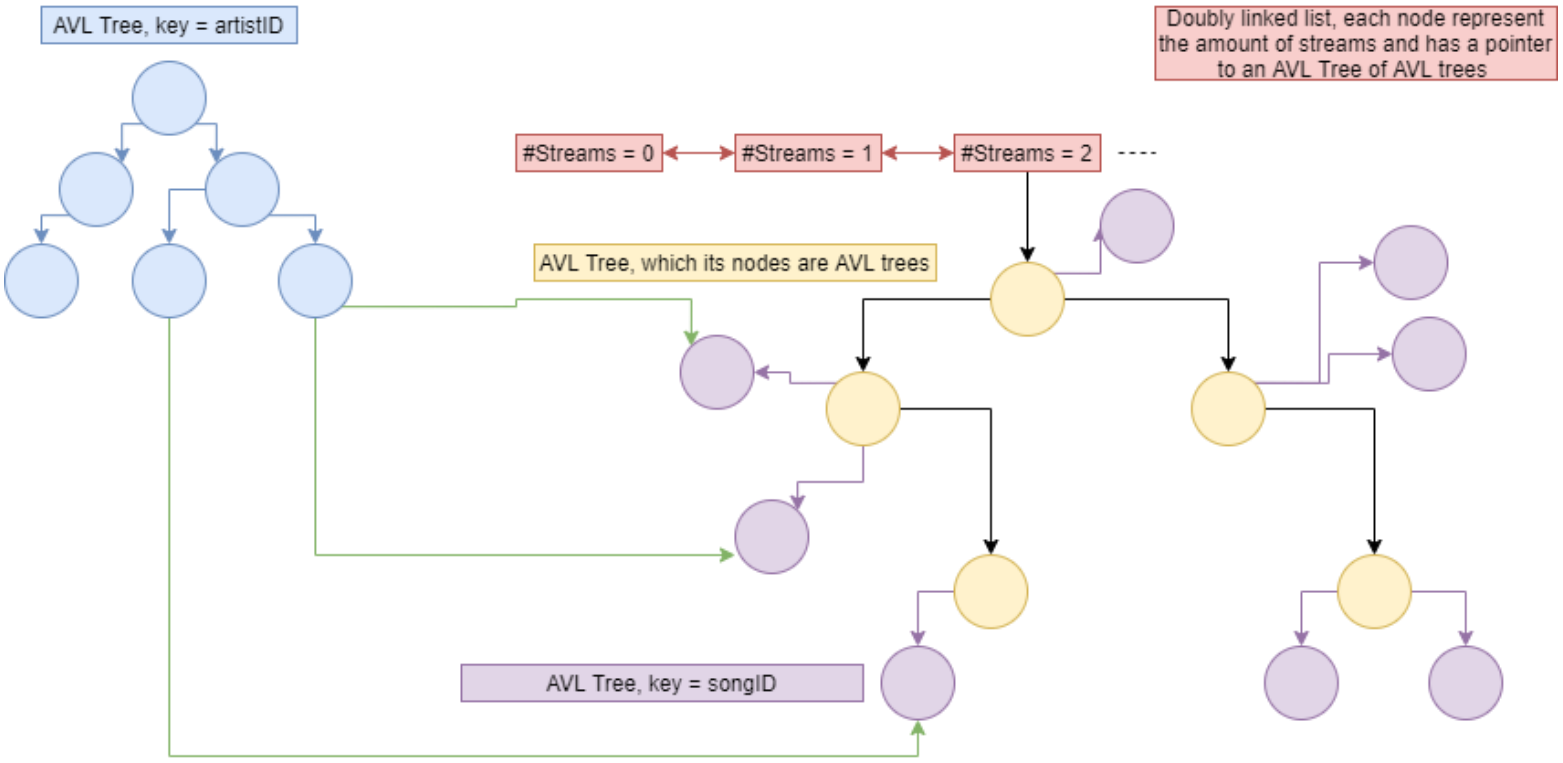
The data structure I have planned consists of two main parts:

- AVL Tree with artistID as it's key. Each node in this tree will have the following attributes:
  1. artistID
  2. NumOfSongs
  3. int array of size numOfSongs; The I'th cell will hold the number of streams of the I'th song.
  4. SongNode array of size numOfSongs; The I'th cell will hold a pointer to the matching SongNode in the second part of this data structure.

- Doubly linked list. Each node in that list annotates "Number Of Streams = *n*" logically and all the songs in our data structure that have *n* streams will be a part of this node.
  Each node of the list holds a pointer to an AVL Tree of AVL Trees of SongNodes.
  The outer AVL Tree holds the artistID and each inner AVL Tree holds the songs ID's of this specific artist that has *n* streams.
  We will maintain a pointer to the head and tail of the list.

  The following diagram should make things clearer.



**Not all green pointers are drawn

For convenience sake from now on I will refer to the first AVL Tree as artistTree, to the Doubly linked list as list, for the outer AVL Tree as outerTree and for the inner AVL Tree as innerTree.

Functions implementations under complexity constraints:

- *Init()* :
  Initialize artistTree as an empty tree in *O(1)*.
  Initialize list with a single node that annotates "Number Of Streams = 0" in *O(1)*.
  Inside that node we created, we will initialize the outerTree as an empty tree in *O(1)*.
  Overall, this function performs in *O(1)*.

- *StatusType AddArtist(void *DS, int artistID, int numOfSongs):*
  First, we will perform a search on artistTree with artistID as key in *O(logn)* where *n* is the number of artists that are already in artistTree.

  At the first call of this function, meaning that when artistTree is empty we will initialize another attribute – minNode as the root we create.
  minNode will hold a pointer to the leftmost node in the tree.

  If artistID wasn't in the artistTree we will create the proper node and add it inside the proper spot found in the search.

  Creating the node includes: (*)
  1. Initializing the int array with a default value of 0 in O(1).
  2. Initializing the SongNode* array with a default value of nullptr in O(1).

  *Note that if we insert our new node left to some other node we will update minNode to the node we have just entered.

  After inserting that node, we might have an unbalanced tree, which we will fix with one rotation in O(1).

  Next, we will create a nearly complete binary tree with numOfSongs nodes and songID as it's key and then fill it with an inorder traversal starting from the leftmost node in O(numOfSongs). This nearly complete binary tree is balanced, hence it's an AVL Tree **(proof?).**

  After creating this tree we will link it's root to the proper outerTree node in O(1).
  To find the proper node to link the tree to we will perform a search in the outerTree looking for artistID, if it's not found we will create this node and link it to the songs tree we have just created in O(logn).

  Assuming we kept a pointer to the artist node after creating it in (*) as a part of the previous operation, as we create the SongNode we will alter the SongNode* array in the artist node.
  For each SongNode with the songID i, we will assign the array[i] with a pointer to that node in O(1) – as a part of the song's tree building which was O(numOfSongs).

  Overall this function performs in *O(2logn + numOfSongs) = O(logn + numOfSongs) = O(logn+m)*.

- *StatusType RemoveArtist(void *DS, int artistID)*:
  First we will search artistID inside artistTree in *O(logn)*.
  If we have found the matching node, we will get an access to one of it SongNodes inside list in O(1) – by accessing the pointer array and then we will get access to the minNode of this tree.

  Next, we will perform a inorder traversal starting from minNode – which is the leftmost node meaning that traversal is done in *O(numOfSongs)* **– (need proof?).**

  Upon arriving at a new node we will delete it and jump to the next one until we are able to delete the entire innerTree and just then we will delete the root of the tree which is inside the outerTree; deleting a node can be done in *O(1)* so deleting numOfSongs nodes while traversing them in the manner I described above can be done in *O(numOfSongs).*

  If as an outcome of the previous deletion operations some node of the list points to an empty tree, we will remove that node and make a link between the previous and next nodes in *O(1).*

  After dealing with the songTree, we get back to artistTree and delete the node we have found in the search we have done before.

  After deleting that node, we might have an unbalanced tree, hence we are performing the proper rotations across the search route in *O(logn).*

  Note that if the node we want to delete is the leftmost node (minNode), we will update minNode to point at the node's parent in *O(1).*
  We can check in *O(1)* whether the node we want to delete is the same node as minNode by comparing their keys.

  Overall this function performs in *O(numOfSongs*logn + logn) = O(mlogn)*
  where m = numOfSongs.

- *StatusType AddToSongCount(void *DS, int artistID, int songID)*:
  First we will search artistID inside artistTree in *O(logn).*
  If we were able to find the matching node, we will increment by one the songID cell inside the int array that store the number of streams of each song which can be done in *O(1).*

  Now, we will have to change the proper SongNode location inside list.
  Using the pointer in SongNode* array inside the artistNode we have found in the beginning of this function we will jump to the proper SongNode inside list.

  Assuming that the number of streams of songID was n, we will search if there's a node inside list that annotates "Number Of Streams = *n+1*".
  The search that I described above can be done in *O(1),* we simply jump to the next node inside list and check whether it's key equals to *n+1*.

  If such node does exist, we will have to determine whether artistID is inside the outerTree.
  We will perform a search for that key in *O(logn).*
  If such node was found, we will insert the SongNode we accessed before to that innerTree and delete it from it's previous location which can be done in *O(logm)* where *m = numOfSongs*.

  If we weren't able to find the artistNode we will have to create one, meaning that we will build a tree with *1* node – which is the SongNode we are about to insert there.
  Inserting this tree can be done in *O(1)* because we can take advantage of the search we did in the previous step – we know where to insert that tree.

  If we didn't find the node inside list that annotates "Number Of Streams = *n+1*", we will create it and link it to the current node we are in and to the next one in *O(1).*

  After creating this node – it's pointer to the AVL Tree of AVL trees is nullptr, hence there is no search that's need to be done, we know for sure that we won't find artistID there, so we simply create the proper tree – one with a single node which will contain the songID of the SongNode we are moving.

  Lastly, we will update the pointer from the artistNode we found at the beginning to point to the new location of the SongNode we had moved in *O(1).*

  Overall this function performs in *O(logn + logm).*

- *StatusType NumberOfStreams(void *DS, int artistID, int songID, int *streams):*
  We will search for artistID inside artistTree in O(logn).
  If were able to find the proper node we will return the value inside the songID cell inside the int array in O(1).
  Overall, the function performs in O(logn).

- *StatusType GetRecommendedSongs(void *DS, int numOfSongs, int *artists, int *songs):*
  Starting from the tail of list, we will access the outerTree in O(1).
  We will perform an inorder traversal starting from the minNode of the outerTree.
  For every node in the outerTree we will perform an inorder traversal starting from the minNode of it's innerTree.
  Upon searching the entire innerTree of all outerTrees in a specific node of the list we will jump to the previous node of the list and preform the same traversal.

  The loop I described above stops whenever we have filled both of the arrays given – meaning that we have already traversed numOfSongs nodes.

  As we know traversing in an inorder manner from the leftmost node takes O(Number of nodes we traverse), hence we can determine that this function performs in O(numOfSongs).

  The traversal I described will output at the songs in the following order:

  - Most streams (by starting from the tail of the list).

  - If streams are equal, ascending order of artistID (by starting from the minNode of the outerTree which has the lowest artistID).

  - If artistID is equal, ascending order of songID (by starting from the minNode of the innerTree which has the lowest songID).

- *<u>void Quit(void **DS):</u>*
  We will call the destructors of list and artistTree.

  To destruct a tree we will traverse its nodes in a postorder manner and delete the current node each time.
  We will start our postorder traversal from the minNode so the operation will cost O(n).

  Now, we will destruct our list.
  In order to do so, we will start to scan the list from head, and delete the Tree of Trees using the same destructor we have used before – again starting from minNode.

  The number of nodes we are traversing equals to the amount of all songs in the data structure, hence we can perform the destruction of list in O(m).

  Overall, this function performs in O(n + m).