Data Structures

Wet Assignment 1

Data Structures Explanations:

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

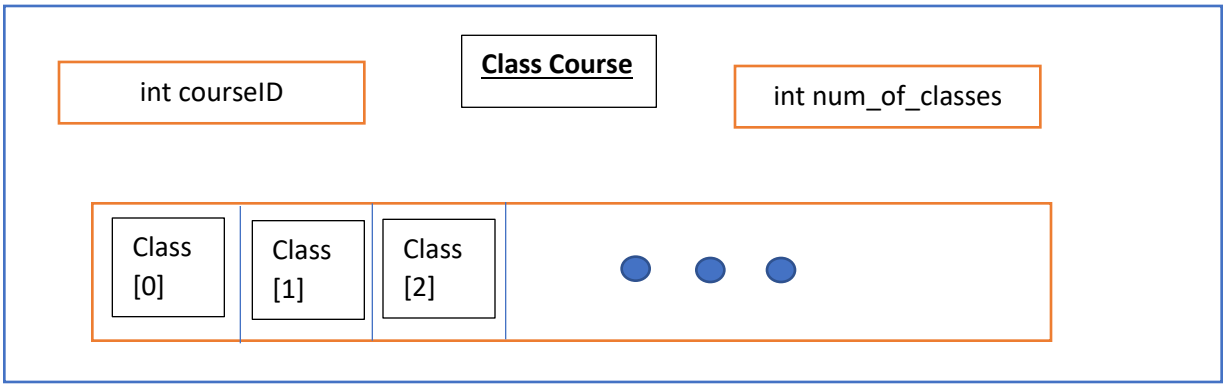
Int time – the watch time of the class.

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

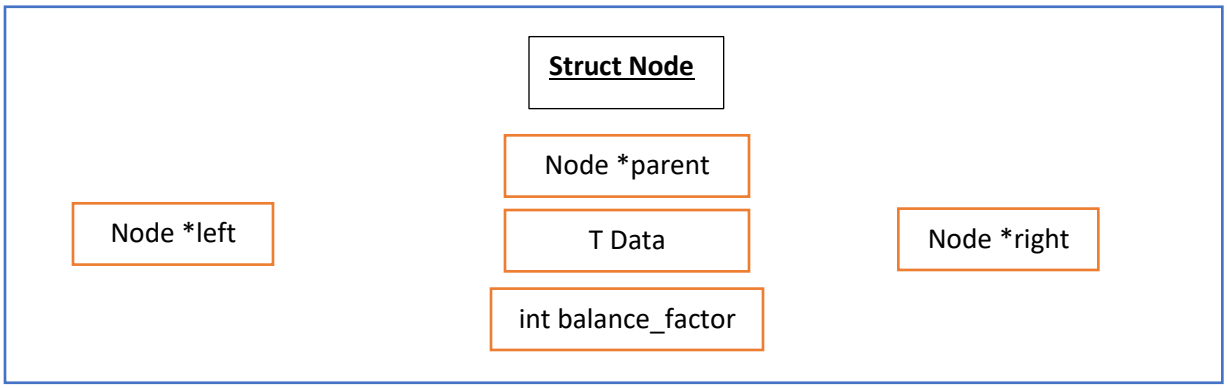
int courseID

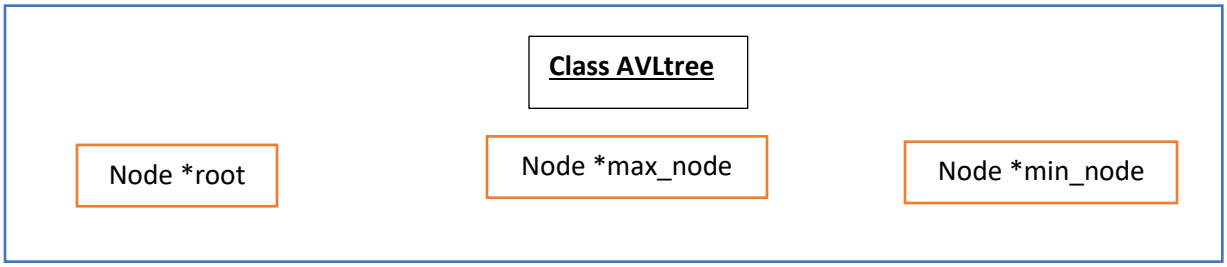
int num_of_classes – how many classes are in the course

Class classes*- array that is of type Class is the size of *num_of_classes* and keeps all the classes that are associated with the course.



AVLtree- This is an AVLtree-like structure that is generic, meaning the data that it holds is template and can be anything. The difference between this structure and a normal AVL tree is that this AVL tree is two way, meaning each node also has a pointer to its parent. Also this AVLtree is made up of pointer to *struct Node* that is the root and it always has a pointer to the biggest and smallest values in the tree. The *struct Node* has fields, *T Data *left, *right, *parent, balance_factor.*





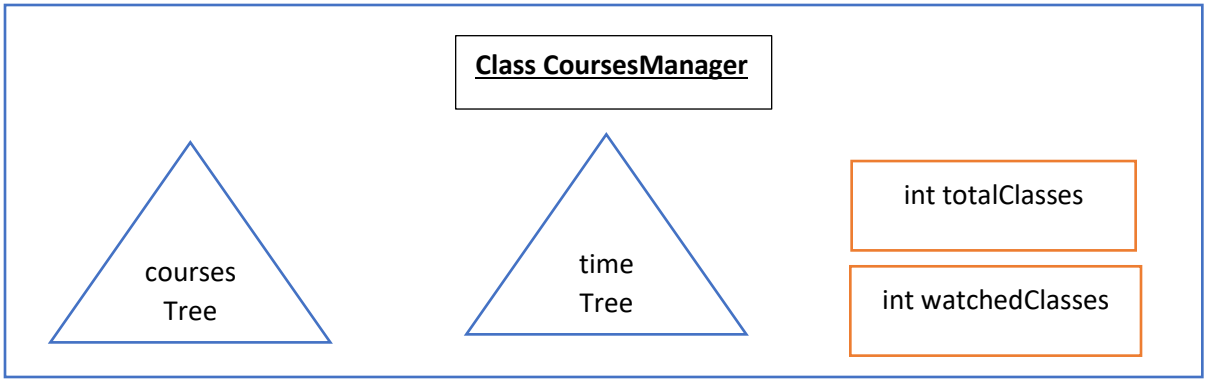
<u>CoursesManager</u>- the CoursesManager class is the main class that *library.h* works with. It includes:

AVLtree<Course> coursesTree – an AVLtree-like field that holds all the initiated courses.

AVLtree<Class> timeTree – an AVLtree-like 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…

Proof of Time Complexity

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

Void *Init

Required complexity: $O(1)$

The function calls the required constructors for the two AVLtrees that initiate AVL trees as blanks. The time complexity for initiating an empty AVLtree is $O(1)$ therefore doing it twice means $O(1) + O(1) = O(1)$

StatusType AddCourse (void *DS, int courseID, int numOfClasses)

Required complexity: $O(\log(n) + m)$

(n is number of courses in system and m=numOfClasses)

This function inserts a Course type into the courses AVLtree. Inserting into an AVLtree is $O(\log(n))$, where n is the number of courses already in the Courses AVLtree. In this new Course type we need to initiate and **set default values** for the different classes inside the Classes array (namely, need to set the time viewed to 0). Doing this for each class is $O(1)$ so $m * O(1) = O(m)$. In total $O(\log(n)) + O(m) = O(\log(n) + m)$.

StatusType RemoveCourse(void *DS, int courseID)

Required complexity: $O(mlog(M))$

(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 course from the time AVLtree (if it exists there). Worst case scenario is that every class exists in the time tree which means that we will have to remove from the AVLtree m classes. Worst case scenario for the time tree is that EVERY class in the system is in the time tree. 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 tree. We know for sure that the total number of courses is smaller than the total number of classes. Removing the course would consist of deleting the Classes array ($O(m)$) and also deleting the course itself: $O(\log n)$.

(n is the number of courses in the system). The second part would therefore take: $O(m) * O(\log n) = O\big(mlog(n)\big) \leq O(mlog(M))$.

Thirdly, we need to establish the new minimum and maximum values in the time tree, if they were the value that we deleted. For this we need to scan the tree again, therefore this would take $O(\log M)$.

In conclusion, removing the course is asymptoticly bound by $O\big(mlog(M)\big)$.

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

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

(M is total number of classes in the system, t is the time we are adding to the class)

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

For both we need to understand that $number\ of\ courses \leq number\ of\ classes$

<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: $O(\log n)$. And change the time watched parameter of that class: $O(1)$. Then we need to insert that new class into the time tree: $O(\log M)$. In total, when t=0, we get: $O(\log n + \log M) \leq O(\log M + \log M\ ) = O(\log M)$.

<u>t!=0</u>

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

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

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

Required complexity: $O(\log n)$

To see how much a class has been watched we find the correct course in the courses tree: $O(\log n)$ and then use the classID given to get to the correct class in the class array $O(1)$. In total the complexity is: $O(\log n)$.

StatusType GetMostViewedClasses(void *DS, int numOfClasses, int *courses, int *classes)

Required complexity: $O(m)$ (m is the number of classes we want to print)

We will divide this into two different scenarios: whether numOfClasses is bigger than the classes we have in the time tree, or not.

numOfClasses <= classes in the time tree:

If this is the case then we do a sort of reverse in-order on the AVLtree time tree. We have a pointer that always points to the maximum node in the AVL time tree so we run on the tree through this until we reach the numOfClasses. The run time for this is entirely dependent on how many classes we want to print. Therefore $O(m)$.

numOfClasses > classes in the time tree

In this case we firstly do a reverse in-order scan on the time tree to "print" out all the classes with t>0.

Afterwards we have a pointer to the minimum value in the courses tree. So we start going through course by course and class by class, until we reach the numOfClasses we wanted. If the class has t=0 we "print" it. If not, then the worst thing that happens is that we scanned the class twice: once when we scanned it in the time tree, and once when we scanned it in the courses tree.

Our worst case scenario for this is scanning everything twice. Therefore $O(2m) = O(m)$.

void Quit(void **DS)

Required complexity: $O(n + m)$

In this function we delete the entire time tree first. Our worst case scenario is that all the courses are in the time tree therefore deleting it is $O(m)$.

Next we need to go over all the classes arrays in the courses tree and delete them which is also $O(m)$. Finally we need to delete all the courses in the courses tree which means $O(n)$.

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

<u>Proof of Memory Complexity</u>

Required complexity: $O(n+m)$

In our class we have 2 AVL trees and 2 ints.

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

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

Our courses AVL tree has all the courses and each course has an array of classes. Because the Course's memory complexity **without** the class array 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: $O\big(m+(m+n)\big) = O(n+2m) = O(n+m)$.