CSE 250: Stacks and Queues

Lecture 16

Oct 6, 2023

Reminders

- WA3 due Sun, Oct 15 at 11:59 PM

Back to Sequence ADTs

- **Sequence**
  - `get(i)`, `set(i, v)`
- **List**
  - ... and `add(v)`, `add(i, v)`, `remove(i)`,
- **Stack**
  - `push(v)`, `pop()`, `peek()`
- **Queue**
  - `add(v)`, `remove()`, `peek()`

The Stack ADT

A stack of objects on top of one another.

- **Push**
  Put a new object on top of the stack.
- **Pop**
  Remove the object from the top of the stack.
- **Top**
  Peek at what's on top of the stack.

Stacks

Demo

Stacks in Practice

- Storing method-local variables ("call stack")
- Certain types of parsers ("context free")
- Backtracking search (more on Friday)
- Reversing sequences

The Stack Interface

```
1  public interface Stack<E> extends List<E> {
2    public boolean empty();
3    public E peek();
4    public E pop();
5    public E push(E item);
6    /* ... */
7  }
```

LinkedListStack

```
1   class LinkedListStack<E> implements Stack<E> {
2     LinkedList<E> data = new LinkedList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { /* ??? */ }
7
8     public E peek() { /* ??? */ }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

LinkedListStack

```
1   class LinkedListStack<E> implements Stack<E> {
2     LinkedList<E> data = new LinkedList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { return data.add(0, item); }
7
8     public E peek() { /* ??? */ }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

LinkedListStack

```
1   class LinkedListStack<E> implements Stack<E> {
2     LinkedList<E> data = new LinkedList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { return data.add(0, item); }
7
8     public E peek() { return data.get(0); }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

LinkedListStack

```
1   class LinkedListStack<E> implements Stack<E> {
2     LinkedList<E> data = new LinkedList();
3
4     public boolean empty() { return data.size() == 0; }   O(1)
5
6     public E push(E item) { return data.add(0, item); }   O(1)
7
8     public E peek() { return data.get(0); }               O(i) = O(1)
9
10    public E pop(E item) { return data.remove(0); }       O(1)
11  }
```

ArrayListStack

```
1   class ArrayListStack<E> implements Stack<E> {
2     ArrayList<E> data = new ArrayList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { /* ??? */ }
7
8     public E peek() { /* ??? */ }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

ArrayListStack

```
1   class ArrayListStack<E> implements Stack<E> {
2     ArrayList<E> data = new ArrayList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { return data.add(item); }
7
8     public E peek() { /* ??? */ }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

ArrayListStack

```
1   class ArrayListStack<E> implements Stack<E> {
2     ArrayList<E> data = new ArrayList();
3
4     public boolean empty() { return data.size() == 0; }
5
6     public E push(E item) { return data.add(item); }
7
8     public E peek() { return data.get(data.size()-1); }
9
10    public E pop(E item) { /* ??? */ }
11  }
```

ArrayListStack

```
1   class ArrayListStack<E> implements Stack<E> {
2     ArrayList<E> data = new ArrayList();
3
4     public boolean empty() { return data.size() == 0; }   O(1)
5
6     public E push(E item) { return data.add(item); }   am. O(1)
7
8     public E peek() { return data.get(data.size()-1); }   O(1)
9
10    public E pop(E item) {
11        return data.remove(data.size()-1); }    O(N − i) = O(1)
12  }
```

Stacks in Java

Java's `Stack` is implemented using a `Vector`.
(Like an `ArrayList`, but grows by adding a constant to its capacity)

What does this mean for the runtime of `push(item)` $O(N)$

What other assumptions make this ok?

Stacks usually have a maximum capacity.

- $2\times$ capacity is wasteful when you have an upper limit.
- Each call to **pop** gives you another **push** credit.
- Keeping elements together in memory is worth the overhead.

The Queue ADT

Outside of the US, "queueing" is lining up.

- **Enqueue** (`add(item)` or `offer(item)`)
  Put a new object at the end of the queue.
- **Dequeue** (`remove()` or `poll()`)
  Remove the object from the front of the queue.
- **Peek** (`element()` or `peek()`)
  Peek at what's at the front of the queue.

Queues

Demo

Queues vs Stacks

- **Queue**
  First in, First out (FIFO)
- **Stack**
  Last in, First out (LIFO, FILO)

Queues in Practice

- Delivering network packets, emails, tootstagrexes.
- Scheduling the CPU
- Deferring long-running tasks

The Queue Interface

```
1  public interface Queue<E> {
2    public boolean add(E item);
3    public E remove();
4    public E peek();
5    /* ... */
6  }
```

LinkedListQueue

```
1  public interface LinkedListQueue<E> implements Queue<E> {
2    LinkedList<E> data = new LinkedList<>();
3
4    public boolean add(E item) { /* ??? */ }
5    public E remove() { /* ??? */ }
6    public E peek() { /* ??? */ }
7  }
```

LinkedListQueue

```
1  public interface LinkedListQueue<E> implements Queue<E> {
2    LinkedList<E> data = new LinkedList<>();
3
4    public boolean add(E item) { data.add(item); }
5    public E remove() { /* ??? */ }
6    public E peek() { /* ??? */ }
7  }
```

LinkedListQueue

```
1  public interface LinkedListQueue<E> implements Queue<E> {
2    LinkedList<E> data = new LinkedList<>();
3
4    public boolean add(E item) { data.add(item); }
5    public E remove() { return data.remove(0); }
6    public E peek() { /* ??? */ }
7  }
```

LinkedListQueue

```
1  public interface LinkedListQueue<E> implements Queue<E> {
2    LinkedList<E> data = new LinkedList<>();
3
4    public boolean add(E item) { data.add(item); }      O(1)
5    public E remove() { return data.remove(0); }        O(1)
6    public E peek() { return data.get(0); }             O(1)
7  }
```

ArrayListQueue

Thought Question:
How could you use an `ArrayList` to build a `Queue`?

ArrayListQueue (v1)

```
1  public interface ArrayListQueue<E> implements Queue<E> {
2    ArrayList<E> data = new ArrayList<>();
3
4    public boolean add(E item) { data.add(item); }        O(1)
5    public E remove() { return data.remove(0); }          O(N)
6    public E peek() { return data.get(0); }               O(1)
7  }
```

ArrayListQueue (v2)

```
1  public interface ArrayListQueue<E> implements Queue<E> {
2    ArrayList<E> data = new ArrayList<>();
3
4    public boolean add(E item) { data.add(item, 0); }     O(N)
5    public E remove()
6                 { return data.remove(data.size()-1); }   O(1)
7    public E peek() { return data.get(0); }                O(1)
8  }
```

Circular Buffer

Demo

ArrayListQueue (final)

Active Array = [start, end)

- `Enqueue(item)` **am.** $O(1)$
  - Resize Array (if needed) — am. $O(1)$
  - `data[end] = item` — $O(1)$
  - `end = (end + 1) % capacity` — $O(1)$
- `Dequeue(item)` $O(1)$
  - `return data[start]` — $O(1)$
  - `start = (start + 1) % capacity` — $O(1)$

This is called a **Circular** or **Ring Buffer**.

Balanced Parenthesis

What does it mean for parenthesis/braces to be balanced.

- Every opening symbol is matched by a closing symbol.
- Every closing symbol is matched by an opening symbol.
- No nested overlaps (e.g., {(}) is not ok).

{()({})} ✓

{()) ×

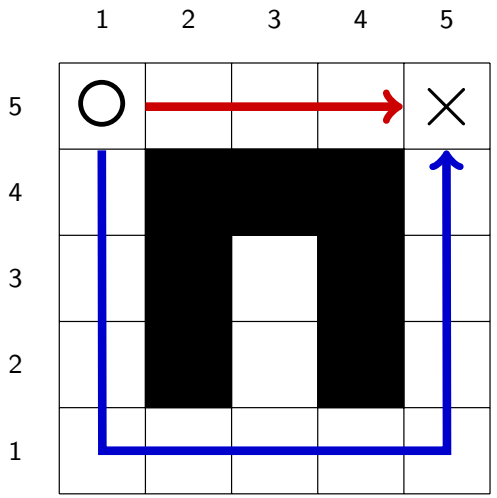
()) ×

Balanced Parenthesis

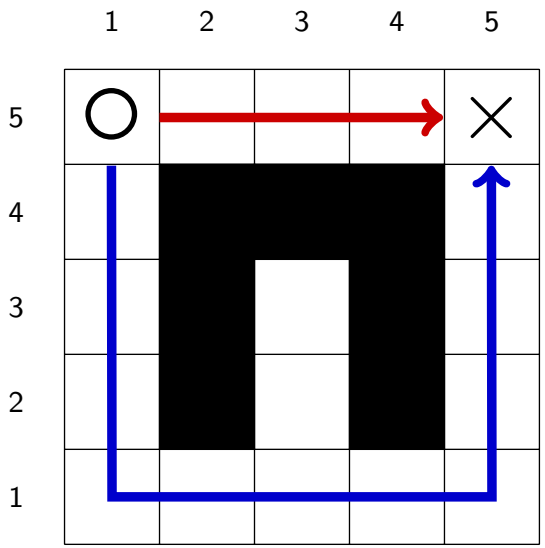
Demo

Mazes

- ○ is the start, × 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**

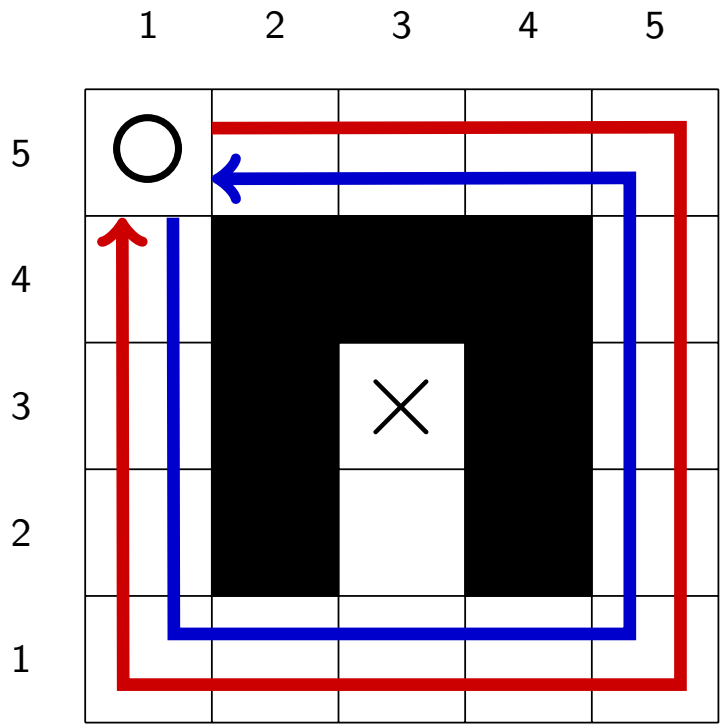


Mazes



- How do we know which one is best?
- Are there any other problems?

Mazes



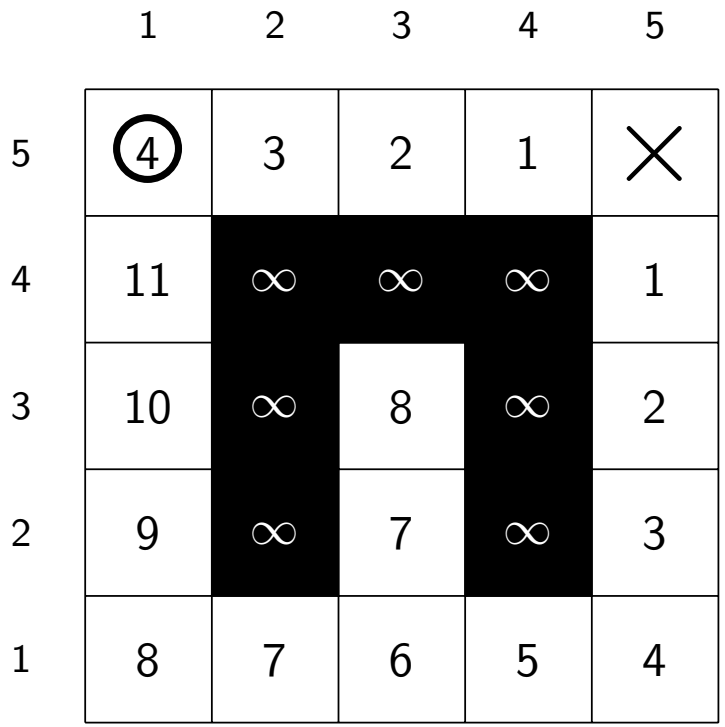
- Priority order doesn't guarantee exploring the entire maze

Formalizing Maze Solving

- **Inputs**
  - The map: An $n \times m$ grid of filled/empty squares.
  - The $\circ$ is at position `start`
  - The $\times$ is at position `dest`
- **Goal**
  - Compute `steps`(`start`, `dest`), the minimum steps from start to end.

How do we define `steps`?

Formalizing Maze Solving



Formalizing Maze Solving

$$\texttt{steps}(\texttt{pos}, \texttt{dest}) = \begin{cases} 0 & \textbf{if } \texttt{pos} = \texttt{dest} \\ \infty & \textbf{if } \texttt{pos} \textbf{ is filled} \\ 1 + \texttt{min\_nearby}(\texttt{pos}, \texttt{dest}) & \textbf{otherwise} \end{cases}$$

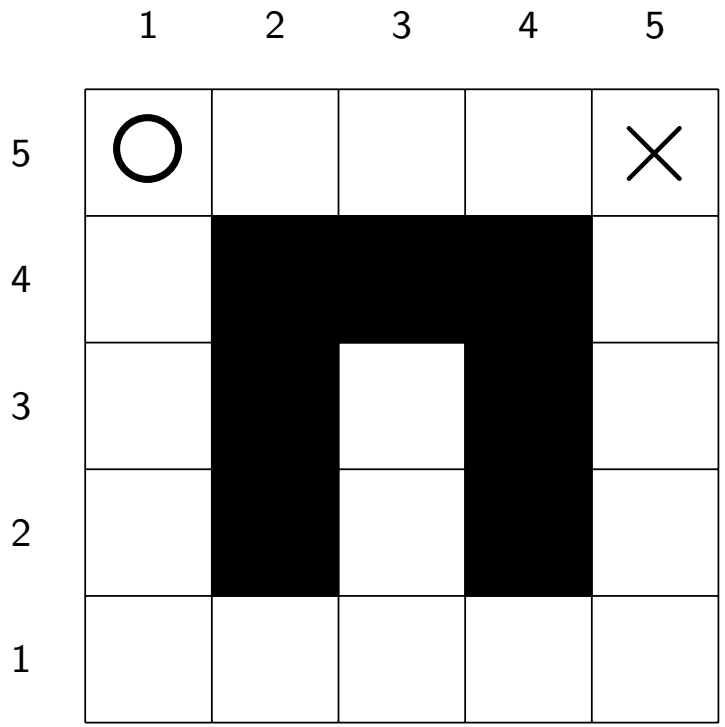
Where...

$$\texttt{min\_nearby}(\texttt{pos}, \texttt{dest}) = \min \begin{cases} \texttt{steps}(\texttt{moveRight}(\texttt{pos}), \texttt{dest}) \\ \texttt{steps}(\texttt{moveDown}(\texttt{pos}), \texttt{dest}) \\ \texttt{steps}(\texttt{moveLeft}(\texttt{pos}), \texttt{dest}) \\ \texttt{steps}(\texttt{moveUp}(\texttt{pos}), \texttt{dest}) \end{cases}$$

Formalizing Maze Solving

```
1   public int steps(Point pos, Point dest)
2   {
3     if(pos == dest){ return 0; }
4     else if(is_filled(pos){ return ∞; }
5     else {
6       return 1 + Math.min(
7         steps(pos.moveRight, dest),
8         steps(pos.moveDown, dest),
9         steps(pos.moveLeft, dest),
10        steps(pos.moveUp, dest)
11      );
12    }
13  }
```

Formalizing Maze Solving



Formalizing Maze Solving

Problem: Infinite Loop

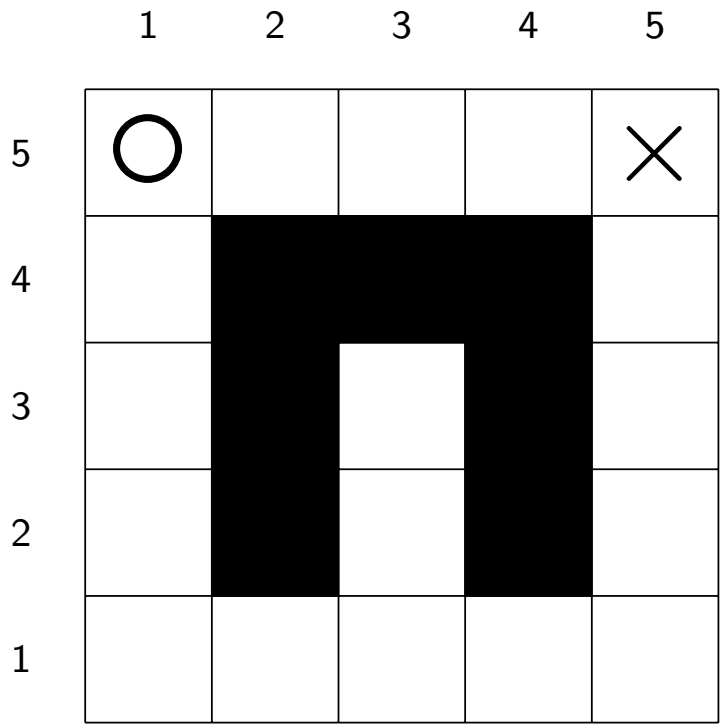
Insight: A path with a loop in it can't be shorter than one without the loop.

Mark nodes as visited so you don't visit them twice.

Formalizing Maze Solving

```
1   public int steps(Point pos, Point dest)
2   {
3     if(pos == dest){ return 0; }
4     else if(is_filled(pos){ return ∞; }
5     else if(is_visited(pos){ return ∞; }
6     else {
7       mark_visited(pos);
8       return 1 + Math.min(
9         steps(pos.moveRight, dest),
10        steps(pos.moveDown, dest),
11        steps(pos.moveLeft, dest),
12        steps(pos.moveUp, dest)
13      );
14    }
15  }
```

Formalizing Maze Solving



Formalizing Maze Solving

Problem: The first time you visit a node it may be via a longer path!

Unmark nodes as you leave them.

Formalizing Maze Solving

```
1   public int steps(Point pos, Point dest)
2   {
3     if(pos == dest){ return 0; }
4     else if(is_filled(pos){ return ∞; }
5     else if(is_visited(pos){ return ∞; }
6     else {
7       mark_visited(pos);
8       int stepCount = 1 + Math.min(
9         steps(pos.moveRight, dest),
10        /* ... */
11      );
12      unmark_visited(pos);
13      return stepCount;
14    }
15  }
```

Formalizing Maze Solving

Question: What path did we take?

Track the current path in a `Stack`

Formalizing Maze Solving

```
1   public Array[Point] steps(Point pos, Point dest, Stack visited)
2   {
3     if(pos == dest){ return visited.toArray(); }          ←
4     else if(is_filled(pos){ return null; }
5     else if(visited.contains(pos){ return null; }         ←
6     else {
7       visited.push(pos);                                  ←
8       int steps = shortest_array(
9         steps(pos.moveRight, dest),
10        /* ... */
11      );
12      visited.pop(pos);                                   ←
13    }
14  }
```