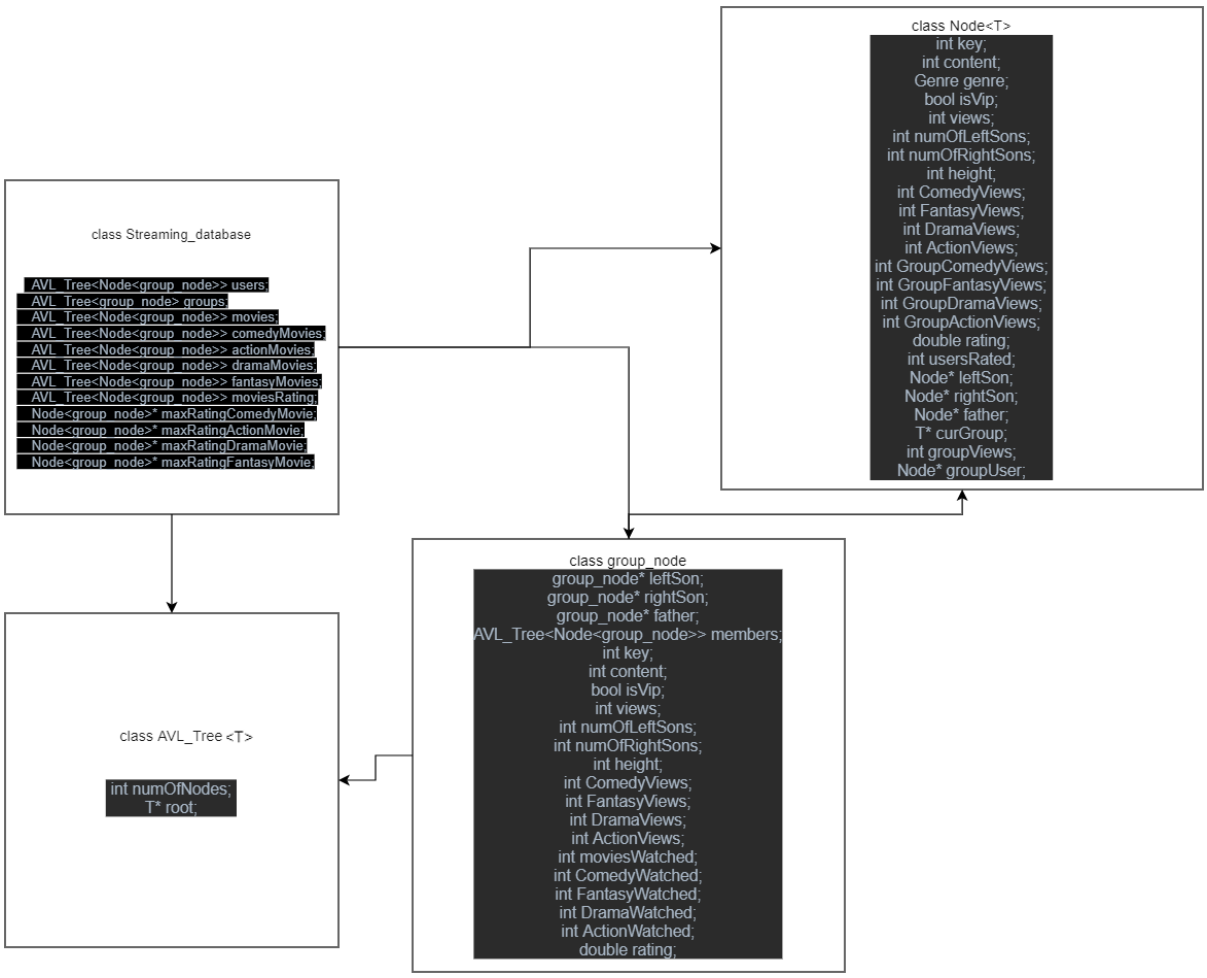
Dry 1 part for wet1:

We used avl tree data structure instances all around the project implementation, first of all we will take a look at the memory needed for the project to do that here is a sketch of the program classes with their member variables (without functions)



As we can see we need memory for 8 trees in streaming database alone, tree for users need n memory space(needs O(n) memory), tree for groups need m memory (needs O(m) memory), two trees for movies need 2k memory space (each needs O(k) memory one sorted according to movieId the other according to rating then views then movieId), the last four trees are for each movie Genre so need k memory space(the sum of their needed memory is equal to needed memory for movies tree which is O(k)).

Then in group_node class we have a tree called members for each group which includes its members according to instructions each user can only be in one

group at a time so the maximum memory needed that is if all users are in group is n (O(n) memory).

All the other member variables need constant memory (don't envolve k,m,n), so the total memory needed for the project is :

n+m+2k+k+n = 2n+3k+m $\leq 3(n+m+k)$

2n+3k+m =O(n+m+k) (n_0 = 1, c = 3) hence the needed memory complexity.

For the time complexity we will review one function at a time and prove that time complexity holds.

First off the initialiazation function **streaming_database():**

As said earlier we have 8 trees now here we initialize them all which takes 8 actions to do so, then initialize the remaining 4 pointers (that point to the movie with highest rating in each genre) to nullptr which takes 4 actions so in all we need 12 actions, 12 = O(1).

Virtual ~streaming_database():

Here we only need to delete every tree we created in the streaming_database, according to lecture deleting a tree that has s nodes requires O(s) time complexity so we have to delete a tree containing n nodes , two trees containing k, a tree containing m nodes, and 4 trees containing k nodes altogether , so for a,b,c integers we need a*n + b*k + c*m actions if we chose d >a,b,c then a*n+b*k+c*m < d(n+k+m) so the time complexity is O(n+m+k)

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

To add a movie we need to create three new node containing the movie's information, and add a node to movies tree which sorts according to movieId, and a node to rankedMovies tree and a node to one of the four genres tree which sorts according to rating then views then movieId, according to lecture we need O(log(k)) time complexity to add a node to a tree of maximum size k , ***[after that we search again for movie with most rating/views/movieId in one of Genre trees according to genre then assign to it the pointer maxRatinggenreMovie (according to genre) which also needs O(log(k)) time complexity (this step is needed everytime we change something in the Movies tree so ill call it from now on "update maxRatinggenreMovie" each time it needs O(log(k)) time complexity all the same cause it functions like search in***

a tree just we always go to the right son)] ,so in all we need O(log(k)) time complexity.

StatusType remove_movie(int movieId):

To remove a movie we use a function to search for the movie in a tree (like the one in lecture which needs O(log(k)) time complexity given the tree is of size k)

After we found the pointer to the movie we need to delete, we delete it from the movies tree, ratedMovies tree and from one of the Genre trees according to the movie's genre (to delete from tree of maximum size k just like in lecture we need O(log(k)) time complexity, then we need to update maxRatinggenreMovie which as stated earlier needs O(log(k)) time complexity, so in all we need O(log(k)) time complexity.

StatusType add_user(int userId, bool isVip):

To add a user we need to create a user Node assign given information to it, and add to the users tree which costs us O(log(n)) time complexity according to lecture (knowing users tree is of maximum size n).

StatusType remove_user(int userId):

We simply search for the node containing userId in the users tree and delete it, just like stated in lecture this costs O(log(n)) time complexity knowing the tree is of size n.

StatusType add_group(int groupId):

We create a new group_node and give it give groupId and add it to groups tree just like in lecture this costs O(log(m)) time complexity knowing groups tree is of size m.

StatusType remove_group(int groupId):

To remove a group we first search in groups tree for group_node containing groupId required to delete, the search costs as usual O(log(m)) time complexity given groups tree is of maximum size m, after finding the group to delete we need to remove all its members first so we iterate through the members tree (each group contains a members tree containing its members) in the iteration we change the total views of each member and the views of each genre in the

following way: (ill use the members of the classes that I already put above in the sketch using a pseudo code to demonstrate how I keep the views)

Member->groupUser->views = member->groupUser->views
+ group->moviesWatched – member->groupUser->groupViews

Where groupUser is a pointer to the user kept in users tree, groupViews is how much group watches were done without the user (before he joined else its 0), and accordingly moviesWatched keeps the value of how much group watches were done till now, we do this for views ,FantasyViews ,ComedyViews , ActionViews and drama views…

The iteration itself costs us according to the number of nodes in the members tree which is $n_{groupUsers}$ so its costs $O(n_{groupUsers})$ so the total time complexity required is $O(\log(m)+ n_{groupUsers})$.

StatusType add_user_to_group(int userId, int groupId):

We need first to search for the user in the users tree using the userId which like stated above costs O(log(n)) time complexity, then likewise we need to find the group using the groupId in the groups tree which also like before costs O(log(m)) time complexity, now if the user is in a group we return Failure, else we assign user->curGroup = group(assign a pointer to group to access groupViews in O(1)), if he is vip the group becomes vip, we assign to the user the number of current groupWaches already done before the user joined (of all genre), then we create a new Node<group_node> called groupUser copy all user information to it and assign groupUser->groupUser = user (a pointer from the member of the group to the user data in users tree to ease changing both), we then add the user views of all genre to the group view of all genre, All this is unaffected by size of n,m,k and costs us O(1) time complexity then the final time complexity is O(log(n) + log(m))

StatusType user_watch(int userId, int movieId):

First like always we find the user using userId in the users tree that costs us O(log(n)) time complexity, then we need to find the movie in the movies tree which also like before costs us O(log(k)) time complexity, then we increment the views of the user, movie, moviegenreviews and if the user is in a group we access the group using the curGroup pointer and increment the views and according genre views, all incrementations are done in O(1), after that we create two new nodes with the new movie views and movieId, we delete the

old movie data from moviesRating tree and according genre tree (for example comedyMovies tree) each deletion from tree just like in lecture costs us O(log(k)) time complexity then we insert the newly created nodes in the same trees (inserting also requires O(log(k)) time complexity) then we update maxRatinggenreMovie which like before also costs us O(log(k)) time complexity, so the function costs us O(log(k)+log(n)) time complexity.

StatusType group_watch(int groupId, int movieId):

Here too we need to find the group in groups tree using the groupId action which costs O(log(m)) as usual, same with finding the movie in the movies tree using the movieId that also costs us O(log(k)), now for incrementation first we increment group->moviesWatched++ which tells us how many times the group did group watches since its creation, and group->genreWatched++ according to genre [for example comedyWatche] (we will connect it with how to find user views again in get_num_views like we did in remove_group), then we change movie's views (movie->views +=group->members.getNumberOfNodes()), all incrementing is done in O(1), now like before we create two new nodes with the movie's information we delete the old nodes containing the movies old info from the movieRating tree and genreMovies tree according to movie genre and add the two new nodes to said trees, each deletion/addition action from one of those two trees is of O(log(k)) time complexity according to lecture so all 4 actions are of O(log(k)) time comlexity too, this proccess we do everytime to keep the 5 movie (movieRating,comedyMovies,fantasyMovies,dramaMovies ,actionMovies) sorted according to rating then views then movieId (in AVL_tree class there are two methods to insert and delete an item from the tree one is like stated above and one only according to integer called content which is movieId/userId/groupId in our cases) **[so from now on instead of explaining again why we create two nodes and delete the old ones we will just call it keep the movies trees sorted and we already explained it costs O(log(k)) each time]**, then we need to refind the maxRatingGenreMovie according to genre in the genreMovies tree (for example actionMovies) it also costs O(log(k)) time complexity, so now all in all the function costs O(log(k)+log(m)) time complexity.

output_t < int > get_all_movies_count(Genre genre):

since we already have a tree for all genre we simply return genreMovies.getNumberOfNodes() which returns how much movies is in according to genre (for example comedyMovies.getNumberOfNodes()

if genre is NONE we return movies.getNumberOfNodes since movies contains all movies. So the time complexity is of O(1).

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

Like stated previously we have 5 trees already sorted according to rating then views then movieId, now if genre is NONE we do an inorder iteration of ratingMovies tree and add the movieId of the smallest in the tree (the one on the far left) as the last element in output array and so on until we reach the biggest in the tree we put its movieId in the first element in output array. Since the ratingMovies has k movies in it, the Inorder traversal is of O(k) time complexity, now if genre is other than NONE, then in genreMovies tree there is say k_{genre} movies an inorder traversal on the tree is of O(k_{genre}) time complexity.

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

first we find the user in the users tree using the userId that costs O(log(n)) time complexity, if the user is not in a group we return user->views if genre is NONE user->genreViews else according to genre (we already dealt with user being in a group and the group getting disbanded in remove group), if the user is in a group we have a pointer to his group in user->curGroup and we have how many group watches were done before he joined in user->groupViews total group watches his group is as stated earlier in curGroup->groupWatches. So if genre = NONE we return:

user->views – user->groupViews + curGroup->groupWatches

else without loss of generality genre = Action we return:

user->actionViews – user->groupActionViews + curGroup->actionWatches.

And since we have all the pointers these actions are of O(1) time complexity, so the overall needed time complexity is of O(log(n)).

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

First we find the movie in the movies tree using movieId (costs us O(log(k)) as usual), then we find the user in the users tree (costs O(log(n)), now in the movie node we keep the number of users rated till now in usersRated and average rating value in rating if we multiply [movie->rating]*[movie->usersRated] and then add [+ rating] (the new rating the user wants to rate) then divide by ++(movie->usersRated) (increment by 1 then devide) we get the new average for the movie in O(1), now again we need

to keep the movies tree sorted after changing this movie's rating this will cost us O(log(k)) time comlexity like before then we need to again find maxRatedGenreMovie according to genre which also costs O(log(n)), then the function costs O(log(k) + log(n)) time complexity.

output_t < int > get_group_recommendation(int groupId):

first we find the group in the groups tree using the groupId which costs us O(log(m)) like stated before, in group node we kept actionViews,fantasyViews, dramaViews and comedyViews we find the maximum between those four in O(1), say without loss of generality its actionViews if actionMovies is not empty we return maxRatingActionMovie->content (content is the movieId) since in every function we kept finding maxRatingGenreMovie here its already updated and we can access it in O(1), so this function costs O(log(m)) time complexity.