A4 – Modeling Diseases with Trees
Developed by Michael Patashnik

Table of Contents

1. Introduction

Note: Please keep track of the time you spend on this assignment. You will have to give it to us when you submit.

A4 uses trees to model the spread of an infectious disease. The root of the tree represents the first person to get the disease, and the children of each node represent the people who were infected by contact with the person represented by that node.

Most of the code you write for A4 involves using recursion to explore a tree. We have supplied you with starter code, developed by Michael Patashnik, that simulates the propagation of the disease as well as a GUI (Graphical User Interface) that allows you to set parameters and visualize the results on the screen. But Michael's simulation won't work until you write recursive methods to process the tree.

Learning objective: Become fluent in using recursion to process data structures such as trees.

Collaboration policy: You may do A4 with one other person. If you are going to work together, then, as soon as possible —and certainly before you submit the assignment— visit the CMS for the course and form a group. Both people must do something to form a group: one person proposes and the other accepts. Need help with the CMS? Visit www.cs.cornell.edu/Projects/CMS/userdoc/.

If you do this assignment with another person, you must work together. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. You should take turns "driving" —using the keyboard and mouse— and "navigating" —reading and reviewing the code on the screen.

Academic Integrity: With the exception of your CMS-registered partner, you may not look at anyone else's code from this semester or a previous semester, in any form, or show your code to anyone else, in any form. You may not show or give your code to another person in the class.

Getting help: If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY—a course instructor, a TA, a consultant, the Piazza for the course. Do not wait.

2. Disease propagation

Understanding how a disease propagates through a population is important for public health professionals, epidemiologists, politicians, etc. In this assignment, we use an extremely simple model of an infectious disease, where in each time step, an individual may become infected with some probability given as a parameter. This model provides a real world example of the general tree data structure. If you're interested in learning more about using Math/CS to model and understand diseases, check out some of these resources:

- http://idmod.org/home
- https://en.wikipedia.org/wiki/Mathematical_modelling_of_infectious_disease
- http://ocw.jhsph.edu/courses/epiinfectiousdisease/pdfs/eid_lec4_aron.pdf

You are not responsible for writing any of the code that implements the simulation of the disease.

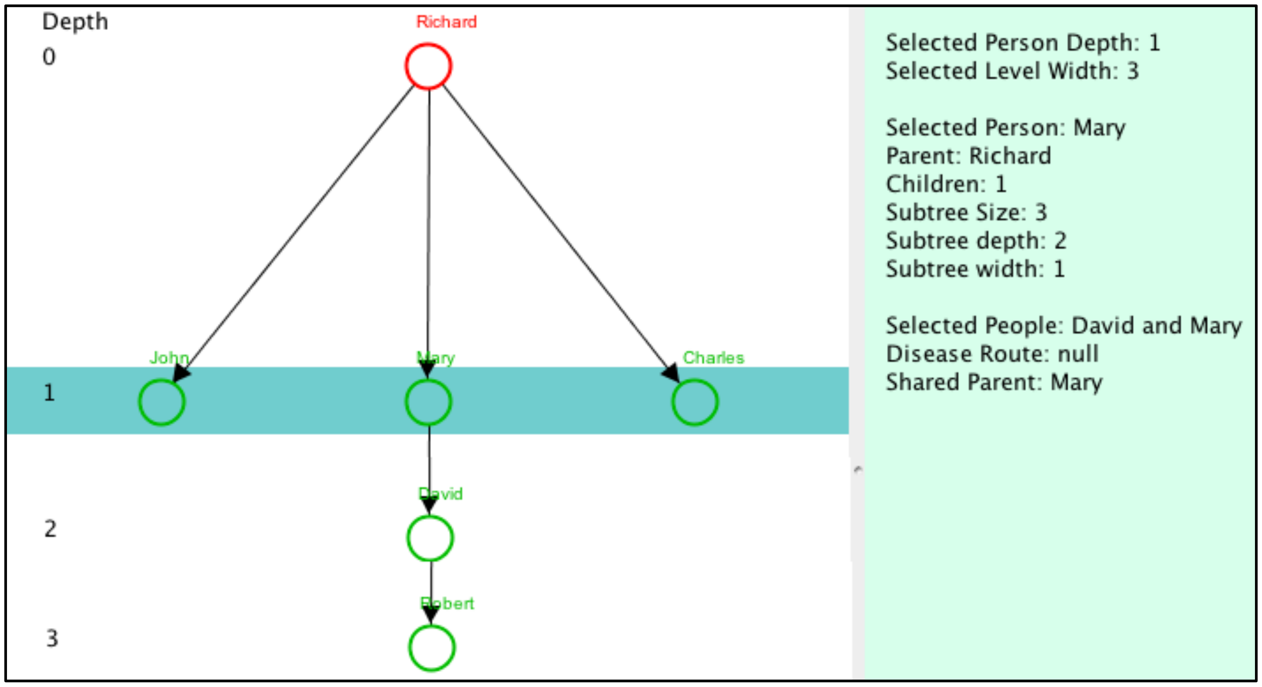
The model is initialized with a population of people —1, 2, 10, 50— however many you choose. Each person has a health range person, say 0..10. Here, a health of 10 means the person is very healthy while 0 means the person has died. At each time frame during the simulation, a sick person's health will decrease by 1 or the person may become healthy. Generally speaking, the larger the health range, the longer the program will run since there are more possible states of the simulation.

At the beginning of the simulation, a graph is constructed with people as nodes and with random edges between people, indicating they are in close proximity. You may supply a probability indicating that any two people are close. A probability of 1 means everyone is close to everyone, a probability of 0 means there are no edges. When you start the program, you supply two other probabilities: the probability that a person will get sick and the probability that a person will become immune (and thus healthy).

Once you have given this input, the program starts by making one random person sick and making that person the root of a new disease tree. Initially, that is the only node in the tree.

A series of time steps follows. In each time step, several things happen. (1) one random healthy neighbor of each sick person may catch the sickness and be added to the disease tree as the sick person's child. (2) Each sick person will get sicker (their health decreases) or may become well and immune from the disease. (3) A sick person whose health decreases to 0 dies.

These time steps continue until everyone in the disease tree is either healthy or dead. At that time, the results are shown in a GUI on your monitor.



Selected Person Depth: 1
Selected Level Width: 3

Selected Person: Mary
Parent: Richard
Children: 1
Subtree Size: 3
Subtree depth: 2
Subtree width: 1

Selected People: David and Mary
Disease Route: null
Shared Parent: Mary

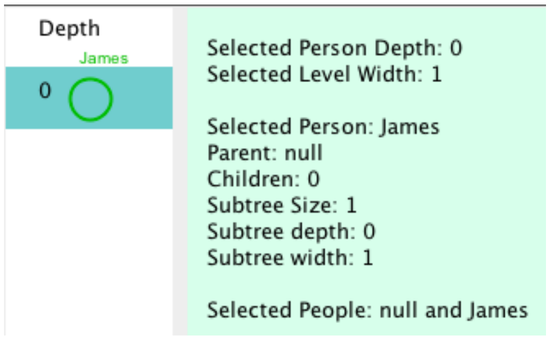
Above is an example of the output in the GUI. At depth 0 (the root) is Richard; at depth 1 is John, Mary, and Charles; at depth 2 is David, and at depth 3 is Robert.

Richard got sick first; he infected those on level 1; Mary infected David, and David infected Robert. Six people got ill but most of them survived; only the root person, the one who got sick first, died.

To the right in the image above is data that comes from selecting (with your mouse) David and then Mary. It shows: Mary's depth, 1; the width at (number of nodes at) that level, 3; Mary's parent, number of children, subtree depth, and subtree width; and the shared ancestor of David and Mary.

If John and Charles had been selected, their shared ancestor would have been the root, Richard.

A run may result in a tree with one node, as shown to the right. This happens when the one sick person doesn't infect anyone else, either because they have no neighbors or because when generating a random number to test whether a neighbor is sickened, the result is always "no".



Selected Person Depth: 0
Selected Level Width: 1

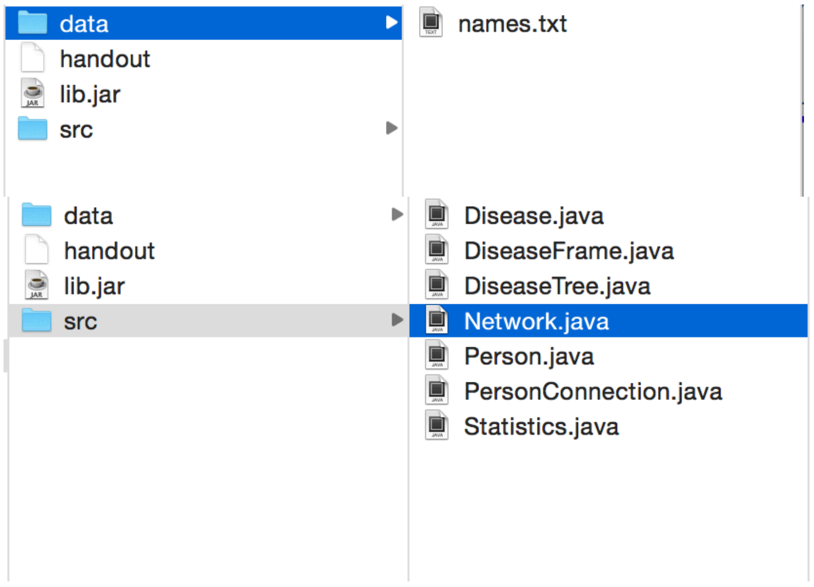
Selected Person: James
Parent: null
Children: 0
Subtree Size: 1
Subtree depth: 0
Subtree width: 1

Selected People: null and James

3. Installation

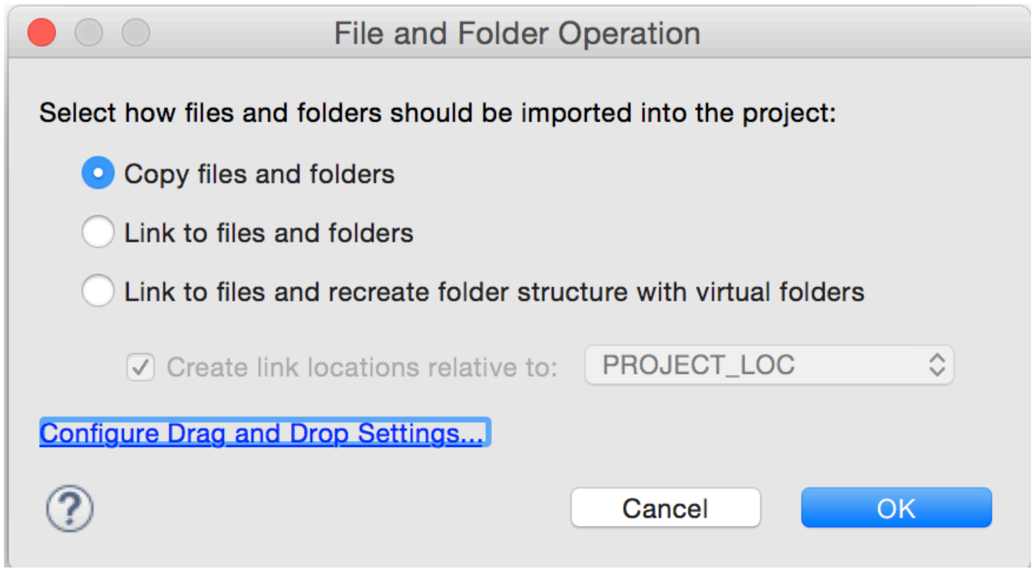
1. Download the A4 assignment zip file from the CMS or the course website.

2. Unzip the downloaded file. The folder should contain two folders, data and src; a file lib.jar; and this handout. The two folders should have these contents:



3. Create a new Java project (called A4). **IMPORTANT** – you must use java 1.8 for this project. Earlier assignments would have worked with java 1.7, but some of the code relies on functionality specific to java 1.8.

4. Copy-paste all four items in the downloaded folder into the root of the new Java project in Eclipse (i.e. data, handout, lib.jar, and src). When asked how to copy, select "Copy files and folders", then OK.


File and Folder Operation

Select how files and folders should be imported into the project:

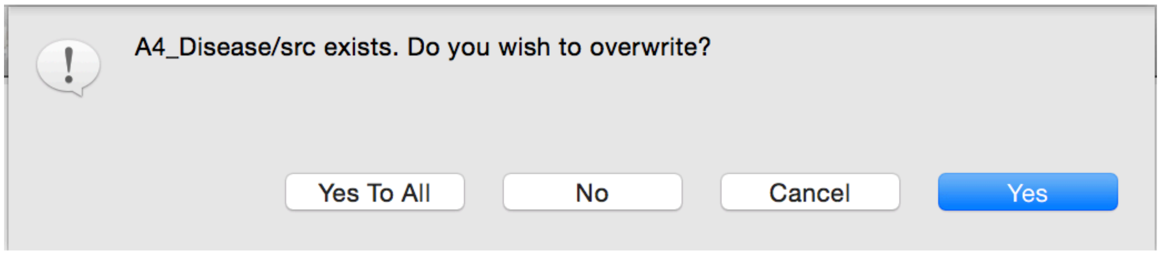
[✓ Copy files and folders]

Link to files and folders

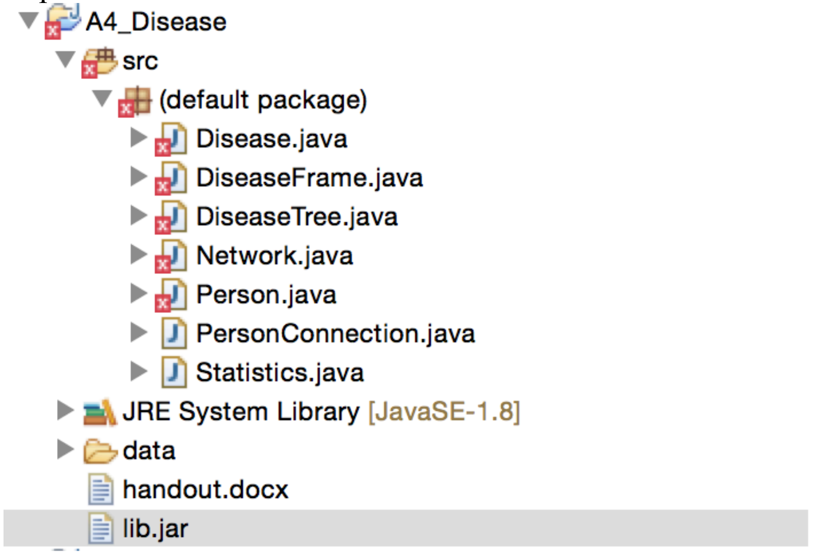
Link to files and recreate folder structure with virtual folders

☑ Create link locations relative to: PROJECT_LOC

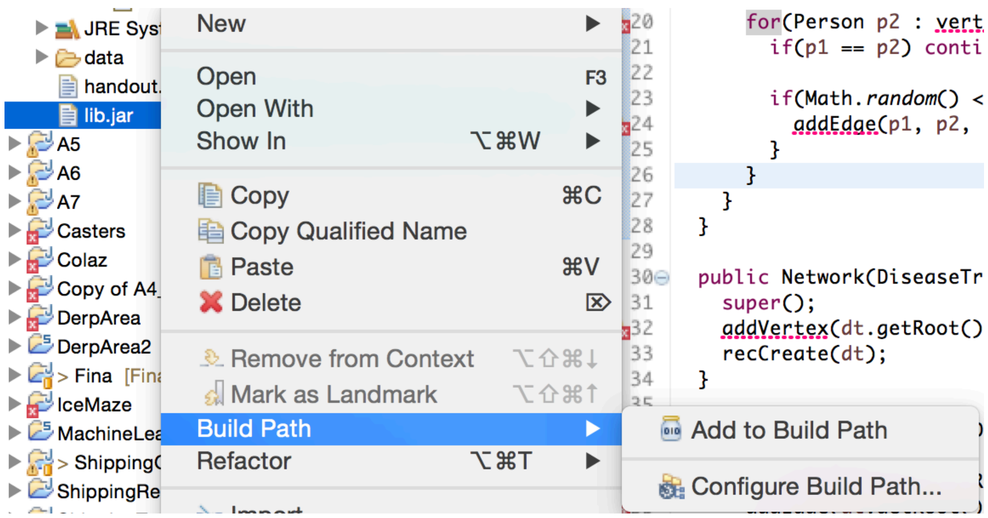
When asked what you want to do about the folder already named src there, click "Yes" or "Yes to All"


A4_Disease/src exists. Do you wish to overwrite?

5. Right-click the project root and select refresh (F5). You should then see the following project structure. Many of the files (as seen below) will have errors on them. This is resolved in steps 6..7.

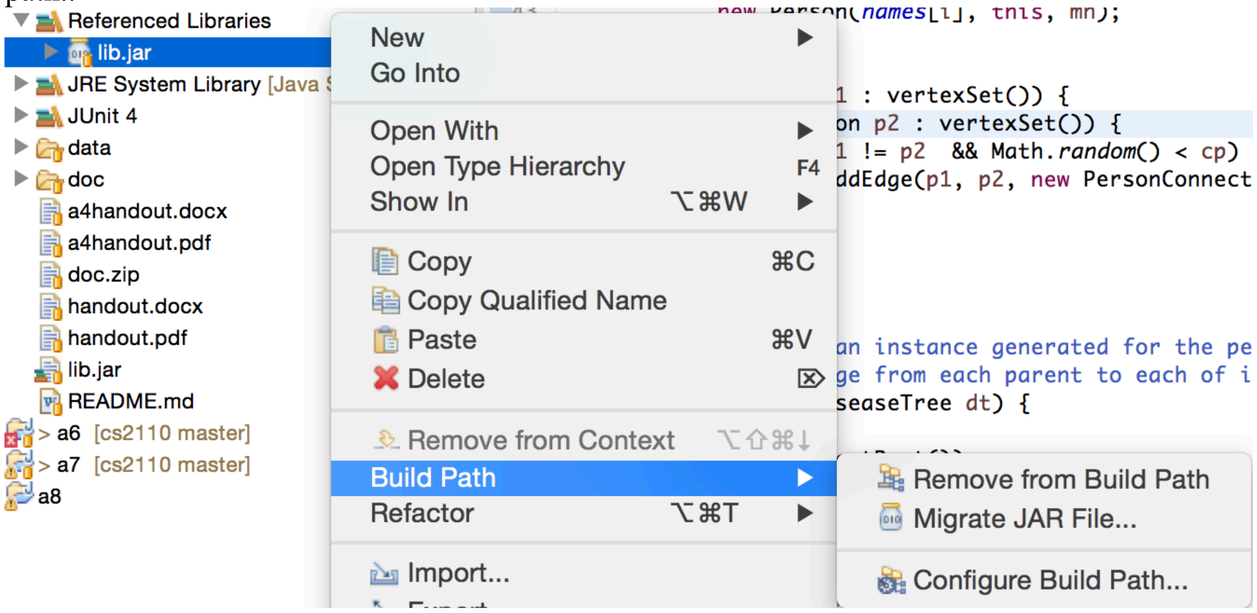

```
▼ A4_Disease
  ▼ src
    ▼ (default package)
      ▶ Disease.java
      ▶ DiseaseFrame.java
      ▶ DiseaseTree.java
      ▶ Network.java
      ▶ Person.java
      ▶ PersonConnection.java
      ▶ Statistics.java
  ▶ JRE System Library [JavaSE-1.8]
  ▶ data
    handout.docx
    lib.jar
```

6. Add lib.jar to the build path by right clicking lib.jar and selecting Build Path → Add to build path



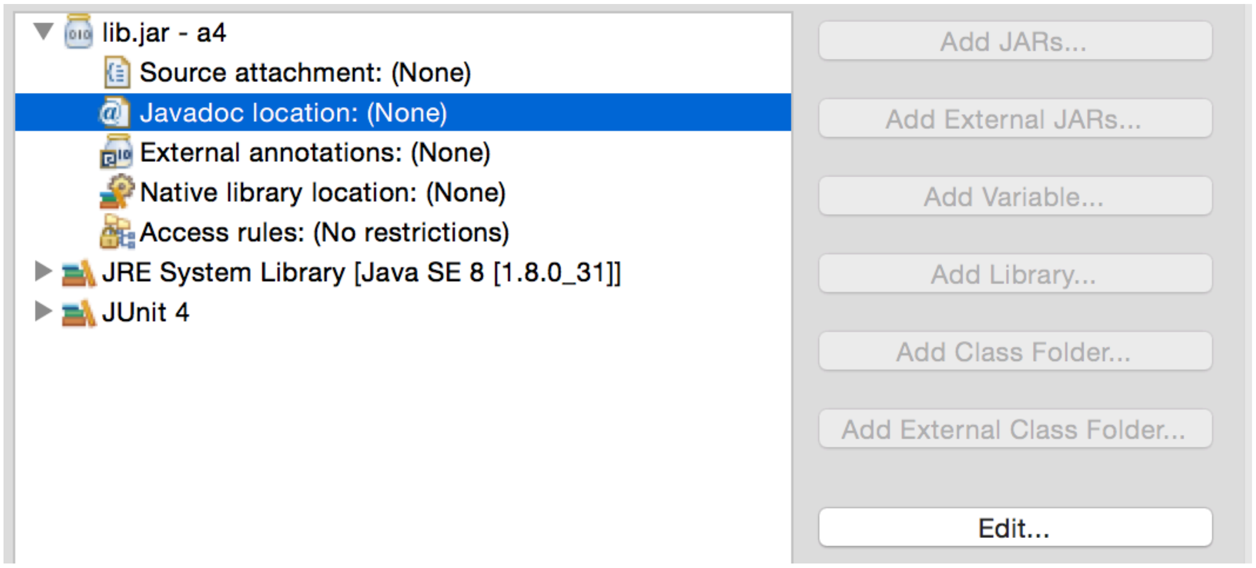
```
    for(Person p2 : vert
      if(p1 == p2) conti

      if(Math.random() <
        addEdge(p1, p2,
      }
    }
  }
}

public Network(DiseaseTr
  super();
  addVertex(dt.getRoot()
  recCreate(dt);
}
```

7. Attach the javadoc to the lib file by selecting lib.jar → build path → configure build path..



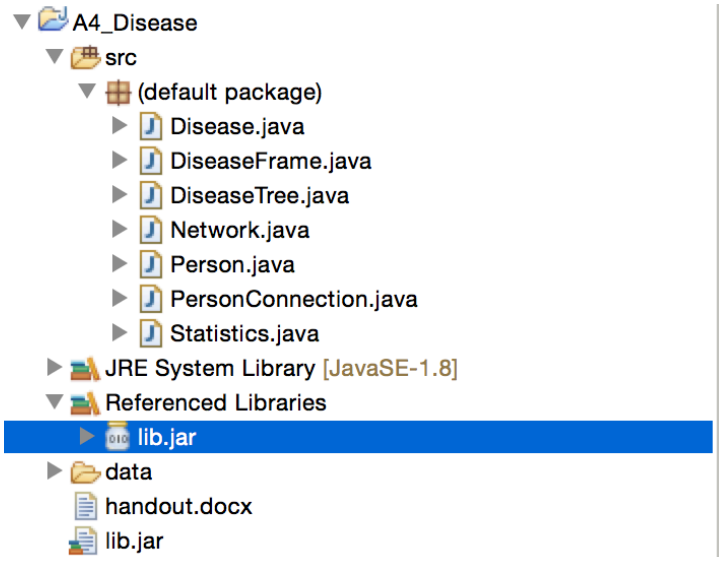
```
new Person(names[i], this, mh);

1 : vertexSet()) {
on p2 : vertexSet()) {
1 != p2  && Math.random() < cp)
ddEdge(p1, p2, new PersonConnect

an instance generated for the pe
ge from each parent to each of i
seaseTree dt) {
```

Select Javadoc location, click Edit… click the top browse button, navigate to the doc folder, and select open, then ok, then ok.



lib.jar - a4
Source attachment: (None)
Javadoc location: (None)
External annotations: (None)
Native library location: (None)
Access rules: (No restrictions)
JRE System Library [Java SE 8 [1.8.0_31]]
JUnit 4

8. Add JUnit4.jar to the build path as follows: (1) Select the project, right click, and select Build Path -> Add libraries. (2) In the window that opens, select JUnit and click Next; in the window that opens, select JUnit 4 and click Finish. (3) Refresh your project root (f5) again. Your project structure should look like this.



```
A4_Disease
  src
    (default package)
      Disease.java
      DiseaseFrame.java
      DiseaseTree.java
      Network.java
      Person.java
      PersonConnection.java
      Statistics.java
  JRE System Library [JavaSE-1.8]
  Referenced Libraries
    lib.jar
  data
  handout.docx
  lib.jar
```

4. Running

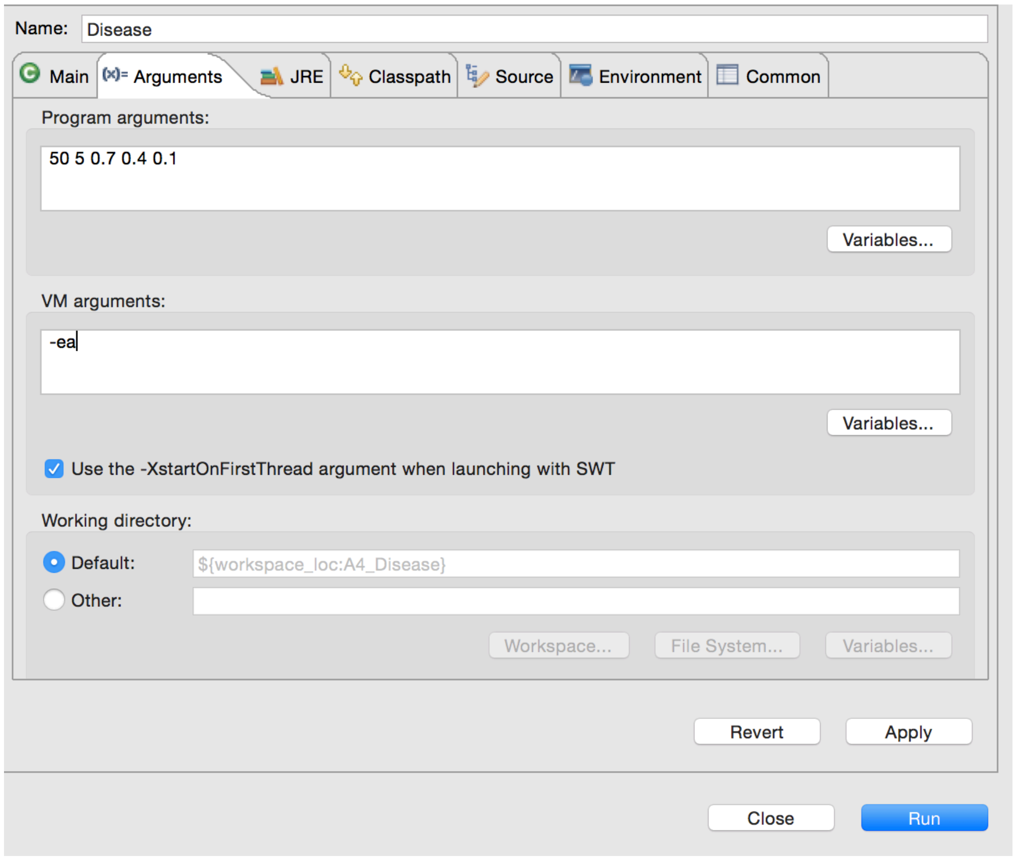
The executable class of this project is Disease.java. To run the project, open Disease.java and click run (green button with white arrow) or use menu item Run -> Run. You will be prompted for a few seeding values via the console at the bottom of eclipse. These are:

1. Size of population: how many people to model. A positive integer. A higher number may result in a larger tree.

2. Amount of health per person: how long a person can be sick before dying. A positive integer. A higher number may result in a larger tree.
3. Probability of connection: how likely two random people are to come into contact with one another. (On page 2, we called them neighbors). A float in the range [0,1]. A higher number may result in a larger tree.
4. Probability of becoming sick: how likely a person is to become sick when in contact with a sick person. A float in the range [0,1]. A higher may number result in a larger tree.
5. Probability of becoming immune: how likely a sick person is to become immune (fight off the disease). A float in the range [0,1]. A *lower* number may result in a larger tree.

A nice set of starting values is: (50, 5, 0.7, 0.4, 0.1)

If you want to run Disease.java repeatedly with the same five parameters, you can put them in the program arguments instead of having to type them into the console every time you run the program. This is done the usual way (Run Configurations… → arguments). For example, the following run configurations would run the program with the above arguments every time the run button is clicked.


Name: Disease

Program arguments:
```
50 5 0.7 0.4 0.1
```

VM arguments:
```
-ea
```

☑ Use the -XstartOnFirstThread argument when launching with SWT

Working directory:
● Default: ${workspace_loc:A4_Disease}
○ Other:

If there is any issue with the arguments provided, either though the console or the program arguments (health is less than 0, for example), you will be re-prompted to enter the arguments through the console. Thus, if you have entered arguments in the arguments tab but are still prompted to enter arguments via the console, there may be something wrong with the arguments you entered.

From there, the disease will run —starting with a randomly chosen patient and spreading across that person's connections until no one is left alive and infected.

After the disease has finished running, the full tree is printed out by calling toStringVerbose(); then the tree explorer GUI pops up.

5. Your Tasks

The only file you have to edit to complete this assignment is DiseaseTree.java. Before jumping into the methods, be sure to thoroughly read the javadoc description of the class at the top of the file.

In order to complete the assignment, complete each method marked with a //TODO comment to the specification listed above the method. You may assume that all preconditions to these methods are respected. In the case that they are not, any behavior (even non-deterministic) is acceptable. It's best to leave the //TODO comments in —note that they are marked in blue on the right of the text in Eclipse.

Recursion is your friend! The bulk of these functions are best and most easily written using recursion. Iteration (using loops instead of recursive calls) may be possible but will certainly be more work both to reason through and to debug.

Hint: Many of the functions you are asked to write can be written very simply or even trivially (one or two lines) simply by relying on previous functions you have already written or ones that we wrote. The order in which the //TODOs are given (top to bottom) and numbered in DiseaseTree.java is the order in which to implement them.

Warning: Every time application Disease (i.e. method main in Disease.java) is called, a new, random, unrepeatable DiseaseTree is created, so you cannot debug the methods you are writing using that application. It is best to use a JUnit testing class to test your methods and to run the program only when you know your methods are correct.

Dos and Donts:

- Do read all the Javadoc in DiseaseTree.java very thoroughly. You may choose to read the Javadoc in other files, but it should not be too important. Read the files outside the default package only if you are particularly interested in them —you shouldn't need to know more than their javadocs in order to complete the assignment.
- Do not alter any of the other files given to you. You won't be able to submit them, so your DiseaseTree.java must work with unaltered versions of the other files.
- Do not change the method signatures of any method in DiseaseTree. The name and types of parameters should not be changed.
- Do not have println statements (which you added to help debug) in your code when you submit. Comment them out or delete them before submitting.
- You may add new methods to DiseaseTree to help complete the required functions (some are only workable with the use of helper methods). Make sure that methods you add are **private** and have a javadoc comment (/** … */) specification.

6. Debugging

It is strongly recommended that you create a JUnit test file to systematically test the functionality of DiseaseTree. Use the same testing practices you learned since A1 and used in A2 and A3. You won't be submitting the test file, so creating it isn't required. However, since Disease.java is entirely probabilistic, a JUnit file is the only way to systematically ensure that your DiseaseTree is correct.

Debugging your tree method can be difficult. It's harder than with linked lists, where we were easily able to test *all* fields using toString(), toStringRev(), and size().

We will place on the A4 FAQ note on the Piazza, some help to get started on testing and debugging.

7. What to submit

Complete the information at the top of file DiseasteTree.java: your netid(s), the hours and minutes that you spent on this assignment, and any comments you would like to make on this assignment.

Submit file DiseaseTree.java on the CMS.