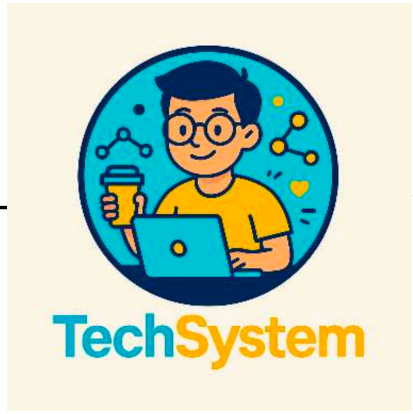
Data Structures Wet 1 - Dry Part

Name	Id
Roee Eldar	207637364
Nadav Oved	211880257

December 17, 2025



AvlTree<Student>
studentMap of
size n

AvlTree<Course>
courseMap of
size m

awardAcademicPoints

Student
global bonus bonusPenalty courseCount completionPoints

enroll/complete

Course
AvlTree<Student ref>
students of size
$n_{courseId}$

1 Structural Overview

The overall TechSystem data structure consists of:

- studentsMap: an AVL tree owning the instances of Students and maps each student id to its corresponding Student instance. Its size is denoted as n.
  - Student: All students share access a global bonus field. In addition, each student has its own bonus penalty, course count, and completion points.
- courseMap: an AVL tree owning the instances of Courses and maps each course id to its corresponding Course instance. Its size is denoted as m.
  - Course: A wrapper encapsulating an AVL Tree of references to students which are enrolled to a given course Id. Its size is denoted by $n_{courseId}$.

The only interface between courses and students is through enroll or complete calls. A student is **not** aware of its courses. A given course is aware of all its students and sends update to them through enroll or complete calls only.

2 Overall Space Complexity

Except the recursive call stack created in each tree's destructor, in any given point in execution, we store:

- size of a Student: $O(1)$, contains only some counters.
- size of a Course: $O(n_{courseId})$, since it contains a single AVL tree with $n_{courseId}$ vertices, in which each vertex contains a reference to a student ($O(1)$ each).
- size of studentsMap: $O(n)$, since this AVL tree has n vertices, each is a Student of $O(1)$ size as mentioned above.
- size of courseMap: $O\left(m + \sum_{id \in \text{course Ids.}} n_{id}\right)$. This is due to the fact that each of its m vertices owns a Course instance, which size, as mentioned above, is $O(n_{courseId})$. So in total, we get:

$$O\left(\sum_{id \in \text{course Ids.}} (1 + n_{id})\right) \underbrace{=}_{|\text{course Ids.}|=m} O\left(m + \sum_{id \in \text{course Ids.}} n_{id}\right)$$

.

Therefore, in total: structure space = size of studentsMap+size of courseMap = $O\left(n + m + \sum_{id \in \text{course Ids.}} n_{id}\right)$. Now, we'll look at each one's recursion depth, which is $O(log(\text{tree size}))$, since the destructor performs post order traversal of a balanced AVL tree.

1. Destructing the studentMap would result in $O(log(n))$ stack frames.
2. Destructing the courseMap would result in $O(log(m))$ stack frames. In this case, we also incur the cost of recursively destructing its tree vertices, which would result in an additional $O(log(n_{\text{id}}))$, but since we only destruct one course at a time, we get in total: $O\left(log(m) + \max_{id \in \text{course Ids.}} (log(n_{id}))\right)$

So, we get that destruction space cost (running destructor 1,2 one at a time) is $O\left(max\left(log(n), log(m) + \max_{id \in \text{course Ids.}} (log(n_{id}))\right)\right)$, which implies total space cost remains $O\left(n + m + \sum_{id \in \text{course Ids.}} n_{id}\right)$ as required.

3 Overall Time Complexity

Note: During enrollCourse(), we take care of not inserting the student's reference to the course's inner tree if it's already there, and we remove students when they unenroll. Therefore, for any given course id, actual size of the course's inner tree is exactly $n_{id} \leq n$.

- techSystems(): default initializes two empty trees: courseMap and studentsMap.
- ~techSystems():
  - releases students tree which is of size n in postorder, therefore $O(n)$ time, as releasing each student is $O(1)$ (simple struct of integers).
  - In same manner, releasing courseMap costs postorder traversal of $O(m)$ nodes, plus an added destruction cost per node of $O(n_{id})$ for freeing each course's inner tree. Therefore, in total is $O\left(m + \sum_{id \in \text{course Ids.}} n_{id}\right)$ time.

  Therefore, total time is $O\left(n + m + \sum_{id \in \text{course Ids.}} n_{id}\right)$. Since $n_{id} \leq n$ for all $id \in$ course Ids., and size of course Ids. is m, we get total time of $O(n + m + m \cdot n) = O(m \cdot n)$ as required.
- addStudent(): Involves adding student with default counters (and value of global bonus at join time) to the studentMap. So $O(1)$ for creating the student instance, plus insertion to and AVL tree of size n: $O(log(n))$.
- removeStudent(): Involves removal from the studentMap AVL tree in $O(log(n))$ time. Destruction of removed student is $O(1)$.
- addCourse(): Requires insertion of an empty Course (empty inner tree so $O(1)$ additional overhead) to the courseMap which is of size m, in $O(log(m))$ since courseMap is an AVL tree. So in total: $O(log(m))$.
- removeCourse(): Since we are always removing an **empty course**, its destruction time on its own costs $O(1)$ (no inner tree traversal). Therefore, the cost is for node removal only from the course tree, so $O(log(m))$ in total.
- enrollStudent(): This function involves:
  - finding the student by id in studentMap - $O(log(n))$
  - finding the course by id in courseMap - $O(log(m))$
  - inserting a reference to the found student to the found course's inner tree: $O(log(n_{id}))$
  - incrementing the said student's course count by 1 - $O(1)$

  Since $n_{id} \leq n$, we get in total $O(log(n) + log(m))$.

- completeCourse():
  - finding the course by id in courseMap - $O(log(m))$
  - finding a reference to the student with given id in the found course's inner tree: $O(log(n_{id}))$
  - decrementing the said student's course count by 1 and adding it the courses' completion points - $O(1)$

  Note we always assume in this function that student is an enrolled one (otherwise, return an error), so we never have to look up student in the student tree. So, in total: $O(log(n_{id}) + log(m))$.

- awardAcademicPoints(): Increments the global bonus counter so $O(1)$.

- getStudentsPoints(): returns current global bonus minus global bonus when student joined (bonus penalty), plus the student's completion points. So it correctly enforces that students only benefit from bonuses that were awarded **after** they joined. This is all done in $O(1)$. Therefore, since searching the student in studentMap is $O(log(n))$, we get cost of $O(log(n))$ in total.