

CSE 12: PA3

1-21-21

Focus: PA3, BFS & DFS Maze Search

Tips on PAs overall

- If you know you can't finish, at least get the Gradescope Questions done.
 - Can get points back from Autograded portion, but not from the questions
- Look at the starter code! It is there to help and guide you
- Start early! The sooner we can help you the better you will do



PA 3

PA3 Breakdown

- Solving a 2D `Maze` represented by a 2D `Square` array (`Square[][]`)
- To maintain your search data and results as you progress through the maze, you will be implementing the `SearchWorklist` interface *twice*
 - Once using a queue to perform **BFS** in the `QueueWorklist` class
 - Once using a stack to perform **DFS** in the `StackWorklist` class
- To solve it, you will be implementing the `solve` method in the `MazeSolver` class
 - This will work by traversing the `previous` fields of squares from `finish` to `start`
- To see the solution path, you will implement the method `storePath()` in `Maze.java`
- You will also be making JUnit tests to test your `solve` method and two worklist classes

Some recommendations for PA3

- You can use java built-in tools like the `Stack` class and `LinkedList` interface
- Check out the provided tests in `TestSolvers.java` to see how to create tests that will compare an expected maze solution against what your `solve` method returns
- Create dummy methods (that do nothing meaningful) for the `StackWorklist` class, `QueueWorklist` class, and `solve` method of the `MazeSolver` class
 - `TestSolvers.java` will not compile until the above items are implemented with the bare minimum components
 - We recommend creating dummy methods so you can compile and run `TestSolvers.java` and work incrementally on each of the methods you're to implement
 - example dummy methods will be posted to demonstrate what we mean as well as 2 additional tests on BFS and DFS solutions for the maze during discussion today with some extra comments to clarify parts of `TestSolvers.java`. You can find them on the course Github in the discussion directory.

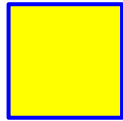
DFS and BFS with a Maze

To mirror the tasks for PA3, we will be going through the two following searches step-by-step to find solution paths for a maze:

- **DFS** using a stack
- **BFS** using a queue

The following step-by-step processes should be relevant to your task of implementing the `solve` method in the `MazeSolver` class

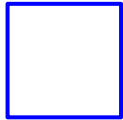
An example starting Maze and key



Start square



Finish square (goal)



Empty, unvisited space



Wall



Visited

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

Remember: Order matters when adding neighbors of a square to a worklist. For PA3, that order is: **NORTH, SOUTH, EAST, WEST** (this is different from lecture!)

DFS (Depth First Search)

Stepping through an example

Algorithm Pseudocode

```
initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)
```


s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4		
s5		
s6		
s7		
s8	true	---
s11		
s12		
s13		
s14		
s15		

- Pop the top element off of the stack (s8)

[illegible]

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13		
s14		
s15		

- s8 neighbors: s4, s12, s9
- Neighbors to push: s4, s12 (s9 is a wall)
- For s4, s12: Mark as visited, Set previous to s8
- Push s4, s12

Remember: For PA3 order matters (NORTH, SOUTH, EAST, WEST)

TOP

BOTTOM

s12	s4								
-----	----	--	--	--	--	--	--	--	--


```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

- s12 neighbors: s8, s13
- Neighbors to push: s13
- For s13: Mark as visited, set prev to s12
- Push s13

TOP

BOTTOM

s13	s4								
-----	----	--	--	--	--	--	--	--	--

- Pop the top element off of the stack (s13)

s4									
----	--	--	--	--	--	--	--	--	--

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- s13 neighbors: s9, s14, s12
- Neighbors to push: s14
- For s14: Mark as visited, set prev to s13
- Push s14

TOP

BOTTOM

s14	s4								
-----	----	--	--	--	--	--	--	--	--

- Pop the top element off of the stack (s14)

s4									
----	--	--	--	--	--	--	--	--	--

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

```
initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)
```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

square	visited	prev
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- s14 neighbors: s10, s15, s13
- Neighbors to push: s15
- For s15: Mark as visited, Set previous to s14
- Push s15

TOP

BOTTOM

s15	s4								
-----	----	--	--	--	--	--	--	--	--

- Pop the top element off of the stack (s15)

s4									
----	--	--	--	--	--	--	--	--	--

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- s15 neighbors: s11, s14
- Neighbors to push: s11
- For s11: Mark as visited, Set previous to s15
- Push s11

TOP

BOTTOM

s11	s4								
-----	----	--	--	--	--	--	--	--	--

- Pop the top element off of the stack (s11)

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

[illegible]

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- s11 neighbors: s7, s10, s15
- Neighbors to push: s7
- For s7: Mark as visited, Set previous to s11
- Push s7

TOP

BOTTOM

s7	s4								
----	----	--	--	--	--	--	--	--	--

- Pop the top element off of the stack (s7)

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

TOP									BOTTOM
s4									

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3	true	s7
s4	true	s8
s5		
s6	true	s7
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- s7 neighbors: s3, s11, s6
- Neighbors to push: s3, s6
- For s3, s6: Mark as visited, Set previous to s7
- Push s3, s6

TOP

BOTTOM

s6	s3	s4							
----	----	----	--	--	--	--	--	--	--


```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3	true	s7
s4	true	s8
s5		
s6	true	s7
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- Pop the top element off of the stack (s6)

TOP

BOTTOM

s3	s4								
----	----	--	--	--	--	--	--	--	--

```

initialize w1 to be a new empty Stack
push the start square to the Stack
mark the start as visited
while w1 is not empty:
    let current = pop the first element from Stack
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            push the neighbor to the Stack
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3	true	s7
s4	true	s8
s5		
s6	true	s7
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

- Return s6
-
- WE ARE DONE!!!! YAY!

TOP

BOTTOM

s3	s4								
----	----	--	--	--	--	--	--	--	--

How do we get the solution path with just the finish square (s6) returned?

Work backwards from finish to start!

- Check the finish square's previous square
- Check that square's previous square, then the next, and so forth until you hit the start square
- This gives you the solution path in reverse!

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

Path solution: s8, s12, s13, s14, s15, s11, s7, s6

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3	true	s7
s4	true	s8
s5		
s6	true	s7
s7	true	s11
s8	true	---
s11	true	s15
s12	true	s8
s13	true	s12
s14	true	s13
s15	true	s14

BFS (Breadth First Search)

Stepping through an example

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

square	visited	prev
s1		
s2		
s3		
s4		
s5		
s6		
s7		
s8	true	---
s11		
s12		
s13		
s14		
s15		

- Create a new queue
- enqueue s8 onto the queue
- Mark s8 as visited

FRONT

s8									
----	--	--	--	--	--	--	--	--	--

BACK

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4		
s5		
s6		
s7		
s8	true	---
s11		
s12		
s13		
s14		
s15		

- Dequeue the first element of the queue (s8)

BACK

[illegible]

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

square	visited	prev
s1		
s2		
s3		
s4	true	s8
s5		
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13		
s14		
s15		

- s8 neighbors: s4, s12, s9
- Neighbors to enqueue: s4, s12
- For s4, s12: mark as visited & set prev to s8
- enqueue s4, s12

FRONT

s4	s12							
----	-----	--	--	--	--	--	--	--

BACK


```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5	true	s4
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13		
s14		
s15		

- s4 neighbors: s0, s8, s5
- Neighbors to enqueue: s5 (s8 visited, s0 wall)
- For s5: mark as visited & set prev to s4
- enqueue s5

FRONT

s12	s5								
-----	----	--	--	--	--	--	--	--	--

BACK

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5	true	s4
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13		
s14		
s15		

- Dequeue the first element of the queue (s12)

FRONT

s5									
----	--	--	--	--	--	--	--	--	--

BACK

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

square	visited	prev
s1		
s2		
s3		
s4	true	s8
s5	true	s4
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

- s12 neighbors: s8, s13
- Neighbors to enqueue: s13 (s8 visited)
- For s13: mark as visited & set prev to s12
- enqueue s13

FRONT

s5	s13							
----	-----	--	--	--	--	--	--	--

BACK

- Dequeue the first element of the queue (s5)

BACK

s13									
-----	--	--	--	--	--	--	--	--	--

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1		
s2		
s3		
s4	true	s8
s5	true	s4
s6		
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2		
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

- s5 neighbors: s1, s9, s6, s4
- Neighbors to enqueue: s1, s6 (s4 visited, s9 wall)
- For s1, s6: mark as visited & set prev to s5
- enqueue s1, s6

FRONT

BACK

s13	s1	s6							
-----	----	----	--	--	--	--	--	--	--

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2		
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14		
s15		

- Dequeue the first element of the queue (s13)

FRONT

BACK

s1	s6								
----	----	--	--	--	--	--	--	--	--

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2		
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- s13 neighbors: s9, s14, s12
- Neighbors to enqueue: s14 (s12 visited, s9 wall)
- For s14: mark as visited & set prev to s13
- enqueue s14

FRONT

s1	s6	s14							
----	----	-----	--	--	--	--	--	--	--

BACK


```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2		
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- Dequeue the first element of the queue (s1)

FRONT

BACK

s6	s14								
----	-----	--	--	--	--	--	--	--	--

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2	true	s1
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- s1 neighbors: s5, s2, s0
- Neighbors to enqueue: s2 (s5 visited, s0 wall)
- For s2: mark as visited & set prev to s1
- enqueue s2

FRONT

s6	s14	s2							
----	-----	----	--	--	--	--	--	--	--

BACK

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2	true	s1
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- Dequeue the first element of the queue (s6)

FRONT

BACK

s14	s2								
-----	----	--	--	--	--	--	--	--	--

```

initialize w1 to be a new empty Queue
enqueue the start square to the Queue
mark the start as visited
while w1 is not empty:
    let current = dequeue the first element from Queue
    if current is the finish square
        return current
    else
        for each neighbor of current that isn't a wall and isn't visited
            mark the neighbor as visited
            set the previous of the neighbor to current
            enqueue the neighbor to the Queue
if the loop ended, return null (no path found)

```

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2	true	s1
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		

- Return current (s6)
- WE ARE DONE!!!

FRONT

BACK

s14	s2								
-----	----	--	--	--	--	--	--	--	--

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

Path solution: s8, s4, s5, s6

<u>square</u>	<u>visited</u>	<u>prev</u>
s1	true	s5
s2	true	s1
s3		
s4	true	s8
s5	true	s4
s6	true	s5
s7		
s8	true	---
s11		
s12	true	s8
s13	true	s12
s14	true	s13
s15		