YEAH A2

Fun with Collections!
Agenda

• Logistics
• Part 1: Maze
• Part 2: Index
Logistics

• Due: Friday, October 2nd (11:59PM PDT)
• Grace period: Sunday October 4th (11:59PM PDT)
• Must be completed individually!
Questions about logistics?
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!

- This first part is simply a test to verify that you're able to visualize ADT's in the debugger!

- You should be able to run the warmup program in the debugger and see a view in the top right like this when looking at ADT's. If not, check out the website for details on how to patch this!

- 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!

```java
def 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

• In the top right corner of QT, 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!

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[statics]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>&lt;5 items&gt;</td>
<td>Queue&lt;int&gt; &amp; int</td>
</tr>
<tr>
<td>front</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>back</td>
<td>5</td>
<td>int</td>
</tr>
<tr>
<td>s</td>
<td>&lt;0 items&gt;</td>
<td>Stack&lt;int&gt;</td>
</tr>
<tr>
<td>val</td>
<td>28672</td>
<td>int</td>
</tr>
</tbody>
</table>
Warmup 2: Debugging an ADT question

• For this next part, you’re going to use your new debugging skills to debug this function:

```cpp
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
        }
    }
}
```

• Step through the function for a few iterations and determine where things go wrong!
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.

```cpp
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.

```c++
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

```c++
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 \}
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

```c++
void readSolutionFile(string filename, Stack<GridLocation>& soln) {
    // Code here...
}
```

{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 (N, S, E, W) directions that are non-walls and in bounds!
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!
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!

```cpp
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.
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!
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.

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

```javascript
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. `{ maze.numRows() - 1, 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:

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

  that verifies that PATH is 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()

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

```c++
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()

• To programmatically generate a solution to a given maze, you’ll need to use a Breadth-First Search (BFS) algorithm.

• This is the first complex algorithm you'll need to write up in this class! Luckily, it's not too bad, and we'll go over it together.

  • 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:
Part I: Maze

solveMaze()

• Let’s walk thru a very small example:

1. 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:

1. 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:

1. Create a length-one path containing just the entry location. Enqueue that path.
Part I: Maze
solveMaze()

Let’s walk thru a very small example:

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

```python
{ {0, 0} }
```

MAZE

PATHS
Part I: Maze
solveMaze()

- Let’s walk thru a very small example:

MAZE

PATHS

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

\{ \{0, 0\} \}
Part I: Maze solveMaze()

Let’s walk thru a very small example:

1. Start at the current position: \{0, 0\}

2. Explore paths:
   - Dequeue path from queue.
   - CURRENT_PATH:

3. While there are still more paths to explore:
   - Dequeue path from queue.

MAZE:
- cur
- wall
- wall

PATHS:
- \{0, 0\}
Part I: Maze

solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ◦ Dequeue path from queue.
   ◦ If this path ends at exit, this path is the solution!

```javascript
{{0, 0}}
```

CURRENT_PATH
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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:

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:

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:

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:

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:

```
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.
```

- MAZE
  - cur
  - wall
  - wall

- PATHS
  - \{ \{0, 0\} \}

- CURRENT_PATH
  - \{ \{1, 0\} \}

- CURRENT_PATH_CPY
  - \{ \{0, 0\} \}

- VALID_NEIGHBORS
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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:

  1. MAZE: { {0, 0} }  
  2. CURRENT_PATH: { {1, 0} }  
  3. VALID_NEIGHBORS: { {0, 0}, {1,0} }  
  4. CURRENT_PATH_CPY: { {0, 0}, {1,0} }  
  5. PATHS: { {0, 0}, {1,0} }  

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:

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

Repeat!
Part I: Maze

solveMaze()

Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.

{ {0, 0}, {1,0} }
Part I: Maze solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.

{ {0, 0}, {1,0} }
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.

```
{ {0, 0}, {1,0} }
```
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.

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

CURRENT_PATH
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.
   ○ If this path ends at exit, this path is the solution!

current_PATH = \{ \{0, 0\}, \{1,0\} \}
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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:

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

- seen
- cur
- wall
- open

PATHS

- current_path

[ {0, 0}, {1, 0} ]

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:

```
let's walk thru a very small example:

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.

paths

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

current_path
```
Part I: Maze solveMaze()

• Let’s walk thru a very small example:

  ![Diagram showing the maze, current path, and valid neighbors](image)

  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:

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:

```plaintext
MAZE
{ {0, 0}, {1,0} }

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

VALID_NEIGHBORS
{ 1,1 }
```

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

- seen
- cur
- valid
- open

PATHS

CURRENT_PATH

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

CURRENT_PATH_CPY

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

VALID_NEIGHBORS

{ 1,1 }

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:

```plaintext
MAZE = \{ \{0, 0\}, \{1,0\} \}
CURRENT_PATH = \{1,1\}
VALID_NEIGHBORS = \{ \{0, 0\}, \{1,0\}, \{1,1\} \}

PATHS = \{ \{0, 0\}, \{1,0\}, \{1,1\} \}
CURRENT_PATH_CPY = \{ \{0, 0\}, \{1,0\}, \{1,1\} \}

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:

Repeat!
Part I: Maze

solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.

$$\{ {0, 0}, {1,0}, {1,1} \}$$
Part I: Maze solveMaze()

Let’s walk thru a very small example:

3. While there are still more paths to explore:
   - Dequeue path from queue.

\[
\text{CURRENT_PATH} = \{ [0, 0], [1,0], [1,1] \}
\]
Part I: Maze
solveMaze()

Let’s walk thru a very small example:

1. Push path {0,0} onto queue.
2. Push path {1,0} onto queue.
3. While there are still more paths to explore:
   - Dequeue path from queue.
   - Push path {1,1} onto queue.
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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

MAZE

PATHS
Part I: Maze

solveMaze()

• Let’s walk thru a very small example:

3. While there are still more paths to explore:
   ○ Dequeue path from queue.
   ○ If this path ends at exit, this path is the solution!

### Maze

```
wall
wall
```

### Paths

```
\{ \{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:

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.

```
CURRENT_PATH
{{0, 0}, {1,0}, {1,1}}
```
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

   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.

   CURRENT_PATH = \{ \{0, 0\}, \{1,0\}, \{ 1,1 \} \}

   VALID_NEIGHBORS = \{1,2\}
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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.

MAZE

wall

wall

PATHS

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

CURRENT_PATH

CURRENT_PATH_CPY

VALID_NEIGHBORS

seen

seen

cur

valid

seen

seen

cur

valid
Part I: Maze

solveMaze()

• Let’s walk thru a very small example:

Let’s walk through a very small example of a maze-solving algorithm.

1. Initialize the maze with walls and valid paths.
2. Start at the current position (cur) and mark as seen.
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.
4. The solution is the last path in the queue.

See the diagram for a visual representation of the algorithm.
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:

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:

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:

    \[
    \text{VALID_NEIGHBORS} = \{ \{0, 0\}, \{1,0\}, \{1,1\}, \{1,2\} \}
    \]

Repeat!
Part I: Maze

\texttt{solveMaze()}

- Let’s walk thru a very small example:
Part I: Maze solveMaze()

• Let’s walk thru a very small example:
Part I: Maze
solveMaze()

• Let’s walk thru a very small example:
Part I: Maze solveMaze()

- Let’s walk thru a very small example:
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!
- Read the handout for more deets about BFS!
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!
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 that formats the provided TOKEN and returns the formatted version.

    string cleanToken(string token)

• 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.
     • ,,.EG’GS!; becomes EG’GS
  3. Convert TOKEN to lowercase.
     • EGGS becomes eggs

• You might find the char functions isalpha() and ispunct() helpful here!
Part II: Search engine

Next, you’ll need to implement the following function:

```
Set<string> gatherTokens(string bodytext)
```

Where BODYTEXT is a string consisting of all tokens found in a webpage.

- You must first tokenize the bodyText string into individual tokens separated by spaces. Look at stringSplit() for help there!
- Next, clean each token with your shiny new cleanToken() function, and store them in a Set<string> you'll return!
Part II: Search engine

• Any questions about gatherTokens()?
Part II: Search engine buildIndex()

• It’s now time for you to build an inverted index!
• This function takes in the name of a database file that you will parse, and a map that you will populate, pairing keywords with sets of urls that contain the keywords.

```java
int buildIndex(string dbfile, Map<string, Set<string>>& index)
```

Small sample inverted index:

```java
{
  “seach” : { “google.com, bing.com” } 
  “login” : { “webkinz.com” }
}
```
Part II: Search engine buildIndex()

• 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 content tokens to sets of URL's that contain the content tokens.
Part II: Search engine
buildIndex()

• Here’s the important thing:
  • You’ve already written a function gatherTokens() that turns a string into a set of cleaned tokens! You should probably use it here...

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
Part II: Search engine buildIndex()

- 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 an entire file into a vector using just the file name!
Part II: Search engine buildIndex()

- Some final notes about the problem:
  - 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"

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
```
Questions about buildIndex()?
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!

```java
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...
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”.

[Diagram of Union and Intersection]

- **Union (no operator)** - return all urls that contain at least 1 of the words!
- **Intersection (+ operator)** - return all urls that 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”
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”
• 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 -)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set1 + set2</code></td>
<td>O(N)</td>
<td>Returns the union of sets <code>set1</code> and <code>set2</code>, which is the set of elements that appear in at least one of the two sets.</td>
</tr>
<tr>
<td><code>set + value</code></td>
<td>O(N)</td>
<td>Returns the union of set <code>set1</code> and individual value <code>value</code>.</td>
</tr>
<tr>
<td><code>set1 += set2</code></td>
<td>O(N)</td>
<td>Adds all of the elements from <code>set2</code> (or the single specified value) to <code>set1</code>.</td>
</tr>
<tr>
<td><code>set += value</code></td>
<td>O(log N)</td>
<td>Adds the single specified value to the set.</td>
</tr>
<tr>
<td><code>set1 - set2</code></td>
<td>O(N)</td>
<td>Returns the difference of sets <code>set1</code> and <code>set2</code>, which is all of the elements that appear in <code>set1</code> but not <code>set2</code>.</td>
</tr>
<tr>
<td><code>set - value</code></td>
<td>O(N)</td>
<td>Returns the set <code>set</code> with <code>value</code> removed.</td>
</tr>
<tr>
<td><code>set1 -= set2</code></td>
<td>O(N)</td>
<td>Removes the elements from <code>set2</code> (or the single specified value) from <code>set1</code>.</td>
</tr>
<tr>
<td><code>set -= value</code></td>
<td>O(log N)</td>
<td>Removes the single specified value from the set.</td>
</tr>
<tr>
<td><code>set1 * set2</code></td>
<td>O(N)</td>
<td>Returns the intersection of sets <code>set1</code> and <code>set2</code>, which is the set of all elements that appear in both.</td>
</tr>
<tr>
<td><code>set1 *= set2</code></td>
<td>O(N)</td>
<td>Removes any elements from <code>set1</code> that are not present in <code>set2</code>.</td>
</tr>
</tbody>
</table>
Part II: Search engine
findQueryMatches()

• Some more tips for findQuery Matches()
  • 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!
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:

```cpp
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?

```cpp
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!
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!
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!