A4 – Modeling Bugs with Trees

Table of Contents

1. Introduction

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

We all get bugs, or infections, or viruses, from time to time. Sometimes they are just an annoyance. Sometimes they are dangerous —like the earlier zika virus. For a dangerous one, the United States Centers for Disease Control and Prevention (CDC) may track how it is spreading and attempt to determine where it will spread next, so preparations can be made to fight it. People develop programs to simulated the spread of bugs in order to learn about them, just as programs are written to simulate weather. If you are interested in learning more about using Math/CS to model and understand infectious bugs, 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

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

In A4, you write code to manipulate and explore the tree. We supply you with code that simulates the propagation of a bug as well as a GUI (Graphical User Interface) that allows you to set parameters and visualize the results on the screen. But the simulation won't work until you write several 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.

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 from a similar assignment in 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 others until after the last deadline for submission. Finally, groups must follow the rules stated above in section *Collaboration policy*.

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

Tutorials: Before starting to work on A4, watch the two tutorials about recursion on trees in the JavaHyperText: http://www.cs.cornell.edu/courses/JavaAndDS/recursion/recursionTree.html

2. Bug-infection propagation

A4 uses a very simple model of an infectious bug, which provides a real-world example of the general tree data structure. You are not responsible for writing any of the code that implements the simulation of the bug spreading, but the simulation won't work until you write methods that manipulate trees.

The model is initialized with a population of people —1, 2, 10, 50— however many you choose. Each human has a health range, say 0..10. A health of 10 means the human is very healthy and 0 means the human has died. Generally speaking, the larger the health range, the longer the program will run since there are more possible states of the simulation. In the beginning, everyone is as healthy as possible.

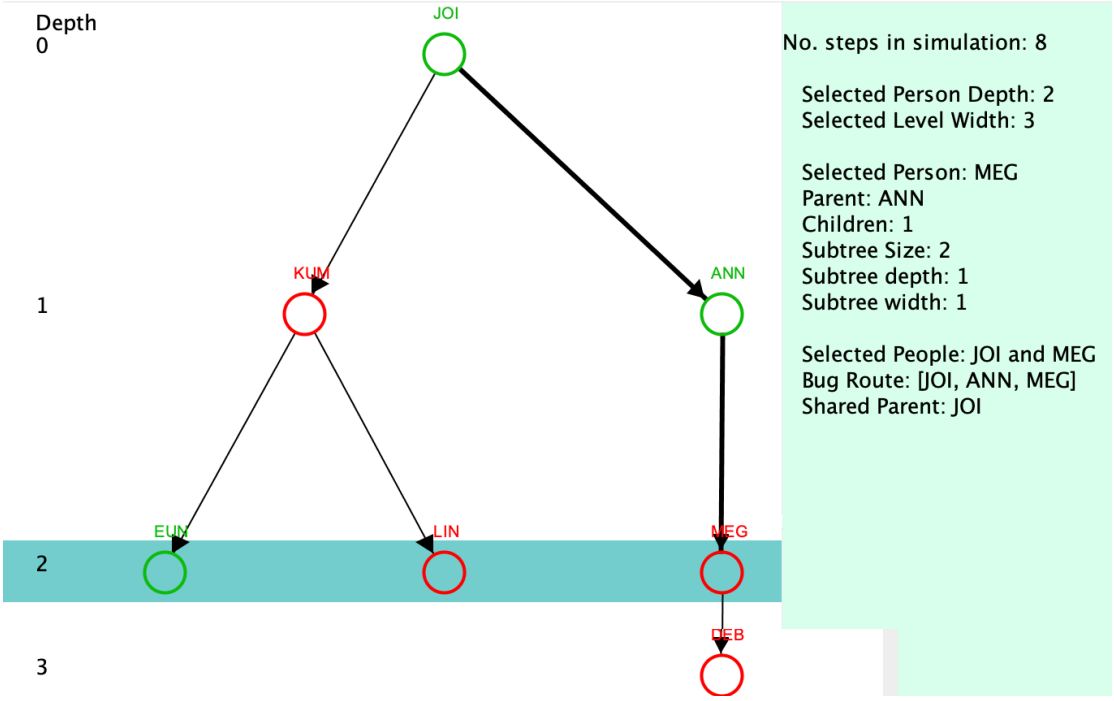
At the beginning of the simulation, you supply several probabilities: (1) The probability that any two humans are close to each other —call them *neighbors*. A probability of 1 means everyone is close to everyone; a probability of 0 means no one is close to anyone. (2) The probability that a human will get the bug. (3) The probability that a human will become immune —once immune, the person is healthy forever.

Once you have given this input, the program starts by making one random person sick and making that person the root of a new bug 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 human may get the bug and be added to the bug tree as the sick human's child.
(2) Each sick human either gets well and henceforth immune from the bug or gets sicker (their health decreases).
(3) A sick human whose health decreases to 0 dies.

These time steps continue until everyone in the bug tree is either healthy or dead. When the simulation is over, the results are shown in a GUI on your monitor.

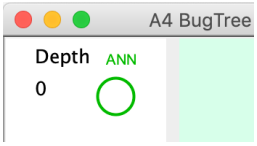


Above is an example of the output in the GUI. JOI is at depth 0 (the root); KUM and ANN on are at depth 1, EUN, LIN, and MEG are at depth 2, and DEB is at depth 3.

JOI got the bug first; JOI infected KUM and ANN; KUM infected EUN and LIN, ANN infected MEG, and MEG infected DEB. All of them got ill, and four died (KUM, LIN, MEG, and DEB, indicate by a red circle). The ones with a green circle are alive and healthy.

The information in the panel to the right appears only when you click some node of the tree. The data changes as you click on different nodes of the tree. The data that appears in the panel comes from selecting (with your mouse) JOI and then MEG. It shows: MEG's depth, 2; the width at (number of nodes at) that level, 2; MEG's parent; number of children; subtree size, depth, and width; and the shared ancestor of JOI and MEG.

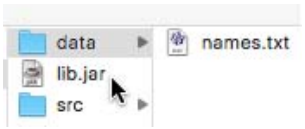
A run may result in a tree with one node, as shown to the right. This happens when the one human with the bug doesn't infect anyone, either because they have no neighbors or because when generating a random number to test whether a neighbor fets infected, the result is always "no".



3. Installation

1. Download the A4 assignment zip file from pinned Piazza note *Assignment A4*.

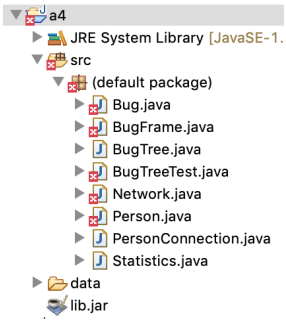
2. Unzip the downloaded file. The folder should contain two folders, data and src; and a file, lib.jar. Folder data should look as shown to the right.



3. Create a new Java project (call it a4, for example). **IMPORTANT** – you must use java 1.8 for this project.

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

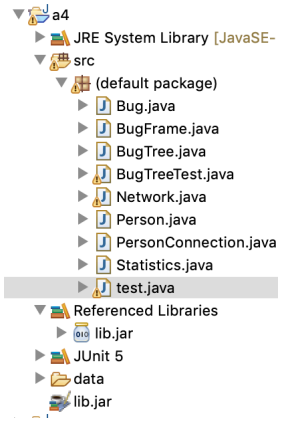
To the right, you see what folder src should be. Note that many of the files have syntax errors that will be fixed in a moment.



4. Add lib.jar to the build path by right clicking lib.jar and selecting Build Path → Add to Build Path. After this, the only file that should show an error is BugTreeTest.java.

5. Create a new JUnit testing class as usual (File → New → JUnit Test Case). Select *New JUnit Jupiter test*. Later, if it asks whether JUnit 5 should be added to the build path, say "yes". Then delete the newly added JUnit test case file (because you don't need it).

Your project should now look as shown to the right, and you're ready to go!

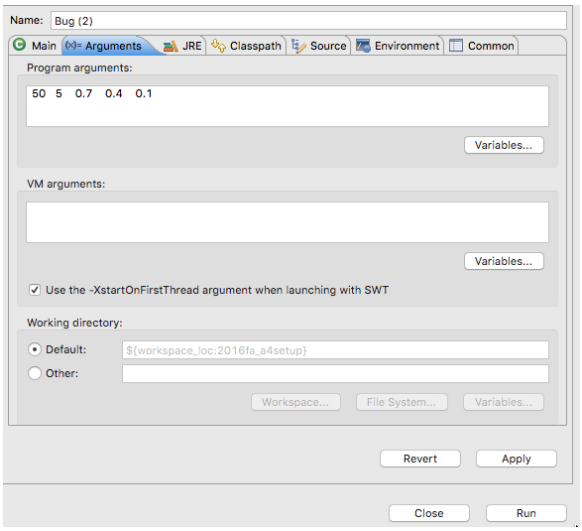


4. Running

The executable class of this project is Bug.java. To run the project, open Bug.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 humans to model. A positive integer. A higher number may result in a larger tree.
2. Amount of health per human: how long a human can have the bug before dying. A positive integer. A higher number may result in a larger tree.
3. Probability of connection: how likely two random humans are likely to be neighbors). A float in the range [0,1]. A higher number may result in a larger tree.
4. Probability of getting the bug: how likely a human is to get the bug when in contact with an infected neighbor. A float in the range [0,1]. A higher may number result in a larger tree.
5. Probability of becoming immune: how likely a human with the bug will become immune (fight off the bug). 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 Bug.java repeatedly with the same five arguments, 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 run configurations shown to the right would run the program with the above arguments every time the run button is clicked.

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 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 bug infection will run —starting with a randomly chosen patient and spreading across that human's connections until no one is left alive and infected.

After the simulation has finished, the full tree is printed out by calling toStringVerbose(); then the tree explorer GUI pops up. But the GUI doesn't print anything.

5. Your Tasks

All the code you write is in BugTree.java. In order to complete the assignment, complete each method marked with a //TODO comment (in Eclipse, they are marked in blue on the right of the text). Complete and test them in the order in which they are numbered. Do not delete the //TODO comments. If a precondition is false, any behavior is acceptable.

Do the following before you code:

1. Read the comments at the top of BugTree.java.
2. Study the already written methods in BugTree.java. You will learn a lot by that.
3. You *must* read the Piazza pinned A4 FAQs note. It continues useful and necessary info.

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: Some 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.

Warning: Every time application Bug (i.e. method main in Bug.java) is called, a new, random, unrepeatable BugTree is created, so you cannot debug the methods you are writing using that application. It is best to use the JUnit testing class to test your methods and to run the program only when you know your methods are correct. See the Piazza A4 FAQs note.

Dos and Don'ts:

- Do read all the Javadoc in BugTree.java 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 don'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 BugTree.java must work with unaltered versions of the other files.
- Do not change the method signatures of any method in BugTree. The name and types of parameters should not be changed.

- Do not leave println statements (which you may have added to help debug) in your code when you submit. You may lose up to 5 points. Comment them out or delete them before submitting.
- You may add new methods to BugTree to help complete the required functions. When we practiced the assignment, we wrote one or two. Make sure that methods you add are **private** and have good javadoc (/** … */) specifications.

6. Debugging

Debugging a 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 give you a JUnit testing class, BugTreeTest. It contains a testing procedure with one test case for every method you have to write. *It is up to you to add more test cases and test your functions thoroughly*. Use Eclipse's code coverage feature to help you. If you don't know about Eclipse's code coverage feature, type *coverage* into JavaHyperText and read about it. Also, don't forget to test extreme/corner cases.

You do not have to submit your test cases, as you did for A3.

Class BugTreeTest contains some new things for you to learn about in a testing class. We explain them here:

1. Static variables. The class has some static variables, Network n and Person[] people.
These will be set up initially and not changed thereafter.

2. Annotation @BeforeClass. The class has this code beginning on line 25:

```
@BeforeClass
public static void setup(){
    n= new Network();
    people= new Person[]{ … }
    personA= people[0];
    ...
}
```

The annotation @BeforeClass says that this method should be called before any others, while the class is "loading". This method creates a Network, n, and an array people to test with. You'll see in the later methods how array people is used. It is easier in places to use the static fields personA.

3. Testing method insert using method toStringBrief. Look at testing procedure test1insert, on about line 91. It creates a new BugTree, with name person for its root. The next line calls method insert to insert a node with name person as its root, and it stores the returned value in variable dt2.

Two calls on assertEquals follow. The first calls method toStringBrief(st). We wrote toStringBrief (near the end of the class) to create a one-line String representation of a tree. The one-line representation of the created tree is "[B[C]]". If B had three children C, E, and D, the representation would be "[B[C D E]]". Read the specification of that method to understand the format of that one-line representation.

Of course, method toStringBrief is recursive. Moreover, it sorts the children of each node on their root name of the children. Remember, the children of a node are maintained in a set, and if we want a unique representation of the children, they must be ordered in some fashion.

The second call on assertEquals tests whether method insert returned the correct tree by checking its root name.

4. Constructing trees with which to test. On line 54 is method makeTree1, with its specification. This is one way to make a tree. We use it in a few of the testing procedures. Of course, this will work only if your method insert is correct.

Look also on line 84, where testing procedure testMakeTree1 tests makeTree1 to make sure it is correct.

As you write the methods in class BugTree, to test them, you may want to create other trees, as we did in method makeTree1. You may also find it useful to make a copy of a tree. The second constructor of class BugTree does that. Take a look.

7. What to submit

Change the value assigned to static field BugTree.timeSpent from -1 to the time you spent to complete assignment A4. It is the first field in class BugTree. Read the comment above it for help in doing this.

In the comment at the beginning of file BugTree.java: put your netid(s), names(s), and comments on what you thought about the assignment.

Submit file BugTree.java on the CMS.

8. Suggested timetable for doing A4

You will find this assignment easier to accomplish and a more educational experience if you do it at a more leisurely pace, starting right now. When you run across a problem, you'll have time to think about it and to get help, if you need it. Our staff will have an easier time, too.

Here is a suggested timetable for A4.

07 March. Read the handout and comments at the top of BugTree.java, study the functions that are already written
08 March. Write and test insert, size, contains.
10 March. Write and test depthOf and widthAtDepth
11 March. Write and test bugRouteTo and sharedAncestorOf.
THESE ARE NOT EASY. Be prepared to spend time on them.
14 March. Write and test equal. This also will take time.
16 March. Look over everything.
17 March. Submit, before the deadline

9. Hints

1. To learn about the foreach statement, e.g. for (String s : set), see JavaHyperText entry "foreach loop".

2. Hints on depthOf(p). Suppose p is in the tree, p is not the root, and p is in the tree. Here's the definition of depthOf(p) in this case:

Let child c of the root contain p.
The depth of p in the root is 1 + (depth of p in c).

Do not use method contains or getTree! There is no need to use it. For any child c of p, if p is not in c, c.depthOf(p) returns -1. This is a recursive definition!

3. Hints on widthAtDepth(d). Suppose d > 0.
The width at depth d is then the sum of the widths of the children of the root at depth d-1.
Just translate this recursive definition into Java.
Read Piazza note @724 on “Stepwise refinement and appropriate recursive thoughts”.

4. Hint on bugRouteTo(c). The ONLY case that this method has to create a list is in the base case, when c is the root.
In the recursive case, just ask each child for the posting route to c, saving their answer in a List variable.
If a child returns a real list (instead of null), then all you have to do is prepend the root to that list and return.

Note: If you implement this calling getParent(c) repeatedly, you will LOSE POINTS.

5. Hints on sharedAncestorOf(c1, c2). There are a few ways to write this method. Do not do it by calling getParent repeatedly. Points will be deducted.
Here is how to do it. You already have function bugRouteTo. Find the route to c1, say a list l1, and the route to c2, say a list l2, and suppose neither is null.
Both l1 and l2 have the Person at the root of the tree as their first element. Look for the largest k such that perhaps l1[0..k-1] equals l2[0..k-1] but that doesn't hold for l1[0..k] equals l2[0..k]. That tells you that l1[k-1] is the shared ancestor you are looking for.

Thus, you have to process the two lists in parallel, looking first at their first elements, then their second, then their third, etc.

Do not write loops that use ll.get(i) obtain element number I of list ll! That yields an algorithm that quadratic in the length of the shortest list.

Instead, you can do it one of two ways: (1) Use a Collections function to change those list into arrays and then write a loop to process the arrays. (2) obtain an iterator for each of the two lists and use the two iterators.

6. Hints on equals(t). The specification outlines what must be done, in detail. Two points to discuss:

(1) Stay away from nested ifs as much as possible! Instead, as soon as it is determined that a property for equality is not met, return false! So the structure could be:

```
if (property 1 not met) return false;
if (property 2 not met) return false;
etc.
```

(2) The difficult part is testing this:

for every BugTree st in one set S1, there is a BugTree st2 in the other set S2 for which st.equals(st2) is true.

Here, set S1 are the children of this tree and S2 are the children of tree ob.

To help you implement this, we suggest you write a helper method with this specification:

```
/** Return true iff st equals some member of s2 */
private boolean help(BugTree st, Set<BugTree> s2)
```

Note that this method is going to call equals, like this: st.equals(...).
That is a call to the method you are trying to write! We have what is called "mutual recursion":
equals calls help, which calls equals, which call help ...
But when thinking about what a call does, USE THE SPECIFICATION to understand what it does