CSCI-UA 9102 DATA STRUCTURES
Assignment 3

Augustin Cosse

April 30, 2021

Given date: April 18
Due date: May 10
Total: 30pts

In this third assignment, we will review linked structures, iterators and trees. This assignment will be the last assignment of the semester.

Part I (14pts) Linked structures and iterators

We consider a meshgrid ADT which is defined by a set of nodes interconnected as shown in Fig. 1 below. In short, each node has four connections labeled left, right, above and below. In this ADT, the above (resp. below, left and right) references of the nodes located on the upper (resp. lower, left and right) boundaries are assigned the value null. You should design the class so that the nodes can encode arbitrary data types.

Question I.1 (6pts)

Complete the file `LinkedMeshgrid.java` by providing implementations for

- The main constructor for the `LinkedMeshgrid` class
- The methods `leftPos`, `rightPos`, `abovePos` and `belowPos`.
- The method `set(Position<E> p, E e)` which should make it possible to modify the value of a (private) node from the knowledge of its position.

When the list is used to store numbers (float or integers) as in Questions I.2 and I.3 below, you can display it as an image by first converting it to a 2D array (do this in the `main`) and then using the following lines

```
int xLenght = arr.length;
int yLength = arr[0].length;
BufferedImage b = new BufferedImage(xLenght, yLength, 3);

for(int x = 0; x < xLenght; x++) {
    for(int y = 0; y < yLength; y++) {
        int rgb = (int)arr[x][y]<<16 | (int)arr[x][y] << 8 | (int)arr[x][y]
        b.setRGB(x, y, rgb);
    }
}
ImageIO.write(b, "Doublearray", new File("Doublearray.jpg"));
System.out.println("end");
```

Question I.2 (8pts)

We want to implement two iterators on the `LinkedMeshgrid` class.

- A first iterator on positions.
- A second iterator on the elements (which should rely on the position iterator)

In your implementation, you might want to consider the following points:

- It is up to you to decide how you want to skim through the nodes (you can for example start from node $n_{0,6}$ then scan the whole first row, then scan the second row, starting from node $n_{0,5}$,.. but any alternative path which goes through all the nodes is perfectly valid.).
- Keep in mind that when you have returned all the nodes, the method `hasNext` from the iterator should return `null`.
- An instance of the `meshgrid.java` class always contains $h * w$ nodes where h (height) and w (width) are defined as instance variables (equivalent of the size parameter). Hence it might be a good idea to define the first node of the instance as the top left node and the last one as the bottom right node. Also note that to make your life easier, you can just consider square ($w = h$) lists.

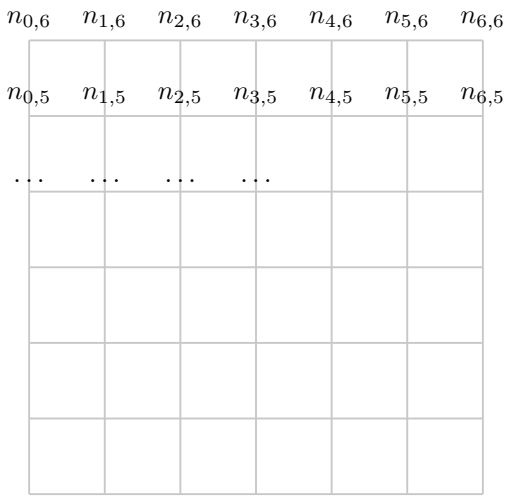

Figure 1: An instance of the class `Meshgrid.java`

Part II: Prim's algorithm (8pts)

We will now use your `mesh` class to create and store a maze. There are several approaches for generating mazes. In this question, we will rely on Prim's algorithm. The algorithm relies on the notions of rooms, walls, pillars and path.

The original maze is shown in Fig. 2 below. In this figure, the yellow points represent the pillars, the green/cyan points represent the rooms and the (dark) blue points represent the walls.

Prim's algorithm can be implemented through the following steps

- We start by marking all walls as closed.
- We then select a room (typically the room located in the top left corner) from the set of all rooms and add it to the path.

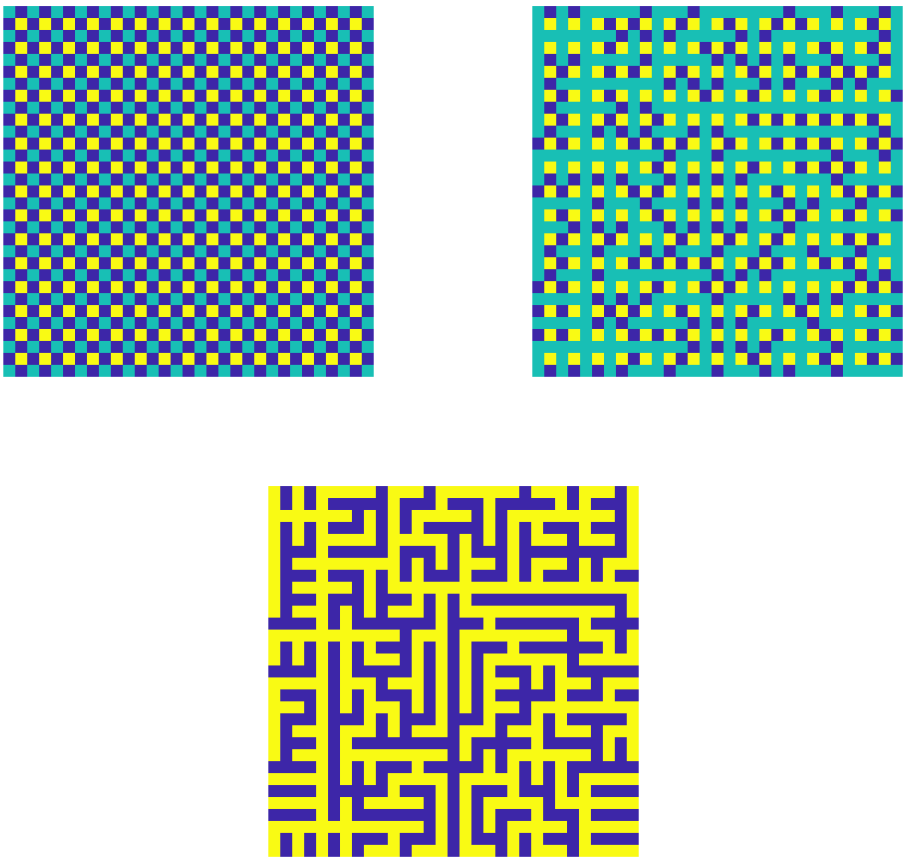

Figure 2: (Top left) Initialization of Prim's algorithm, (Top right) result (rooms, walls and pillars) returned by the algorithm and (Bottom) corresponding maze. The pillars are represented in yellow, the walls are in blue and the rooms are in green.

- We add the four walls of the room to the "wall list". This is the list that we keep processing until it's empty
- While the wall list is not empty:
  - Select a wall from the list (at random)
  - Find the rooms adjacent to the wall
  - If there are two adjacent rooms and exactly one of them is not in the path,
    - Mark the wall as "Open"
    - Add the unvisited room to the path
    - Add the walls adjacent to the unvisited room to the wall list
  - Remove the wall from the wall list (and remove any duplicate entries)

Part III: Escaping (8pts)

There are several algorithm for escaping a maze. In this exercise, we will rely on the fact that the maze can be encoded as a tree.

- Using your maze as well as the file `mySimpleBinaryTree.java` from Lab 9 (see github), create a simple tree by taking as root the entry of the maze (upper left room) and as nodes the set of all rooms, and by connecting the nodes when their corresponding rooms are immediately adjacent to each other.

- Once you have the tree, solve the maze by using a depth-first (in-order) tree traversal (you will need an iterator on the tree) and returning the path (not necessarily shortest) you get when you reach the exit (bottom right corner)