Student Mentor Data Structure Design

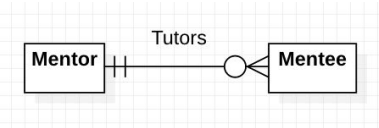
P254033
Prior to peer-review release V1

Solution

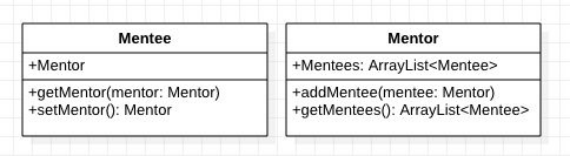
With the specification provided, a suitable solution has been designed in the form of pooled trees, as will be explained forth. From the start, we'll assume implementation in Java. This'll influence the casing and classes used UML, namely *java.util.ArrayList*. We'll also assume that this structure is for non-prolonged use, which results in the lack of disk storage. Instead, the entire data structure will reside in volatile memory only.

To start, we'll follow the scenario to define the entities involved; Mentees and Mentors.

In their most abstract form, a Mentee has one Mentor, and a Mentor has many Mentees. From this we can assume a one to many relationship between mentors and mentees. The diagram to the right shows zero, one or many mentees; this is due to the way these entities were later rolled together, since any student may be either or both, as explained later.

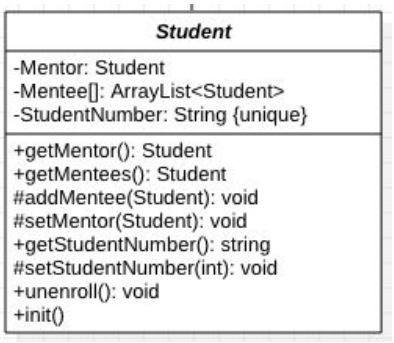


Based purely on these definitions of the entities and their relationships, we can begin prototyping some UML class definitions.



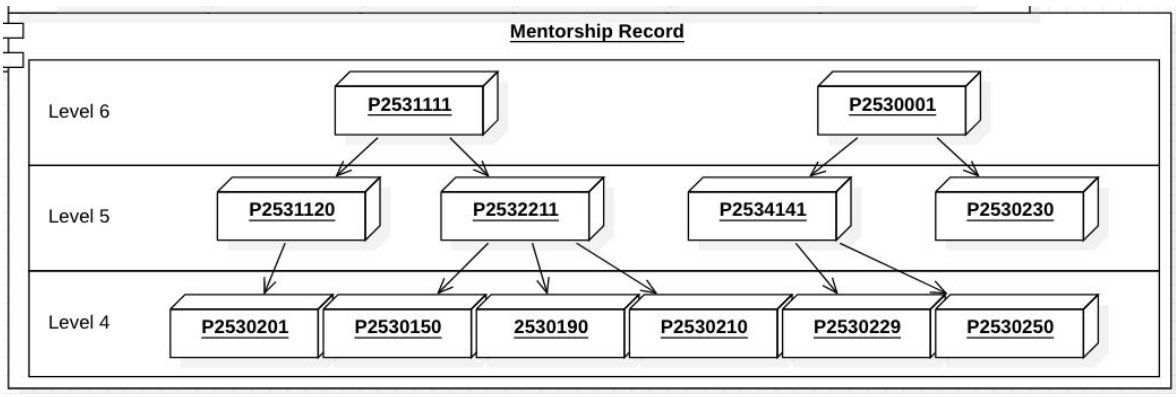
As the specification states, any given student, especially those at level 5, can both be a mentor, and be mentored. This brings about complications such as which class would be used, and how we'd determine how to treat each student if they're more than one, or if they change state, i.e if they start as mentee and mutate to a mentor over time.

To simplify the solution and solve any issues this induces, we'll aggregate the two rolls into a more abstract definition of a student; therefore any given student may be a mentee, or a mentor, or both. It will not strictly be defined as to which they are, but rather their status would be assumed from the relationships they inhabit. This class supersedes both previous classes.



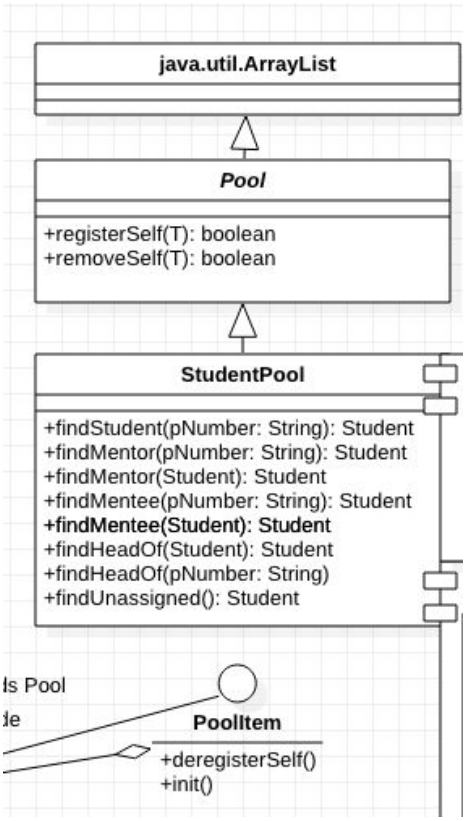
Combining the Mentee and Mentor UML's, and expanding with the data we need to store, and more interfacing methods results with the UML on the right.

We can interpret each student as a node, similar to a linked list item, where it has links to a 'previous' item and many 'next' items. These links create a tree like structure, which we can visualise using the relationships provided in the specification scenario.



To reduce complexity for global operations in this structure, there will be a second section to this solution; a pool, which, in this case, is an unstructured collection of items that is not internally managed. Instead, instances automatically add themselves when they're created, and will remove themselves when they're deconstructed, i.e when a student is unenrolled.

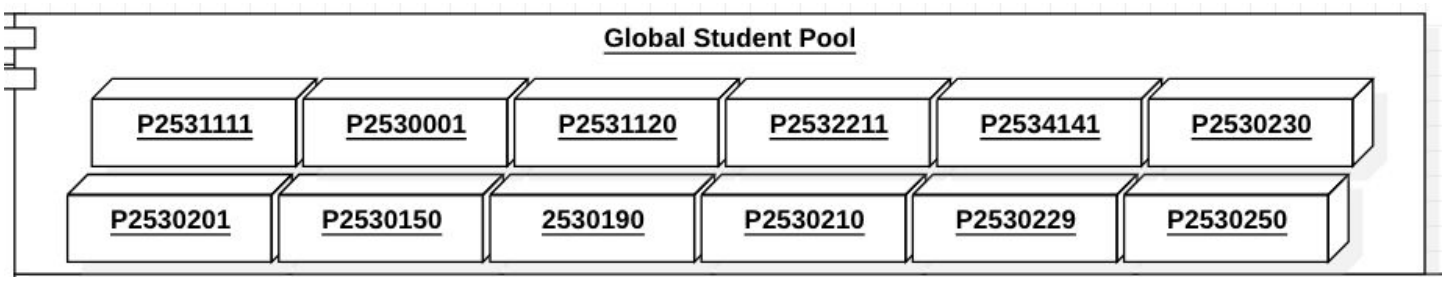
We'll use a pool to store a reference to all students. This is justifiable for multiple reasons, the most important being; firstly there is no head to the structure, as can be seen on the diagram above. Secondly there is no centralised or parent storage for the structure.



With these quirks in mind, how would we search this structure? Where would the object reference to perform a search be, if we have no parent tree object? Even if we had a head reference, Would the structure shown above be one tree? It shows two trees with individual heads with no relationships between them, so how would we know which part to search, or that we've searched all students, even those who have zero relationships leading to a head?

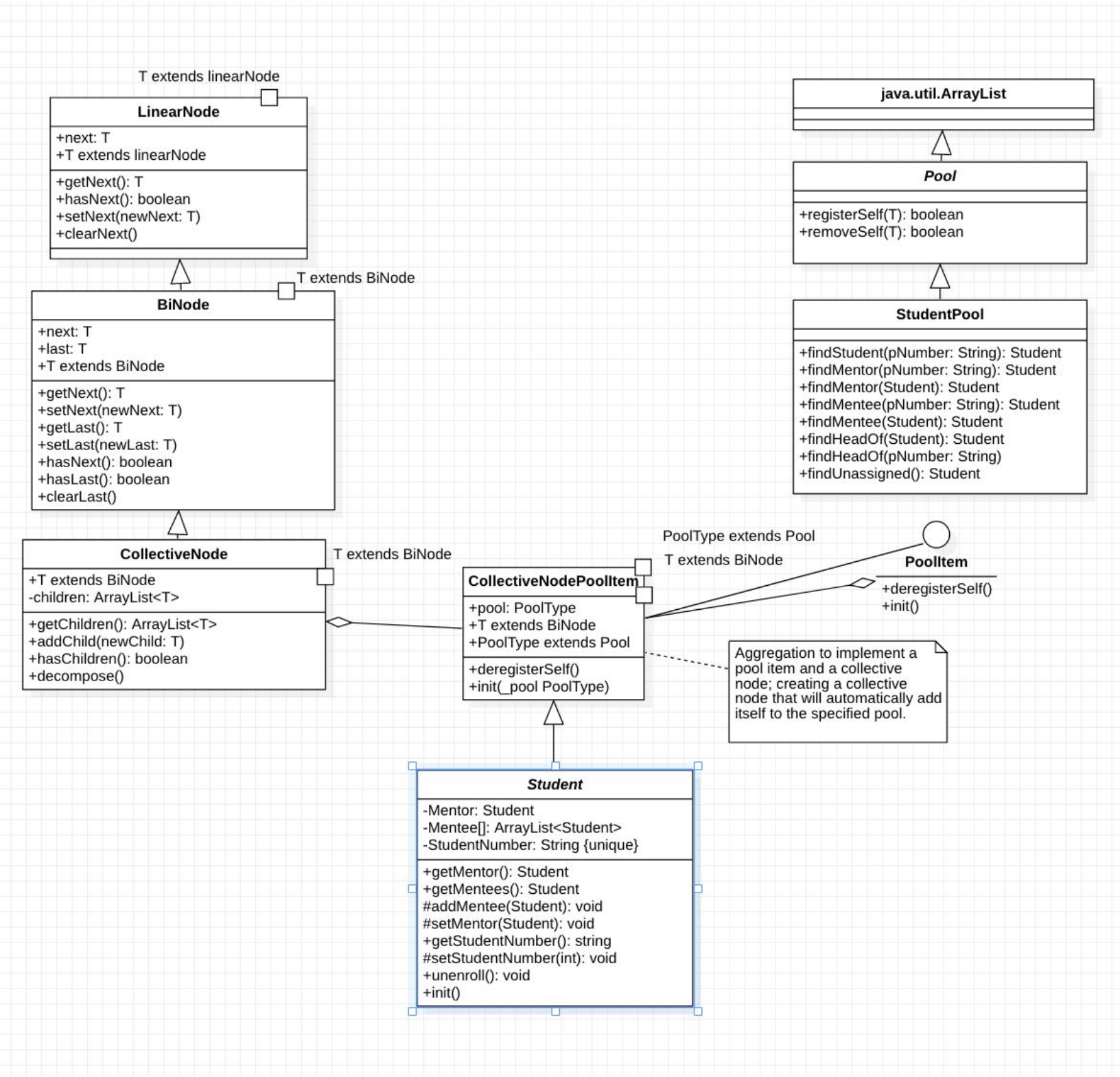
These're difficulties that need to be solved in order to support all functions on the structure, and to have global access to each and every student.

A Pool will coexist with the tree structure, where a tree is orderly and not stored, and the pool contains every student with ignorance in regard to their entity relationships or states. The structure tree shown below would look like this within the pool.



Since this pool will have a reference to all students, it can be used to search for p numbers, mentors, mentees, and unlinked students. Best of all, it's all managed automatically. It will be the key to performing operations globally.

Combining and expanding everything above, and following up with abstraction, aggregation, interface design, and refinement we're left with this completed UML diagram, suitable for a Java implementation.



Operation	Assumptions	Invocation	Time Complexity	Functionality
Assign Mentor to Mentee	We know the relationship, and have references to both students	lib.student.Student#setMentor(Student)	O(1)	Overwrites the '*last*' pointer inherited from BiNode, which stores mentors, with the (new) mentor. The new mentor will be assigned to the mentee, then the mentor is assigned the mentee. If '*last*' is not null, see 'Reassign Mentor'
Search for any student	The student is in the global pool, and the search P is valid.	lib.collection.pool.StudentPool.Global#findByP(String): Student?	O(n), where *n* is the quantity of students in the pool.	Searches the global pool of students, NOT the relationship structure, to find a student with a matching p number. Rejects call if string is not a valid p number. Returns null of rejected, or if no student matches.
Search for a mentee	The student is in the global pool, and has a mentor.	lib.collection.pool.StudentPool.Global#findMenteeByP(String): Student?	""	Uses above to find a student, but only returns them if they're considered to be a mentee (Has a mentor). Otherwise null.
Search for a mentor	The student is in the global pool, and has one or more mentees.	lib.collection.pool.StudentPool.Global#findMentorByP(String): Student?	""	Same as above, but only returns a matching student if they're considered to be a mentor (Has > 0 mentees). Otherwise null.
Determine is a mentor has been assigned	(All below operations assume we have an object reference to the student we're checking.)	student#hasMentor(): boolean student#isMentee(): boolean	O(1) O(1)	Determines if the student has an assigned mentor (is a mentee). Returns true if 'last' points to a mentor.
Determine if a mentee has been assigned		student#hasMentees(): boolean student#isMentor(): boolean	O(1) O(1)	Determines if the student has been assigned at least one mentee (is a mentor). Returns true if '*children*', inherited from CollectiveNode, which stores mentees, contains more than zero elements.
Find mentor of any mentee		student#getMentor(): Student?	O(1)	Returns the '*last*', which stores a student's mentor. May be null if a student has no mentor (Is not a mentee)
Find mentees of any mentor		student#getMentees(): ArrayList<Student>	""	Returns the working copy of '*children*', inherited from CollectiveNode, which holds all mentees for a student. For efficiency, this operation does not create a copy; the result should be treated as read-only.
Reassign mentor		student#setMentor(Student): void	""	Dissolves existing relationship, overwrites '*last*' with the new mentor, and adds self as a mentee in the new mentor.
Transfer mentees		student#transferMenteesTo(Student): void	O(n), Where *n* is the quantity of mentees *student* has.	Uses above on all of a mentor's mentees to assign a new mentor, thus transferring the relationship on all of a mentor's mentees to a new mentor.
Reassign mentees	That we want to remove all existing mentees from a mentor, and add new ones.	student#clearMentees(): void student#addMentee(Student): void	O(1) O(1)	*clearMentees* removes all mentees, decomposing their relationships. Then, *addMentee* can be used to assign new mentees.
Remove mentors		student#clearMentor(): void	O(1)	Decomposes the relationship with the current mentor, and removes the reference to it. If there is no mentor, this has no effect.
Remove mentees		student#clearMentees(): void student#clearMentee(Student): void	? ?	Removes all mentees / specified mentee, and decomposes the relationship.

A Note on Execution

The repo provided (or found at https://github.com/Shinkson47/Data-CWK) has project metas configured for both Eclipse and IntelliJ.

The Class path requires:

- JUnit (4)
- JavaFX
- OPEX

OPEX is a library of my own creation which is used to simplify showing a JavaFX utility window. It's found in */lib/OPEX.jar*, within the repo.

nb Showing the GUI is the only use of the library in this repo.

There are two mains, *OperationImplDemo#main, & GUIDemo#main.*
OperationImplDemo is a ***source only*** *demo, showing how to invoke operations on the data structure; whilst GUIDemo is a utility which allows a user to manually manipulate it.*