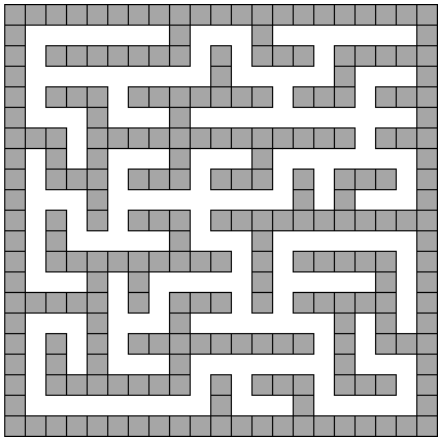
COMP2401 - Assignment #4

(Due: **Sunday, Nov 10, 2019 @ 6pm**)

In this assignment, you will gain practice dynamically allocating/freeing memory as well as working with pointers to allocated structures. You will also use a makefile to compile the code. You will also get to practice your recursion … I hope that you remember how to do that.

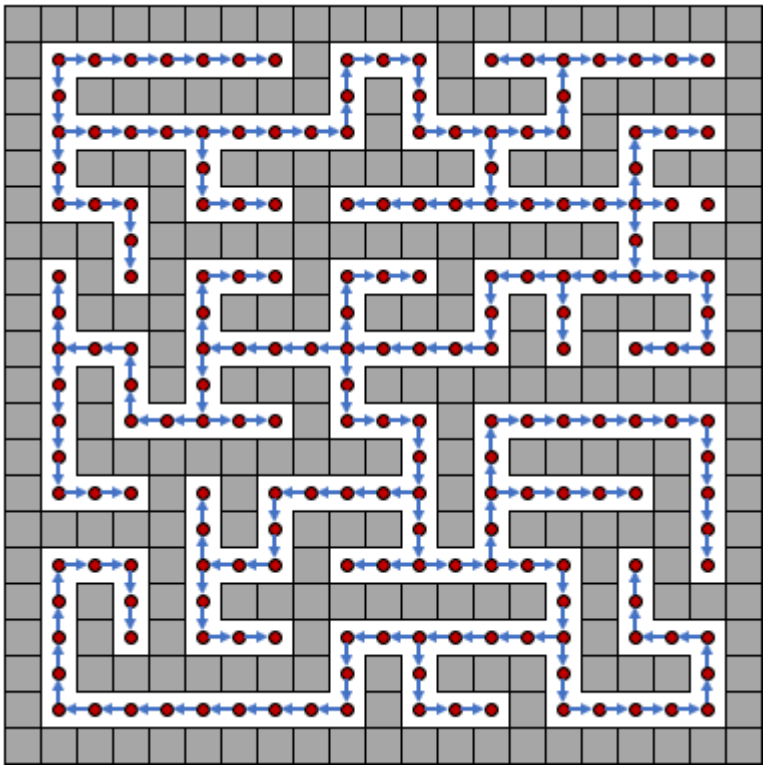
Consider a simple grid-like maze as shown here to the right →



We would like to create a directed graph that can represent the free-spaces in the maze by placing a graph node at each free space grid location and connecting them together with edges as shown in the bottom right image here. Notice that the graph that we will create will actually be a tree representing paths along the maze. We will only test our code on mazes that do not have "loops" in them.

To do this, we will make use of the following **typedef** to represent each **Node** of the graph:

```
typedef struct nd {
  int         x;
  int         y;
  struct nd *up;
  struct nd *down;
  struct nd *left;
  struct nd *right;
} Node;
```



So, each node will have an (**x**, **y**) location … which will be the column and row in the grid (where (0,0) is the top left wall). Also, each Node will keep a pointer to the (up to 4) nodes **up**, **down**, **left** and **right** of it … each of which may be **NULL** of there is no such node in the specific direction. So, for example, the top left Node in this graph has location (1, 1) and the up and left pointers are **NULL**, while the right and down pointers connect to two different nodes. It is important to note that this is a directed graph. So, for example, if node **A** has it's **right** pointer pointing to node **B**, node **B** will NOT have it's **left** pointer pointing to node **A**.

We will be creating multiple graphs … one for each of 5 mazes. A graph be represented as follows:

```
typedef struct gr {
  Node        *rootNode;
  struct gr   *nextGraph;
} Graph;
```

As can be seen, the graph is really just the root node of the tree. The remainder of the graph can be figured out by tracing the nodes that are connected to this root node. Each graph will also connect to another graph … because we will be creating a linked-list of graphs. The linked-list will actually be called a **GraphSet** and will be defined as follows:

```
typedef struct {
  Graph  *firstGraph;
  Graph  *lastGraph;
} GraphSet;
```

where **firstGraph** is the head of the list and **lastGraph** is the tail.

---

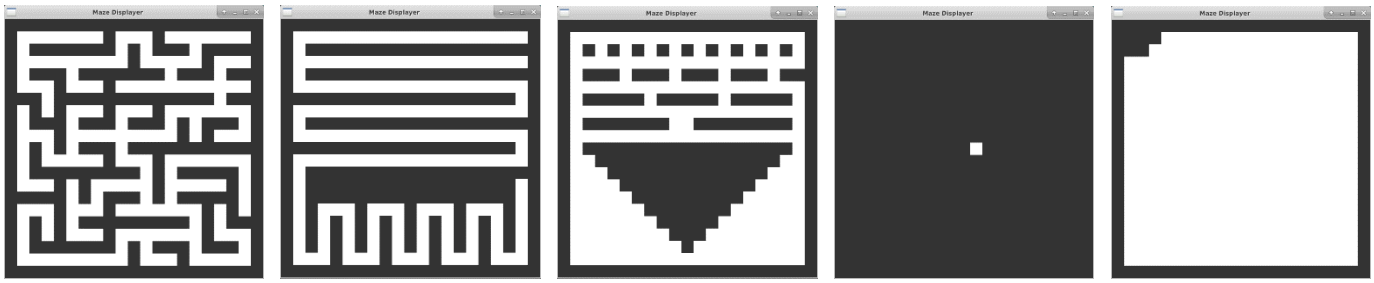
(1) To begin this assignment, you must download the following files:

mazes.c, **graphSet.h**, **mazeDisplay.h**, **mazeDisplay.c**

You may ONLY alter the **mazes.c** code … you MUST NOT alter the other three files. Follow the steps below to complete the assignment. For this assignment, you ARE NOT ALLOWED to create any arrays. You MUST NOT create arrays to store Graphs nor Nodes.

(2) Write a **makefile** that compiles and links the **mazes.c** and **mazeDisplay.c** files into an executable called **mazes** You can use a makefile from the course notes as a template. The makefile should allow **make all** and **make clean** commands to work properly. You will need to include the **`-lX11`** library (that is "minus small L" at the front … not "minus 1") in order for the code to link properly, since we will be using some windows with graphics (described later on in the course).

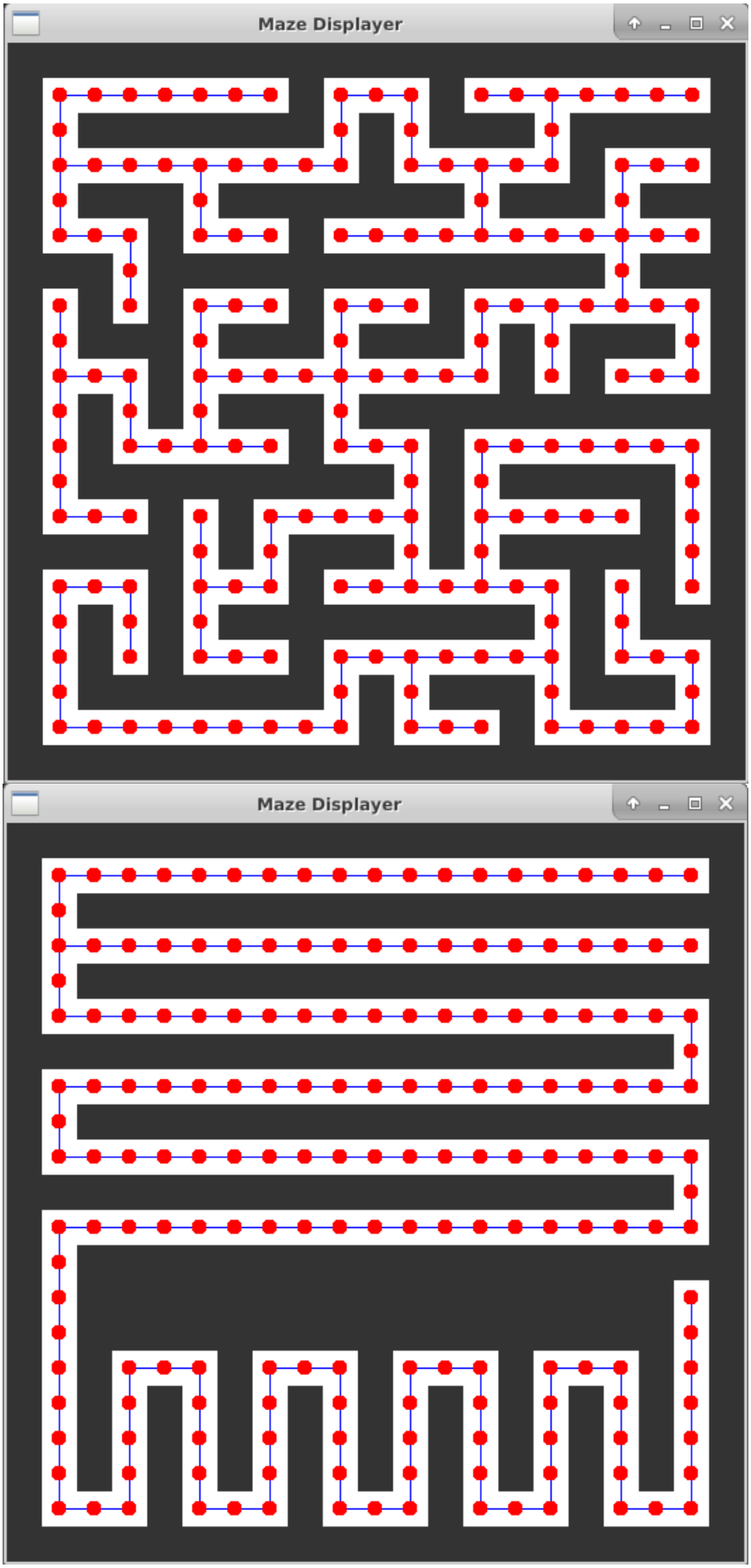
(3) Once the code compiles and produces an executable. Run it. You should see a window appear showing a maze. If you click back on the Terminal window, then press ENTER ten times … it will cycle through the 5 mazes shown below and then quit:

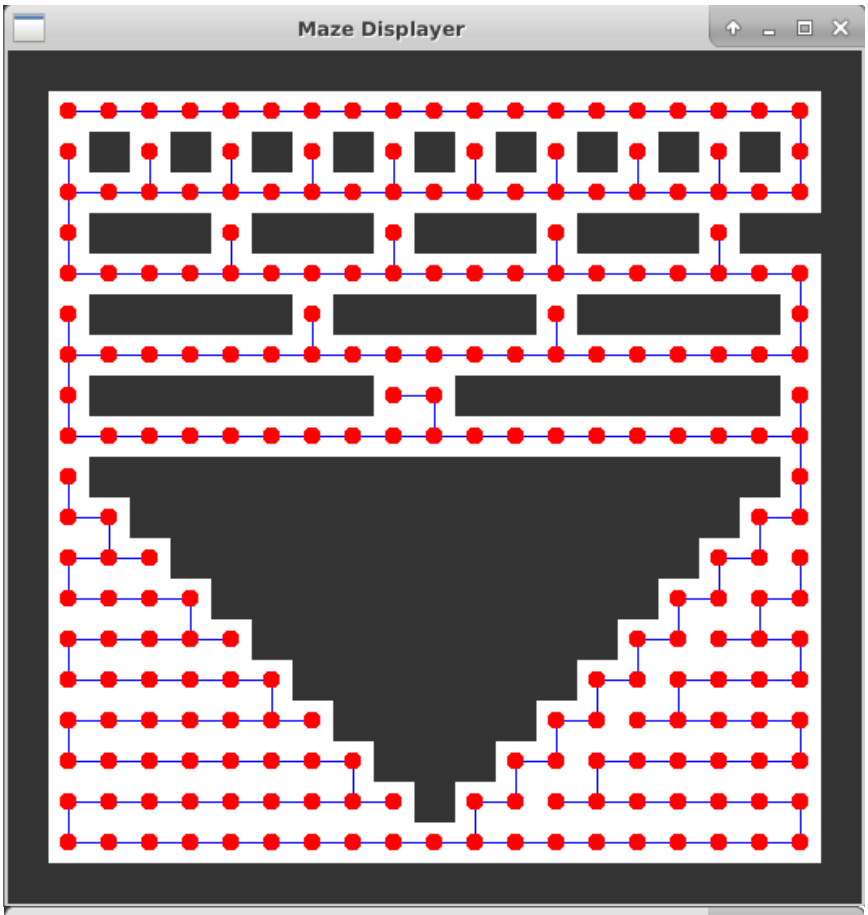


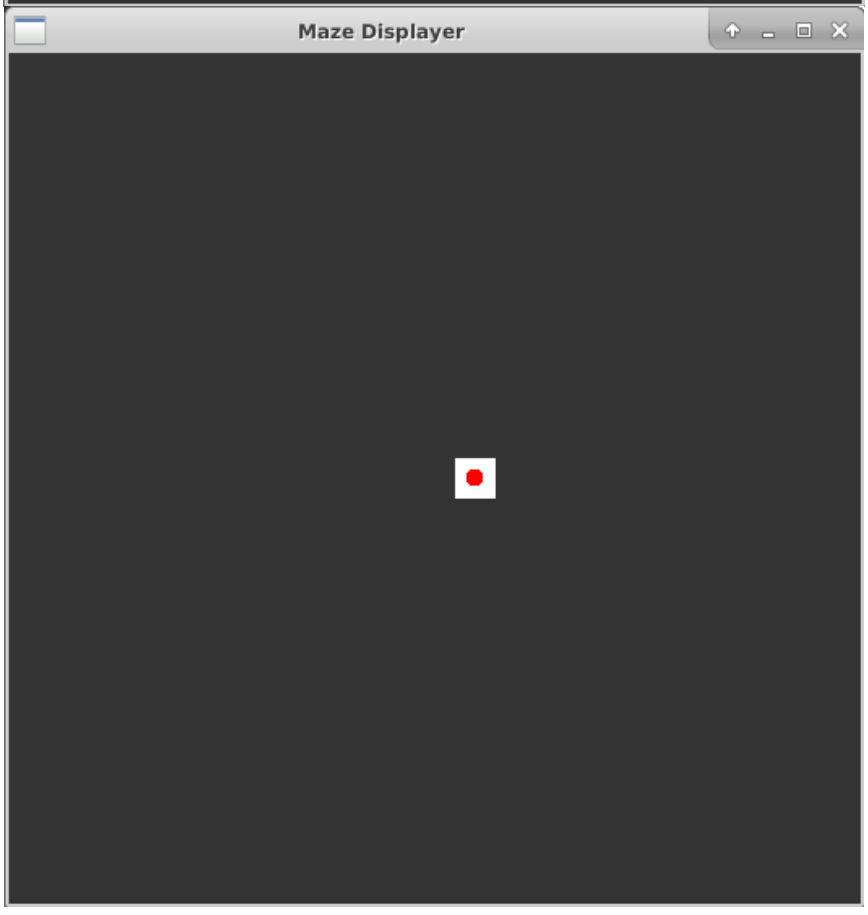
---

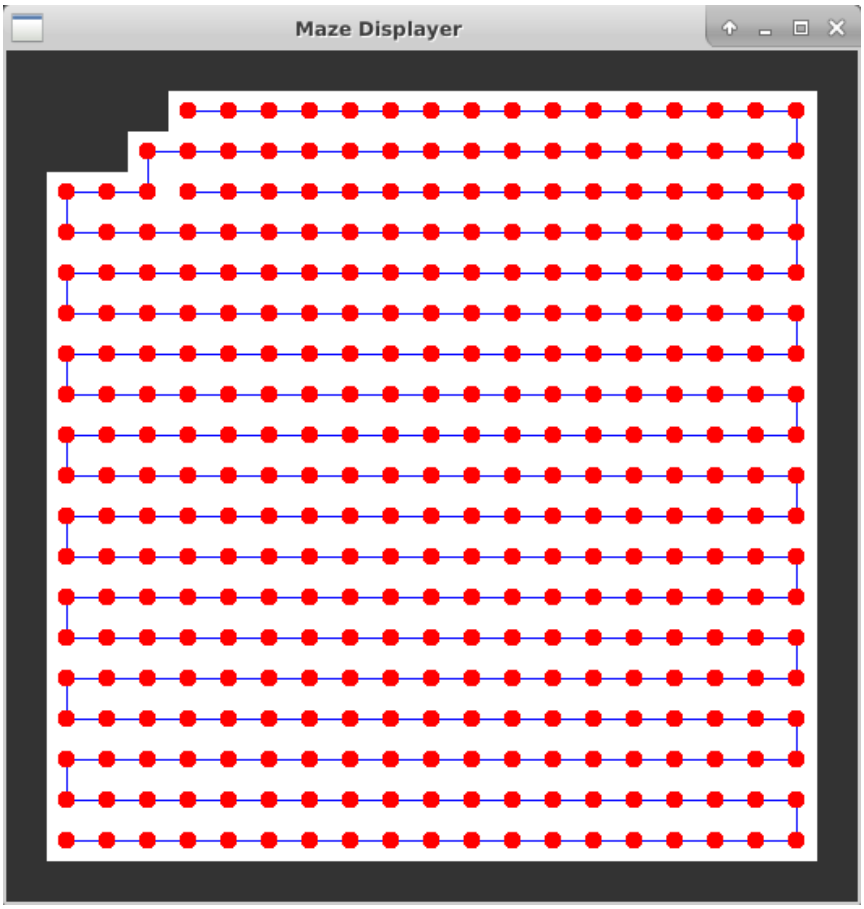
(4) Now you are ready to get to work. The skeleton of a **computeGraph()** procedure has been written for you, with some comments to give you some ideas. You must now complete this procedure so that it produces the graph for the maze given as a parameter. It must allocate space for the new graph and return it. Each node of the graph must be dynamically-allocated and you also must connect them properly by setting the appropriate Node up/down/left/right pointers. Note that the graph must not contain cycles, otherwise the display routine that has been written for you will not work properly. Once you get this working, you need to adjust the FOR loops in the main function. The first FOR loop must add the graph to the graph set. In the 2nd FOR loop, uncomment the **drawGraph()** procedure call so that you can see the graph. Also, add a line at the end of the loop to go to the next graph. Once you do this, you should be able to cycle through all the 5 mazes … showing the graphs each time. Here is how each graph should look when

completed (assuming that the root is the top/left free space and that the traversal ordering is up, down, left then right).




Maze Displayer


Maze Displayer



(5) We will now remove unnecessary nodes from the graph such that it is smaller, yet contains the same path information. To do this, you will complete the **cleanUpGraph()** procedure so that it removes appropriate nodes from the graph.

To do the cleanup, it is good to make the **cleanUpGraph()** procedure recursive, taking the **rootNode** to start, as well as the **previousNode** … which will be **NULL** upon start). As before, you will want to traverse through the nodes. You will not need to use the maze now, because you are just iterating through the graph that you made. You should NOT mark anything as visited. DO NOT alter the **Node** nor **Graph** typedefs. You need to identify nodes that need to be removed. A node **n** should be removed if and only if it satisfies ALL three of these conditions:

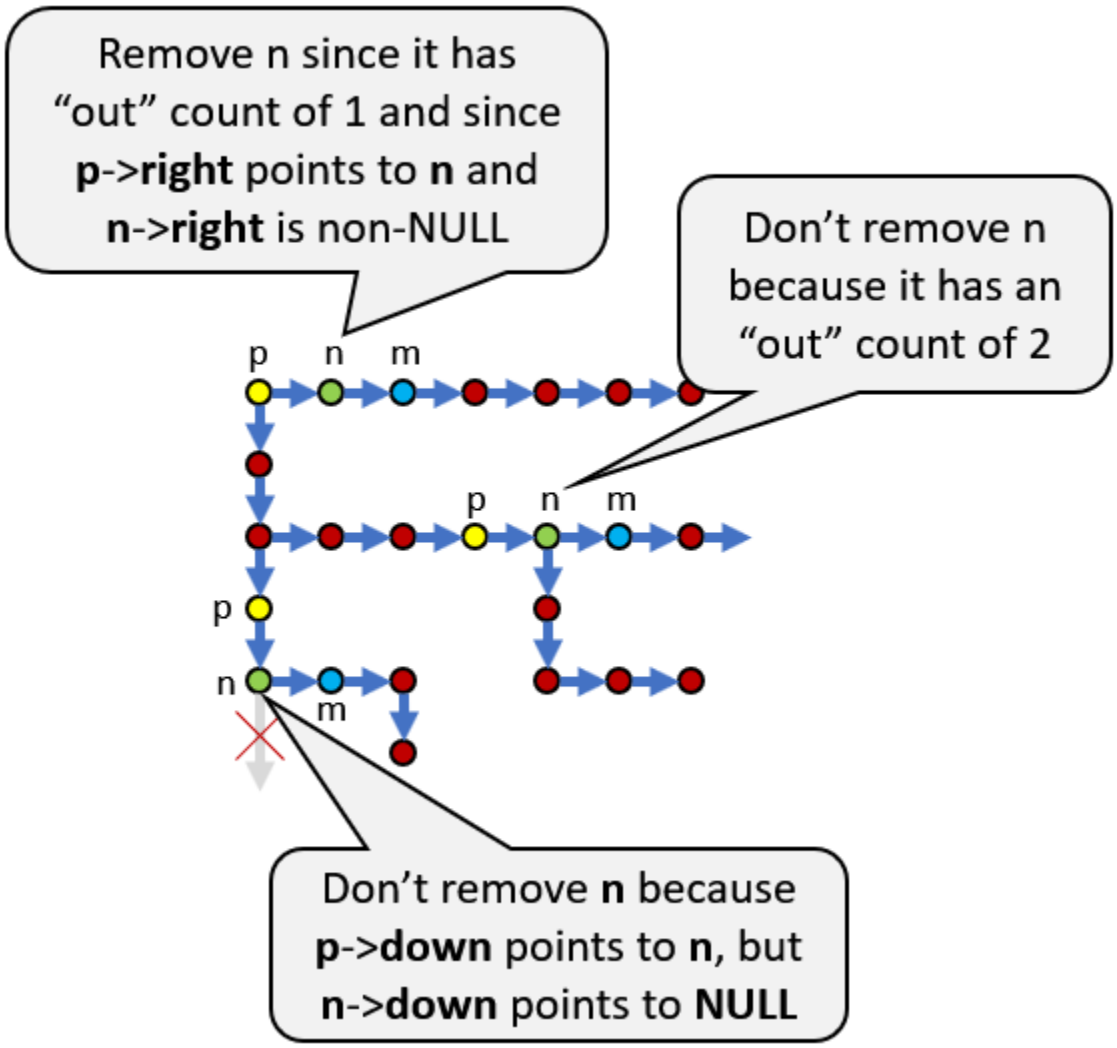
a) **n** must have only one ***out*** connection. That means … exactly **3** of the **up**, **down**, **left** or **right** pointers should be **NULL**.
b) **previousNode** (i.e., the one that led to **n** as you traversed) should not be **NULL**.
c) one of the following must be true:
   - the **previousNode**'s **y** value is greater than **n**'s **y** value and the **up** pointer of **n** (let's call it node **m** for further explanation below) is non-NULL)
   - the **previousNode**'s **y** value is less than **n**'s **y** value and the **down** pointer of **n** (let's call it node **m** for further explanation below) is non-NULL)
   - the **previousNode**'s **x** value is greater than **n**'s **x** value and the **left** pointer of **n** (let's call it node **m** for further explanation below) is non-NULL)
   - the **previousNode**'s **x** value is less than **n**'s **x** value and the **right** pointer of **n** (let's call it node **m** for further explanation below) is non-NULL)

To remove node **n**, it is like removing it from the middle of a linked list. The **previousNode**'s pointer must be changed from pointing to **n**, to now point to **m**. Basically, in each of the 4 sub-cases of case

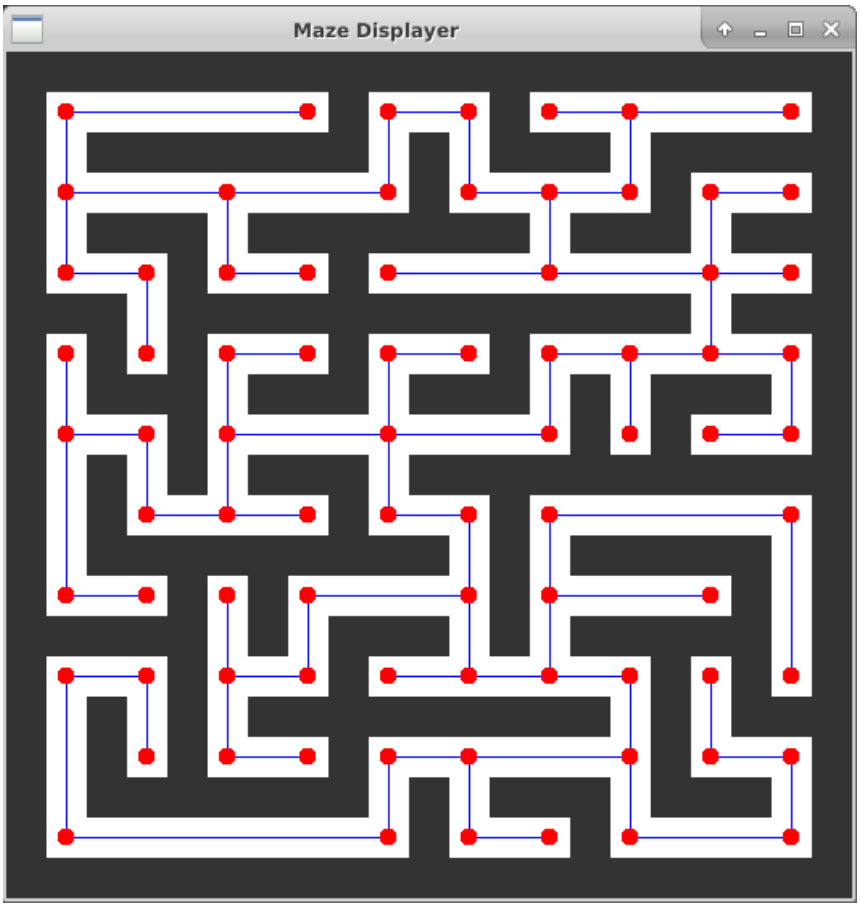
c) above, you need to change the appropriate pointer. There is just one to change. Once this is done, you should recursively call the procedure again with nodes **m** and **previousNode**. Do not remove node **n** until you return from the recursion.

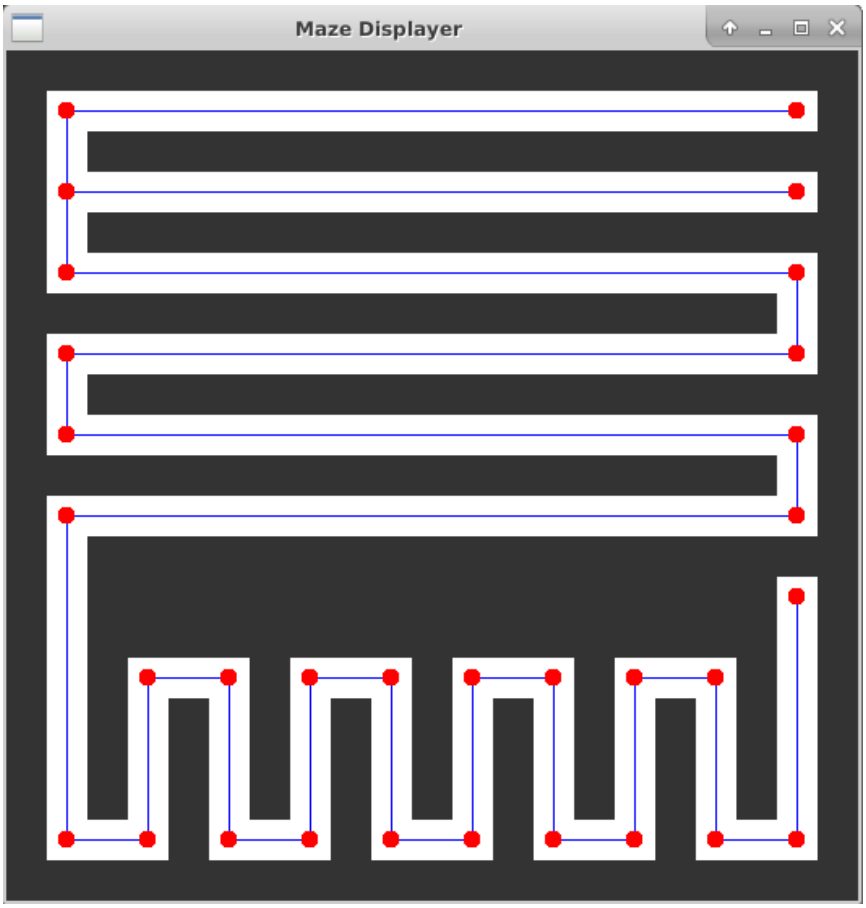
If any of the conditions (a), (b) **OR** (c) above were not satisfied, then **n** is not a node to be removed, so you should just recursively check in all 4 directions up, down, left and right to find any other nodes that need to be removed.

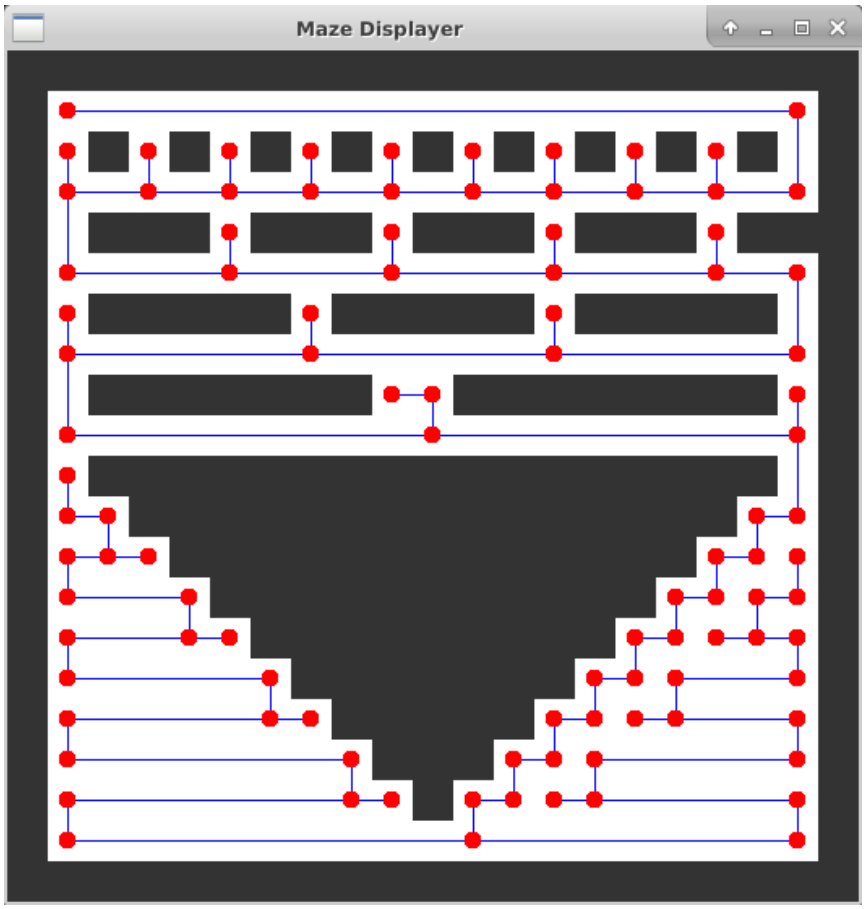
For further help, see the diagram below which has three examples of previous (i.e., **p**), **n** and **m** pointers. Only one case satisfies the conditions for removal of **n**.

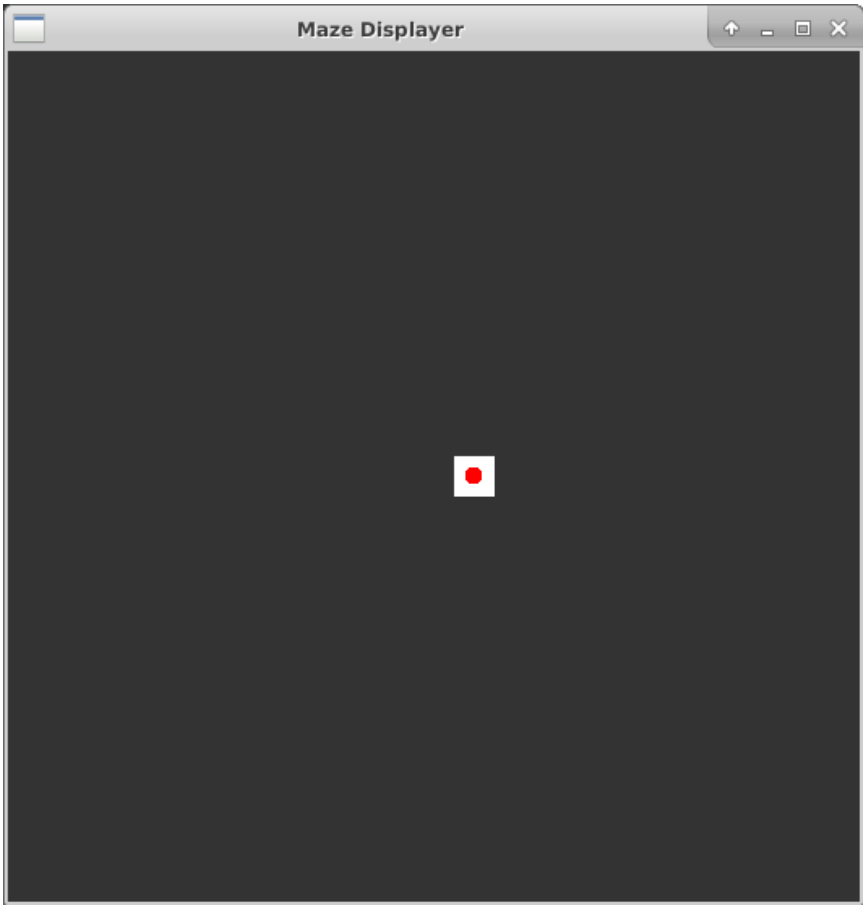


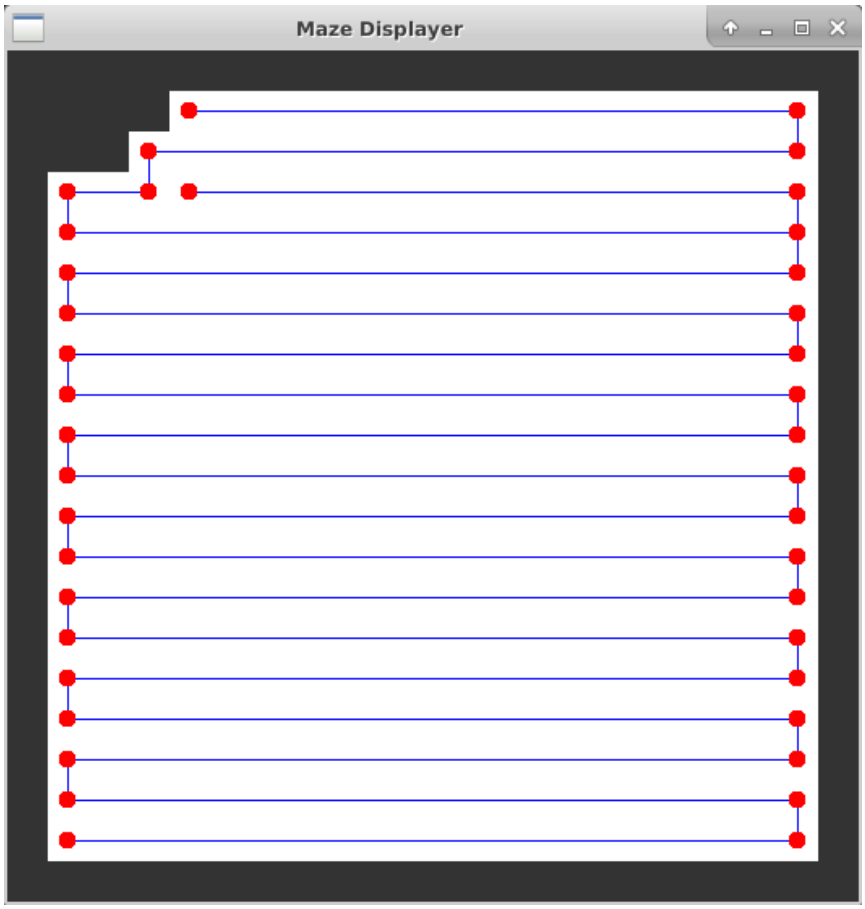
Once you get the code working, you should test it by uncommenting the calls to **cleanUpGraph()**, **drawMaze()** and **drawGraph()** in the FOR loop of the **main()** function. As you cycle through the graphs, you should see the results as shown on the next pages (again … this assumes that the recursive ordering is up/down/left/right).


Maze Displayer


Maze Displayer


Maze Displayer


Maze Displayer



(6) Finish your code so that at the end of the main function, it releases all of the dynamically-allocated memory. To make sure that your allocated memory has been freed properly, use **valgrind** when you run your final version:

```
valgrind --leak-check=yes ./mazes
```

You should see something like this (process ID will vary) appear at the end of the output:

```
==3960== HEAP SUMMARY:
==3960==     in use at exit: 0 bytes in 0 blocks
==3960==   total heap usage: 1,055 allocs, 1,055 frees, 111,503 bytes allocated
==3960==
==3960== All heap blocks were freed -- no leaks are possible
```

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code
2. All of your **.c source** files and all other files needed for testing/running your programs.
3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
