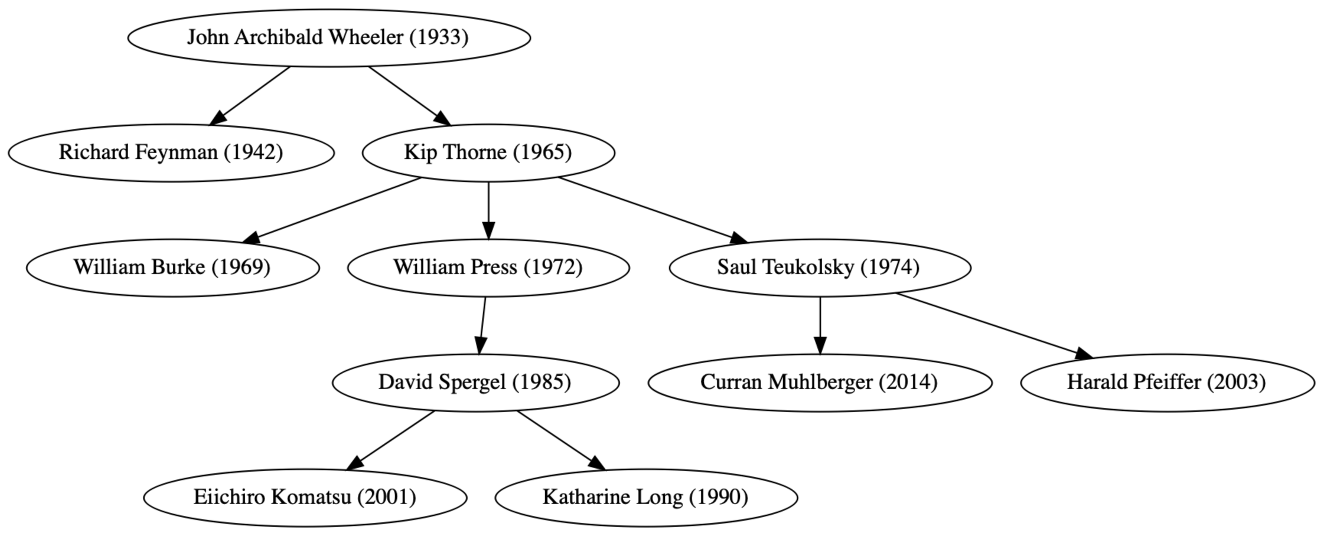
CS 2110: Object-Oriented Programming and Data Structures

Assignment 4: PhD Genealogy Tree

Your assignment is to implement an “academic genealogy” data type to model professors and their advising relationships. A PhD degree is obtained under the mentorship of a professor, so these mentoring relationships can be viewed as a tree in which each student is a child node of their mentor’s node. In order to meet the requirements of a tree, only a student’s relationship with their *primary advisor* will be tracked (that way, each child node only has a single parent). An example of such a tree is shown below:



Each node includes the name of a former PhD student and the year in which they earned their degree. These details, along with their advisor’s name, will be provided in a CSV file, which your program will read in order to construct the tree in memory. The professor at the root of the tree will not have an advisor listed; their academic ancestors will not be considered by your program.

Given this tree representation, you will be able to compute some interesting properties of the geneology, such as finding the most recent ancestral advisor shared by two professors. Similar to A2, your program will provide an interactive interface for querying such properties.

Learning Objectives

- Process general tree data structures recursively.
- Report and respond to failed operations using exceptions.
- Write tests for methods expected to throw exceptions.
- Construct hierarchical structures from sequential input (i.e. a text file).

Recommended Schedule

Start early. Office hours and consulting hours are significantly quieter shortly after an assignment is released than closer to the deadline. We recommend spreading your work over at least 5 days. Here is an example of what that schedule could be:

- Day 1: Skim this entire handout. Download the release code and get it set up in IntelliJ. Make sure you can run the test suite `PhDTreeTest` (most test cases will fail at this point). Study `numLeaves()` and `findProlificMentor()`; if they do not make sense, read the recommended material in JavaHyperText. Complete TODOs 1–3 and test their base cases and exceptional cases.
- Day 2: Complete TODOs 4–6. Add testing utilities to construct larger trees, then expand your testing of all methods so far to cover their recursive cases.
- Day 3: Complete TODOs 7–9, including writing and passing their corresponding tests. Review how to parse CSV files from A3 and start working on TODO 10.
- Day 4: Complete TODO 10 and test it interactively by reading and printing trees from the included CSV files. Then complete TODOs 11–15.
- Day 5: Create new input-tests cases. Re-read your code. Ensure that it conforms to our style guide and complies with all implementation constraints. Submit to CMSX, then confirm that you submitted the files you meant to.

Collaboration Policy

On this assignment you may work together with one partner. Having a partner is not needed to complete the assignment: it is definitely do-able by one person. Nonetheless, working with another person is useful because it gives you a chance to bounce ideas off each other and to get their help with fixing faults in your shared code. If you do intend to work with a partner, you must review the syllabus policies pertaining to partners under “programming assignments” and “academic integrity.”

Partnerships must be declared by forming a group on CMSX *before* starting work. The deadline to form a CMS partnership is **Wednesday, March 22, at 11:59 PM**. After that, CMSX will not allow you to form new partnerships on your own. You may still email your section TA (CCing your partner) to form a group late, but a 5 point penalty will be applied. This is to make sure you are working with your partner on the entire assignment, as required by the syllabus, rather than joining forces part way through.

Pair Programming. If you do this assignment with another person, you must work together with both partners contributing. We recommend trying pair programming, in which the partners sit together—one partner acting as the “pilot” driving the keyboard and mouse and the other as the “navigator,” guiding and critiquing. Here is an effective workflow: first, the pilot proposes a method signature and specification. The navigator then critiques: is the spec clear? They iterate until both agree on the spec. Only when they agree does the pilot write the code. It is then the job of the pilot to convince the skeptical navigator that the code meets the spec. You should switch off the roles of pilot and navigator.

As before, you may talk with others besides your partner to discuss Java syntax, debugging tips, or navigating the IntelliJ IDE, but you should refrain from discussing algorithms that might be used to solve the problems, and you must never show your in-progress or completed code to another student who is not your partner. Consulting hours are the best way to get individualized assistance at the source code level.

Frequently asked questions

If needed, there will be a pinned post on Ed where we will collect any clarifications for this assignment. Please review it before asking a new question in case your concern has already been addressed. You should also review the FAQ before submitting to see whether there are any new ideas that might help you improve your solution.

I. Assignment overview

To complete this assignment, you will need to do the following:

1. Implement a PhDTree data structure.
2. Load a tree from a CSV file.
3. Implement commands to query a PhDTree.
4. Perform end-to-end testing of your application.
5. Create a summary document reflecting on your development process.

Setup. Download the release code from the CMSX assignment page; it is a ZIP file named “a4-release.zip”. Follow the same procedure as for previous assignments to extract the release files, open them as a project in IntelliJ, select a JDK, and add JUnit 5 as a test dependency by resolving import errors in “tests/cs2110/PhDTreeTest.java”. Confirm that you can run the unit test suite; the test case `testConstructorProfToString()` should pass out of the box. Now you’re ready to code.

II. Summary Document

You must write a short document briefly reflecting on the following aspects of your work:

Time. How much time (in hours) was spent on the assignment? (please only write a number here)

Implementation strategy. Describe how you went about implementing the assignment and how the work was divided between the partners.

Testing strategy. How did you perform testing and design test cases?

Known problems. Are there any things that do not work?

Comments. Do you have any comments on the assignment?

Be thinking about these points as you work. When you finish, write your responses in the file "summary.txt", which is included in the release code.

III. Implement a PhDTree (TODOs 1-9)

To represent an academic genealogy as a tree, we need an appropriate data type to store professors and their advisees. The assignment's release code outlines a `PhDTree` class to serve that purpose, as well as a `Professor` class to aggregate a professor's name and the year they earned their degree. A `PhDTree` represents a **general tree** of `Professor`s—each node may have any number of children. The tree is represented recursively—a "node" can be considered as a subtree and is therefore represented as its own `PhDTree`.

Accordingly, `PhDTree` has two fields: its "data" (a `Professor`) and a collection of its "children". By storing the children in a `SortedSet` (ordered chronologically by degree year, then alphabetically by name; a comparison that is deferred to `Professor`), code that iterates over the children will always do so in the same order. This provides **determinism**, which is helpful when debugging and printing.

Your task is to finish the implementation of the `PhDTree` class by completing TODOs 1–9 in "PhDTree.java". As you work on each method, add additional test cases to "PhDTreeTest.java", as indicated by the unnumbered TODOs in that file. You will need to do your testing in stages, as non-trivial trees cannot be created until you have finished implementing `insert()`. Fortunately, this is a good match for recursion. Start by writing tests for your base cases, including exceptional cases. Then, once you can build trees with more than a single node, go back and write tests for your recursive cases.

Many of the methods require recursion. You may find the Java HyperText materials on recursion over trees to be a helpful starting point.

Many of the methods also need to handle exceptional circumstances, such as looking for a professor who is not in the tree. They respond to these circumstances by throwing a custom checked exception named `NotFound`, which is included in the release code (note that exception classes do not need to have names that end in "Exception"). When calling these methods recursively, think about how an exception thrown by a child node should be handled (should it be propagated, or should the method continue recursing on other children?).

Defensive programming. You have been provided an implementation of `assertInv()` that asserts that no professor occurs more than once in the tree and that no node has more than one parent. This can help you catch bugs quickly, but only if you call it. Remember that, at a minimum, a class's invariants should be checked after every mutating operation.

If a method has preconditions that can be checked "locally" (that is, without traversing the tree), they should be asserted. If checking a precondition would require searching the tree, however, then on this assignment you are *not* expected to assert it. Just trust that the client will respect it (there's also a chance that `assertInv()` will catch the result of a violation). Remember, though, that "throws" clauses do not define preconditions—the behavior in these exceptional cases is *defined*.

Testing tips

- You must test all the methods that you implement in `PhDTree` with at least 3 additional test cases, two of which must be non-trivial (that is, involving nodes with multiple children). Think about what tests are needed to achieve coverage in both a black-box and a glass-box sense. In particular, any exceptional cases must be tested (see below for how to test with exceptions).
- Once you have implemented `insert()`, consider writing helper methods that create trees to be used as a starting point by multiple tests (some trivial ones are included in the release code, but you will need bigger trees for thorough testing). Figures in this handout would make good test trees.

Testing exceptions. Since some of the methods promise to throw a `NotFound` exception under certain conditions, you should include tests that check whether an exception is thrown as specified. The JUnit function `assertThrows()` is the right tool for the job, but it works a little differently than other JUnit assertions. You must specify the expected exception's `Class` object, and you must defer execution of the exception-throwing code using an **anonymous function**.

For example, suppose that calling `contains()` on a tree `t` is expected to throw a `NotFound` when passed the argument `prof3`. This behavior would be tested using the following code:

```
assertThrows(NotFound.class, () -> t.contains(prof3));
```

We'll come back to anonymous functions in more detail later in the course, but for now you can use them in this idiomatic way by just putting `() ->` in front of expressions whose evaluation needs to be deferred.

> Why must execution be deferred when testing exceptions? Remember that a function's arguments are evaluated before the function itself is executed. So if an argument to an assertion threw an exception, the assertion function would never be run, and the exception would likely crash the test case.

You would like an assertion that does the following:

```
try {
    t.contains(prof3);  // Call code-under-test here
    fail("Exception was not thrown");
} catch (NotFound e) {
    // Test passed
}
```

Note that the function being tested needs to be called inside the `try` block, so it can't have already been evaluated as an argument to the assertion. But by using an anonymous function, the assertion can decide when to evaluate it and defer evaluation until inside a `try` block similar to the above. We'll revisit this idea of deferring execution when discussing event listeners in graphical user interfaces.

As a general rule in assignments, if you add any helper methods to a given class, those methods must be private, and you should not test them in a submitted test suite. You might want to test such functions in isolation during early development, but ultimately they should be covered by the tests of the public functions that call them.

Tour of PhDTree operations

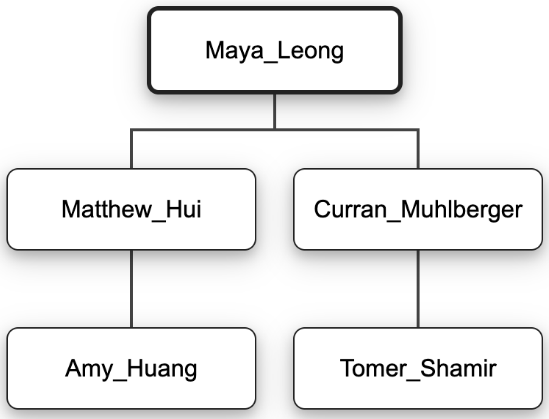
Many useful operations on trees can be categorized as either "counting" (which accumulates a result from all of its children's results) or "searching" (which forwards the first successful result found). The release code includes examples of both in `numLeaves()` and `findProlificMentor()`. Reference these examples for inspiration as you implement the remaining methods. Note: both methods require that you complete TODO 1 before their tests will pass; understanding the fields of `PhDTree` and their types should point to a quick solution.

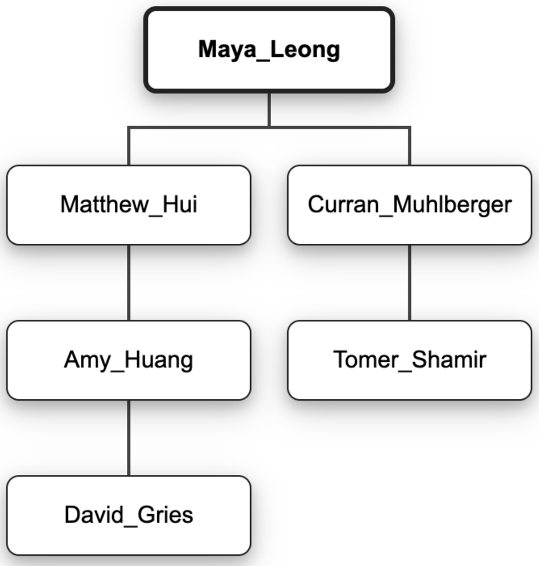
Core tree operations

TODOs 2–3 are intended as a warmup, adapting operations you have seen on binary trees to more general trees.

As mentioned, you will not be able to construct non-trivial trees for testing them without implementing `insert()`, but you cannot implement `insert()` without first implementing `findTree()`, which will help you locate the new PhD's advisor in the tree. Try to validate your approach for TODOs 4–5 as well as you can (for example, by tracing your logic on a whiteboard). Then test these methods thoroughly before moving on. Once `findTree()` is working, the test for `contains()` should pass as well.

Here is an illustration of the desired effect of `insert()`: Given the tree on the left, the code `Maya_Leong.insert(Amy_Huang, David_Gries)` should result in the tree on the right.





Finding a node’s parent

Up to now you have been implementing general-purpose tree functionality specialized to `Professor` data. Now you will implement some useful queries that are more specific to this domain. A common question in academic circles is who a particular professor’s advisor was. While you can now find the node containing that professor, nodes do not contain pointers to their parents. For TODO 6 you will need to implement a new method, `findAdvisor()`, that similarly searches for a target professor but returns the professor in the *parent* node instead. This requires a different base case (likely moved into the loop over children).

For example, given the tree on the right above, we would expect these results from `findAdvisor()` (hint: turn these into test cases):

- `Maya_Leong.findAdvisor(Maya_Leong)` throws `NotFound`
- `Maya_Leong.findAdvisor(Tomer_Shamir)` returns `Curran_Muhlberger`
- `Curran_Muhlberger.findAdvisor(Tomer_Shamir)` returns `Curran_Muhlberger`
- `Maya_Leong.findAdvisor(Matthew_Hui)` returns `Maya_Leong`
- `Amy_Huang.findAdvisor(Tomer_Shamir)` throws `NotFound`
- `Matthew_Hui.findAdvisor(Matthew_Hui)` throws `NotFound`

Tracing academic lineage

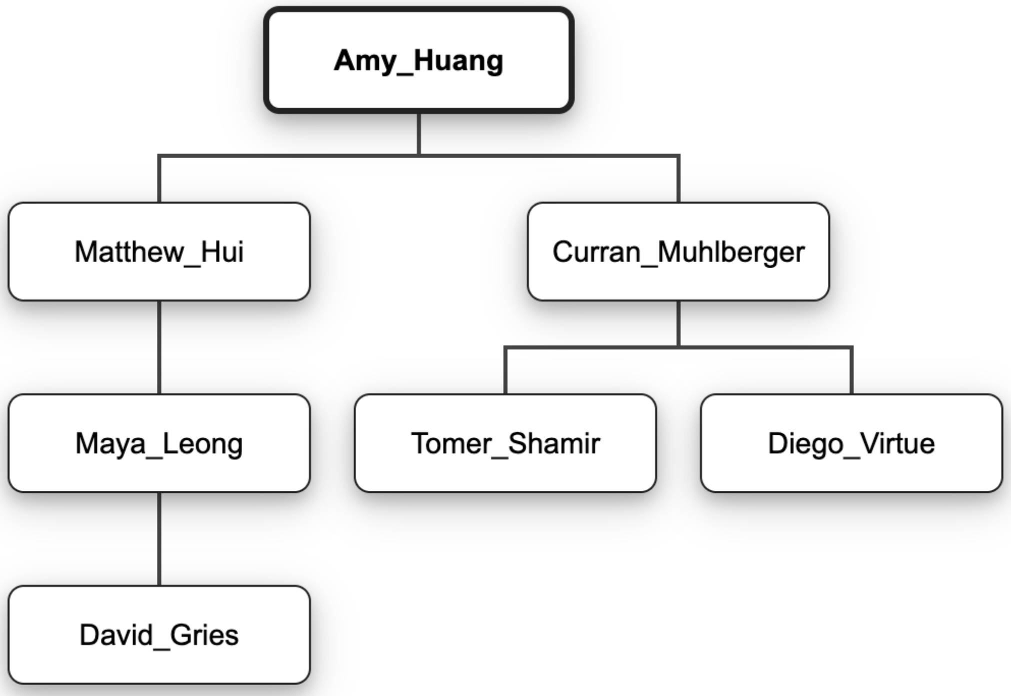
What if we want to go further and trace a contemporary professor’s academic lineage all the way back to someone like Isaac Newton? Such a method would need to accumulate all of the advising edges it followed in order to find the target professor. Fortunately, in a recursive implementation, this information implicitly exists on the stack of activation records for the recursive calls. Each call thus needs to add its contribution to the lineage before returning.

Hint: which call should be responsible for constructing the returned list? (the one at the top of the stack, or the one at the bottom?)

When constructing the list, consider how elements are added to it. Do you primarily append to the list, prepend to it, or insert into the middle? Choose an appropriate implementation of the `List<T>` interface to maximize the efficiency of the predominant operation.

It is possible to implement `findAcademicLineage()` using repeated calls to `findAdvisor()`, but as mentioned in the implementation constraints, this would be very inefficient and is not allowed. Each call to `findAdvisor()` might have to search most of the tree and cannot benefit from knowing how the previous call found its target. A new recursive method should only need to search the tree once.

Given the following tree, below are some expected results from `findAcademicLineage()` (here, the square brackets denote a list). Does this inspire some test cases?



- `Amy_Huang.findAcademicLineage(Tomer_Shamir)` returns the list `[Amy_Huang, Curran_Muhlberger, Tomer_Shamir]`
- `Amy_Huang.findAcademicLineage(Amy_Huang)` returns `[Amy_Huang]`
- `Amy_Huang.findAcademicLineage(Tanvi_Namjoshi)` throws `NotFound`
- `Matthew_Hui.findAcademicLineage(Curran_Muhlberger)` throws `NotFound`
- `Matthew_Hui.findAcademicLineage(Maya_Leong)` returns `[Matthew_Hui, Maya_Leong]`

Finding common ancestors

The last query you need to implement is for finding the most recent common academic ancestor of two professors. Even if two professors can both trace their advisors back to Isaac Newton, at some point their lineage is likely to diverge. The purpose of the `commonAncestor()` method in TODO 8 is to find this divergence point.

As an example, given the tree above, the following results are expected:

- `Amy_Huang.commonAncestor(Matthew_Hui, Amy_Huang)` is `Amy_Huang`
- `Amy_Huang.commonAncestor(Matthew_Hui, Matthew_Hui)` is `Matthew_Hui`
- `Amy_Huang.commonAncestor(Matthew_Hui, Curran_Muhlberger)` is `Amy_Huang`
- `Amy_Huang.commonAncestor(David_Gries, Diego_Virtue)` is `Amy_Huang`
- `Matthew_Hui.commonAncestor(Matthew_Hui, Maya_Leong)` is `Matthew_Hui`
- `Matthew_Hui.commonAncestor(Tomer_Shamir, Maya_Leong)` throws `NotFound`

Finding two professors' most recent common ancestor is straightforward if you have both of their lineages from a distant common ancestor (i.e. the root of the full PhDTree). Both lineages will have a common prefix of advisors (starting with the root), and the last element of that common prefix is their most recent common ancestor.

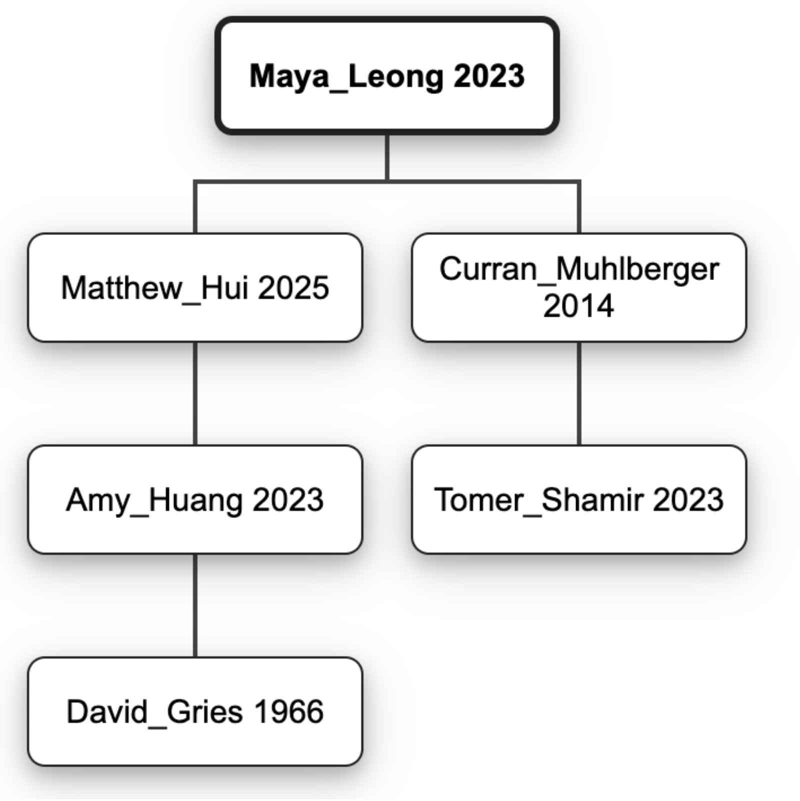
To find the end of the common prefix, you will likely use iteration. But think about the efficiency of the types involved: if you iterate over an index, a linked list will perform poorly (remember the stair climbing analogy). And the signature for `findAcademicLineage()` does not guarantee any particular list implementation. How can you ensure reasonable performance? One option is to copy the data to a type with known performance characteristics (such as an `ArrayList<T>` or array). This is allowed, but the copy is wasteful. As a challenge, can you find the end of the common prefix using only the lists' `Iterator`s?

Printing a tree's contents

The release code includes an implementation of `toString()` to aid in debugging, but since everything is on a single line of text, it is difficult to read for large trees.

In TODO 9 you will implement `printProfessors()` to provide a more legible view into the contents of a PhDTree. The printed output will not convey advising relationships, but it will print every professor in the tree along with the year they earned their degree. For deterministic output, the implementation constraint requires that you perform a **preorder traversal** of the tree.

For example, for the following tree:



`Maya_Leong.printProfessors(new PrintWriter(System.out))` should print the following:

```
Maya Leong - 2023
Curran Muhlberger - 2014
Tomer Shamir - 2023
Matthew Hui - 2025
Amy Huang - 2023
David Gries - 1966
```

Using a preorder traversal with a SortedSet of children, the lines of the output should come in an order such that advisors precede their advisees, as in the example above (e.g. Leong is before Muhlberger), and advisees of a given advisor are ordered chronologically by degree, then alphabetically with respect to each other (e.g., Muhlberger is before Hui).

(Note: The dates in the examples in this handout, as well as in some of the method specs, do not make physical sense, with advisees earning their PhD before their advisor. Fortunately, this does not violate any invariants, so the examples are still valid. The examples also sometimes draw children in a different order than they would be iterated over; we apologize for any confusion here.)

Testing functions that perform I/O can be tricky. Fortunately, because `printProfessors()` takes a `PrintWriter` as an argument, rather than always printing to `System.out`, it is possible to “print” its output to a `String` and perform testing on the result. The example test case in `testPrintProfessors()` shows how to do this using a `StringWriter`.

IV. Implement PhDApp.java (TODOs 10-15)

With a solid PhDTree implementation, you are ready to begin implementing your `PhDApp` class. In this class, much like in A3, you will implement a function that converts a simplified Comma-Separated Values (CSV) file of genealogy data into a `PhDTree` representation. This will allow you to use your `PhDTree` class to explore some interesting data.

Reading a tree from a CSV

Two example CSV files are included with this assignment: “professors.csv”, which contains about a hundred rows of genealogy data, and “professors-shortened.csv”, which contains less data. To avoid confusion, do not modify these files (though you are welcome to create modified copies for your input tests later). You may want to use “professors-shortened.csv” for console testing since the output will be less overwhelming, but it may be more fun to explore “professors.csv” once you’re confident in your code. And of course you can build your own academic genealogy files as well.

Valid CSVs representing academic genealogy must obey the following rules: the first row must be the header `advisee,year,advisor`. The second row represents the root of the tree and has no advisor. Then, the following rows detail the relationships in this genealogy, with the rows introducing advisors always preceding those of their advisees. For example, this CSV data:

```
advisee,year,advisor
Maya Leong,2023,
Matthew Hui,2025,Maya Leong
Amy Huang,2023,Matthew Hui
Curran Muhlberger,2014,Maya Leong
Tomer Shamir,2023,Curran Muhlberger
David Gries,1966,Amy Huang
```

corresponds to the tree shown in the section “printing a tree’s contents”. Maya Leong, as the first row and only data value with no advisor, is the root of the PhD tree. Then, both Matthew Hui and Curran Muhlberger are advisees of Maya Leong, with the given degree years, while Amy Huang, in between, is an advisee of Matthew Hui, etc.

For TODO 10, you must implement `csvToTree()` to read a file in this format and construct the corresponding `PhDTree`. If there are any problems parsing the file, an `InputFormatException` should be thrown with a descriptive message (the exact wording of the message is up to you). Problems might include lines with the wrong number of tokens, years that are not numbers, duplicate advisees, or advisors that were not previously declared as an advisee in the file. Note that `InputFormatException` is another custom exception defined for this assignment, like `NotFound`.

Your solution to A3 should provide a reminder of how to read lines from a CSV file and split them into tokens. Per the restrictions of Simplified CSV, professor names in these files will not contain commas or newlines. You will not be submitting a unit test for `csvToTree()` (though you may certainly write one on your own). Once `csvToTree()` is written, you can start exploring the tree it creates using the `PhDApp` application’s commands.

Interacting with a tree

Once `csvToString()` is written, run `PhDApp` in IntelliJ with the default program arguments. You should see the prompt “Please enter a command: ”. Type “help” and press Enter; the output describes the features you will be adding. The “print” command has already been implemented, so try typing “print”, followed by Enter; you should see the contents of the tree defined by “professors.csv”. Finally, type “exit”, followed by Enter, to stop running the program.

Remember that, by default, `assert` statements used to check preconditions and invariants are disabled when running an application. To enable them (and catch more bugs), edit the Run Configuration for `PhDApp` and add the “-ea” flag under “VM Options” to enable assertions. Get in the habit of doing this whenever you click a green arrow in a new class for the first time.

For TODOs 11-15, you must implement the remaining commands for this interactive interface. The expected outputs are documented below. Please match this output *exactly*, paying careful attention to capitalization, punctuation, and spacing. Reference the implementation of `doPrint()` as an example for how you might handle arguments and exceptions in these command functions.

Command `contains`

Action
Determine whether the professor specified in the argument is contained in the PhDTree.

Arguments
Name of professor to query (required)

Successful response

```
This professor is contained in the tree.
```

Unsuccessful response

```
This professor is not contained in the tree.
```

Command `size`

Action
Count the number of professors contained in a subtree rooted at the professor specified in the argument, or in the whole tree if no argument is provided.

Arguments
Name of professor whose subtree should be measured (optional; default is the entire tree)

Successful response (example)

```
The number of nodes in this tree is: 105.
```

Unsuccessful response

`This professor does not exist in the tree.`

Command `advisor`

Action

Determine the advisor of the professor specified in the argument.

Arguments

Name of professor to query (required)

Successful response (example)

`The advisor of this advisee is: Isaac Newton (1668).`

Response for root of tree

`This professor does not have an advisor in the tree.`

Unsuccessful response

`This professor does not exist in the tree.`

Command `ancestor`

Action

Determine the most recent common academic ancestor of the two professors specified in the argument.

Arguments

Names of professors to query, separated by a comma (required)

Successful response (example)

`The common ancestor of these professors is: Isaac Newton (1668).`

Unsuccessful response

`These professors do not have a common ancestor in the tree.`

Command `lineage`

Action

Print the sequence of advisors from the root of the tree to the professor specified in the argument, separated by `"--"`.

Arguments

Name of professor to query (required)

Successful response (example)

```
The lineage is: Isaac Newton (1668)--Roger Cotes (1706)--Robert Smith (1715)--Walter Taylor (1723).
```

Unsuccessful response

```
This professor does not exist in the tree.
```

You are also welcome to be creative and add your own commands to explore the tree, but the commands listed above are required for this assignment.

V. End-to-end testing

As in A3, you will also be submitting end-to-end tests that provide coverage of the functionality of your `PhDApp` application. These "input tests" should be submitted in a zip file named "input-tests.zip" that contains your project's "input-tests" folder. Your tests should cover each of the commands implemented in `PhDApp`.

Input tests consist of files with standardized names that are placed in subdirectories of the "input-tests" directory. Each such directory should contain a file "input.txt" containing commands for the program, and a corresponding file "output.txt" should contain the expected program output. An example of such a test is in the directory "input-tests/test1". All input tests read CSV input from the file "professors.csv" by default; if the script should be run on a different CSV input, the file to be used should be included in the directory and named "input.csv".

The number of input tests you design is up to you, but keep in mind that the purpose of testing is to achieve coverage. Design enough tests that you feel confident that you have covered the functionality of the program, including in corner cases.

If you extend your program with your own custom commands, do not use those commands in test scripts that you submit to us, since our reference solution will not understand them.

To run your input tests, the main program supports an option to read commands directly from a file rather than from the console. If it is given the option `-i <inputfile>` then it will read commands from the named file. For example, if you set the program arguments to

```
-i input-tests/test1/input.txt
```

your program should produce output that is the same as the expected output in the file "input-tests/test1/output.txt". To test with a custom tree, use arguments like:

```
-i input-tests/test2/input.txt input-tests/test2/input.csv
```

Remember that program arguments can be specified in IntelliJ by editing the Run Configuration for `PhDApp`.

VI. Submission

You made it! Make sure to update the file “summary.txt” with information like the time you spent and your group’s identification. Then upload the files “PhDApp.java”, “PhDTree.java”, “PhDTreeTest.java”, “input-tests.zip”, and “summary.txt” to **Assignment 4** on CMSX before the deadline.

After you submit, CMSX will automatically send your submission to a *smoketester*, which is a separate system that runs your solution against the same tests that we provided to you in the release code. The purpose of the smoketester is to give you confidence that you submitted correctly. You should receive an email from the smoketester shortly after submitting. Read it carefully, and if it doesn’t match your expectations, confirm that you uploaded the intended version of your file (it will be attached to the smoketester feedback). Be aware that these emails occasionally get misclassified as spam, so check your spam folder. It is also possible that the smoketester may fall behind when lots of students are submitting at once. Remember that the smoketester is just running the same tests that you are running in IntelliJ yourself, so don’t panic if its report gets lost—we will grade all work that is submitted to CMSX, whether or not you receive the email.

Last Note: If you do not know where to start, are stuck, do not understand testing, or are lost, please seek help in office hours as soon as possible. Good luck, and happy coding :)

Assignment updates

1. There were some mistakes in the example output of `printProfessors()` in its JavaDoc comment. See the FAQ on Ed Discussion for the correction.
2. The example output for the `advisor` and `ancestor` commands in the handout was inconsistent with the “test1” input test—the latter included degree years, while the former did not. The handout has been updated to match the test case so that your solution can pass the included input-test.