CS240: Data Structures
Fall Semester – Academic Year 2023-2024
Instructor: Panagiota Fatourou
Programming Assignment – 1st Phase

Assignment Due: Monday, 20 November 2023, 23:59
Submission Method: Through the turnin program. Information on using turnin on the course website (https://www.csd.uoc.gr/~hy240/current/submit.php)

General Description

In this assignment you are asked to implement a simplified movie streaming service. The service offers movies classified into different categories. Users register to the service, watch movies by adding them to their history, accept movie suggestions based on other users' watch history, and perform filtered searches across movie categories.

Detailed Description of Requested Implementation

The service you will implement classifies its movies into **6** categories: Horror, Science-Fiction, Drama, Romance, Documentary, Comedy. Each movie belongs to **a single** category and has a **unique** identifier. You will implement the categorization of movies through a 6-position table, the **category table**. At each position of the table is a pointer (of type **struct *movie ****), which points to the first element of the movie list corresponding to that category. This list is **singly-linked** and **sorted** in **ascending order** based on movie identifier (mid). A node in this list describes a movie that belongs to the category, through a structure (***struct movie***) with the following fields:

- **info:** Auxiliary struct of type **struct movie_info** describing the available information for a movie. Its fields are as follows:
    - **mid**: Unique identifier of the movie, type ***unsigned int***.
    - **year**: Year of release of the movie, type ***unsigned int***.
- **next**: Pointer (of type **struct *movie***) pointing to the next item in the list of movies in the category.

Before being inserted into the appropriate list of the category table, new movies added to the service are inserted into a separate list, the **new releases** list. This list contains movies of different categories and is **singly-linked** and **sorted in ascending order based on movie ID,** just like the lists in the category table. Its nodes are implemented with ***struct new_movie***, which has the following fields:

- **info**: Information about the movie, type **struct *movie_info***, as in struct movie.
- **category**: The category to which this movie belongs, which is represented as an enum of type ***movieCategory_t***.
- **next**: Pointer (of type **struct *new_movie***), pointing to the next node in the list of new releases.

Figure 1 shows the state of the category table on a hypothetical execution of the program. It is of fixed size, with 6 elements and one list of movies per element.

Users registered to the service are organized into an **unsorted, singly-linked list with a sentinel node**, the user list. The items in the user list are of type **struct user**. This structure contains for each user, in addition to their identifier, a pointer to the beginning of a doubly-linked list, called the user's suggested movies list, which contains the films suggested by the service to the user. The user list also contains, for each user, a pointer to the top element of a stack, which implements the user's movie watch history. The ***struct user*** fields are as follows:

- **uid**: Unique identifier of the user, type ***int***.

- **suggestedHead**: ***struct suggested_movie*** pointer (more information below), pointing to the first item in the user's doubly-linked list of suggested movies.
- **suggestedTail**: ***struct suggested_movie*** pointer, pointing to the last item in the user's doubly-linked list of suggested movies.
- **watchHistory**: ***struct movie*** pointer (as defined above), pointing to the top of the user's movie watch history stack.
- **next**: ***struct user*** pointer, pointing to the next node in the user list.

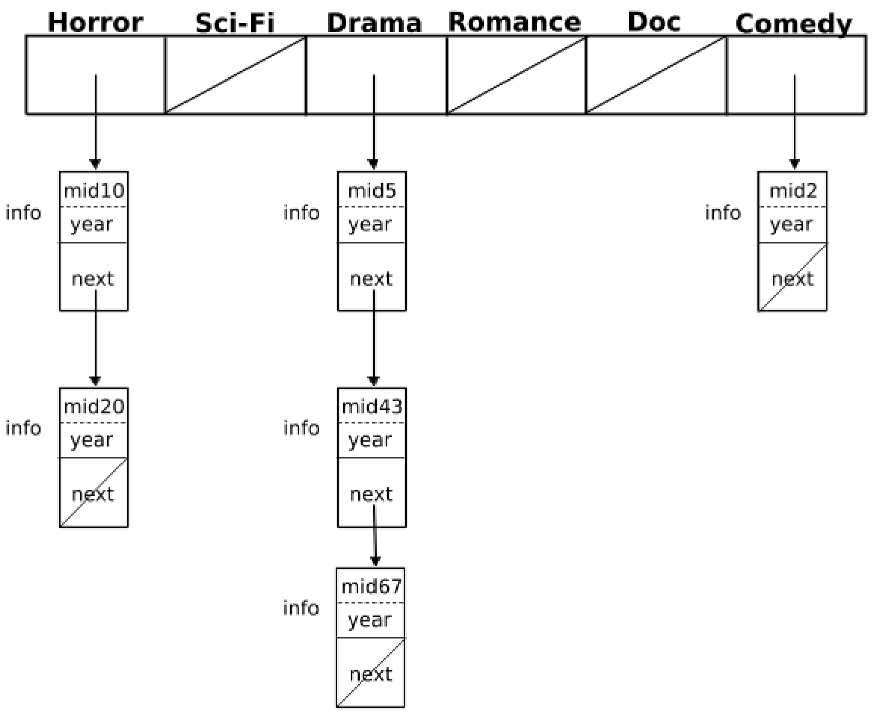


Figure 1: Category table with lists of films by category. Each list is sorted in increasing order by movie identifier (mid)

The sentinel node of the user list is a node that is used to properly manage the list (and is therefore an auxiliary node). It is of type ***struct user***, but its **uid** equals -1. The **suggestedHead, suggestedTail**, **watchHistory,** and **next** fields are initialized to NULL for each user. A possible state of the user list in a hypothetical execution of the program is illustrated in Figure 2.

To implement the user's doubly-linked list of suggested movies you will use the struct suggested_movie structure, the fields of which are:

- **info**: ***struct movie_info*** field, which contains the information about the movie.
- **prev**: Pointer of type ***struct suggested_movie***, pointing to the previous item in the suggested movies list.
- **next**: Pointer of type ***struct suggested_movie***, pointing to the previous item in the suggested movies list.

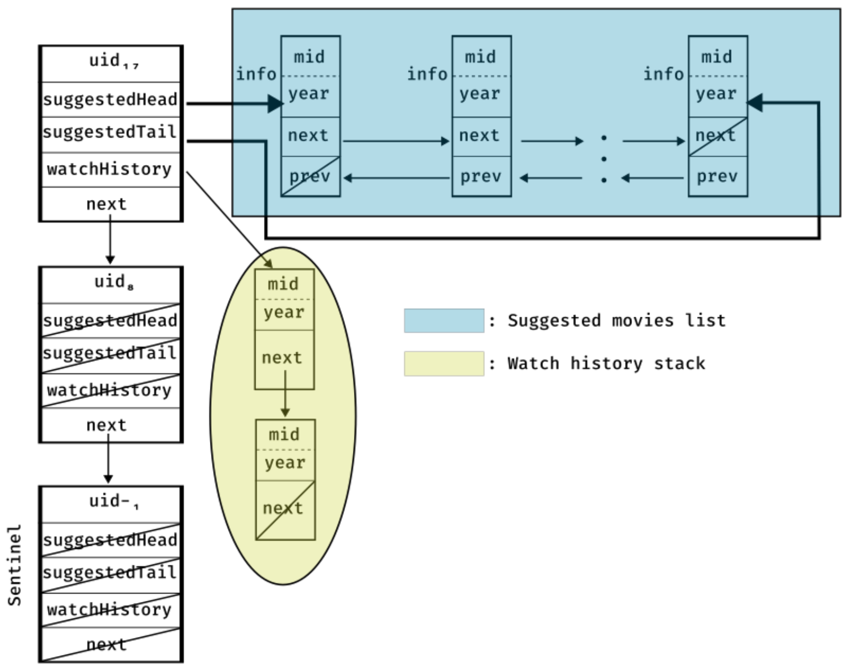


Figure 2: User list structure

Program Operation

The program you will implement should be invoked with the following command:

<executable> <input file>

where <executable> is the name of your executable (e.g., cs240StreamingService) and <input file> is the name of an input file (e.g., testFile), a file that contains one event per line. Events must be implemented by your program and follow one of the following formats:

– R <uid>

Register User event, in which a new user with the identifier <uid> is registered to the service. You must first iterate the user list to check that no user with identifier <uid> already exists. If this holds, you will need to implement an algorithm for inserting the new user in the user list. Your algorithm should have **time complexity O(1)**. The new user's suggestedHead, suggestedTail, and watchHistory fields are initialized to NULL. At the end of the execution of such an event, the program should print the following information:

```
R <uid>
   Users = <uid_1>, <uid_2>, ... , <uid_n>
DONE
```

where n is the number of nodes in the user list and for each i ∈ {1, … , n} uid_i is the user ID that corresponds to the i-th node in the user list.

- U <uid>

Unregister User event, in which the user with identifier <uid> leaves the service. You must first empty the user's doubly-linked list of suggested movies and their watch history stack by removing all their elements, if any, before removing the user from the user list. At the end of the execution of such an event, the program should print the following information:

```
U <uid>
   Users = <uid_1>, <uid_2>, ... , <uid_n>
DONE
```

where n is the number of nodes in the user list and for each i ∈ {1, ... , n} uid_i is the user ID that corresponds to the i-th node in the user list.

– A <mid> <category> <year>

Add New Movie event, in which the movie with identifier <mid>, category <category> and release year <year> is added to the service. **You must insert the movie into the sorted list of new releases, not the category table list.** The list of new releases should remain sorted (in ascending order based on the movie ID) after each insertion. At the end of the execution of such an event, the program should print the following information:

```
A <mid> <category> <year>
   New movies = <mid_1,category_1,year_1>, <mid_2,category_2,year_2>, ... , <mid_n, category_n,year_n>
DONE
```

where n is the number of items in the list of new releases and mid_ i, category_i , year_i, i ∈ {1, ... , n}, are respectively the identifier, category and release year of the film, corresponding to the i-th element of the new releases list.

– D

Distribute New Movies event, in which the movies in the new release list are distributed, based on the category they belong to, among the lists of the category table. This event must be implemented in **time complexity O(n),** where n is the size of the list of new releases. You must iterate through the list of new releases once, removing any movie you come across and inserting it to the appropriate list of the category table. At the end of this process, the list of new releases should be empty and the lists in the category table should be sorted in increasing order based on the movie identifier. At the end of the execution of such an event, the program should print the following information:

```
D
Categorized Movies:
   Horror: <mid_1,1>, <mid_1,2>, ... , <mid_1,n1>
   Sci-fi: <mid_2,1>, <mid_2,2>, ..., <mid_2,n2>
   Drama: <mid_3,1>, <mid_3,2>, ... , <mid_3,n3>
   Romance: <mid_4,1>, <mid_4,2>, ... , <mid_4,n4>
   Documentary: <mid_5,1>, <mid_5,2>, ... , <mid_5,n5>
   Comedy: <mid_6,1>, <mid_6,2>, ... , <mid_6,n6>
DONE
```

where n1, n2, ..., n6 are the sizes of the 6 category lists and <mid_i,j>, is the movie identifier corresponding to the j-th node of the category i movie list.

– W <uid> <mid>

User Watches Movie event, in which the user with identifier <uid> watches the movie with identifier <mid>. Initially, the information of the movie (struct movie_info) must be found by searching the lists of the category table and the user by searching the list of users. You must then initialize a new struct movie and push it to the user's watch history stack. At the end of the execution of such an event, the program should print the following information:

```
W <uid> <mid>
   User <uid> Watch History = <mid_1>, <mid_2>, ... , <mid_n>
DONE
```

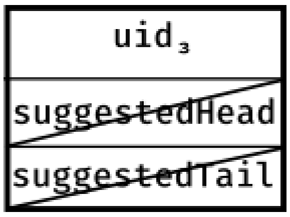
where n is the number of items in user <uid>'s watch history stack and mid _ i, i ∈ {1, ... , n}, the movie identifier corresponding to the i-th element of the user's watch history stack.

- S <uid>

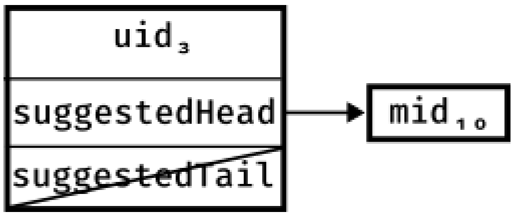
Suggest Movies to User event, in which the user with identifier <uid> receives movie suggestions from the service, based on the watch history of other users. For each user in the list of users –excluding user <uid>- you must remove one movie from the top of their watch history stack -if any- and insert it to the doubly-linked suggested movies list of user <uid>. These insertions should be done **"in alternating fashion",** to the beginning and the end of the list, as described below. The first item will be inserted as the first item of the list, and the suggestedHead pointer will point to it. The second item will be inserted as the last item of the list, and the suggestedTail pointer will point to it. The (2i+1)-th item, i > 0, will be inserted as the next of the (2i-1)-th item, while the (2i)-th item, i > 1, will be inserted as the previous of the (2i-2)-th item. Therefore, the third item will be inserted as the next of the first, while the fourth item will be inserted as the previous of the second, etc.

For example, suppose that there are 5 users -excluding the sentinel node- in the user list, with uids 7, 3, 15, 22, 37, and suppose that the requested event is S 3. In this case we want to remove one movie from the watch history stacks of users 7, 15, 22, 37, and at each step the suggested movies list of user 3 will be modified as follows:

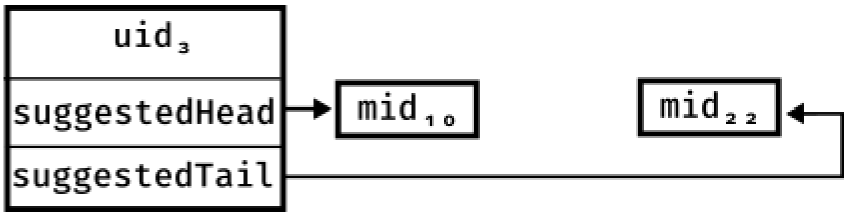
- Initial state: User 3's suggested movies list is empty.



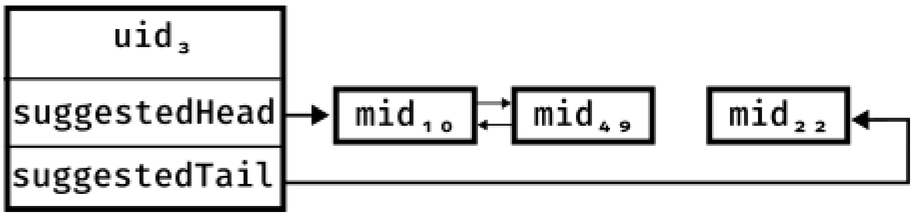
- Pop (User 7 Watch History) → mid 10



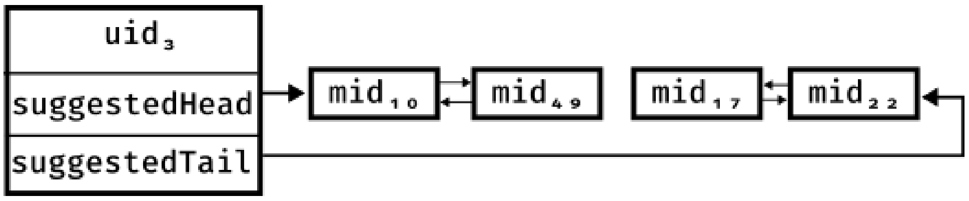
- Pop (User 15 Watch History) → mid 22



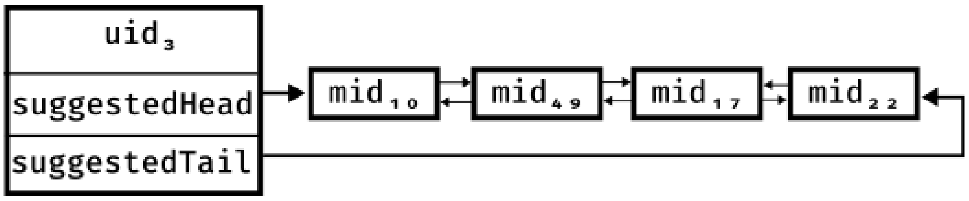
- Pop (User 22 Watch History) → mid 49



- Pop (User 37 Watch History) → mid 17



- User 3's final suggested movies list:



This process should be implemented in time complexity O(n), where n is the number of users of the service. At the end of the execution of such an event, the program should print the following information:

```
S <uid>
   User <uid> Suggested Movies = <mid_1>, <mid_2>, ... , <mid_n>
DONE
```

where n is the number of items in the suggested movies list of user <uid>, and mid_i, i ∈ {1, ... , n}, is the movie identifier that corresponds to the i-th node of the suggested movies list.

– F <uid> <category1> <category2> <year>

Filtered Movie Search event, in which user <uid> asks the service to suggest movies that belong to one of the two categories <category1> and <category2> and have a release year greater than or equal to <year>. You must iterate the movie lists for the two categories in the category table, locate the movies with the appropriate release year and join them into a new doubly-linked list. This list should be **sorted in ascending order by movie ID,** like the two category lists. This list is then appended to the end of user <uid>'s suggested movies list, if it is not empty. Otherwise the user's suggested movies list is initialized to equal the new list. **You should not remove items from the two category lists**, instead you should create a new node of the user's suggested movies list for each movie you find.

This event must be implemented in **time complexity O(n + m), where n, m is respectively the number of elements of the two category lists to iterate**. At the end of the execution of such an event, the program should print the following information:

```
F <uid> <category1> <category2> <year>
   User <uid> Suggested Movies = <mid_1>, <mid_2>, ... , <mid_n>
DONE
```

where n is the number of items in the suggested movies list of user <uid>, and mid_i, i ∈ {1, ... , n}, is the movie identifier that corresponds to the i-th node of the suggested movies list.

– T <mid>

Take Off Movie event, where the movie <mid> is removed from the service. You should remove the movie from any user's suggested movies list and from the appropriate category list of the category table. At the end of the execution of such an event, the program should print the following information:

```
T <mid>
   <mid> removed from <uid_1> suggested list.
   <mid> removed from <uid_2> suggested list.
   ...
   <mid> removed from <uid_n> suggested list.
   <mid removed from <category> category list.
   Category list = <mid_1>, <mid_k>
DONE
```

where n is the number of users with movie <mid> in their suggested movies list, uid_i, i ∈ {1, ... , n}, is the identifier of the i-th user in the user list with movie <mid> in their suggested movies list, <category> the category to which the film belonged, k the new size of the category list after the deletion of <mid> and mid_j, j ∈ {1, ... , k} is the identifier of the j-th element in the category list. *Hint: in order not to need to "remember" from which lists the movie was removed, you can print one line of information per removal you perform.*

– M

Print Movies event, in which you must print information about all movies in the category table lists. At the end of the execution of such an event, the program should print the following information:

```
M
Categorized Movies:
   Horror: <mid_1,1>, <mid_1,2>, ... , <mid_1,n1>
   Sci-fi: <mid_2,1>, <mid_2,2>, ..., <mid_2,n2>
   Drama: <mid_3,1>, <mid_3,2>, ... , <mid_3,n3>
   Romance: <mid_4,1>, <mid_4,2>, ... , <mid_4,n4>
   Documentary: <mid_5,1>, <mid_5,2>, ... , <mid_5,n5>
   Comedy: <mid_6,1>, <mid_6,2>, ... , <mid_6,n6>
DONE
```

where n1, n2, ..., n6 are the sizes of the 6 category lists and <mid_i,j>, is the movie identifier corresponding to the j-th node of the ith category list.

– P

Print Users event, in which you must print information about each user in the user list. At the end of the execution of such an event, the program should print the following information:

```
P
Users:
  <uid_1>:
    Suggested: <mid_1,1,>, <mid_1,2>, ... , <mid_1,s_1>
    Watch History: <mid'_1,1,>, <mid'_1,2>, ... , <mid'_1,w_1>
  <uid_2>:
    Suggested: <mid_2,1>, <mid_2,2>, ... , <mid_2,s_2>
    Watch History: <mid'_2,1>, <mid'_2,2>, ... , <mid'_2,w_2>
  ....
  <uid_n>
    Suggested: <mid_n,1>, <mid_n,2>, ... , <mid_n, s_n>
    Watch History: <mid'_n,1>, <mid'_n,2>, ... , <mid'_n,w_n>
DONE
```

where n is the number of items in the user list, s_i, w_i, i ∈ {1, ... , n}, are respectively the sizes of the suggested movies list and the watch history stack of the i-th user in the user list, mid_i,j, i ∈ {1, ... , n}, j ∈ {1, ... , s_i} is the movie identifier of the j-th item in the i-th user's suggested movies list, and mid'_i,j, i ∈ {1, ... , n}, j ∈ {1, ... , w_i} is the movie identifier of the j-th item in the i-th user's watch history stack.

Data Structures

You may not use out-of-the-box data structures (e.g., ArrayList) in your implementation regardless of the programming language you choose (C, C++, Java). Next are presented the structures in C that need to be implemented for the task:

```
typedef enum {
        HORROR,
        SCIFI,
        DRAMA,
        ROMANCE,
        DOCUMENTARY,
        COMEDY
} movieCategory_t;

struct movie_info {
        unsigned mid;
        unsigned year;
};

struct movie {
        struct movie_info info;
        struct movie *next;
};

struct new_movie {
        struct movie_info info;
        movieCategory_t category;
        struct new_movie *next;
};

struct suggested_movie {
        struct movie_info info;
        struct suggested_movie *prev;
        struct suggested_movie *next;
};

struct user {
        int uid;
        struct suggested_movie *suggestedHead;
        struct suggested_movie *suggestedTail;
        struct movie *watchHistory;
        struct user *next;
};
```