Mivne Wet 1

Data Structures - 234218

Ori Razon - 204903397
Gabriel Uram - 941200156

We will analyse the complexity of each of the following functions, and prove that their complexity meets the requirements.

streaming_database():

This function only calls the default constructors of the trees which are private members of the streaming_database. There is a constant number of trees (8), and constructing an empty tree takes constant time. Therefore this function also takes constant time:

O(1)

virtual ~streaming_database():

Since we implemented a destructor for our "tree" class, and since the only data members of our streaming database are trees, this destructor will call ~tree() for every tree. ~tree() calls the recursive function destroyTree, passing a pointer to the root node of the tree as a parameter. During this traversal, the destroyTree function visits each node once, executing commands of constant time (deleting the node). Therefore, for a tree with n nodes, ~tree() has a time complexity of O(n). It follows that ~streaming_database() is of time complexity O(n) + O(m) + 6*O(k), since every "genre" tree has at most k nodes. This will then simplify to:

O(n+m+k)

StatusType add_movie(int movieId, Genre genre, int views, bool vipOnly):

Besides performing a fixed number of catch and if operations, O(1), the function calls twice the member function *void insert(nodeType& data, keyType key)* which is part of the class tree, which is implemented of the DataTypes.h file (one call is to insert the movie in the tree with all movies and the other to insert in the tree with the movies of its genre). Let's show that the complexity of insert function is O(logn), where n is the amount of movies in the tree. This function makes O(1) operations, calls *node<nodeType, keyType>* rightLeaf(node<nodeType, keyType>* Node)*, which is a recursive binary search function to get to the righmost leaf (complexity O(logn)), and calls a recursive function called *node<nodeType, keyType>* insert_recursion(node<nodeType, keyType>* Node, nodeType& data, keyType key).*

In each iteration of the recursive function, the complexity is O(1), since it performs a fixed amount of O(1) operations (including the rotate functions). Since the function performs a binary search (since it is an AVL tree) by recursively calling itself until finding the correct destination of the node that will be inserted, it calls itself O(logn) times. Therefore we get that the total complexity of the recursive function is O(logn). As a result, the complexity of the *insert* function is also O(logn) and, finally, the complexity of the *add_movie* is O(logk), as required.

StatusType remove_movie(int movieId):

The behavior of this function is similar to the previous function, but we will analyze it for the sake of certainty.

The function calls some O(1) functions such as *catch* and *if* a fixed amount of times. In addition to that, it calls twice the function *void remove(keyType key),* a member function of the class tree, once for each tree, just like in the previous function. Let's look at this *remove* function to find its complexity.

It calls the *rightLeaf* function, which we showed in the last function that its complexity is O(logn) and calls a recursive function called *node<nodeType, keyType>* remove_recursion(node<nodeType, keyType>* Node, keyType key),* that performs a fixed amount of O(1) operations in each iteration (including the rotate functions). This function performs a recursive binary search to find the correct node to be removed (O(logn) times). Therefore, the complexity of the recursive function is O(logn), so the complexity of the *remove* function is also O(logn) and, as a result, the complexity of the remove_movie function is also O(logk).

StatusType add_user(int userId, bool isVip):

The behavior of this function is equal to the function *add_movie*, since both of them perform a few O(1) operations and then call the same function *void insert(nodeType& data, keyType key)*, which is a template function. Therefore, their complexity is the same: O(logn).

Note: it calls the *insert* function only once instead of twice, but it doesn't affect the complexity.

StatusType remove_user(int userId):

The behavior of this function is equal to the function *remove_movie*, since both of them perform a few O(1) operations and then call the same function *void remove(nodeType& data, keyType key)*, which is a template function. Therefore, their complexity is the same: O(logn).

Note: it calls the *insert* function only once instead of twice, but it doesn't affect the complexity.

StatusType add_group(int groupId):

Exactly the same as in the function *add_user*, so same complexity O(logm).

StatusType remove_group(int groupId):

To remove the group , it works exactly the same as in the function *remove_user* O(logm), but in addition to that, we have to remove the ID of the group from the each user from the group O(n_{group}) , so the overall complexity is:

O(logm+n_{group}).

StatusType add_user_to_group(int userId, int groupId):

As we proved in the lectures, since our AVL tree is always balanced with every insert and remove operation, the height of the tree is logarithmic. It follows that the findNode function for a tree with N nodes is of time complexity O(logN), since it only traverses down the tree and thus will at most visit one node for a given “node-height”. The operations involved in finding which node to go to next and ultimately returning the required node are all of constant time. All in all, finding the correct user node is of time complexity O(logn), finding the correct group is O(logm), and adding the user to the group is also O(logn) - since a group has at most all the users in it (n), and it calls the same insert function which we already proved is if log(n) complexity. The remaining operations in the function are of constant time. All in all we get 2*O(logn) + O(logm) which simplifies to:

O(logn + logm)

StatusType user_watch(int userId, int movieId):

In the beginning, we have to fund thne movie and the user in their respective trees. As we can perform binary searches on the tree, the complexity of each one of these searches is O(logn) and O(logm), respectively. Then, we check if the user can watch the movie, meaning, if the movie is VIP, so must be the user. If the user can’t watch the movie, we return Failure. If he can, we increase the number of views of the user, the movie, and the genre O(1). Therefore the total complexity is O(logn + logk).

StatusType group_watch(int groupId, int movieId):

This function is very similar to the previous one, the only differences are that instead of finding the user, we find the user, so complexity O(logm) instead of O(logn), and that we add the views for the amount of people in the group (we have a field with this number in each group). So the complexity is O(logm + logk).

Note: we don’t iterate through each user in the group, since we already have a field with the amount of users in the group, so there is no need to iterate through each one of them.

output_t < int > get_all_movies_count(Genre genre):

Our streaming_database has a tree for each genre as a private member, and within each tree there is a private member which tracks the number of nodes in the tree - incrementing by one whenever a node is added. As a result, get_all_movies_count simply accesses that field in the relevant genre tree and returns its value in constant time:

O(1)

StatusType get_all_movies(Genre genre, int * const output):

We traverse through the tree of the relevant genre and insert it into the array in “descending” order. Our “insertDescendingOrderRecursion” function visits every node in the appropriate genre tree exactly once, the remaining operations are of constant time, and thus, all in all we get:

O(k_{genre}) or O(k) if genre = none

output_t < int > get_num_views(int userId, Genre genre):

Within the data of each node in userTree we hold the number of views of that user in each genre. Thus, all that is left is to locate the user with findNode(), which, as shown previously, is logarithmic with regard to the number of nodes. All other operations are of constant time. All in all:

O(logn)

StatusType rate_movie(int userId, int movieId, int rating):

In order to check if the user and the movie are compatible in terms of VIP status, we must locate both nodes with findNode → O(log(n) + log(k)). Beyond that the rest of the operations are merely changing values of private fields in constant time. All in all we’ll get:

O(logn + logk)

output_t < int > get_group_recommendation(int groupId):

Initially, after checking that the input is valid, we iterate through the group tree in order to find the groupn that is getting the recommendation, with complexity O(logm). Then, with O(1) iterations, we can find the most watched genre of the group, since we keep the views of each genre. To get the movie with the best reviews in this genre is also simple, because we save the address of the rightmost element of each tree, and in the case of the genre trees, it is the best reviewed, so we don’t have to iterate through the tree. Therefore, the complexity of the function is O(logm).