Data Structres II - HW1

Instructions:

1. You must already have Java and Eclipse installed correctly on your machine with the workspace for this class already created. There are links on D2L under **Content -> Software** that you can use to install these tools. Videos walking you through the process are available under **Content → Weekly Materials → Week 0 - Setting up Java and Eclipse**. If you have other versions of Java and/or Eclipse installed already, it is recommended that you uninstall them before installing the correct versions to eliminate the risk of inadvertently using the wrong version on your homework. Also, different versions of these tools don't always work well together.

2. Download the file hw1-files.zip

3. Extract the files from hw1-files.zip into the src folder that is inside the ds folder that is inside the workspace you set up previously for this class. Again, the videos in Week 0 walk you through this process.

4. Start up Eclipse. When asked for a workspace use the workspace you created while setting up eclipse. (It should provide you with that workspace as the default unless you have used eclipse with a different workspace.) Open up the src folder that is in the project you created for this class. If you do not see the hw1 package, right click the ds project and select Refresh. You should now see the hw1 package.

Note:

This assignment has two parts and multiple things to submit. Make you submit all four of the following:

- A copy/paste of lines 11-15 of your modified DebuggerExercise.java file in the submission text box (as described in the instructions below).
- Your modified DebuggerExercise.java file
- Your modified WordFrequencyAnalayzer.java file
- A screenshot of what happens when you run HW1Test.java

Again, detailed instructions for each of these parts appear in the pages that follow.

Part 1 – Using the Debugger (60 points)

1. Try running the DebuggerExercise.java file. You should see the following message in the console window at the bottom of the screen:
   ```
   Congratulations! You've found a way out of your labyrinth.
   Congratulations! You've found a way out of your twisty labyrinth.
   ```

2. Now change line 11 so that the String assigned to the YOUR_NAME variable is your full name. For example, if I were doing the assignment, I would change line 11 to be:
   ```
   private static final String YOUR_NAME = "Will Marrero";
   ```

3. Try running DebuggerExercise again. You should see the following message in the console window:
   ```
   Sorry, but you're still stuck in your labyrinth.
   Sorry, but you're still stuck in your twisty labyrinth.
   ```

4. Your task is now to change the Strings assigned to the variables PATH_OUT_OF_MAZE and PATH_OUT_OF_TWISTY_MAZE on lines 14 and 15 so that your get the success messages you saw in step number 1 with the new value of YOUR_NAME that you did in step number 2.

5. To do so, you will need to set a breakpoint on line 22 and then use the debugger to explore all the MazCells beginning with startLocation. You should draw out a picture of the various MazeCells and how they are connected together so that you can come up with a correct String value for PATH_OUT_OF_MAZE. Once you have solved that, you need to set a breakpoint on line 33 and repeat the process to find a correct String value for PATH_OUT_OF_TWISTY_MAZE. Details of what constitutes a correct path are provided below.

Maze Description

The MazeCell class is defined as follows:

```
public class MazeCell {
    public String whatsHere = ""; // One of "", "Potion", "Spellbook", and "Wand"
    public MazeCell north = null;
    public MazeCell south = null;
    public MazeCell east  = null;
    public MazeCell west  = null;
}
```

MazeCell objects are connected to each other via the north, south, east, and west links. Exactly one MazeCall has whatsHere equal to “Potion”, exactly one MazeCall has whatsHere equal to “Spellbook”, and exactly one MazeCall has whatsHere equal to “Wand”. All other MazeCell objects have whatsHere equal to “”. You must come up with a sequence of moves (N to follow a north link, S to follow a south link, E to follow an east link, and W to follow a west link) that gets you from the MazeCell stored in startLocation or twistyStartLocation to each of the cells containing the potion, spellbook and wand (the order in which you gather the objects does not matter). An example of each follows.

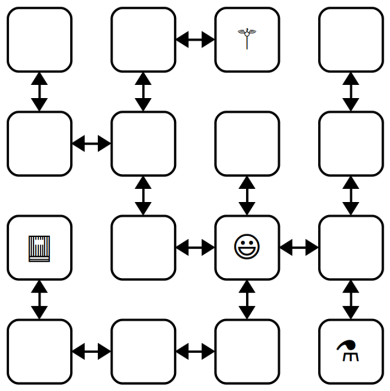
<u>First Maze (30 points)</u>

The variable startLocation is the starting cell in the easy maze. In this maze, the cells are arranged in a grid in such a way that the directions in and out of each cell are consistent. That is to say:

- If cell1.west == cell2 then cell2.east == cell1
- If cell1.east == cell2 then cell2.west == cell1
- If cell1.north == cell2 then cell2.south == cell1
- If cell1.south == cell2 then cell2.north == cell1

Note however, that the values in some directions may be null, indicating there is no passage in that direction (even though there might be a cell in that direction). You will want to explore the maze using the debugger to look at how the cells are connected to each other **<u>and draw a picture for yourself</u>**. You can then use the picture to come up with the path.

Below is a diagram of a potential easy maze. The smiley face indicates the value of stratLocation, and three other nodes have icons in them signifying the locations of the potion, spellbook, and wand.



If this picture is accurate, then here are three possible paths you could take to get you to the three objects starting from starting location:

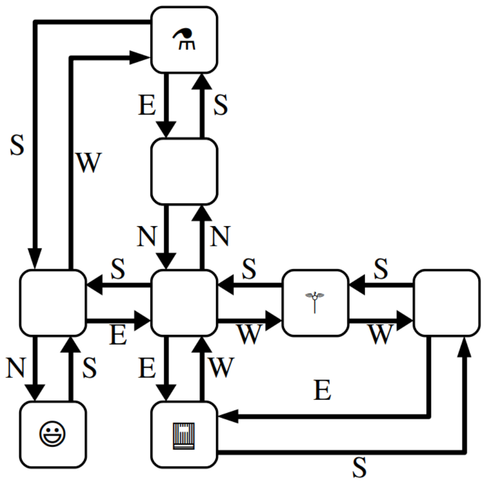
- ESNWWNNEWSSESWWN
- SWWNSEENWNNEWSSEES
- WNNEWSSESWWNSEENES

Any one of these would be a correct value to use for PATH_OUT_OF_MAZE.

Second Maze (30 points)

The variable twistyStartLocation is the starting cell in a difficult maze. In this maze, cells are again connected to each other via north, south, west, and east links, however, the directions are not consistent. For example, if cell1.north == cell2, it is possible that cell2.south is NOT cell1, but instead cell2.north == cell1!

Below is a diagram of a potential difficult maze.



Notice how in this maze, if we start in the bottom left (smiley face) and move S then E we get to a cell from which two N moves brings back to the same place! In this example, how would we know that we came back to the same node after two N moves? The debugger provides an ID value for every object. When you are looking at the vales of two variables and the object in both variables have the same ID, then they are the same object. So in this example, maybe the cell has ID=14, cell.north has ID=21 and cell.north.north has ID=14.

Submission

For this part of the assignment, you must copy and paste of lines 11-15 after you have made your changes into the text box for your submission. For example, I might copy and paste the java code below that we would then copy and paste into DebuggerExercise.java to test your solution.

```
private static final String YOUR_NAME = "Will Marrero";

/* Change these constants to contain the paths out of your mazes. */
private static final String PATH_OUT_OF_MAZE = "WNEESNWSWWSNESEEN";
private static final String PATH_OUT_OF_TWISTY_MAZE = "EWSWNESNEW";
```

(Note that this is only meant to demonstrate what a submission looks like and is not an actual solution.)

You should also submit your modified DebuggerExercise.java.

Part 2 – Fixing the WordFrequencyAnalyzer class (40 points)

1. Try running HW1Test.java which will be used to grade your HW1. To run it, select the HW1Test.java file and then click the run button. Many of the tests will fail because you have not written the code for HW1 yet. Close the Junit tab that appears above the failed tests.

2. You can now double click on WordFrequencyAnalyzer.java to open it in the editor. The file already has code for the constructor and the getCount method. You need to modify their implementations as well as implement the maxCount method which currently is not defined. All of them should make use of the counters field that is currently not being used. Instead of reading through the file to count the number of occurrences of a word every time someone calls getCount, you must change the class so that the constructor reads through the file once and popultes the counter field with the counts of all the words in the file so that later calls to getCount can simply look up the count rather than read through the file again.

Objectives

- The purpose of this assignment is to get some familiarity with the symbol table abstract data type by using one to build a class that can be used to find the frequency counts of words appearing in text files. The class is called WordFrequencyAnalyzer and this is the primary class you will be developing.

- Note that the WordReader class returns words as character arrays and that the getCount method takes a character array representing a word as an argument. However, you will run into problems if you try using a character array as a key for the symbol table. You should use Strings as keys and you can easily create a String from a character array using the String constructor.

Files

- **HW1Test** – This is a JUnit test for your solution. Each test method has a number in its name. The sum of all the numbers in all the tests that pass will give you your base score on the assignment. Note that this unit test is used for grading. They may not be the best tests to use for debugging. You will probably want to write your own tests when debugging.

- **SequentialSearchST** – This is the symbol table class you will be using to store the frequencies of all the words. You may not change this file in any way. It is provided solely to allow you to set breakpoints in it and trace through it in the debugger to help you understand how the symbol table works.

- **WordFrequencyAnalyzer** - This is the primary class you will be implementing. You need to implement the class by modifying the definitions for the constructor and methods. Note that the purpose of the field called **counters** is to keep track of (store) how many times each word appears in the given text file. Your code must use the counters field in this way.

- **WordFrequencyAnalyzerClient** – This class uses the WordFrequencyAnalyzer class you are implementing to find the frequency of a few specific words in a couple of text files as well as to determine the number of times the most frequently occurring word appears in each file. You are not asked to modify or use this class in any way. It is provided solely to demonstrate how the WordFrequencyAnalyzer class you are implementing could be used.

Restrictions

For pedagogical reasons, there are a number of restrictions you must adhere to in your solution

- The WordFrequencyAnalyzer.java file contains a single field of type SequentialSearchST<String, Integer> called **counters** that you will use to store the counts of all the words appearing in the text file. You are not allowed to change this declaration in any way nor are you allowed to add any other fields to the WordFrequencyAnalyzer class.

- The only way you are allowed to access the text file being analyzed is through a single WordReader object that you create once and loop through once inside the WordFrequencyAnalyzer constructor. This means you must count the frequencies of all the words in a single pass through the file done at the moment that a WordFrequencyAnalyzer object is created and you are not allowed to access the text files in any way when getCount or maxCount are called. The counts of all the words should have already been calculated and stored in the counters field when the WordFrequencyAnalyzer object was constructed. Note that the current implementation of getCount uses the WordReader class to read through the file. You must remove this code from getCount, but you can look at the code currently in getCount to help you write the new code for the constructor as now **it** will read through the file using the WordReader class.

WordFrequencyAnalyzer API

- **WordFrequencyAnalyzer(String filename)** - The constructor takes the name of a file and then processes the text file, keeping track of all the words appearing in the file and their counts. To aid you, I have provided you with a WordReader class that given a file name provides you with an iterator over the words in the file. You can see an example of how this class works in the code for getCount() that was given to you. (You will need to move that loop into the constructor.)

- **int getCount()** - The getCount method is only invoked after the file has been processed (after the constructor has finished executing). If the constructor correctly populated the symbol table, you can just get the counts directly out of the symbol table. You are not allowed to read from the file when getCount is called. (You must remove the loop currently is in this method and replace it with a look up in the symbol table.)

- **int maxCount()** - The maxCount method will require slightly more work. You will need to loop over the contents of the symbol table to determine the count for the most frequently appearing word. You are not allowed to read from the file when maxCount is called.

Submission

Your submission for this part of HW1 consists of two files. You must submit both files to get full credit.

- Submit your **WordFrequencyAnalyzer.java** file. This is the only source file you are allowed to submit, so don't add any other files nor modify any other files. On my windows machine, I am able to drag file from the Eclipse window into the submission folder on D2L. Make sure you do not have any print statements in your submission! Do not submit any other .java files.

- Submit a screenshot of what happens when you run HW1Test.java. The screenshot must clearly show the results of all the tests that ran.

Grading

The name of each test method in HW1Test.java ends with a number. That number is the number of points that test is worth. These numbers add up to 40 points. The other 60 points come from the debugger exercise that is part of HW1. Your overall grade on HW1 is the sum of your points from these two parts.

Note that the programming part of HW1 requires that you submit a screenshot of what happens when you run HW1Test.java. Failure to submit that screenshot will result in a deduction of 10 points.