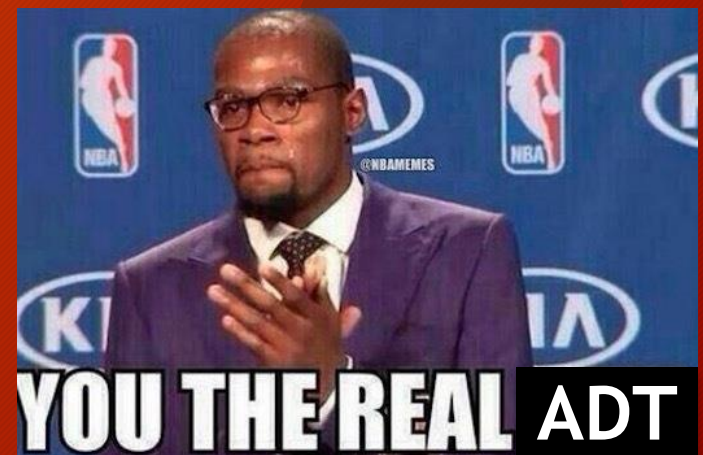


YEAH A2

Fun with Collections!



Agenda

- Logistics
- Part 1: Maze
- Part 2: Index



Logistics

- Due: Wed July 8th
- Grace period: 1 day (EOD Thurs!)



Questions about logistics?



Agenda

- ~~Logistics~~
- Part 1: Maze
- Part 2: Index



Warmups

- The warmups were designed to be fairly straightforward problems that equip you with the tools needed to complete the rest of the assignment!
 - Please do the warmups **before** you attempt the assignment.
 - I promise they'll make your life easier 😊
 - Your only deliverables for this part will be in **shortanswer.txt**.
- 

Warmup 0: Patch the Debugger!

- In order to view ADT's cleanly in the debugger, you'll need to install a little patch to your debugger. Installation steps are on the handout, and are straightforward!
- Thanks to SL Jeremy Barenholtz for the fix!



Warmup 1: Observing ADT's in the Debugger

- For the first part, you'll examine the following function:
- This function is already implemented for you, and you'll be using the **debugger** to step through it!
- To view the internals, put a **breakpoint** on the first line of the function and run the program in the debugger!

```
void reverse(Queue<int>& q) {  
    Stack<int> s;  
    while (!q.isEmpty()) {  
        int val = q.dequeue();  
        s.push(val);  
    }  
    while (!s.isEmpty()) {  
        int val = s.pop();  
        q.enqueue(val);  
    }  
}
```

Warmup 1: Observing ADT's in the Debugger

- When you run the warmup tests, you'll see a window pop up in the top right corner of your debugger.
 - Inside you'll see the names of the variables in your function! You can expand them via the arrows on the left!
- As you can see, you can peek into the queue at runtime! As you step thru the debugger, you can watch these values change in real time!

Name	Value	Type
q	@0x7fffc510...	Queue<int>
vptr	_vptr.Queue	
_elements	@0x7fffc510...	Deque<int>
vptr	_vptr.Deque	
_elements	<4 items>	std::dequ...
[0]	3	int
[1]	9	int
[2]	1	int
[3]	6	int
_version	@0x7fffc510...	stanfordc...
sunetid	"abby"	std::string

There's some scary stuff here :o. Don't worry about it just yet, we'll talk about it in a few weeks!

Warmup 2: Debugging an ADT question

- For this next part, you're going to use your new debugging skills to debug this function:
- Step through the function for a few iterations and determine where things go wrong!

```
void duplicateNegatives(Queue<int>& q) {  
    for (int i = 0; i < q.size(); i++) {  
        int cur = q.dequeue();  
        q.enqueue(cur);  
        if (cur < 0) {  
            q.enqueue(cur);    // double up on negative numbers  
        }  
    }  
}
```

Warmup 3: Debugging an Error

- For the final warmup, you'll be debugging the following function:
 - This function raises an error, meaning during runtime, it encounters a problem and terminates!
- The problem is, you're not allowed to modify a data structure while you loop thru it with a for each loop.

```
void removeMatchPairs(Map<string, string>& map) {  
    for (string key: map) {  
        if (map[key] == key) {  
            map.remove(key);  
        }  
    }  
}
```

Warmup 3: Debugging an Error

- You'll need to step thru this function in the debugger to determine **exactly where** it throws the error!
- Pro tip: remember this error for later - if you're looping thru some data, **don't modify it**. If the code doesn't crash, it'll usually give you strange and incorrect behavior.

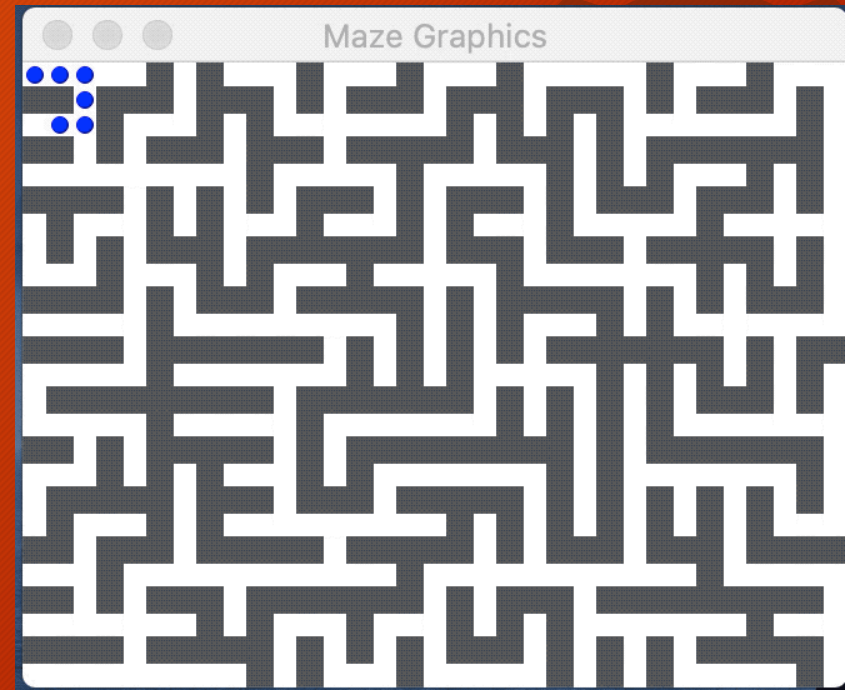
```
void removeMatchPairs(Map<string, string>& map) {  
    for (string key: map) {  
        if (map[key] == key) {  
            map.remove(key);  
        }  
    }  
}
```

Questions about the warmups?



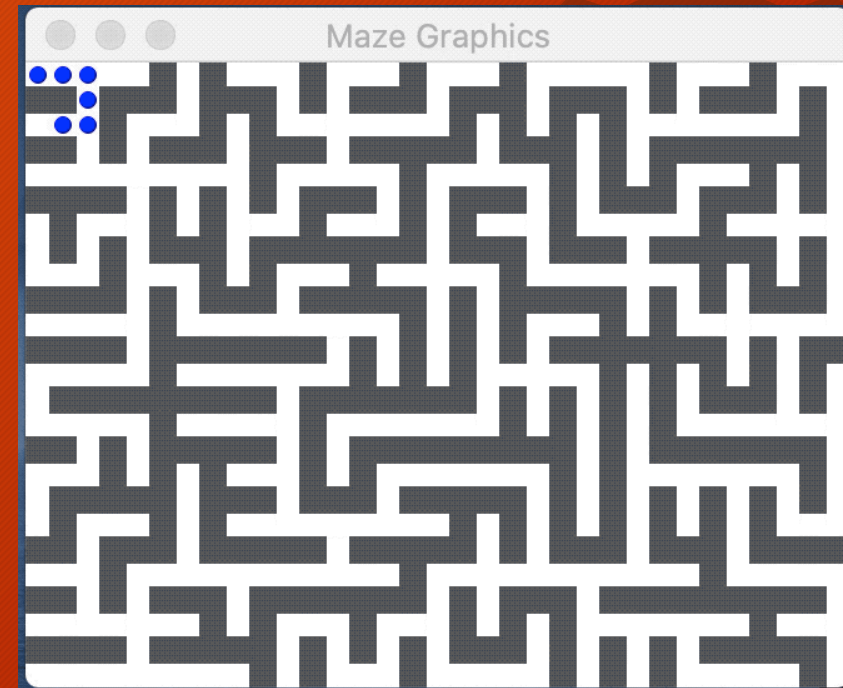
Part I: Maze

- Welcome to the first coding part of the assignment!
- You'll be working with a mazes like the one to your right - these mazes will start in the top left and have exits in the bottom right.



Part I: Maze

- Let's talk a little more about these mazes:
 - The maze itself is actually represented as a `Grid<bool>` in your program, where a cell marked `true` is an open corridor and a cell marked `false` is a wall.
 - `mazeGrid[row][col]` returns a Boolean that indicates whether `{row, col}` are the coordinates of a wall or a corridor!



Part I: Maze

- Where do these `Grid<bool>` come from?
 - Great question, rhetorical viewer. Mazes are provided to you in the starter code as **text files**.
 - We've provided you the function

```
void readMazeFile(string filename, Grid<bool>& maze) {
```

that reads in a text file like the one on the right and converts it into a `Grid<bool>`. We use '@' to represent walls and '-' to represent corridors.

```
- - - - -  
- @@@@-  
- - - - @-  
- @@@- @-  
- @- - - @-
```

Part I: Maze

GridLocations

- There's a new abstraction you'll need to become comfortable with using!
- A GridLocation represents a pair of coordinates. You can think of it like
 { row, col }

```
// You can create a GridLocation by separately setting its row and col fields
GridLocation chosen;
chosen.row = 3;      // separate assignment to row
chosen.col = 4;      // then col

// Or you can create a GridLocation by setting its row and col during initialization
GridLocation exit = { maze.numRows()-1, maze.numCols()-1 }; // last row, last col

// You can use GridLocations to index into a Grid (maze is a Grid)
if (maze[chosen]) // equivalent to maze[3][4]
...

// You can directly compare two GridLocations
if (chosen == exit)
...

// You can also access and use a GridLocation's row,col separately
if (chosen.row == 0 && chosen.col == 0)
...
```

Questions about the Grid<bool> or GridLocations?

```
// You can create a GridLocation by separately setting its row and col fields
GridLocation chosen;
chosen.row = 3;      // separate assignment to row
chosen.col = 4;      // then col

// Or you can create a GridLocation by setting its row and col during initialization
GridLocation exit = { maze.numRows()-1, maze.numCols()-1 }; // last row, last col

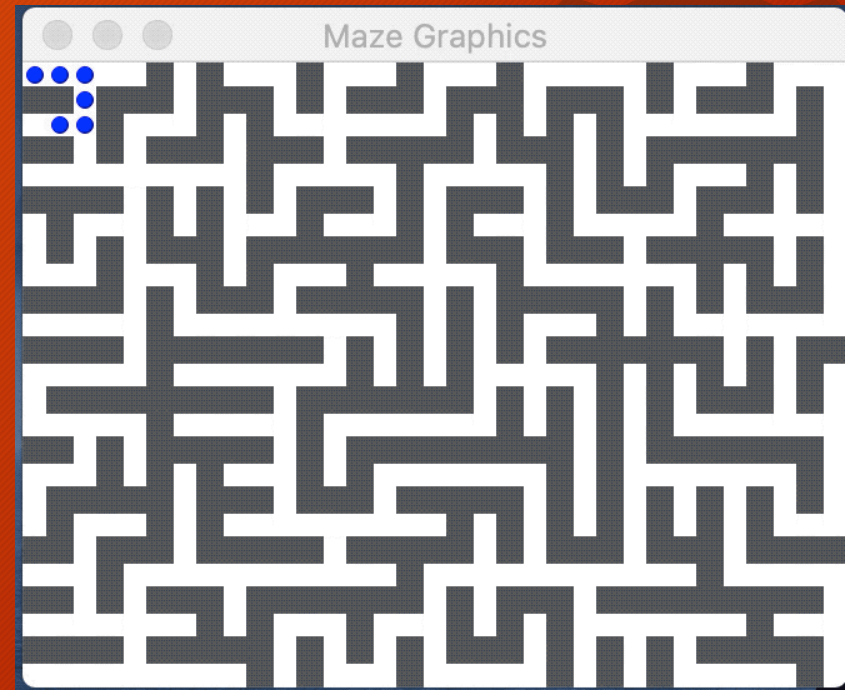
// You can use GridLocations to index into a Grid (maze is a Grid)
if (maze[chosen]) // equivalent to maze[3][4]
...

// You can directly compare two GridLocations
if (chosen == exit)
...

// You can also access and use a GridLocation's row,col separately
if (chosen.row == 0 && chosen.col == 0)
...
```

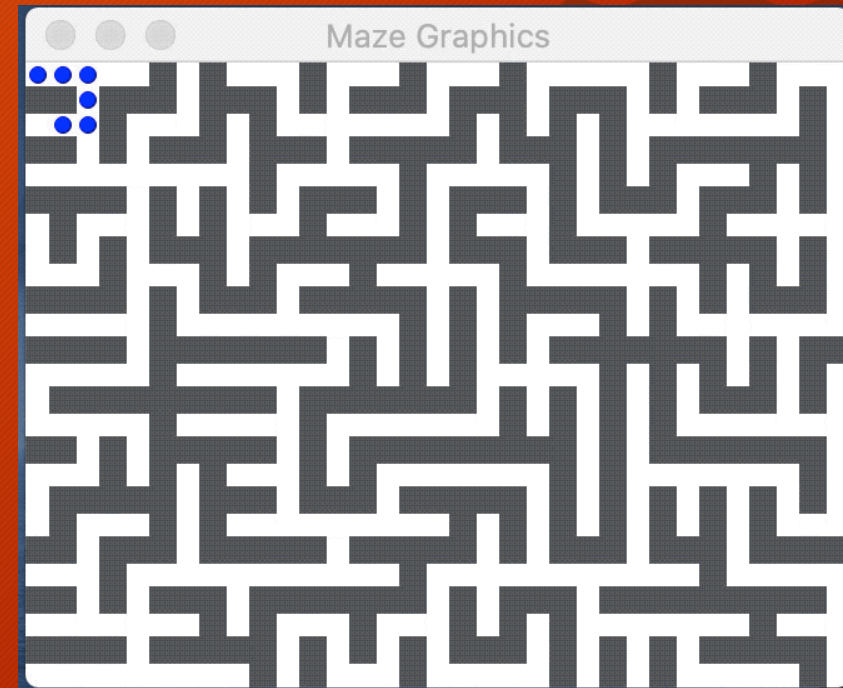
Part I: Maze

- How can we store a path in our maze?



Part I: Maze

- How can we store a **path** in our maze?
- We can use a `Stack<GridLocation>`
 - In this case, the **top** of the stack would represent the **last visited location** (the exit in a complete path!)
 - The bottom would be the **start** of the path, (typically the top left corner).



Part I: Maze

- For some of the mazes we provide for you, we also give you the maze **solutions** stored in a text file!
- Take a second to verify that the locations below represent a valid path out of the maze!
 - The path reads left(start) to right(end)

```
-----  
-@@@@@-  
-----@-  
-@@@-@-  
-@---@-
```

{r0c0, r0c1, r0c2, r0c3, r0c4, r0c5, r0c6, r1c6, r2c6, r3c6, r4c6}

Part I: Maze

- We've also written a function for you that turns a solution .txt file into a `Stack<GridLocation>` called

```
void readSolutionFile(string filename, Stack<GridLocation>& soln) {
```

that you'll need to fortify later on!

```
{r0c0, r0c1, r0c2, r0c3, r0c4, r0c5, r0c6, r1c6, r2c6, r3c6, r4c6}
```

```
-----  
-@@@@@-  
-----@-  
-@@@-@-  
-@---@-
```

Questions about mazes, text files or data structures?

```
-----  
-@@@@@-  
-----@-  
-@@@-@-  
-@---@-
```

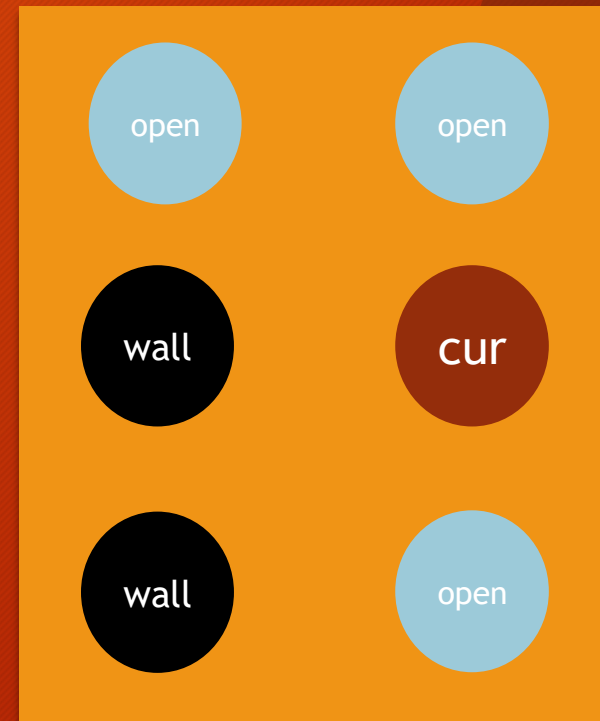
{r0c0, r0c1, r0c2, r0c3, r0c4, r0c5, r0c6, r1c6, r2c6, r3c6, r4c6}

Part I: Maze

generateValidMoves()

- To begin, you're going to implement the following function:
- This function takes in a maze as well as a current location in that maze.
- You are tasked with returning a set of **valid neighbors**
 - These are locations one step away in the (NSEW) directions that are non-walls and in bounds!

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```

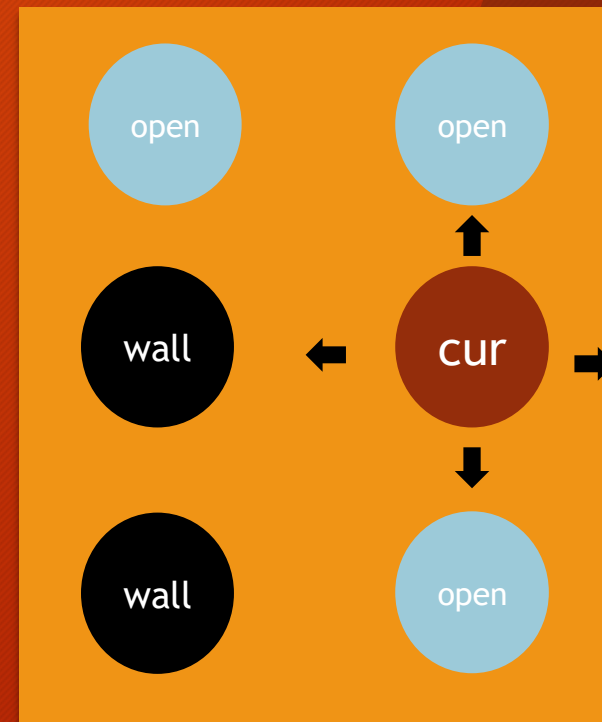


Part I: Maze

generateValidMoves()

- To begin, you're going to implement the following function:
- This function takes in a maze as well as a current location in that maze.
- You are tasked with returning a set of **valid neighbors**
 - These are locations one step away in the (NSEW) directions that are non-walls and in bounds!

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```

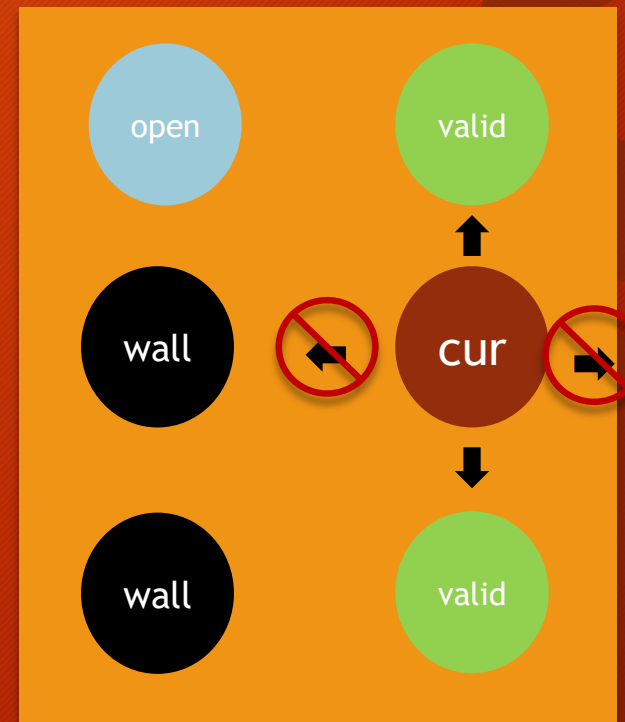


Part I: Maze

generateValidMoves()

- To begin, you're going to implement the following function:
- This function takes in a maze as well as a current location in that maze.
- You are tasked with returning a set of **valid neighbors**
 - These are locations one step away in the (NSEW) directions that are non-walls and in bounds!

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```

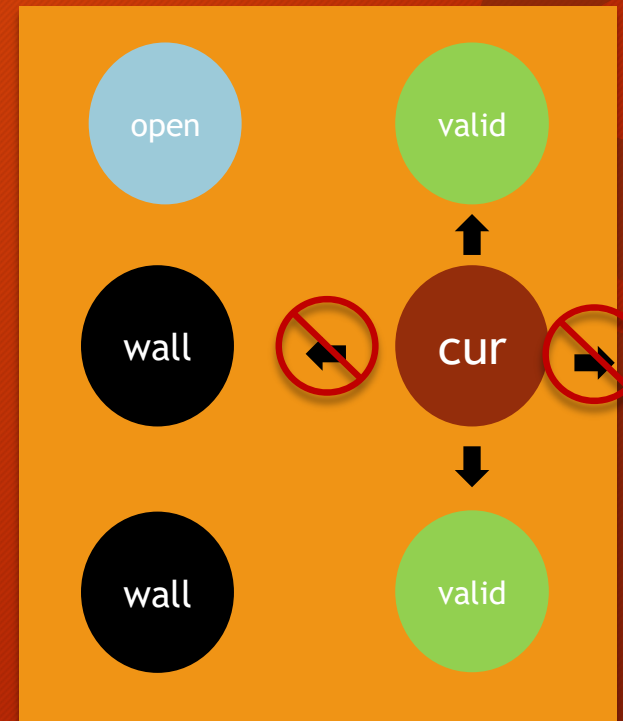


Part I: Maze

generateValidMoves()

- In this case, you'd return a `Set<GridLocation>` that contained 2 things: the coordinates of the above location and the coordinates of the below location.

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```

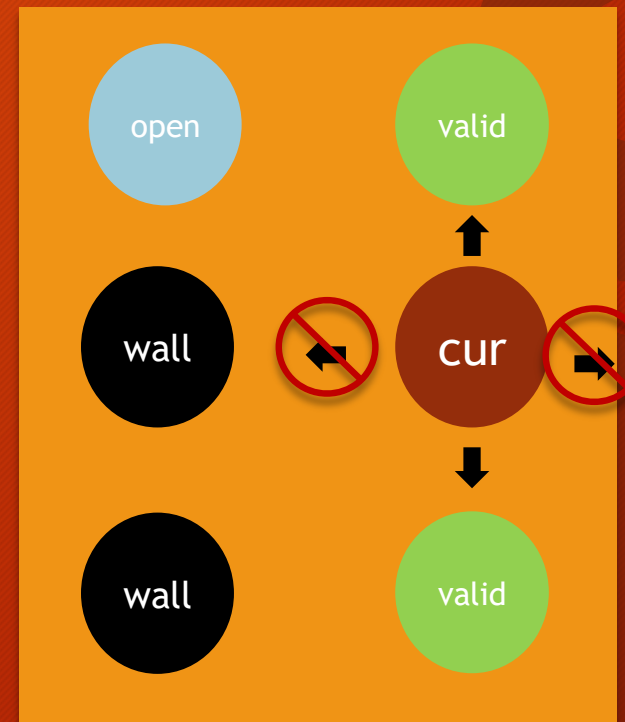


Part I: Maze

generateValidMoves()

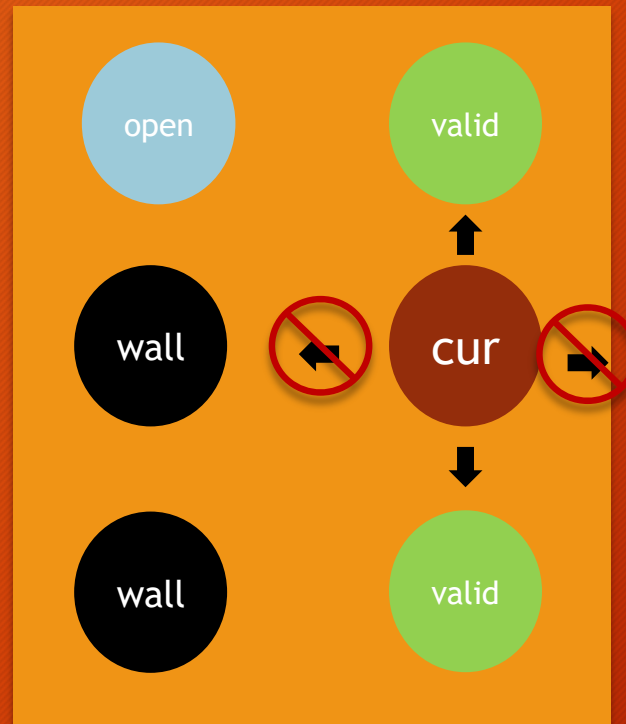
- Tips about generateValidMoves();
 - You can use the Grid's inbounds() function to tell whether a coordinate pair is within the bounds of a grid.
 - Be sure to add good tests for this part - we specifically leave edge cases out of the tests we provide you.
 - **You need to generalize your routine for validating a neighbor.** It is **poor style** to repeat the process of validation 4 times - once for each valid direction.
 - Think about how you might use a loop to fix this!

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```



Questions about generateValidMoves()?

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur)
```



Part I: Maze checkSolution()

- Let's say now that you generated a `Grid<bool>` and a `Stack<GridLocation>` representing a maze and a solution, respectively, and you wanted to verify that it actually was the solution to a maze.

```
-----  
-@@@@@-  
-----@-  
-@@@-@-  
-@---@-
```

```
{r0c0, r0c1, r0c2, r0c3, r0c4, r0c5, r0c6, r1c6, r2c6, r3c6, r4c6}
```

- How would you do it?

Part I: Maze checkSolution()

- Let's say now that you generated a `Grid<bool>` and a `Stack<GridLocation>` representing a maze and a solution, respectively, and you wanted to verify that it actually was the solution to a maze.
- How would you do it?



Part I: Maze checkSolution()

- Here's the criteria for a valid solution:

A **path** represents a valid solution through the **maze** if it meets the following criteria:

- The path must start at the entry (upper left corner) of the maze. $\{ 0,0 \}$
- The path must end at the exit (lower right corner) of the maze. $\{ \text{maze.numRows()} - 1, \text{maze.numCols()} - 1 \}$
- Each location in the path is within the maze bounds.
- Each location in the path is an open corridor (not wall).
- Each location is one cardinal step (N,S,E,W) from the next in path.
- The path contains no loops, i.e. a location appears at most once in the path.

Part I: Maze checkSolution()

If you identify that any of these things is **incorrect**, you can raise an error like this:

```
error("Here is my message about what has gone wrong");
```

A **path** represents a valid solution through the **maze** if it meets the following criteria:

- The path must start at the entry (upper left corner) of the maze. $\{ 0,0 \}$
- The path must end at the exit (lower right corner) of the maze. $\{ \text{maze.numRows()} - 1, \text{maze.numCols()} - 1 \}$
- Each location in the path is within the maze bounds.
- Each location in the path is an open corridor (not wall).
- Each location is one cardinal step (N,S,E,W) from the next in path.
- The path contains no loops, i.e. a location appears at most once in the path.

Part I: Maze

checkSolution()

- You'll be implementing the following function:

```
void checkSolution(Grid<bool>& maze, Stack<GridLocation> path)
```

that verifies that PATH contains the correct sequence of locations that navigate through MAZE without doing anything fishy.

- The function raises an ERROR if PATH is invalid, and it does nothing if the path is valid.
- You can test this functionality with the EXPECT_ERROR() and EXPECT_NO_ERROR() functions in the simple test framework!

Part I: Maze

checkSolution()

```
void checkSolution(Grid<bool>& maze, Stack<GridLocation> path)
```

- A few more points about checkSolution:
 - One of the things you're going to have to do is examine the elements in PATH - but you can't use a for loop or a for-each loop to examine the internals of a Stack: what can you do instead?
 - Be sure that you test A LOT for this function - because there are so many cases, there's a lot of functionality and edge cases that you're responsible for here!
 - To verify that each location is a cardinal step away from the next, think about how you can reuse your generateValidMoves() function to help.
 - If you need to keep track of "visited" items, Sets are great!

Questions about checkSolution()?

"Hey can I copy your homework?"

"Yeah just make sure to change it a little"

"Alright"



If you can't
checkSolution(), you might
run into this problem!
(please abide by the honor
code!)

Part I: Maze

`solveMaze()`

- Now it's time for the big code in this part: `solveMaze()`!

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

- Up to this point, you've been validating pre-generated maze solutions from text files. It's now time to generate your own solutions to a given maze!
- Given a `Grid<bool> MAZE`, it's your job to return a `Stack<GridLocation>` that contains the valid steps to escape it!

Part I: Maze solveMaze()

```
Stack<GridLocation> solveMaze(Grid<bool>& maze)
```

- To programmatically generate a solution to a given maze, you'll need to use a Breadth-First Search (BFS) like the one you learned on Wednesday 7/1.
 - Nick's great lecture covers BFS in depth, so I won't waste your time duplicating the logic. Check out his slides for more deets!
 - Here's the algorithm:

1. Create a queue of paths. A path is a stack of grid locations.
2. Create a length-one path containing just the entry location. Enqueue that path.
 - For simplicity, assume entry is always the upper-left corner and exit in the lower-right.
3. While there are still more paths to explore:
 - Dequeue path from queue.
 - If this path ends at exit, this path is the solution!
 - If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.
 - A location has up to four neighbors, one in each of the four cardinal directions. A neighbor location is viable if it is within the maze bounds, the cell is an open corridor (not a wall), and it has not yet been visited.

Part I: Maze `solveMaze()`

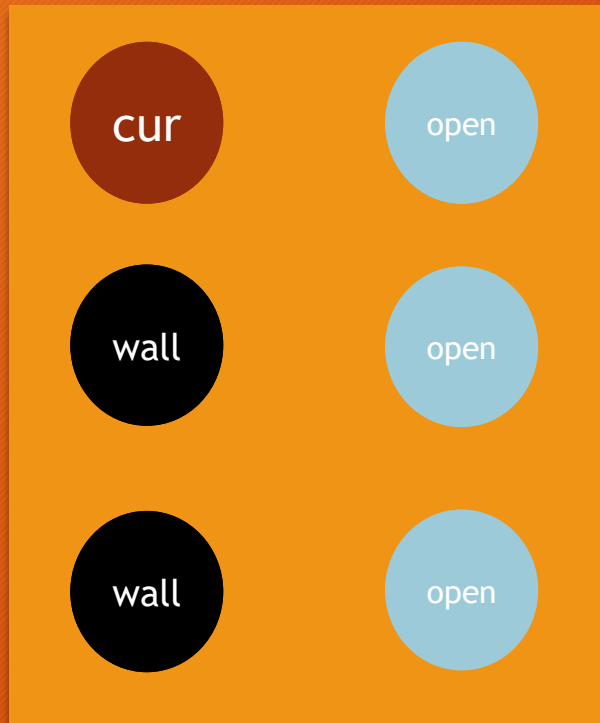
- Let's walk thru a very small example:



Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



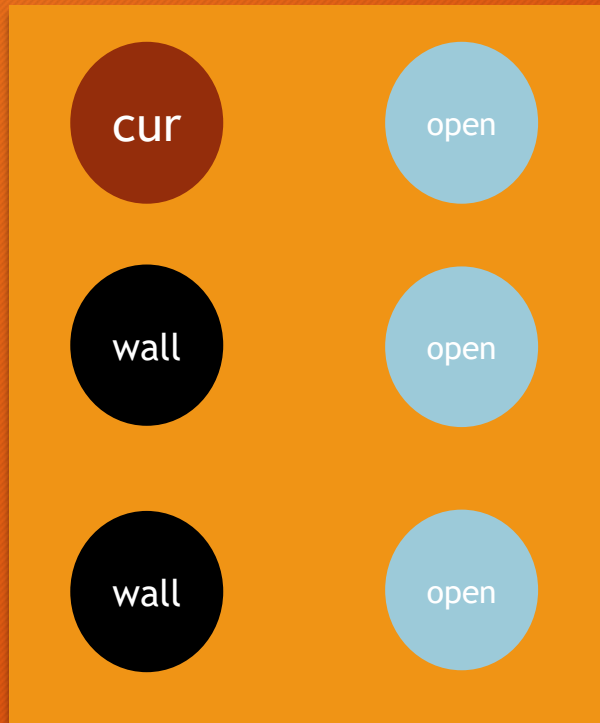
MAZE



Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



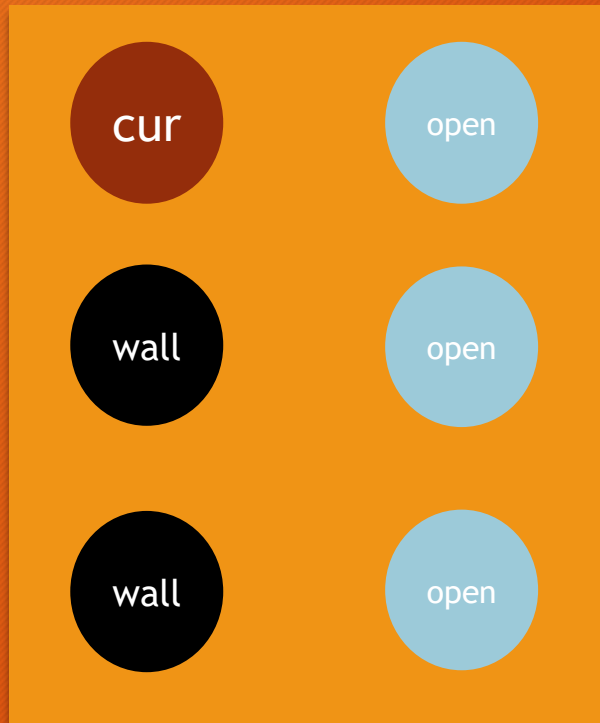
MAZE

- Create a queue of paths. A path is a stack of grid locations.

Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:

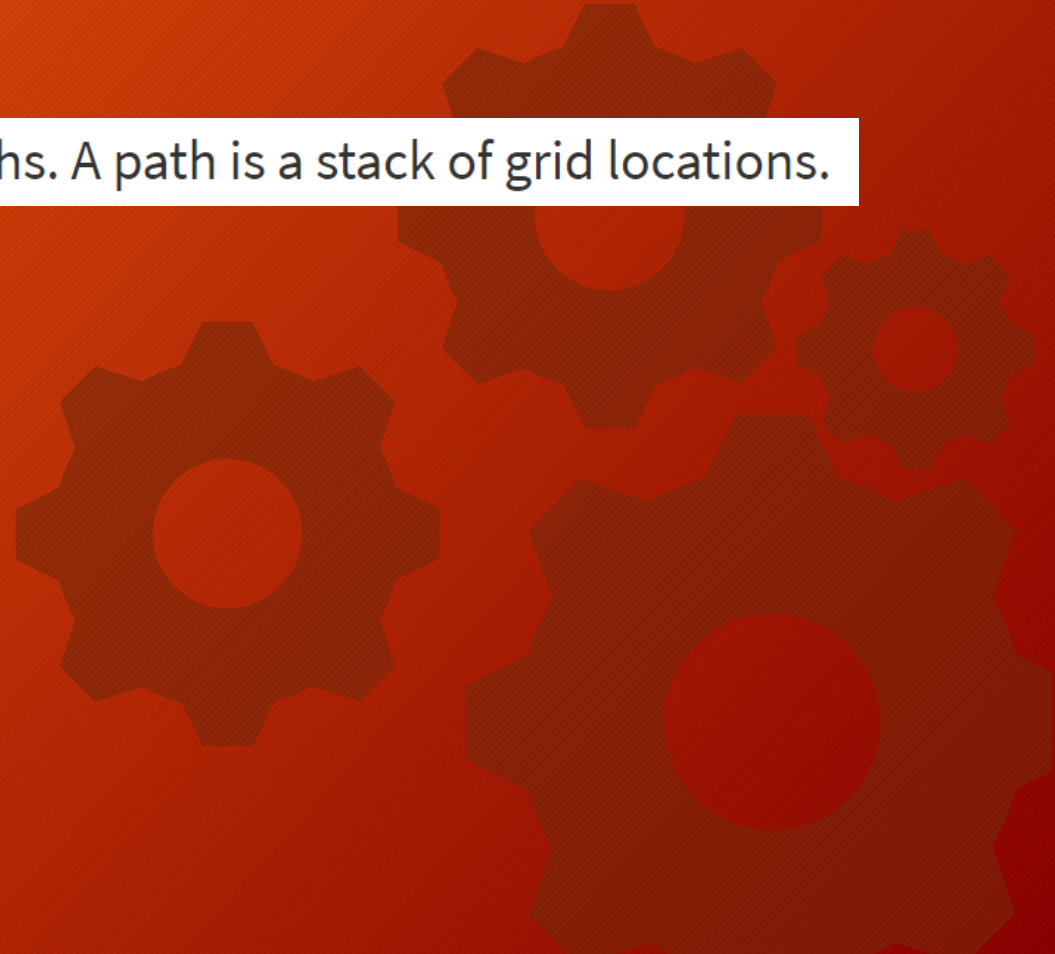


MAZE

1. Create a queue of paths. A path is a stack of grid locations.



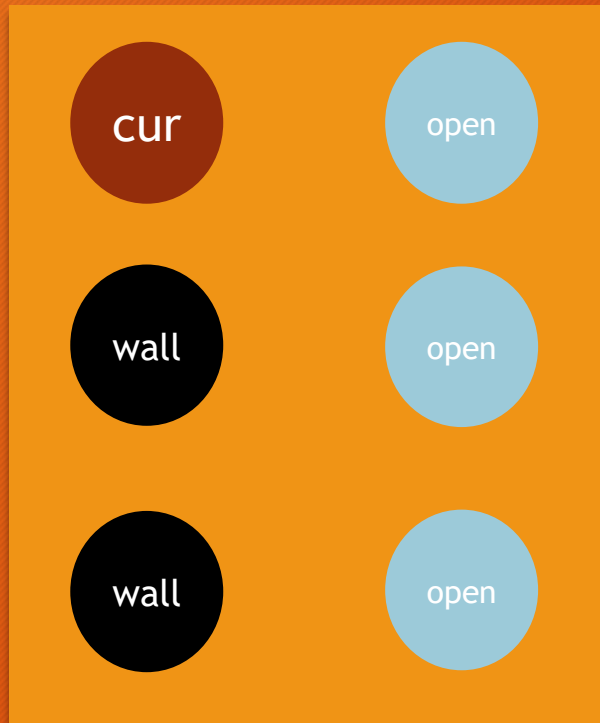
PATHS



Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:

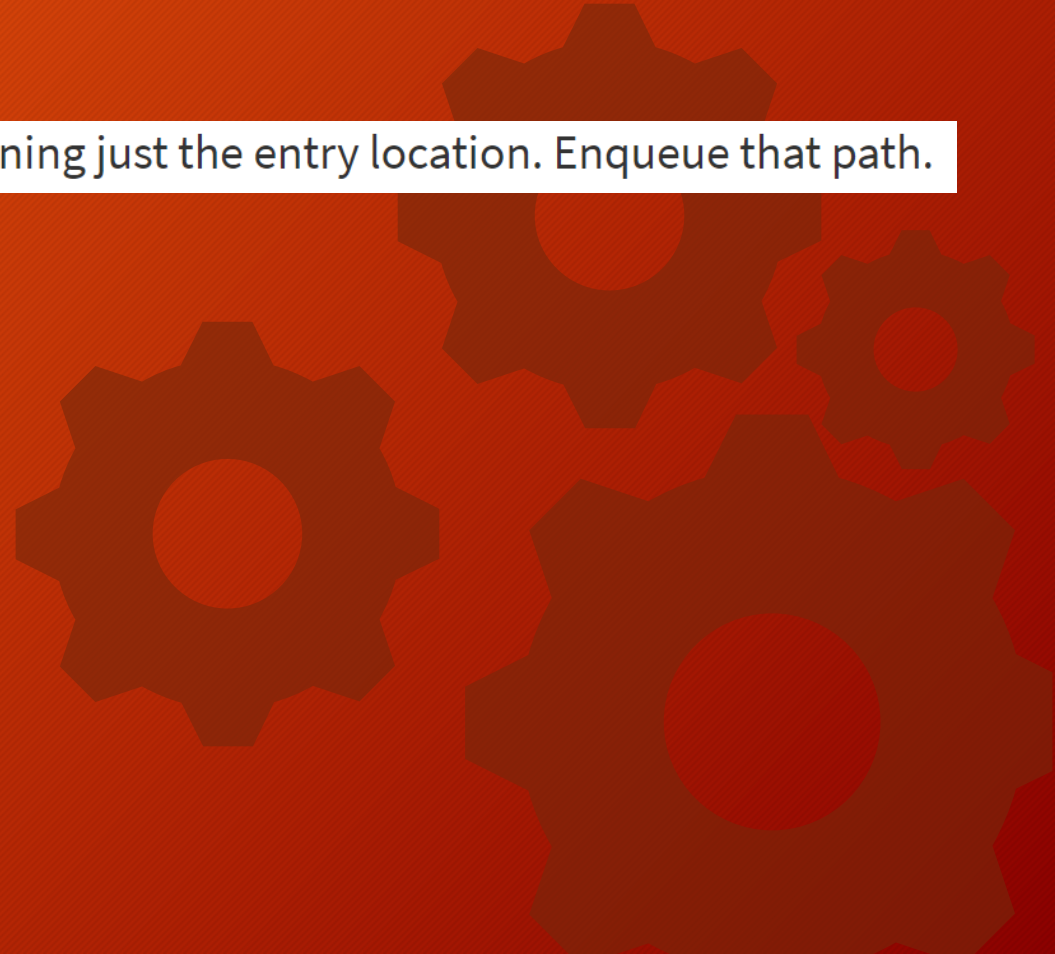


MAZE

2. Create a length-one path containing just the entry location. Enqueue that path.



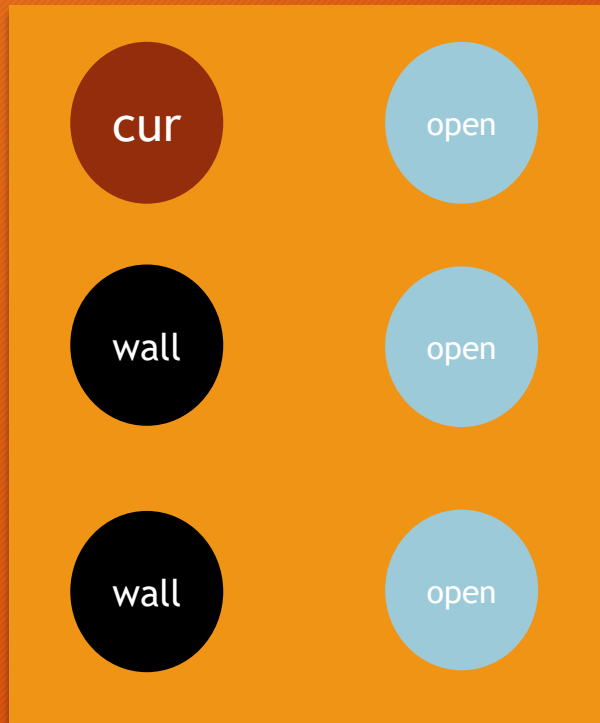
PATHS



Part I: Maze

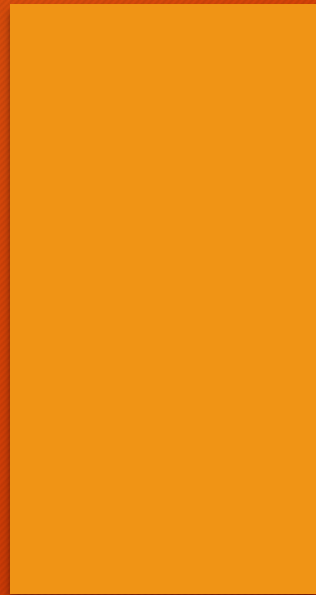
`solveMaze()`

- Let's walk thru a very small example:



MAZE

2. Create a length-one path containing just the entry location. Enqueue that path.



PATHS

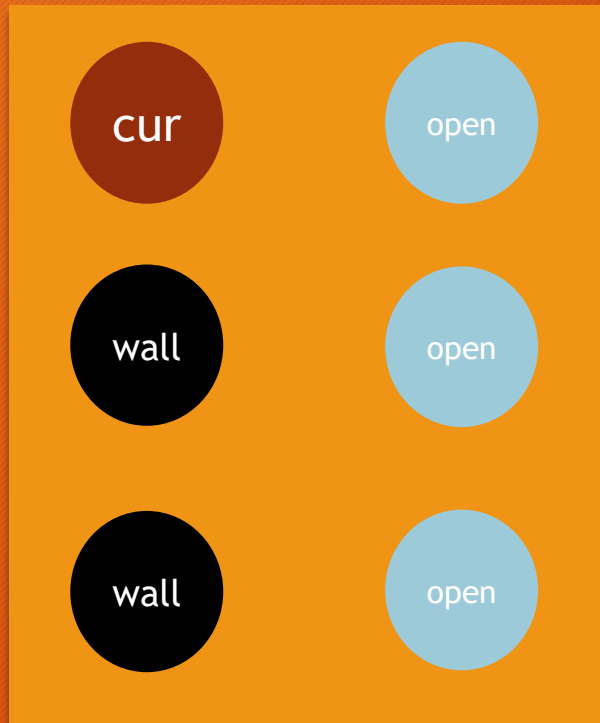
`{{0, 0}}`

PATH

Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



MAZE

2. Create a length-one path containing just the entry location. Enqueue that path.

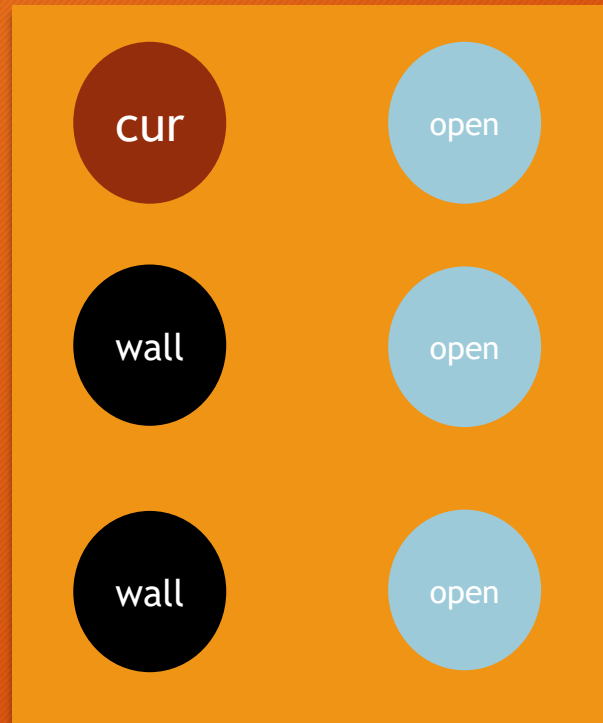
{ {0, 0} }

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

{ {0, 0} }



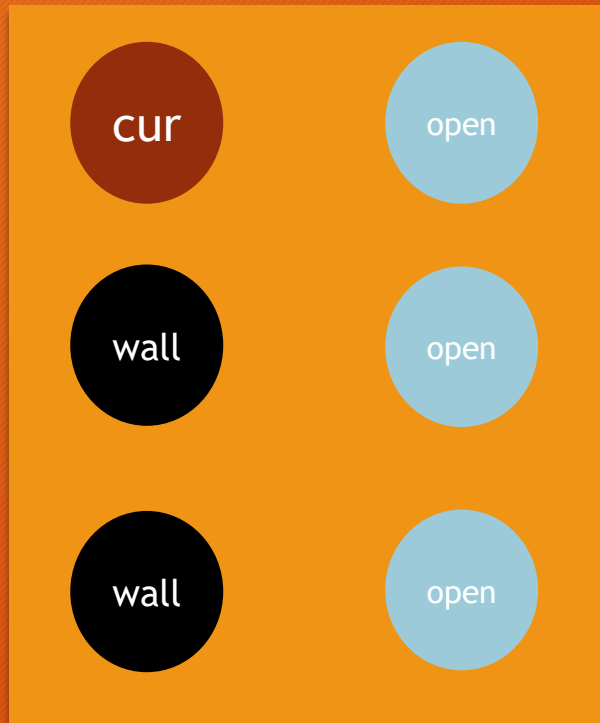
CURRENT_PATH

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.



PATHS

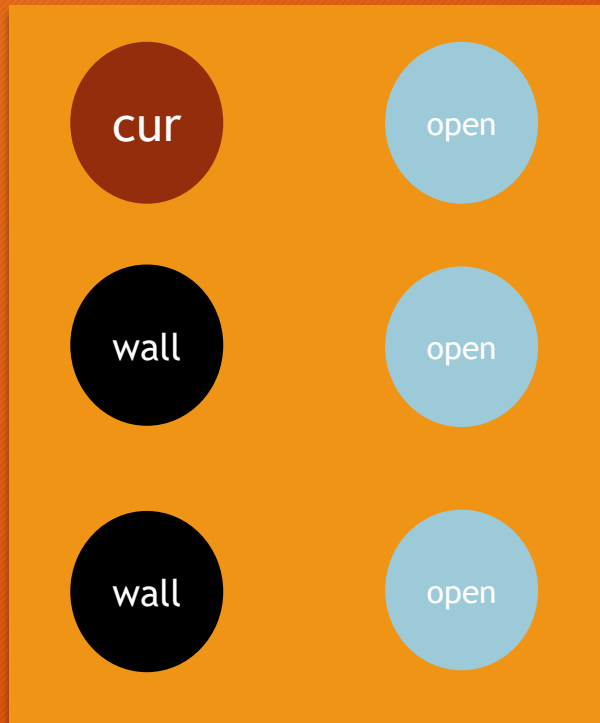
{ {0, 0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

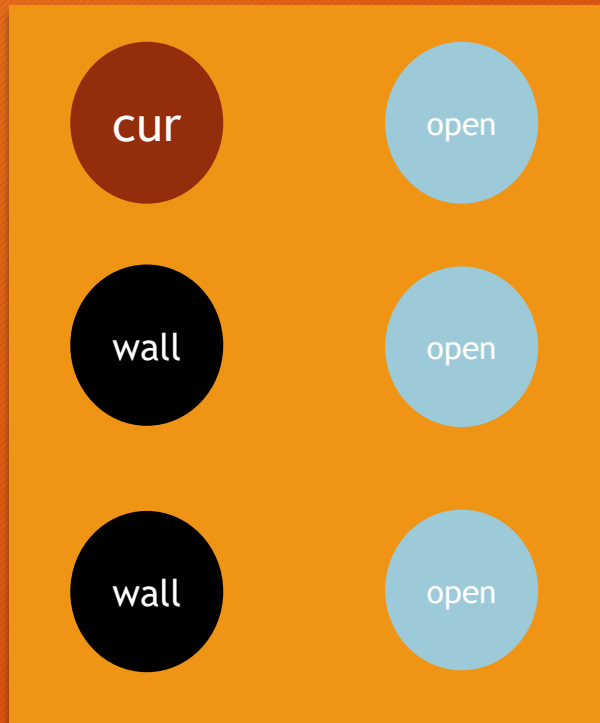
{ {0, 0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

{ {0, 0} }

CURRENT_PATH

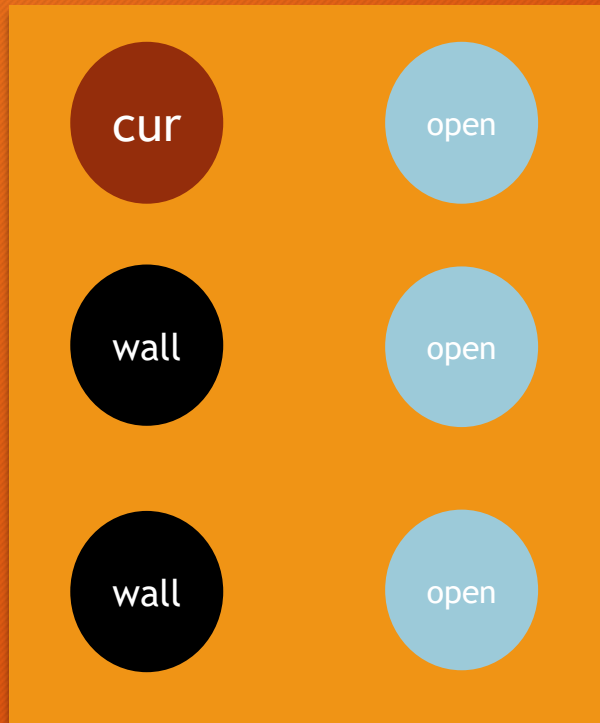
{0,0}

CURRENT_PATH_EXIT

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

{ {0, 0} }

CURRENT_PATH

{0,0}

CURRENT_PATH_EXIT

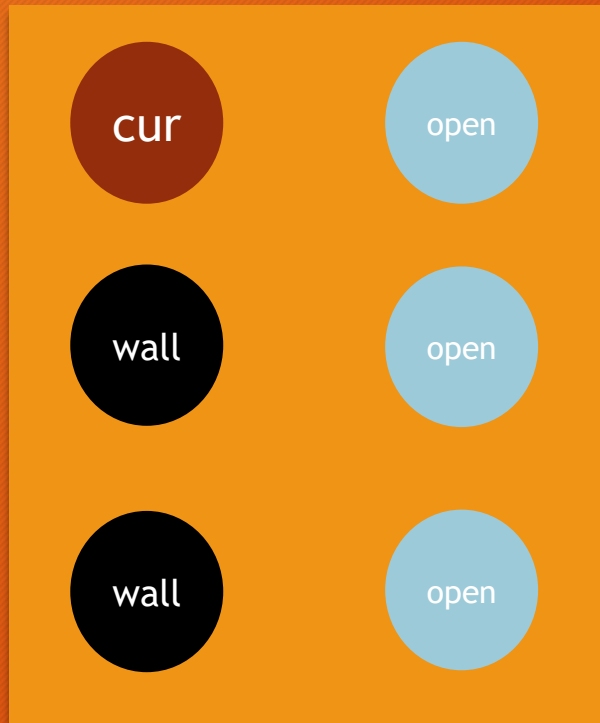
=

{1,2}

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

{ {0, 0} }

CURRENT_PATH

{0,0}

CURRENT_PATH_EXIT

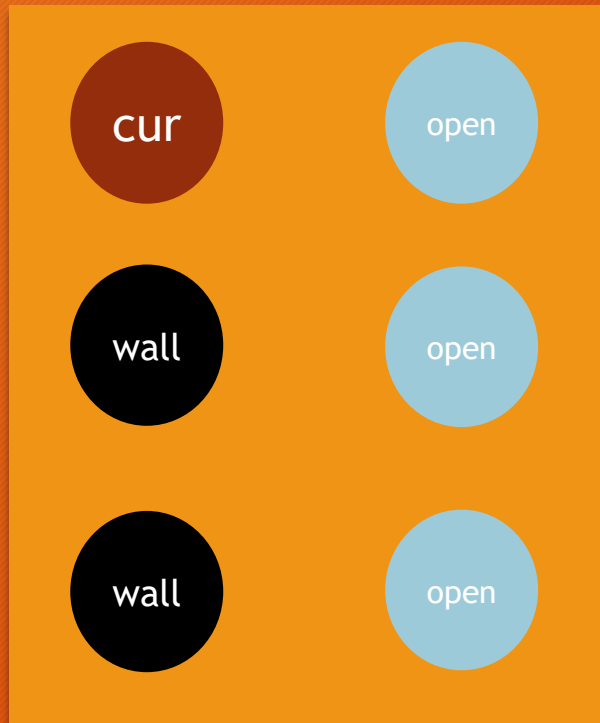


{1,2}

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

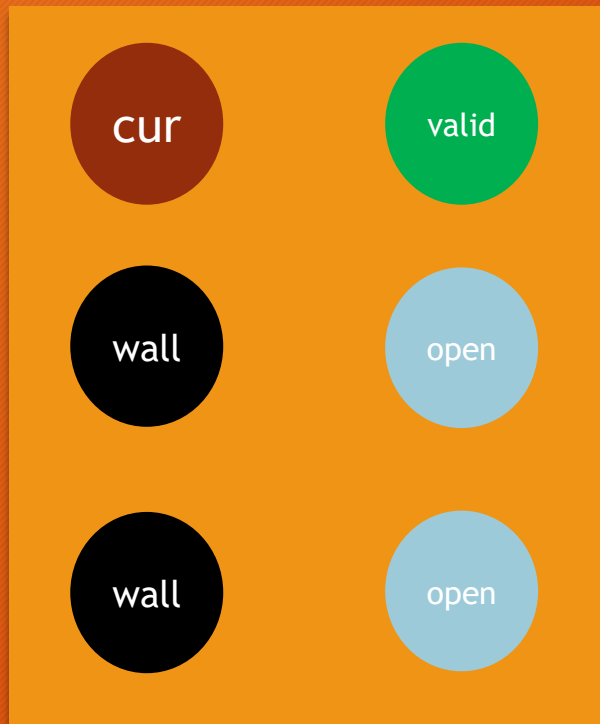
{ {0, 0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0} }

CURRENT_PATH

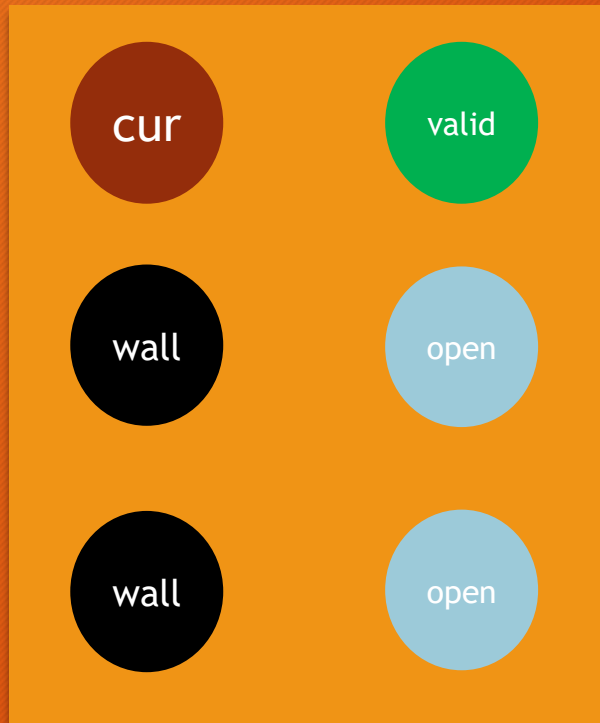
{ {1, 0} }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0} }

CURRENT_PATH

{ {0, 0} }

CURRENT_PATH_CPY

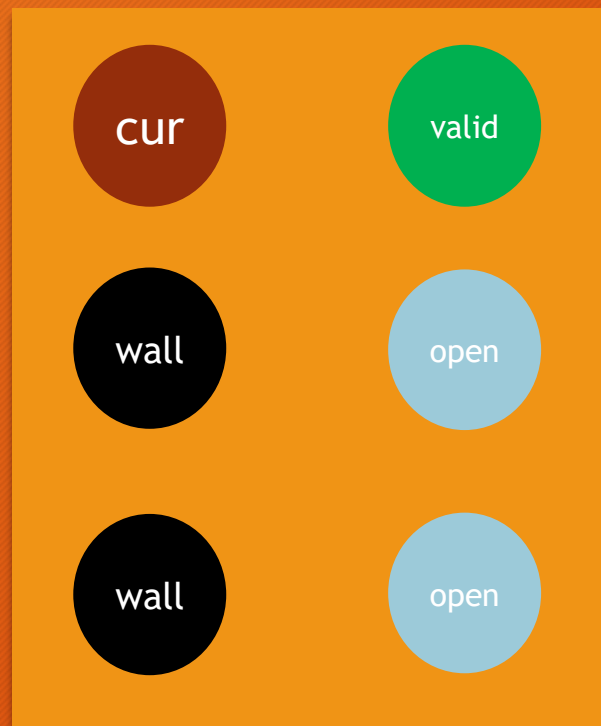
{ {1, 0} }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0} }

CURRENT_PATH

{ {0, 0}, {1,0} }

CURRENT_PATH_CPY

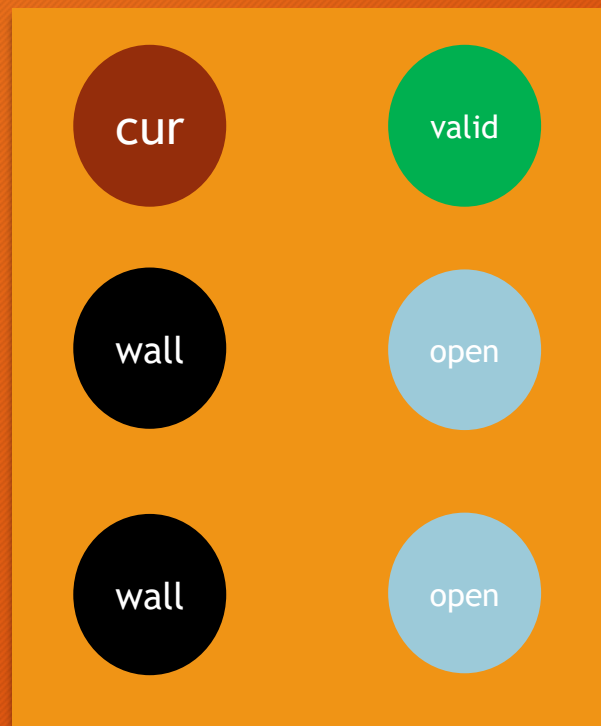
{ {1, 0} }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0} }

CURRENT_PATH

{ {0, 0}, {1,0} }

CURRENT_PATH_CPY

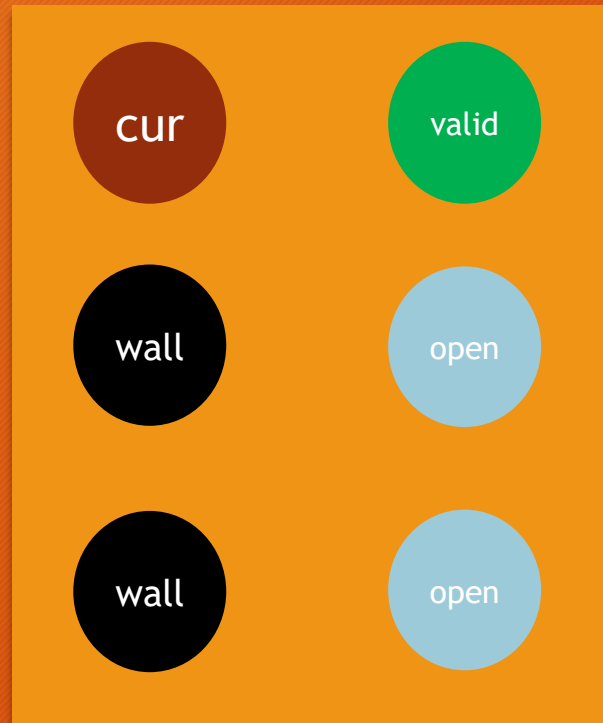
{ {1, 0} }

VALID_NEIGHBORS

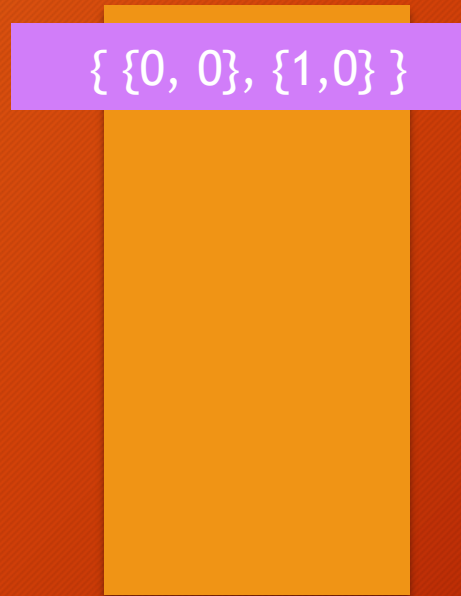
Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



MAZE



PATHS

Repeat!



CURRENT_PATH



CURRENT_PATH_CPY

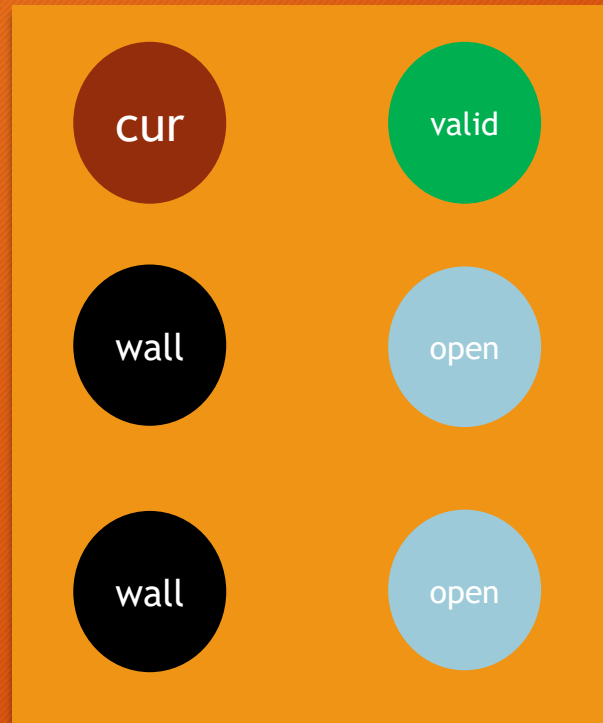


VALID_NEIGHBORS

Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

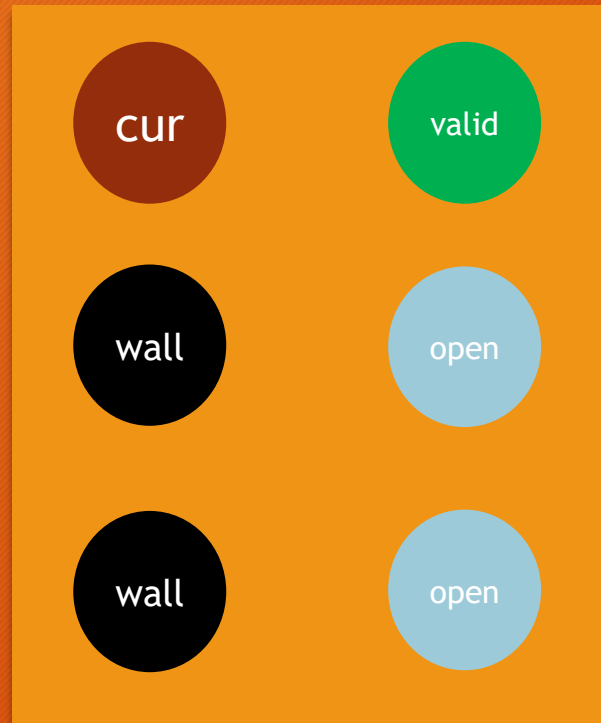
{ {0, 0}, {1,0} }

PATHS

Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

{ {0, 0}, {1,0} }

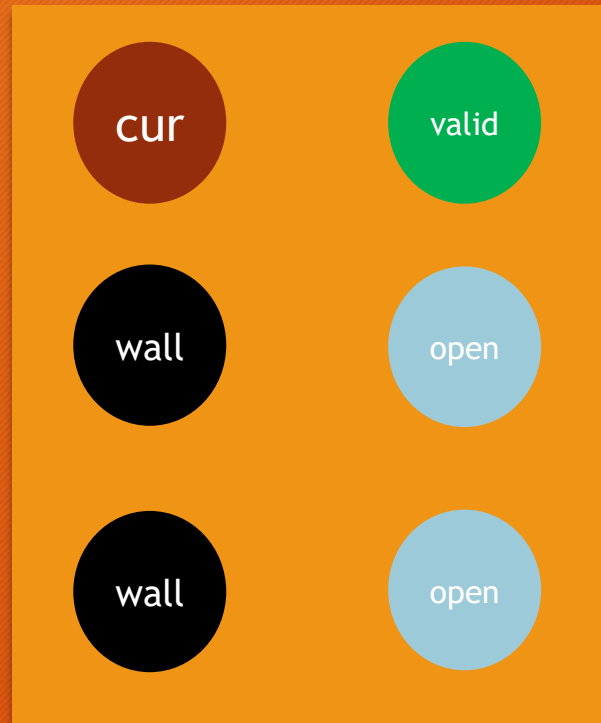


PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

{ {0, 0}, {1,0} }



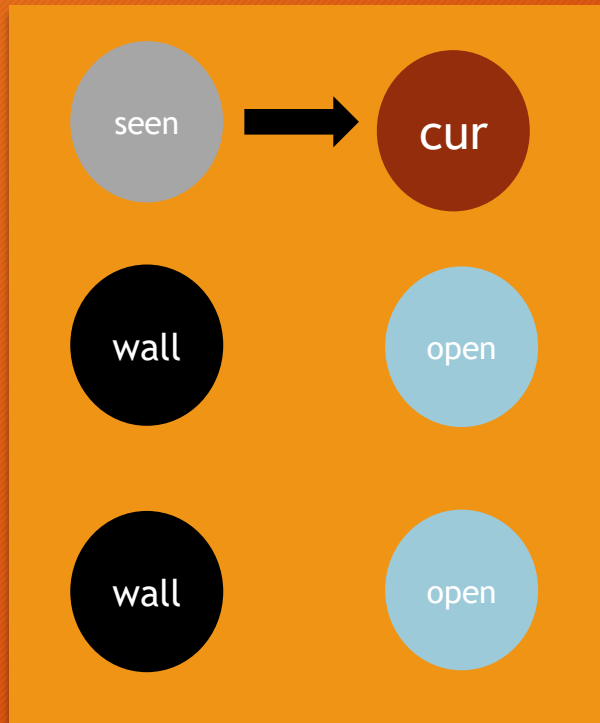
CURRENT_PATH

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.



PATHS

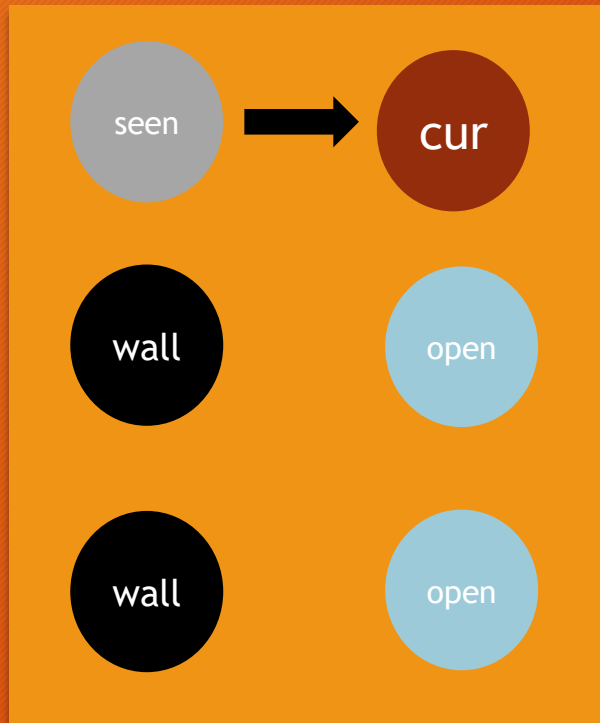
{ {0, 0}, {1,0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

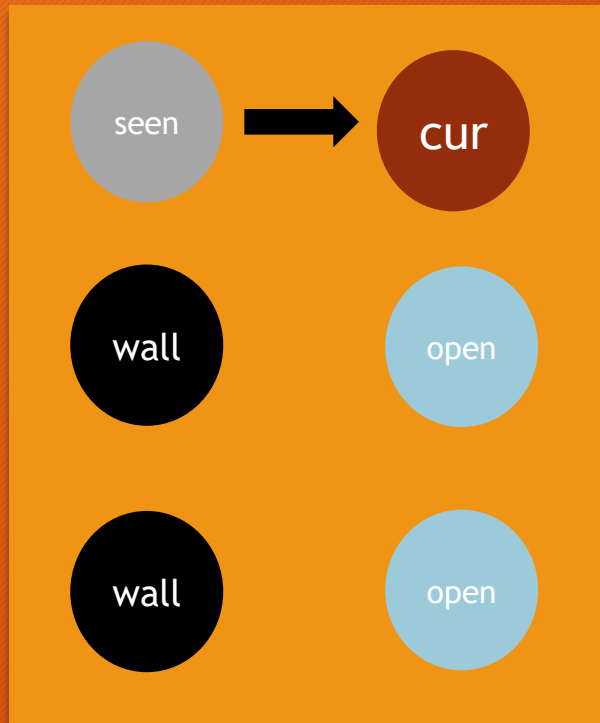
{ {0, 0}, {1,0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

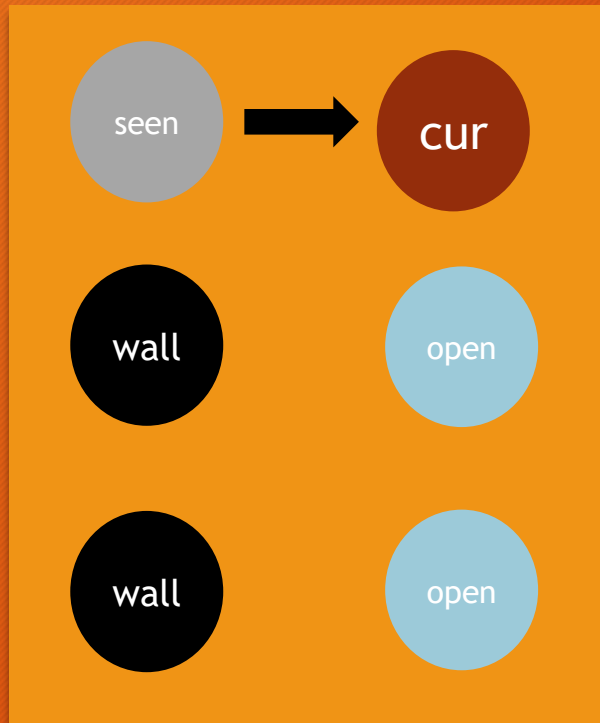
$\{1,0\} = \{1,2\}$
CURRENT_PATH_EXIT

$\{ \{0,0\}, \{1,0\} \}$
CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



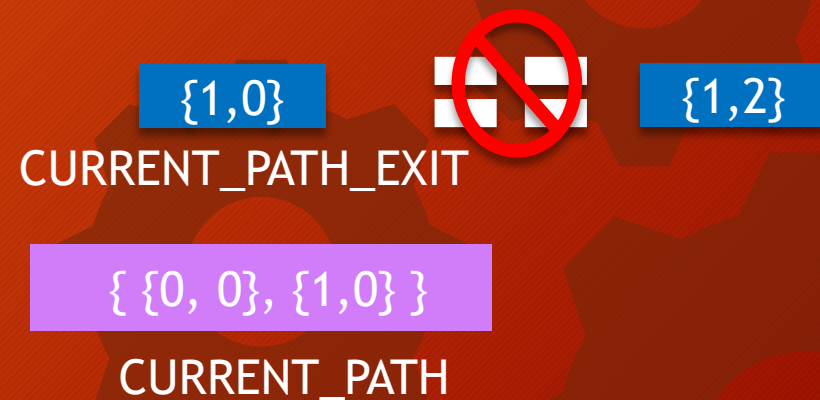
MAZE



PATHS

3. While there are still more paths to explore:

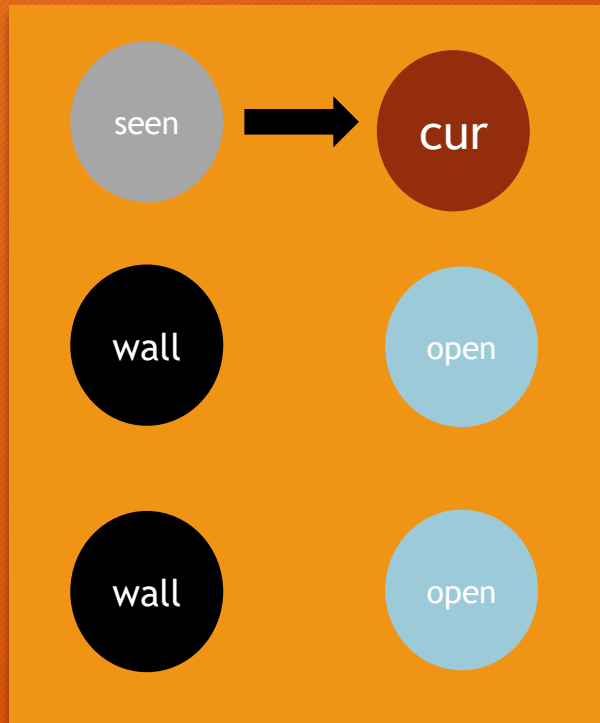
- Dequeue path from queue.
- If this path ends at exit, this path is the solution!



Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

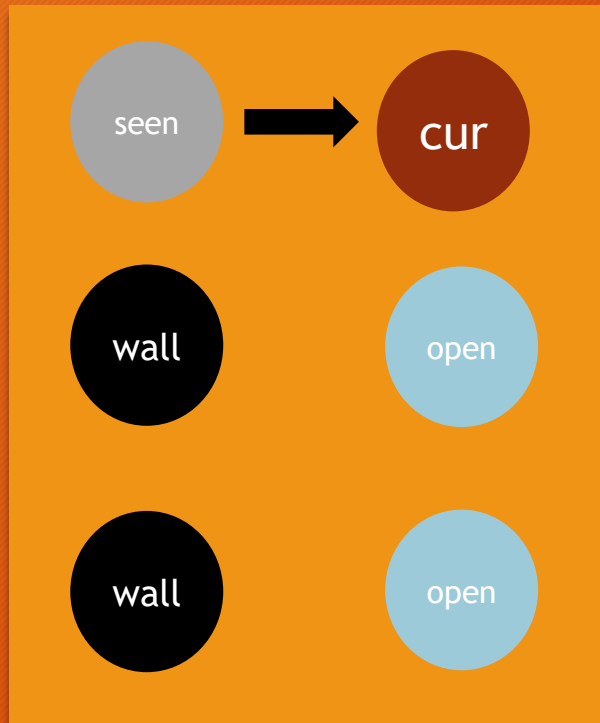
{ {0, 0}, {1,0} }

CURRENT_PATH

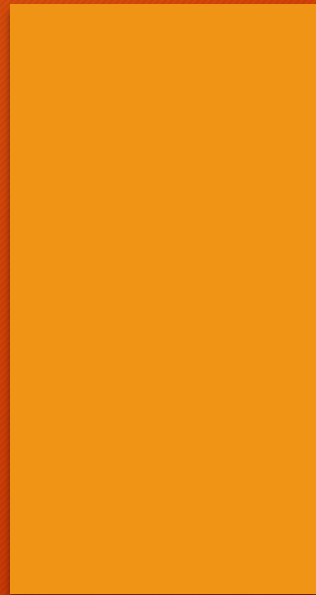
Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

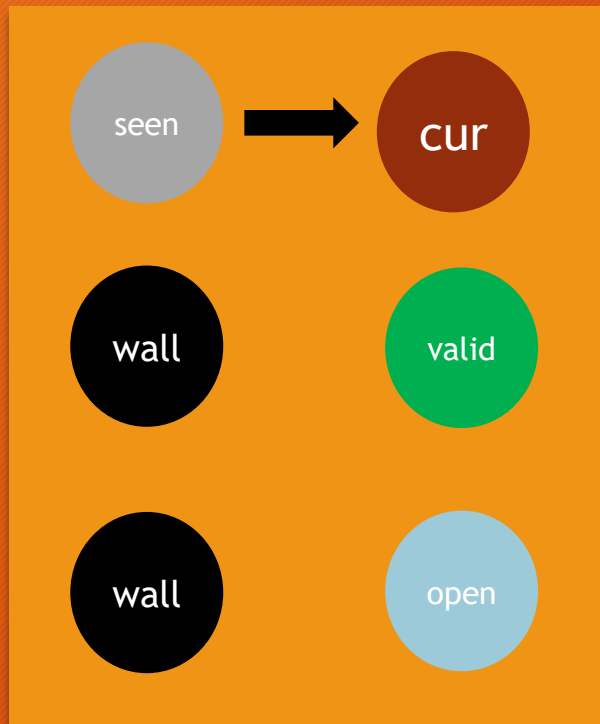
{ {0, 0}, {1,0} }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0} }

CURRENT_PATH

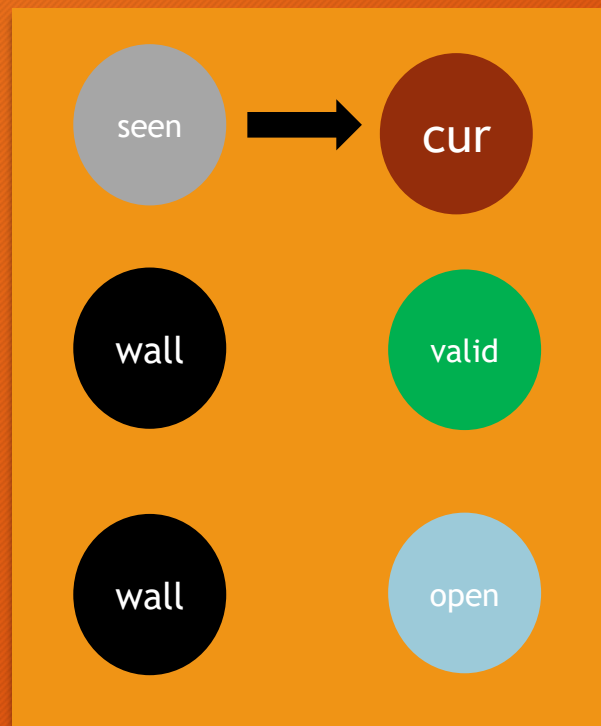
{ 1,1 }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0} }

CURRENT_PATH

{ {0, 0}, {1,0} }

CURRENT_PATH_CPY

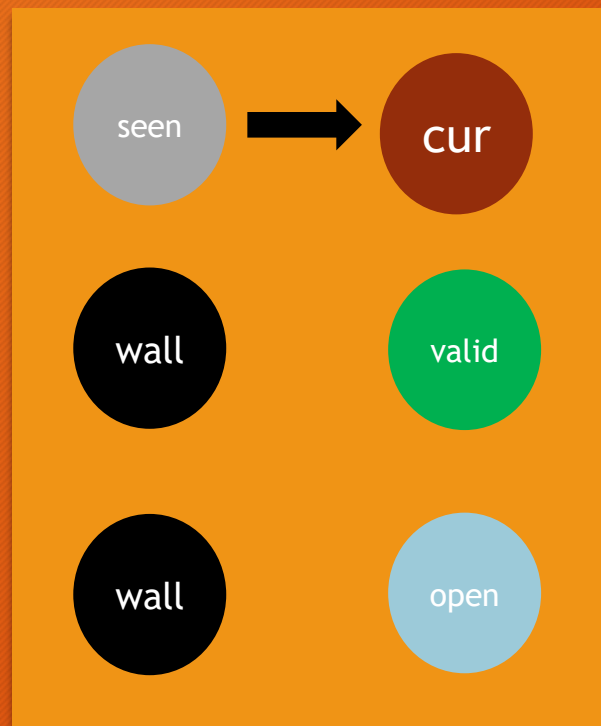
{ 1,1 }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



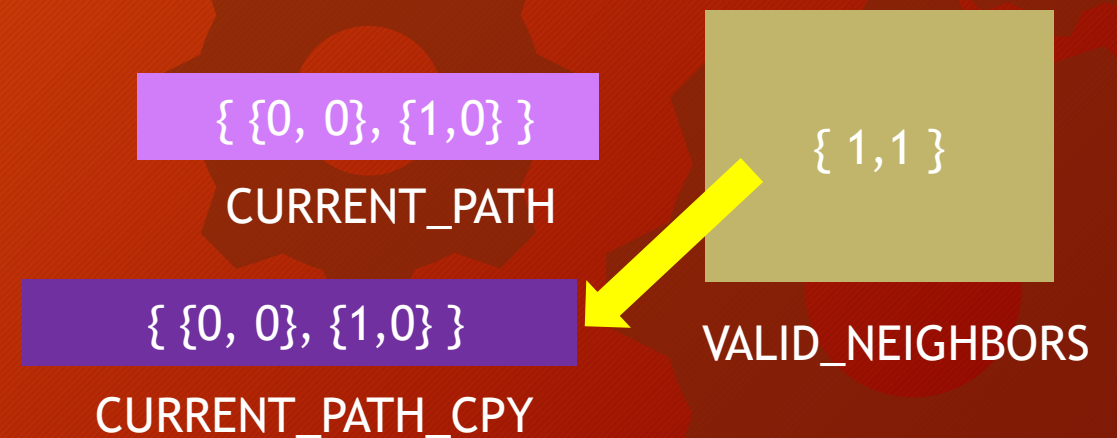
MAZE



PATHS

3. While there are still more paths to explore:

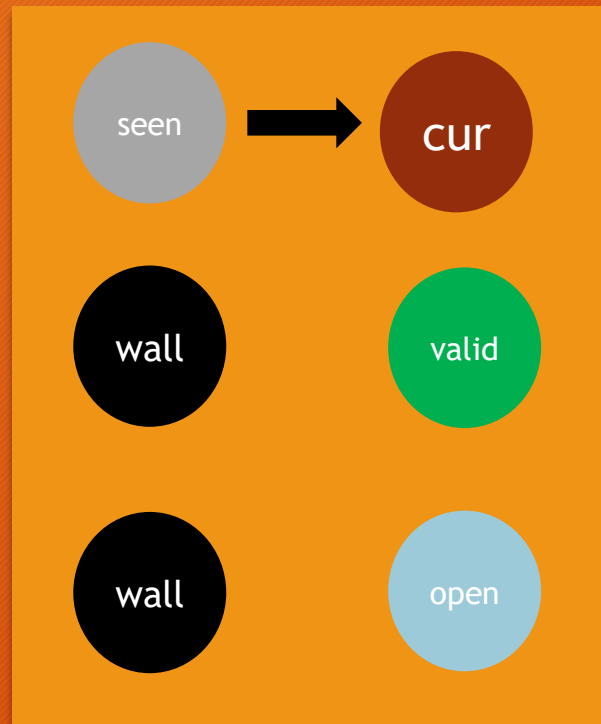
- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.



Part I: Maze

solveMaze()

- Let's walk thru a very small example:



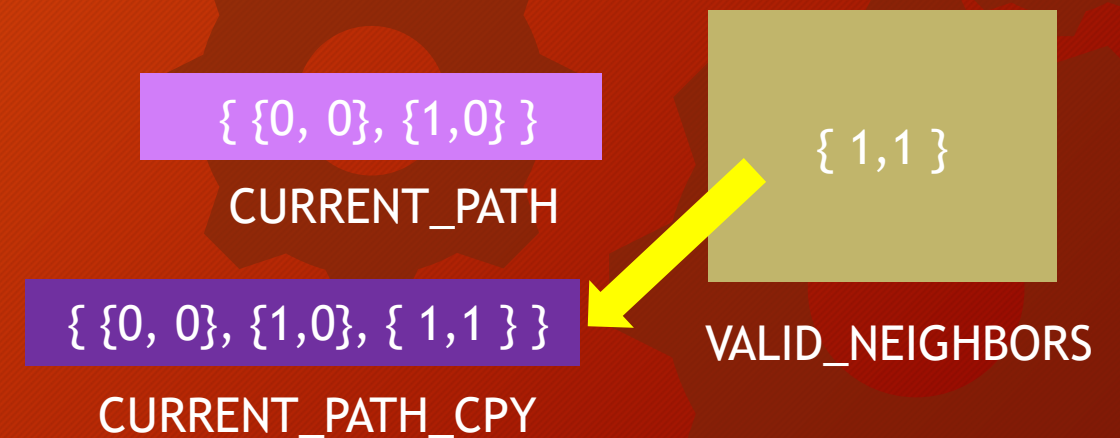
MAZE



PATHS

3. While there are still more paths to explore:

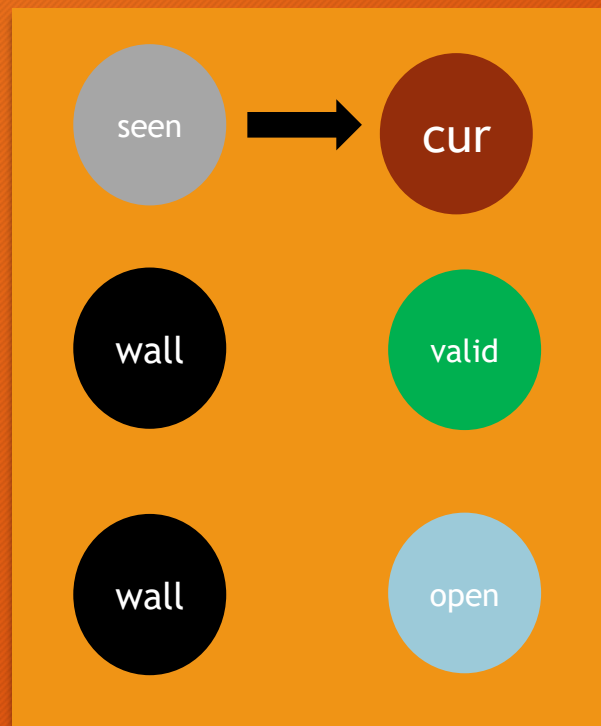
- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.



Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0} }

CURRENT_PATH

{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH_CPY

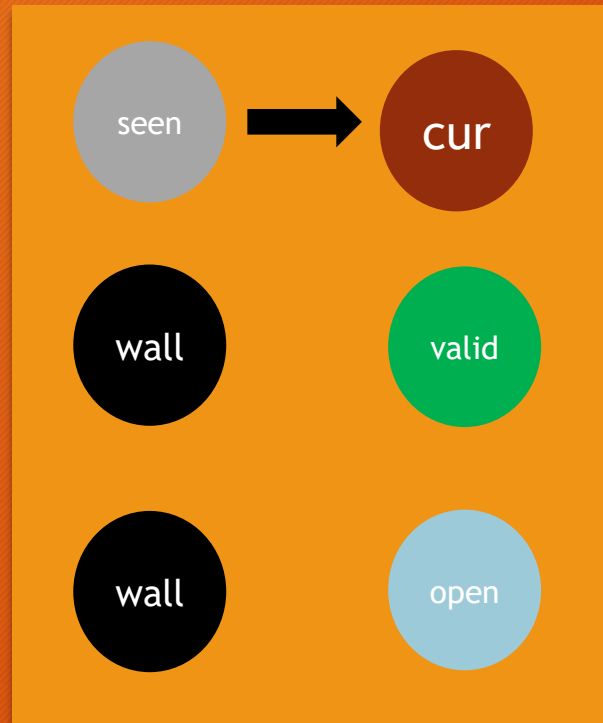
{ 1,1 }

VALID_NEIGHBORS

Part I: Maze

`solveMaze()`

- Let's walk thru a very small example:



MAZE

`{ {0, 0}, {1,0}, { 1,1 } }`



PATHS

Repeat!



CURRENT_PATH



CURRENT_PATH_CPY

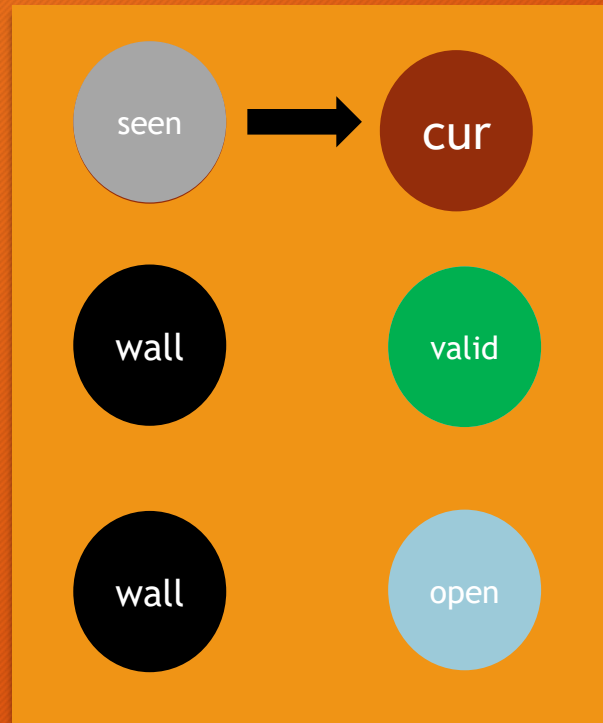


VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

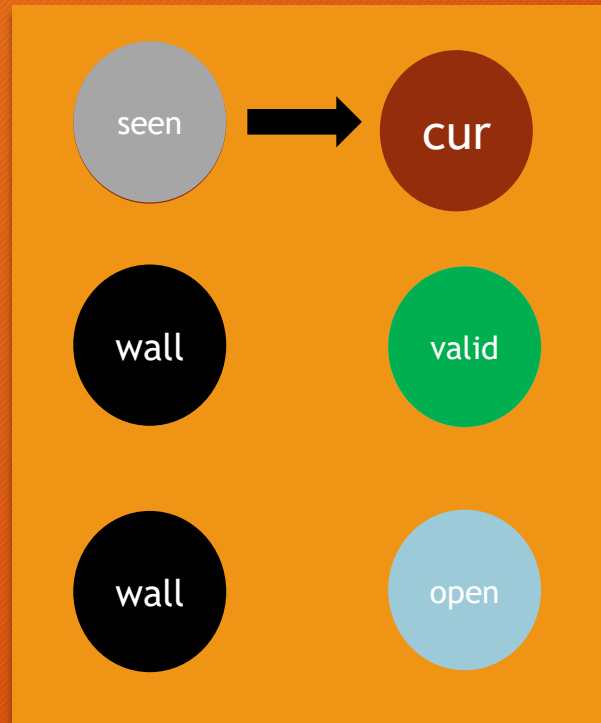
$\{ \{0, 0\}, \{1, 0\}, \{1, 1\} \}$

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

{ {0, 0}, {1,0}, { 1,1 } }

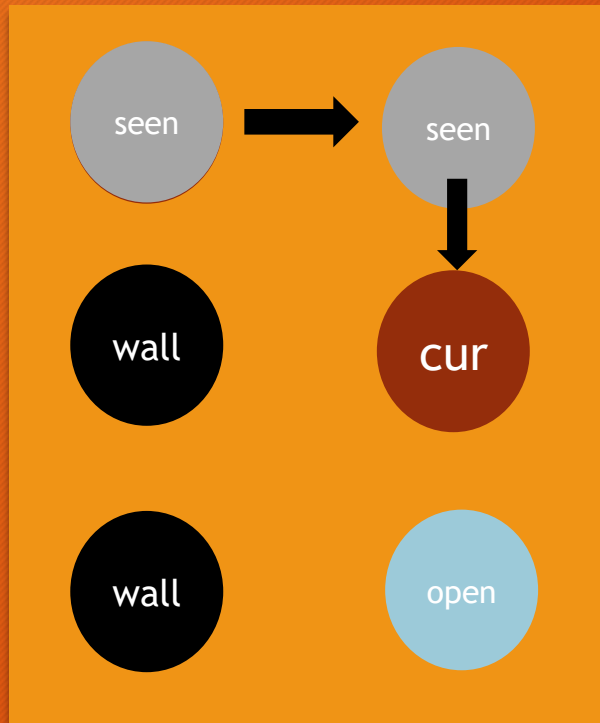
CURRENT_PATH

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

3. While there are still more paths to explore:

- Dequeue path from queue.

{ {0, 0}, {1,0}, { 1,1 } }

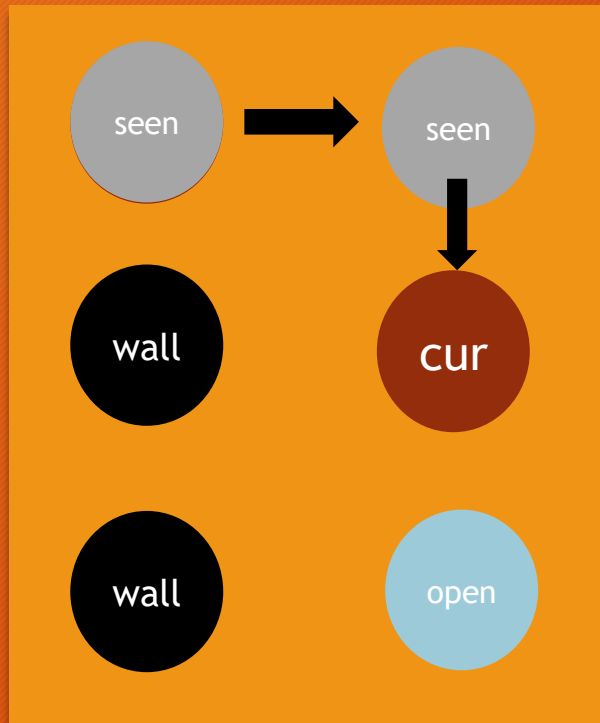
CURRENT_PATH

PATHS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

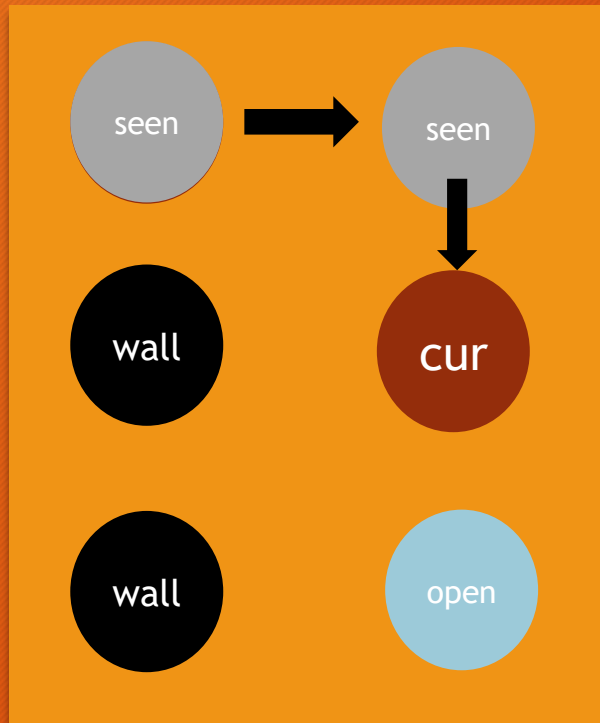
{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!

{ {0, 0}, {1,0}, { 1,1 } }

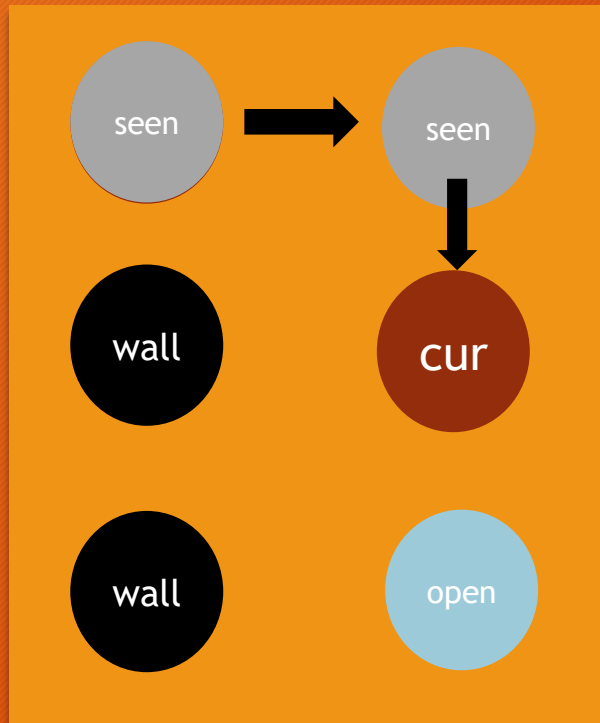
CURRENT_PATH

{1,1} !=
{1,2}, QED

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

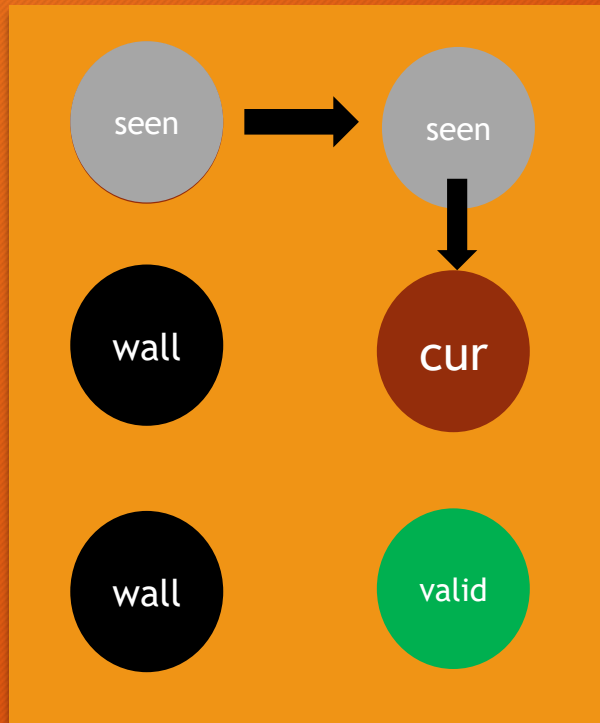
{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH

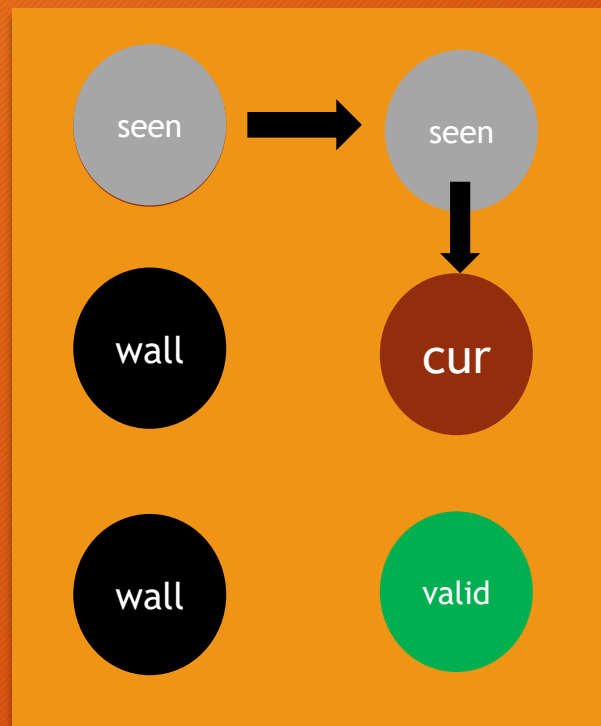
{ 1,2 }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH

{ {0, 0}, {1,0}, {1,1} }

CURRENT_PATH_CPY

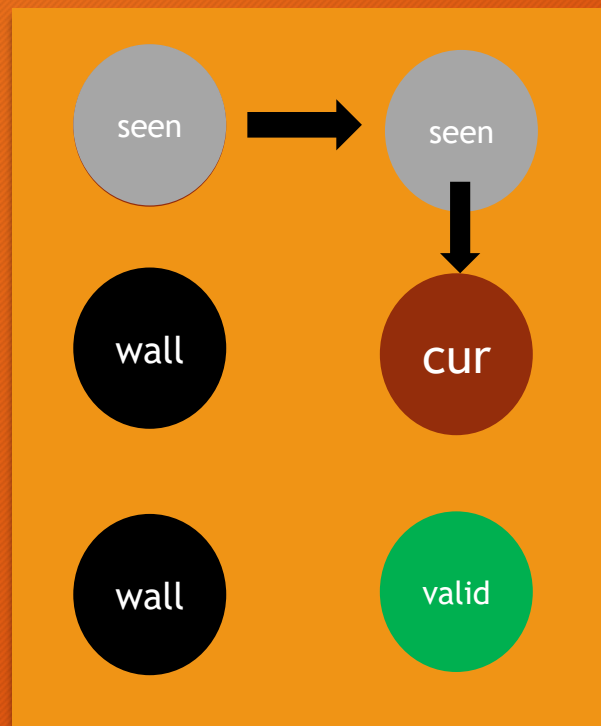
{ 1,2 }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



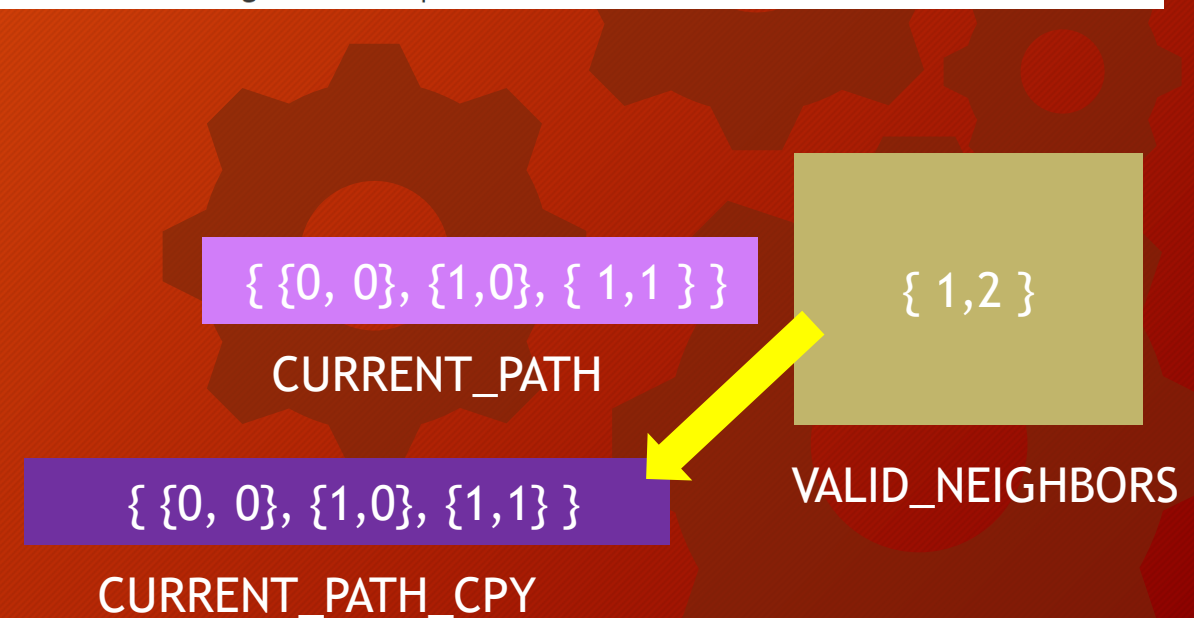
MAZE



PATHS

3. While there are still more paths to explore:

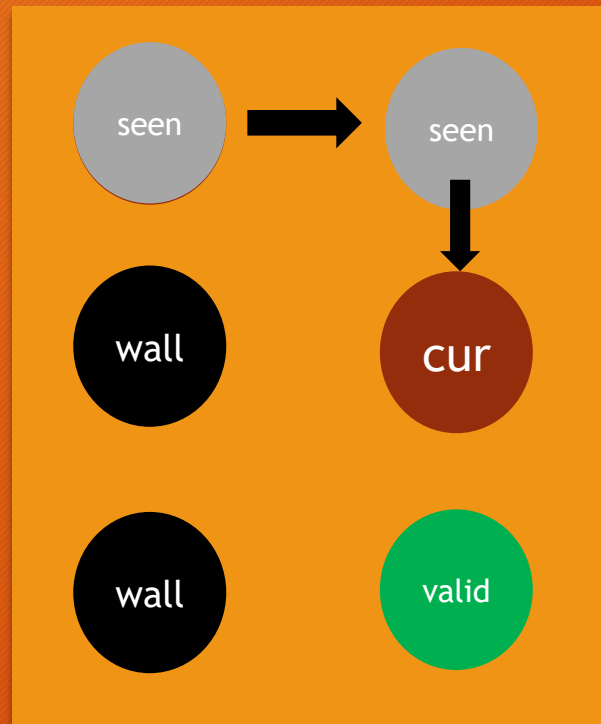
- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.



Part I: Maze

solveMaze()

- Let's walk thru a very small example:



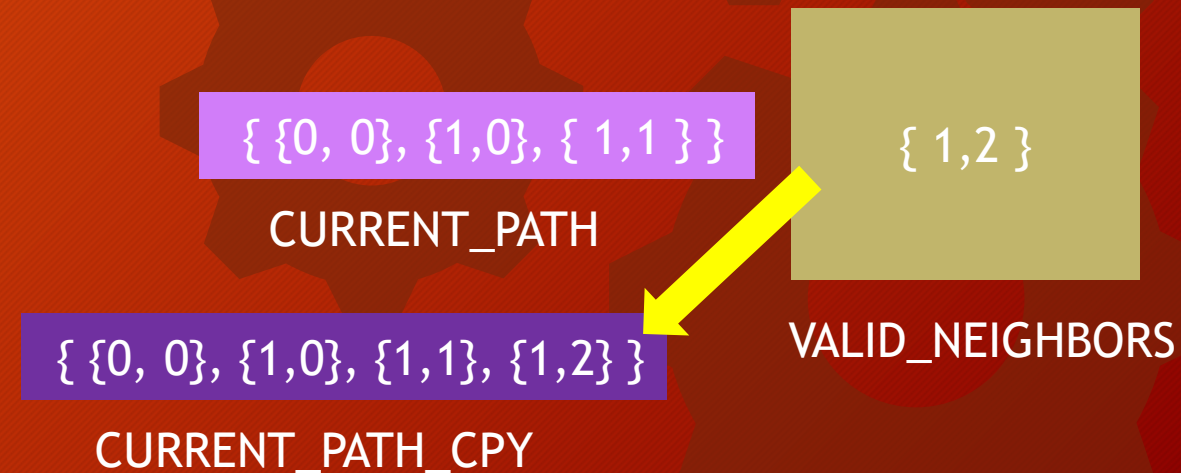
MAZE



PATHS

3. While there are still more paths to explore:

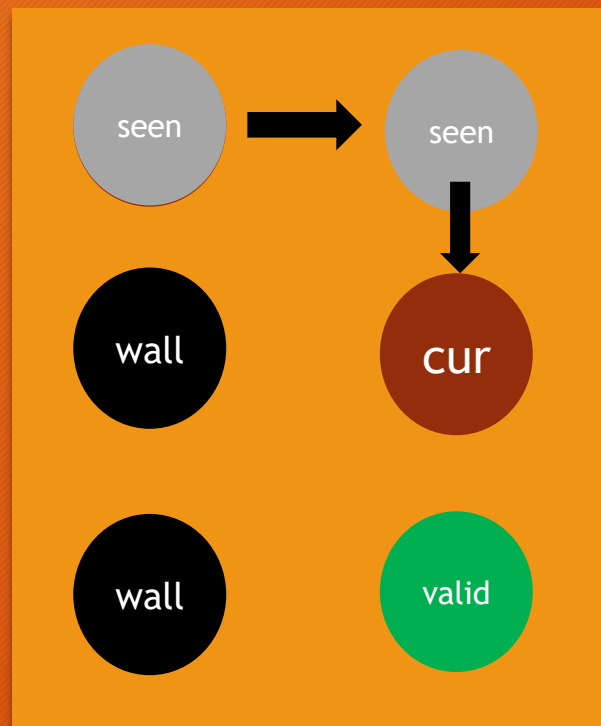
- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.



Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



PATHS

3. While there are still more paths to explore:

- Dequeue path from queue.
- If this path ends at exit, this path is the solution!
- If path does not end at exit:
 - For each viable neighbor of path end, make copy of path, extend by adding neighbor and enqueue it.

{ {0, 0}, {1,0}, { 1,1 } }

CURRENT_PATH

{ {0, 0}, {1,0}, {1,1}, {1,2} }

CURRENT_PATH_CPY

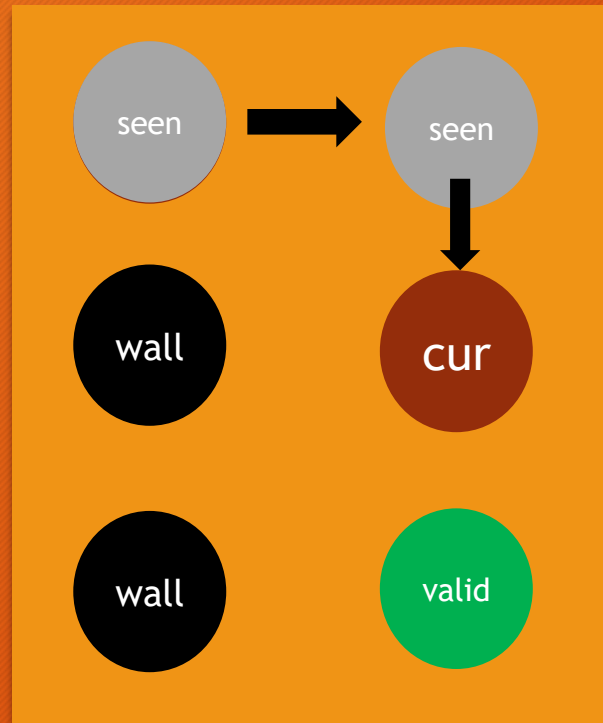
{ 1,2 }

VALID_NEIGHBORS

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE

{ {0, 0}, {1,0}, {1,1}, {1,2} }



PATHS

Repeat!

CURRENT_PATH

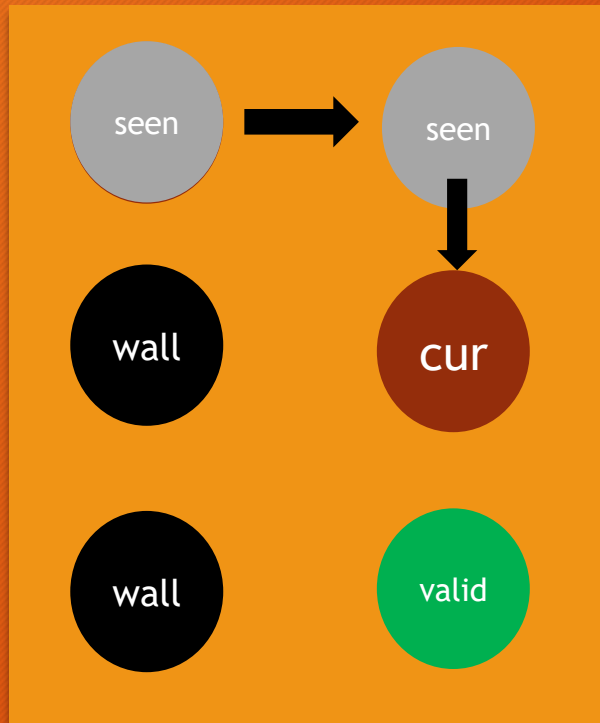
CURRENT_PATH_CPY

VALID_NEIGHBORS

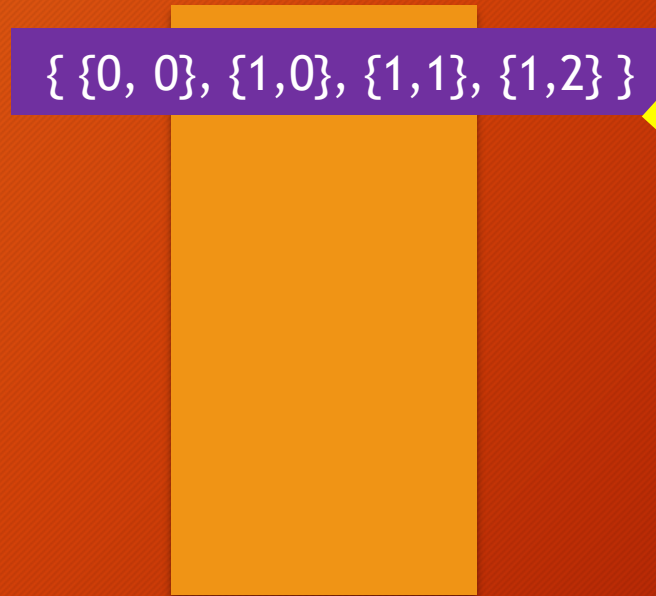
Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



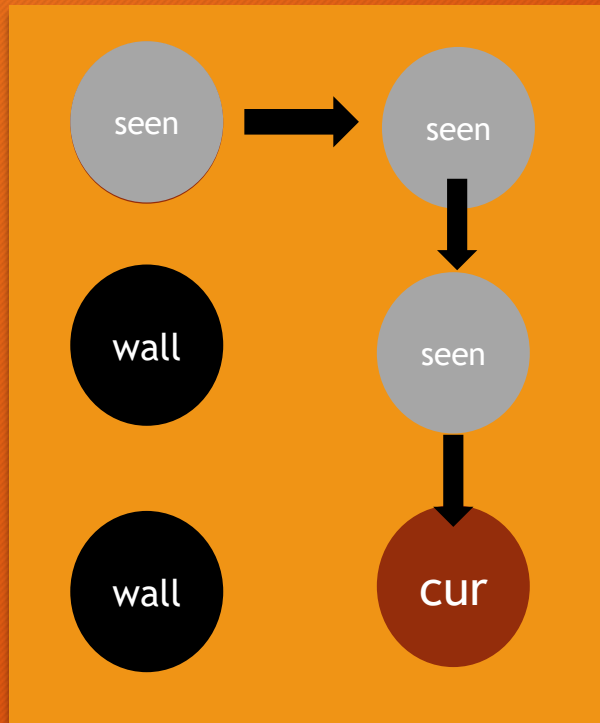
PATHS

CURRENT_PATH

Part I: Maze

solveMaze()

- Let's walk thru a very small example:



MAZE



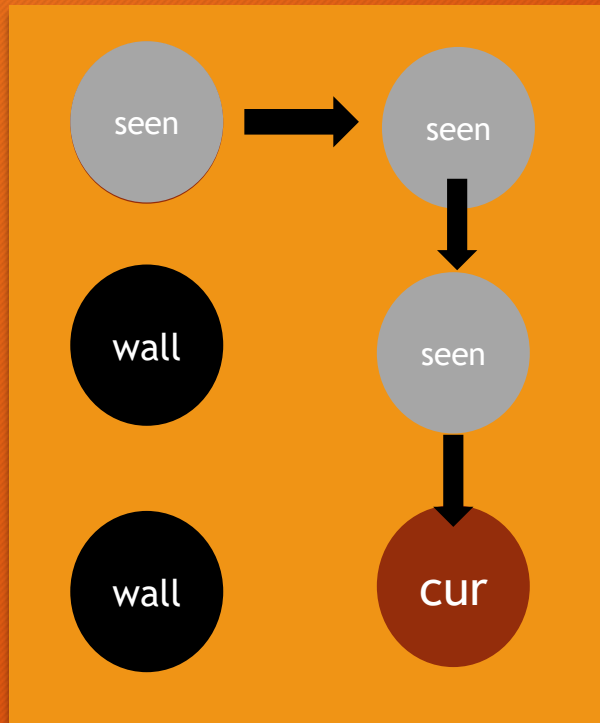
PATHS

{ {0, 0}, {1,0}, {1,1}, {1,2} }

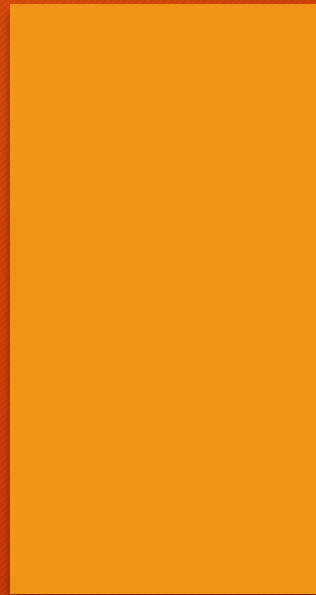
CURRENT_PATH

Part I: Maze solveMaze()

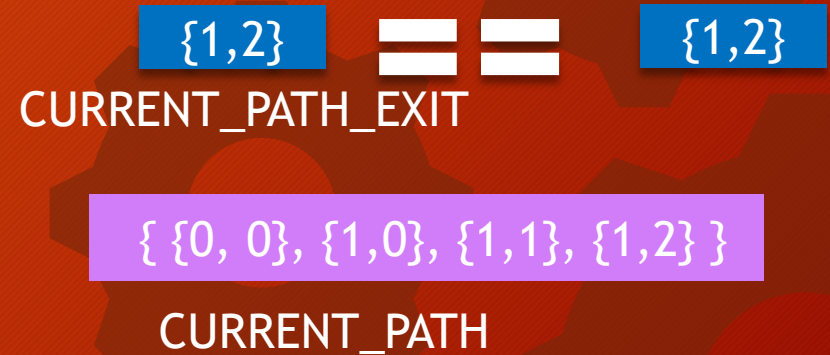
- Let's walk thru a very small example:



MAZE



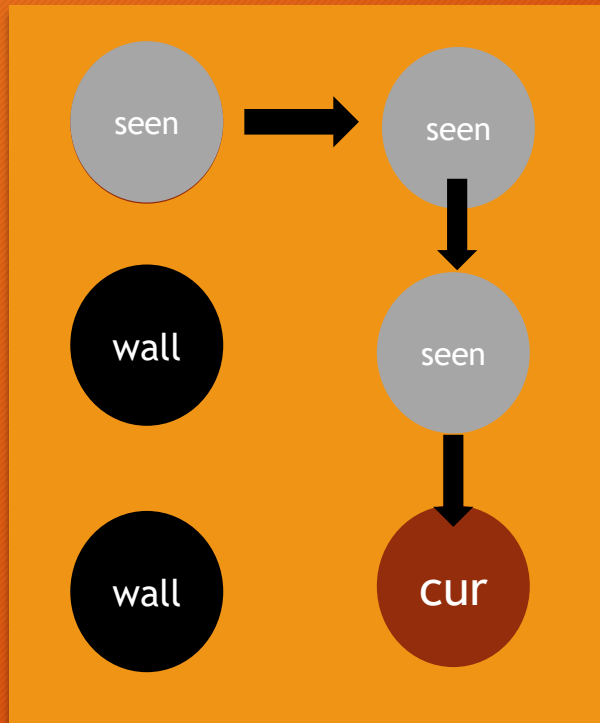
PATHS



Part I: Maze

solveMaze()

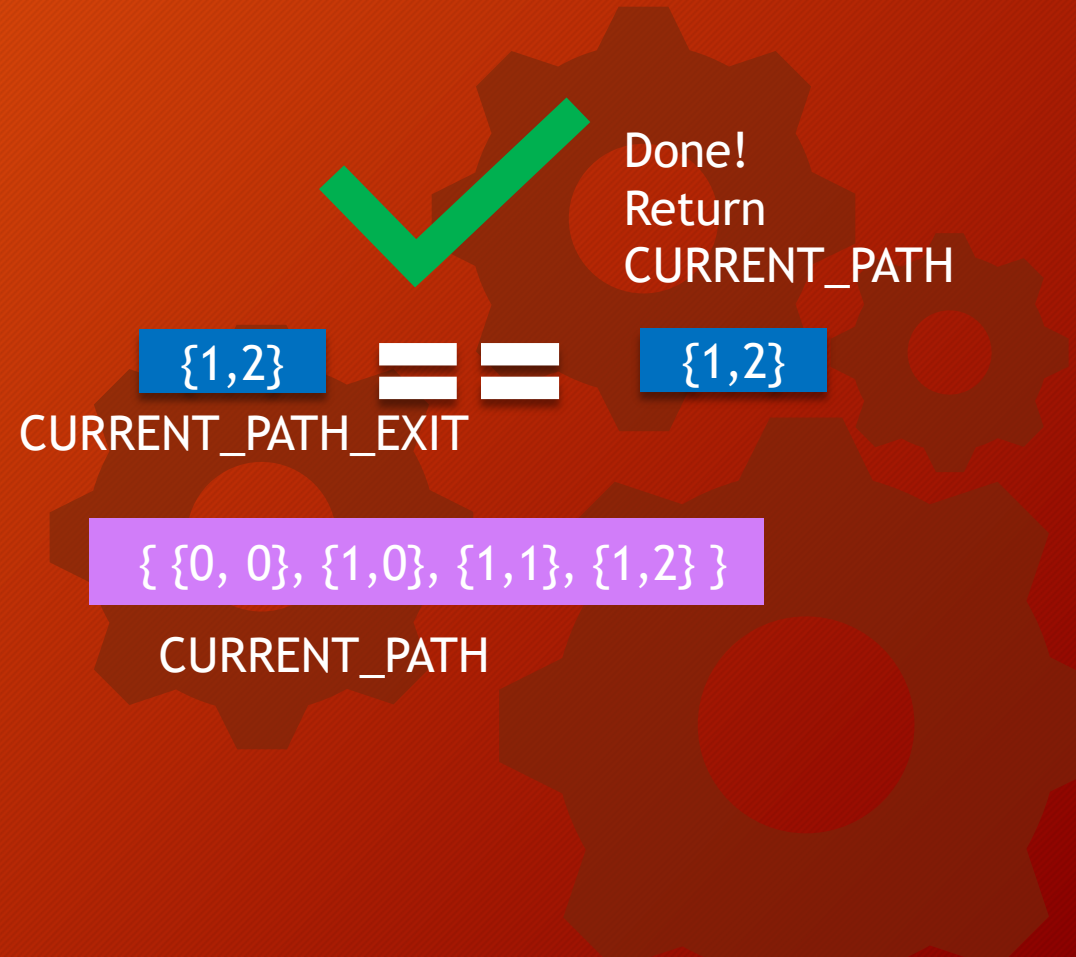
- Let's walk thru a very small example:



MAZE

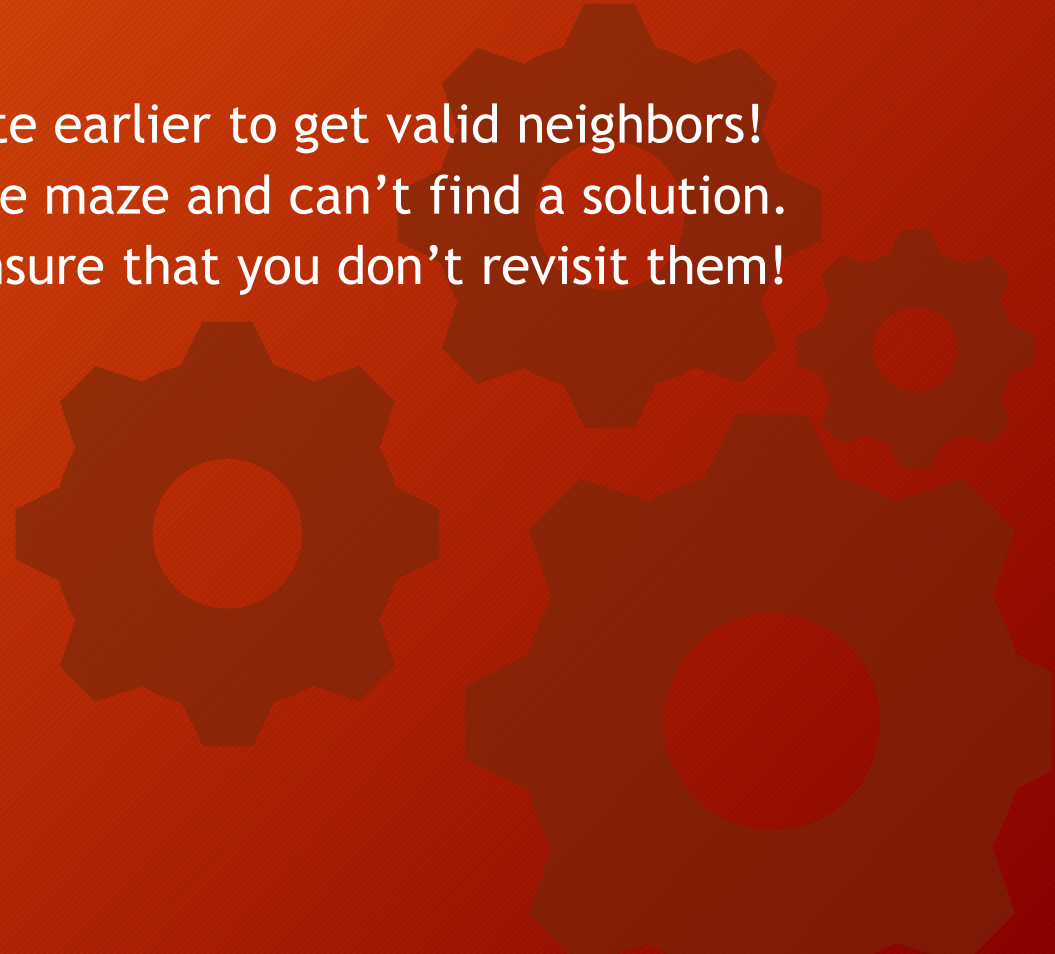


PATHS



Part I: Maze solveMaze()

- A few notes about this problem:
 - Once again, use your validMoves() helper you wrote earlier to get valid neighbors!
 - You can return an empty stack {} if you exhaust the maze and can't find a solution.
 - You need to keep track of **visited** locations and ensure that you don't revisit them! Else you'll enter a loop :/
 - Follow the provided algorithm closely!



Part I: Maze solveMaze()

- Once you get a working solution, you'll need to **add graphics** to the maze.
- We give you access to the `MazeGraphics` class to do this - all you need to do is call the function

```
MazeGraphics::highlightPath(Stack<GridLocation> path, string color)
```

- to highlight the current path that you're considering.
 - For more details about this documentation, look at `MazeGraphics.h` in your starter project!

Questions about solveMaze()?



Part I: Maze

readMazeFile()

- The last thing you'll have to do is pretty short - you need to ensure the robustness of the pre-written readMazeFile() function!

```
void readMazeFile(string filename, Grid<bool>& maze) {
```

- This function reads in a maze as a .txt file and populates MAZE with the corresponding maze format.
- You'll need to beef this function up to ensure that it doesn't have any bugs!

Part I: Maze

readMazeFile()

- There are 2 places where you'll need to do some checking:
 1. A maze row that is longer or shorter than the other rows.
 - Each maze row is required to have same length as all the others.
 2. A maze location containing a character that is not @ or - .
 - The valid options for a location are open or wall. Anything else is an error.
- I don't want to give anything more specific away here - check the source code in the project and you'll need to figure out *exactly* where these changes need to be made!
- You should raise an **error** if you find any infractions.

Part I: Maze

- Any questions about part 1? That's it!



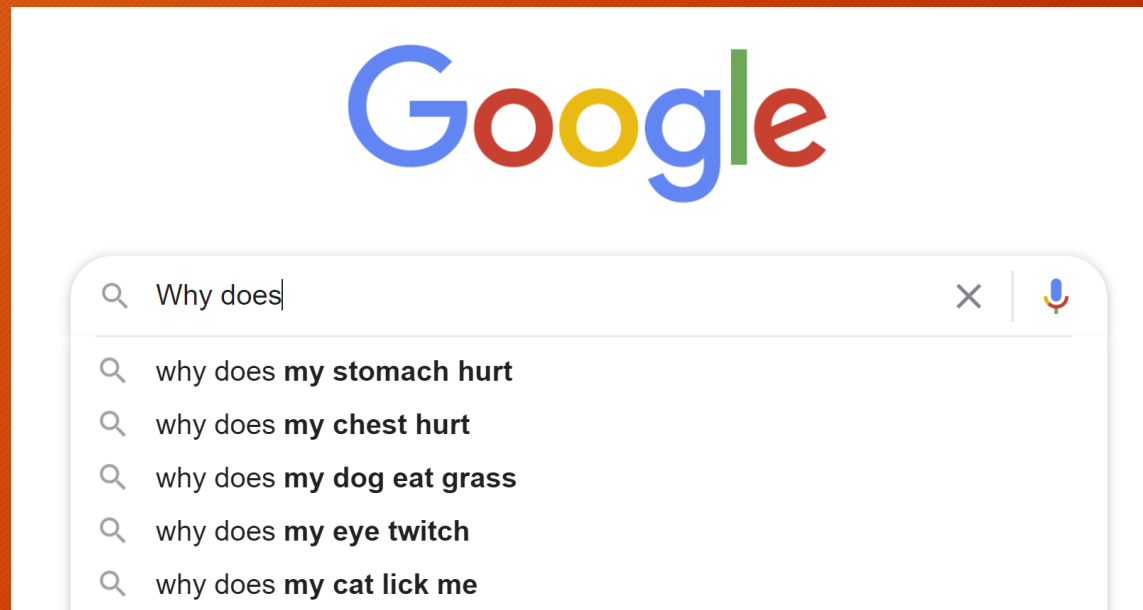
Agenda

- ~~Logistics~~
- ~~Part 1: Maze~~
- Part 2: Index



Part II: Search engine

- In this final part, you'll be putting your ADT skills to the test by creating a data structure that can power a **search engine**.
 - This is not an easy feat, so we'll go step by step!



Shows you something about the human condition, no?

An old A6 in CS106b was "MiniBrowser," where you implement page history, autocorrect and auto scrolling! It was a toughie :p

Part II: Search engine

- The first thing you'll need to do is implement the helper function

```
string cleanToken(string token)
```

that formats the provided TOKEN and returns the formatted version.

- Here are the steps to formatting a token:
 1. If TOKEN does not contain any letters at all, return empty string (“”);
 2. Trim all leading and trailing punctuation from TOKEN.
 - ,,.EGGS!; becomes EGGS
 3. Convert TOKEN to lowercase.
 - EGGS becomes eggs

Part II: Search engine

- Next, you'll need to do is read in the database of websites from a text file!
 - Namely, you'll be implementing this function:

```
Map<string, Set<string>> readDocs(string dbfile)
{
    Map<string, Set<string>> result;
    // TODO: your code here
    return result;
}
```

that takes in a database file name DBFILE and returns a Map<string, Set<string>.

Part II: Search engine readDocs()

- The database file that you're parsing will look something like this:
 - Basically, each collection of 2 lines is a **url-content** combo.
 - The first line of the pair will be a URL and the second will be the text content (all in one line) of the corresponding page.
- You will need to populate the map RESULT with pairs of URL to sets of unique words in the corresponding page.

db file

Pairs!



```
www.shoppinglist.com
EGGS! milk, fish,      @ bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Part II: Search engine readDocs()

- You'll notice a few things here:
 - First you'll see that the **tokens** (each individual word in the line of text) on the content lines are **formatted** in the map! You'll need to **format each token with cleanToken()** before it goes in the map!
 - Second, you might notice that the order of URL's isn't the same between the first and the second examples - this is out of your control, so you shouldn't worry about it 😊

Pairs!



db file

```
www.shoppinglist.com
EGGS! milk, fish,      @ bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Part II: Search engine readDocs()

- Let's talk about how to format a **line of content** from the db file:
 - URL's are easy - you don't need to modify them.
 - **Lines of page content are more complex:**
 1. Tokenize the line into a collection of strings, separated by whitespace (**stringSplit** will be very helpful here), and call **cleanToken()** on each token. Store the result in a **Set<string>** that will be the value corresponding to a url!

Pairs!



db file

```
www.shoppinglist.com
EGGS! milk, fish,      @ bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Part II: Search engine readDocs()

- A few tips about creating this map:
 - You're going to need to store an entry in the map **once every two lines** (it takes 2 lines to get a single key-value pair) - can you manipulate a loop to help you do this?
 - readEntireFile() in filelib.h is an easy way to read a file into a vector!

db file

Pairs!



```
www.shoppinglist.com
EGGS! milk, fish,      @ bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Part II: Search engine readDocs()

- To discard non-words, `isalpha()` will be helpful, and to trim punctuation, `ispunct()` will help.
 - Trimming punctuation can be difficult so I'd recommend handling leading punctuation and trailing punctuation in separate steps.
 1. Until your first letter is not punctuation, remove the first letter
 2. Until your last letter is not punctuation, remove the last letter

Pairs!



db file

```
www.shoppinglist.com
EGGS! milk, fish,      @  bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Part II: Search engine readDocs()

- Some final notes about the problem:
 - “Strlib.h” may be helpful here for little string tips
 - To convert a string to lowercase, use `toLowerCase()`, but be aware that **it returns a new string** instead of modifying the existing one!
 - Because the value in `RESULT` is a `Set<string>`, you can treat `RESULT[somekey]` as a `Set<string>` and do things like:
 - `RESULT[somekey] += “hello”`

db file

Pairs!



```
www.shoppinglist.com
EGGS! milk, fish,      @ bread cheese
www.rainbow.org
red ~green~ orange yellow blue indigo violet
www.dr.seuss.net
One Fish Two Fish Red fish Blue fish !!!
www.bigbadwolf.com
I'm not trying to eat you
```

```
{
  "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
  "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
  "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
  "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

Questions about readDocs?



Part II: Search engine buildIndex()

- It's now time for you to build an **inverted index**!
 - This function takes in the structure you created in **readDocs()** and returns a new Map that pairs a **unique word** to a **set of URLs** at which that word can be found.
 - Remember, the map in **readDocs()** paired **url's** to **unique words in the website**, and this one pairs **unique words** to **sets of urls** in which that word appears!

```
Map<string, Set<string>> buildIndex(Map<string, Set<string>>& docs)
```

Small sample inverted index:

```
{  
  "seach" : { "google.com, bing.com" }  
  "login" : { "webkinz.com" }  
}
```

Part II: Search engine buildIndex()

- It's now time for you to build an **inverted index**!
 - This function takes in the structure you created in **readDocs()** and returns a new Map that pairs a **unique word** to a **set of URLs** at which that word can be found.
 - Remember, the map in **readDocs()** paired **url's** to **unique words in the website**, and this one pairs **unique words** to **sets of urls** in which that word appears!

```
Map<string, Set<string>> buildIndex(Map<string, Set<string>>& docs)
```

Small sample inverted index:

```
{  
  "seach" : { "google.com, bing.com" }  
  "login" : { "webkinz.com" }  
}
```

Questions?

Part II: Search engine findQueryMatches()

- In this next part, you're actually going to be servicing a user query, meaning that you'll take in a search request and return a result!

```
Set<string> findQueryMatches(Map<string, Set<string>>& index, string query)
```

- Given a specific **query** string and an **inverted index**, mapping **unique words** to **sets of urls** in which they appear, you need to return a set of url's that satisfy the **query**.
- But what does a query look like?

Part II: Search engine

findQueryMatches()

- Here's an example query:
 - flake
 - Handling this request is easy! You'd just return a list of urls in which the above word appears (in this case, www.Stanford.edu).
 - Conveniently, the given parameter INDEX does this for you - it maps a string to a set of urls in which that string appears!
 - Return index[query]
 - Let's try a more complex one...
- 

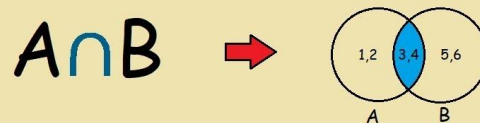
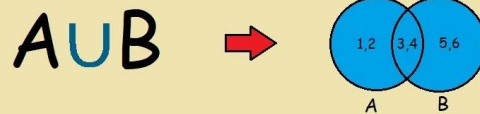
Part II: Search engine findQueryMatches()

- Here's an example query:
 - Fugu; +fish
- Here we have TWO tokens! You can isolate them with `stringSplit()`.
- Notice that the second string contains (+)! The (+) sign is a special operator in your query - it performs a set **intersection** between the left and right sets.
 - An **intersection** means the `Set<string>` returned should contain only the urls that contain both "fugu" AND "fish"
- **VERY important note:** notice that the **same punctuation stripping and lowercase** rules apply to tokens in your query string
- You'll need to keep an eye out for the (+) operator; it'll be the first character of a query token - your string cleaning routine will attempt to remove it, so keep that in mind!

Part II: Search engine findQueryMatches()

- Here's another query:
 - Cat Dog
- Like the last example, you'll `stringSplit()` the line into two tokens. Because there's no operator here, you'll use set **union**, meaning the set you return should be a set of urls that **either** contain "Cat" **OR** "Dog".

Union and Intersection

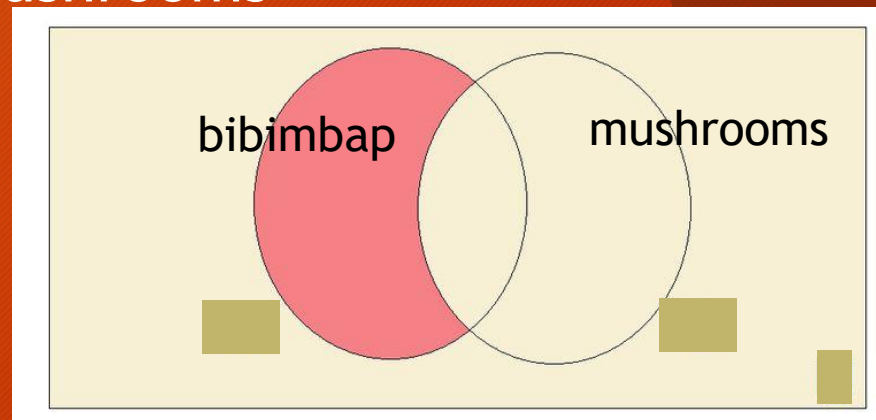


Union (no operator) - return all url's that contain at least 1 of the words!

Intersection (+ operator) - return all url's tht contain BOTH tokens (small subset!)

Part II: Search engine findQueryMatches()

- Here's another example:
 - Bibimbap -mushrooms
- Here we've introduced another example with the (-) operator. This operator performs set **subtraction**, meaning, the resulting set should contain all urls that contain "bibimbap" that DO NOT contain "mushrooms"



Part II: Search engine

findQueryMatches()

- Here's one last complex example:
 - CS +106B -RECURSION!!! fun
- Here we have multiple tokens in our query string. We'll first need to split them and clean them to get
 - { "cs", "106b", "recursion", "fun" }
 - You'll of course need to remember the placement of the (+ and -) operators, but I just wanted to show you the string cleaning here.
- From there, you will process the query **from left to right**.
 - ((cs +106b) -recursion) fun
 - or
 - (("cs" intersection "106b") subtracted from "recursion") union "fun"

Part II: Search engine

findQueryMatches()

- Some tips for findQueryMatches()
 - First and foremost, be sure to break this problem down into actionable pieces! Please use helper functions to solve this problem.
 - I would recommend reusing your cleanToken() helper function!
 - The tricky set operation stuff isn't actually as spooky as it might initially seem - check out [this set documentation](#) and bask in the operators we've provided you 😊 (you'll probably want +, *, and -)

set1 + set2	O(N)	Returns the union of sets set1 and set2 , which is the set of elements that appear in at least one of the two sets.
set + value	O(N)	Returns the union of set set1 and individual value value .
set1 += set2;	O(N)	Adds all of the elements from set2 (or the single specified value) to set1 .
set += value;	O(log N)	Adds the single specified value to the set.
set1 - set2	O(N)	Returns the difference of sets set1 and set2 , which is all of the elements that appear in set1 but not set2 .
set - value	O(N)	Returns the set set with value removed.
set1 -= set2;	O(N)	Removes the elements from set2 (or the single specified value) from set1 .
set -= value;	O(log N)	Removes the single specified value from the set.
set1 * set2	O(N)	Returns the intersection of sets set1 and set2 , which is the set of all elements that appear in both.
set1 *= set2;	O(N)	Removes any elements from set1 that are not present in set2 .

Part II: Search engine findQueryMatches()

- Some more tips for findQueryMatches()
 - You can always assume that a (+ or -) operator, if extant, will appear as the first character of a query token. You **cannot** assume that the characters following are non-punctuation characters
 - Ex. { Apple +windows -!!linux }
 - Remember (because this causes much grief later) that your query needs to be **case insensitive**. “Apple” and “apple” should be looked up the same (“apple”)!

Questions about findQueryMatches?

- This part is pretty complex, remember to decompose!



Part II: Search engine building searchEngine()

- It's finally time for you to put your querying skills to the test - you're going to write a function that serves as a search engine!
- You'll be implementing the following function:

```
void searchEngine(string dbfile)
```

Part II: Search engine building searchEngine()

- You will need to read in the provided DBFILE and convert it into an inverted index.
 - You've already written the functions to do this 😊
- You'll then need to display to the user how many URL's were processed to build the index and how many distinct words were found in all of the files
 - Think about your data structures - is this data stored anywhere?

```
void searchEngine(string dbfile)
```

Part II: Search engine building searchEngine()

- You'll then need to repeatedly prompt the user for search queries like the ones discussed earlier.
 - You will find the appropriate `Set<string>` result for that query and print it.
- Repeat until the user enters “”, and then exit the program.
- Your goal is to match this functionality exactly!

```
Stand by while building index...
```

```
Indexed 50 pages containing 5595 unique terms.
```

```
Enter query sentence (RETURN/ENTER to quit): llama
```

```
Found 1 matching pages
```

```
{"http://cs106b.stanford.edu/assignments/assign2/searchengine.html"}
```

```
Enter query sentence (RETURN/ENTER to quit): suitable +kits
```

```
Found 2 matching pages
```

```
{"http://cs106b.stanford.edu/assignments/assign2/searchengine.html", "http://cs106b.stanford.edu/qt/troubleshooting.html"}
```

```
Enter query sentence (RETURN/ENTER to quit): Mac linux -windows
```

```
Found 3 matching pages
```

```
{"http://cs106b.stanford.edu/lectures/sets-maps/qa.html", "http://cs106b.stanford.edu/qt/install-linux.html", "http://cs106b.stanford.edu/qt/install-mac.html"}
```

```
Enter query sentence (RETURN/ENTER to quit): as-is wow!
```

```
Found 3 matching pages
```

```
{"http://cs106b.stanford.edu/about_assignments", "http://cs106b.stanford.edu/assignments/assign1/soundex.html", "http://cs106b.stanford.edu/assignments/assign2/searchengine.html"}
```

```
Enter query sentence (RETURN/ENTER to quit):
```

```
All done!
```

Part II: Search engine building searchEngine()

- Some notes about searchEngine()
 - You shouldn't have to write a ton of new code here - virtually all of the lifting has already been done by the other functions in the program - you're just bringing them together now!
 - If you've written some great tests for your helper functions, this part should just work! If you encounter a bug, try to isolate it to a particular function by using the debugger!

Stand by while building index...

Indexed 50 pages containing 5595 unique terms.

Enter query sentence (RETURN/ENTER to quit): llama

Found 1 matching pages

{"http://cs106b.stanford.edu/assignments/assign2/searchengine.html"}

Enter query sentence (RETURN/ENTER to quit): suitable +kits

Found 2 matching pages

{"http://cs106b.stanford.edu/assignments/assign2/searchengine.html", "http://cs106b.stanford.edu/qt/troubleshooting.html"}

Enter query sentence (RETURN/ENTER to quit): Mac linux -windows

Found 3 matching pages

{"http://cs106b.stanford.edu/lectures/sets-maps/qa.html", "http://cs106b.stanford.edu/qt/install-linux.html", "http://cs106b.stanford.edu/qt/install-mac.html"}

Enter query sentence (RETURN/ENTER to quit): as-is wow!

Found 3 matching pages

{"http://cs106b.stanford.edu/about_assignments", "http://cs106b.stanford.edu/assignments/assign1/soundex.html", "http://cs106b.stanford.edu/assignments/assign2/searchengine.html"}

Enter query sentence (RETURN/ENTER to quit):

All done!

Any questions about Part II?



That's it!

Congrats!



Nice job!



Great work!



You did it!



Stack Efron, 106B alum,
congratulating you on a
job well done!