CS 152 Computer Programming Fundamentals
Project 9: Maze Exploration[*]

Brooke Chenoweth

Spring 2025

This exercise will allow you to explore a recursively defined data structure and get some practice with the debugger.

1 Problem Description

You have been trapped in a labyrinth, and your only hope to escape is to cast the magic spell that will free you from its walls. To do so, you will need to explore the labyrinth to find three magical items:

- The *Spellbook*, which contains the spell you need to cast to escape.
- The *Potion*, containing the arcane compounds that power the spell.
- The *Wand*, which concentrates your focus to make the spell work.

Once you have all three items, you can cast the spell to escape to safety.

This is, of course, no ordinary maze. It's a data structure built from a collection of objects of type `MazeCell`, where each `MazeCell` may contain an item (a `String`, empty if nothing is there) and references to other cells reachable in the four cardinal directions (or null if we can't go that way).

```java
public class MazeCell {
  public String whatsHere; // Item present, if any
  public MazeCell north; // cell to the north, or null if can't go north
  public MazeCell south;
  public MazeCell east;
  public MazeCell west;

}
```

[*]This lab is based on the project at `http://nifty.stanford.edu/2021/schwarz-linked-list-labyrinth/`

For example, figure 1 is a 4x4 maze. The cell where you begin would have its north, south, east, and west references each refering to `MazeCell` objects located one step in each of those directions from you. On the other hand, the `MazeCell` with the book would have its north, south, and east fields set to null, and only its west variable would refer to somewhere.

Each `MazeCell` has a variable named `whatsHere` that indicates what item, if any, is at that position. Empty cells will have whatsHere set to the empty string. The cells containing the Spellbook, Potion, or Wand will have those fields set to corresponding string values.

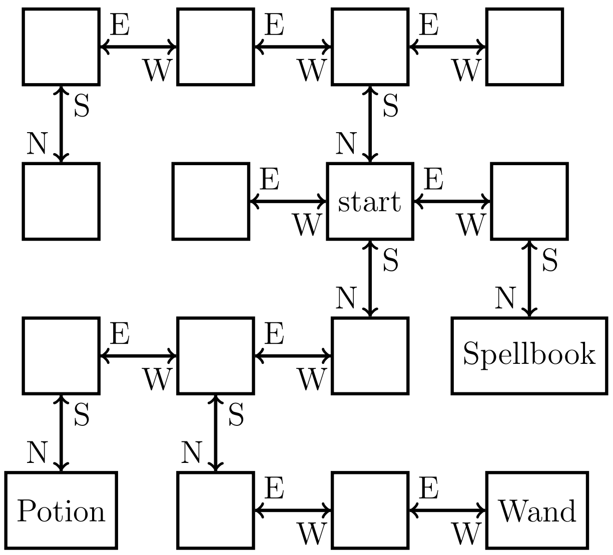

Figure 1: A 4x4 maze

If you were to find yourself in this labyrinth, you could walk around a bit to find the items you need to cast the escape spell. There are many paths you can take; here's three of them:

- ESNWSWWSNESEE
- ESNWSWSEEWWNWS
- SWWSNESEEWWNENES

Each path is represented as a sequence of letters (N for north, W for west, etc.) that, when followed from left to right, trace out the directions you'd follow. For example, the first sequence represents going east, then south (getting the Spellbook), then north, then west, etc. Trace though those paths and make sure you see how they pick up all three items.

2 What you have to do

You will need to find a path out of two personalized mazes specifically constructed for you. The starter code we've provided will use your name to build you a personalized labyrinth. By "personalized," we mean "no one else in the course is going to have the exact same labyrinth as you." Your job is to figure out a path through that labyrinth that picks up all the three items, allowing you to escape.

Open the file `MazeMain.java` and you'll see three constants. The first one, `YOUR_NAME` is a spot for your name. Right now, it's marked with a TODO message. Edit this constant so that it contains your name. *You will not receive credit for this assignment if you do not edit this string!*

2.1 Escape from your personal labyrinth

This first half of main generates a personalized labyrinth based on the YOUR_NAME constant and returns a reference to one of the cells in that maze. It then checks whether the constant PATH_OUT_OF_MAZE is a sequence that will let you escape from the maze. Right now, PATH_OUT_OF_MAZE is a TODO message, so it's not going to let you escape from the labyrinth. You'll need to edit this string with the escape sequence once you find it.

To come up with a path out of the labyrinth, use the debugger! Set a breakpoint at the indicated line in main. Run the program in your IDE with the debugger turned on. When you do, you should see the local variables window pop up, along with the contents of startLocation, which is the MazeCell where we've dropped you into the labyrinth. Clicking the arrow next to the variable in the debugger window will let you read the contents of the whatsHere field of startLocation (it'll be an empty string), as well as the four directions leading out of the cell.

Depending on your maze, you may find yourself in a position where you can move in all four cardinal directions, or you may find that you can only move in some of them. The fields in directions you can't go are all equal to null. The fields that indicate directions you can go will all have arrows near them. Clicking one of these arrows will show you the MazeCells reachable by moving in the indicated directions. You can navigate the maze further by choosing one of those expansion arrows, or you could back up to the starting maze cell and explore in other directions. It's really up to you!

Draw a lot of pictures. Grab a sheet of paper and map out the maze you're in. There's no guarantee where you begin in the maze – you could be in the upper-left corner, dead center, etc. The items are scattered randomly, and you'll need to seek them out. Once you've mapped out the maze, construct an escape sequence and stash it in the constant PATH_OUT_OF_MAZE, then see if you pass the first test. If so, fantastic! You've escaped! If not, you have lots of options. You could step through the isPathToFreedom method to see if one of the letters you entered isn't what you intended and accidentally tries to move in an illegal direction. Or perhaps the issue is that you misdrew your map and you've ended up somewhere without all the items. You could alternatively set the breakpoint at the test case again and walk through things a second time, seeing whether the picture of the maze you drew was incorrect.

To summarize, here's what you need to do:

1. Edit the constant YOUR_NAME at the top of MazeMain.java with a string containing your name. *Don't skip this step!* If you forget to do this, you'll be solving the wrong maze!

2. Set a breakpoint at the first indicated line in MazeMain.java and run the program in the debugger.

3. Map out the maze on a sheet of paper and find where all the items are. Once you're done, stop the running program.

4. Find a path that picks up all three items and edit the constant PATH_OUT_OF_MAZE at the top of MazeMain.java with that path. Run the test a second time with the debugger turned off to confirm you've escaped.

2.2 Escape from your personal twisty labyrinth

Now, let's make things a bit more interesting. In the previous section, you escaped from a labyrinth that nicely obeyed the laws of geometry. The locations you visited formed a nice grid, any time you went north you could then go back south, etc.

In this section, we're going to relax these constraints, and you'll need to find your way out of trickier mazes that look like the one shown in figure 2.

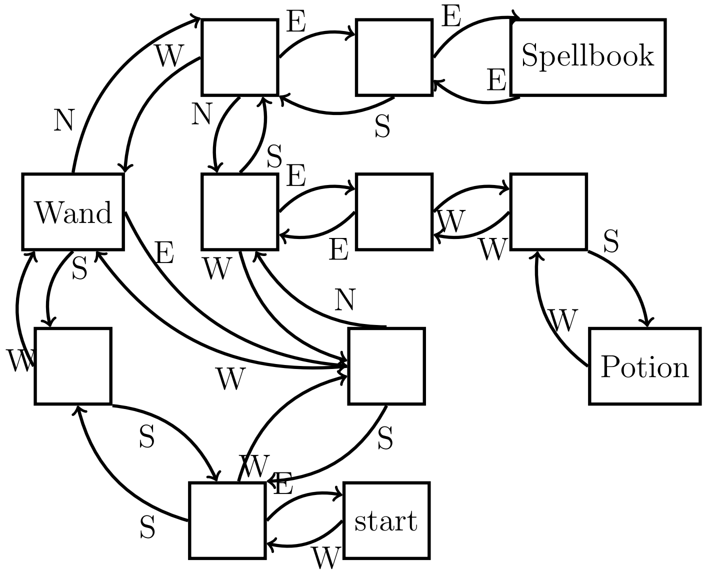

Figure 2: A twisty maze

This maze is stranger than the previous one you explored. For example, you'll notice that these `MazeCells` are no longer in a nice rectangular grid where directions of motion correspond to the natural cardinal directions. There's a `MazeCell` here where moving east and then east again will take you back where you started. In one spot, if you move west, you have to move south to return to where you used to be. In that sense, the names "north," "south," "east," and "west" here no longer have any nice geometric meaning; they're just the names of four possible exits from one `MazeCell` into another.

The one guarantee you do have is that if you move from one `MazeCell` to a second, there will always be a direct link from the second cell back to the first. It just might be along a direction of travel that has no relation to any of the directions you've taken so far.

The second half of `main` contains code that generates a twisty labyrinth personalized with the `YOUR_NAME` constant. As before, you'll need to find a sequence of steps that will let you collect the three items you need to escape.

In many regards, the way to complete this section is similar to the way to complete the previous one. Set a breakpoint in the indicated spot and use the debugger to explore the maze. Unlike the previous section, though, in this case you can't rely on your intuition for what the geometry of the maze will look like. For example, suppose your starting location allows you to go north. You might find yourself in a cell where you can then move either east or west. One of those directions will take you back where you started, but how would you know which one?

This is where memory addresses come in. Internally, each object in Java has a memory address associated with it. You can think of memory addresses as sort of being like an "ID number" for an object – each object has a unique address, with no two objects having the same address. When you pull up the debugger view of a maze cell, you should see the `MazeCell` address as part of the string representation.[1]

[1]This isn't exactly true. The number used in the default `toString` method is actually the object's

For example, suppose that you're in a maze and your starting location is `MazeCell@704` (the actual number you see will vary based on your system), and you can move to the north (which ways you can go are personalized to you based on your name, so you may have some other direction to move). If you expand out the variable for the south field, you'll find yourself at some other `MazeCell`. One of the links out of that cell takes you back where you've started, and it'll be labeled `MazeCell@704`. Moving in that direction might not be productive – it just takes you back where you came from – so you'd probably want to explore other directions to search the maze.

It's going to be hard to escape from your maze unless you **draw lots of pictures** to map out your surroundings. To trace out the maze that you'll be exploring, we recommend diagramming it on a sheet of paper as follows. For each `MazeCell`, draw a circle labeled with the address. As you explore, add arrows between those circles labeled with which direction those arrows correspond to. What you have should look like the picture in figure 2, except that each circle will be annotated with an address. It'll take some time and patience, but with not too much effort you should be able to scout out the full maze. Then, as before, find an escape sequence from the maze!

To recap, here's what you need to do:

1. Set a breakpoint at the indicated line in `main` and run the program in the debugger.

2. Map out the twisty maze on a sheet of paper and find where all the items are and how the cells link to each other. Once you're done, stop the running program.

3. Find an escape sequence, and edit the constant `PATH_OUT_OF_TWISTY_MAZE` at the top of `MazeMain.java` with that constant. Run the tests again – this time without the breakpoint – and see if you've managed to escape!

Some notes on this problem:

- The memory addresses of objects are not guaranteed to be consistent across runs of the program. This means that if you map out your maze, stop the running program, and then start the program back up again, you are not guaranteed that the addresses of the `MazeCell`s in the maze will be the same. The shape of the maze is guaranteed to be the same, though. If you do close your program and then need to explore the maze again, you may need to relabel your circles as you go, but you won't be drawing a different set of circles or changing where the arrows link.

- You are guaranteed that if you follow a link from one `MazeCell` to another, there will always be a link from that second `MazeCell` back to the first, though the particular directions of those links might be completely arbitrary. That is, you'll never get "trapped" somewhere where you can move one direction but not back where you started.

hashcode, not the actual memory address itself, but it's related and should still serve the same purpose of providing a way of distinguishing between different maze cells. If we were making mazes with thousands of cells, this assumption might break, but if I were doing that, I certainly wouldn't want to explore the whole thing manually anyway.

- Attention to detail is key here – different `MazeCell` objects will always have different addresses, but those addresses might be really similar to one another. Make sure that as you're drawing out your diagram of the maze, you don't include duplicate copies of the same `MazeCell`.
- The maze you're exploring might contain loops or cases where there are multiple distinct paths between different locations. Keep this in mind as you're exploring or you might find yourself going in circles!
- Remember that you don't necessarily need to map out the whole maze. You only need to explore enough of it to find the three items and form a path that picks all of them up.

At this point, you have a solid command of how to use the debugger to analyze linked structures. You know how to recognize a null reference, how to manually follow links between objects, and how to reconstruct the full shape of the linked structure even when there's bizarre and unpredictable cross-links between them. We hope you find these skills useful in future courses as you write code that works on linked lists and other linked structures!

3 Turning in your assignment

Submit your **MazeMain.java** file on Canvas. Do not attach `.class` files or any other files.

4 Grading Rubric (total of 25 points)

5 points The `YOUR_NAME` variable contains your name. It can be your full name, a nickname, your username, etc., but you have to change it to a different string to get a personalized maze. *You will not receive any points for the rest of the assignment if you do not change the value of this variable!*

10 points The `PATH_OUT_OF_MAZE` variable contains a valid path out of the normal maze.

10 points The `PATH_OUT_OF_TWISTY_MAZE` variable contains a valid path out of the twisty maze.