De Anza College
Summer 2016

CIS 22C: Data Abstraction and Structures
Programming Project 4 (*Rat-in-a-Maze*)

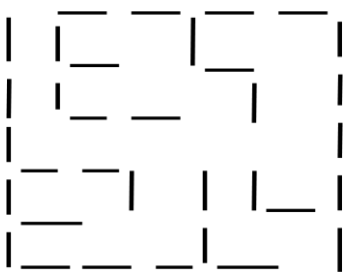
For this last programming project, you will be implementing several data structures and algorithms that we discussed in class. Since this programming project is relatively larger than the first 3 programming projects you had, you can pair up with one of your classmates and complete this project in 3 weeks. I highly recommend that you start working on this project as soon as possible and not to wait to the last week or days before starting. Additionally, the weight for this last project will be twice the weight of each of the first 3 projects you had.

This project asks you to write a program that can:

- Generate a random two-dimensional square maze whose size is specified by the user; and
- Read in a maze from a given text file (more about this later).

Once the program has generated a random maze or read in a maze from a file, the program would solve the maze by finding a path from the start position to the goal position in the maze. Because of the how the maze is generated (to be discussed next) or specified in the file, there will always be a path from the starting position to the goal position in the maze.

The maze structure: The maze is an $n \times n$ grid of cells (which we also call rooms), where n is a positive integer specified by the user. Each room in the maze can have at most 4 doors, each of which (if open) can lead to the adjacent room to the north, south, east, or west. There is a passage way between two adjacent rooms if and only if both doors connecting the two adjacent rooms are open. There are two special rooms in the maze called the start room and the goal room. The start room is always the upper-left room in the maze and the goal room is always the bottom-right room in the maze. The start room has its door leading to the north always open while the goal room has its door leading to the south always open. Rooms on the boundary of the maze (except the start and goal rooms) have their doors leading out of the maze always close. As an example, the following figure shows a randomly generated 5 × 5 maze rendered in ASCII characters where each horizontal or vertical line character denotes closed door(s).



***Maze generation*:** To randomly generate an $n \times n$ maze, we use the algorithm below. The algorithm assumes that the rooms are uniquely numbered from left to right, top to bottom. The numbering of rooms starts from 0 and ends with N − 1, where N is the total number of rooms in the maze (i.e., $N = n^2$). For the example maze given above, the start room is numbered 0 while the goal room is numbered 24.

```
Initialize a disjoint sets S of room numbers 0, 1, 2,...., N - 1

While (S.find(0) != S.find(N - 1))
     Choose randomly a pair (i, j) of adjacent rooms i and j
     If (S.find(i) != S.find(j))
        Open the doors connecting rooms i and j in the maze
        S.union(S.find(i), S.find(j))
```

***Reading a maze from file*:** In addition to having the program generate a random maze from scratch, the program should also be able to read in a maze from a file whenever the program is executed with a command line argument representing the name of the text file containing the maze. For example, suppose the name of the executable program is **main.exe** and the name of the text file containing the maze is **maze.txt**, whenever the user types **"main maze.txt"** on the command-line, the program should read in the maze specified in the file **maze.txt** instead of randomly generating one. Assume that the text file containing the maze has the following format:

<value of n>
<door 0 in room 0> <door 1 in room 0> <door 2 in room 0> <door 3 in room 0>
<door 0 in room 1> <door 1 in room 1> <door 2 in room 1> <door 3 in room 1>
<door 0 in room 2> <door 1 in room 2> <door 2 in room 2> <door 3 in room 2>
.
.
.

where each <door x in room y> in the above is either a 0 or a 1. If <door x in room y> is a 0, it means that door *x* in room *y* is open. If <door x in room y> is a 1, it means that door *x* in room *y* is close. As a convention, we let door 0 in every room be the door leading to the north, door 1 in every room be the door leading to the south, door 2 in every room be the door leading to the east, and door 3 in every room be the door leading to the west. For example, the above given maze is represented by the following file contents (note that only the first 7 rooms are actually shown):

```
5
0 0 1 1
1 1 0 1
1 0 1 0
1 1 0 1
1 0 1 0
0 0 1 1
1 1 0 1
.
.
.
```

***Solving the maze*.** Once the program has generated a random maze from scratch or has read in a maze from a file, the program should then solve the maze by finding a path from the start room to the goal room. To solve the maze, the program would try two algorithms: breadth-first search **(BFS)** and depth-first search **(DFS).** The **BFS** algorithm for solving the maze is as follows:

```
Create an empty queue Q of room numbers
Enqueue room number 0 to Q
Mark room number 0 as visited

While Q is not empty {
  Dequeue an element from Q and store it to i
  If i is equal to (N - 1) then break from the while-loop and print
  the path found
  For each room j adjacent to room i such that there is a passage way
  between rooms i and j
     and room j is not marked visited {
        Enqueue room number j to Q
        Mark room number j as visited
}
}
```

The DFS algorithm for solving the maze is as follows:

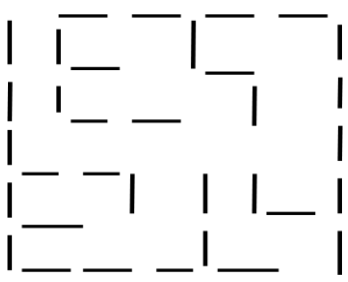
```
Create an empty stack S of room numbers
Push room number 0 to S
Mark room number 0 as visited

While S is not empty {
    Pop an element from S and store it to i
    If i is equal to (N - 1) then break from the while-loop and print
    the path found
    For each room j adjacent to room i such that there is a passage
    way between rooms i and j
        and room j is not marked visited {
            Push room number j to S
            Mark room number j as visited
}
}
```

In both the BFS and DFS algorithms above, we assume that adjacent rooms are considered in the following order: north, south, east, and west. <u>This is important</u>.

The following shows a sample run of the program and the desired text/ASCII output.

Enter size of maze: 5



```
Rooms visited by BFS: 0 5 10 11 12 17 13 22 8 18 14 21 7 23 9 19 16 20
2 6 24
This is the path (in reverse): 24 23 18 13 12 11 10 5 0
This is the path.
X
X
X X X X
      X
      X X
```

```
Rooms visited by DFS: 0 5 10 11 12 13 14 19 9 4 3 18 23 24
This is the path (in reverse): 24 23 18 13 12 11 10 5 0
This is the path.
X
X
X X X X
      X
      X X
```

For this project, you should implement and use your own stack, queue, and disjoint sets template classes. For computing the path found by BFS and DFS, you may use the C++ STL map class.

The following are the additional instructions for this programming project:

- This programming project is to be done individually or in groups of 2.
- Create a **`readme.txt`** file that describes exactly how to compile and execute your program.
- Collect your source codes, readme file, and other files needed to compile and execute your program into one ZIP file called **YourLastName_YourLastName2_prog4.zip**. **Please DO NOT include any executable files in your ZIP file.**
- Make sure you follow good object-oriented programming approach, good coding style, and proper documentation.

Submitting Assignments:

Submit the ZIP file on Catalyst.

No late submission will be accepted.

Your .zip file should contain one project workspace. You can delete the "Debug" folders and any .ncb files before creating your zip. They are not needed.

Your projects must build properly. Any assignment that does not build properly receives a grade of 0.