תרגיל רטוב 1 – חלק יבש

שרה גריפית – 341312304
שלמה אבנון – 322677063

Description of the Data Structure:

For Webflix's application, we will build a data structure made up of four main types of trees:

- m_moviesByID – an AVL tree of all the movies in the system, organized by the singular ID of each movie
- m_usersByID – an AVL tree of all the users in the system, organized by the singular ID of each movie
- m_groupsByID – an AVL tree of all the groups in the system, organized by the singular ID of each group
- m_moviesByRating – an AVL tree of movies in the system, organized by three different parameters (MultiTree):
    - Firstly, by the rating of each movie (from high to low)
    - If equal, then by the number of views each movie has (from high to low)
    - If still equal, then by the ID number of each movie (from low to high)

  There will be five AVL trees of this type, where one of them is a tree of all the movies in the system, and the other four are trees of all the movies of the same genre (one tree per genre). Each tree of this type maintains a pointer to its "largest" node.

The streaming service will also contain a variable that holds the number of movies that exist in the entire system.

We will add three classes, User, Movie, and Group, that contain the details of the users, movies, and groups in the system:

- The **User** class will contain the basic details of the users in the streaming system: ID number, VIP status, an array containing the user's views according to genre, and a pointer to the user's group (equal to nullptr if the user does not belong to a group)
- The **Movie** class will contain the basic details of the movies in the streaming system: ID number, genre, the number of views the movie has, the movie's VIP status (whether it is only available to VIP users or not), and the rating the movie has received (saved as the number of raters and the sum of the ratings to avoid potential computational errors)
- The **Group** class will contain the basic details of the groups in the streaming system: ID number, VIP status (the group is considered a VIP group if it contains at least one VIP member), the number of users in the group, and two arrays tracking the group's and its members' views according to genre. One array tracks the number of movies the group watches together, and one array tracks the total views of all the members of the group, both within and outside the group, since joining. Each time a user is added to the group, the group's views are subtracted from the user's views, updating the user's views array. When returning the number of views a user has, its group's views are added to it. In this way, each time the group watches a movie together, all its users' views are updated accordingly. Additionally, the group holds an AVL tree of pointers to the users in the group, organized according to their ID numbers.

Let us note that there is only one copy of each object (users, movies, and groups), and that the trees in the system hold pointers to that original copy.

Space Complexity: In each function, there is no additional space complexity used, beyond what is required by the assignment. As such, the space complexity remains O(n+k+m), where n is the number of users, k the number of movies, and m the number of groups in the system.

The functions required for the system and an explanation of their implementations and complexities:

streaming_database -

The counter m_totalMovies is initialized to zero, and the eight AVL trees described above are initialized as empty through a call to their constructors.

Time complexity: An empty tree contains a single, empty node, and the time it takes to create it is O(1). Similarly, it takes O(1) to initialize an int value. Therefore, the time complexity is **O(1).**

~streaming_database –

Firstly, the data of all the users, movies, and groups in the system are destroyed. This is done by calling a function that frees the memory held in the data in each of the trees, m_usersByID, m_moviesByID, m_groupsByID, which hold all the users, movies, and groups held by the system. Afterwards, the default destructors of each of the eight trees are called, freeing the memory held by the nodes in each tree.

Finally, the default destructor of the int variable, m_totalMovies, is called, freeing its memory as well.

Time complexity:

To free the data, it is required to go over each of the users, movies, and groups in the system. This is done in time O(n+k+m), where n is the number of users in the service, k is the number of movies, and m is the number of groups. While destroying each group, the tree holding the group's members is destroyed. Each user can at most belong to one group, so at the maximum, the users in the system are gone over twice throughout destruction.

After this, the destructors for the various trees are called, which go over all the nodes in the streaming system:

- There are two trees that hold all the movies, and one additional tree per genre. Therefore, each movie appears a total of three times in the system's trees. Therefore, the time complexity of deleting these trees is O(3k) = O(k)
- As explained above, each user is gone over twice at most throughout destruction. Therefore, the time complexity of deleting the users is O(2n) = O(n)
- There is one tree of groups, so each group is gone over once during deletion, giving a time of O(m).

As such, in total, the time complexity is O(3k+2n+m) = **O(n+k+m).**

add_movie –

In this function, a movie is added to the streaming system. First, the input is checked, and INVALID_INPUT is returned if the movie's ID is less than or equal to zero, the genre is NONE, or the views are less than zero. Additionally, the system is checked to ensure that no other movie already exists with the same ID number. If there is such a movie, FAILURE is returned. If the input is valid in all the above ways, memory is allocated for the new movie. ALLOCATION_ERROR is returned in the event of failure.

The new movie is added to the AVL tree of all the movies organized by their IDs, m_moviesByID. The movie is then added to the AVL MultiTree of all the movies organized by their ratings (and by views and ID number as explained above), m_moviesByRating. The number of total movies in

the system is increased by one, and the movie is then added to the appropriate AVL MultiTree organized by rating according to genre (for example, m_dramaByRating).
Finally, SUCCESS is returned.
<u>Time complexity:</u>
To insert the new movie into the AVL tree by IDs, a search is done according to its ID number, in time O(log(k)). After insertion, rotations are done in order to rebalance the tree, in time O(1) per rotation as shown in class. Going over each node along the search path until the root (maximum log(k) nodes), the height and balance factor of each node is updated. Therefore, the rebalancing takes a time of O(log(k)).
This process is repeated similarly in each of the other trees, where the search is done according to the movie's rating, views, and then ID number. Each of the other trees can contain at most k nodes, the number of movies in the system. After rebalancing, the tree's maximum node is updated. This is done by going down the right side of the tree in time O(log(k)). As such, insertion to each tree takes at most time of O(log(k)).
Each movie is inserted into three trees, and so the total time it takes to add a movie to the system is: O(3log(k)) = **O(log(k)).**

remove_movie –
In this function, a movie is deleted from the streaming service system. Firstly, the validity of the input is checked. If the given movie ID is less than or equal to zero, INVALID_INPUT is returned. If the input is valid, the movie is searched for in the AVL tree m_moviesByID according to its given ID number. A pointer to the movie is returned from the search. If the movie does not exist in the system, FAILURE is returned. Otherwise, the movie is then removed from the appropriate genre's AVL MultiTree. The movie's rating and views are accessed through the pointer to the desired movie. Afterwards, the movie is removed from the trees m_moviesByRating and m_moviesByID. The number of total movies in the system is decreased, and the memory the movie uses is freed. SUCCESS is returned.
<u>Time complexity:</u>
To remove the movie from the system, a search is performed according to its ID number, in time O(log(k)). Once found, the movie is removed from each of the AVL trees it appears in (a total of three). To do so, the movie is searched for in each of the AVL trees, and its node is removed. The tree is rebalanced through rotations if necessary, and the balance factor and height of each node along the search path is updated. After rebalancing, the tree's maximum node is updated. This is done by going down the right side of the tree in time O(log(k)). As shown in the previous function, rebalancing and updating each tree takes a time of O(log(k)).
As such, the movie is searched for four times, once to receive a pointer to the object, and once in each of the three trees it appears in, m_moviesByID, m_moviesByRating, and the genre-related rating tree. The time complexity of each search is O(log(k)) because there is a maximum of k movies in each tree. Therefore, the total time of removing a movie from the streaming system is O(4log(k)) = **O(log(k)).**

add_user –
In this function, a user is added to the streaming system. Firstly, the validity of the input is checked. If the user's ID number is less than or equal to zero, INVALID_INPUT is returned. If there is already a user with the same ID number in the system, FAILURE is returned. Each new user does not belong to a group, so the user only needs to be inserted into the AVL tree that holds all

the users in the system, m_usersByID. Memory is allocated for the user, and it is inserted into the AVL tree. If there is a problem with the memory allocation, ALLOCATION_ERROR is returned. Otherwise, SUCCESS is returned.

Time Complexity:

To add the user to the tree, the tree is searched according to the ID number given in time O(log(n)). Once the user's proper location is found, a node containing a pointer to the user is added, and the tree goes through a process of rebalancing. Rotations are performed as needed, in time O(1) per rotation. The height and balance factor of each node along the search path is updated (maximum of log(n) nodes). Therefore, the total time it takes to add a user to the system is **O(log(n)).**

remove_user –

In this function, a user leaves the streaming service. Firstly, the validity of the input is checked. If the user's ID number is less than or equal to zero, INVALID_INPUT is returned. If the user does not exist in the system, FAILURE is returned. Otherwise, a pointer to the user is found by searching and returning the data held by the user's node in the AVL tree m_usersByID. If the user belongs to a group, it must be removed from that group's tree of users. Additionally, the user's views are subtracted from the group's views. This is done by going over each cell in the arrays held by the group and user, finding the user's total views and then subtracting them from the group's total views. This keeps the group's total views updated, with only its remaining members' views. Finally, the user is removed from the AVL tree that contains all the users in the system, m_usersByID. SUCCESS is returned.

Time complexity:

To access a pointer to the user, the AVL tree m_usersByID is searched in time O(log(n)) as shown in class. Once found, in the worst case, the user is removed from its group's tree. This takes a time of O(log(g)), including rotations and rebalancing, where g is the number of users in that group. Because each user can only be in one group, g is less than or equal to n. Therefore, the time it takes to remove the user from its group's tree is O(log(n)) in the worst case. To subtract the user's views from the total views of the group, two arrays are gone over at the same time. Each array has four cells, so this takes time O(4) = O(1). Thereafter, the user is removed from the AVL tree containing all of the nodes in the system, m_usersByID. This is done by searching for the user, removing and deleting its node, and rebalancing the tree as necessary. As explained above, this takes time O(log(n)).

In total, the time it takes to remove a user from the system is, at worst, O(3log(n)+1) = **O(log(n)).**

add_group –

In this function, a new group is added to the streaming system. Firstly, the validity of the input is checked. If the group's ID is less than or equal to zero, INVALID_INPUT is returned. If a group with the same ID number already exists in the system, FAILURE is returned. Otherwise, memory is allocated for the new group. If there is a problem with the memory allocation, ALLOCATION_ERROR is returned. Otherwise, the group is inserted into the AVL tree containing all the groups in the system, sorted according to ID number, m_groupsByID. Each group begins with no users, so the group's AVL tree of its users is initialized as empty. SUCCESS is returned.

Time complexity:

When creating a new group, its tree of users is empty, allowing it to be initialized in O(1). Additionally, each group contains two arrays of four cells each, initialized to zero. The other internal fields of the group are of type int. Therefore, initialization of a new group takes time O(1+2*4) = O(1). To insert the group into the AVL tree m_groupsByID, the tree is searched according to the group's ID number in time O(log(m)), where m is the number of groups in the system, as shown in class. Rotations and rebalancing of the tree is done in time O(log(m)) as previously explained. Therefore, in total, the time it takes to add a new group to the streaming system is O(log(m) + 1) = **O(log(m)).**

remove_group –

In this function, a group is removed from the streaming system. Firstly, the validity of the input is checked. If the group ID given is less than or equal to zero, INVALID_INPUT is returned. If a group with the given ID does not exist in the system, FAILURE is returned. Otherwise, the AVL tree m_groupsByID is searched and a pointer to the requested group is returned. From there, using the pointer to the relevant group, the AVL tree containing the group's members is accessed. For each member, the number of group views in each genre is added to the user's views of that genre. In this way, the total number of views for each user is updated according to the number of group views that user participated in. This action is done by going over two arrays of four cells each in tandem. Each member's internal field pointing to the group they belong in is reset to nullptr. After properly removing each user from the group, the group is removed from the AVL tree containing all the groups in the system, m_groupsByID. SUCCESS is returned.

<u>Time complexity:</u>

Firstly, a search in the AVL tree m_groupsByID is performed, in time O(log(m)), as shown in class. Afterward, each node of the AVL tree of the group's users is gone over, updating the users' fields as explained above. For each user, the updates take time of O(4) = O(1) because it requires going over each cell of an array of length four, and updating the value of a single pointer. Therefore, removing the users from the group takes time of $O(4n_{groupUsers}) = O(n_{groupUsers})$. Removing the group from the AVL tree m_groupsByID requires time of O(log(m)), where m is the number of groups in the system. This is due to the requirement to search the AVL tree for the relevant group, rebalance the tree, and update the height and balance factor of the nodes along the search path. In total, removing a group from the system takes time of: **$O(\log(m)+n_{groupUsers})$.**

add_user_to_group –

In this function, a user is added to a group in the system. Firstly, the validity of the input is checked. If the user ID or group ID given is less than or equal to zero, INVALID_INPUT is returned. If there is no user with the given user ID, or no group with the given group ID, FAILURE is returned. If the user already belongs to a group, FAILURE is returned. Otherwise, the user and group given are found through a search according to their ID numbers in the AVL trees m_usersByID and m_groupsByID. A pointer to the user and group are returned from the search. The group's AVL tree of users is accessed through the pointer, and the user is inserted into the tree. The array of the group's total views is updated; the user's views are added to the group's array. Additionally, the group's group views are subtracted from the user's views. From here on, each time the user's views are returned, they are added to the group's group views. In this way, each time the group watches a movie, the group's members' views are updated accordingly. If the user is a VIP user, the number of VIP users the group has is increased by one. The number of users in the group is increased by one. Finally, the user's pointer to its group is updated. SUCCESS is returned.

<u>Time complexity:</u>

Searching for the user in the AVL tree m_usersByID takes time of O(log(n)) as shown in class. Similarly, searching for the group in the AVL tree m_groupsByID takes time of O(log(m)). Insertion to the group's AVL tree of users takes time of $O(\log(n_{groupUsers}))$. Insertion requires searching the tree, rebalancing, rotating, and updating the balance factor and height of the nodes along the search path, thus taking time of $O(\log(n_{groupUsers}))$. Because each user can belong to only one group at most, the number of group's users is bound by the number of users in the system. Therefore, the time it takes to insert a user into the group's tree of users is $O(\log(n_{groupUsers})) = O(\log(n))$. Updating the user's and group's views takes time of O(4) = O(1) because it requires going over two arrays of size four in tandem. Updating the group's other fields, along with the user's pointer to its group takes time O(1). In total, the time it takes to add a user to the group in the system is O(2*log(n) + log(m)) = **O(log(n) + log(m)).**

user_watch –

In this function, a user watches a movie in the system. Firstly, the validity of the input is checked. If the user ID or movie ID is less than or equal to zero, INVALID_INPUT is returned. If there is no user or movie that matches the given IDs in the system, FAILURE is returned. Otherwise, pointers to the user and movie are found through a search in the trees m_usersByID and m_moviesByID accordingly. If the movie is only available for VIP users and the user is not VIP, FAILURE is returned. Otherwise, a view is added to the movie's views and to the user's views in the appropriate genre. The movie is then removed from the two AVL MultiTrees it appears in (m_moviesByRating and its genre's tree) and reinserted, adjusting its position according to its new data. SUCCESS is returned.

<u>Time complexity:</u>

Searching for the user in the tree m_usersByID takes time O(log(n)) and searching for the movie in the tree m_moviesByID takes time O(log(k)) as shown in class. Updating the movie's views and the user's views takes time O(1) each. Because each movie can only be in one genre, the number of movies in each genre's tree is bound by the total number of movies in the system. Therefore, removing the movie from two AVL trees takes time of O(2log(k)) as explained in the remove_movie function. Inserting the movie from two AVL trees takes time of O(2log(k)) as explained in the add_movie function. In total, user_watch takes a time of O(log(n) + 5log(k)) = **O(log(n) + log(k)).**

group_watch –

In this function, a group watches a movie together. Firstly, the validity of the input is checked. If the group ID or the movie ID is less than or equal to zero, INVALID_INPUT is returned. If there is no group or movie that matches the given ID, FAILURE is returned. Otherwise, pointers to the group and the movie are found through a search in the AVL trees m_groupsByID and m_moviesByID accordingly. If the movie is only available for VIP users, and the group is not a VIP group (contains at least one VIP member), FAILURE is returned. If the group doesn't have any members, FAILURE is returned. The views of the movie are updated according to the number of members the group has, and the group's group views are increased by one according to the movie's genre. Finally, the movie is removed from the two AVL MultiTrees it appears in (m_moviesByRating and its genre's tree) and reinserted, adjusting its position according to its new data. SUCCESS is returned.

<u>Time complexity:</u>

Searching for the group in the AVL tree m_groupsByID takes time O(log(m)), and searching for the movie in the AVL tree m_moviesByID takes time O(log(k)) as shown in class. Increasing the views of the movie and the group's views takes time O(1), as it is done through direct access to the relevant arrays through pointers of the objects. Each movie can only belong to one genre, and therefore the number of movies in each genre's tree is bound by the number of total movies in the system. Therefore, removing the movie from the two AVL trees takes time O(2log(k)) as explained in remove_group. Reinserting the movie to each of the two AVL trees takes time O(2log(k)) as explained in add_group. Therefore, in total, group_watch takes time O(log(m) + 5log(k)) = **O(log(m) + log(k)).**

get_all_movies_count –

In this function, the number of movies in each genre is returned. If the genre given is NONE, the total number of movies in the system is returned. Otherwise, the AVL tree containing the movies of the given genre is accessed, and the tree's field m_numOfNodes is returned, along with SUCCESS.

<u>Time Complexity:</u> Each tree is accessible in O(1) because they are internal fields of the streaming system. The number of nodes in the tree is updated during each insertion and removal, and therefore can be directly returned in time O(1). In total, this function takes time **O(1).**

get_all_movies –

In this function, a given array is filled with the ID numbers of all the movies in the system in a given genre. If the given genre is NONE, all the movies in the system are returned. The movies are ordered according to rating, views, and ID number. Firstly, the input is checked: if the array given points to nullptr, INVALID_INPUT is returned. Otherwise, the relevant AVL tree (MultiTree) is accessed according to the given genre. The MultiTree is organized in the desired order, so the movies are inserted into the given array according to their order in the tree. This is done using a helper function that goes over each tree in an opposite inorder walk. Finally, SUCCESS is returned.

<u>Time complexity:</u>

Inserting the movies into the array is done through an opposite inorder walk on the nodes of the MultiTree. Each insertion is done in O(1), and there are a total of k_{genre} movies in the tree. Therefore, if the given genre is NONE, the time it takes to return the movies is **O(k).** Otherwise, the action takes a time of **O(k_{genre}).**

get_num_views –

In this function, a user's views of movies in a certain genre are returned. If the given genre is NONE, the user's total views are returned. Firstly, the validity of the input is checked. If the user ID given is less than or equal to zero, INVALID_INPUT is returned. If there is no user with the given user ID in the system, FAILURE is returned. Otherwise, a pointer to the user is found by searching for the user in the tree m_usersByID. Once found, the user's views are returned according to the given genre:

- If the given genre is NONE, the cells in the user's views array are summed.
- Otherwise, the cell representing the relevant genre is accessed.
- If the user belongs to a group, the group's group views in the relevant genre are added to the user's user views. That value is returned. If the user does not belong to a group, the value received from the user's views array is returned.

<u>Time complexity:</u>

To return the user's views, the user is first searched for in the AVL tree m_usersByID, in time O(log(n)) as shown in class. Calculating the user's views in the worst case requires summing the values held by two arrays, each of length four. Therefore, this takes a time of O(8) = O(1). In total, returning the user's views takes a time of O(log(n) + 1) = **O(log(n)).**

rate_movie –

In this function, a user rates a movie in the system. Firstly, the validity of the input is checked. If the user ID or movie ID is less than or equal to zero, INVALID_INPUT is returned. Similarly, if the rating given is less than zero or greater than 100, INVALID_INPUT is returned. If there is no movie or user in the system that matches the given IDs, FAILURE is returned. Otherwise, a pointer to the user is found through a search in the AVL tree that contains all the users in the system, m_usersByID. A pointer to the movie is found through a search in the AVL tree that contains all the movies in the system, m_moviesByID. If the movie is only available for VIP users and the user is not a VIP user, FAILURE is returned. Otherwise, the rating is added to the movie's rating field. The movie is then removed and then reinserted from the two AVL MultiTrees it is in, m_moviesByRating and the AVL MultiTree relevant to the movie's genre. SUCCESS is returned.

<u>Time complexity:</u>

Searching for the user in the AVL tree m_usersByID takes a time of O(log(n)) as shown in class. Similarly, searching for the movie in the AVL tree m_moviesByID takes a time of O(log(k)) as shown in class. Adding the rating to the movie is done by adding the given rating to the sum of the ratings held by the Movie class and increasing the number of ratings by one. Thus, adding the rating takes a time of O(1). As explained in previous functions, removing the movie from the two AVL MultiTrees it belongs to requires time O(2log(k)) (because the number of movies in each genre is bound by the number of movies in the entire system). Similarly, reinserting the movie to each of the two AVL MultiTrees requires time O(2log(k)). In total, the action takes a time of O(log(n) + 5log(k) + 1) = **O(log(n) + log(k)).**

get_group_recommendation –

In this function, the top-rated movie in a group's favorite genre is returned. Firstly, the validity of the input is checked. If the given group ID is less than or equal to zero, INVALID_INPUT is returned. Otherwise, a pointer to the group is found through a search in the AVL tree m_groupsByID. If there is no group with the given ID, FAILURE is returned. If the group does not have any members, FAILURE is returned. The group's favorite genre is then found by comparing the group's total views in each genre and taking the maximum. In the event of two genres with maximum views, the genre with the lower int value is chosen. Each AVL MultiTree holds a pointer to the largest node in its tree that is updated during insertion and deletion. If the relevant genre does not have any movies in it, FAILURE is returned. Otherwise, the ID number of the "maximum" movies in that genre is returned, where "maximum" is determined according to rating, views, and ID number. SUCCESS is also returned.

<u>Time complexity:</u>

To find the group, the AVL tree m_groupsByID is searched in time O(log(m)) as shown in class. To find the group's favorite genre, the maximum value held by the group's total views array, m_totalViews, is found. This is done in a "for" loop. The array is of length four, so the time it takes is O(4) = O(1). Once found, the "maximum" movie is found through the MultiTree's pointer to the maximum node in time O(1). In total, the time it takes to get a group's recommendation is O(log(m) + 1) = **O(log(m)).**