Describing the data structure:

In order to implement the data structure, we used:

- AVL tree for all the users called m_allUsers, where the key is the userID and the data is the user object itself, and it is ordered in a rising manner by the Key
- AVL tree for all the groups called m_allGroups, where the key is the groupID and the data is the statistics of the group, and it is ordered in a rising manner by the Key, in addition to that each group holds an AVL tree of its users, where the key is the userID and the data is a pointer to that user
- AVL tree for all the movies called m_allMovies, where the key is the movieID and the data is the movie object itself, and it is ordered in a rising manner by the Key
- A size 5 array(for all the types of genre and one for None genre) of AVL trees called m_treeArrayByGenre, where the key is GenreTree - a class that orders the movies using the hierarchy defined in the get_all_movies function and the data is the id of the movie that the key represents his relevant stats.
- An array of ints called m_topRatedMovieIDByGenre, which holds the id of the top-rated movie in each respective genre

Explaining the algorithm, proving complexity bounds:

- `streaming_database();`

  Initializes an empty database, creates empty trees through initialization list

  Time complexity: $O(1)$

  Space complexity: $O(1)$

- `virtual ~streaming_database();`

  Algorithm: Deallocates all the memory in the database, deletes the m_allUsers in $O(n)$, m_allGroups in $O(m)$, m_allMovies in $O(k)$, m_groupUsers $O(n)$ in worst case scenario

  Time complexity: $n + m + k + n = O(m + n + k)$

- `StatusType add_movie(int movieId, Genre genre, int views, bool vipOnly);`

  Algorithm: Adds a new movie to the m_allMovies tree, and to the GenreTree it belongs to in the m_treeArrayByGenre, and also to the general GenreTree while making sure the top rated movie is the root of the GenreTree we're adding to, and also updates the m_topRatedMovieIDByGenre array.

  Time complexity: inserting into the m_allMovies is done in $O(logk)$, since it is normal AVL tree insertion, and in the worst-case scenario, all movies in the system are of the same genre so they all belong to the same GenreTree, which means inserting into it is also $O(logk)$, and inserting it to the GenreTree from type None which is also $O(logk)$ and updating the top rated movie array (which holds the top rated movie by genre) so the algorithm works in $logk + logk + logk + logk = 4 * logk = O(logk)$ time complexity in the worst case scenario

  Space complexity: each movie has it's actual value in m_allmovies and 2 pointer (one for his genre tree and one for the full genre tree) so in total the space complexity is 3*k=$O(k)$

- `StatusType remove_movie(int movieId);`

  Algorithm: Removes the movie from m_allMovies tree and from the GenreTree it belongs to in the m_treeArrayByGenre, and also the general GenreTree while making sure the top rated movie is the root(in case the movie we're removing is the top rated movie) using the correct rolls in the AVL like we saw in class

  Time complexity: removing from the m_allMovies is done in $O(logk)$, since it is normal AVL tree removal, and and in the worst-case scenario, all movies in the system are of the same genre so they all belong to the same GenreTree by the movie genre and also from the general genre tree and updating to top rated movie if needed , which means the algorithm works in $logk + logk + logk + logk = 4 * logk = O(logk)$ time complexity in the worst case scenario

  Space complexity: $O(1)$

- `StatusType add_user(int userId, bool isVip);`

  Algorithm: This function is a normal insert to an AVL tree just like we saw in class, so we know it works in $O(logn)$ time complexity

  Time complexity: $O(logn)$ time complexity, like we saw in class

  Space complexity: normal AVL tree - $O(n)$

- `StatusType remove_user(int userId);`

  Algorithm: we want to remove a user from the m_allUsers tree, and if he belongs to a viewing group, we want to remove him from that group as well and reduce his views from the total group views.

  Check if the user has a group(bool as a private field), if he does then we access that group through a ptr in the user in $O(1)$ time complexity, update the viewing statistics for the user and the group, and then we delete from the m_groupUsers tree in $O(logn)$ worst case and then we delete from m_allUsers $O(logn)$

Time complexity: removing user from m_allUsers + finding the group he belongs to(if) + removing the user from the group tree in m_groupUsers + updating group statistics:

$O(logn)$ + $O(1)$ +$O(logn)$ + $O(1)$ = $O(logn)$

Space complexity: $O(1)$

- `StatusType add_group(int groupId);`

  Algorithm: we want to add a new group into the database, so we want to insert a new node into the m_allGroups tree which is standard insertion into an AVL tree which we saw in the lecture are performed in $O(logm)$ time complexity

  Time complexity: Adding a node to m_allGroups = $logm = O(logm)$

  Space complexity: the group has its actual value in m_allGroups = $O(m)$

- `StatusType remove_group(int groupId);`

  Algorithm: we want to remove a group from the system, meaning we have to remove it from m_allGroups, and update the statistics for the users in the group accordingly. First, we will find the group in the m_allGroups and make sure we store the statistics before removing it from the tree using the remove we saw in the lecture. We will update the statistics for each user that belonged to the group through m_groupUsers using inorder traversal, and then we remove the group from m_allGroups

  Time complexity: removing from m_allGroups + updating statistics =

  $logm + n_{groupUsers} * 1 = O(logm + n_{groupUsers})$

  Space complexity: since we used an inorder traversal over all the groupUsers nodes, we have a space complexity of $O(n_{groupUsers})$

- `StatusType add_user_to_group(int userId, int groupId);`

  Algorithm: first, we find the user in the m_allUsers tree, then we find the group in m_allGroups and make sure to synchronize the user's viewing statistics, and we insert the user into the group's user tree in m_groupUsers using the insert we saw in the lecture, worst case scenario all users belongs to one group.

  Time complexity: finding the right user + finding the right group + updating statistics and fields + inserting into user tree = $logn + logm + 1 + logn = O(logm + logn)$

  Space complexity: $O(1)$

- `StatusType user_watch(int userId, int movieId);`

  Algorithm: we find the correct user and movie in their respective trees, make sure that the user can watch the movie and then update the statistics of the user, movie and the group in case the user is a part of one, then we will update the GenreTree of the movie and of the general GenreTree, by deleting the old movie before watching the movie, and then insert them back into the same trees with updated statistics

  Time complexity: finding the user + getting the user's group pointer +finding the movie + updating statistics + removing movie + adding movie =

  $logn + 1 + logk + logk + 1 + logk + log(k) = O(logn + logk)$

- `StatusType group_watch(int groupId,int movieId);`

  Algorithm: we find the correct group and movie in their respective trees(two trees each), make sure the group can watch the movie and then update the statistics of the group(not the users) and movie accordingly, then we update the GenreTree of the movie in case anything changed in the hierarchy of the movies by deleting the old movie before watching the movie, and then insert them back into the same trees with updated statistics

Time complexity: finding the group + finding the movie + updating statistics + removing movie + inserting movie =

$logm + logk + logk + 1 + logk + logk = O(logk + logm)$

Space complexity: $O(1)$

- `output_t<int> get_all_movies_count(Genre genre);`

  Algorithm: return the size of the tree corresponding to the genre we get as an argument

  Time complexity: access the right index in the m_treeArrayByGenre = $O(1)$

  Space complexity: $O(1)$

- `StatusType get_all_movies(Genre genre, int *const output);`

  Algorithm: using an inorder traverse over the right GenreTree in m_treeArrayByGenre, we go over the entire tree and create an array of all the IDs of the movies in the hierarchy given to us in the function description.

  Time complexity: access the right GenreTree + traverse the tree =

  $1 + k_{genre} = O(k_{genre})$

  Space complexity: $O(k_{genre})$

- `output_t<int> get_num_views(int userId, Genre genre);`

  Algorithm: we find the user we want to calculate the views of the m_allUsers, then we check if he has a group so that we can calculate his viewership statistics correctly.

  Calculating the user's views:

  $\#userViewsByGenre[genre] - \#viewsOfGroupBeforeJoining[genre] + \#viewsAsGroup[genre]$

  (this data exists inside the user's group in case he belongs to one)

  Time complexity: finding the right user in the tree + calculating viewership statistics =

  $logn + 1 = O(logn)$

  Space complexity: $O(1)$

- `StatusType rate_movie(int userId, int movieId, int rating);`

  Algorithm: we find the right user and movie in their trees(3 trees for the movie) and we increase the total rating and rater count of the movie, before making a temporal copy of the movie, removing it from the GenreTrees using the remove we saw in the lecture it belongs to and inserting them like we saw in the lecture with updated statistics in case the hierarchy of the movies changed after rating the movies

  Time complexity: finding the user + finding the movie in 3 trees + updating statistics + removing movie from 2 trees + adding movie to 2 trees =

  $logn + logk + 1 + 2 * logk + 2 * logk + logk = O(logk + logn)$

  Space complexity: $O(1)$

- `output_t<int> get_group_recommendation(int groupId);`

  Algorithm: we find the group in m_allGroups and then we look for the genre with most views by this group

  Time complexity: finding the group in m_allGroups + finding out the favorite genre of the group = $logm + 1 = O(logm)$

  Space complexity: $O(1)$