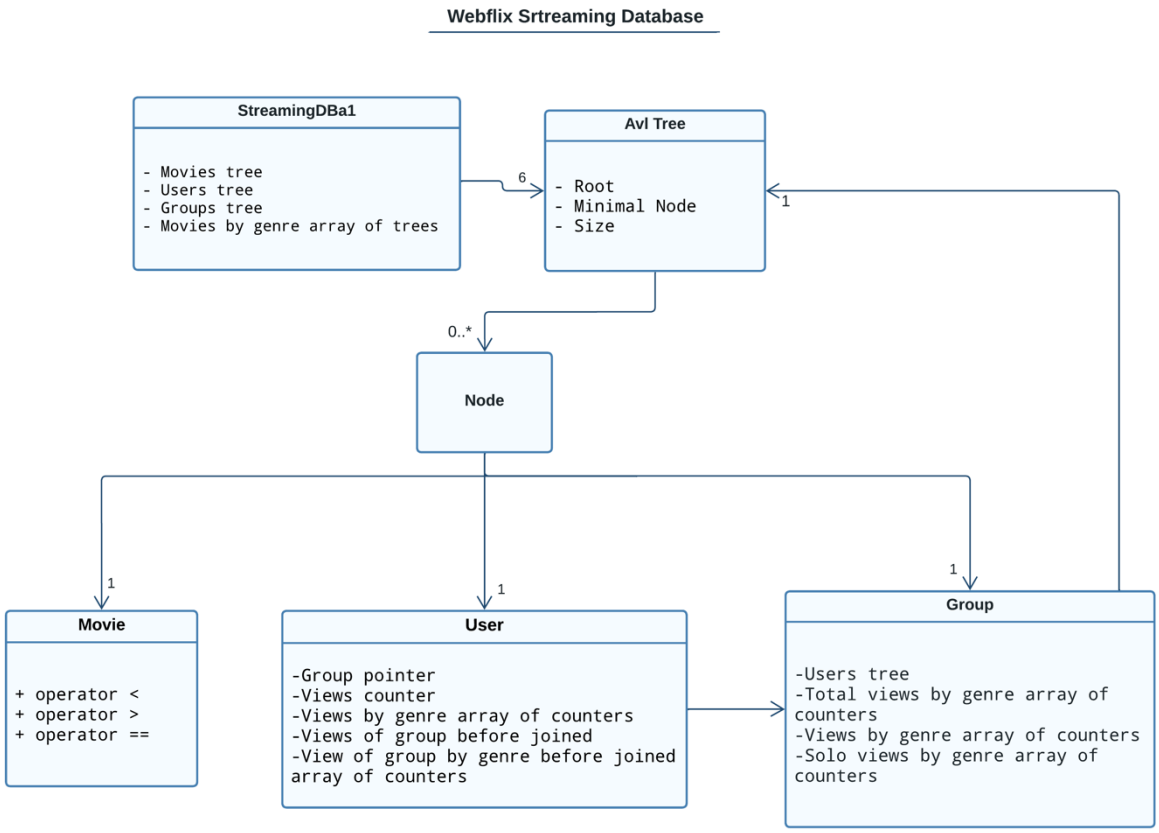
Data structure description:



The data structure consists of the following classes:

Movie

The class represents the movies in the database. Comparison overloaded operators are implemented in the class to allow output of the movies in the requested order.
To allow this output while in-order traversing a tree the comparison is made in somewhat unusual, reversed way for example:
first_movie is considered < (less than) second_movie if:
first_movie_rating > second_movie_rating
OR
first_movie_rating = second_movie_rating AND first_movie_views > second_movie_news
OR
first_movie_rating = second_movie_rating AND first_movie_views = second_movie_news AND first_movie_id < second_movie_id

User
The class represents the users of the database. Among the other fields user instance holds a pointer to the group it is assigned to. And keeps views counters values the group had prior the user's join.

Group
The class represents the groups of users inside the database. Group instance holds an AVL tree of users that has been assigned to it, sorted by their ids. It also holds counter for views that are gained as part of a group as well as views gained in solo watches performed by its users.

Node
This class represent the node that AVL tree holds. It holds all the regular fields like pointers to the left and right children and its height.

Tree
This class implements AVL tree with some modified methods to suite the database needs. Among the other fields it holds a pointer to the **minimal node** which holds the leftmost node in the tree, according to requirements of the database and sorting algorithm its actual purpose is to be able to retrieve the movie with the highest rating in constant time.

StreamingDBa1
The main class of the database. The class holds 3 AVL trees one for storing movies, one for storing users, and one for storing groups, these trees are sorted by unique id of the instances that are stored in them. Additionally, the class holds an array of 5 AVL trees, 4 for each of the movie genres and one that stores all the movies. These trees are sorted by the movie instances and algorithm described in the movie class description above.

Database operations:
Note: In the complexity analysis of the operations on the database, it is important to note that the explanation of "sub operations" within the methods is mostly omitted. This omission is justified by the fact that the arrays of counters used within the classes have a constant size. As a result, complexity of the operations that involve them is constant, in other terms O(1), regardless of the size of the data stored within the overall data structure. Therefore, contribution of these operation to the overall complexity analysis can be considered constant and independent of the input size.

StatusType streaming_database::add_movie(int movieId, Genre genre, int views, bool vipOnly)
- Create a **Movie** object using the provided parameters in O(1).
- Create **node** to store the movie object O(1).
- Attempt to insert the movie into the **movies tree**. As there are O(k) movies in the tree the search for the right spot in the tree takes O(log k), like it was shown in class.

- Insert the movie into the **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. As number of movies in both trees is O(k) therefore the process will also take O (log k).

Overall, for the operation we get O(log k) time complexity.

StatusType streaming_database::remove_movie(int movieId)

- Search for the **MovieNode** corresponding to the given **movieId** in the **movies tree**. The search takes O(log k) like it was shown in class.
- Remove the movie from the **movies tree**. The process takes O(log k) like it was shown in class.
- Using the same process remove the movie from the corresponding **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees in O(log k).

Overall, for the operation we get O(log k) time complexity.

StatusType streaming_database::add_user(int userId, bool isVip)

- Create a **User** object using the provided parameters O(1).
- Create **node** to store the user object O(1).
- Attempt to insert the user into the **users tree**. As there are O(n) users in the tree the search for the right spot in the tree takes O(log n) like it was shown in class.

Overall, for the operation we get O(log n) time complexity.

StatusType streaming_database::remove_user(int userId)

- Search for the **UserNode** corresponding to the given **userId** in the **users tree**. The search takes O(log n).
  If user is in group:
  - update the associated group views counters in O(1).
- Remove the user from the **users tree**. The process takes O(log n) like it was shown in class.

Overall, for the operation we get O(log n) time complexity.

StatusType streaming_database::add_group(int groupId)

- Create a **Group** object using the provided parameters O(1).
- Create **node** to store the group object O(1).
- Attempt to insert the group into the **groups tree**. As there are O(m) groups in the tree the search for the right spot in the tree takes O(log m) like it was shown in class.

Overall, for the operation we get O(log m) time complexity.

StatusType streaming_database::remove_group(int groupId)

- Update the data for each user assigned to the group. As users are stored in tree in group class, we will perform an in-order traverse. The process will take $O(n_{groupUsers})$.
- Remove the group from the **groups tree**. The process will take O(log m) as the time it takes to find the right node like was shown in class.

Overall, for the operation we get $O(\log m + n_{groupUsers})$ time complexity.

StatusType streaming_database::add_user_to_group(int userId, int groupId)

- Retrieve the **UserNode** from the **users tree**. The process of searching takes O(log n) like it was shown in class.
- Retrieve the **GroupNode** from the **groups tree**. The process of searching takes O(log m).
- Copy group's views counters to the user instance O(1).
- Update group's views counters with user's views O(1).

Overall, for the operation we get O(log n + log m) time complexity.

StatusType streaming_database::user_watch(int userId, int movieId)

- Retrieve the **UserNode** from the **users tree**. The process of searching takes O(log n) like it was shown in class.
- Retrieve the **MovieNode** from the **movies tree**. The process of searching takes O(log k).
- Remove the movie from the corresponding **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the removal process it will take O(log k).
- Update user's views counter O(1).
  If user is in group:
  - Update group's views counter O(1).
- Update movie's views counter O(1).
- Insert the updated movie into the **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the insertion process it will also take O(log k).

Overall, for the operation we get O(log n + log k) time complexity.

StatusType streaming_database::group_watch(int *groupId*,int *movieId*)

- Retrieve the **Group** from the **groups tree**. The process of searching takes O(log m) like it was shown in class.
- Retrieve the **MovieNode** from the **movies tree**. The process of searching takes O(log k).
- Remove the movie from the corresponding **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the removal process it will take O(log k).
- Update group's views counter O(1).
- Update movie's views counter O(1).
- Insert the updated movie into the **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the insertion process it will also take O(log k).

Overall, for the operation we get O(log m + log k) time complexity.

output_t<int> streaming_database::get_all_movies_count(Genre *genre*)

As **tree class** keeps **size** variable the process will take O(1) time.

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

- Perform an in-order traversal on the corresponding **moviesByGenre[genre]** tree while writing the movie ids to the given array.
  If genre=Genre::NONE: the number of movies in the tree is O(k) therefore the in-order process will take O(log k) like it was shown in class.
  Else: the number of movies in the tree is O(k_{genre}) and the in-order process will take O(log k_{genre})

  Overall, for the operation we get O(log k) if genre=Genre::NONE or O(log k_{genre}) if genre!=Genre::NONE time complexity like requested.

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

- Retrieve the **UserNode** from the **users tree**. The process of searching takes O(log n) like it was shown in class.
- Retrieve user's views in corresponding genre O(1).

If user is in group:

- Retrieve the **group**. As we store **group pointer** inside **user class,** we can do it in O(1).
- Retrieve group's views in corresponding genre. As number of views counters inside the **group class** is constant, we do it in O(1).

Overall, for the operation we get O(log n) time complexity.

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

- Retrieve the **UserNode** from the **users tree**. The process of searching takes O(log n) like it was shown in class.
- Retrieve the **MovieNode** from the **movies tree**. The process of searching takes O(log k).
- Remove the movie from the corresponding **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the removal process it will take O(log k).
- Update movie's rating O(1).
- Insert the updated movie into the **moviesByGenre[genre]** and **moviesByGenre[Genre::NONE]** trees. Like in the insertion process it will also take O(log k).

Overall, for the operation we get O(log n + log k) time complexity.

output_t<int> streaming_database::get_group_recommendation(int *groupId*)

- Retrieve the **GroupNode** from the **groups tree**. The process of searching takes O(log m) like it was shown in class.
- Calculate the favorite genre for the group. As the group class holds constant number of views counters the process will take O(1) time.
- Retrieve the top-rated movie of the specified **favorite_genre** from the **moviesByGenre[favorite_genre]** tree. As tree class holds the top-rated node the process will also take O(1).

Overall, for the operation we get O(log m) time complexity.

Space complexity

Movies tree and each of the **moviesByGenre[genre]** trees hold *k* movies at most so the space complexity is O(k)
Users tree holds *n* users at most so the space complexity is O(n)
Groups tree holds *m* groups at most, and group can store *n* users in the worst case, so the space complexity is O(m + n)

Therefore, overall space complexity of the data structure is O(n + m + k)