Learning Outcomes

To gain experience with

- Working with linked data structures
- Recursive approaches to problem solving
- Putting together many concepts into one project
- Algorithm design

Introduction

Minesweeper is a solitaire computer game in which there are bombs hidden in a grid of cells, and the objective is to clear out all the non-bomb cells without hitting a bomb. If a bomb is clicked, the game is over in failure.

To help the player determine where the bombs are located, the other cells contain a number that indicates the number of bombs in the eight adjacent (both orthogonal and diagonal) cells. The number on a cell is revealed when that cell is clicked. Additionally, when a cell that has no adjacent bombs (let's call this a 0-cell) is clicked, it triggers a "region clear" which means that a region beginning with the clicked cell and expanding outward will be revealed at once. The region will include all 0-cells connected to the clicked cell, and numbered cells connected to those 0-cells. Figure 1 shows examples of 3 cleared regions. For each of these regions, one of the inner 0-cells was clicked, and the surrounding region was consequently cleared.

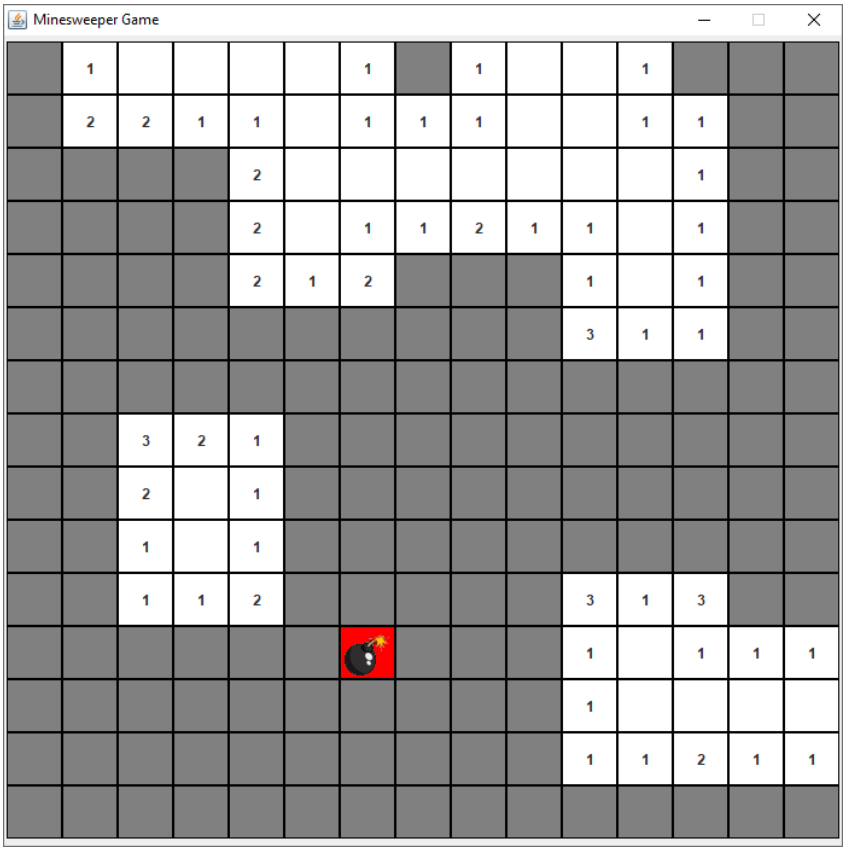

Figure 1. A screenshot of a completed (lost) game in which a bomb cell was clicked after revealing 3 regions throughout the board. Cells with a number indicate how many bombs cell are neighbouring (orthogonally or diagonally adjacent).

Provided files

The following is a list of files provided to you for this assignment. Please do not alter these files in any way.

- BombRandomizer.java – provides a method for randomly placing bombs
- GUI.java – provides the visual GUI for the program
- GUICell.java – represents each cell in the game board
- LinearNode.java – represents a node for a singly-linked list
- LinkedListException.java – a simple exception class
- TestLinkedGrid.java – provides several tests to check that LinkedGrid is working
- TestGame.java – provides several tests to check that the Game methods are working

Classes to implement

For this assignment, you must implement 2 Java classes: *LinkedGrid* and *Game*. Follow the guidelines for each one below.

In both of these classes, you can implement more private (helper) methods, if you want to, but you may **not** implement more public methods. You may **not** add instance variables other than the ones specified below. Penalties will be applied if you break these rules.

LinkedGrid.java

This class represents a 2D grid (matrix) that is created as an array of singly linked lists. This class must work for any generic type T and must be created using the descriptions explained here. The array, which must be the width of the grid, will contain the front (or top) nodes of each of the linked lists. The height of the grid is represented by the number of nodes in each of the linked lists. The example below shows a 5 by 4 grid (remember the front node of each list is contained in the array so there are 4, not 3, nodes in each list).

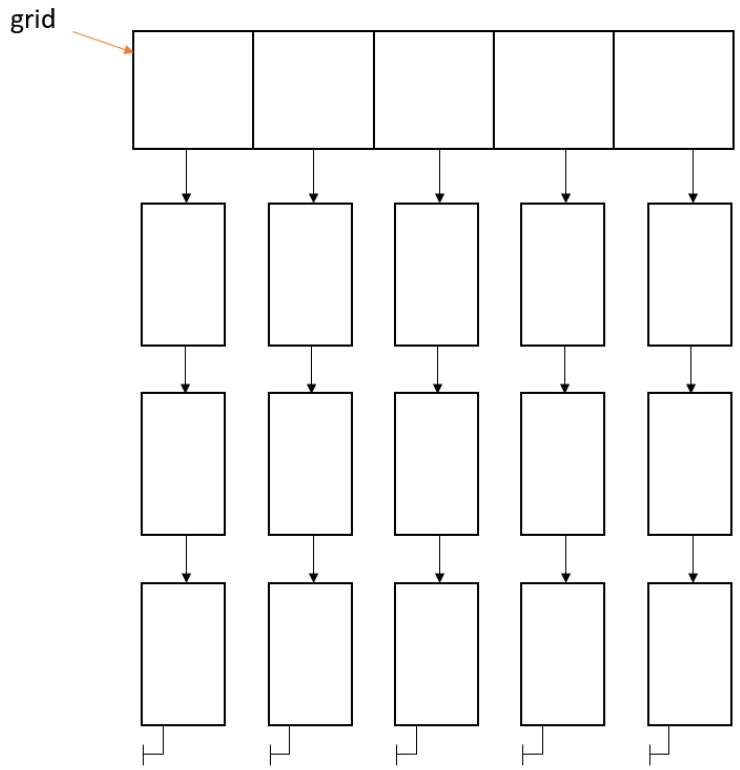

Figure 2. A diagram depicting the LinkedGrid structure as an array of linked lists.

The class must have the following *private* variables:

- width (int)
- height (int)
- grid (LinearNode<T> array)

The class must have the following *public* methods:

- LinkedGrid (constructor) – takes in 2 parameters for width and height and assigns their values into the corresponding instance variables. The grid must be initialized properly by creating the LinearNode objects and connecting them as singly linked lists. The first node of each list must be stored in the corresponding array cell in the grid parameter.
- setElement – takes in 3 parameters: int col, int row, T data. If the col or row is outside the bounds of the grid, throw a LinkedListException. Otherwise, find the cell in the grid at the given col, row pair and set the element of that node to the given value, data.
- getElement – takes in 2 parameters: int col, int row. If the col or row is outside the bounds of the grid, throw a LinkedListException. Otherwise, find the cell in the grid at the given col, row pair and return the element contained in that node.
- getWidth – returns the width
- getHeight – returns the height
- toString – builds a string that represents the entire grid of elements. Include 2 spaces between each of the elements and use a newline character at the end of a row to continue the string on the next line. Return the completed string.

Game.java

This class contains the Minesweeper-related code. It is used to initialize the game board, determine the numbers for each cell (which indicate how many bombs are in neighbouring cells), the recursive "region clearing", etc.

The class must have the following *private* (unless stated otherwise) variables:

- board (LinkedGrid<Character>)
- cells (LinkedGrid<GUICell>)
- width (int) – must be public static
- height (int) – must be public static
- isPlaying (boolean)
- gui (GUI)

The class must have the following methods:

- Game (constructor #1) – takes in int width, int height, boolean fixedRandom, int seed. Set the width and height class variables and then initialize the board grid with an '_' (underscore) character in every cell. Then initialize the cells grid of GUICell elements.

Use the BombRandomizer's placeBombs method (sending in the board and fixedRandom parameter) to randomly place bombs throughout the game board. Call the determineNumbers method (see description below) to get the number for each cell which indicates how many bomb cells are neighbouring each cell. Set isPlaying to true. Lastly, initialize the GUI object using: **`gui = new GUI(this, cells);`**

  - NOTE: the fixedRandom variable is used to determine if you want to use the same random seed each time you run it for testing purposes. If fixedRandom is true, then it uses the fixed seed (4th parameter) so the board will be the same every time you run it (assuming the size is not changed). Otherwise set fixedRandom to false if you want to run the game with a different, actually random board each time.

- Game (constructor #2) – takes in LinkedGrid<Character> board. This constructor is almost the same as the first one above. The only differences are that the board is already set so you should not follow the same board initialization and random bomb placement. Instead, simply assign the board parameter to the corresponding instance variable. The rest of the method is identical.
- getWidth – returns the width
- getHeight – returns the height
- getCells – returns the LinkedGrid cells
- determineNumbers – go through every single node in the board and calculate how many bombs are in surrounding cells, and insert that number into the corresponding node in the cells grid. Cells that contain bombs must have a number of -1 (even if they are adjacent to other bombs), but all other cells must have a number (from 0 to 8) that indicates the number of bombs around that cell.
- processClick – this method processes a cell being clicked or a simulated click and returns an int value representing how many cells are being revealed OR -1 if a bomb is revealed upon this click OR -10 if a bomb has previously been revealed (meaning isPlaying is false) regardless of where this current click might be. It takes in two integers for col and row, and gets the GUICell from that position in the grid.
  - If the given cell contains -1 (bomb) then set the background of this cell to red (use cell.setBackground(Color.red)) and reveal the cell (look at the GUICell methods to see how to do this).
  - If the cell contains a 0, begin the recursive "region clearing" from this cell and return the result that is returned from the recursive method.
  - If the cell contains any other number (1 through 8), then check if it was previously revealed. Return 0 if previously revealed. If it wasn't previously revealed, make sure to reveal it and set its background colour to white and then return 1.
- recClear [private, and name is not important] – takes in two integers for col and row and returns an int representing the number of cells being revealed from this method call. This is the recursive helper method invoked from the processClick method. Read the description of "Region clearing" and the pseudocode for recClear below.

Region clearing

One of the terms used in this assignment is "region clearing". This occurs when a 0-cell (a cell that does not have a bomb nor is adjacent to a bomb) is clicked. The cell itself is revealed but it also triggers its region to become revealed immediately. The region is defined as the contiguous (connected) 0-cells and even up to numbered cells along the perimeter of the region but no further than the numbered cells. This form of clearing can be done very elegantly with recursion.

Examine the following illustrations that demonstrate "region clearing" in a sample 5 by 5 game grid with 7 bombs. Figure 3 shows the original game board before clicking any cells, and it shows the bombs' locations just for demonstrative purposes (bombs are not shown in the actual game!) Figures 4-6 illustrate each recursive step in the algorithm from clicking the middle cell.

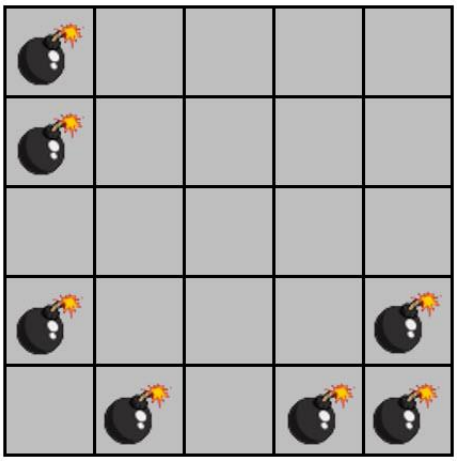

Figure 3. A sample 5 by 5 game grid with 7 bombs (visible for demonstration purposes)

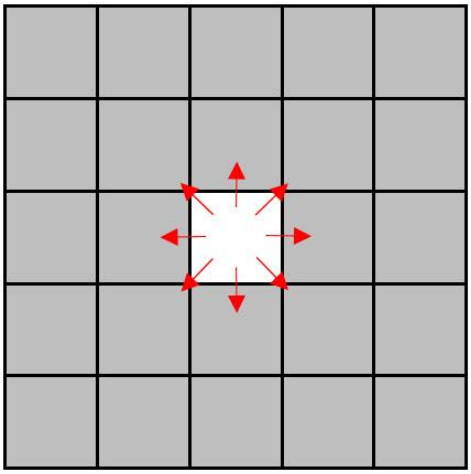

Figure 4. The sample game board after clicking the middle cell, which is a 0-cell. Red arrows indicate which cells will be revealed in the first recursive step of the region clearing.

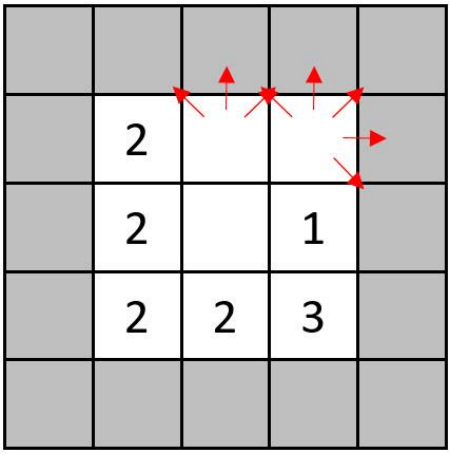

Figure 5. After one recursive step, the cells surrounding the clicked (middle) cell are revealed. The next step will reveal the cells surrounding the 0-cells at the top of the existing region. Numbered cells indicate the region's perimeter, so the clearing doesn't go beyond those cells.

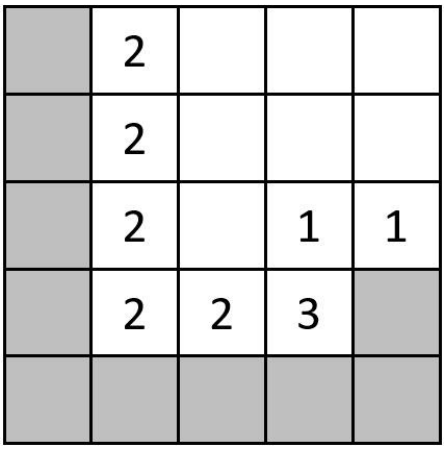

Figure 6. The clearing has completed since Figure 4. The cells just revealed are either numbered or are along the edge of the board itself, so there are no more recursive calls. This is the finished "region clearing" from clicking that middle cell (NOTE: the region would have been the exact same from clicking on **any** of the 0-cells in the top-left corner of this region).

Pseudocode

function recClear (int c, int r)

Input: column c and row r of cell being clicked

Output: int value representing how many cells were revealed

if c or r is outside bounds of game board, return 0

if cells(c, r) is already revealed, return 0

if cells(c, r) is a bomb (-1), return 0

else if cells(c, r) is a numbered cell (> 0)

reveal cells(c, r)

set cells(c, r) colour to white in gui (if gui != null)

return 1

else

reveal cells(c, r)

set cells(c, r) colour to white in gui (if gui != null)

result = 1;

result += recClear(c-1, r)

… (all other recursive calls – total of 8 recursive calls)

return result

Marking Notes

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Does the code run properly on Gradescope (even if it runs on Eclipse, it is **up to you** to ensure it works on Gradescope to get the test marks)
- Does the program produces compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

Assignment 4

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)?
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting and white-space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a penalty of 5%
- Including a "package" line at the top of a file will receive a penalty of 5%

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through similarity-detection software.

Submission (due Tuesday, December 6 at 11:55 pm)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below.
- Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. **If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.**
- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works for other dice combinations as well. We are providing you with some tests but we may use additional tests that you haven't seen before for marking.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Assignment 4

Files to submit

- LinkedGrid.java
- Game.java

Grading Criteria

Total Marks: [20]

Functional Specifications:

[3] LinkedGrid.java

[3] Game.java

[10] Passing Tests (some additional, hidden tests will be run on Gradescope)

Non-Functional Specifications:

[1] Meaningful variable names, private instance variables

[1] Code readability and indentation

[2] Code comments