CSC115 Assignment 4: Backtracking through a maze

Objectives

Upon completion of this assignment, you need to be able to:

- Implement a Double-ended Queue ADT using a doubly-linked list.
- Implement a backtracking maze-solver *using backtracking*.
- Make use of Java's command line argument list.
- Continue accumulating good programming skills.

References:

- CSC115 Java coding conventions.
- Textbook Chapter 6 (Recursion), reference to 6.1: Backtracking
- Textbook chapter 8 (Queue), reference to Deque.

Introduction

A *maze* may be represented as a two-dimensional array of characters, each array item occupying a *cell*. We will let the `'*'` character represent a wall and the space `' '` character represent an open spot in the maze. One open cell in the first row or the first column will represent the entry point of the maze and another open cell in the last row or last column will represent the exit point. Finding a path through the maze means finding a path of connected cells that contain space characters, cells being connected only horizontally or vertically.

The path itself is a list of (r, c) pairs where r is the row number from $0 \ldots$numRows-1 and c is the column number from $0 \ldots$ numCols-1. The path list includes both the starting cell and the finishing cell.

A *maze file* is a plain text file containing all the details of the maze itself, minus the actual path. The following is an example of what a maze file would contain:

```
9 11
0 1
8 9
* *********
*     *   *
*** * *** *
*   *   * *
******* * *
*     *   *
* ******* *
*         *
********* *
```

The first three lines indicate that the maze described in the file is 9 rows by 11 columns, the starting cell is $(0, 1)$ and the finishing cell is $(8, 9)$. The rest of the lines represent the placements of the wall ("*") squares and the empty (" ") cells.

Quick Start

(1) Create a fresh directory to contain this assignment: `CSC115/assn4` is a recommended name. Download this pdf file and the `.java` files to this directory.

(2) Download and extract the `docs.zip`, `javafiles.zip`, and `testMazes.zip` in the same directory. A `docs` directory will be created to store the specifications for the `public` classes. All the document links in this document are accessible if they are stored in the `docs` directory local to this pdf file.

(3) The following files are complete:

- `Cell.java`
- `CellNode.java`
- `DequeEmptyException.java`
- `RunSolver.java`

(4) You will create and complete the following two classes:

- `CellDeque.java`
- `Maze.java`

Detailed Instructions

The following java classes are completed already:

- `Cell` defines a cell in a 2-dimensional grid or matrix by its row and column numbers. We follow the common programming convention that `row` $\in \{0, \ldots,$ `numRows`$-1\}$ and `col` $\in \{0, \ldots,$ `numCols`$-1\}$.
- `CellNode` is a Node for a doubly-linked list that holds `Cell` items.
- `DequeEmptyException` for throwing when a call to remove an item from an empty Deque object. This is a standard home-made exception, that should be familiar.
- `RunSolver` is an external tester that will be described in detail after the `Deque` and `Maze` sections.

You must create and implement the following two classes.

CellDeque

As part of the process of finding a successful path through the maze, we require a list to contain Cell objects: we add a cell when it is potentially part of the solution or remove it during the backtracking if it leads to a dead-end. We could use a general List ADT, but we only need to insert and remove from the beginning or the end of the list as we wander through the maze. Adding a cell to the list is easy, but when we backtrack, that cell needs to be removed, a process that a regular Queue will not allow. A Stack ADT allows us to remove the last try, but when we are finished, the Stack ADT will serve the solution cells in reverse order.

Hence, our choice is the Double-Ended Queue, or Deque (pronounced "deck"), which is described briefly in the textbook's chapter on the Queue ADT. It behaves as if it were a combination of both a Stack and Queue. You are to implement this class, using the doubly-linked `CellNode`.

See the CellDeque specifications for the required public methods for this class Be sure to do the necessary internal testing. Add whatever private or public methods help make the program easy to test and debug. A method that prints out the contents is most helpful.

Maze

You can use whatever data type you wish inside the Maze class to represent the maze object. We recommend a 2-dimensional array of characters, with specific characters representing a free cell, or a blocked cell, or a cell that has already been *visited*. A *visited* cell lets you know which cells are currently part of the part of the solution, much like leaving a stone in a maze to let you know that you've already tried that direction.

Note that the constructor takes, as input, an array of Strings to represent the maze. Using what you know about String objects, you can extract whatever you need from this to store the maze object as the data type of your choice.

Although it is worth thinking about loops or multiple solutions during the algorithm design, the mazes that will be used to test your code will have a single solution or no solution. You may create whatever data fields and extra methods that you need to make this work. However there are specific methods you must implement: See the Maze specifications for the required public methods for this class.

In Java, a public method should not use recursion; we leave that to a private method. The public method simply initiates the recursion by calling the recursive method with its initial values. In this assignment, `solve` must call the following `private` recursive method:

- `private boolean findPath(Cell src, Cell dest)`

This method answers the question: "Is there a path from a given source cell to the destination cell?" The following recursive definition of a path may help: A path from point A to point B exists

if A is equivalent to B, or

if C is adjacent to A and there is a shorter path from C to B.

In the implementation of this method, the base cases must be clearly identified and dealt with before the recursive statements. If no path exists in any direction between the source and destination, then backtracking is necessary. Backtracking may include removing the current cell from the solution list and/or flagging the current cell as not part of the path and then returning `false`.

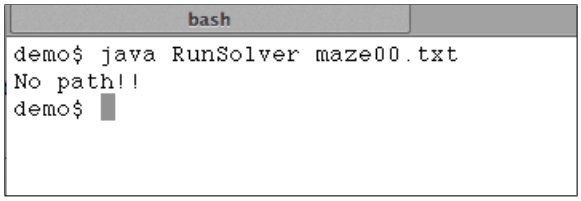
Since the internal testing involves creating test mazes (lots of typing), we supply a set of test mazes in files. *So for the* `Maze` *class, you are not required to create internal test cases.* An external tester `RunSolver` is supplied, with a single `main` method that extracts the information from a maze text file and calls the `Maze` constructor and `solve` methods.

RunSolver

This class was written primarily to extract information from tester files to check the correctness of the Maze class. You are expected to understand this class well enough to have written it yourself. You may add some `println` statements to help you, but do not change the statements that interact with the Maze class.

Examples

The following screen shots demonstrate the expected results, using the mazes in files `maze00.txt` and `maze01.txt`




```
bash
demo$ java RunSolver maze01.txt
Solution found
(0,1) (1,1) (1,2) (1,3) (1,4) (1,5) (2,5) (3,5) (3,6) (3,7) (4,7) (5,7) (5,8) (5,9) (6,9) (7,9) (8,9)
demo$
```

In the example below, we added a statement: `System.out.println(this);` near the end of the `solver` method in the `Maze` class to test whether the maze looked the way it should. You can use whatever structure you find meaningful; in our case, we let `'P'` represent the current path.

```
bash
demo$ java RunSolver maze01.txt
*P*********
*PPPPP*   *
*** *P*** *
*   *PPP* *
*******P* *
*     *PPP*
* *******P*
*        P*
*********P*

Solution found
(0,1) (1,1) (1,2) (1,3) (1,4) (1,5) (2,5) (3,5) (3,6) (3,7) (4,7) (5,7) (5,8) (5,9) (6,9) (7,9) (8,9)
demo$
```

Submission

Submit the following completed file to the Assignment folder on conneX.

- `CellDeque.java`
- `Maze.java`

Please make sure you have submitted the required file(s) and conneX has sent you a confirmation email. Do not send [`.class`] (the byte code) files. Also, make sure you *submit* your assignment, not just save a draft. Draft copies are not made available to the instructors, so they are not collected with the other submissions. We *can* find your draft submission, but only if we know that it's there.

A note about academic integrity

It is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

Grading

Marks are allocated for the following:

- Proper programming style is demonstrated, as per the coding conventions on CSC115 conneX Resources.
- The `Deque` class must be implemented using a doubly-linked list, with no data structures from the `java.util` package.
- The `Maze` `findPath` method must be
  - recursive and
  - organized to clearly determine the base case(s) and the recursive statement(s).
- Use of private helper methods where applicable.
- Internal testing in the `Deque` main method.

You will receive no marks for any Java files that do not compile.