CSE 12 Week 9 Discussion

5-28-21

Focus: PA8, Heaps, and Dijkstra's Algorithm

Reminders

- PA8 **(open!)** due Friday, June 4th @ 11:59 PM
  - All test cases visible. No resubmission

- PA6 Resubmission due TODAY @ 11:59 PM

- PA7 Resubmission due Friday, June 4th @ 11:59 PM

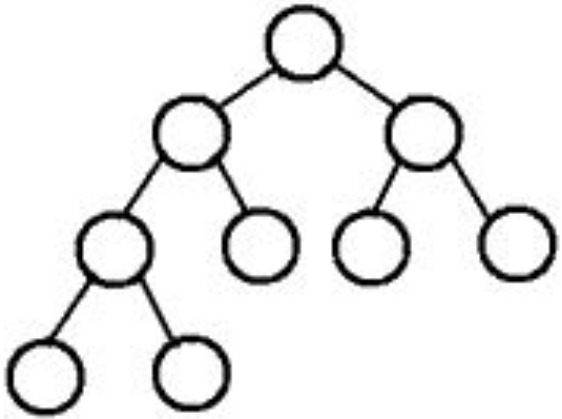
Overview of PA8

- Part I: An Implementation of `Heap`
  - Create a new file named Heap.java
  - All method headers and descriptions are given in the writeup. You will have to do the whole file from scratch!
  - Make sure to implement the PriorityQueue.java interface methods
- Part II: Implementation of MazeSolver
  - Utilize your heap based PriorityQueue to implement Dijkstra's Algorithm and solve a maze via the shortest path.
- Part III: Questions
  - Automatically graded multiple choice questions
  - Given a coding question, choose the data structure that you can use in your code to achieve an optimal time complexity

Heaps and Priority Queues

Heaps

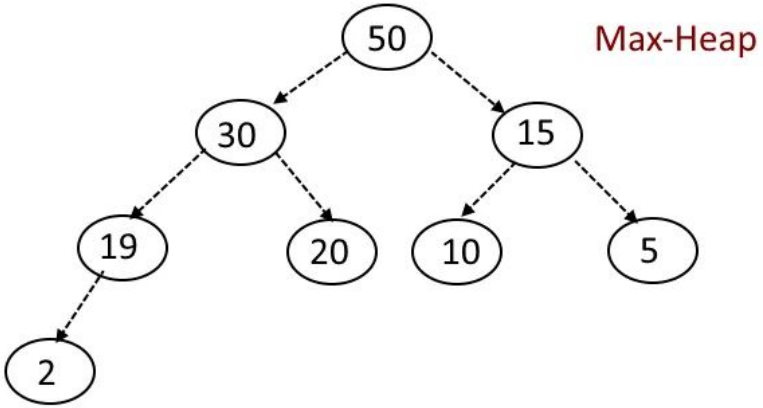
- A heap is a **complete** tree
  - Every level is full except possibly the last, and all nodes are as far left as possible.
- It might not necessarily be a **full** tree
  - Every node other than the leaves have two children



complete tree

Heaps

- Implemented with a list
- min/max heap
  - useful when we care about the next largest/smallest value



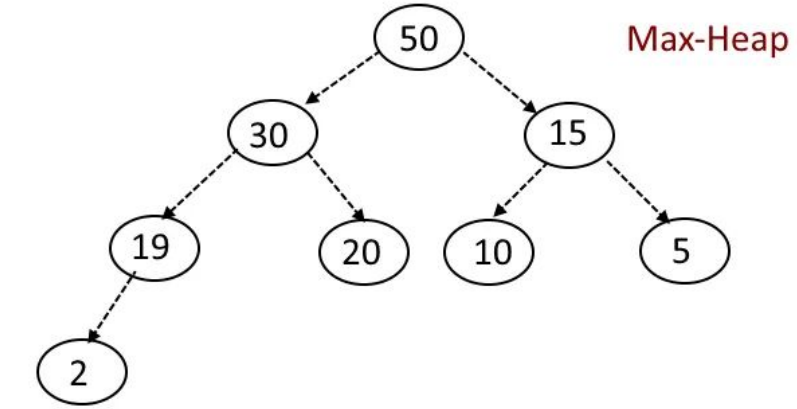
0	1	2	3	4	5	6	7	8
	50	30	15	19	20	10	5	2

for Node at i : Left child will be 2i and right child will be at 2i+1 and parent node will be at [i/2].

Question

How can I get the parent node of a node in the following heap?

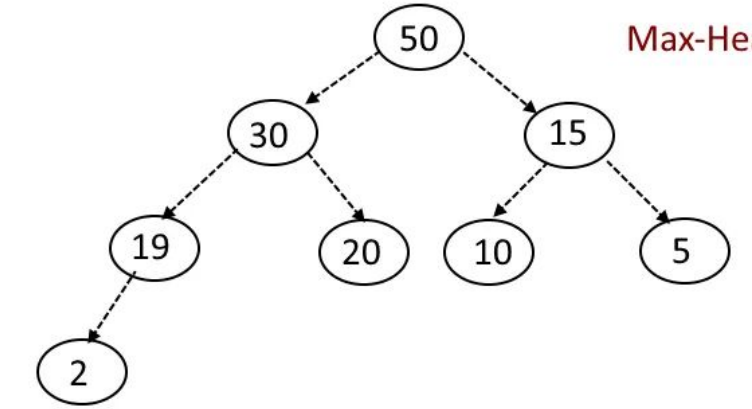
A) i/2
B) i/2 -1
C) i - 2
D) None of these



Question

How can I get the parent node of a node in the following heap?

A) i/2
B) i/2 -1
C) i - 2
D) None of these



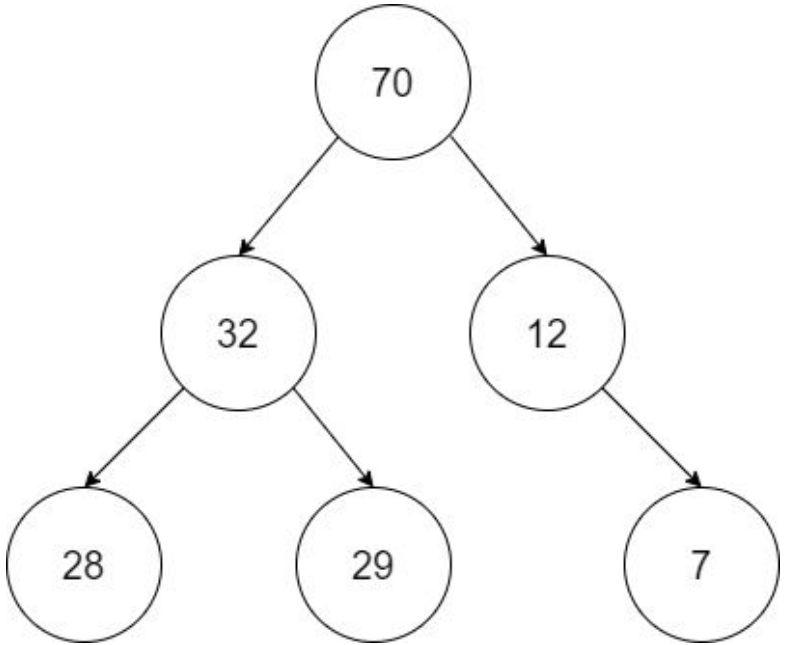
0	1	2	3	4	5	6	7	8
	50	30	15	19	20	10	5	2

for Node at i : Left child will be 2i and right child will be at 2i+1 and parent node will be at [i/2].

Question

Is this a valid Heap?

A) Yes
B) No

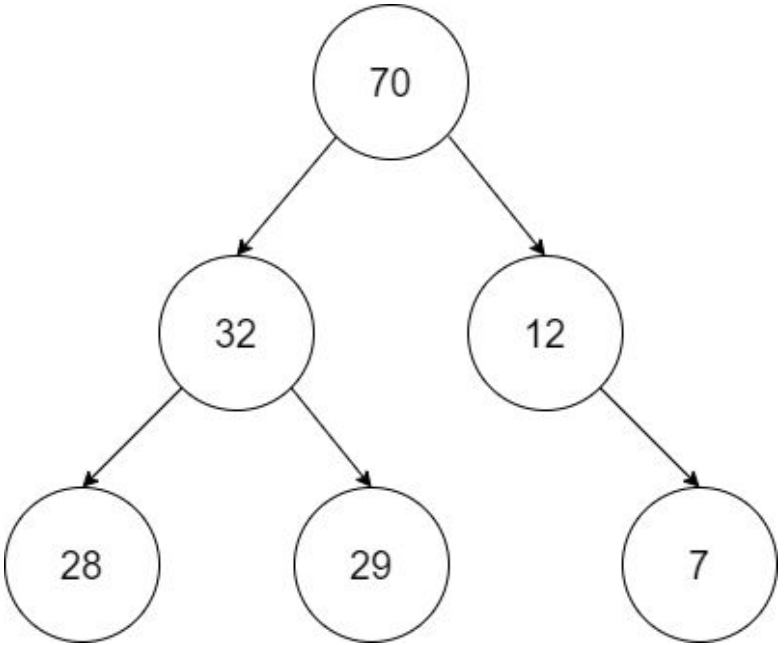


Question

Is this a valid Heap?

A) Yes

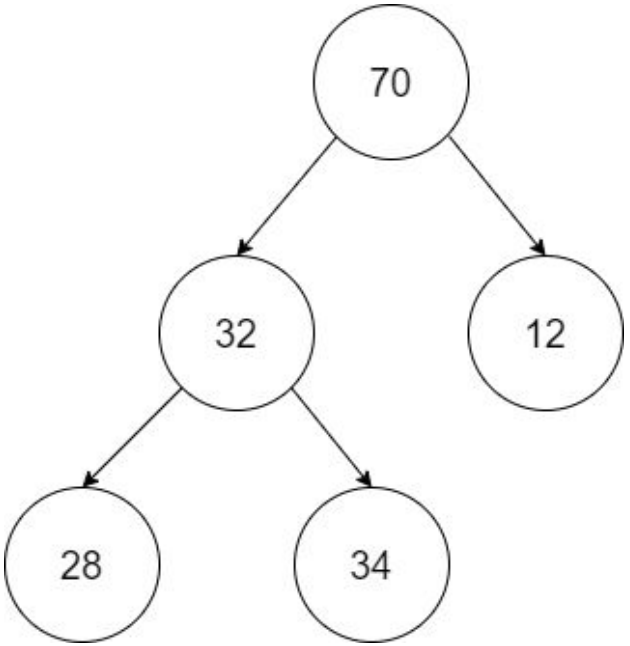
B) No



Question

Is this a valid Heap?

A) Yes
B) No

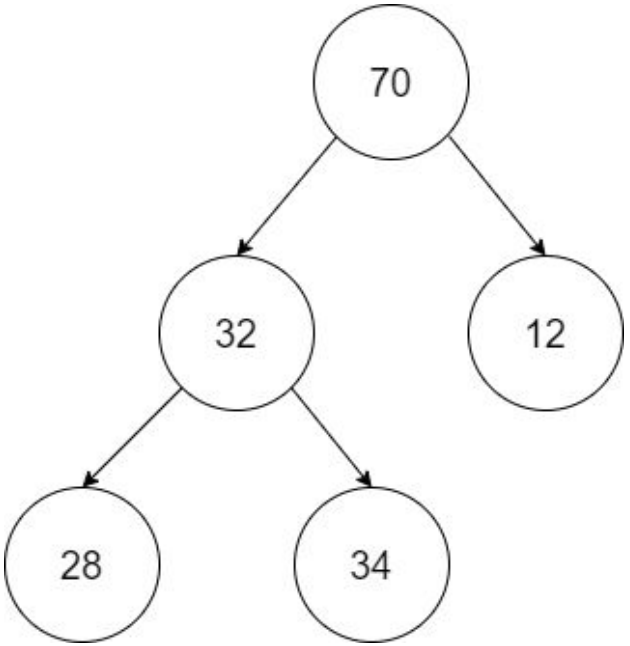


Question

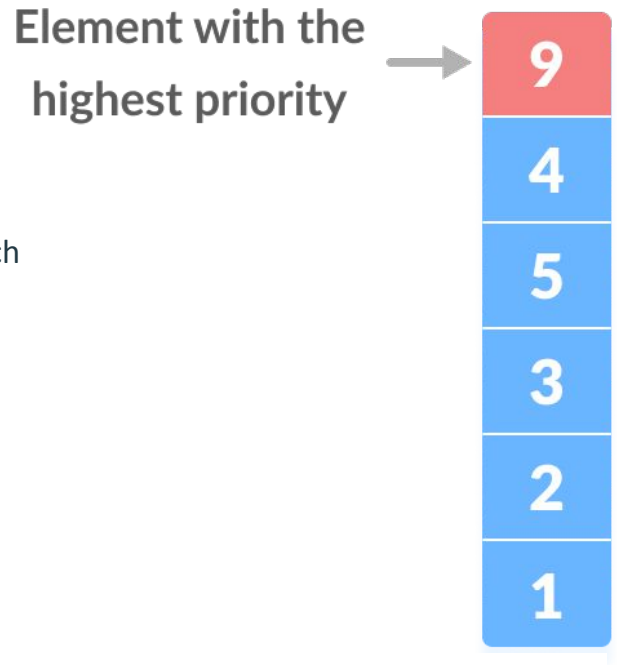
Is this a valid Heap?

A) Yes

B) No



Priority Queues



- Similar to a queue in the sense that we are adding/removing from the same location each time. However, now there is an order that is based on a priority value
- Methods
  - `poll()`
  - `add()`
  - `peek()`
  - `toArray()`
  - `isEmpty()`

Heaps vs Priority Queues

Heap is a **Data Structure**

Priority Queue is an **Abstract Data Type (ADT)**

Heaps are the most popular way to implement a Priority Queue because they efficient at finding the largest or smallest values (the priority). In fact, many times when people refer to a priority queue they are referring to a heap!

You will be creating a Heap class however you will use the Priority Queue interface provided.

What about the helper methods?

Bubble Down

- Used for deleting an element from the heap
- Take last element of heap and put it at the index of the element to be deleted
- Check and Swap
  a. Min-heap: if replaced element > any child node, swap element with the child that is smaller
  b. Max-heap: if replaced element < any child node, swap element with the child that is greater
- Keep repeating till conditions are not met

Bubble Up

- Used for inserting an element into the heap
- Insert element at the last leaf of the tree
- Check and Swap
  a. Min-heap: if inserted element < parent node, swap element with parent node
  b. Max-heap: if inserted element > parent node, swap element with parent node
- Keep repeating till the inserted element is in place
- http://btv.melezinek.cz/binary-heap.html

Dijkstra’s Algorithm

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Create a new heap (pq)
- Push <0, s8> into the pq

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4			inf
s5			inf
s6			inf
s7			inf
s8		---	inf
s11			inf
s12			inf
s13			inf
s14			inf
s15			inf

TOP **BOTTOM**

<0, s8>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s8)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4			inf
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12			inf
s13			inf
s14			inf
s15			inf

TOP **BOTTOM**


```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
          for each neighbor of currentSquare that isn't a wall and isn't visited
             let currentCost = current's key plus the neighbors cost
             if currentCost is less than neighbor's runningCost
                 set the previous of the neighbor to currentSquare
                 set the neighbors runningCost to currentCost
                 add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- s8 neighbors: s4, s12, s9
- Neighbors to consider: s4, s12 (s9 is a wall)
- For s4, s12
  - calculate runningCost by adding current's key + s4/s12 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s8
    - update runningCost
    - Push <14, s4>, <1, s12>

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12		s8	1
s13			inf
s14			inf
s15			inf

TOP **BOTTOM**

<1, s12>	<14, s4>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s12)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13			inf
s14			inf
s15			inf

TOP **BOTTOM**

<14, s4>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13		s12	4
s14			inf
s15			inf

- s12 neighbors: s8, s13
- Neighbors to consider: s13 (s8 is visited)
- For s13
  - calculate runningCost by adding current's key + s13 cost,
  - If calculated runningCost is less than current runningCost
    - Set previous to s12
    - update runningCost
    - Push <4, s13>

TOP **BOTTOM**

<4, s13>	<14, s4>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s13)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13	true	s12	4
s14			inf
s15			inf

TOP **BOTTOM**

<14, s4>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13	true	s12	4
s14		s13	5
s15			inf

- s13 neighbors: s12, s9, s14
- Neighbors to consider: s14 (s12 is visited, s9 is a wall)
- For s14
  - calculate runningCost by adding current's key + s14 cost,
  - If calculated runningCost is less than current runningCost
    - Set previous to s13
    - update runningCost
    - Push <5, s14>

TOP **BOTTOM**

<5, s14>	<14, s4>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s14)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15			inf

TOP **BOTTOM**

<14, s4>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15		s14	7

- s14 neighbors: s13, s10, s15
- Neighbors to consider: s15 (s13 is visited, s10 is a wall)
- For s15
  - calculate runningCost by adding current's key + s15 cost,
  - If calculated runningCost is less than current runningCost
    - Set previous to s14
    - update runningCost
    - Push <7, s15>

TOP **BOTTOM**

<7, s15>	<14, s4>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s15)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11			inf
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<14, s4>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11		s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

- s15 neighbors: s14, s11
- Neighbors to consider: s11 (s14 is visited)
- For s11
  - calculate runningCost by adding current's key + s11 cost,
  - If calculated runningCost is less than current runningCost
    - Set previous to s15
    - update runningCost
    - Push <9, s11>

TOP **BOTTOM**

<9, s11>	<14, s4>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s11)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7			inf
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<14, s4>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4		s8	14
s5			inf
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

- s11 neighbors: s10, s7, s15
- Neighbors to consider: s7(s10 is a wall, s15 is visited)
- For s7
  - calculate runningCost by adding current's key + s7 cost,
  - If calculated runningCost is less than current runningCost
    - Set previous to s11
    - update runningCost
    - Push <18, s7>

TOP **BOTTOM**

<14, s4>	<18, s7>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s4)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0			inf
s1			inf
s2			inf
s3			inf
s4	true	s8	14
s5			inf
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
          return currentSquare
     else
          for each neighbor of currentSquare that isn't a wall and isn't visited
             let currentCost = current's key plus the neighbors cost
             if currentCost is less than neighbor's runningCost
                 set the previous of the neighbor to currentSquare
                 set the neighbors runningCost to currentCost
                 add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0		s4	15
s1			inf
s2			inf
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

- s4 neighbors: s0, s5, s8
- Neighbors to consider: s0, s5 (s8 is visited)
- For s0, s5
  - calculate runningCost by adding current's key + s0/s5 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s4
    - update runningCost
    - Push <15, s0>, <20, s5>

TOP **BOTTOM**

<15, s0>	<18, s7>	<20, s5>							

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s0)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1			inf
s2			inf
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>	<20, s5>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
          return currentSquare
     else
          for each neighbor of currentSquare that isn't a wall and isn't visited
             let currentCost = current's key plus the neighbors cost
             if currentCost is less than neighbor's runningCost
                 set the previous of the neighbor to currentSquare
                 set the neighbors runningCost to currentCost
                 add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- s0 neighbors: s1, s4
- Neighbors to consider: s1 (s4 is visited)
- For s1
  - calculate runningCost by adding current's key + s1 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s0
    - update runningCost
    - Push <16, s1>

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1		s0	16
s2			inf
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<16, s1>	<18, s7>	<20, s5>							

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s1)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2			inf
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>	<20, s5>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2		s1	17
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

- s1 neighbors: s0, s2, s5
- Neighbors to consider: s2, s5 (s0 is visited)
- For s2, s5
  - calculate runningCost by adding current's key + s2/s5 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s1
    - update runningCost
    - Push <17, s2>, <22, s5>

TOP **BOTTOM**

<17, s2>	<18, s7>	<20, s5>							

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s2)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3			inf
s4	true	s8	14
s5		s4	20
s6			inf
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>	<20, s5>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3		s2	18
s4	true	s8	14
s5		s4	20
s6		s2	20
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

- s2 neighbors: s1, s3, s6
- Neighbors to consider: s3, s6 (s1 is visited)
- For s3, s6
  - calculate runningCost by adding current's key + s3/s6 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s2
    - update runningCost
    - Push <18, s3>, <20, s6>

TOP **BOTTOM**

<18, s3>	<18, s7>	<20, s6>	<20, s5>						

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s3)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6		s2	20
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>	<20, s6>	<20, s5>							

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
          return currentSquare
     else
          for each neighbor of currentSquare that isn't a wall and isn't visited
             let currentCost = current's key plus the neighbors cost
             if currentCost is less than neighbor's runningCost
                 set the previous of the neighbor to currentSquare
                 set the neighbors runningCost to currentCost
                 add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- s3 neighbors: s2, s7
- Neighbors to consider: s7 (s2 is visited)
- For s7
  - calculate runningCost by adding current's key + s7 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s3
    - update runningCost
    - Push <27, s7>

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6		s2	20
s7		s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<18, s7>	<20, s6>	<20, s5>							

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s7)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6		s2	20
s7	true	s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<20, s6>	<20, s5>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
          return currentSquare
     else
          for each neighbor of currentSquare that isn't a wall and isn't visited
             let currentCost = current's key plus the neighbors cost
             if currentCost is less than neighbor's runningCost
                 set the previous of the neighbor to currentSquare
                 set the neighbors runningCost to currentCost
                 add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- s7 neighbors: s6, s3, s11
- Neighbors to consider: s6 (s3, s11 are visited)
- For s6
  - calculate runningCost by adding current's key + s6 cost
  - If calculated runningCost is less than current runningCost
    - Set previous to s7
    - update runningCost
    - Push <21, s6>

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6		s2	20
s7	true	s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<20, s6>	<20, s5>								

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Remove the first element of the pq (s6)
- Mark it as visited

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6	true	s2	20
s7	true	s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<20, s5>									

```
initialize pq to be a new empty heap
add the start square's cost as the key
and the start square itself as the value to pq
while pq is not empty:
     let current = remove the first entry from pq (poll)
     let currentSquare = current's value
     Mark currentSquare as visited
     if currentSquare is the finishing square
         return currentSquare
     else
         for each neighbor of currentSquare that isn't a wall and isn't visited
            let currentCost = current's key plus the neighbors cost
            if currentCost is less than neighbor's runningCost
                set the previous of the neighbor to currentSquare
                set the neighbors runningCost to currentCost
                add the currentCost as key and neighbor as value to the pq (add)
if the loop ended, return null (no path found)
```

- Return finish square

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

1	1	1	1
14	6	3	9
0	0	0	2
1	3	1	2

square	V	P	RC
s0	true	s4	15
s1	true	s0	16
s2	true	s1	17
s3	true	s2	18
s4	true	s8	14
s5		s4	20
s6	true	s2	20
s7	true	s11	18
s8	true	---	inf
s11	true	s15	9
s12	true	s8	1
s13	true	s12	4
s14	true	s13	5
s15	true	s14	7

TOP **BOTTOM**

<20, s5>									