|  |  |
| --- | --- |
| **אוריאל אברהם** | **315479808** |
| **יוסף גורן** | **211515606** |

**The General Architecture:**

Our data structure contains two main parts:

The first part keeps any relevant information of the watched classes in a way that we can adjust each class when needed, and in a time complexity that fits the requirements and the functions implementations. our DS has a ranked tree, in which each node represents a watched class and the classes are arranged by their time, courseID and classID. this help us update, use and arrange our watched classes in a way that fits the exercise's complexity requirements.

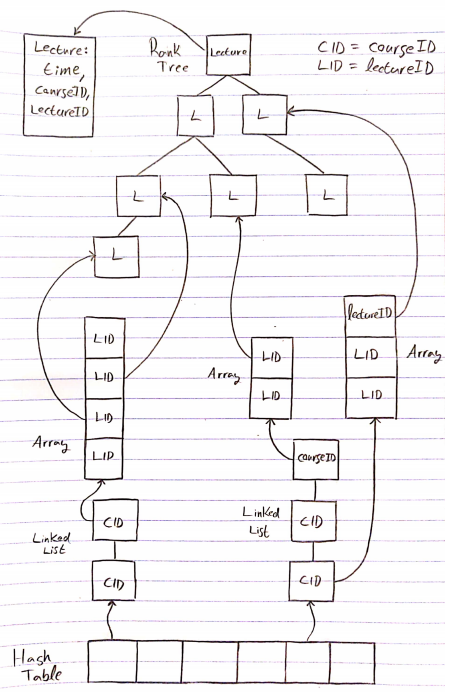
The second part keeps the pointers to each node of a tree, that represents a class, so we have an access to adjust the classes saved in the first part, with a time complexity of O(1) on average, amortized. The way it works is by using a chained hashtable and each one of its nodes consist courseID for key and pointer to array of pointers to the nodes in the ranked tree, so each pointer points to an existing and watched class of a specific course, and if a pointer points to nullptr then the class has not been watched.

The data structure is handled by CoursesManager2 class that supports each function, except Quit, and has fields as mentioned below:

table - a chained hash table in which each node represents a course, and this node has CourseID for key and array of pointers to the course's watched classes as data.

Tree – a ranked tree that keeps an extra information about how many elements there are in its right and left subtrees. Each node represents a watched class and the classes are organized by the time they were watched in an ascending order, and then by their courseID and classID in a descending order.

(The details are visualized in the illustration at the next page):



**The Data Structures we have implemented and used:**

**Dynamic chained hash table:** resembles what we saw in one of the tutorials.

The table is implemented using a dynamic array, so that array[i] point to nullptr when uninitialized, and point to a doubly linked list which consists all the courses with ID so that ID%m = i.(m is the size of the array).

Each node in the doubly linked list represents a course and has the courseID for key and a pointer to array that holds the course's classes for data.

-we are working with pointers for data in the doubly linked list nodes, and do not copy or create an entire array of lectures when we expand the table array, so the complexity that we saw in the tutorial holds in our implementation.

-The array is expanded or shrinked in order to keep the equation:

0.25m≤ n ≤ m(n = number of courses in our DS, m = size of array).

-the chosen hash function is key%m where m is the current size of the array.

**Ranked SearchTree**: just as seen in the course material.

For this exercise we used a rank – search tree, that uses the lectures themselves (what we consider a ‘lecture’ is a triple of: courseID, classID, and time watched variables) as the keys of the tree nodes, and we held an additional rank-variable to allow us to quickly (log(n) complexity) find the i’th biggest node in the tree.

As required: **Lecture A is considered bigger than Lecture B**, in accordance to the following criteria:

* A would be bigger if it’s time variable is bigger, if the time variable is smaller than A is smaller.
* Otherwise, A would be bigger if it’s courseID variable is **smaller**, and if A’s courseID is bigger, than A is smaller.
* Else, we make the same consideration as the one above, but with classID instead of courseID.

This sorting rule is enough to allow us to use the Rank-Tree algorithm/structure as seen in the course material, note we can make the comparison between Lectures in constant complexity.

The rank tree – variable mentioned before, is a variable (each node within the tree holds it’s own value for this variable) that **represents the number of nodes in the sub-tree** in which the node is considered the head.

The reason we can consider this node-variable a ‘rank tree variable’ is simple, **we can define it in a recursive manner**: it’s value is the sum of those of it’s sons plus one, or zero if the node is null. And if the value of our variable in a given node – only depends on this variable’s value in the node’s immediate sons (the value might ‘depend’ on lower nodes, but it can be updated by only looking at the immediate sons, assuming that their subtrees are already updated).

Now in such a case where we have a variable such a this (recursive definition), to see why we can maintain all of the action complexities from the regular search Tree, we will look at all of the reasons to update the variable:

* **Removing a node from the tree:** in this case – in addition to the regular removal algorithm, when we ‘go back up’ in the tree to find all of the places that might need fixes to the AVL-condition, we can also make sure to update our variable (in constant complexity) as explained before therefore we did not change the number of nodes we went through and in each one of them, we still only do actions to the

complexity of O(1), so the total complexity did no change.

* Roll Action: when we do a roll action on nodes, we later want to update our nodes: all we have to do is make the update for each node involved in the roll, in the order from lowest to highest in the new tree, so we made three actions of constant complexity and the complexity of roll actions is still O(1).
* Inserting a node: much like in the part of removing a node, after we insert the node we go all the way back to the head of the tree and update the variable in each node we go through, we have log(n) nodes on the way up so total complexity of insertion did not change either.

The algorithm for finding the i’th biggers item in the rank-tree that we have used, is exactly the one shown in the tutorials: we go down from the head of the tree with a counter of how many nodes we’ve skipped, and skip a sub-tree if we can without our counter surpassing the index we are looking for, eventually we get to a node where the number of nodes bigger than it which we’ve skipped is the same as the index we are looking for. The reason this works in log(n) complexity is simple, we only go through one node from each layer of the tree (in the worst case) and in each one we do operations of constant complexity, therefore the total cannot be more than log(n).

**Functions:**

Note: each of the functions below calls the corresponding function in CoursesManager2 class.

**Init():**

-initialize an empty tree for our ranked tree: O(1).

-initialize an empty array with a constant size for our hash table: O(1).

**Overall: O(1)**

**AddCourse(courseID):**

Insert courseID to the right list in our hash table, with

Data = nullptr because there are no classes in this course yet.

**Overall: O(1) on average, amortized**.

**RemoveCourse(courseID):**

-Find courseID in our table – O(1) on average, amortized.

-iterate over its pointers to classes array, and using each pointer which is not null, access the lecture in our watched lectures tree and delete it. There are m lectures in this specific course and M lectures in our watched lectures tree, so each deletion costs logM and we have maximum of m deletions. – O(mlogM)

-finally we can delete the course's node in the linked list- O(1) on average, amortized.

**Overall: O(mlogM) on average, amortized.**

**AddClass(courseID,\*classID):**

-find the course in our hash table – O(1) on average, amortized.

If the course already has classes, add a new class the the classes array at the top free place and write this place to classID using the pointer argument. O(1)

If it’s the first class in the course, allocate a new empty dynamic array and update

\*classID = 0. This also takes O(1)

**Overall: O(1) on average, amortized.**

**WatchClass(courseID,classID,time):**

-find the course in our hash table – O(1) on average, amortized.

-in the worst case the wanted class has not been watched yet so in the array of classes of this course, we get that array[classID] = nullptr. we need to insert this class to our watched class ranked tree and update array[classID] to point to the corresponding new node in the tree. – O(logM).

**Overall:** **O(logM) on average, amortized.**

**TimeViewed(courseID,classID,\*timeViewed):**

Find the node of courseID in hash table: O(1) on average, amortized.

Access the pointer to the class that represents classID with array[classID] pointer, if it points to nullptr then the class has not been watch,

so we can update \*timeViewed = 0, otherwise return the value of time\_watched that is stored there: O(1).

**Overall: O(1) on average, amortized.**

**GetIthMostViewedClasses(i,\*coureID,\*classID):**

Using Select algorithm that we saw in the tutorial, we can find the Ith most viewed class using select(i) on our ranked tree, and in this node we also have the classID and courseID of the wanted class, so we can update the pointers accordingly: O(logM) as we saw in the tutorial.

**Overall: O(logM).**

**Quit():**

Deleting the ranked tree: there are m nodes(classes) in the tree in the worst case, and deleting each node take O(1) so overall O(m).

Deleting the hash table:

-iterate over the table array and for each array[i], iterate over the list it holds and for each node in the list, delete the classes array in the node.

Iteration costs O(k+n), where k is the size of the table array, and n is the number of nodes in all of the list combined, which is also the number of courses. we know that k = O(n) so this iteration actually costs O(n).

-deleting each classes array costs Mi which is the number of classes in the array, so deleting all of the arrays costs ∑Mi = m = total classes in the data structure. So overall O(m).

-iterate again over the table array to delete all the lists take O(n) and deleting all of the lists is equivalent to deleting n nodes(n courses) so it also takes O(n)

-finally, we can delete the table array and it takes O(k)=O(n)

**Overall(General for Quit):** O(m) +3O(n) = **O(m+n).**

**Space Complexity:**

**Functions:(temporary data allocated to help implementation):**

in each of our functions we don’t allocate new memory at all, or only allocate memory and saving it in our hash table or ranked tree.

**Fields:**

**hashtable:**

-array sized k when k = O(n)

-lists with total nodes as the number of courses = O(n)

-a tree with m nodes in the worst case(all of the classes have been watched) = O(m)

**Overall space complexity:** 2O(n)+O(m) = **O(m+n)**