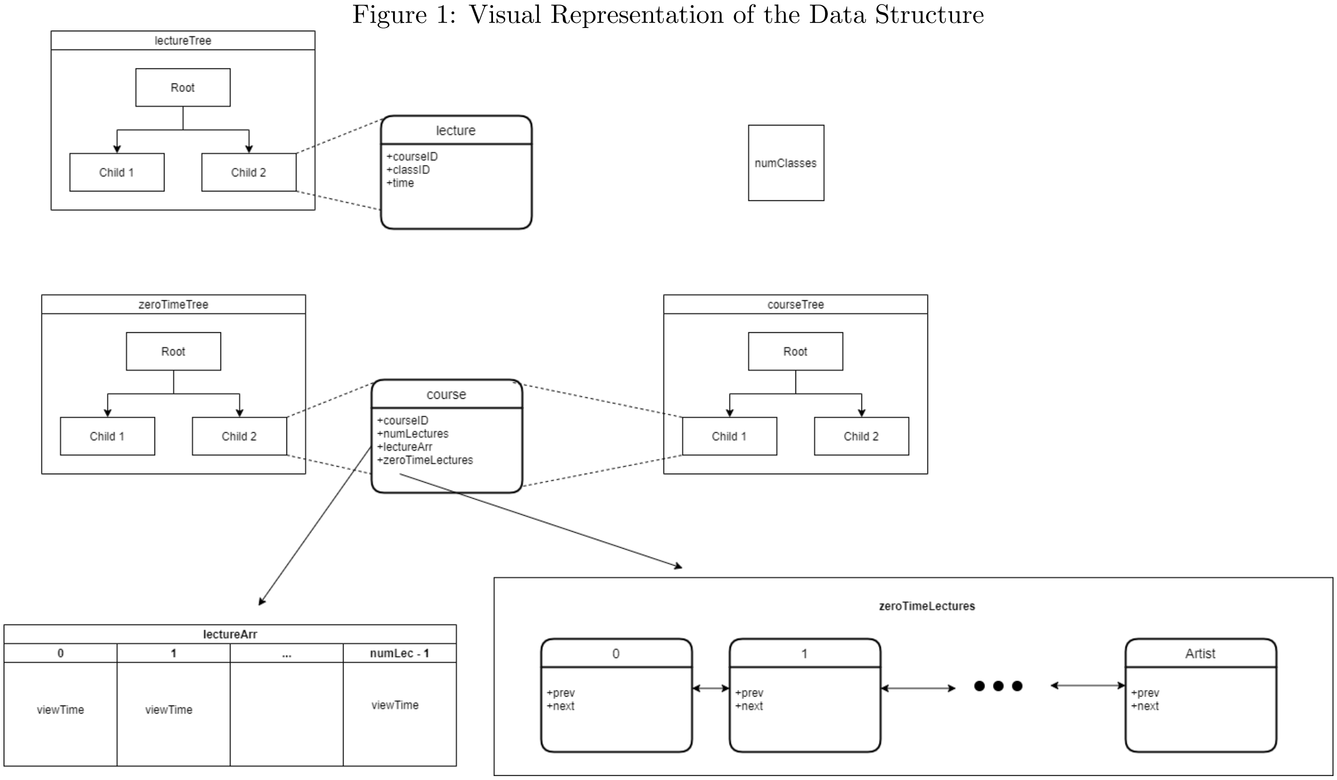
HW1 Wet

Description of the Data Structure:

Figure 1: Visual Representation of the Data Structure



The data structure contains three AVL trees that have additional fields of `min` and `max` to allow for in-order and post-order traversal of k nodes in the tree in $O(k)$ time:

- `lectureTree`: Contains all the lectures of all the courses in the system. Each lecture in the tree maintains exactly three pieces of information: The ID of the lecture, the ID of the course in which the lecture is given, and the total watch time for the lecture. Together, these three pieces of information form the key by which the lectures in the tree are sorted (according to the sort order stipulated in `getMostViewedClasses`): By descending view time, then by ascending course ID and then by ascending lecture ID.

- `courseTree`: Contains all the courses in the system. Is sorted by course ID.

- `zeroTimeTree`: Contains all the courses in the system that have at least one lecture with a watch time of zero. Is also sorted by course ID.

The courses in both `courseTree` and `zeroTimeTree` are represented by objects with which contain four fields:

- `courseID`: The ID of the course

- `numLectures`: The number of lectures in the course

- `lectureArr`: An array conataining a number of cells equal to the the number of lectures in the course. The index of each cell is the ID of the relevant lecture and the contents of the cell is that lecture's watch time.

- `zeroTimeLectures`: An array similar to `lectureArr` except that it holds only the lectures of the course that have a watch time of zero. In addition, it has features of a doubly linked list, meaning that every cell has `prev` and `next` fields that hold the indices of the previous and next non-empty cell. This means thst `zeroTimeLectures` has both the capability of adressing a cell in $O(1)$ time as an array would, as well as the the cabaility of traversing k non-empty cells in $O(k)$ time as a doubly-linked would be able to do.

In addition, the structure maintains a variable `numClasses` which tracks the total number of lectures of all the courses in the system at any given moment.

Space Complexity of the Data Structure

At any moment, there is at most three copies of every lecture in the system ($3m$) - two in the course object (one being in the `lectureArr` and the other in `zeroTimeLectures` of the course) in `courseTree`, and if the watch time of the lecture is zero than two more in the course in `zeroTimeTree` and otherwise, one copy in `lectureTree`. Furthermore, there is always at most two copies of every course in the system ($2n$): one in `courseTree`, and in the case that the course has a lecture with a view time equal to zero, an additional copy in `zeroTimeTree`. Therefore, the total space complexity of the structure is $O(n + m)$.

Implementation of the Data Structure

For all the function that receive parameters, the validity of those parameters is checked. In the case that one or more of the parameters is invlid, `INVALID_INPUT` is returned. This check always only takes a constant amount of time to perform ($O(1)$). Additonally, in all the functions that have a return value, in the case that the function successfuly finnished, it returns `SUCCESS`, which also only takes $O(1)$ time.

Init

`lectureTree`, `courseTree`, and `zeroTimeTree` are all initiallized (as empty), which each take $O(1)$ time. In addition `numCLasses` is initiallized to zero ($O(1)$ time). Therefore, the total time complexity for `Init` is $O(1)$.

AddCourse

1. First, we search `courseTree` to see if the course that is being requested to be added is already in the system. If this is the case, we return `FAILURE`. The size of `courseTree` at any given moment is exactly the n, the number of courses in the system, and therefore this action is a search on a AVL tree of size n, which according to the lecture takes $O(\log(n))$ time.

2. If the course is not yet in the system, we create a course object for the course to be inserted. This consists of initializing `lectureArr`, an array of zeroes of size $m =$ `numOfClasses` which therefore takes $O(m)$ time, initialization of `zeroTimeLectures` also to a size of m which also takes $O(m)$ time, and initialization of `numLectures` to m ($O(1)$).

3. Next, the course object created in the previous step is inserted into `courseTree` and into `zeroTimeTree`, each being an insertion into an AVL tree of size n, and therefore each taking $O(\log(n))$.

4. Finally, `numOfClasses` is added to `numClasses` ($O(1)$).

Therefore in total, `AddCourse` is of $O(\log(n)) + 2 \cdot O(m) + O(1) + 2 \cdot O(\log(n)) + O(1)) = O(\log(n) + m)$ time complexity.

RemoveCourse

1. First, we search `courseTree` to see if the course that is being requested to be removed is actually in the system. If this is not the case, we return `FAILURE`. The size of `courseTree` at any given moment is exactly the n, the number of courses in the system, and therefore this action is a search on a AVL tree of size n, which according to the lecture takes $O(\log(n))$ time.

2. If the course is indeed in the system, for each of its lectures we delete the matching lecture from `lectureTree` (if the lecture is in that tree, meaning if it has a watch time that is greater than zero). At worst this will take $O(m \cdot \log(M))$ time, if all m of the course's lectures have greater than zero watch time, since we are doing m removals from an AVL tree of maximal size M (because at most it cpontains all the lectures in the system) and then deleting each lecture which was removed, each being of $O(1)$ size.

3. Next, if the course has lectures of zero watch time, we delete the course from the `zeroTimeTree`. This involves removal from an AVL tree of at most n (because the tree at most contains all the courses in the system) which therefore takes $O(\log(n)$ time, and then deletion of the course that was removed ($O(m)$ each for deleting `lectureArr` and `zeroTimeLectures` and $O(1)$ each for deleting `courseID` and `numLectures`).

4. In an identical fashion, the course is removed from `courseTree`.

5. Finally, the number of lectures that were in the course that was removed is subtracted from `numClasses` ($O(1)$).

Therefore in total, `RemoveCourse` is of

$$\begin{aligned}
&O(\log(n)) + O(m \cdot \log(M)) \cdot O(1) + 2 \cdot (O(\log(n)) + 2 \cdot O(m) + 2 \cdot O(1)) + O(1) \\
&= O(\log(M)) + O(m \cdot \log(M)) \cdot O(1) + 2 \cdot (O(\log(M)) + 2 \cdot O(m) + 2 \cdot O(1)) + O(1) && \text{(we can assume } n < M) \\
&= O(m \log(M)) && (m = O(m \log(M)))
\end{aligned}$$

time complexity.

WatchClass

1. First, we search `courseTree` to see if the course that is supplied is actually in the system. If this is the case, but the `classID` supplied is not one of the classes of `courseID`, than we return `INVALID_INPUT`. As previously shown, the search for the course in the tree takes $O(\log(n))$, and the further validation of the class being a valid class of the course requires merely checking if `classID` $+ 1 >$ `numLectures` which is only $O(1)$ time.

2. Next, we search `courseTree` to see if the course that is supplied is actually in the system. If this is the not the case than we return `FAILURE`. As previously shown, the search for the course in the tree takes $O(\log(n))$.

3. Next, we check what the watch time of `classID` is by getting the course from `courseTree` and checking `lectureArr[classID]` ($O(\log(n)) + O(1)$).

    - If the watch time is zero, we remove `classID` from `zeroTimeLectures` in the same course object. This invloves marking the applicable cell in `zeroTimeLectures` as empty and then adjusting the `prev` and `next` field in the adjacent cells apporpraitely as is done in a doubly linked list (which is a finite amount of constant time complexity actions: $O(1)$). If after this removal `zeroTimeLectures` is empty, then the course is removed from `zeroTimeTree` ($O(\log(n))$).
    - If the watch time is not zero, then the matching lecture is removed from `lectureTree` ($O(\log(M))$).

4. Next, the `time` that was supplied to `WatchClass` is added to the `time` of the lecture that was removed in the previos step ($O(1)$). The lecture is then re-inserted into `lectureTree` ($O(\log(M))$).

5. Finally, the `time` that was supplied to `WatchClass` is added to the `courseArr[classID]` in the course in `courseTree` ($O(\log(n)) + O(1)$).

Therefore in total, `WatchClass` is of

$$\begin{aligned}
&5 \cdot O(\log(n)) + 2 \cdot O(\log(M)) + 5 \cdot O(1) \\
&= 5 \cdot O(\log(M)) + 2 \cdot O(\log(M)) + 5 \cdot O(1) && \text{(we can assume } n < M) \\
&= O(\log(M)) \\
&= O(\log(M) + t)
\end{aligned}$$

time complexity.

TimeViewed

1. First, we search `courseTree` to see if the course that is supplied is actually in the system. If this is the case, but the `classID` supplied is not one of the classes of `courseID`, than we return `INVALID_INPUT`. As previously shown, the search for the course in the tree takes $O(\log(n))$, and the further validation of the class being a valid class of the course requires merely checking if $\texttt{classID} + 1 > \texttt{numLectures}$ which is only $O(1)$ time.

2. Next, we search `courseTree` to see if the course that is supplied is actually in the system. If this is the not the case than we return `FAILURE`. As previously shown, the search for the course in the tree takes $O(\log(n))$.

3. Finally, $\texttt{courseArr}[\texttt{classID}]$ in the course in `courseTree` is coppied into `timeViewed` $(O(\log(n)) + O(1))$.

Therefore in total, `TimeViewed` is of

$$\begin{aligned} &3 \cdot O(\log(n)) + 2 \cdot O(1) \\ &= O(\log(n)) \end{aligned}$$

time complexity.

GetMostViewedClasses

1. First, we check if $\texttt{numOfClasses} > \texttt{numCLasses}$. If so, then we return `FAILURE` $(O(1))$.

2. Otherwise, we perform an in-order traversal of `numOfClasses` number of lectures in `lectureTree` beginning with the one with the highest key (with the highest view time). At each lecture that is stopped at, its course ID is coppied into the next cell in `courses`, and its class ID is coppied into the next cell in `classes`.

3. If `numOfClasses` is greater than the number of lectures in `lectureTree`, than the following procedure is followed for the remaining number of `numOfClasses` times, beginning with the course with the lowest `courseID` in `zeroTimesTree`, for each course in an in-order traversal:

   (a) For each cell that is not empty in `zeroTimeLectures`, copy the course ID of the current course we are in into `courses` and copy the current index of `zeroTimeLectures` into `classes`.

At every step, either an in-order traversal is being performed on an AVL tree or a linear traversal of a linked list. The actions peformed at every stop are a copy of two pieces of data, which each take $O(1)$ time to perform. The total number of steps is $m = \texttt{numOfClasses}$. Therefore in total, `getMostViewedClasses` is of

$$\begin{aligned} &m \cdot O(1) \\ &= O(m) \end{aligned}$$

time complexity.

Quit

1. First, we delete every lecture in `lectureTree`. The number of lectures in `lectureTree` is at most equal to the total number of lectures in the system, m. Therefore this takes $O(m)$ time to complete.

2. Next, we delete every lecture held in every course in `zeroTimeTree`, as well as all the courses themselves in the tree. Every lecture in the system can appear at most twice in `zeroTimeTree`: once in a `lectureArr` and once in a `zeroTimeLectures`. Therefore, the total number of lectures in the tree is at most equal to the total number of lectures in the system, m, multiplied by two: $2m$, and the total number of courses in `zeroTimeTree` is at most equal to the total number of courses in the system, n. Therefore it takes $O(2m + n) = O(m + n)$ time to delete `zeroTimeTree`.

3. In the identitcal way that we deleted `zeroTimeTree`, we delete `courseTree`. For the identical reasoning, it also takes $O(m + n)$ time.

Therefore in total, `Quit` is of

$$\begin{aligned} &O(m) + 2 \cdot O(m + n) \\ &= O(m + n) \end{aligned}$$

time complexity.