Data Structures

Wet Assignment 2

Data Structures Explanations:

ListNode- A generic (template) ListNode that is made up of:

T data* – a pointer to the data type of the list node

ListNode next*- a pointer to the next list node in the chain.

List- A generic (template) chain of ListNodes. It is made up of:

ListNode head* – pointer to the head of the list.

int length – length of the list.

The last ListNode in the list points to nullptr.

HashTable- This is a Hash Table using chain-hashing. The hash table is generic (template). It is made up of:

*List<T>** table* – an array of list pointers that keep the values. The hash function assignes each value a space in the method of "chain-hashing". The Lists are of the same type we want our hash table to be.

Int arraySize – size of the array of table.

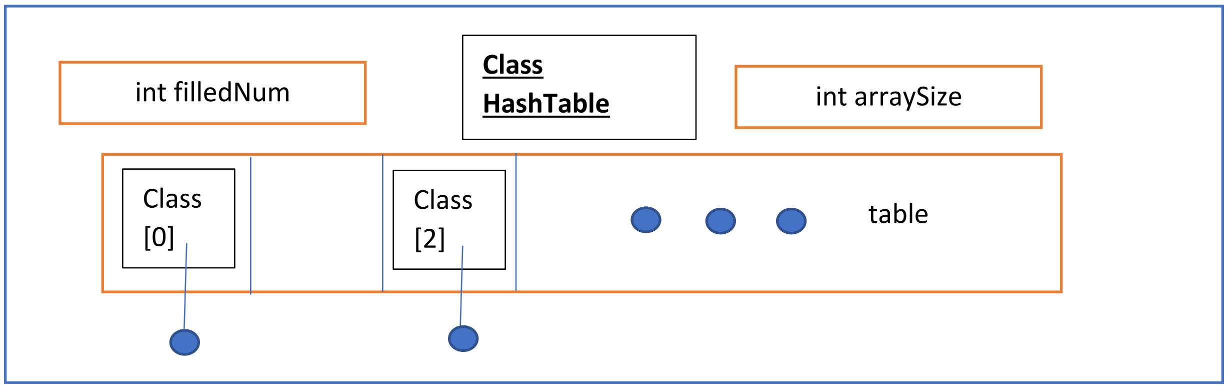
Int filledNum – number of items that are in the hash table.

The array in the hash table is a dynamic array.

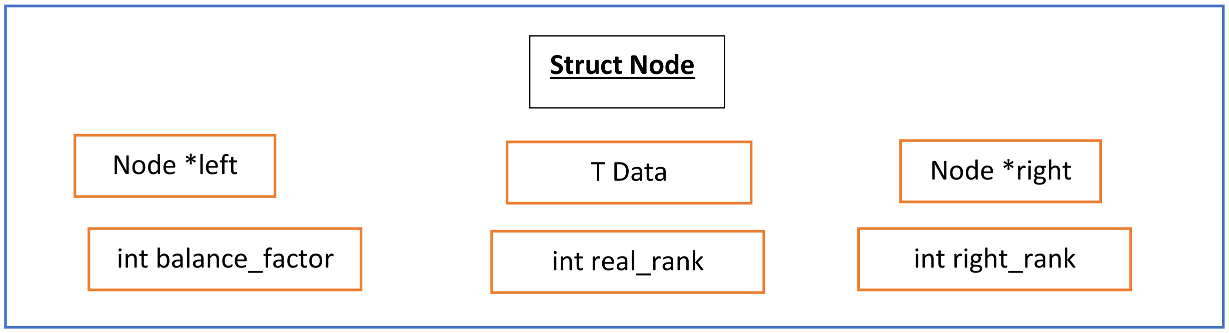
During insert, once the ratio of filledNum/arraySize reach a predetermined threshold the array doubles itself in size and re-hashes all the values already inside the array to new hash values.

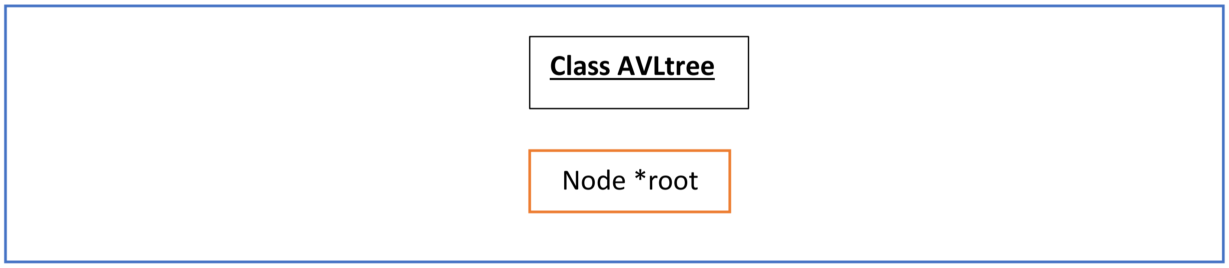
During delete, once the ratio of filledNum/arraySize reach a predetermined threshold the array decreases its size so the new size will be arraySize/2 and re-hashes all the values already inside the array to new hash values.

It supports Insert, Remove, and Search Functions.



AVL Rank Tree- This is an AVL rank tree structure that is generic, meaning the data that it holds is template and can be anything. The nodes of the rank tree have a real rank (which is the rank of the node like we learned in class) and a right_rank, which is the size of the right subtree plus the node itself. The AVLRanktree itself is made up of a pointer to *struct Node* that is the root. The *struct Node* has fields, *T Data *left, *right, *parent, balance_factor, real_rank, right_rank.*





Class- The Class class is a class that keeps the Class type. It is made up of

Int courseID – the course ID of the associated course.

Int classID – ID of the class

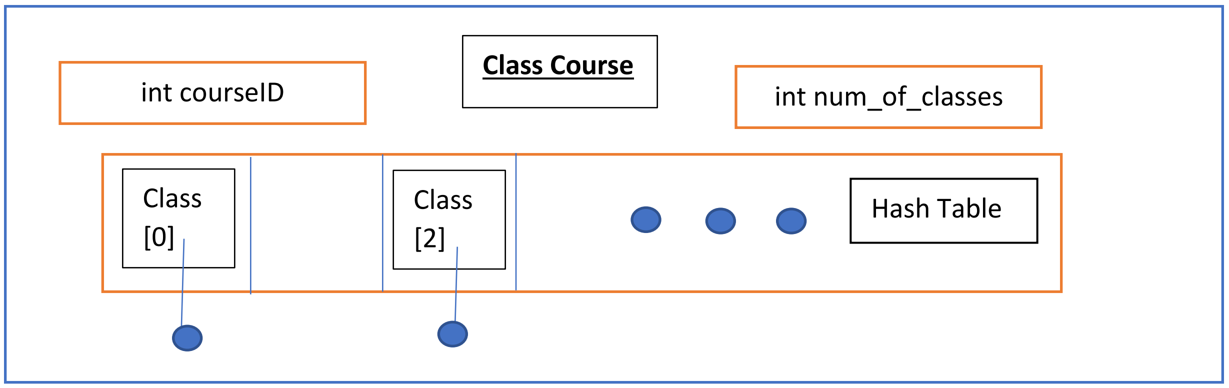
Int time – the watch time of the class.

Course – The course class is a class that keeps the course. It is made up of

int courseID

int num_of_classes – how many classes are in the course

HashTable<Class> classes- Hash Table that is of type Class and keeps all the classes that are associated with the course.



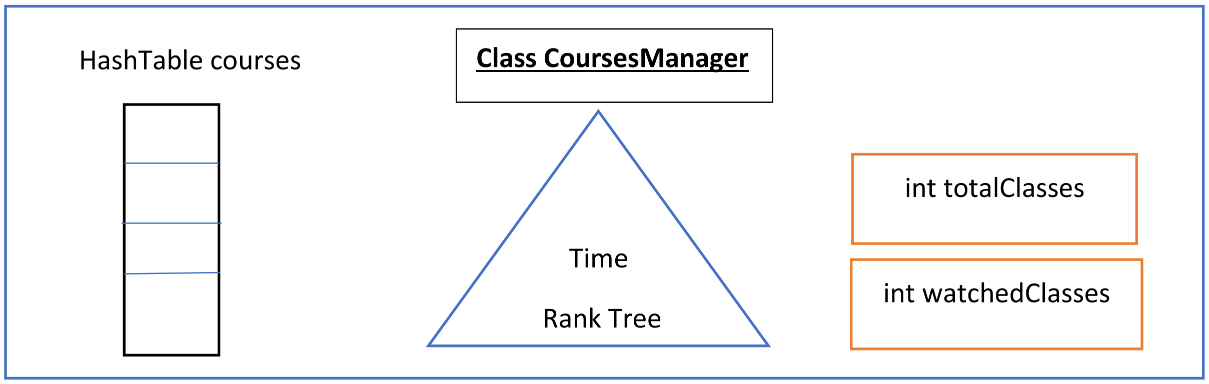
CoursesManager- the CoursesManager class is the main class that *library.h* works with. It includes:

HashTable<Course> courses – a hash table field that holds all the initiated courses.

AVLRanktree<Class> timeTree – an AVLRanktree field that hold all the classes from all the courses that have been watched at least 1 minute.

int totalClasses – the total number of classes in the structure.

int watchedClasses – the number of classes that have been watched at least 1 minute.



Besides all these data structures, we have an Exceptions class that helps us throw exceptions to help us know what is wrong with the inputs, allocations, etc…

<u>Proof of Time Complexity</u>

Let us now show the proof that we manage to get the program to run by the required time complexity. We will show for each function individually.

<u>Void *Init</u>

Required complexity: $O(1)$

The function calls the required constructors for a HashTable and a AVLRankTree that initiate them as blanks. The time complexity for initiating an empty AVLRanktree is $O(1)$. The time complexity for initiating a HashTable is $O(Const)$ where $Const$ is the default starting size of the hash table array. In total: $O(1) + O(Const) = O(1) + O(1) = O(1)$.

<u>StatusType AddCourse (void *DS, int courseID)</u>

Required complexity: $O(1)\ on\ average$

This function inserts a Course type into the courses HashTable. Inserting into a HashTable is $O(1)\ on\ average$. When we create a new course, we also initiate an empty HashTable for the classes which takes $O(1)$. In total, the time complexity is $O(1)\ on\ average$.

<u>StatusType RemoveCourse(void *DS, int courseID)</u>

Required complexity: $O(mlog(M))$ *on average*

(m is the number of classes in the course, and M is the number of classes in the system)

Firstly, for every class in the course we must remove the class from the AVLRankTree (if it exists there). Worst case scenario is that every class exists in the time tree which means that we will have to remove m classes from the AVLRankTree. Removing one class in that case would be $O(\log M)$. Doing this m times means that the time complexity would be $O(mlog(M))$.

Secondly, we would need to remove the course from the courses HashTable. This is $O(1)\ on\ average$. Therefore, we found that the time complexity is $O\big(mlog(M)\big)\ on\ average$.

<u>StatusType AddClass(void *DS, int courseID, int* classID)</u>

Required complexity: $O(1)$ $on\ average$

First we find the desired course in the HashTable: $O(1)$ $on\ average$. Then we add the desired class into the Classes HashTable which is also $O(1)$ $on\ average$ and return a pointer to that class number. In total: $O(1)$ $on\ average$.

<u>StatusType WatchClass(void *DS, int courseID, int classID, int time)</u>

Required complexity: $O(\log(M+2))$ $on\ average$

(M is total number of classes in the system)

This splits into 2 cases: whether the class previously had $t = 0$ or not.

<u>t = 0</u>

If there is no time in the class then we need to add the time to the class. We first need to find the course: $O(1)$ $on\ average$. Then we need to find the desired class in the Classes HashTable and change the timeWatched parameter: $O(1)$ $on\ average$. Then we need to insert that new class into the time tree: $O(\log M + 2)$ (the +2 is only so that the argument in the log won't be zero). In total, when t=0, we get: $O(\log M + 2)$ $on\ average$.

<u>t != 0</u>

If there is time, then we still need to update the time of the class in the courses HashTable which takes $O(1)$ $on\ average$. The thing that is different now is that we have to remove the node of the particular class from the time tree: $O(\log M + 2)$ and then insert a new one into the time tree: $O(\log M + 2)$. In total: $O(1)$ $on\ average + O(\log M + 2) + O(\log M + 2) = O(1)$ $on\ average + O(\log M + 2) = O(\log M + 2)$ $on\ average$.

We see that in both cases, the time complexity is $O(\log M + 2)$ $on\ average$.

<u>StatusType TimeViewed(void *DS, int courseID, int classID, int *timeViewed)</u>

Required complexity: $O(1)$ $on\ average$

To see how much a class has been watched we find the correct course in the courses HashTable: $O(1)$ $on\ average$. Then we search for the correct class in the classes HashTable: $O(1)$ $on\ average$. In total the complexity is: $O(1)$ $on\ average$.

StatusType GetIthWatchedClass(void *DS, int i, int *courseID, int *classID)

Required complexity: $O(\log(M+2))$

(M is total number of classes)

We will search through the watched classes rank tree with an algorithm similar to the one we saw in the tutorials. The most time the algorithm takes is the height of the tree, which is $\log(m)$ where m is the total number of classes that have been watched at least one minute. The worst case for this is that every class has been watched so this abides by the time complexity constraints: $O(\log(M+2))$

Proof of Memory Complexity of Entire Structure

Required complexity: $O(n+m)$

In our class we have a HashTree for the courses and inside each course we have a hash table for the classes. We have a rank tree with all the watched classes. We also have two ints.

The memory complexity for the two ints is $O(1)$.

The worst case for our time AVL rank tree is that all the classes are in it. In that case its memory complexity is $O(m)$.

Our courses Hash Table has all the courses and each course has a Hash Table of classes. Because the Course's memory complexity **without** the class HashTable is $O(1)$, (because without the class array it is only two ints) we can think of the space complexity of each course as the number of classes that it has plus a constant. So in total if we add up all the courses it will be m classes and a constant that is a function of n: $O(m+Cn) = O(m+n)$.

In total: $O\big(m+(m+n)\big) = O(n+2m) = O(n+m)$.

Proof of Memory Complexity of Recursion

Required complexity: $O(n+m)$

Whenever we enter a recursive function it is either to search for a node in the AVL rank tree, remove a node from the AVL rank tree, or add a node to the AVL rank tree, we enter the recursive function $O(\log m)$ times, thereby taking up only $O(\log m)$ frames in the stack. That means that we abide by the memory complexity requirements.