CS240: Data Structures

Winter Semester – School year 2023-2024

Professor: Panagiota Fatourou

Programming Exercise– 2η Phase

Submission Deadline: Friday, 22 Decembe2023, 23:59

Submission method : Through the turnin program. Information on the use of turnin on the course website.

(https://www.csd.uoc.gr/~hy240/current/submit.php)

General Description

In this assignment, you are called upon to implement a simplified movie streaming service. The service offers movies sorted into different thematic categories. Users register for the service, watch movies, adding them to their history, and perform filtered searches in movie categories."

Detailed Description of the Required Implementation

The service you will implement categorizes the available movies into **6** thematic categories: Horror, Science-Fiction, Drama, Romance, Documentary, Comedy. Each movie belongs to **only one** category and has a **unique** identifier. You will implement the categorization of movies through a 6-position array named the **'categories array**.' Each position in the array contains a pointer (of type **struct movie** *) to the root of a **binary search tree** that has a **sentinel node**. The tree of a category is sorted based on the movieID field of its nodes, following an in-order traversal in **ascending order**. Each element of the movie tree within a specific category is a **struct of type 'movie'** with the following fields:

- **info**: Helper structure of type **'struct movie_info'** describing the available information for a movie. Its fields are as follows:
  - **mid**: Unique identifier of the movie, of type ***int***.
  - **year**: Release year of the movie, of type ***int***.
  - **watchedCounter:** The number of users who have watched the movie .
  - **sumScore**: The sum of ratings given by users for the movie .
- A pointer '**lc**' pointing to the left child node of the node corresponding to the movie.
- A pointer **rc** pointing to the right child node of the node corresponding to the movie.

The mid of the sentinel node has an initial value of -1. The fields 'year,' 'watchedCounter,' and 'sumScore' of the sentinel node are initialized with the value 0. The pointers 'leftChild' and 'rightChild' will be initialized with the value NULL. Each category tree of movies is initially empty, containing only the sentinel node.

In **Figure 2**, the array of 6 positions with the category tree in each position is depicted.

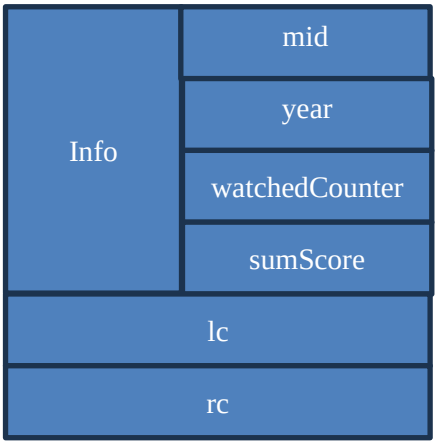


Figure 1 Record type struct movie

Before being inserted into the appropriate movie tree within the category array, new movies added to the service are introduced into a separate tree, the **'new releases tree**.' The new releases tree is a **sorted binary tree** based on the '**mid**' field, sorted via an in-order traversal. **Unlike the movie tree within a category, the new releases tree does not have a sentinel node.**

The nodes are implemented with the **struct new_movie,** which has the following fields:

- **info**: Information about the movie, of type **struct movie_info**.
- **category**: The category in which this movie belongs, which is represented as an enum of type movieCategory_t.
- **lc**: Pointer (type ***struct new_movie***), pointing to the left child node of the node corresponding to the new movie.
- **rc**: Pointer (type ***struct new_movie***), pointing to the right child node of the node corresponding to the new movie.

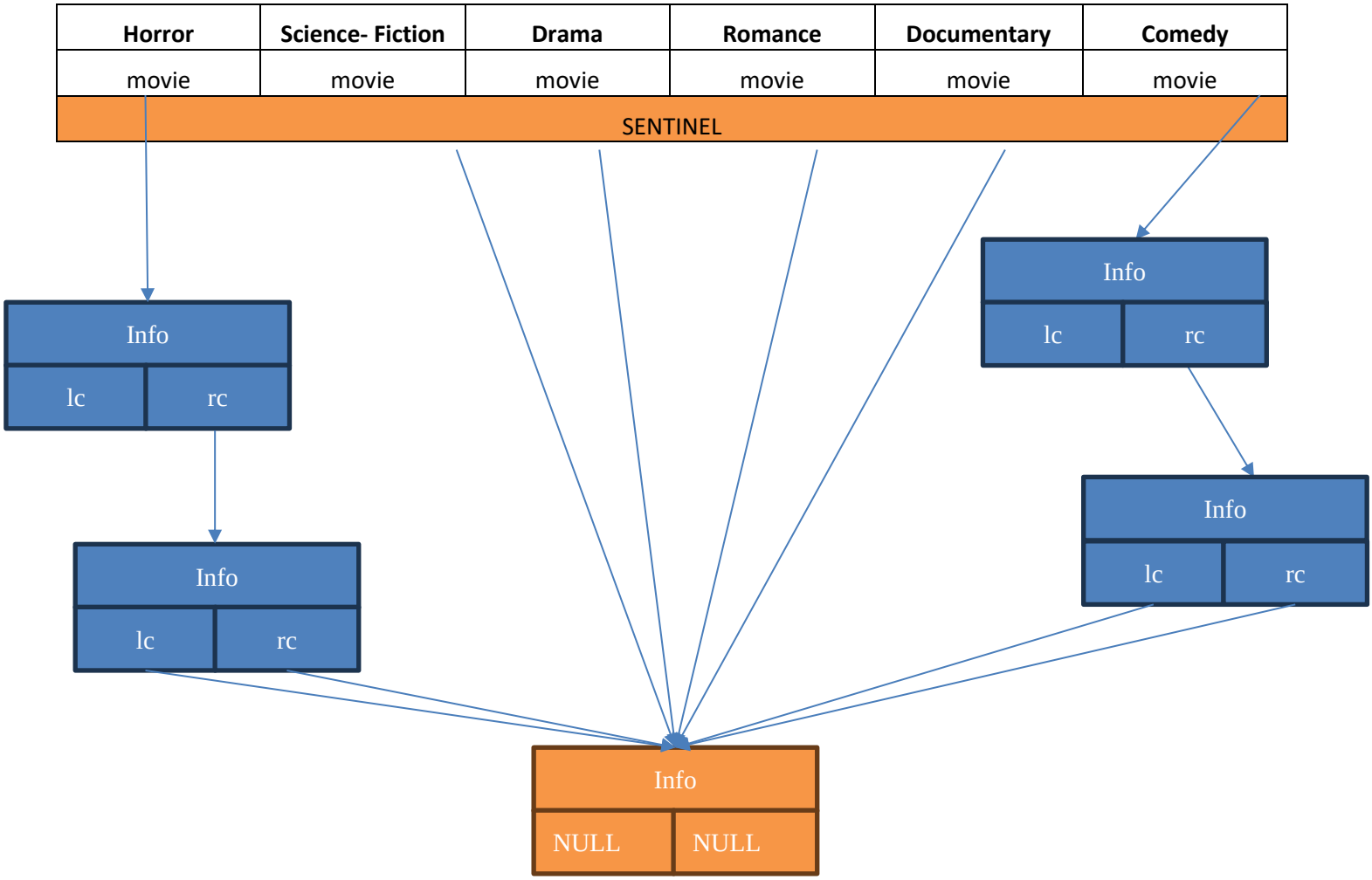


Figure 2: : The category array and category trees.

In **Figure 3**, you can see how the new releases tree looks in a hypothetical execution.

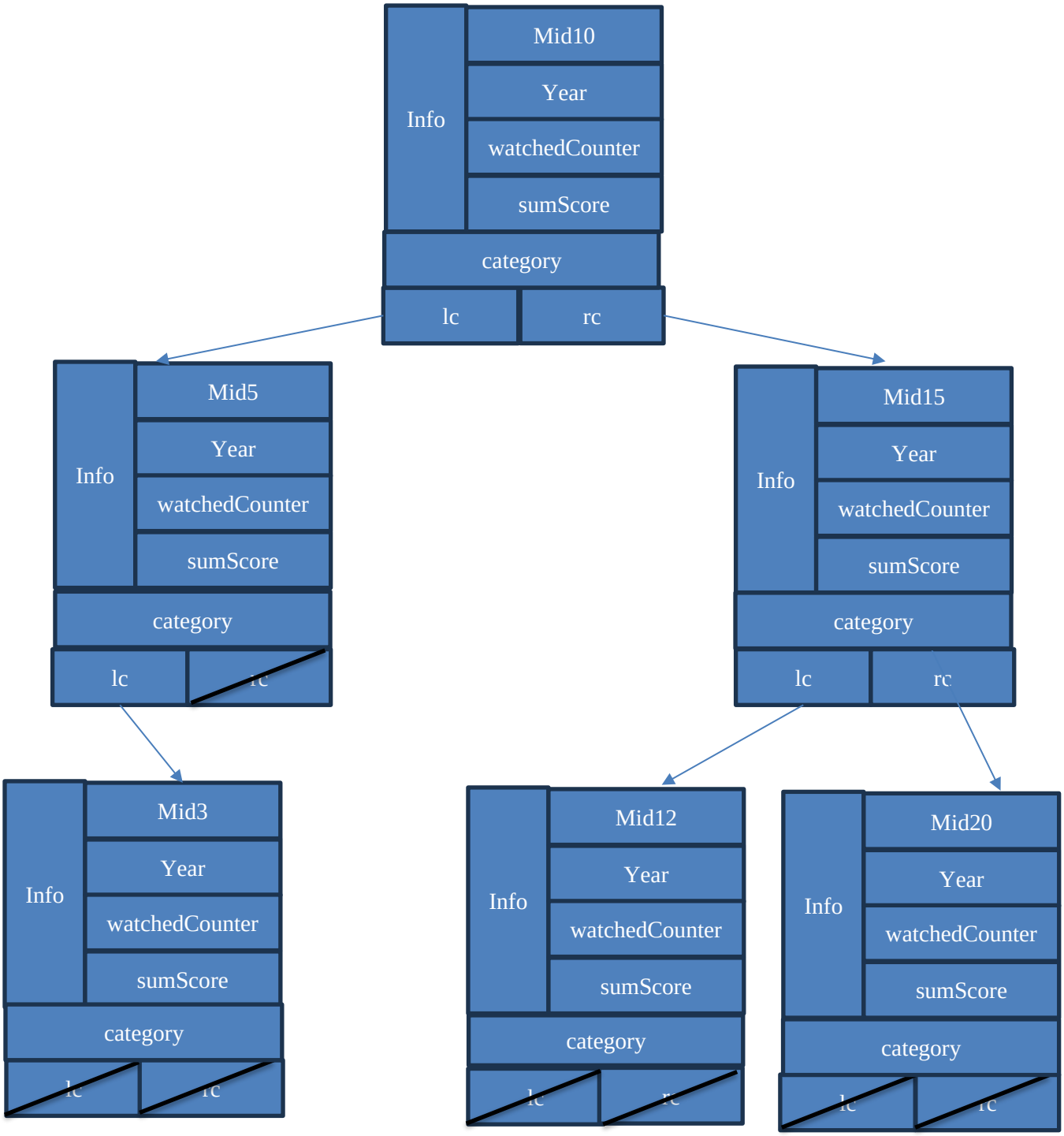


Figure 3: New Releases Tree - Movies

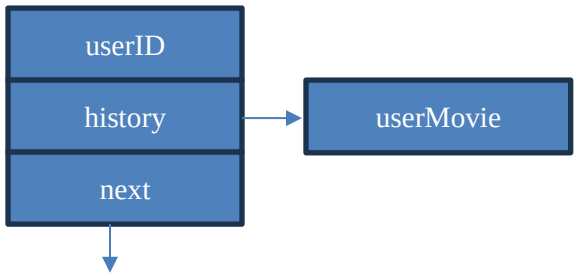
Data structures related to the user

The service serves a set of registered users. Users will be maintained in a **hash table USER[hash_table_size]**, which contains information about the users. To resolve collisions, you will follow the method of **non-sorted chains.**

The size of the hash table, **hash_table_size**, should be carefully chosen by you, and you should be able to justify your choice. Each position **i, 0 ≤ i < hash_table_size**, contains a pointer to the first element of a singly linked list that implements the chain of position i in the hash table. Each element of a chain is a record (struct) of type user, with the following fields (see Figure 4):

- An integer, **userID**, uniquely identifying the user .
- A pointer, **history**, into a userMovie struct (see below), which points to a **doubly-linked leaf-oriented binary search tree** named the **user's movie history tree**. The **tree is sorted** based on the **movieID** field according to an **in-order traversal** and contains movies that the user has already watched and rated..
- A pointer, **next**, that points to the next element of the chain i..

Προσέξτε ότι το **userID** κάθε χρήστη της αλυσίδας που δεικτοδοτείται από τη θέση i του πίνακα κατακερματισμού, έχει τιμή κατακερματισμού i.



Σχήμα 3: Εγγραφή τύπου User

For the implementation of the hashing function, you should rely on the technique of **universal hashing**. For the implementation of **universal hashing**, the following will be provided:

1) An array primes[], that contains prime numbers in ascending order .
2) The maximum number of users, via the variable max_users.
3) The maximum user identifier, via the variable max_id.

These variables are global, declared in the Movie.h file, and will be initialized in the main based on values specified in the first lines of each test_file.

The user hash table is shown in **Figure 4**.

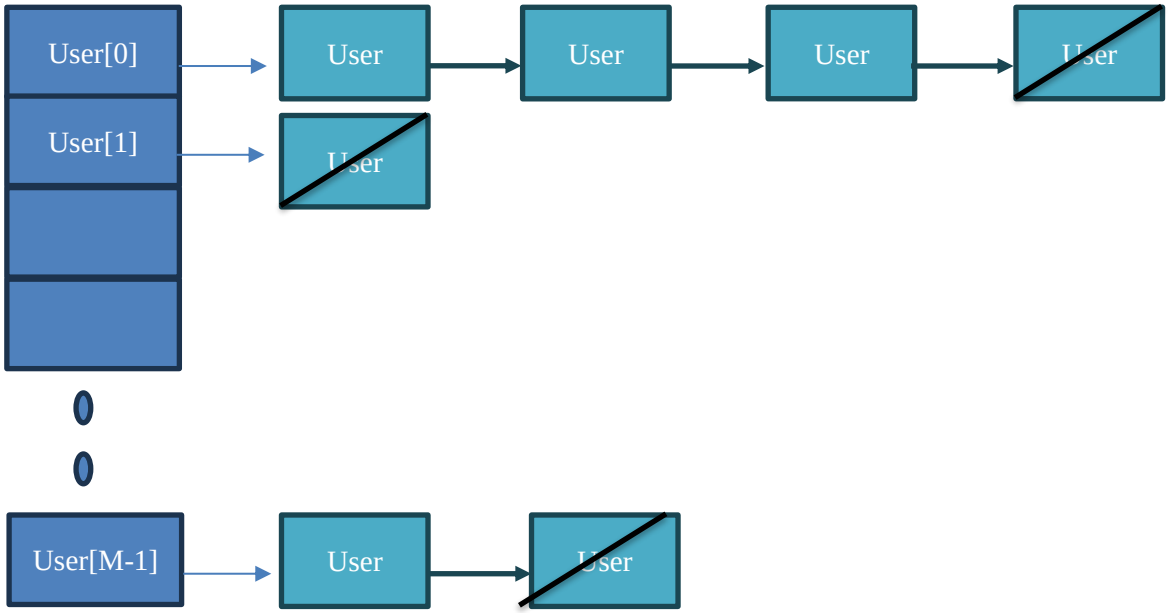
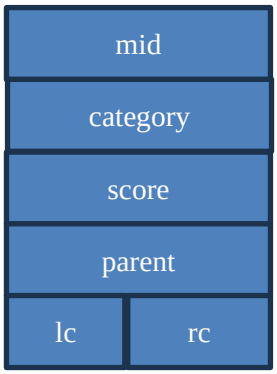


Figure 4 :The hash table for users using the method of unsorted chains.

Each node of the user's movie history tree corresponds to a record of type userMovie. The **struct userMovie** contains the following fields:

- An integer **mid** που χαρακτηρίζει μοναδικά την ταινία
- Έναν ακέραιο **category** που αντιστοιχεί στη θεματική κατηγορία της ταινίας. Η μεταβλητή αυτή λαμβάνει τιμές από 0 έως 5, όπου 0: Horror, 1: Science- Fiction, 2: Drama, 3:Romance, 4:Documentary, 5:Comedy.
- Έναν ακέραιο **score** που αντιπροσωπεύει την βαθμολογία που έδωσε ο χρήστης στη ταινία. Η μεταβλητή αυτή παίρνει τιμές στο διάστημα 1 έως 10, με 1 να είναι η χαμηλότερη βαθμολογία για μια ταινία και 10 η μεγαλύτερη.
- Έναν δείκτη **parent** που δείχνει στον πατέρα του κόμβου
- Έναν δείκτη **lc** που δείχνει στον αριστερό θυγατρικό κόμβο.
- Έναν δείκτη **rc** που δείχνει στον δεξιό θυγατρικό κόμβο.

Η εγγραφή τύπου userMovie παρουσιάζεται στο Σχήμα 5.



Σχημα 5: Εγγραφή τύπου userMovie

Το *δένδρο ιστορικού* είναι **διπλά-συνδεδεμένο φυλλο-προσανατολισμένο δένδρο δυαδικής αναζήτησης**. Τα φυλλοπροσανατολισμένα δένδρα δυαδικής αναζήτησης (**leaf-oriented binary search trees)** αποτελούν μια εναλλακτική υλοποίηση του αφηρημένου τύπου δεδομένων του λεξικού. Ορίζονται ως εξής:

a) All dictionary keys are stored in the tree's leaves, from left to right in non-decreasing key value.

b) Internal nodes store keys (which do not necessarily correspond to dictionary keys) so that the following invariant condition holds for each node v:

Το κλειδί του αριστερού παιδιού του v είναι μικρότερο από αυτό του v, ενώ το δεξιό παιδί του v διαθέτει κλειδί μεγαλύτερο ή ίσο από εκείνο του v.

Note that according to the definition, internal nodes have both non-empty pointers, while both leaf pointers are empty. Therefore, a leaf-oriented tree is full. The history tree of a user indexed by one of the chain nodes of the hash table is shown in **Figure 6**.

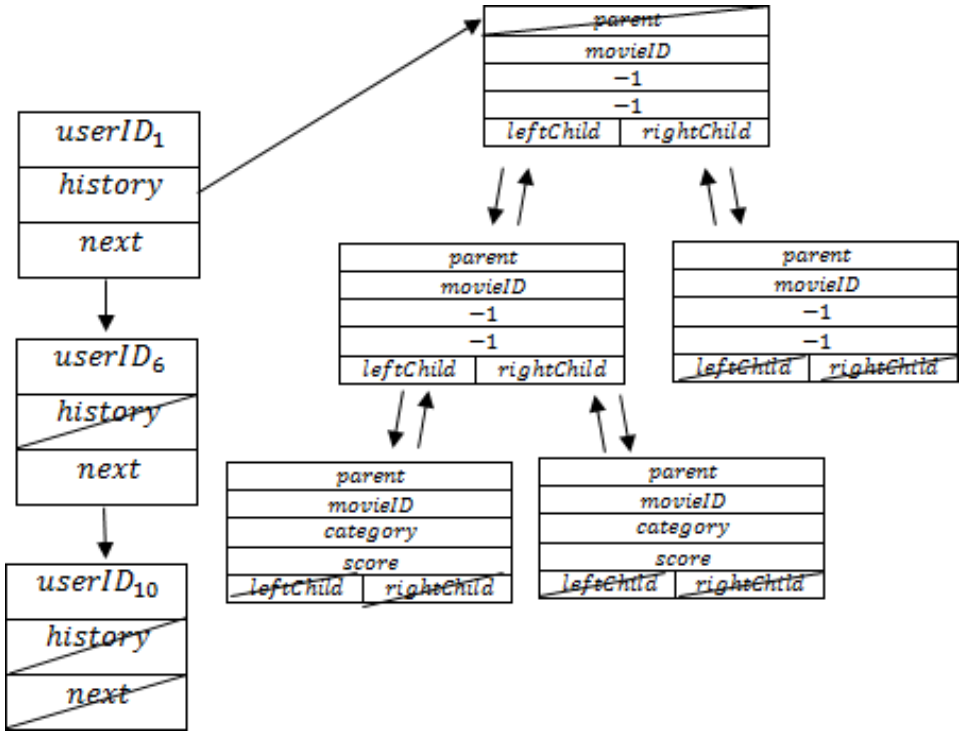


Figure 6: The history tree of a user indexed by one of the nodes in a chain of the hash table

Rules for insertion and deletion in a leaf-oriented tree.

1. To **insert** a new node, v, with key K into a leaf-oriented binary search tree, we perform a search to find the leaf, v', which should be the parent node of v in the tree. However, the key, K', of v' must still appear in a leaf of the tree. To achieve this, we replace v' with a three-node tree consisting of an internal node with two leaf children. The left of these two leaves has a key of min{K, K'}, while the right leaf and v have a key of max{K, K'} (Figure 6).).
2. To **delete** a node, v, from a leaf-oriented binary search tree, we find its parent node, v', and also the parent node of v', denoted as v''. For deletion, we replace the pointer of v'' that points to v' so that it points to the sibling node of v. An example of a leaf-oriented tree is shown in Figure 7, while insertion and deletion examples in a leaf-oriented tree are presented in Figures 8 and 9.

An example of a leaf-oriented tree is presented in Figure 7, while examples of insertion and deletion in a leaf-oriented tree are shown in Figures 8 and 9.

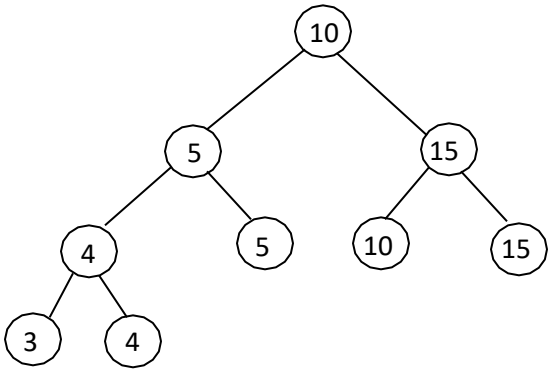


Figure 7: Example of leaf-oriented binary search tree.

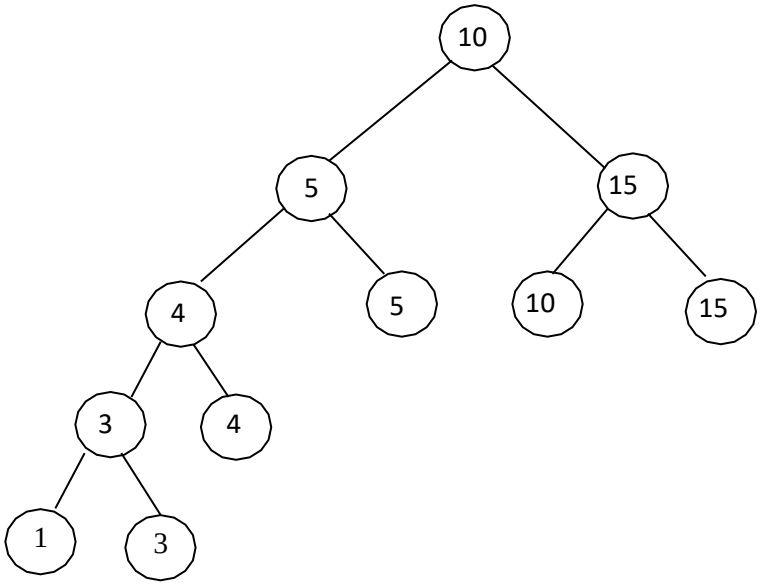


Figure 8 : Inserting the key 1 into the tree shown in Figure

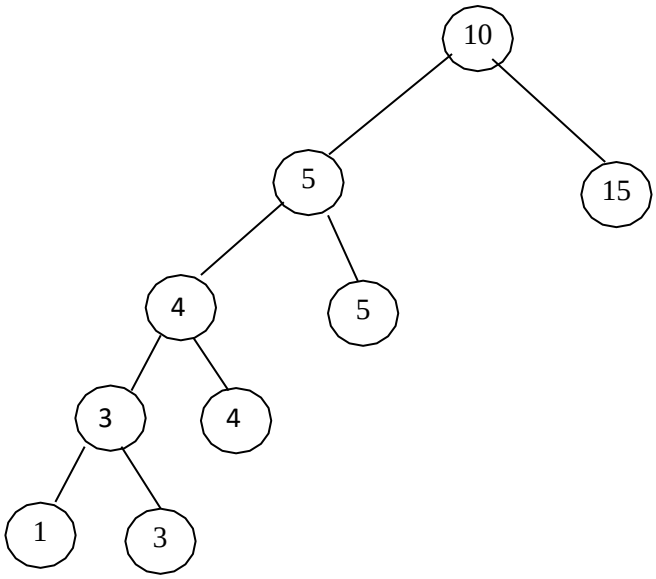


Figure 9: Deletion of key 10

Program Operation Method

The program that will be created should be executed by calling the following command:

<executable> <input-file>

where <executable> is the name of the program's executable file (e.g., a.out), and <input-file> is the name of an input file (e.g., testfile) containing the events..

The input events are as follows:

R <userID >

The event type **'register user'** signifies the registration of a new user with an identifier <userID>. This event adds the new user to the service's user hash table. The 'history' field of the user must have an initial value of NULL.

Upon completion of such an event, the program should print the following information:

```
R <userID>
Chain <j> of Users:
    <userID1>
    <userID2>
       ...
    <userIDn>
DONE
```

where <j> is the hash value of the key <userID>, n is the number of users in the chain indexed by the position <j> of the Users array, and for each i ∈ {1, ..., n}, <userIDi> is the identifier of the user corresponding to the i-th node of this chain.

U < userID >

A **unregister user** event signifies the deletion of a user with the identifier <userID> from the users' hash table. Prior to the definitive removal of the user from the user list, all elements within the user's movie history tree should be deleted if it contains any elements

During this event, the appropriate chain is located based on the hash function, followed by a search to find the suitable node within that chain.

Upon completion of such an event, the program should print the following information

```
U <userID>
Chain <j> of Users:
    <userID1>
    <userID2>
       ...
    <userIDn>
DONE
```

where <j> is the hash value of the <userID> key, n is the number of users in the chain indexed by the position <j> in the Users array, and for each i ∈ {1, ..., n}, <userIDi> is the identifier of the user corresponding to the i-th node of that chain

A <mid > <category> <year>

Event of type add **new movie**, which indicates the arrival of a new movie available to users. During this event, a new movie will be created with the identifier <movieID> and release year <year>, belonging to the thematic category <category>. **Regardless of its category, the new movie will be inserted into the new releases tree**. The fields watchedCounter and sumScore will be initialized with a value of 0.

After the execution of such an event, the program should print the following information:

```
A <movieID> <category> <year>
New releases Tree:
   <new_releases>: <movieID1>, ... , < movieIDn>
DONE
```

where n is the number of nodes in the new releases tree. The nodes should have been inserted in such a way that if an **in-order traversal is performed** on the tree, the movies will be accessed in **ascending order based on the movieID** field**.**

D

The **Distribute movies** event signifies categorizing the movies contained in the new releases tree into the remaining thematic categories. In this event, you'll traverse the new releases tree, and for each node, v, found in it, you'll insert a new node into the tree of the appropriate thematic category. Subsequently, you'll delete node v from the new releases tree. Pay attention, the tree for each category should have a height of O(log n) (refer to exercise set 3 to create trees with the appropriate height).

After the execution of such an event, the program should print the following information:

```
D
Movie Category Array:
    <category0>: <movieID01>, ... , < movieID0n0>
    <category1>: <movieID11>, ... , < movieID1n1>
    ...
   <category5>: <movieID51>, ... , < movieID5n4>
DONE
```

where for each i, 0 ≤ i ≤ 5, ni is the size of the movie tree in category i of the categories array, and for each j, 1 ≤ j ≤ ni, <movieIDij> is the identifier of the movie corresponding to the j-th node of the movie tree in category i, as obtained from its in-order traversal.

I <movieID> <category>

An event of type **search movie** which signifies the search for the movie identified by <movieID> in the movie tree under the category <category>.

Upon completion of this event, the program should print the following information

```
I <movieID> <category> <year>
DONE
```

"where <year> is the release year of the movie identified by <movieID>."

W <userID > <category><movieID> <score>

Event of type **'watch movie'** indicating that the user with ID <userID> has watched the movie identified by <movieID> and rated it with a score of <score>. During this event, a search for the movie with ID <movieID> occurs in the category tree <category>. When the movie node is found, the 'watchedCounter' field is incremented by one, and the 'sumScore' field is increased by the value <score>

Subsequently, a node named 'userMovie' is created. This node will have 'movieID' and 'category' fields with the same values as those in the struct corresponding to the movie with ID <movieID>. The 'score' field of the 'userMovie' node will have the value <score>. This node is then inserted into the user's history-oriented tree with ID <userID>.

Upon completion of this event, the program should display the following information:

```
W<userID><category><movieID><score>

History Tree of user <userID>:

       <userMovieID_1, score_1>

       <userMovieID_2, score_2>

       …

       <userMovieID_n, score_n>

DONE
```

The number of nodes in the user's movie history tree with ID <userID>, denoted as n, and for each i ∈ {1, …, n}, <movieIDi> represents the identifier of the movie corresponding to the i-th node in the tree, as determined by its in-order traversal, while <scorei> is the rating associated with that movie node

F <uid > <score>

Event type: 'filter movies', where the user requests the service to suggest movies belonging to any category with a rating greater than or equal to <score>. You'll need to traverse the category trees to find the number of movies, numMovies, that have an average rating greater than <score>. The average rating of a movie is calculated as follows::

$$average = \frac{sumScored}{watchedCounter}$$

Next, you should use an auxiliary array of size equal to the number, numMovies. Then, traverse each category's movie tree <category> and store pointers to the tree nodes corresponding to movies with an average rating greater than the score in the auxiliary array. Finally, apply the heapsort algorithm to sort the array based on the average ratings of the stored movies.

After the execution of such an event, the program should print the following information :

```
F <uid><score>
      <movieID0> <avgScore0> … <movieIDn-1> <avgScoren-1>
DONE
```

It should display, for each i where $0 \leq i \leq n-1$, ki j where $1 \leq j \leq ni$, <movieIDi> as the identifier of the movie corresponding to the j-th position in the auxiliary array after the application of the heapsort algorithm

Q <userID>

Event of type 'users' average rate', which signifies the computation and printing of statistics for the rating of the user with the identifier <userID>. Specifically, in this event, you should follow these steps:

Locate the appropriate user chain based on the hash function and perform a search to find the node within the chain that corresponds to the user with the identifier <userID>>.

Then, traverse through the history tree of this user and store in a helper variable, ScoreSum, the sum of the score fields of the tree nodes. Additionally, store in another helper variable, counter, the count of movies stored in the tree. Finally, divide the ScoreSum by the counter to find the average rating given by the user with the identifier <userID> to the movies they have watched.

Traversal of the leaves of the leaf-oriented tree should occur as follows: Initially, locate the leftmost leaf, v, in the tree. To find the leaf with the immediately larger key than v, implement an algorithm, FindNextLeaf(), which takes a pointer to node v as a parameter and executes in O(h) time. Traversal of the leaves will be done by iteratively calling FindNextLeaf() starting from the leftmost leaf until all the leaves are exhausted.

After the execution of such an event, the program should print the following information:

```
Q <userID><MScore>
DONE
```

Where MScore is the average rating in the movies watched by the user with the identifier <userID>, as calculated above.

M

Event of type 'print movies', indicating the printing of the tree of movies for all categories. In this event, for each category, perform an in-order traversal of the tree and print the nodes traversed.

After the execution of such an event, the program should print the following information:

```
M
Movie Category Array:
    <category0>: <movieID01>, ... , < movieID0n0>
    <category1>: <movieID11>, ... , < movieID1n1>
    ...
   <category5>: <movieID51>, ... , < movieID5n4>
DONE
```

For every i, $0 \leq i \leq 4$, ni is the size of the i-th category movie tree in the category array, and for every j, $1 \leq j \leq ni$, <movieIDij> is the movie ID that corresponds to the j-th node of the i-th category movie tree, as it results from the in-order traversal of that tree

P

A print users event of type that signals the printing of the user hash table. For each user, all the fields of the struct that correspond to it (excluding the pointers) should be printed, including the history tree.

After the execution of such an event, the program should print the following information:

```
P

…

Chain <j> of Users:
       <userID_1>
        History Tree:
              <movieID^1_1> <score^1_1>
              …
              <movieID^1_n1> <score^1_n1>
       <userID_2>
        History Tree:
              <movieID^2_1> <score^2_1>
              …
              <movieID^2_n2> <score^2_n2>
       …
       <userID_n>
        History Tree:
              <movieID^n_1> <score^n_1>
              …
              <movieID^n_nm> <score^n_nm>
…
DONE
```

Where <j> is the j-th chain of the hash table, n is the number of users in the chain pointed to by the position <j> of the Users table, and for every i Î{1, …, n},<userIDi> is the user ID that corresponds to the i-th node of that chain. Also, 1 ≤ k ≤ ni, <movieIDik> is the movie from the history tree of the user with ID <userIDi>. In this event, the entire hash table should be printed.

Δομές Δεδομένων

In your implementation, you are not allowed to use pre-made data structures (e.g., ArrayList) whether the implementation is done in C, C++, or Java. The following C structures must be used for the implementation of this assignment

```
/**
* Structure defining a node of movie binary tree (dendro tainiwn kathgorias)
*/
typedef struct movie{
    int movieID;                /* The movie identifier*/
    int category;               /* The category of the movie*/
    int year;                   /* The year movie released*/
    int watchedCounter;         /* How many users rate the movie*/
    int sumScore;               /* The sum of the ratings of the movie*/
    struct movie *lc;           /* Pointer to the node's left child*/
    struct movie *rc;           /* Pointer to the node's right child*/
}movie_t;


/**
* Structure defining movie_category
*/

typedef struct movie_category{
    movie_t *movie;             /* Pointer to movie tree */
    movie_t *sentinel;          /* Pointer to movie tree sentinel */
}movieCategory_t;

/**
* Structure defining a node of user_movie for history doubly linked binary
* tree (dentro istorikou)
*/
typedef struct user_movie{
    int movieID;                    /* The movie identifier*/
    int category;                   /* The category of the movie*/
    int score;                      /* The score of the movie*/
    struct user_movie *parent;      /* Pointer to the node's parent*/
    struct user_movie *lc;          /* Pointer to the node's left child*/
    struct user_movie *rc;  /* Pointer to the node's right child*/
}userMovie_t;

/**
* Structure defining a node of users' hashtable (pinakas katakermatismou
* xrhstwn)
*/
typedef struct user {
    int userID;             /* The user's identifier*/
    userMovie_t *history;   /* A doubly linked binary tree with the movies
watched by the user*/
    struct user *next;      /* Pointer to the next node of the chain*/
}user_t;

/* Global variables for simplicity. */
```

```
movieCategory_t *categoryArray[6];  /* The categories array (pinakas
kathgoriwn)*/
movie_t *new_releases;     /* New releases simply-linked binary tree*/
user_t **user_hashtable_p; /* The users hashtable. This is an array of
chains (pinakas katakermatismoy xrhstwn)*/
int hashtable_size;         /* The size of the users hashtable)*/
int max_users;              /* The maximum number of registrations (users)*/
int max_id;                 /* The maximum acount ID */
int primes_g[160];          /* Prime numbers for hashing*/
```