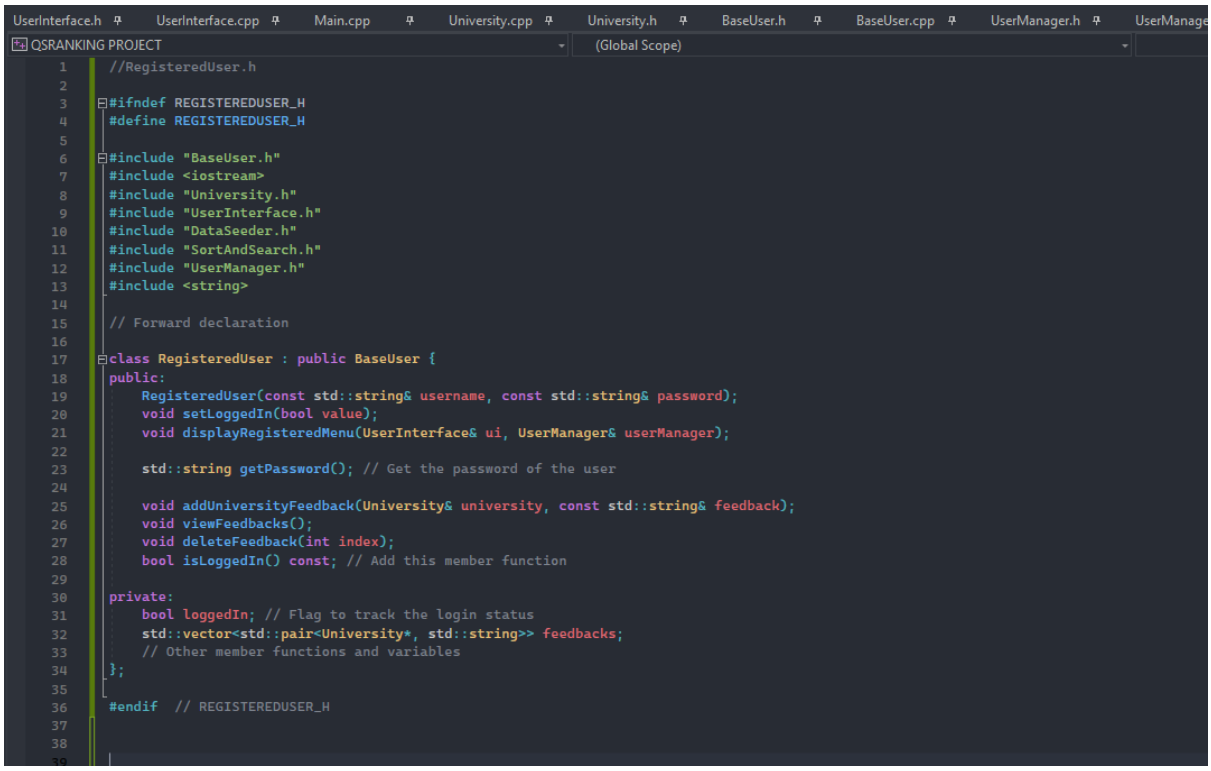
The `modifyPassword` function allows a user's password to be updated by calling the `setPassword` function of the user object. The `addFavoriteUniversity` function adds a favorite university to the list of favorites for a registered user by inserting it into the vector of universities associated with the user in the `favoriteLists` unordered map.

The `getFavoriteUniversities` function retrieves the list of favorite universities for a user by returning the vector of universities corresponding to the user in the `favoriteLists` unordered map. The `deleteFavoriteUniversity` function allows a registered user to remove a favorite university from their list of favorites by iterating through the vector of universities and erasing the specified university.

Lastly, the `getAllUsers` function retrieves all users from the `users` unordered map and returns them as a vector of `BaseUser` pointers.

Overall, the `UserManager` class provides functionality for managing user accounts, modifying passwords, and managing favorite universities for registered users.

RegisteredUser.h




```cpp
//RegisteredUser.h

#ifndef REGISTEREDUSER_H
#define REGISTEREDUSER_H

#include "BaseUser.h"
#include <iostream>
#include "University.h"
#include "UserInterface.h"
#include "DataSeeder.h"
#include "SortAndSearch.h"
#include "UserManager.h"
#include <string>

// Forward declaration

class RegisteredUser : public BaseUser {
public:
    RegisteredUser(const std::string& username, const std::string& password);
    void setLoggedIn(bool value);
    void displayRegisteredMenu(UserInterface& ui, UserManager& userManager);

    std::string getPassword(); // Get the password of the user

    void addUniversityFeedback(University& university, const std::string& feedback);
    void viewFeedbacks();
    void deleteFeedback(int index);
    bool isLoggedIn() const; // Add this member function

private:
    bool loggedIn; // Flag to track the login status
    std::vector<std::pair<University*, std::string>> feedbacks;
    // Other member functions and variables
};

#endif  // REGISTEREDUSER_H
```

Explanation

The code represents the declaration of the `RegisteredUser` class in the `RegisteredUser.h` file. Here's an explanation of its functionality:

The `RegisteredUser` class is derived from the `BaseUser` class, which means it inherits its attributes and methods. It represents a registered user in the system. The class has member functions to set and retrieve the user's password, add feedback for universities, view feedbacks, and delete feedback. It also has a `loggedIn` flag to track the login status of the user.

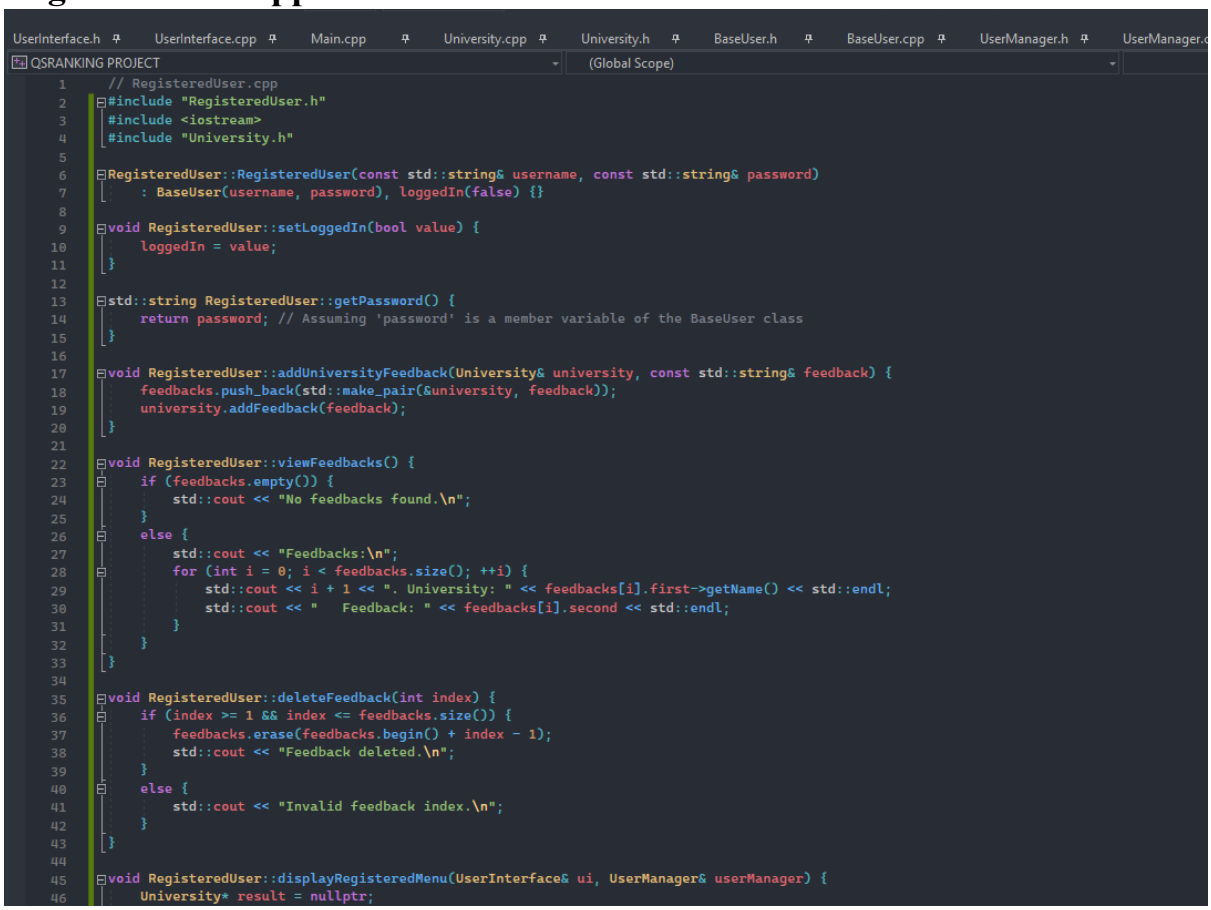
The constructor `RegisteredUser` takes the `username` and `password` as parameters and initializes the `loggedIn` flag as `false`. This constructor is responsible for creating instances of registered users with the provided username and password.

The `setLoggedIn` function allows setting the login status of the user by updating the `loggedIn` flag with the provided value. This function is useful for updating the login status when the user logs in or logs out.

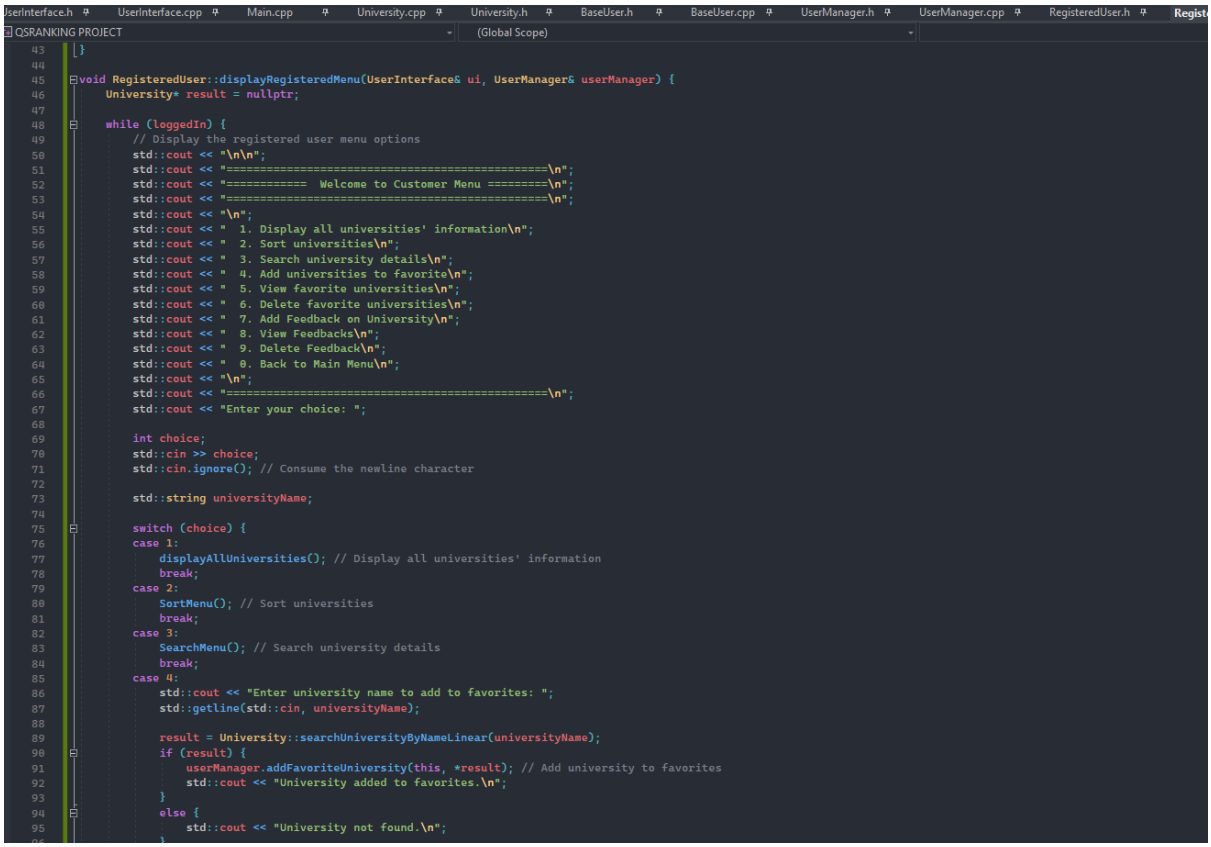
The `displayRegisteredMenu` function takes references to the `UserInterface` and `UserManager` objects as parameters. It is responsible for displaying the registered user menu, allowing the user to interact with the program as a registered user. This function serves as a bridge between the user interface and the registered user functionalities.

Overall, the `RegisteredUser` class encapsulates the behavior and attributes specific to registered users, providing functions to manage their feedbacks, retrieve password information, and handle the login status. It extends the functionality of the `BaseUser` class to cater to the specific requirements of registered users in the program.

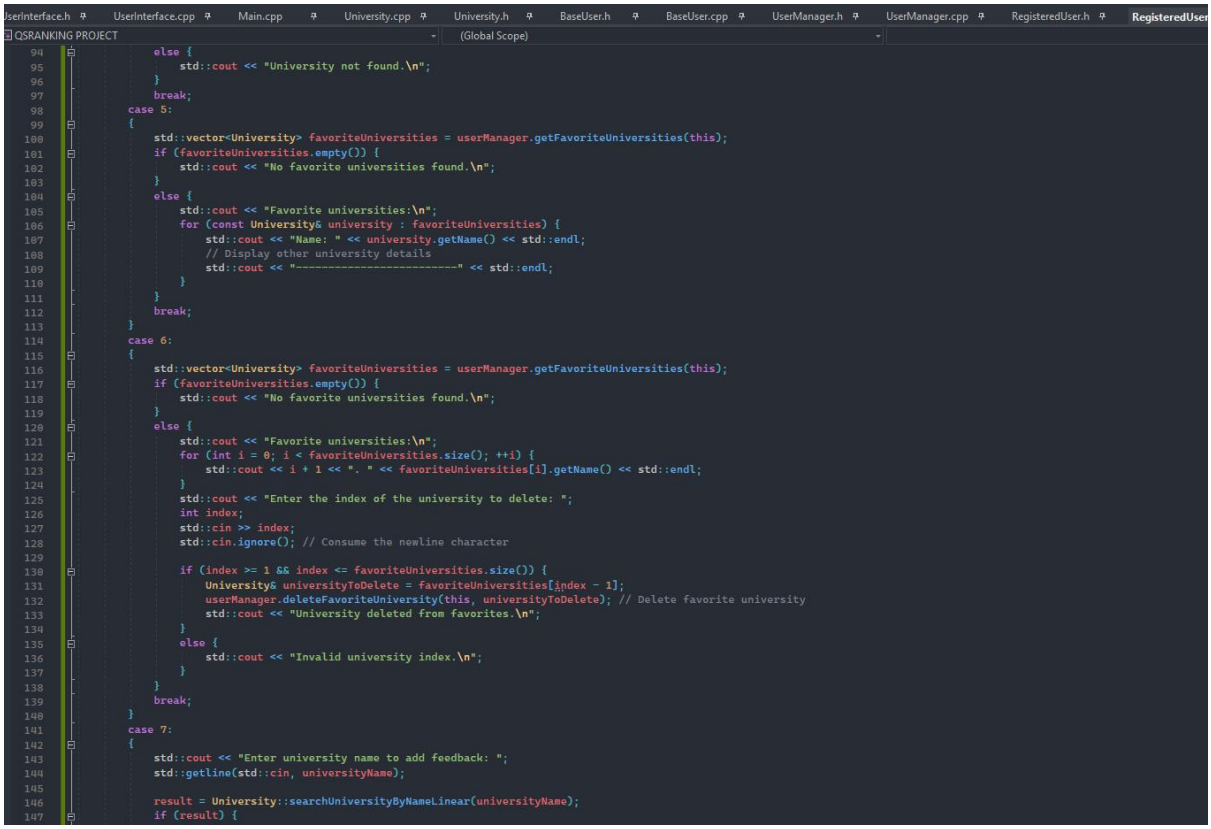
RegisteredUser.cpp



```cpp
// RegisteredUser.cpp
#include "RegisteredUser.h"
#include <iostream>
#include "University.h"

RegisteredUser::RegisteredUser(const std::string& username, const std::string& password)
    : BaseUser(username, password), loggedIn(false) {}

void RegisteredUser::setLoggedIn(bool value) {
    loggedIn = value;
}

std::string RegisteredUser::getPassword() {
    return password; // Assuming 'password' is a member variable of the BaseUser class
}

void RegisteredUser::addUniversityFeedback(University& university, const std::string& feedback) {
    feedbacks.push_back(std::make_pair(&university, feedback));
    university.addFeedback(feedback);
}

void RegisteredUser::viewFeedbacks() {
    if (feedbacks.empty()) {
        std::cout << "No feedbacks found.\n";
    }
    else {
        std::cout << "Feedbacks:\n";
        for (int i = 0; i < feedbacks.size(); ++i) {
            std::cout << i + 1 << ". University: " << feedbacks[i].first->getName() << std::endl;
            std::cout << "   Feedback: " << feedbacks[i].second << std::endl;
        }
    }
}

void RegisteredUser::deleteFeedback(int index) {
    if (index >= 1 && index <= feedbacks.size()) {
        feedbacks.erase(feedbacks.begin() + index - 1);
        std::cout << "Feedback deleted.\n";
    }
    else {
        std::cout << "Invalid feedback index.\n";
    }
}

void RegisteredUser::displayRegisteredMenu(UserInterface& ui, UserManager& userManager) {
    University* result = nullptr;
```



```cpp
}

void RegisteredUser::displayRegisteredMenu(UserInterface& ui, UserManager& userManager) {
    University* result = nullptr;

    while (loggedIn) {
        // Display the registered user menu options
        std::cout << "\n\n";
        std::cout << "================================================\n";
        std::cout << "============  Welcome to Customer Menu =========\n";
        std::cout << "================================================\n";
        std::cout << "\n";
        std::cout << "  1. Display all universities' information\n";
        std::cout << "  2. Sort universities\n";
        std::cout << "  3. Search university details\n";
        std::cout << "  4. Add universities to favorite\n";
        std::cout << "  5. View favorite universities\n";
        std::cout << "  6. Delete favorite universities\n";
        std::cout << "  7. Add Feedback on University\n";
        std::cout << "  8. View Feedbacks\n";
        std::cout << "  9. Delete Feedback\n";
        std::cout << "  0. Back to Main Menu\n";
        std::cout << "\n";
        std::cout << "================================================\n";
        std::cout << "Enter your choice: ";

        int choice;
        std::cin >> choice;
        std::cin.ignore(); // Consume the newline character

        std::string universityName;

        switch (choice) {
        case 1:
            displayAllUniversities(); // Display all universities' information
            break;
        case 2:
            SortMenu(); // Sort universities
            break;
        case 3:
            SearchMenu(); // Search university details
            break;
        case 4:
            std::cout << "Enter university name to add to favorites: ";
            std::getline(std::cin, universityName);

            result = University::searchUniversityByNameLinear(universityName);
            if (result) {
                userManager.addFavoriteUniversity(this, *result); // Add university to favorites
                std::cout << "University added to favorites.\n";
            }
            else {
                std::cout << "University not found.\n";
```



```
            else {
                std::cout << "University not found.\n";
            }
            break;
        case 5:
        {
            std::vector<University> favoriteUniversities = userManager.getFavoriteUniversities(this);
            if (favoriteUniversities.empty()) {
                std::cout << "No favorite universities found.\n";
            }
            else {
                std::cout << "Favorite universities:\n";
                for (const University& university : favoriteUniversities) {
                    std::cout << "Name: " << university.getName() << std::endl;
                    // Display other university details
                    std::cout << "--------------------------" << std::endl;
                }
            }
            break;
        }
        case 6:
        {
            std::vector<University> favoriteUniversities = userManager.getFavoriteUniversities(this);
            if (favoriteUniversities.empty()) {
                std::cout << "No favorite universities found.\n";
            }
            else {
                std::cout << "Favorite universities:\n";
                for (int i = 0; i < favoriteUniversities.size(); ++i) {
                    std::cout << i + 1 << ". " << favoriteUniversities[i].getName() << std::endl;
                }
                std::cout << "Enter the index of the university to delete: ";
                int index;
                std::cin >> index;
                std::cin.ignore(); // Consume the newline character

                if (index >= 1 && index <= favoriteUniversities.size()) {
                    University& universityToDelete = favoriteUniversities[index - 1];
                    userManager.deleteFavoriteUniversity(this, universityToDelete); // Delete favorite university
                    std::cout << "University deleted from favorites.\n";
                }
                else {
                    std::cout << "Invalid university index.\n";
                }
            }
            break;
        }
        case 7:
        {
            std::cout << "Enter university name to add feedback: ";
            std::getline(std::cin, universityName);

            result = University::searchUniversityByNameLinear(universityName);
            if (result) {
```

```
            }
        case 7:
        {
            std::cout << "Enter university name to add feedback: ";
            std::getline(std::cin, universityName);

            result = University::searchUniversityByNameLinear(universityName);
            if (result) {
                std::cout << "Enter your feedback: ";
                std::string feedback;
                std::getline(std::cin, feedback);

                addUniversityFeedback(*result, feedback); // Add feedback on university
                std::cout << "Feedback added to the university.\n";
            }
            else {
                std::cout << "University not found.\n";
            }
            break;
        }
        case 8:
            viewFeedbacks(); // View feedbacks
            break;
        case 9:
            std::cout << "Enter the index of the feedback to delete: ";
            int index;
            std::cin >> index;
            deleteFeedback(index); // Delete feedback
            break;
        case 0:
            ui.logoutUser(); // Back to the main menu
            return;
        default:
            std::cout << "Invalid option. Please try again.\n";
            break;
        }
    }
}

bool RegisteredUser::isLoggedIn() const {
    return loggedIn;
}
```

Explanation

The `RegisteredUser` class represents a registered user within the system. Here's an explanation of the code:

The `RegisteredUser` constructor initializes the `BaseUser` base class with the provided username and password. The `loggedIn` flag is set to `false` initially. The `setLoggedIn` function allows updating the login status of the user.

The `getPassword` function returns the password of the registered user.

The `addUniversityFeedback` function allows the registered user to add feedback to a specific university. It creates a pair of the university and the feedback and adds it to the `feedbacks` vector. It also calls the `addFeedback` function of the university to store the feedback.

The `viewFeedbacks` function displays all the feedbacks given by the registered user. It iterates over the `feedbacks` vector and prints the university name and corresponding feedback.

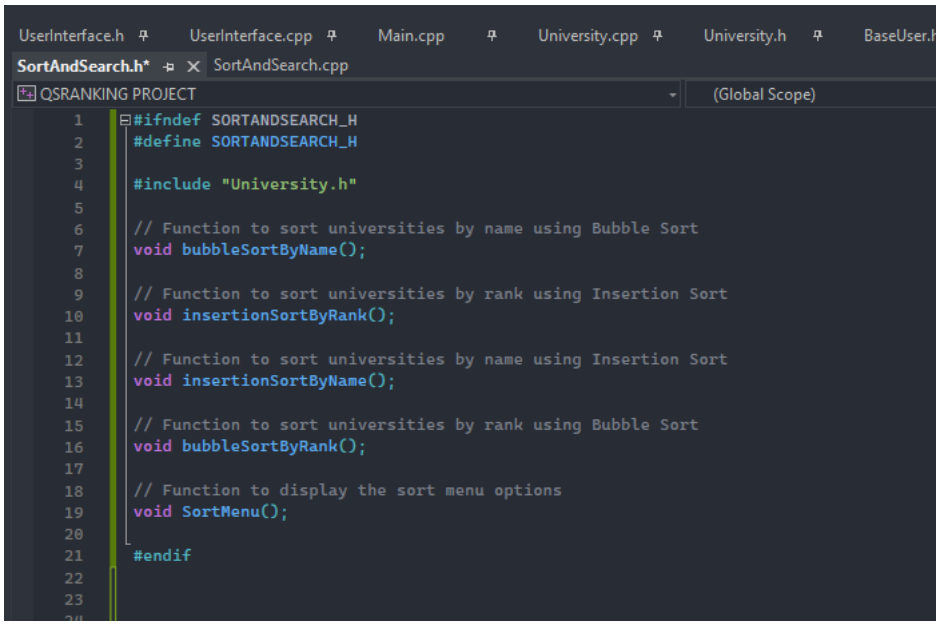
The `deleteFeedback` function removes a feedback based on the provided index. It checks if the index is valid and erases the feedback from the `feedbacks` vector.

The `displayRegisteredMenu` function presents the menu options for a logged-in registered user. It prompts the user for their choice and performs the corresponding actions, such as displaying university information, sorting universities, searching for universities, adding universities to favorites, viewing favorite universities, deleting favorite universities, adding feedback on a university, viewing feedbacks, deleting feedbacks, or going back to the main menu.

The `isLoggedIn` function returns the login status of the registered user. It checks the `loggedIn` flag and returns `true` if the user is logged in and `false` otherwise.

Overall, the `RegisteredUser` class provides functionalities for a registered user to interact with universities, provide feedback, and manage their favorite universities.

SortAndSearch.h



Explanation

This part consists of function declarations related to sorting and searching universities in the header file "SortAndSearch.h". It includes function declarations for sorting universities by name and rank using bubble sort and insertion sort algorithms. The "SortMenu()" function is also declared, which is responsible for displaying the sorting menu options to the user. These declarations provide a blueprint for the implementation of sorting and searching functionalities in the program.

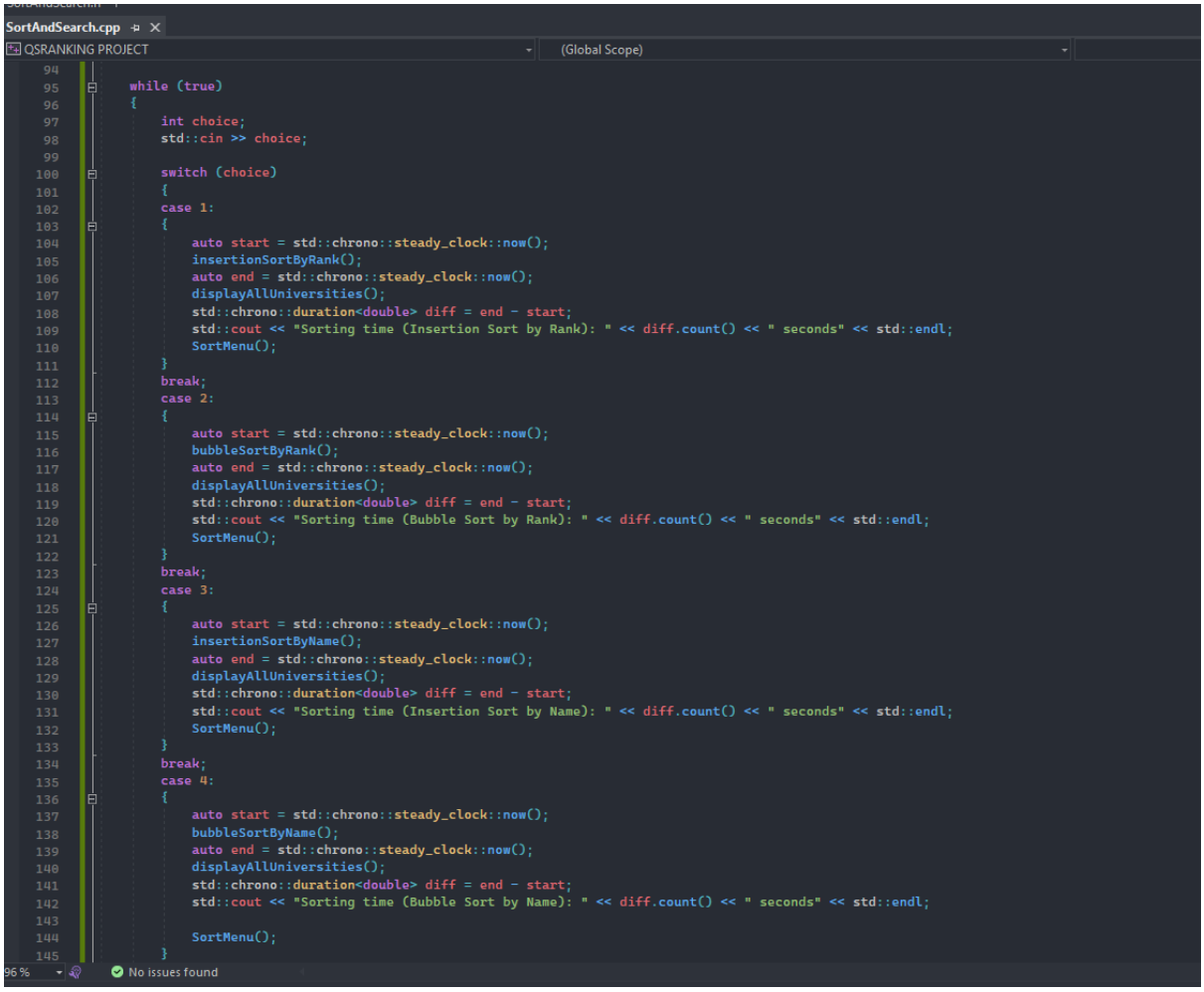
SortAndSearch.cpp

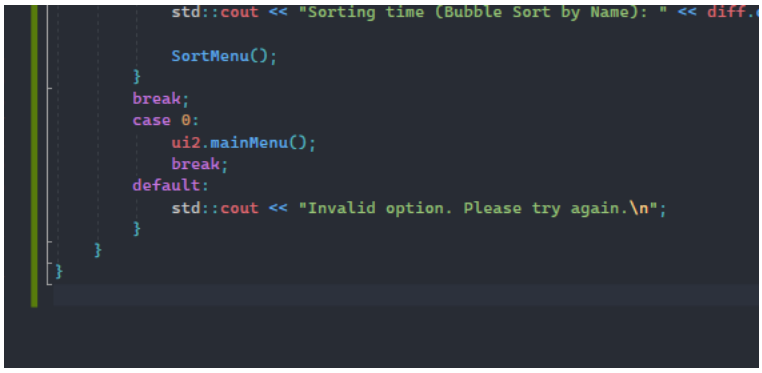
SortAndSearch.cpp

QSRANKING PROJECT (Global Scope)

```cpp
//SortAndSearch.cpp

#include "SortAndSearch.h"
#include "University.h" // Include the University.h header file
#include <algorithm>
#include <chrono>
#include <iostream>
#include "UserInterface.h"
#include "DataSeeder.h"
#include "RegisteredUser.h"
#include <string>
#include <vector>
#include <unordered_map>

UserManager userManager;  // Create a UserManager object
UserInterface ui2(userManager);

// Function to sort universities by name using Bubble Sort
void bubbleSortByName() {
    std::vector<University>& universityData = DataSeeder::getUniversities();
    int n = universityData.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (universityData[j].name > universityData[j + 1].name) {
                std::swap(universityData[j], universityData[j + 1]);
            }
        }
    }
}

// Function to sort universities by name using Insertion Sort
void insertionSortByName() {
    std::vector<University>& universityData = DataSeeder::getUniversities();
    int n = universityData.size();
    for (int i = 1; i < n; i++) {
        University key = universityData[i];
        int j = i - 1;

        while (j >= 0 && universityData[j].name > key.name) {
            universityData[j + 1] = universityData[j];
            j = j - 1;
        }
        universityData[j + 1] = key;
    }
}
```

```cpp

// Function to sort universities by rank using Insertion Sort
void insertionSortByRank() {
    std::vector<University>& universityData = DataSeeder::getUniversities();
    int n = universityData.size();
    for (int i = 1; i < n; i++) {
        University key = universityData[i];
        int j = i - 1;

        while (j >= 0 && universityData[j].rank > key.rank) {
            universityData[j + 1] = universityData[j];
            j = j - 1;
        }
        universityData[j + 1] = key;
    }
}

// Function to sort universities by rank using Bubble Sort
void bubbleSortByRank() {
    std::vector<University>& universityData = DataSeeder::getUniversities();
    int n = universityData.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (universityData[j].rank > universityData[j + 1].rank) {
                std::swap(universityData[j], universityData[j + 1]);
            }
        }
    }
}

// Function to display the sort menu options to the user
void SortMenu()
{
    std::cout << "\n\n";
    std::cout << "======================\n";
    std::cout << "=====  Sort Menu =====\n";
    std::cout << "======================\n";
    std::cout << "\n";

    std::cout << "  1. Sort universities By Rank (Insertion Sort)\n";
    std::cout << "  2. Sort universities By Rank (Bubble Sort)\n";
    std::cout << "  3. Sort universities By Name (Insertion Sort)\n";
    std::cout << "  4. Sort universities By Name (Bubble Sort)\n";
    std::cout << "   0. Main Menu\n";

    std::cout << "\n";
    std::cout << "==================\n";
    std::cout << "Enter your choice: ";

```





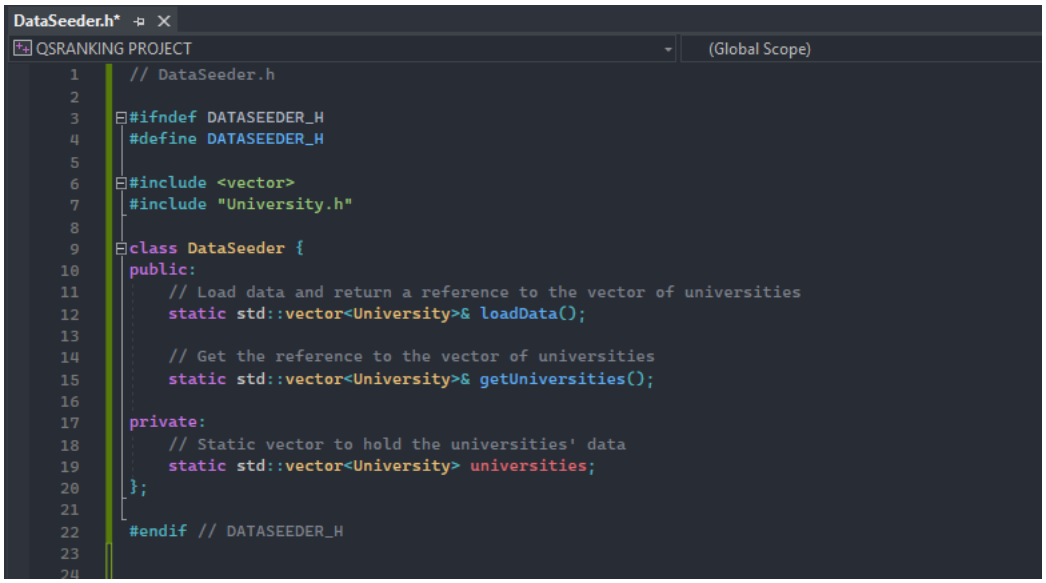
Explanation

In this code, the "SortAndSearch.cpp" file contains the implementation of sorting and searching functions for universities. The functions `bubbleSortByName()`, `insertionSortByName()`, `insertionSortByRank()`, and `bubbleSortByRank()` are sorting functions that use the bubble sort and insertion sort algorithms.

The bubble sort algorithm compares adjacent elements and swaps them if they are in the wrong order, repeatedly iterating through the list until the list is sorted. The insertion sort algorithm builds the final sorted array one item at a time, comparing each item with the previous items and shifting them if necessary. These sorting functions are applied to the `std::vector<University>` container obtained from the `DataSeeder::getUniversities()` function, which provides access to the university data.

The `SortMenu()` function displays a menu of sorting options to the user and performs the sorting operation based on the user's choice. It uses a while loop to repeatedly prompt the user for their choice and then calls the corresponding sorting function. It also measures the time taken for sorting using the `std::chrono` library and displays it to the user. The `SortMenu()` function is designed to be interactive and allows the user to sort the universities by name or rank using different sorting algorithms.

DataSeeder.h



Explanation

The code in the `DataSeeder.h` header file defines a class called `DataSeeder`. This class provides static functions to load data and access a vector of universities. The `loadData()` function is responsible for loading the data and returns a reference to the vector of universities. The `getUniversities()` function returns a reference to the same vector of universities. The private static member variable `universities` is used to store the data. This header file allows other parts of the code to easily access and manipulate the university data through the static functions provided by the `DataSeeder` class.

DataSeeder.cpp

DataSeeder.cpp

QSRANKING PROJECT (Global Scope)

```cpp
//DataSeeder.cpp
#include "DataSeeder.h"
#include <iostream>
#include <fstream>
#include <sstream>

std::vector<University> DataSeeder::universities;

// Function to load data from a CSV file and populate the universities vector
std::vector<University>& DataSeeder::loadData() {
    std::ifstream file("data.csv");

    if (!file.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        return universities;
    }

    universities.clear();

    std::string line;
    std::getline(file, line);  // Skip the header line

    while (std::getline(file, line)) {
        std::istringstream iss(line);
        std::string token;
        University university;

        try {
            std::getline(iss, token, ',');
            university.rank = (token.empty() ? 0 : std::stoi(token));
            std::getline(iss, university.name, ',');
            std::getline(iss, university.locationCode, ',');
            std::getline(iss, university.location, ',');

            std::vector<double> scores;
            std::vector<int> ranks;

            // Parse the scores and ranks for different criteria
            for (int i = 0; i < 13; i++) {
                std::getline(iss, token, ',');
                if (token.empty() || token == "-" || token == "NA" || token == "601+") {
                    scores.push_back(0.0);
                    ranks.push_back(0);
                }
                else {
                    scores.push_back(std::stod(token));
                    std::getline(iss, token, ',');
                    if (token.empty() || token == "-" || token == "NA" || token == "601+") {
                        ranks.push_back(0);
                    }
                    else {
```

```
                    }
                    else {
                        ranks.push_back(std::stoi(token));
                    }
                }
            }

            // Assign the parsed scores and ranks to the university object
            university.arScore = scores[0];
            university.arRank = ranks[0];
            university.erScore = scores[1];
            university.erRank = ranks[1];
            // Assign other scores and ranks in a similar manner...

            universities.push_back(university);
        }
        catch (const std::invalid_argument& e) {
            // Handle the case of invalid data in the CSV file
            continue;
        }
        catch (const std::exception& e) {
            // Handle other exceptions that might occur during parsing
            continue;
        }
    }

    file.close();

    return universities;
}

// Function to retrieve the universities vector
std::vector<University>& DataSeeder::getUniversities() {
    return universities;
}

```

Explanation

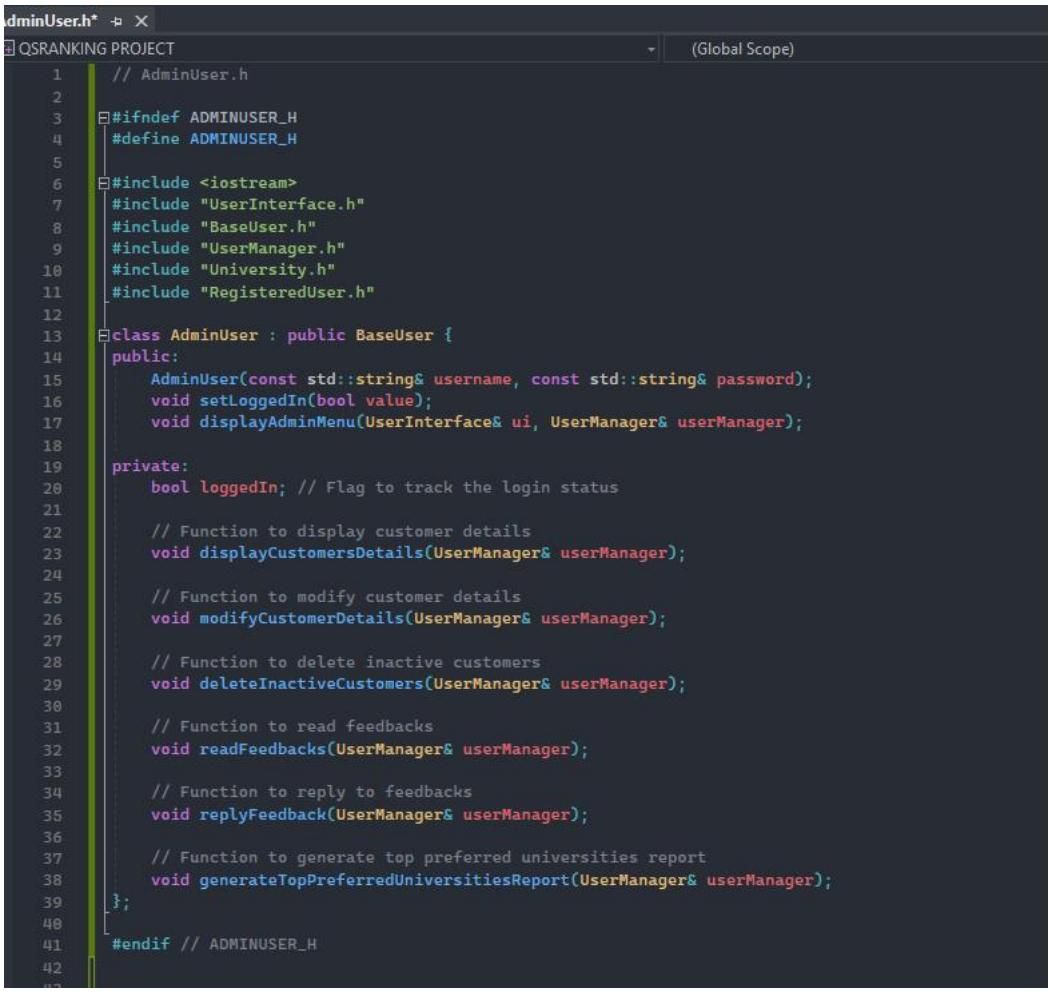
This part of the code is responsible for loading data from a CSV file and populating a vector of `University` objects.

The `loadData()` function starts by opening the "data.csv" file using `std::ifstream`. If the file fails to open, an error message is printed, and an empty vector is returned. Next, the function clears the `universities` vector to ensure it's empty before populating it with data from the CSV file.

Using `std::getline`, the function reads the lines from the file one by one. Each line represents a university entry with different fields separated by commas. The function then uses `std::istringstream` and string manipulation operations to extract the data from each field and assign it to the corresponding fields of a `University` object. Invalid or missing data is handled by assigning default values (e.g., 0) or skipping the line altogether. The constructed `University` object is then added to the `universities` vector.

Finally, the file is closed, and the function returns a reference to the populated `universities` vector. This allows other parts of the code to access and work with the loaded university data.

AdminUser.h



Explanation:

The `AdminUser` class represents an admin user and is derived from the `BaseUser` class. It has a constructor to initialize the username and password, and the `setLoggedIn` function to update the login status of the admin user. The `displayAdminMenu` function is responsible for displaying the admin menu and handling admin-specific actions. It takes references to the `UserInterface` and `UserManager` objects as parameters. The private member functions in the class include displaying customer details, modifying customer details, deleting inactive customers, reading and replying to feedbacks, and generating a report of the top preferred universities. The `loggedIn` flag is used to track whether the admin user is currently logged in or not.

AdminUser.cpp

AdminUser.cpp

QSRANKING PROJECT (Global Scope)

```cpp
// AdminUser.cpp

#include "AdminUser.h"
#include <iostream>

AdminUser::AdminUser(const std::string& username, const std::string& password)
    : BaseUser(username, password), loggedIn(false) {}

void AdminUser::setLoggedIn(bool value) {
    loggedIn = value;
}

void AdminUser::displayCustomersDetails(UserManager& userManager) {
    std::cout << "Customers' details:\n";

    // Retrieve all users from the user manager
    std::vector<BaseUser*> users = userManager.getAllUsers();

    // Display each customer's details
    for (BaseUser* user : users) {
        // Check if the user is a registered customer
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            std::cout << "Username: " << registeredUser->getUsername() << std::endl;
            std::cout << "Password: " << registeredUser->getPassword() << std::endl;
            // Display other customer details
            std::cout << "--------------------------" << std::endl;
        }
    }
}

void AdminUser::modifyCustomerDetails(UserManager& userManager) {
    std::string username;
    std::cout << "Enter the username of the customer to modify: ";
    std::getline(std::cin, username);

    // Find the user in the user manager
    BaseUser* user = userManager.findUser(username);
    if (user) {
        // Check if the user is a registered customer
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            // Perform modifications on the customer's details
            // For example, modify username or password
            // ui.modifyCustomerDetails(registeredUser);
        }
        else {
            std::cout << "User is not a registered customer.\n";
        }
    }
    else {
        std::cout << "User not found.\n";
    }
}

void AdminUser::deleteInactiveCustomers(UserManager& userManager) {
    // Get all users from the user manager
    std::vector<BaseUser*> users = userManager.getAllUsers();

    // Iterate over the users and delete inactive customers
    for (auto it = users.begin(); it != users.end(); ) {
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(*it);
        if (registeredUser && !registeredUser->isLoggedIn()) {
            // Delete the customer from the user manager
            it = users.erase(it);
            delete registeredUser;
        }
        else {
            ++it;
        }
    }

    std::cout << "Inactive customers deleted.\n";
}
```

```cpp
void AdminUser::readFeedbacks(UserManager& userManager) {
    std::cout << "Feedbacks:\n";

    // Retrieve all users from the user manager
    std::vector<BaseUser*> users = userManager.getAllUsers();

    // Iterate over the users and read their feedbacks
    for (BaseUser* user : users) {
        // Check if the user is a registered customer
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            std::cout << "Username: " << registeredUser->getUsername() << std::endl;

            // Retrieve the favorite universities of the customer
            std::vector<University> favoriteUniversities = userManager.getFavoriteUniversities(registeredUser);

            // Display the feedbacks for each favorite university
            for (const University& university : favoriteUniversities) {
                std::cout << "University: " << university.getName() << std::endl;
                std::cout << "Feedbacks:\n";
                std::vector<std::string> feedbacks = university.getFeedbacks();
                for (const std::string& feedback : feedbacks) {
                    std::cout << "- " << feedback << std::endl;
                }
                std::cout << "-------------------------" << std::endl;
            }
        }
    }
}
```

```cpp
void AdminUser::replyFeedback(UserManager& userManager) {
    std::string username;
    std::cout << "Enter the username of the customer to reply to: ";
    std::getline(std::cin, username);

    // Find the user in the user manager
    BaseUser* user = userManager.findUser(username);
    if (user) {
        // Check if the user is a registered customer
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            // Retrieve the favorite universities of the customer
            std::vector<University> favoriteUniversities = userManager.getFavoriteUniversities(registeredUser);

            // Display the feedbacks for each favorite university
            for (University& university : favoriteUniversities) {
                std::cout << "University: " << university.getName() << std::endl;
                std::cout << "Feedbacks:\n";
                std::vector<std::string> feedbacks = university.getFeedbacks();

                for (const std::string& feedback : feedbacks) {
                    std::cout << "- " << feedback << std::endl;
                }

                // Prompt the admin to enter a reply for each feedback
                std::string reply;
                std::cout << "Enter a reply to the feedback: ";
                std::getline(std::cin, reply);

                // Store the admin's reply
                university.addReply(reply);

                std::cout << "Reply added.\n";
                std::cout << "-------------------------" << std::endl;
            }
        }
        else {
            std::cout << "User is not a registered customer.\n";
        }
    }
    else {
        std::cout << "User not found.\n";
    }
}
```

```cpp
void AdminUser::generateTopPreferredUniversitiesReport(UserManager& userManager) {
    std::cout << "Top 10 Preferred Universities Report:\n";

    // Retrieve all users from the user manager
    std::vector<BaseUser*> users = userManager.getAllUsers();

    // Map to store the count of each university
    std::unordered_map<std::string, int> universityCounts;

    // Iterate over the users and count the preferred universities
    for (BaseUser* user : users) {
        // Check if the user is a registered customer
        RegisteredUser* registeredUser = dynamic_cast<RegisteredUser*>(user);
        if (registeredUser) {
            // Retrieve the favorite universities of the customer
            std::vector<University> favoriteUniversities = userManager.getFavoriteUniversities(registeredUser);

            // Increment the count for each university
            for (const University& university : favoriteUniversities) {
                universityCounts[university.getName()]++;
            }
        }
    }

    // Sort the universities based on their counts in descending order
    std::vector<std::pair<std::string, int>> sortedUniversities(
        universityCounts.begin(),
        universityCounts.end()
    );
    std::sort(sortedUniversities.begin(), sortedUniversities.end(),
        [](const std::pair<std::string, int>& a, const std::pair<std::string, int>& b) {
            return a.second > b.second;
        }
    );

    // Display the top 10 preferred universities
    int count = 0;
    for (const auto& pair : sortedUniversities) {
        if (count >= 10) {
            break;
        }
        std::cout << "University: " << pair.first << std::endl;
        std::cout << "Count: " << pair.second << std::endl;
        std::cout << "--------------------------" << std::endl;
        count++;
    }
}

```

```cpp
void AdminUser::displayAdminMenu(UserInterface& ui, UserManager& userManager) {
    while (loggedIn) {
        std::cout << "\n\n";
        std::cout << "==============================================\n";
        std::cout << "=======  Welcome to Admin User Menu  ==========\n";
        std::cout << "==============================================\n";
        std::cout << "\n";

        std::cout << "  1. Display all customers' details\n";
        std::cout << "  2. Modify a customer's details\n";
        std::cout << "  3. Delete inactive customers\n";
        std::cout << "  4. Read feedbacks\n";
        std::cout << "  5. Reply to feedback\n";
        std::cout << "  6. Generate report for Top 10 preferred universities\n";
        std::cout << "  0. Back to Main Menu\n";

        std::cout << "\n";
        std::cout << "==============================================\n";
        std::cout << "Enter your choice: ";

        int choice;
        std::cin >> choice;
        std::cin.ignore(); // Consume the newline character

        switch (choice) {
        case 1:
            displayCustomersDetails(userManager);
            break;
        case 2:
            modifyCustomerDetails(userManager);
            break;
        case 3:
            deleteInactiveCustomers(userManager);
            break;
        case 4:
            readFeedbacks(userManager);
            break;
        case 5:
            replyFeedback(userManager);
            break;
        case 6:
            generateTopPreferredUniversitiesReport(userManager);
            break;
        case 0:
            ui.logoutUser();
            return;
        default:
            std::cout << "Invalid option. Please try again.\n";
            break;
```

```cpp
        case 0:
            ui.logoutUser();
            return;
        default:
            std::cout << "Invalid option. Please try again.\n";
            break;
        }
    }
}
```

Explanation

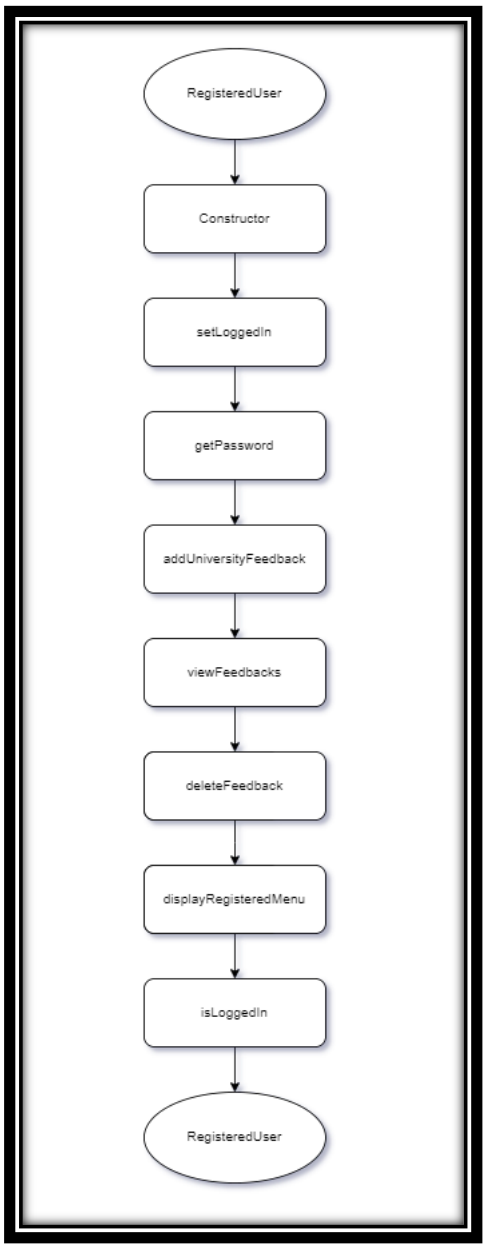
The `AdminUser` class represents an admin user and is derived from the `BaseUser` class. It has a constructor to initialize the username and password, and the `setLoggedIn` function to update the login status of the admin user. The class includes various member functions to perform admin-specific actions.

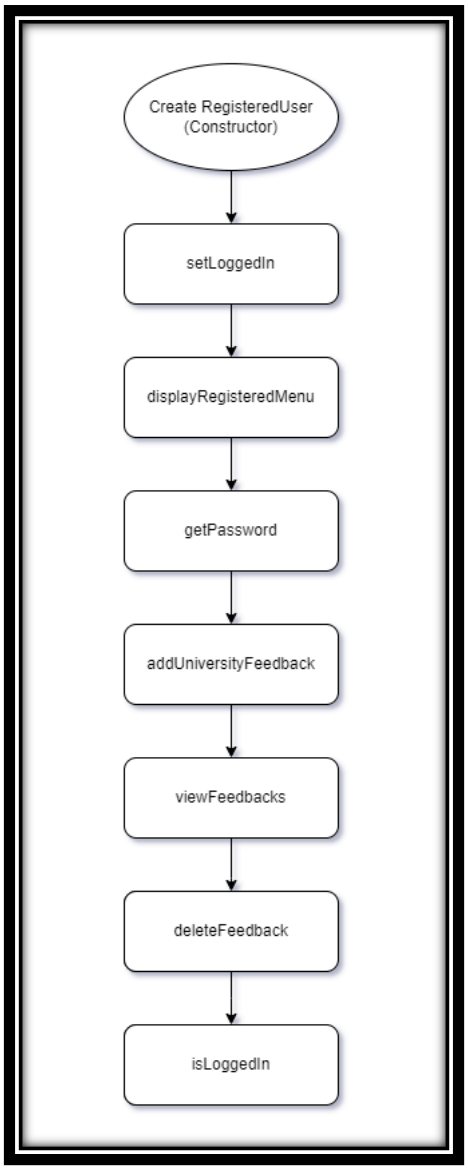
The `displayCustomersDetails` function retrieves all users from the `UserManager` and displays the details of registered customers. The `modifyCustomerDetails` function prompts the admin to enter a

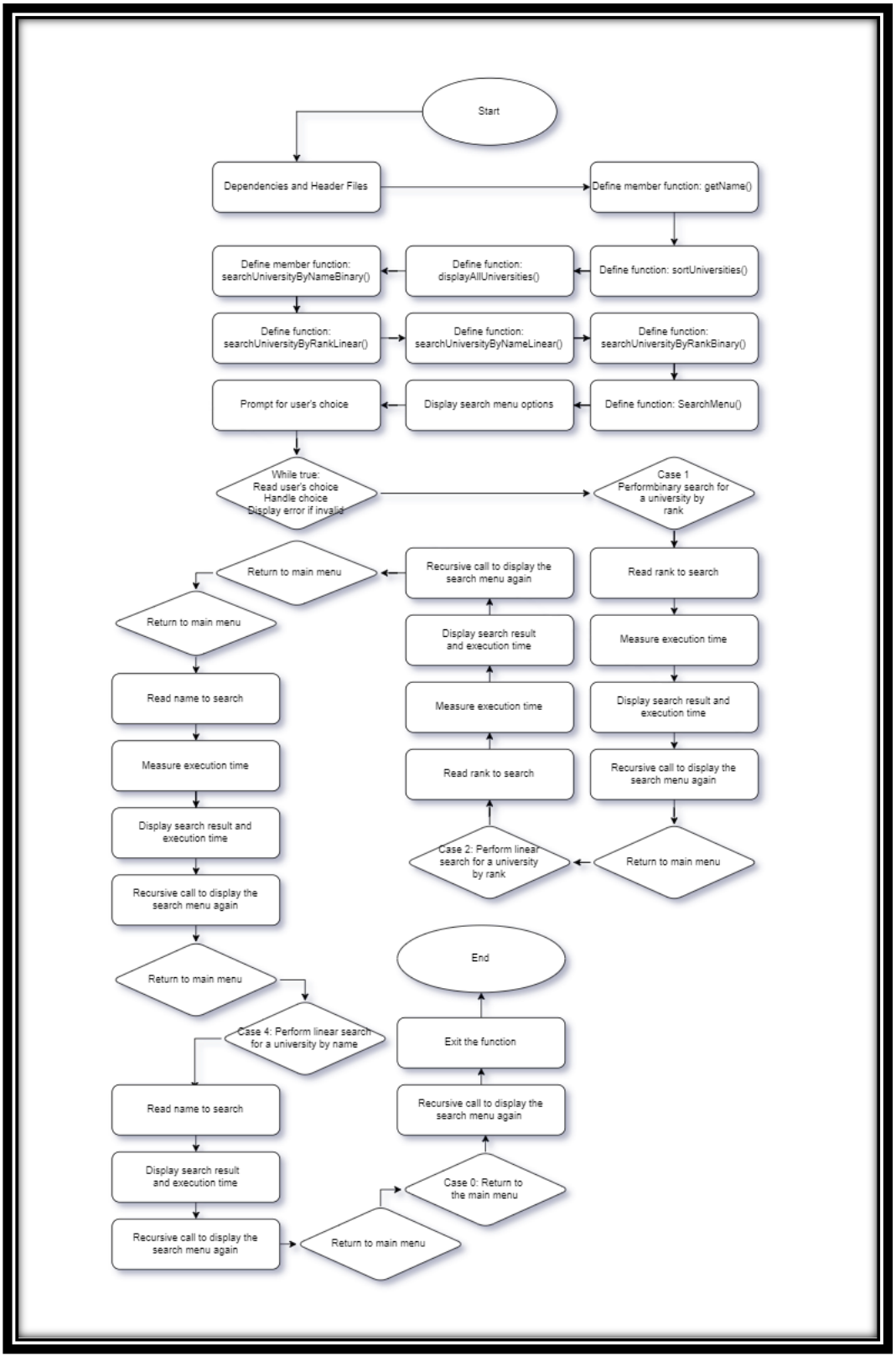
username and allows them to modify the details of a specific customer. The `deleteInactiveCustomers` function removes inactive customers from the `UserManager`. The `readFeedbacks` function retrieves all users and their favorite universities from the `UserManager`, and displays the feedbacks for each university. The `replyFeedback` function prompts the admin to enter a username and allows them to reply to the feedbacks of a specific customer. The `generateTopPreferredUniversitiesReport` function counts the number of times each university is preferred by customers, sorts them in descending order, and displays the top 10 preferred universities.

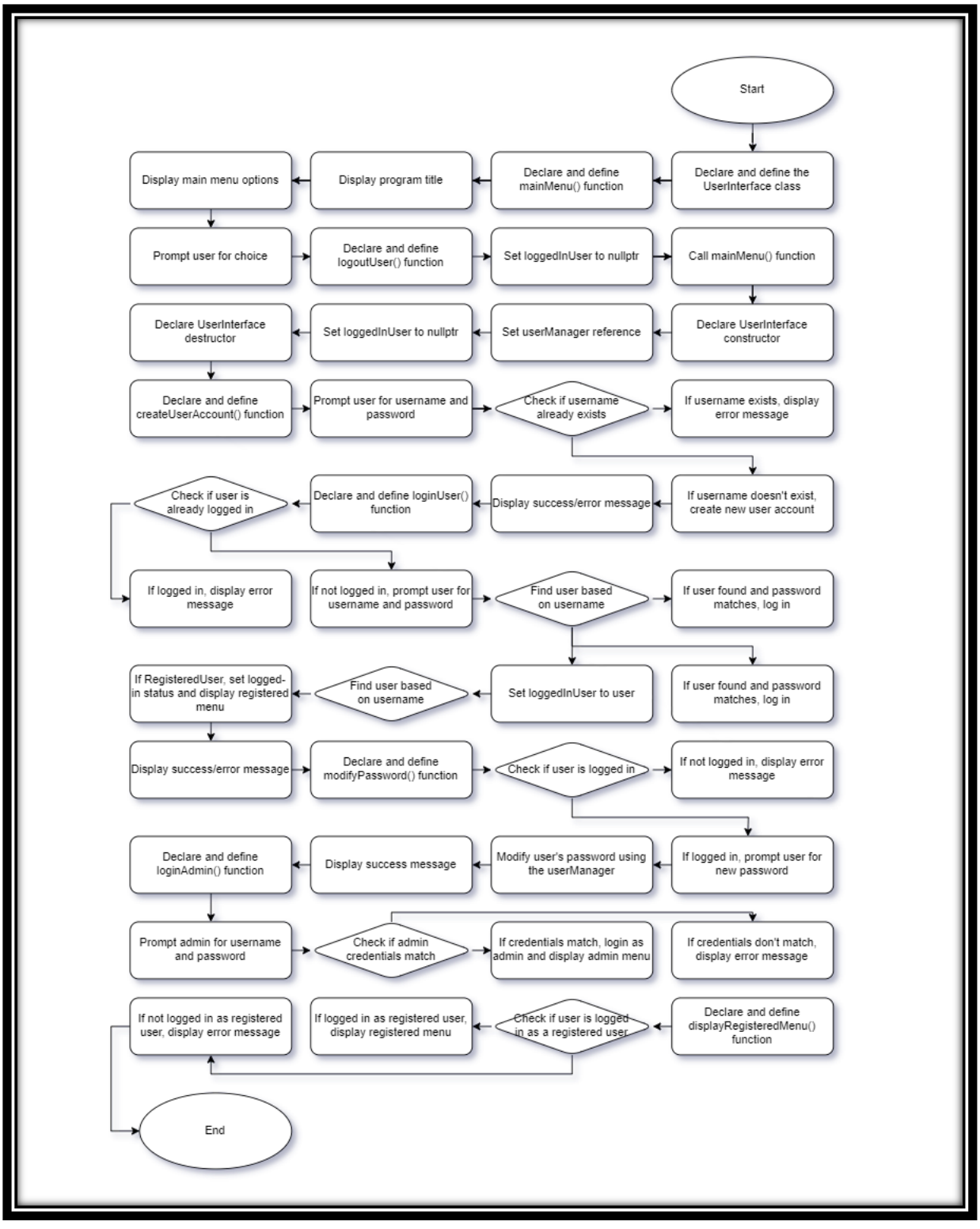
The `displayAdminMenu` function is responsible for displaying the admin menu and handling admin-specific actions. It takes references to the `UserInterface` and `UserManager` objects as parameters. The admin can choose an option from the menu, and the corresponding function will be called to perform the desired action. The admin can also choose to log out and return to the main menu.

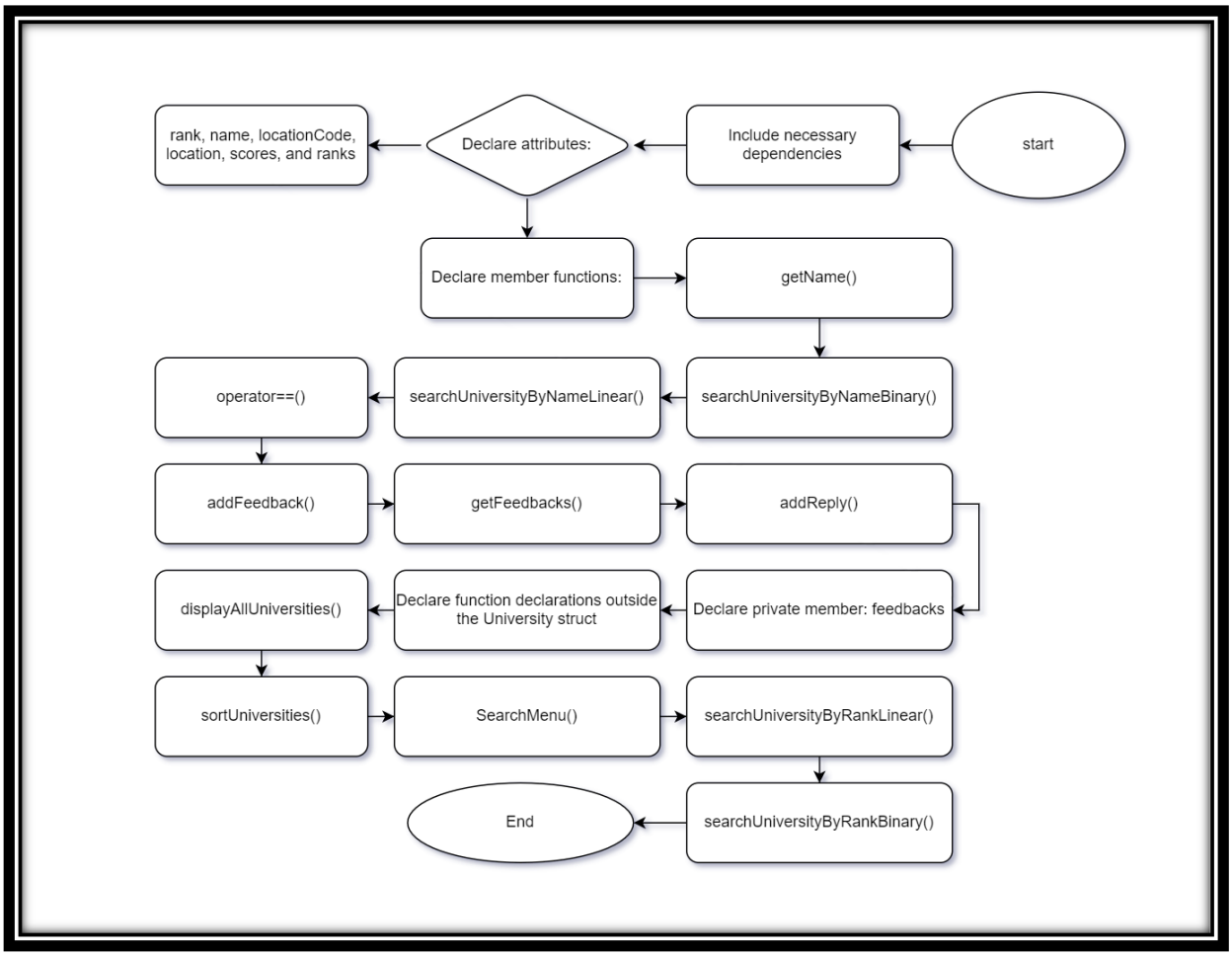
Flowcharts

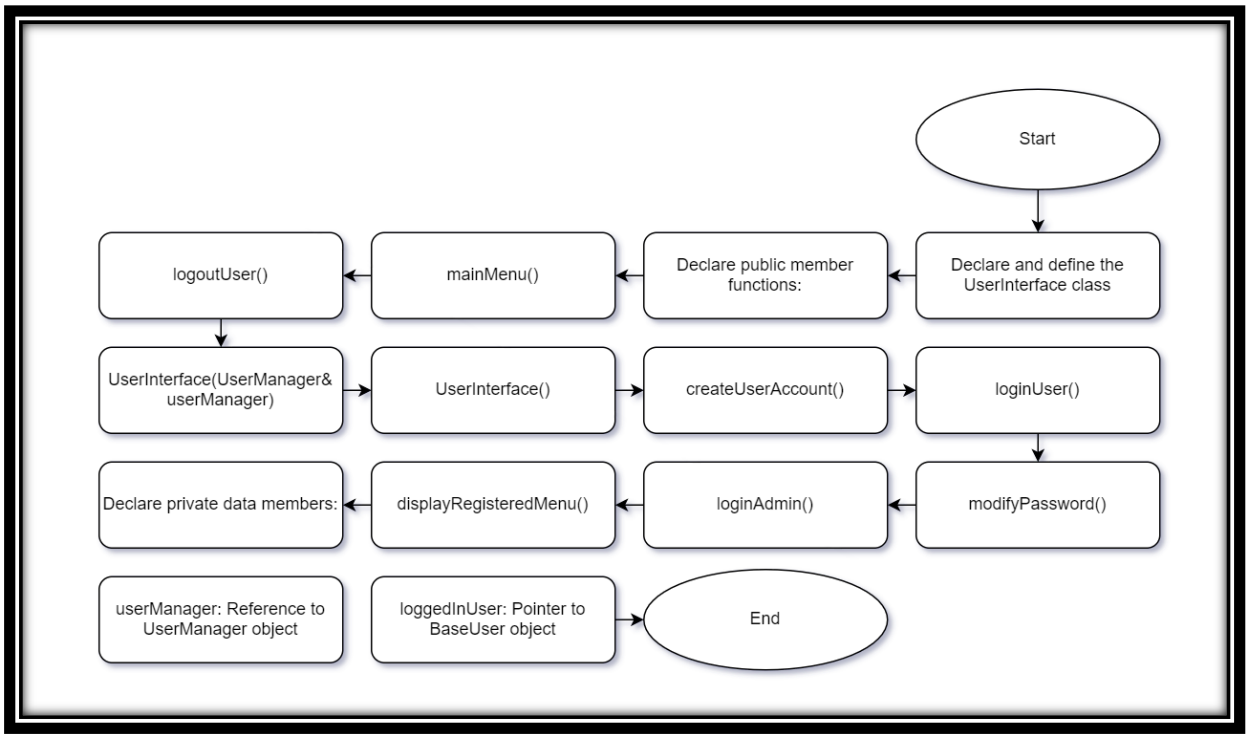

RegisteredUser
Constructor
setLoggedIn
getPassword
addUniversityFeedback
viewFeedbacks
deleteFeedback
displayRegisteredMenu
isLoggedIn
RegisteredUser

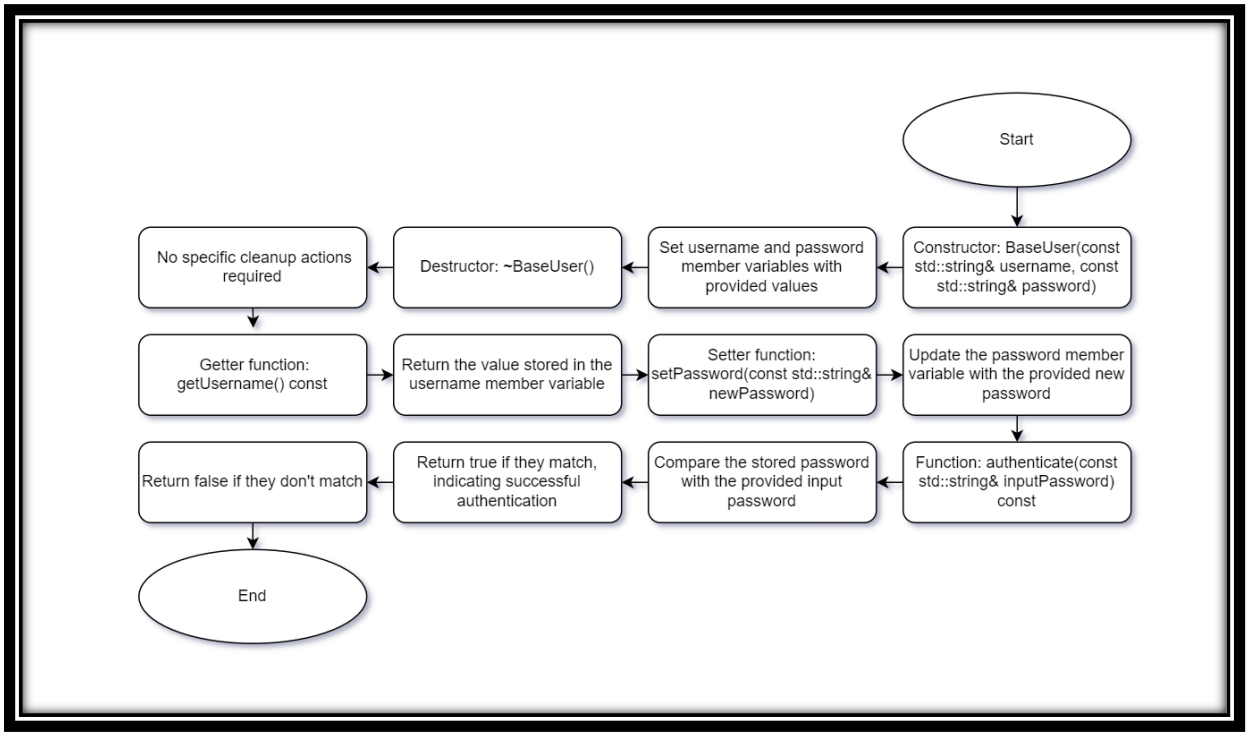

Create RegisteredUser
(Constructor)
setLoggedIn
displayRegisteredMenu
getPassword
addUniversityFeedback
viewFeedbacks
deleteFeedback
isLoggedIn

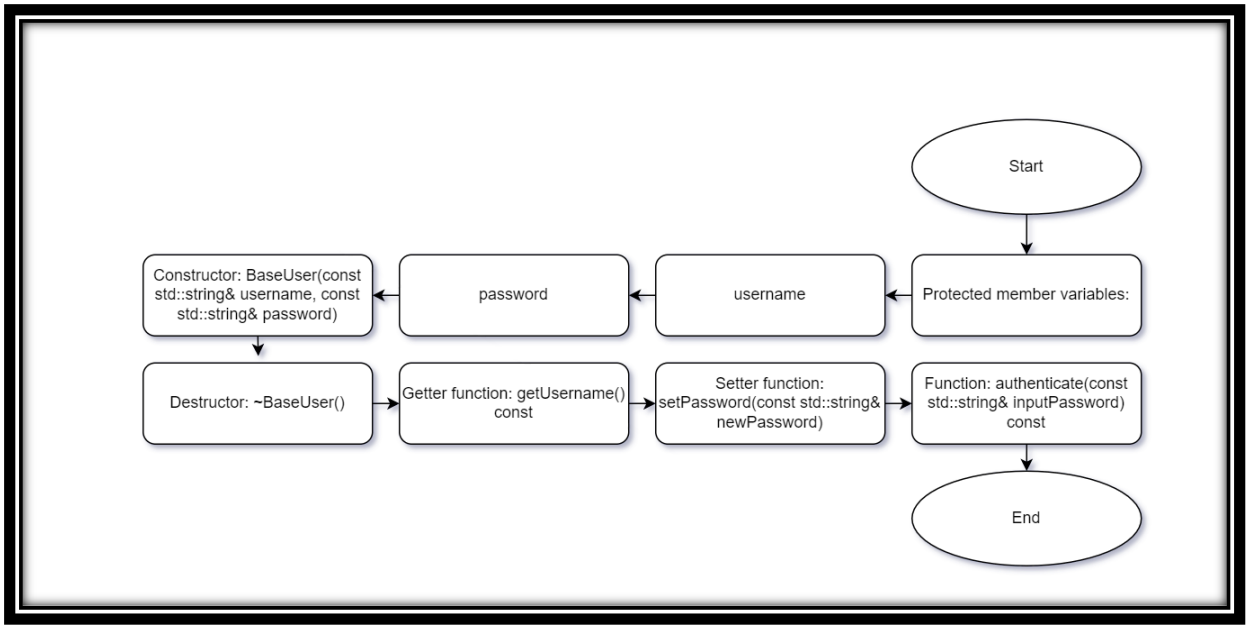

Start
Dependencies and Header Files
Define member function: getName()
Define member function: searchUniversityByNameBinary()
Define function: displayAllUniversities()
Define function: sortUniversities()
Define function: searchUniversityByRankLinear()
Define function: searchUniversityByNameLinear()
Define function: searchUniversityByRankBinary()
Prompt for user's choice
Display search menu options
Define function: SearchMenu()
While true:
Read user's choice
Handle choice
Display error if invalid
Case 1
Performbinary search for a university by rank
Return to main menu
Recursive call to display the search menu again
Read rank to search
Return to main menu
Display search result and execution time
Measure execution time
Read name to search
Measure execution time
Display search result and execution time
Measure execution time
Display search result and execution time
Read rank to search
Recursive call to display the search menu again
Recursive call to display the search menu again
Case 2: Perform linear search for a university by rank
Return to main menu
End
Return to main menu
Case 4: Perform linear search for a university by name
Exit the function
Read name to search
Recursive call to display the search menu again
Display search result and execution time
Case 0: Return to the main menu
Recursive call to display the search menu again
Return to main menu

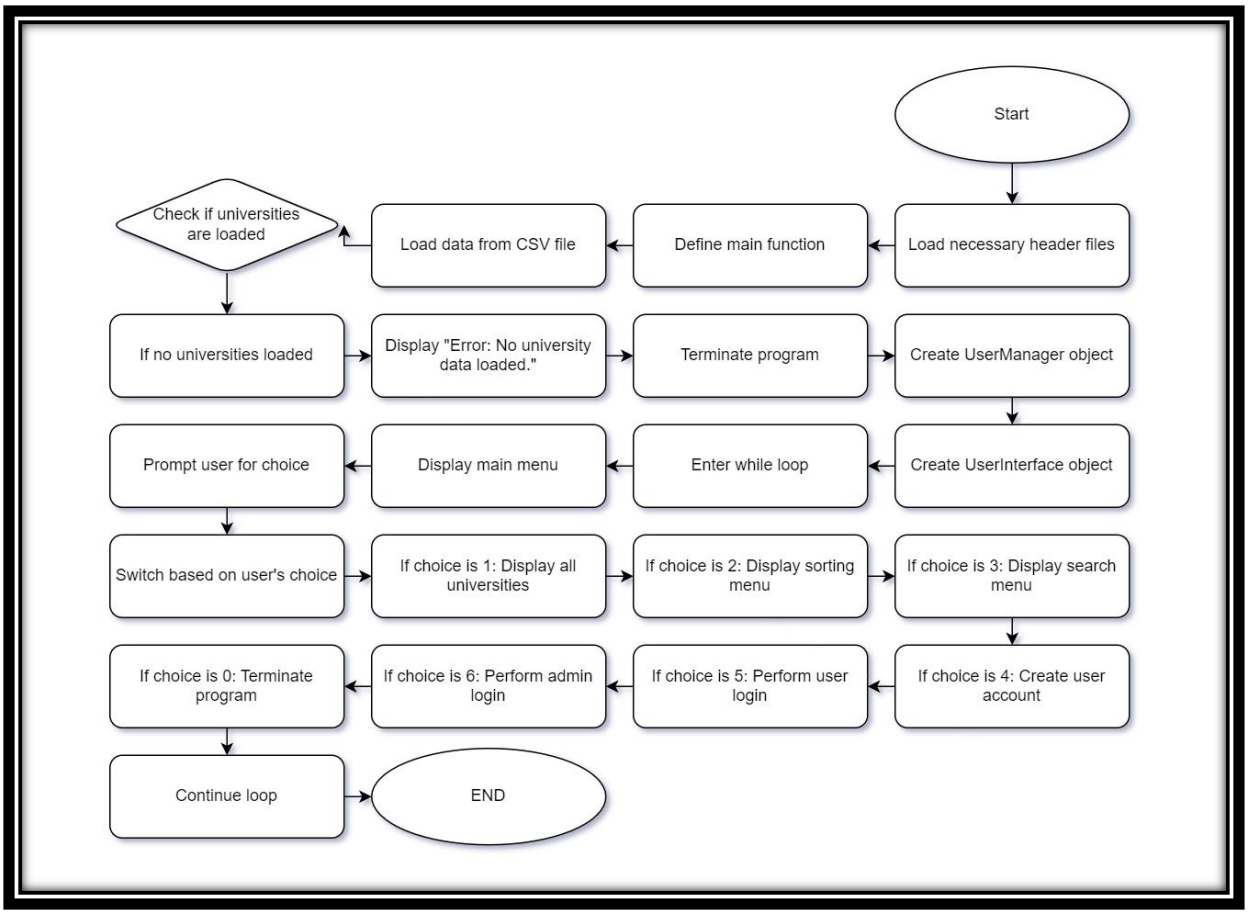

Start
Declare and define the UserInterface class
Declare and define mainMenu() function
Display program title
Display main menu options
Prompt user for choice
Declare and define logoutUser() function
Set loggedInUser to nullptr
Call mainMenu() function
Declare UserInterface constructor
Set userManager reference
Set loggedInUser to nullptr
Declare UserInterface destructor
Declare and define createUserAccount() function
Prompt user for username and password
Check if username already exists
If username exists, display error message
If username doesn't exist, create new user account
Display success/error message
Declare and define loginUser() function
Check if user is already logged in
If logged in, display error message
If not logged in, prompt user for username and password
Find user based on username
If user found and password matches, log in
Set loggedInUser to user
Find user based on username
If RegisteredUser, set logged-in status and display registered menu
If user found and password matches, log in
Display success/error message
Declare and define modifyPassword() function
Check if user is logged in
If not logged in, display error message
If logged in, prompt user for new password
Modify user's password using the userManager
Display success message
Declare and define loginAdmin() function
Prompt admin for username and password
Check if admin credentials match
If credentials match, login as admin and display admin menu
If credentials don't match, display error message
Declare and define displayRegisteredMenu() function
Check if user is logged in as a registered user
If logged in as registered user, display registered menu
If not logged in as registered user, display error message
End


start
Include necessary dependencies
Declare attributes:
rank, name, locationCode, location, scores, and ranks
Declare member functions:
getName()
searchUniversityByNameBinary()
searchUniversityByNameLinear()
operator==()
addFeedback()
getFeedbacks()
addReply()
Declare private member: feedbacks
Declare function declarations outside the University struct
displayAllUniversities()
sortUniversities()
SearchMenu()
searchUniversityByRankLinear()
searchUniversityByRankBinary()
End


Start
Declare and define the UserInterface class
Declare public member functions:
mainMenu()
logoutUser()
loginUser()
createUserAccount()
UserInterface()
UserInterface(UserManager& userManager)
modifyPassword()
loginAdmin()
displayRegisteredMenu()
Declare private data members:
loggedInUser: Pointer to BaseUser object
userManager: Reference to UserManager object
End


Start
Constructor: BaseUser(const std::string& username, const std::string& password)
Set username and password member variables with provided values
Destructor: ~BaseUser()
No specific cleanup actions required
Getter function: getUsername() const
Return the value stored in the username member variable
Setter function: setPassword(const std::string& newPassword)
Update the password member variable with the provided new password
Function: authenticate(const std::string& inputPassword) const
Compare the stored password with the provided input password
Return true if they match, indicating successful authentication
Return false if they don't match
End


Start
Protected member variables:
username
password
Constructor: BaseUser(const std::string& username, const std::string& password)
Function: authenticate(const std::string& inputPassword) const
Setter function: setPassword(const std::string& newPassword)
Getter function: getUsername() const
Destructor: ~BaseUser()
End


Start
Load necessary header files
Define main function
Load data from CSV file
Check if universities are loaded
If no universities loaded
Display "Error: No university data loaded."
Terminate program
Create UserManager object
Create UserInterface object
Enter while loop
Display main menu
Prompt user for choice
Switch based on user's choice
If choice is 1: Display all universities
If choice is 2: Display sorting menu
If choice is 3: Display search menu
If choice is 4: Create user account
If choice is 5: Perform user login
If choice is 6: Perform admin login
If choice is 0: Terminate program
Continue loop
END

Code Outputs

```
================================================
=====  Welcome to Hamada QSRanking Program =====
================================================

=================== Main Menu ==================
  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Register as customer
  5. Login as registered customer
  6. Login as Admin
  0. Exit program

================================================
Enter your choice: 1
```

```
IFR Score: 5.2
IFR Rank:
ISR Score: 2.8
ISR Rank: 0
IRN Score: 51.1
IRN Rank: 0
GER Score: 0
GER Rank: 0
Score Scaled: 0
-------------------------
Name: University of Craiova
Location: Romania
Rank: 1421
AR Score: 3.3
AR Rank: 501
ER Score: 1.8
ER Rank: 501
FSR Score: 2.7
FSR Rank: 0
CPF Score: 2
CPF Rank: 0
IFR Score: 0
IFR Rank:
ISR Score: 0
ISR Rank: 0
IRN Score: 7.5
IRN Rank: 0
GER Score: 8.7
GER Rank: 0
Score Scaled: 0
-------------------------
Name: University of Oradea
Location: Romania
Rank: 1422
AR Score: 4
AR Rank: 501
ER Score: 2.1
ER Rank: 501
FSR Score: 3.3
FSR Rank: 0
CPF Score: 1.5
CPF Rank: 0
IFR Score: 1.7
IFR Rank:
ISR Score: 7.9
ISR Rank: 0
IRN Score: 16.6
IRN Rank: 0
GER Score: 9.7
GER Rank: 0
Score Scaled: 0
-------------------------
```

```
================================================
=====  Welcome to Hamada QSRanking Program =====
================================================

=================== Main Menu ==================
  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Register as customer
  5. Login as registered customer
  6. Login as Admin
  0. Exit program

================================================
Enter your choice: 2


======================
=====  Sort Menu =====
======================

  1. Sort universities By Rank (Insertion Sort)
  2. Sort universities By Rank (Bubble Sort)
  3. Sort universities By Name (Insertion Sort)
  4. Sort universities By Name (Bubble Sort)
   0. Main Menu

==================
Enter your choice: 1
```

```
IRN Rank: 0
GER Score: 8.7
GER Rank: 0
Score Scaled: 0
-------------------------
Name: University of Oradea
Location: Romania
Rank: 1422
AR Score: 4
AR Rank: 501
ER Score: 2.1
ER Rank: 501
FSR Score: 3.3
FSR Rank: 0
CPF Score: 1.5
CPF Rank: 0
IFR Score: 1.7
IFR Rank:
ISR Score: 7.9
ISR Rank: 0
IRN Score: 16.6
IRN Rank: 0
GER Score: 9.7
GER Rank: 0
Score Scaled: 0
-------------------------
Sorting time (Insertion Sort by Rank): 0.0032104 seconds


======================
=====  Sort Menu =====
======================

  1. Sort universities By Rank (Insertion Sort)
  2. Sort universities By Rank (Bubble Sort)
  3. Sort universities By Name (Insertion Sort)
  4. Sort universities By Name (Bubble Sort)
   0. Main Menu

==================
Enter your choice:
```

```
IRN Rank: 0
GER Score: 8.7
GER Rank: 0
Score Scaled: 0
-------------------------
Name: University of Oradea
Location: Romania
Rank: 1422
AR Score: 4
AR Rank: 501
ER Score: 2.1
ER Rank: 501
FSR Score: 3.3
FSR Rank: 0
CPF Score: 1.5
CPF Rank: 0
IFR Score: 1.7
IFR Rank:
ISR Score: 7.9
ISR Rank: 0
IRN Score: 16.6
IRN Rank: 0
GER Score: 9.7
GER Rank: 0
Score Scaled: 0
-------------------------
Sorting time (Bubble Sort by Rank): 0.0061982 seconds


======================
=====  Sort Menu =====
======================

  1. Sort universities By Rank (Insertion Sort)
  2. Sort universities By Rank (Bubble Sort)
  3. Sort universities By Name (Insertion Sort)
  4. Sort universities By Name (Bubble Sort)
   0. Main Menu

==================
Enter your choice:
```

```
GER Rank: 289
Score Scaled: 56.6
-------------------------
Name: Óbuda University
Location: Hungary
Rank: 1260
AR Score: 3.1
AR Rank: 501
ER Score: 3.6
ER Rank: 501
FSR Score: 2
FSR Rank: 0
CPF Score: 7.5
CPF Rank: 0
IFR Score: 8.7
IFR Rank:
ISR Score: 7.3
ISR Rank: 0
IRN Score: 27.9
IRN Rank: 0
GER Score: 19.1
GER Rank: 0
Score Scaled: 0
-------------------------
Sorting time (Insertion Sort by Name): 0.128375 seconds


======================
=====  Sort Menu =====
======================

  1. Sort universities By Rank (Insertion Sort)
  2. Sort universities By Rank (Bubble Sort)
  3. Sort universities By Name (Insertion Sort)
  4. Sort universities By Name (Bubble Sort)
   0. Main Menu

==================
Enter your choice:
```

```
Score Scaled: 56.6
-------------------------
Name: Óbuda University
Location: Hungary
Rank: 1260
AR Score: 3.1
AR Rank: 501
ER Score: 3.6
ER Rank: 501
FSR Score: 2
FSR Rank: 0
CPF Score: 7.5
CPF Rank: 0
IFR Score: 8.7
IFR Rank:
ISR Score: 7.3
ISR Rank: 0
IRN Score: 27.9
IRN Rank: 0
GER Score: 19.1
GER Rank: 0
Score Scaled: 0
-------------------------
Sorting time (Bubble Sort by Name): 0.038548 seconds


======================
=====  Sort Menu =====
======================

  1. Sort universities By Rank (Insertion Sort)
  2. Sort universities By Rank (Bubble Sort)
  3. Sort universities By Name (Insertion Sort)
  4. Sort universities By Name (Bubble Sort)
   0. Main Menu

==================
Enter your choice:
```

```
========================
=====  Search Menu =====
========================

  1. Search universities By Rank (Binary Search)
  2. Search universities By Rank (Linear Search)
  3. Search universities By Name (Binary Search)
  4. Search universities By Name (Linear Search)
  0. Main Menu

========================
Enter your choice: 1
Enter Rank to Search: 3
University found:
Name: Stanford University
Location: United States
Rank: 3
Execution Time: 0.0059 ms
```

```
========================
=====  Search Menu =====
========================

  1. Search universities By Rank (Binary Search)
  2. Search universities By Rank (Linear Search)
  3. Search universities By Name (Binary Search)
  4. Search universities By Name (Linear Search)
  0. Main Menu

========================
Enter your choice: 4
Enter Name to Search: Stanford University
University found:
Name: Stanford University
Location: United States
Rank: 3
Execution Time: 0.0454 ms
```

```
========================
=====  Search Menu =====
========================

  1. Search universities By Rank (Binary Search)
  2. Search universities By Rank (Linear Search)
  3. Search universities By Name (Binary Search)
  4. Search universities By Name (Linear Search)
  0. Main Menu

========================
Enter your choice: 3
Enter Name to Search: Stanford University
University found:
Name: Stanford University
Location: United States
Rank: 3
Execution Time: 18.2098 ms
```

```
========================
=====  Search Menu =====
========================

  1. Search universities By Rank (Binary Search)
  2. Search universities By Rank (Linear Search)
  3. Search universities By Name (Binary Search)
  4. Search universities By Name (Linear Search)
  0. Main Menu

========================
Enter your choice: 2
Enter Rank to Search: 3
University found:
Name: Stanford University
Location: United States
Rank: 3
Execution Time: 0.0014 ms
```

```
==========================================
=====  Welcome to Hamada QSRanking Program =====
==========================================

=================== Main Menu ==================
  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Register as customer
  5. Login as registered customer
  6. Login as Admin
  0. Exit program

==========================================
Enter your choice: 4
Enter username: Hamada
Enter password: 123
User account created successfully.


==========================================
=====  Welcome to Hamada QSRanking Program =====
==========================================

=================== Main Menu ==================
  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Register as customer
  5. Login as registered customer
  6. Login as Admin
  0. Exit program

==========================================
Enter your choice: 5
Enter username: Hamada
Enter password: 123
Login successful.
```

```
==========================================
==========  Welcome to Customer Menu =========
==========================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==========================================
Enter your choice: 4
Enter university name to add to favorites: Stanford University
University added to favorites.
```

```
==========================================
==========  Welcome to Customer Menu =========
==========================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==========================================
Enter your choice: 5
Favorite universities:
Name: Stanford University
------------------------
```

```
==============================================
============  Welcome to Customer Menu =========
==============================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==============================================
Enter your choice: 6
Favorite universities:
1. Stanford University
Enter the index of the university to delete: 1
University deleted from favorites.


==============================================
============  Welcome to Customer Menu =========
==============================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==============================================
Enter your choice: 5
No favorite universities found.
```

```
==============================================
============  Welcome to Customer Menu =========
==============================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==============================================
Enter your choice: 8
Feedbacks:
1. University: Stanford University
   Feedback: This is a good university :)


==============================================
============  Welcome to Customer Menu =========
==============================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==============================================
Enter your choice: 9
Enter the index of the feedback to delete: 1
Feedback deleted.
```

```
==============================================
==========  Welcome to Customer Menu =========
==============================================

  1. Display all universities' information
  2. Sort universities
  3. Search university details
  4. Add universities to favorite
  5. View favorite universities
  6. Delete favorite universities
  7. Add Feedback on University
  8. View Feedbacks
  9. Delete Feedback
  0. Back to Main Menu

==============================================
Enter your choice: 7
Enter university name to add feedback: Stanford University
Enter your feedback: This is a good university :)
Feedback added to the university.
```

```
================================================
====  Welcome to Hamada QSRanking Program =====
================================================

================== Main Menu ==================
 1. Display all universities' information
 2. Sort universities
 3. Search university details
 4. Register as customer
 5. Login as registered customer
 6. Login as Admin
 0. Exit program

================================================
nter your choice: 6
nter your login details:
sername: admin
assword: 123
ogin successful.
dmin Menu:


================================================
======  Welcome to Admin User Menu  ==========
================================================

 1. Display all customers' details
 2. Modify a customer's details
 3. Delete inactive customers
 4. Read feedbacks
 5. Reply to feedback
 6. Generate report for Top 10 preferred universities
 0. Back to Main Menu

================================================
nter your choice:
```

```
================================================
======  Welcome to Admin User Menu  ==========
================================================

 1. Display all customers' details
 2. Modify a customer's details
 3. Delete inactive customers
 4. Read feedbacks
 5. Reply to feedback
 6. Generate report for Top 10 preferred universities
 0. Back to Main Menu

================================================
nter your choice: 1
ustomers' details:
sername: Hamada
assword: 123
------------------------
```

Comparing between Linear search & Binary Search

SEARCH TYPE	SEARCH METHOD	EXECUTION TIME (IN MS)	PROS	CONS
SEARCH UNIVERSITY BY RANK	Binary	0.0059	Efficient, works on sorted data	Complexity of implementation, requires data to be sorted
SEARCH UNIVERSITY BY RANK	Linear	0.0014	Simple implementation, works on unsorted data	Inefficient for large data sets, time taken increases linearly with data size
SEARCH UNIVERSITY BY NAME	Binary	18.2098	Efficient, works on sorted data	Complexity of implementation, requires data to be sorted
SEARCH UNIVERSITY BY NAME	Linear	0.0454	Simple implementation, works on unsorted data	Inefficient for large data sets, time taken increases linearly with data size

Comparing between Bubble Sort & Insertion Sort

SORT TYPE	SORT METHOD	EXECUTION TIME (IN MS)	PROS	CONS
SORT UNIVERSITY BY RANK	Insertion	0.0032104	Efficient for small data sets, adaptive	Inefficient for large data sets, slower than more advanced sorting algorithms
SORT UNIVERSITY BY RANK	Bubble	0.0061982	Simplicity, works well on small or nearly sorted data sets	Inefficient for large data sets, slower than more advanced sorting algorithms
SORT UNIVERSITY BY NAME	Insertion	0.128375	Efficient for small data sets, adaptive	Inefficient for large data sets, slower than more advanced sorting algorithms
SORT UNIVERSITY BY NAME	Bubble	0.038548	Simplicity, works well on small or nearly sorted data sets	Inefficient for large data sets, slower than more advanced sorting algorithms

Appendix
Workload Matrix

Team Member	***Percentage of Work Done***
MOHAMED KHAIRY MOHAMED ABDELRAOUF	65% (documentation with everything, registered user with all functionality , admin user, Login, logout, Binary Search , Data Seeder, University, Base User, UserManager, etc)
Valentino jaya	32% (normal user bubble sort, insertion sort, main menu, Display All University ,etc)
Ananda Jotty	0% (sends me code not working and didn't update me)
Hashem Mohammed Husain Baageel	3% (Tried to do his part but sends r incomplete and not compatible code)

References

- Bubble sort:
    - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.
    - Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
- Dynamic array:
    - Weiss, M. A. (2014). Data structures and algorithm analysis in C++ (4th ed.). Pearson.
    - Malik, D. S. (2010). C++ programming: From problem analysis to program design (5th ed.). Course Technology.
- Insertion sort:
    - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.
    - Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
- Linked lists:
    - Weiss, M. A. (2014). Data structures and algorithm analysis in C++ (4th ed.). Pearson.
    - Malik, D. S. (2010). C++ programming: From problem analysis to program design (5th ed.). Course Technology.