Practical Assignment 4

Continuing implementation of a management system for TV series

1. General Information

The Practical Assignment 4 applies concepts of Object-Oriented Programming and aims to implement classes based in graphs and hash tables.

This assignment must be done autonomously, by each group, until the defined deadline. It is acceptable to browse the different available information sources, but the submitted code must solely be by the group members. Any detected copied work will be penalised accordingly. The inability to explain the submitted code by any of the group members will also be penalised.

The submission deadline (in Moodle) is **April 28th at 23:59h**.

2. Concept

A TV series platform intends to obtain a system to manage the series watched by different users.

3. Assignment implementation

The compressed folder **`EDA_2024_MTP4.zip`** contains the files needed to perform the assignment, namely:

- `TVseries.hpp`: definition of the classes to represent the system (`User`, `TVSeries`, `TVSeriesManagement`, `UserManagement`, `TVSeriesManagementList`, `UserManagementList`, `NodeUser`, `UserManagementTree`, `HashTable` and `UserManagementGraph`).
- `TVseries.cpp`: implementation of the methods of the classes defined in `TVseries.hpp`.
- `series_testTP4.cpp`: includes the main program that invokes and performs basic tests to the implemented functions.
- `.txt`: text files to test the implemented functions.

Important remarks:

1. **The `TVseries.cpp` file is the only file that must be changed, and all that is required is to include the implementation of each function in the code submission through CodeRunner, in Moodle.**
2. **Each attribute and method of the classes defined has additional details next to each of them in `TVseries.hpp`.**
3. **In yellow, are the changes made to the classes in relation to the third assignment.**

The file `TVseries.hpp` contains the classes:

1. `User`,
2. `TVSeries`,
3. `TVSeriesManagement`,
4. `UserManagement`,
5. `TVSeriesManagementList`,
6. `UserManagementList`,
7. `NodeUser`,
8. `UserManagementTree`,
9. `HashTable`,
10. `UserManagementGraph`.

The first characterises each TV series, the second each user, the third allows the management of the series in the platform, and the fourth defines the users registered in the platform. The fifth and sixth do the same as the second and third, respectively, but using lists. The seventh represents a BST tree node of the tree defined in the eighth class with all the registered users. The ninth class defines a hash table to store the statistics per country and, finally, the tenth defines a graph with all the registered users.

Class `User`

The objects of the class `User` have the following attributes:

1) usrname (`username`)
2) user's name (`name`)
3) *string* with the user's country (`country`)
4) string vector with the user's favourite TV series genres (`favoriteGenres`)
5) vector with the series watched by the user (`watchedSeries`)
6) integer vector with the ratings given by the user (`ratings`); i.e., the rating of the series at index 0 in `watchedSeries` is at index 0
7) integer vector with the number of episodes watched of each TV series (`episodesWatched`); i.e., the number of episodes watched of the series at index 0 in `watchedSeries` is at index 0
8) wish queue with the series we want to see, in order of arrival (`wishSeries`)
9) integer to store the length to a certain node (`length`)

Class `TVSeries`

The objects of the class `TVSeries` have the following attributes:

1) series title (`title`)
2) current number of seasons (`numberOfSeasons`)
3) integer vector with the current number of episodes of each season (`episodesPerSeason`); i.e., the number of episodes of season 1 is at index 0
4) series genre (`genre`)
5) float with the series rating (`rating`)

6) boolean flag to indicate if the series is or not complete (`finished`)

Class `TVSeriesManagement`

The objects of the class `TVSeriesManagement` have a pointer vector to objects of the class `TVSeries`, which represents all TV series available in the platform.

Class `TVSeriesManagementList`

The objects of the `TVSeriesManagementList` class have a linked list of pointers to objects of the `TVSeries` class, representing all series available on the platform.

Class `UserManagement`

The objects of the class `UserManagement` have a pointer vector to objects of the class `User`, which represents all the registered users of the platform.

Class `UserManagementList`

The objects of the `UserManagementList` class have a linked list of pointers to objects of the `User` class, representing all users registered on the platform.

Class `NodeUser`

The objects of the `NodeUser` class are nodes of the tree(s) of the `UserManagementTree` class. They have a pointer to an object of the `User` class representing a user registered on the platform, and two more pointers to the nodes on the left and right in the tree, respectively.

Class `UserManagementTree`

Objects of the `UserManagementTree` class have a BST tree of pointers to objects of the `User` class, representing all users registered on the platform.

Class `HashTable`

The objects of the `HashTable` class define a hash table to represente the statistics per country, and have the following attributes:

1) size of the hash table (`tableSize`)
2) pointer vector to elements of type `CountryStats`, which represents the hash table (`table`)
    a. `CountryStats` is a *struct* with the fields `country`, `nUsers`, `nTVSeries`, `averageTVseries` e `nGenre` which correspond, respectively, to the country name, number of users of that country, number of series watched in the country, average value of the number of distinct series watched in the country, and a vector with the number of series watched per genre in that country.
3) effective number of elements in the hash table (`totalCountryStats`)

Class `UserManagementGraph`

The objects of the `UserManagementGraph` class represent the network of interactions between the users (who follows who), and have the following attributes:

1) total number of nodes in the graph (`totalUsers`)
2) pointer vector to objects of the class `User`, which identifies the users included in the graph, which means the graph nodes (`userNodes`)
3) vector of lists of pointers to objects of the class `User`, which represents the adjacency list that identifies the graph edges, i.e. who follows who (`network`)

The functions to be implemented in this assignment are methods of each class.

Class `UserManagementGraph`

1. `vector<User*>` **`mostFollowing`**`();`
   Determines which user(s) follow the most users. Returns a pointer vector to that(those) user(s), i.e., the vector will have more than one element in case more than one user follows the same amount of people.

2. `TVSeries*` **`followingMostWatchedSeries`**`(User *userPtr);`
   Determines which series is(are) the most watched among the users a certain user (`userPtr`*) follows, considering the number of episodes watched of each series by each of those users. If two or more series have equal totals of episodes watched, the series watched by most users is selected; in case this criterion still results in a tie, the series with higher alphabetical order is selected. Verifies if the node exists in the graph and, if no error occurs, returns* `NULL`*.*

   HINTS
   To verify if a node is in the graph, you can use the method `userNodePosition()`.

3. `int` **`shortestPaths`**`(User *userSrc, User *userDst);`
   Finds the shortest distance between two nodes of the graph (`userSrc` → `userDst`*), using the Dijkstra algorithm. Verifies if the nodes exist in the graph. Returns the distance calculated, -1 if na error occurs or -2 if there is no path between the nodes.*

   HINTS
   To use the Dijkstra algorithm, apply a *min-heap* with the comparison function `CompareP`. Remind the `length` attribute and the methods `setLength()` and `getLength()` of `User`.

Class `HashTable`

4. `int` **`insertCountryStats`**`(CountryStats &countryS);`
   Insert a new element (`countryS`*) to the hash table and return the index of the inserted element or -1 if the element already exists or an error occurs. The key of the elemento to be inserted corresponds to the struct field* `country`*. Updates the value of the effective number of elements in the hash table. If there is a collision, the new element can be inserted either in a free position or in a position from where an element was previously removed (index key equal to "apagado").*

5. `int` **`importFromVector`**`(UserManagement &userManager);`
   Build the hash table based in the users' vector from the object `userManager`*. Creates a new element of type* `CountryStats` *per each distinct country of the users and calculates the value of each of its fields. Returns -1 if an error occurs or 0 if successful.*

Note: The input files and testcases which will be used to evaluate the submitted functions might contain different content and include critical cases, such as invalid function parameters. Thus, it is your own responsibility to ensure that the function parameters are properly tested to only be considered if valid.

4. Testing the function library

The library can be tested by executing the program `series_testTP4`. There is a test per each implemented function which assesses if that function has the expected behaviour. Nevertheless, the tests are not extensive and should be considered as an indicator of an apparent accurate implementation of the expected functionality.

If all functions pass the included testcases, the program `series_testTP4`, when executed, shall present the following result:

```
INICIO DOS TESTES


...verifica_mostFollowing: (Grafo vazio) - retorno é um vetor vazio (ok)

...verifica_mostFollowing: (Grafo preenchido) Tamanho do vetor (=4) e' o esperado (ok)

...verifica_mostFollowing: (Grafo preenchido) Os elementos  (=john_doe - mia_davis - rodrigo8 - emily_c) são os esperados (ok)

OK: verifica_mostFollowing passou


...verifica_followingMostWatchedSeries: (User não existe) retorno é Null (ok)

...verifica_followingMostWatchedSeries: (User existe),  A serie (=The Office) e' o esperado (ok)

...verifica_followingMostWatchedSeries: (User existe-empate entre series),  A serie (=Stranger Things) e' o esperado (ok)

OK: verifica_followingMostWatchedSeries passou


...verifica_shortestPaths: (Nó não existe) o retorno é o esperado (=-1)  (ok)

...verifica_shortestPaths: (Nó existe), Distancia (=3) (ok)

...verifica_shortestPaths: (Caminho não existe), Retorno (=-2) (ok)

OK: verifica_shortestPaths passou


...verifica_insertCountryStats: (1º Insersão 'Portugal' numa tabela de tamanho 11) campo onde foi inserido (=6) (ok)

...verifica_insertCountryStats: (Nó existe), campo onde a Alemanha foi inserido (=10) (ok)

OK: verifica_insertCountryStats passou


...verifica_importFromVector: (Importar o vetor) numeros de elementos inseridos (=7) (ok)

...verifica_importFromVector: (Importar o vetor) Portugal foi encontrado (ok)

...verifica_importFromVector: (Importar o vetor) numero de utilizadores em Portugal (=13) (ok)

...verifica_importFromVector: (Importar o vetor) numero de series vistas em Portugal (=17) (ok)

...verifica_importFromVector: (Importar o vetor) média de series vistas por utilizador em Portugal (=2.23077) (ok)
```

```
...verifica_importFromVector: (Importar o vetor) numero de series de Action vistas em Portugal (=2) (ok)

OK: verifica_importFromVector passou


FIM DOS TESTES: Todos os testes passaram
```

5. Development tools

Using an IDE or Visual Studio Code for the assignment development is recommended as it allows for a more effective debugging. Information on the usage of Visual Studio Code can be found in a short tutorial in Moodle.

It is possible to implement the functions requested in this assignment directly in CodeRunner, but it is advisable to check the files provided, so that the assignment context can be well understood.

6. Evaluation

The classification of this assessment is given by the evaluation of the implementation submitted by the students (automatically calculated in Moodle) and by assessing the capacity of the students to explain their work. The final classification of the assessment (MTP4) is given by:

$$MTP4 = Implementation \times Oral\ assessment$$

The classification of the implementation is mainly determined by additional automatic testcases (e.g., using larger test files). If the submitted implementation does not compile, this component will be 0%.

The oral assessment will be divided into 4 levels: 100% – masters the code; 75% – some flaws; 40% – several flaws detected in the explanation; 0% – demonstrates serious gaps.

7. Submission

The submission is solely possible through Moodle and up until the deadline mentioned at the beginning of this document. The submission of the implemented functions must be done in CodeRunner, in the dedicated areas in Moodle.