UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Assignment 1 Specifications: Recursive Maze Solving Algorithm

Release date: 26-02-2024 at 06:00
Due date: 20-03-2024 at 23:59
Total marks: 150

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this assignment, you will have gained experience with the following:

- Implementing a recursive implentation of a Singly Linked List.
- Implementing a recursive maze-solving algorithm.

4 Warning

For this assignment, you are **only allowed to use recursion**. The following rules must be adhered to for the entirety of this assignment. Failure to comply with any of these rules **will result in a mark of 0**.

- The words "for","do" and "while" may not appear anywhere in any of the files you upload.
- You are not allowed to create any extra classes, and not allowed to upload any java files not in the following list:
  - CoordinateNode.java
  - Maze.java
  - LinkedList.java
  - Main.java
- Note that you are also not allowed to add extra classes in the allowed files. If the word "class" appears twice in any file you will also receive 0.
- You are not allowed to add any global variables to the classes, or change the global variables given. The only global variables allowed are as follows:
  - CoordinateNode
    - public int *x*
    - public int *y*
    - public CoordinateNode *next*
  - LinkedList
    - public CoordinateNode *head*
  - Maze
    - private String[] *map*

5 Introduction

Recursion is an important tool for programming, as some problems have solutions that can be easily solved using recursion. Recursion occurs when a function calls itself directly or indirectly through other functions. Recursion is used in conjunction with iteration, as both techniques have their own advantages and disadvantages. Languages like Scheme or LISP does not allow iteration, and the only solution is recursion. For this assignment, iteration is not allowed, to ensure that students understand how to use recursion for tasks which they would not usually solve recursively.

Maze-solving is a problem in which you are given an input that represents a maze. This is a collection of walls and coridors, where the goal is to find a path from a starting point to an ending point. The rules usually state that you are not allowed to move outside of the maze or over walls. Thus, corridors must be followed from the starting point to the ending point. Maze-solving algorithms like Dijkstra's algorithm or A* search requires some domain knowledge, and algorithms like depth-first search and breadth-first search require additional data structures (Graphs) to be implemented. The backtracking property of recursion can be explored to solve mazes without the need of any domain knowledge or additional data structures. This implementation will be investigated for this assignment.

6 Tasks

You are tasked with creating a Singly linked list which will be used for outputting purposes in the maze solving algorithm. You will then implement a recursive maze solving algorithm that can take a file name and then print the solutions to the maze. Note that for this assignment, **everything must be completed recursively**, and failing to follow this rule will lead to a mark of 0. See Section 4 for more information.

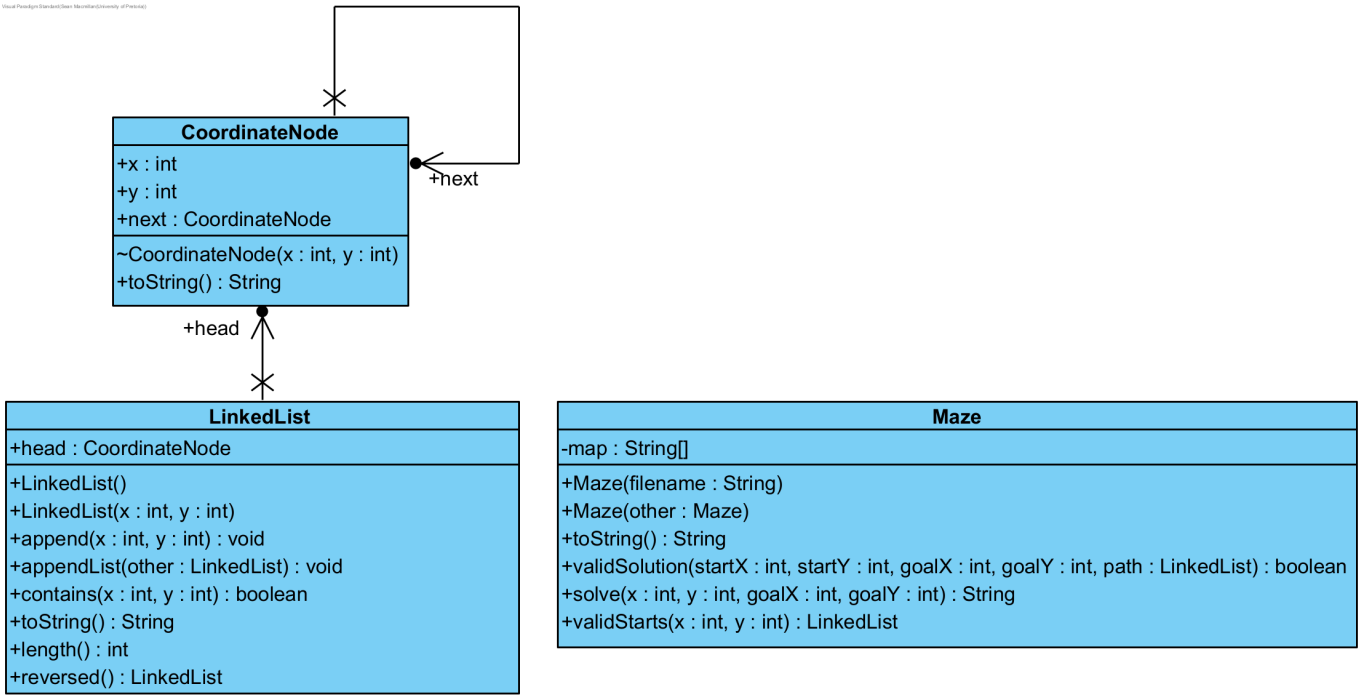

Figure 1: UML

6.1 CoordinateNode

This class is given to you. It will act as the nodes in the linked list. **Do not change anything in this class.**

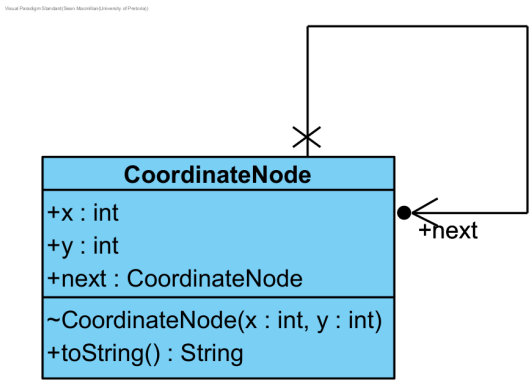

Figure 2: CoordinateNode UML

6.2 LinkedList

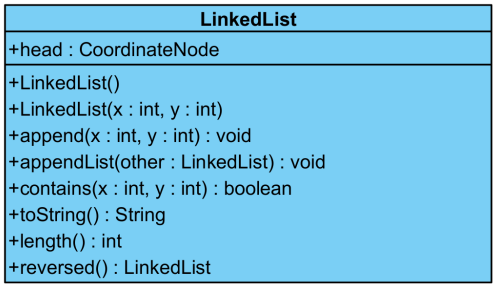

Figure 3: LinkedList UML

Note that you are allowed to add helper functions. It is highly recommend to use helper functions to implement the recursive functions, as they are easier to implement this way.

- Members
  - head : CoordinateNode
    - This is the head of the linked list.
    - If the linked list is empty, then this will be null.
- Functions
  - LinkedList()
    - This is the default constructor.
    - Initialise the list as an empty list.
  - LinkedList(x: int, y: int)
    - This constructor creates a linked list with one node.
    - The parameters passed in should be used to create a *CoordinateNode*, which will become the only node in the list.
  - append(x: int, y: int): void
    - This function should create a new *CoordinateNode* using the passed-in parameters.
    - This node should be added to the end of the list.
  - appendList(other: LinkedList): void
    - This function will be used to combine two lists.
    - All of the nodes in the passed in list should be appended at the back of the current list.
    - The passed-in list, should not be changed.
    - Example:

```
LinkedList l1 = [0,1] -> [2,3]
LinkedList l2 = [5,6] -> [7,8]
l1.appendList(l2);
l1 = [0,1] -> [2,3] -> [5,6] -> [7,8]
```

- contains(x: int, y: int): boolean
  - This function should return true if there is a node inside the list which has the same coordinate as the passed-in parameter.
  - If no node was found with matching coordinates, then return false.
- toString()
  - If the list is empty, then the following String should be returned:

```
Empty List
```

  - If the list is not empty, then the toString function of each node should be called. These strings should then be appended using *<space><-><>><space>*.
  - Example

```
[0,1] -> [2,3] -> [4,5] -> [6,7]
```

- length(): int
  - This function should return the length of the list.
  - If the list is empty, then this should return 0.
- reversed(): LinkedList
  - This function should return a copy of the linked list where the order of the nodes should be reversed.
  - Example: Original

```
[0,1] -> [2,3] -> [4,5] -> [6,7]
```

    Reversed

```
[6,7] -> [4,5] -> [2,3] -> [0,1]
```

6.3 Maze

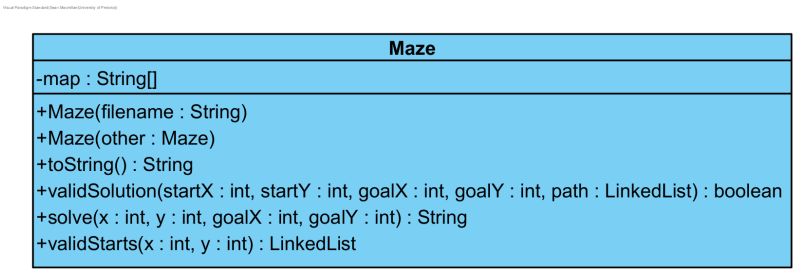

Figure 4: Maze UML

Note that the student files contains an example which can be used to ensure that all formatting is correct.

- Members
  - map: String[]
    - This is an array of strings representing the map.
    - Note that the strings do not need to be the same length, and thus non-rectangular mazes are allowed.
    - If it is an empty maze, then this will be an array of length 0.

- Functions
  - Maze(filename: String)
    - This is the maze constructor.
    - If the passed-in filename cannot be found, then create an empty maze.
    - If the file is found, read the first line and convert that into an integer. This will be the number of rows in the maze. You may assume that this will be a valid line and that the number will always be greater than 0.
    - After that, there are a number of lines in the textfile. These lines should be read and then saved in the map array. Note that the text file will always contain enough lines to fill the array. The textfile might contain too many lines, but these should be ignored and not read in.
    - You may assume that the lines will be valid and that no illegal characters will be added. The only valid characters are "-" for a corridor and "X" for a wall.
  - Maze(other: Maze).
    - This is a copy constructor. A copy of the passed-in maze should be created.
  - toString(): String
    - If the Maze is empty, then return the following string:

```
Empty Map
```

    - If the Maze is not empty, then the map should be returned as a string. There should be a new line between each element in the map array. Note that there should not be a new line on the last line.
  - validSolution(startX: int, startY: int, goalX: int, goalY: int, path LinkedList): boolean
    - This function will take in a starting and ending coordinates as well as a *path* to verify. Return true if the passed-in path is a valid solution to the current maze from the beginning at passed-in start coordinate and ending at the passed-in end coordinate. Otherwise, return false.
    - A path is considered valid if all of the following is true:
      - The head of the *path* linked list must be the start coordinate and the tail of the *path* linked list must be the end coordinate.
      - The path does not contain any walls or contain nodes that are outside of the maze.
      - The path only moves 1 step at every node. Therefore, moving more than one step in a single direction at a time is not allowed. You are also not allowed to move diagonally.
      - The path does not double back on itself. Thus, if a coordinate is visited at a point in a valid path, then it is not allowed to be visited again.

- solve(x: int, y: int, goalX: int, goalY: int): String
  - This function returns a String which gives the solution to the maze with the passed-in start and end coordinates.
  - If the maze does not have a valid solution according to the rules of validSolution(), then return the string:

```
No valid solution exists
```

  - If a valid solution exists, then print the following. Note that each line is printed on its own line and that the last line does not end on a new line.

```
Solution
<The solved Maze>
<The linked list toString on the solution>
```

  - <The solved Maze> refers to taking the current maze, and then replacing the start point with "S" and the end point with "E". Every coordinate visited in the path should be replaced with "@". Note that the original map should remain unchanged.
  - The linked list solution should be calculated using the following rules:
    - A LinkedList of CoordinateNodes should be returned starting with the start coordinate, and ending with the end coordinate.
    - The Euclidean distance between any two consequetive nodes must be 1.
    - The coordinates may not be visited more than once.
    - The search should start at the starting node. From here, the neighbours should be checked in the following order:

```
Left, Up, Down, Right
```

    - At each neighbour, the order of checking neighbours should remain the same.
    - Thus, from any point in the maze, the algorithm will first try to go as leftward as possible, then as upward as possible, then as downward as possible, then as rightward as possible.
    - If a valid path was not found, then an empty list is used as a solution.

- validStarts(x: int, y: int): LinkedList
  - The passed-in parameters represent a goal coordinate.
  - This function should return a LinkedList of all possible coordinates which have a valid path starting from itself and ending at the goal coordinate for the current maze.
  - Note that the LinkedList should be sorted by the y coordinate first and then by the x coordinate. Thus, when reading the coordinates, you should be reading from left to right, and then from top to bottom.

7 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the Main.java file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
    -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
    ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

8 Upload checklist

The following files should be in the root of your archive

- Main.java
- CoordinateNode.java
- LinkedList.java
- Maze.java
- Any textfiles needed for your Main
- **Don't include any other files**

9 Allowed libraries

- java.io.File
- java.io.FileNotFoundException
- java.util.Scanner

10 Submission

You need to submit your source files to the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 3 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**