CS 2110 Fall 2022
Assignment A4
PhD Genealogy Tree

Table of Contents

Updates

The following changes have been made to these instructions since their initial release.

- Oct 5: the output format of the "find" command has been defined more precisely.
- Oct 7: the output format of the "find" command was fixed to match the output included with input-test test1.
- Oct 8, 10: the behavior of the "size" command was clarified to agree with the code release.
- Oct 11: the example genealogy figure was replaced

Learning Objectives

In this assignment you will gain experience in using recursion to process tree data structures and learn how to test recursive methods on trees. You will use and handle checked exceptions in a more sophisticated way. You will practice skills you learned in previous assignments such as using a linked list, reading from a .csv file, and writing your own test scripts. Finally, you should get into the habit of checking Ed, especially the pinned post "A4 FAQs". **You are responsible for any clarifications made in that post.**

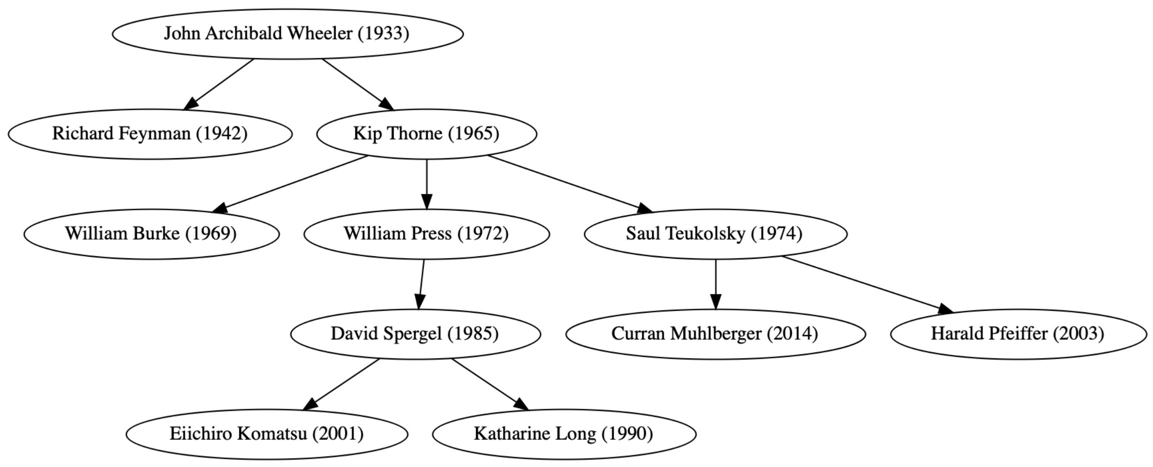
Dos and Don'ts

1. DO read the entire handout before you start. Make sure to start early and go to OH/consulting hours with any questions.
2. DO read the method specifications, the corresponding directions on this handout, and check the Ed pinned A4 FAQ post before you start coding a method.
3. DON'T alter the files `Professor.java` and `NotFound.java`

4. DON'T change the method signatures or specs of any method in class `PhDTree`. However, you may add your own helper methods.
5. DON'T leave print statements in your code.
6. DON'T leave testing till the end. You should be writing methods and testing them in conjunction.
7. DO remember to turn on assertions for your project.

1. Introduction

A tree is a useful data structure that organizes data hierarchically. You probably use tree structures every day. For example, the file system on the computer you are currently looking at is structured as a tree! In this assignment you will be implementing an academic genealogy tree to model professors and their mentorship relationships. A PhD degree is obtained under the mentorship of a professor, so these mentorship relationships can be viewed as a tree in which each student is a child node of their mentor's node. An example of such a tree is shown below:



A4 uses a tree data structure to model PhD genealogy. The root of the tree is the first professor in your data, and since it has no parent, this means that this professor does not have an advisor; the children of each node represent advisees of the professor at that node. In reality, some people have multiple advisors, but we'll just keep track of one to ensure we have a tree structure. You can assume that the CSV only specifies a single advisor.

You will write code to manipulate and explore the tree in the class `PhDTree`. In addition, you will also write code in the file `Main.java`. The code in `Main.java` allows you to interact with the data by using various console commands to explore the tree structure and its properties. These commands will run the methods in `PhDTree`, so if you haven't implemented these methods, this feature will not work. Commands must be entered correctly or the program will not recognize it (commands are case sensitive, space sensitive, etc.: it must be exactly how it is written in the .csv file). Please see section 2c for more information.

Collaboration Policy

You may do this assignment alone or with *one* other person. If you work with a partner, form a group on CMSX *as soon as you decide to work together* to formalize that agreement (if any files are submitted before confirming your group, then the group cannot be formed). *Both* partners must do something to form the group: one invites, the other accepts.

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, with 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.

Academic Integrity. With the exception of your CMSX-registered partner, you may not look at anyone else's code, in any form, or show your code to anyone else, in any form. You may not look at solutions to similar previous assignments in 2110. You may not show or give your code to another person in the class. You can talk to others, but your discussions should not include writing code and copying it down.

2. Tasks

Start by extracting the contents of “a4.zip” onto your computer, then open the contained “a4” folder in IntelliJ IDEA (you may see multiple nested “a4” folders after extracting; open the one that contains the “src” and “tests” subdirectories). Then open “src/a4/PhDTree” from the project browser. You will probably see a banner in your editor saying “Project JDK is not defined”; click the “Setup SDK” link in the corner, then select your version 17 JDK. Now you’re ready to code.

2a) Summary document. You must write a short document briefly describing the following aspects of your work:

- 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?
- Time spent on the assignment, in hours.

- Comments on the assignment.

In the source release you will find a template file named `summary.txt`, which you can edit and put these answers into.

2b) PhDTree.java

The biggest part of the assignment is to implement the class `PhDTree` by completing the source file `PhDTree.java`. Complete and test each method marked with a comment `// TODO`. You will be implementing most of the methods except for `toString()` — we gave you this method to help you with testing. *Hint:* Some methods you are asked to write can be written more simply by using methods you wrote earlier in the assignment.

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

Please use asserts as tool to help you write correct code. However, remember that if a precondition is false, any behavior is acceptable; you may write assert statements for preconditions but are not required to unless they are specified in a Checks clause.

We have provided for you an implementation of a `classInv()` method that checks if a tree is well formed. This method is likely to catch bugs in your code, but only if 1) you use it properly and 2) assertions are turned on.

Please use test-driven development for this assignment. This means that you should write test cases for a method *before* you implement it. See the testing part of this handout for more information on testing.

`PhDTree` will be tested using JUnit unit tests. 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 useful for this purpose. It must specify the expected exception class, and it must defer execution of the exception-throwing code using a *lambda expression.* For example, suppose that a tree is supposed to throw a `NotFound` exception when the `contains()` method is called on tree `t`, passing an object in variable `prof3`. This behavior could be tested using the following code:

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

We'll talk about lambda expressions in more detail later, but for now you can use them in this idiomatic way by just putting "`() ->`" in front of expressions whose evaluation needs to be delayed.

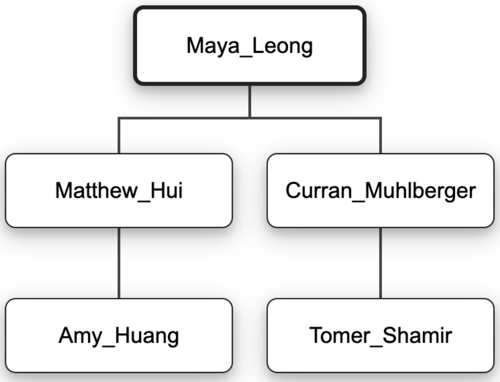
To Do #1
Implement `numAdvisees()` according to specification. This code can be a one-liner.

To Do #2
Implement the `size()` method according to specification. It counts the number of nodes in the tree, which requires recursively traversing the whole tree. *Hint:* the size of a tree is just the sum of the sizes of the subtrees rooted at its children, plus 1. So, call `c.size()` on each immediate child node `c`, add the results, and add 1 more!

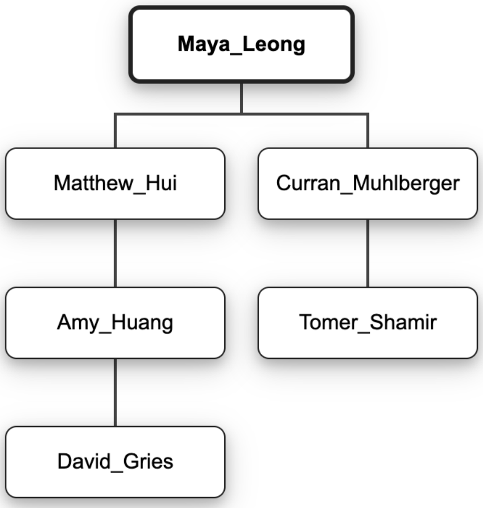
To Do #3
Implement `maxDepth()` according to the spec. This method should be recursive. *Hint:* how can you use the result of `c.maxDepth()`, for each child `c`, to obtain the maximum depth from the current node?

To Do #4
Implement `findTree()` according to specifications. This method should recursively traverse the tree to find the correct node. Since the method can throw an exception, your recursive calls will need to happen inside a try–catch block so that you can handle that exception and do the right thing.

To Do #5
Implement `insert()` according to spec. This method should not be recursive. Do not traverse the tree twice looking for the same professor — don't duplicate work! (*Hint*: look at the implementation of `contains()`). See below for an example.

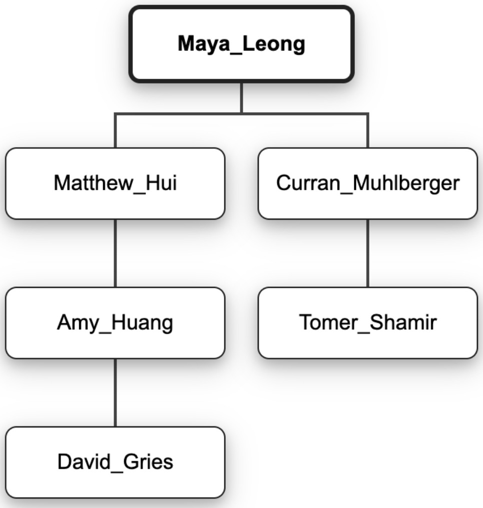


The code `Maya_Leong.insert(Amy_Huang, David_Gries)` should result in the following tree.



To Do #6
Implement `findAdvisor()` according to specification. To find a scholar's advisor, it will be necessary to search the whole tree in general, recursively. An exception should be thrown if the scholar is not in the tree or there is no advisor.

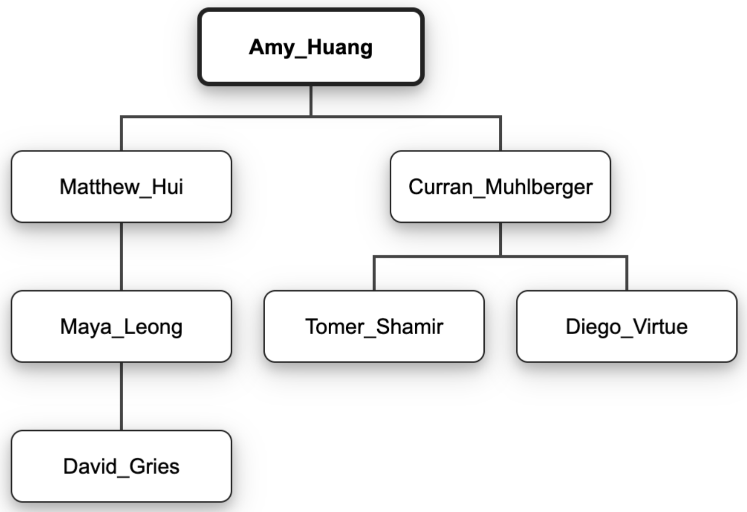
For example, consider the following tree:



`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.getAdvisor(Tomer_Shamir)` throws `NotFound`
`Matthew_Hui.getAdvisor(Matthew_Hui)` throws `NotFound`

To Do #7
Implement `findAcademicLineage()` according to specification. Do not use `findAdvisor()` in this method: it is too inefficient, leading to multiple traversals of the tree. This method must return an object satisfying the interface `List<Professor>`. Recall from the week of 9/26 recitation that `List` is an interface, so use a class that implements it, like `LinkedList`.
Hint: the base case is when the root of this `PhDTree` is just `p`, i.e. the route is just `[p]`.



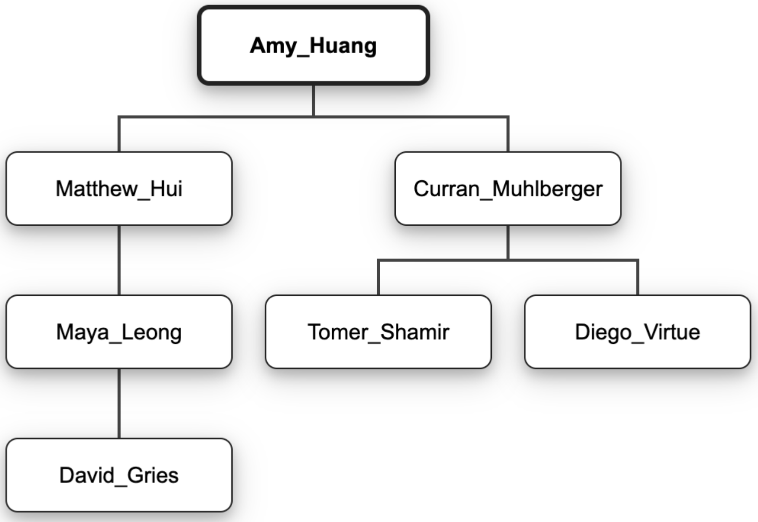
- `Amy_Huang.findAcademicLineage(Tomer_Shamir)` returns the list `[Amy_Huang, Curran_Muhlberger, Tomer_Shamir]`
- `Amy_Huang.findAcademicLineage(Amy_Huang)` returns `[Amy_Huang]`
- `Amy_Huang.findAcademicLineage(Tanvi_Namjoshi)` throws `NotFound`
- `Matthew_Hui.findAcademicLineage(Curran_Muhlberger)`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

To Do #8
Implement `commonAncestor()` according to specifications.

Again, do not use `findAdvisor()` for this: it is too inefficient. Instead, use `findAcademicLineage()` to find the respective routes r1 and r2 to the two advisees. Using the notation r1[i] to represent the node at position i in route r1, then if both advisees are found, two facts are known:

1) r1[0] = r2[0]

2) The answer r1[i] is the largest i such that r1[i] = r2[i]. If this is not clear, draw an example!

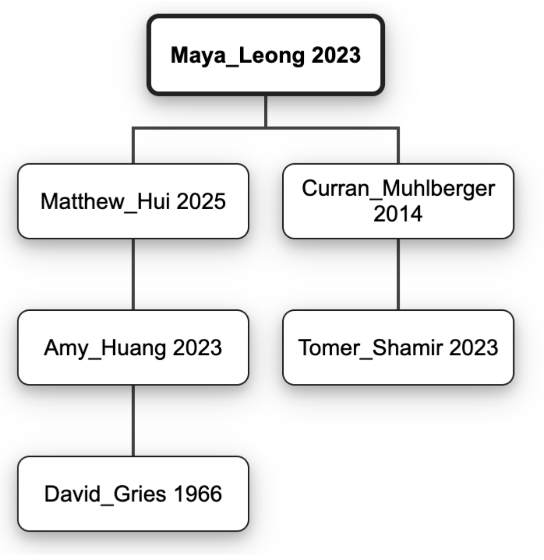


```
Amy_Huang.commonAncestor(Matthew_Hui, Amy_Huang) is Amy_Huang
Amy_Huang.commonAncestor(Matthew_Hui, Matthew_Hui) is Matthew_Hui
Amy_Huang.commonAncestor(Matthew_Hui, Curran_Muhlberger) is Amy_Huang
Amy_Huang.commonAncestor(David_Gries, Diego_Virtue) is Amy_Huang
Matthew_Hui.commonAncestor(Matthew_Hui,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

To Do #9

The final function to implement in this file is `toStringVerbose()`. Please implement it according to specifications. This method should be recursive.

For example, for the following tree:



Maya_Leong.toStringVerbose() should return a string containing the following:

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

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 alphabetically with respect to each other (e.g., Hui is before Muhlberger). This ordering should correspond to a preorder traversal of the tree, since advisees are stored in a sorted set, ordered alphabetically.

2c) Main.java
With a solid `PhDTree` implementation, you are ready to begin implementing your `Main` class. In this class, much like in A3, you will implement a function that converts a 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.

As part of the source files for this assignment, you see two CSV files: `professors.csv`, which contains about a hundred of rows of genealogy data, and `professors-shortened.csv`, which contains less data. Do not modify the CSVs given. 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`! And of course you can build your own academic genealogy files as well.

The format of both CSVs is such that the first row is the header: `advisee,year,advisor`. Then, the following rows detail the relationships in this genealogy, with the rows introducing advisors always preceding those of their advisees. For example, consider this CSV data:

```
advisee,year,advisor
Maya Leong,2019
Matthew Hui,2021,Maya Leong
Amy Huang,2022,Maya Leong
```

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 Amy Huang are advisees of Maya Leong, with the given graduation years.

The main methods to be implemented in this file are `csvToTree()`, which parses a valid CSV file in the format given above, and `processCommands()`, which handles inputs from the console. Note that an invalid CSV file that does not follow the format given above would violate the preconditions. In your implementation of the CSV parsing, you do not need to use the header row. You can consider it as existing for readability of the CSV file. You are not allowed to use third-party libraries to parse the CSV or handle processing of console commands; stick to classes in the Java standard library.

Hint: Some classes that may be useful to explore in the Java API are `BufferedReader`, `FileReader`, `File`, and `Scanner`. That being said, there are multiple valid ways of implementing these methods that fulfill the specifications without violating the set guidelines, so do not feel pressured to follow this hint if you have found another way that works.

Preparations. Run class `Main` in IDEA. You should see the prompt "Please enter a command: ". Type "help" and press Enter; the output describes the features you will be adding. Then type "exit", followed by Enter, to stop running the program.

Next, edit the Run Configuration for Main and add the "`-ea`" flag under "VM Options" to enable assertions. Get in the habit of doing this whenever you click a green arrow for a new class for the first time.

To Do #1
Implement `csvToTree()` according to the specifications.

To Dos #2–8
Finish implementing `processCommands()` by writing the helper methods `doContains()`, `doSize()`, `doAdvisor()`, `doAncestor()`, `doFind()`, `doPrint()`, and `doLineage()`. Here is the documentation for the commands that `processCommands()` will support:

- *help*. This command should print out all other commands your program supports.
- *contains <first name> <last name>.* This command should check if the PhD tree has a professor whose name corresponds to *<first name> <last name>*. Print "This professor is contained in the PhD tree." if it is contained, and otherwise print. "This professor is not contained in the PhD tree."
- *size <first name> <last name>*. This command should print the size of the PhD tree whose root is the node where the named scholar is found, in the format "The number of nodes in this tree is: *n*." where *n* is the number of nodes in the tree. If they are not found in the tree, it prints "This person does not exist in the tree.". The name may be omitted, in which case the command should report the size of the entire tree in the same format.
- *advisor <first name> <last name>.* This command should find and print the advisor of the professor whose name corresponds to *<first name> <last name>* in the format "The advisor of this advisee is: *n*." where *n* is the full name of the advisor. If a professor whose name corresponds to *<first name> <last name>* does not have an advisor, print "This person does not have an advisor." If they do not exist in the tree, print "This professor does not exist in the tree."
- *ancestor <first name 1> <last name 1> <first name 2> <last name 2>.* This command should find and print the most recent common ancestor of the professors whose names correspond to *<first name 1> <last name 1>* and *<first name 2> <last name 2>* in the format "The common ancestor of these scholars is: *n*." where *n* is the name of the common ancestor. If no common ancestor exists, then print "These scholars do not have a common ancestor."
- *find <first name> <last name>.* This command should find and print the PhD tree whose root is the professor whose name corresponds to *<first name> <last name>*. The printout should have the form "The PhDTree with *<firstname> <lastname>* at the root is *<tree>*." where <tree> stands for the output generated by the `toString()` method. If a professor *<first name> <last name>* does not exist in the tree, it should print "This person does not exist in the tree."
- *lineage <first name> <last name>.* This command should find and print the sequence of professors that are related and come before the professor whose name corresponds to *<first name> <last name>*. The format of the output should be "The lineage is: *n*." where n is a dash-separated string of names beginning from the root and ending in the professor whose name corresponds to *<first name> <last name>*. For example, referring to the sample CSV defined previously, *lineage Amy Huang* should print "The lineage is: Maya Leong--Amy Huang." If a professor whose name corresponds to *<first name> <last name>* does not exist in the tree, print "This person does not exist in the tree."
- *print*. This command prints out all the scholars in the tree, in the format returned by the method `toStringVerbose()`, which is described above.
- *exit*. This command should end the reading of inputs from the console.

When typing in these commands, make sure that you use the correct spelling (double-check that the upper and lower case letters match) for any <*first name*> or <*last name*>, with the definition of correct case being the case used in the input CSV files. You are, of course, free to be creative and add your own commands to explore the tree, but the following listed above are required for this assignment.

2d) Testing

As in A3, you will be using JUnit to test your PhDTree.

1. There should be a folder called `tests` in the a4 folder containing JUnit tests. If it is not already marked as a test sources root (colored green), right-click the `tests` folder and select from the dropdown: Mark directory as → Test Sources Root
2. You will need to add JUnit 5 as a dependency of your project. One way to do this is to open PhDTreeTest and right-click on one of the red "@Test" annotations, then select "Show Context Actions". In the resulting menu, select "Add JUnit5.8.1 to classpath" (it's okay if the version number is a little different, but it should start with "5"). Click "OK" in the popup dialog, and the errors in the file should go away.
3. Start by running all of the given tests (constructorTests and getterTests should pass out of the box). Then modify the PhDTreeTest run configuration and ensure that "-ea" is specified in the VM Options box. Now you're ready for test-driven development, re-running your test suite after every change.
4. In the file `PhDTest.java`, add at least 3 different tests for each method that you implement.

Notes on testing:

- To the extent practical, each method in `PhDTree` should be tested in a different test method. Tests that do not need to be in one method should be in separate methods.
- Make sure that you include `@Test` directly above **all** of your test methods. Test methods that do not have `@Test` will not be run by JUnit.
- Do not use assert statements and/or try–catch statements in tests (Note that assert statements are not the same as `assertEquals()` method calls. You should be using `assertEquals()`).
- You must test all the methods that you implement with at least **3 different** test cases. Think about what tests are needed to achieve coverage in both a black-box and a glass-box sense.
- Do not use a tree constructed with the CSV files to test your `PhDTree` class. Instead, use the constructors you have implemented to construct a `PhDTree` to test. Consider writing helper methods that create trees to be used as starting points by multiple tests.

As in A3, you will also be submitting input tests that provide coverage of the functionality of your Main class. Input tests should be submitted in a zip file named `input-tests.zip`. Your tests should cover each of the commands implemented in Main. Input tests are scripts 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 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.

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 help you in testing, 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 shown in the file `input-tests/test1/output.txt`. Remember that program arguments can be specified in IDEA by editing the Run Configuration for Main.

4. 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 `Main.java`, `PhDTree.java`, `PhDTreeTest.java`, `input-tests.zip`, and `summary.txt` to **Assignment 4** on CMSX before the deadline. Remember to first make a group on CMSX with your partner :)

Smoke testing will be available. Look for the appearance of the assignment A4 Smoke Test, which is where submissions should be made to run the smoke test on your code.

Last Note: If you do not know where to start, are stuck, do not understand testing, or are lost, please see someone on course staff immediately. This can be in the form of an instructor, TA, or consultant. Good luck, and happy coding :)