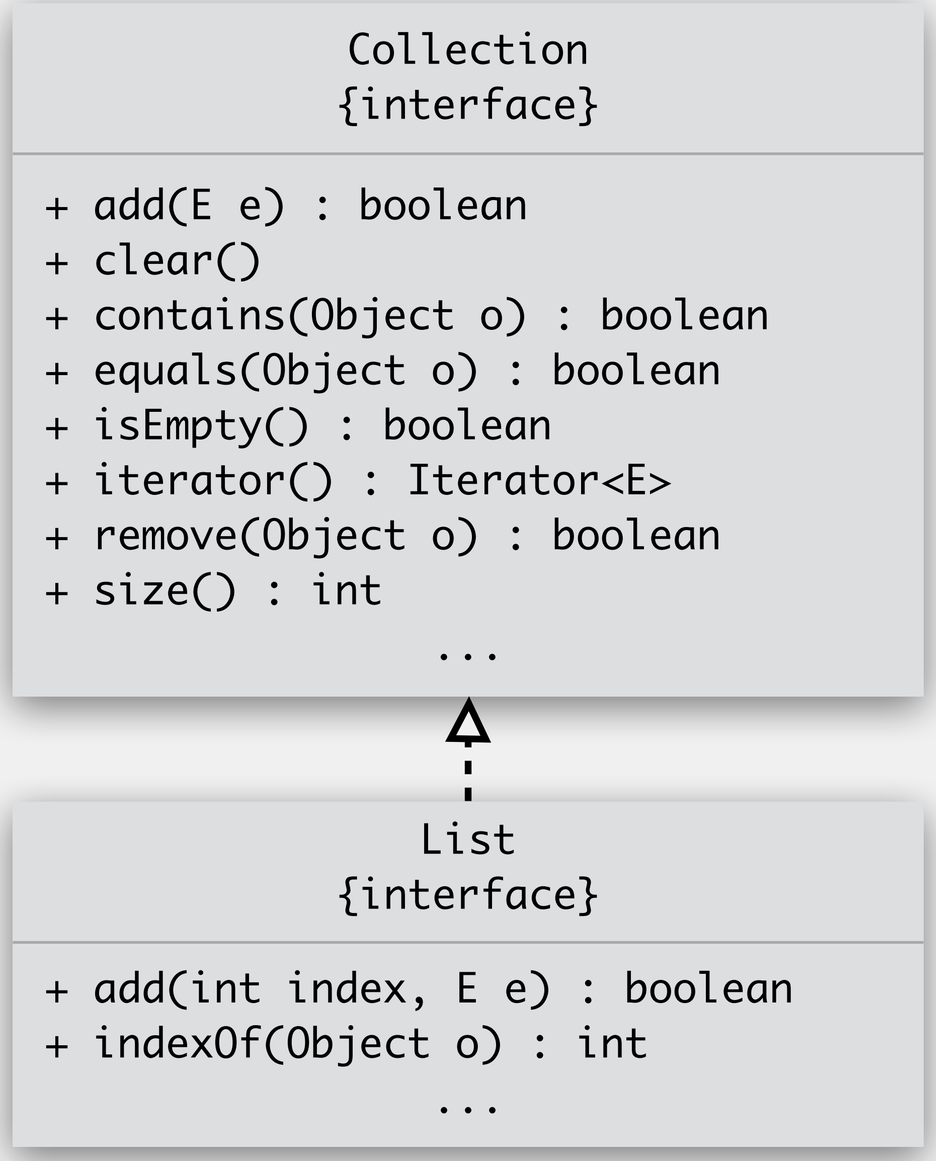
University of Wisconsin
LA CROSSE
Computer Science

Week 13:
Stacks and Queues

CS 220: Software Design II — D. Mathias

The `Collection` and `List` Interfaces



`Collection` describes a group of objects

`List` holds data in a linear fashion

Together, we can ask questions like...

- what is the last index of a particular value?
- is the list empty?
- how many values are in there?

Two different ways to implement

- array
- linked nodes

Linear Data Structures

`ArrayList` and `LinkedList` are relatively unconstrained data structures

- data is held in a linear fashion
- can (seemingly) contain as many values as required
- can add/remove/change values anywhere

What if we want constraints on our linear data structure?

Constrained Linear Data Structures

Consider the following scenarios

- representing ~~victims~~ customers and their order at the DMV
- edit or browser history
- student IDs at a university
- Amazon wish list items and their quantity

What are the constraints and/or unique data storage requirements in each of these scenarios?

- consider how data is added/removed
- relationship between different pieces of data

Abstract Data Types

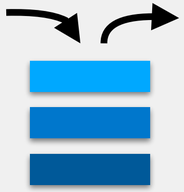
abstract data types (*ADT*) describe how methods should modify the stored data, without specifying what underlying actions are required

Other common ADTs used in programming

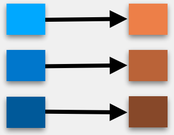
queues

sets

stacks



maps



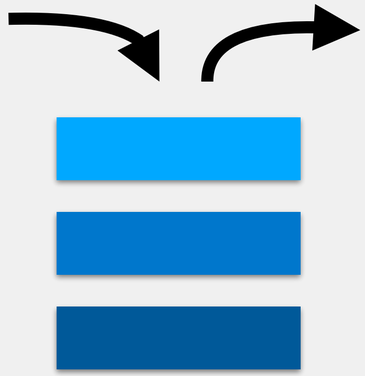
Stacks

Constrains our linear data structure such that we can only add and remove elements from the top

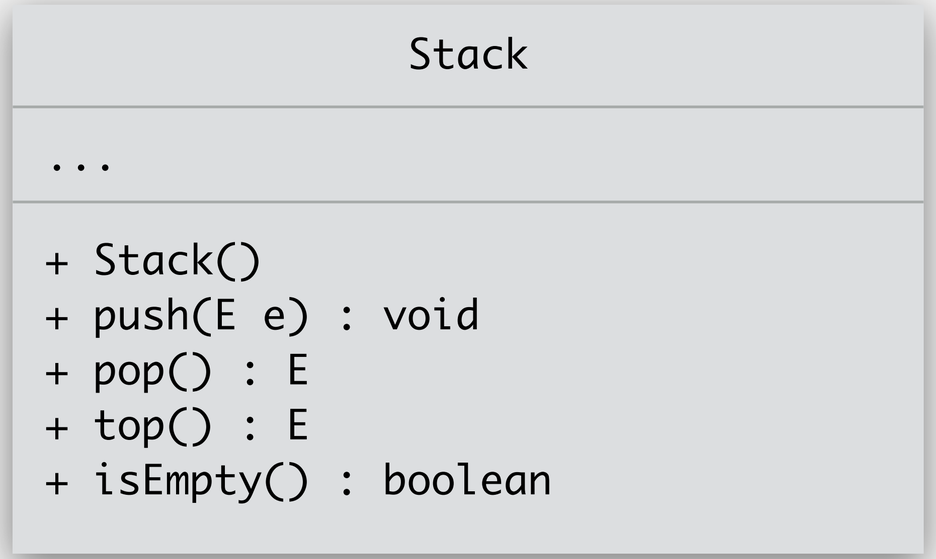
- often referred to as a *last in, first out* data structure
- i.e., the last value added will be the first value removed

Good for getting elements back in the reverse order they are added

Have already seen one particular use of this with Java memory management



The `Stack` Class



Related to `Collection`, `List`

`push` adds an element to the top of the stack

`pop` removes and returns the element from the top of the stack

`top` (or `peek`) returns (but does not remove) the element from the top of the stack

`isEmpty` returns whether or not the stack is empty

Implementing the Stack Class

```
public class Stack<E> {
    private List<E> stack = /* instantiation omitted */;

    public void push(E e) {
        // TODO: implement me
    }

    public E pop() {
        // TODO: implement me

    }

    public E top() {
        // TODO: implement me

    }
}
```

Implementing the Stack Class

```
public class Stack<E> {
    private List<E> stack = /* instantiation omitted */;

    public void push(E e) {
        stack.add(0, e);
    }

    public E pop() {

        return stack.remove(0);
    }

    public E top() {

        return stack.get(0);
    }
}
```

Implementing the Stack Class

```
public class Stack<E> {
    private List<E> stack = /* instantiation omitted */;

    public void push(E e) {
        stack.add(0, e);
    }

    public E pop() {
        if(isEmpty()) { throw new EmptyStackException(); }
        return stack.remove(0);
    }

    public E top() {
        if(isEmpty()) { throw new EmptyStackException(); }
        return stack.get(0);
    }
}
```

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

> myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
> myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

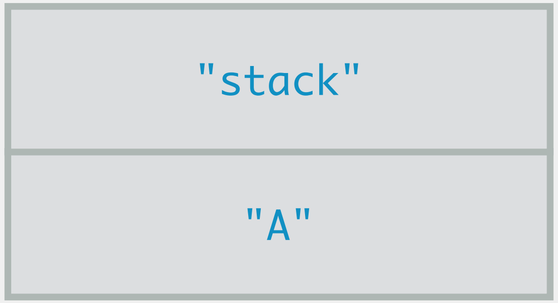
myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
> System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```



myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
> myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

```
  stack
```

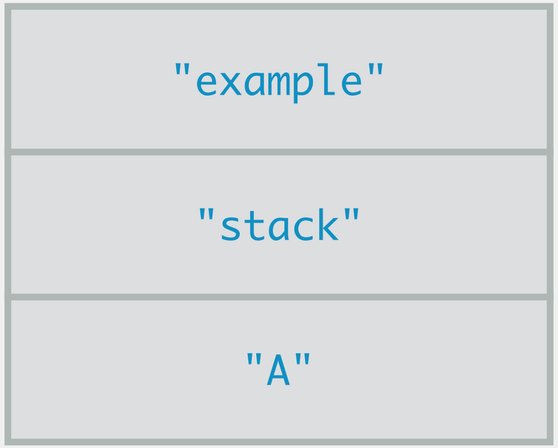
Exercise: Using Stacks

```
Stack<String> myStack = new Stack<>();

myStack.push("A");
myStack.push("stack");
System.out.println(myStack.top());
myStack.push("example");

> while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

```
stack
```



myStack

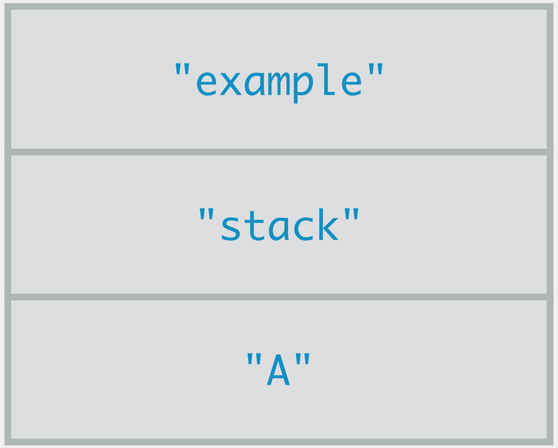
Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
>     System.out.println(myStack.pop());
  }
```

```
  stack
```



myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
> }
```

```
stack
example
```

Exercise: Using Stacks

```
Stack<String> myStack = new Stack<>();

myStack.push("A");
myStack.push("stack");
System.out.println(myStack.top());
myStack.push("example");

> while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

```
stack
example
```

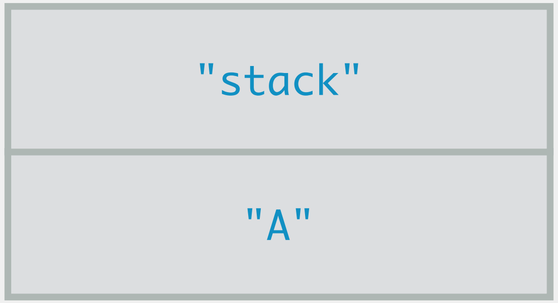
Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
>     System.out.println(myStack.pop());
  }
```

```
  stack
  example
```



myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
> }
```

```
stack
example
stack
```

myStack

Exercise: Using Stacks

```
Stack<String> myStack = new Stack<>();

myStack.push("A");
myStack.push("stack");
System.out.println(myStack.top());
myStack.push("example");

> while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

```
stack
example
stack
```

myStack

Exercise: Using Stacks

```
Stack<String> myStack = new Stack<>();

myStack.push("A");
myStack.push("stack");
System.out.println(myStack.top());
myStack.push("example");

while(!myStack.isEmpty()) {
>     System.out.println(myStack.pop());
}
```

```
stack
example
stack
```

myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

  while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
> }
```

```
stack
example
stack
A
```

myStack

Exercise: Using Stacks

```
  Stack<String> myStack = new Stack<>();

  myStack.push("A");
  myStack.push("stack");
  System.out.println(myStack.top());
  myStack.push("example");

> while(!myStack.isEmpty()) {
      System.out.println(myStack.pop());
  }
```

```
  stack
  example
  stack
  A
```

myStack

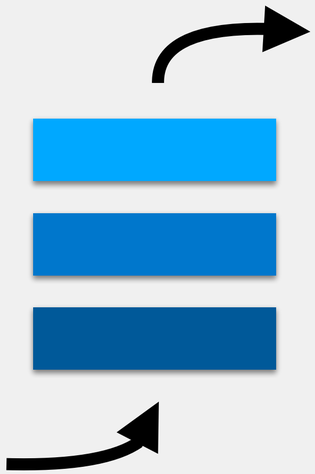
Queues

Constrains our linear data structure such that we can only add elements to the end and remove elements from the beginning

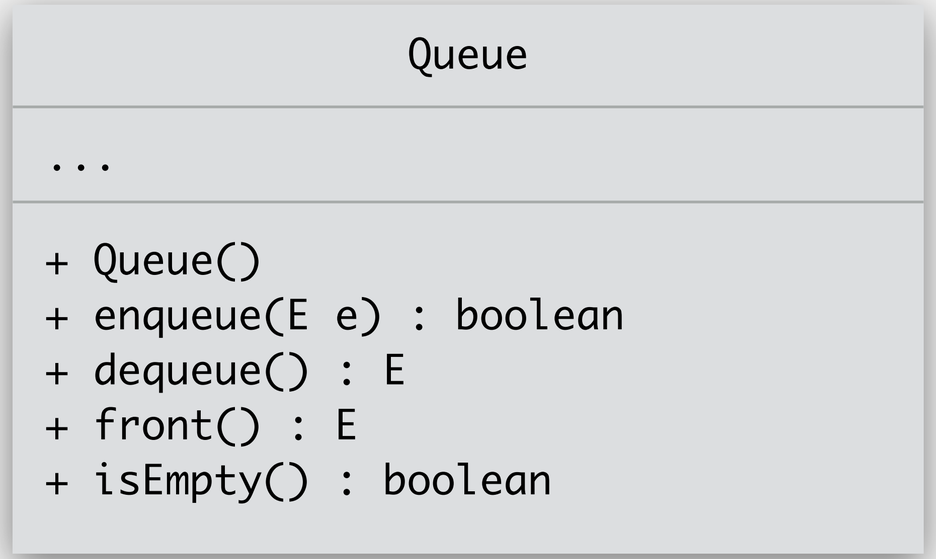
- often referred to as a *first in, first out* data structure

Good for getting elements back in the order they are added

Just like queues for a music playlist, lines for ordering tickets...



The Queue Class



(For some reason, Java designers decided to change the commonly used method names.)

Related to `Collection`, `List`

`enqueue` (or *add*) adds an element to the end of the queue

`dequeue` (or `poll`) removes and returns the element from the front of the queue

`front` (or `peek`) returns (but does not remove) the element from the front of the queue

`isEmpty` returns whether or not the queue is empty

Implementing the Queue Class

```
public class Queue<E> {
    private List<E> queue = /* instantiation omitted */;

    public boolean enqueue(E e) {
        // TODO: implement me
   }

    public E dequeue() {
        // TODO: implement me

    }

    public E front() {
        // TODO: implement me

    }
}
```

Implementing the Queue Class

```
public class Queue<E> {
    private List<E> queue = /* instantiation omitted */;

    public boolean enqueue(E e) {
        return queue.add(e);
    }

    public E dequeue() {
        if(isEmpty()) { throw new NoSuchElementException(); }
        return queue.remove(0);
    }

    public E front() {
        if(isEmpty()) { throw new NoSuchElementException(); }
        return queue.get(0);
    }
}
```

Exercise: Using Queues

```
  Queue<String> myQueue = new Queue<>();

> myQueue.enqueue("A");
  myQueue.enqueue("queue");
  System.out.println(myQueue.front());
  myQueue.enqueue(“example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

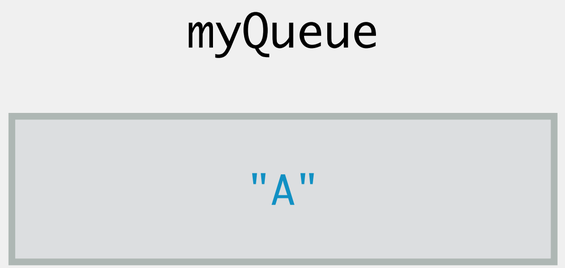
myQueue

Exercise: Using Queues

```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
> myQueue.enqueue("queue");
  System.out.println(myQueue.front());
  myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```



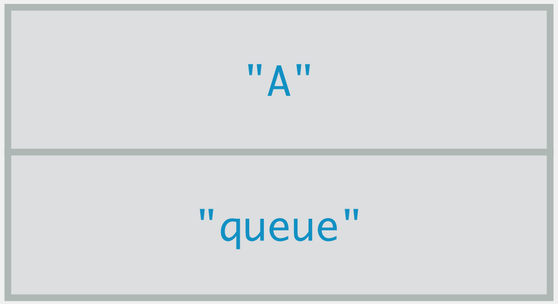
Exercise: Using Queues

```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
  myQueue.enqueue("queue");
> System.out.println(myQueue.front());
  myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

myQueue



Exercise: Using Queues

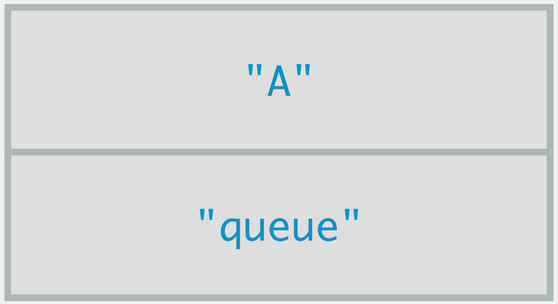
```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
  myQueue.enqueue("queue");
  System.out.println(myQueue.front());
> myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

```
  A
```

myQueue



Exercise: Using Queues

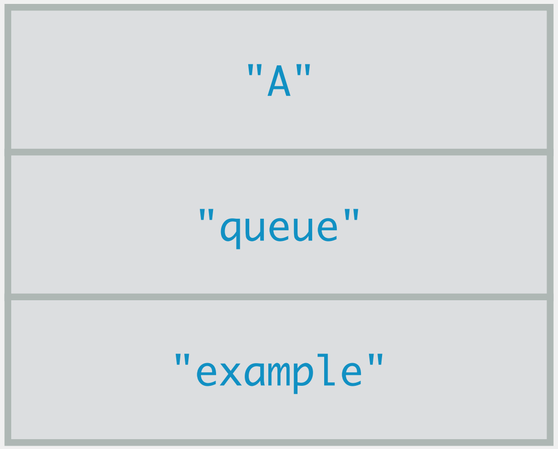
```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

> while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

```
A
```

myQueue



Exercise: Using Queues

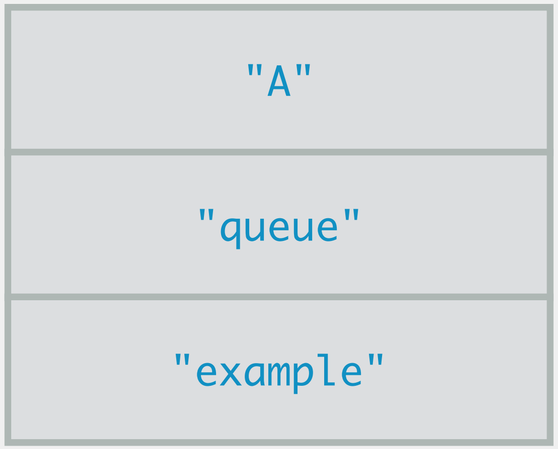
```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

while(!myQueue.isEmpty()) {
>     System.out.println(myQueue.dequeue());
}
```

```
A
```

myQueue



Exercise: Using Queues

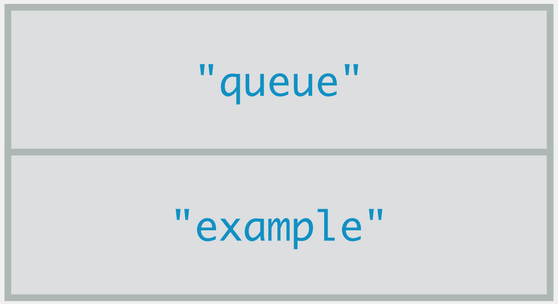
```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
  myQueue.enqueue("queue");
  System.out.println(myQueue.front());
  myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
> }
```

```
A
A
```

myQueue



Exercise: Using Queues

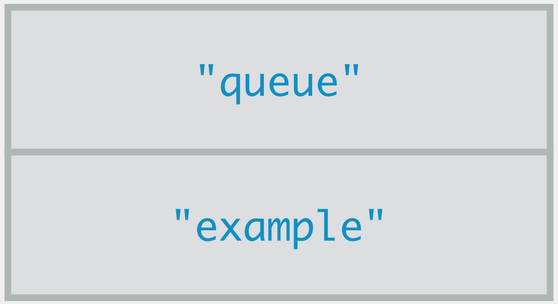
```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

> while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

```
A
A
```

myQueue



Exercise: Using Queues

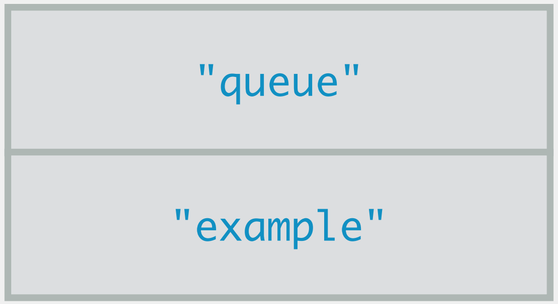
```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

while(!myQueue.isEmpty()) {
>     System.out.println(myQueue.dequeue());
}
```

```
A
A
```

myQueue



Exercise: Using Queues

```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
  myQueue.enqueue("queue");
  System.out.println(myQueue.front());
  myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
> }
```

```
A
A
queue
```

myQueue

Exercise: Using Queues

```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

> while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

```
A
A
queue
```

myQueue

Exercise: Using Queues

```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

while(!myQueue.isEmpty()) {
>     System.out.println(myQueue.dequeue());
}
```

```
A
A
queue
```

myQueue

Exercise: Using Queues

```
  Queue<String> myQueue = new Queue<>();

  myQueue.enqueue("A");
  myQueue.aenqueue("queue");
  System.out.println(myQueue.front());
  myQueue.enqueue("example");

  while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
> }
```

```
A
A
queue
example
```

myQueue

Exercise: Using Queues

```
Queue<String> myQueue = new Queue<>();

myQueue.enqueue("A");
myQueue.enqueue("queue");
System.out.println(myQueue.front());
myQueue.enqueue("example");

> while(!myQueue.isEmpty()) {
      System.out.println(myQueue.dequeue());
  }
```

```
A
A
queue
example
```

myQueue

Using Lists for Stacks and Queues

Stack and queue implementations ultimately need some underlying structure to hold the data

- just like linked lists need list nodes and array lists need arrays

Can use either arrays or linked lists to store the stack/queue data[1]

- notice that our stack/queue implementations (in lecture) rely on the `List` interface, which is implemented by both types of lists

1: https://stackoverflow.com/a/7477556

Stack Implementation Revisited

```
public class Stack<E> {
    private List<E> stack = /* instantiation omitted */;

    public void push(E e) {
        stack.add(0, e);
    }

    public E pop() {
        if(isEmpty()) { throw new EmptyStackException(); }
        return stack.remove(0);
    }

    public E top() {
        if(isEmpty()) { throw new EmptyStackException(); }
        return stack.get(0);
    }
}
```

How would this do from a runtime perspective with…

- a linked list?
- an array list?

Queue Implementation Revisited

```
public class Queue<E> {
    private List<E> queue = /* instantiation omitted */;

    public boolean enqueue(E e) {
        return queue.add(e);
    }

    public E dequeue() {
        if(isEmpty()) { throw new NoSuchElementException(); }
        return queue.remove(0);
    }

    public E front() {
        if(isEmpty()) { throw new NoSuchElementException(); }
        return queue.get(0);
    }
}
```

How would this do from a runtime perspective with…

- a linked list?
- an array list?

What Does Java Provide?

There is a `Stack` class which looks like what is presented in the slides

- can also use `LinkedList`, which provides implementations of `push`, `pop`, and `peek`

`Queue` is only an interface

- easiest to use `LinkedList`, which provides implementations of `add`, `poll`, and `peek`

Problem Solving w/Stacks & Queues

- Queues are fairly straightforward
  - need to store something in the order it comes in? use a queue!
- Stacks are more interesting
  - want to do something in reverse? use a stack!
  - can also emulate recursive algorithms in an iterative fashion
    - recursion implicitly leverages Java's runtime stack for memory management
    - we can instead explicitly maintain our own stack for an algorithm

Search using Stacks & Queues

Depth-first search begins at some starting point (i.e., a *root*) and explores as far down a path as it can before backtracking and exploring another path

- implemented using a stack

Breadth-first search begins at some starting point (i.e., a *root*) and explores all surrounding neighbors before exploring all of their neighbors

- implemented using a queue

Depth-First Search

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

myStack

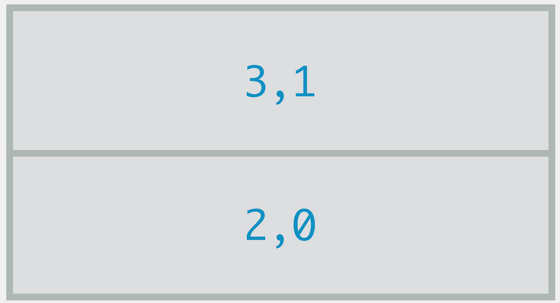
Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3



myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3

myStack

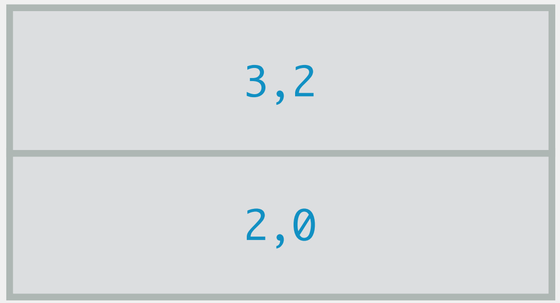
Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3



myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	4,2	4,3

myStack

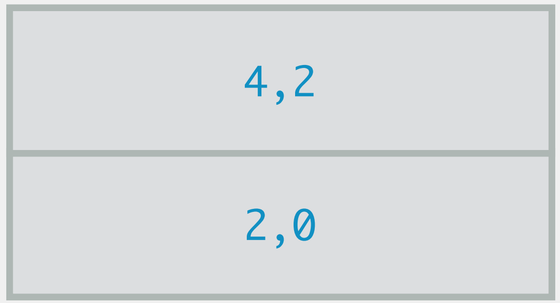
Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	4,2	4,3



myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	4,3

myStack

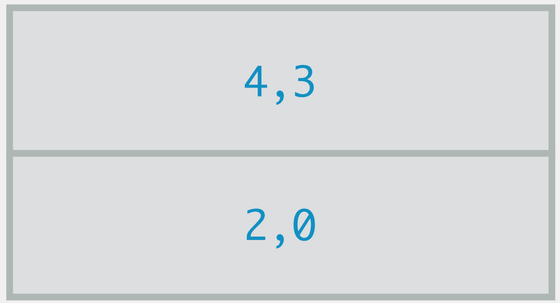
Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	4,3



myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

myStack

Depth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

myStack

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Depth-First Search

Add each neighbor to the stack

Once all neighbors are added, pop the top of the stack and explore that node

Will continue down a path until their are no neighbors to add

Order visited: 2,1 3,1 3,2 4,2 4,3 2,0 1,0 0,0

Breadth-First Search

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

`myQueue`

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

```
2,0
3,1
```

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

```
3,1
```

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

3,1
1,0

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

`1,0`

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

1,0
3,2

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

```
3,2
```

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	3,2	3,3
4,0	4,1	4,2	4,3

myQueue

```
3,2
0,0
```

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	4,2	4,3

myQueue

```
0,0
```

Add each neighbor to the queue

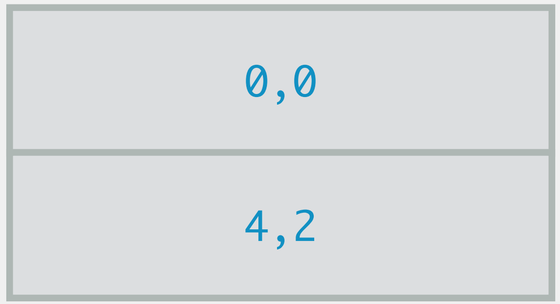
Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	4,2	4,3

myQueue



Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

myQueue

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	4,3

`myQueue`

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	4,3

myQueue

```
4,3
```

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**2,1**	2,2	2,3
3,0	**3,1**	**3,2**	3,3
4,0	4,1	**4,2**	**4,3**

myQueue

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Breadth-First Search

Add each neighbor to the queue

Once all neighbors are added, poll the front of the queue and explore that node

Will continue exploring levels of neighbors until their are no neighbors to add

Order visited: 2,1 2,0 3,1 1,0 3,2 0,0 4,2 4,3