CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Stacks, Queues, and Mazes (oh my)
Textbook Ch. 7
Recap

Stacks: Last In First Out (LIFO)
- Push (put item on top of the stack)
- Pop (take item off top of stack)
- Top (peek at top of stack)

Queues: First in First Out (FIFO)
- Enqueue (put item on the end of the queue)
- Dequeue (take item off the front of the queue)
- Head (peek at the item in the front of the queue)
Recap

**Stacks: Last In First Out (LIFO)**
- Push (put item on top of the stack) $\Theta(1)$ (or amortized $O(1)$)
- Pop (take item off top of stack) $\Theta(1)$
- Top (peek at top of stack) $\Theta(1)$

**Queues: First in First Out (FIFO)**
- Enqueue (put item on the end of the queue) $\Theta(1)$
- Dequeue (take item off the front of the queue) $\Theta(1)$
- Head (peek at the item in the front of the queue) $\Theta(1)$
Thought Question: How could you use an array to build a queue?
Queues

ArrayBuffer Attempt 1

Enqueue: Append(…)

Dequeue: Remove(0)
Queues

ArrayBuffer Attempt 1

**Enqueue:** Append(...)  
**Dequeue:** Remove(0)

*What is the complexity?*
Queues

ArrayBuffer Attempt 1

**Enqueue:** Append(...)  \[\text{Amortized } O(1)\]

**Dequeue:** Remove(0) \[O(n)\]

*What is the complexity?*
Queues

ArrayOfBuffer Attempt 2

**Enqueue:** Insert(0)

**Dequeue:** Remove(last)
ArrayBuffer Attempt 2

**Enqueue:** Insert(0)

**Dequeue:** Remove(last)

*What is the complexity?*
ArrayBuffer Attempt 2

**Enqueue:** Insert(0) \(\mathcal{O}(n)\)

**Dequeue:** Remove(last) \(\Theta(1)\)

*What is the complexity?*
Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?
Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

*Why didn't we have to pay that cost with a list?*
Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

Why didn't we have to pay that cost with a list?

Update our values of "first" and "last"!
Queues

Front: 17
Back: 26
Queues

```
17  88  13  14  26  5
```

enqueue(5)

front

back
Queues

17  88  13  14  26  5  4

enqueue(5)
enqueue(4)

front
back
Queues

enqueue(5)
enqueue(4)
dequeue()
Enqueue(5)
Enqueue(4)
Dequeue()
Dequeue()
Queues

enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
Queues

enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
dequeue()
enqueue(7)
Queues

enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
Queues

enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
Queues

enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
enqueue(-3)
ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)
ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

**Enqueue**
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

**Dequeue**
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

*What is the complexity?*
ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

**Enqueue** Amortized $O(1)$
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

**Dequeue** $\Theta(1)$
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

*What is the complexity?*
Why Ring Buffer?

front

back
Why Ring Buffer?

Conceptually, we can think of this as a ring...
Why Ring Buffer?
Why Ring Buffer?

enqueue(42)
Why Ring Buffer?

enqueue(28)
Why Ring Buffer?

enqueue(1)
Why Ring Buffer?
Why Ring Buffer?

dequeue()
Applications of Stacks and Queue

**Stack:** Checking for balanced parentheses/braces

**Queue:** Scheduling packets for delivery

**Both:** Searching mazes
Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie {()} is not ok).

{(()){{}}}   {()}   ()
Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie {()} is not ok).

{(()(}}{)} }  {()}   ()

✓
Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie {()} is not ok).

{((){}}} {()} ()

✓ ✗
Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie {()} is not ok).

{(()){{}}}  {()()  ()

✓  ×  ×
Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on (, decrement on )
Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on (, decrement on )

Problem: allows for {()}
Idea #2

**Idea:** Track nesting on a stack!

On ( or {, push the symbol on the stack.

On ) or }, pop the stack and check for a match.
Demo from last fall:

[https://odin.cse.buffalo.edu/teaching/cse-250/2022fa/slide/14b-QueueStackApps.html#/13]
Network Packets

Router: 1gb/s internal network, 100mb/s external
- 1 gb/s sent to the router, but only 100mb/s can leave.
- How do we handle this?

Queues
- Enqueue data packets in the order they are received.
- When there is available outgoing bandwidth, dequeue and send.

Avoiding Queueing Delays
- Limit size of queue; Packets that don't fit are dropped

TCP: blocked packets are retried
UDP: application deals with dropped packets
Mazes

○ is the start, X is the objective
  ● There may be multiple paths
  ● Generally, we want the shortest

Approach 1: Take the first available route in one direction
  ● Right, Down, Left, or Up
  ● Down, Right, Up, or Left
Mazes

How do you know which one is best?

Is there anything wrong with this algorithm?
Mazes
Priority order doesn't guarantee exploring the entire maze
Formalizing Maze-Solving

Inputs:

- The map: an $n \times m$ grid of squares which are either filled or empty
- The $O$ is at position $start$
- The $X$ is at position $dest$

Goal: Compute $\text{steps}(start, \ dest)$, the minimum number of steps from $start$ to $end$.

How do we define the $\text{steps}$ function?
Mazes
How many steps are required for the squares right next to X?
How many steps are required for the squares right next to X?
How many steps are required for the squares right next to X?
And the squares next to those?
Mazes
Mazes
Mazes
# Mazes

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>O</strong></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Mazes
Mazes
Mazes
Mazes
# Mazes

![Maze Diagram]

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>∞</td>
<td>8</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
So what is the number of steps from O to X?
Mazes

So what is the number of steps from O to X?

4 (min of neighbors + 1)
Mazes

Does this solution remind you of anything?
Mazes

Does this solution remind you of anything?

Recursion!
Mazes

\[
\text{steps}(\text{pos}, \text{dest}) = \begin{cases} 
0 & \text{if } \text{pos} = \text{dest} \\
\infty & \text{if } \text{is\_filled}(\text{pos}) \\
1 + \text{min\_adjacent}(\text{pos}, \text{dest}) & \text{otherwise}
\end{cases}
\]

where...

\[
\text{min\_adjacent}(\text{pos}, \text{dest}) = \min \left\{ \begin{array}{l}
\text{steps}(\text{moveRight}(\text{pos}), \text{dest}) \\
\text{steps}(\text{moveDown}(\text{pos}), \text{dest}) \\
\text{steps}(\text{moveLeft}(\text{pos}), \text{dest}) \\
\text{steps}(\text{moveUp}(\text{pos}), \text{dest})
\end{array} \right\}
\]
Mazes

steps(pos, dest):
    if pos == dest then return 0
    elif is_filled(pos) then return \infty
    else return 1 + \min of
        steps(moveRight(pos, dest))
        steps(moveDown(pos, dest))
        steps(moveLeft(pos, dest))
        steps(moveUp(pos, dest))
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
## Mazes

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

- **O** represents the starting point.
- **X** represents the goal.
- Numbers indicate steps or levels.

---

**Note:** The maze is represented with a grid, where each cell is numbered from 1 to 15. The path from **O** to **X** should be determined by following the numbers in ascending order.
Problem: Infinite loop!
Mazes

Problem: Infinite loop!

Insight: A path with a loop in it can't be shorter than one without the loop
Mazes

steps(pos, dest):
    if pos == dest then return 0
    elif is_visited(pos) then return ∞
    elif is_filled(pos) then return ∞
    else
        Mark pos as visited
        return 1 + min of all 4 steps
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
**Problem:** The first time you visit a node may be from a longer path!
Problem: The first time you visit a node may be from a longer path!

Insight: Unmark nodes as you leave them.
Mazes

steps(pos, dest):
    if pos == dest then return 0
    elif is_visited(pos) then return ∞
    elif is_filled(pos) then return ∞
    else
        Mark pos as visited
        min = 1 + min of all 4 steps
        Mark pos as unvisited
    return min
Mazes
Mazes
Mazes
Mazes
Mazes
# Mazes

![Maze Diagram]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

O
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Mazes
Formalizing Maze-Solving

Inputs:

- The map: an $n \times m$ grid of squares which are either filled or empty
- The $\text{O}$ is at position $\text{start}$
- The $\text{X}$ is at position $\text{dest}$

Goal: Compute $\text{steps}(\text{start, dest})$, the minimum number of steps from start to end.
Formalizing Maze-Solving

**Inputs:**
- The map: an $n \times m$ grid of squares which are either filled or empty
- The $O$ is at position $\text{start}$
- The $X$ is at position $\text{dest}$

**Goal:** Compute $\text{steps}(\text{start}, \text{dest})$, the minimum number of steps from start to end.
Formalizing Maze-Solving

Inputs:

- The map: an $n \times m$ grid of squares which are either filled or empty
- The $O$ is at position $start$
- The $X$ is at position $dest$

Goal: Compute $steps(start, dest)$, the minimum number of steps from start to end. ✓

What path did we take?
Mazes

Idea: Keep track of the nodes marked visited...that's our path!
Mazes: Now with...some data structure?

```python
steps(pos, dest, visited):
    if pos == dest then return visited.copy()
    elif pos ∈ visited then return no_path
    elif is_filled(pos) then return no_path
    else
        visited.append(pos)
        bestPath = 1 + min of all 4 steps
        visited.removeLast()
    return bestPath
```
steps(pos, dest, visited):
    if pos == dest then return visited.copy()
    elif pos ∈ visited then return no_path
    elif is_filled(pos) then return no_path
    else
        visited.append(pos)
        bestPath = 1 + min of all 4 steps
        visited.removeLast()
    return bestPath

What could this data structure be??
Mazes: Now with...Stacks!

steps(pos, dest, visited):
    if pos == dest then return visited.copy()
    elif pos ∈ visited then return no_path
    elif is_filled(pos) then return no_path
    else
        visited.push(pos)
        bestPath = 1 + min of all 4 steps
        visited.pop()
    return bestPath

A stack!
Queues?

Thought Experiment: Can we do something similar with queues?