PA2 due today

PA1 grades released
to regrade request Gradescope

PA1 late/Resubit → Google Form

Exam 1 → released Thursday @ 6pm

Queue Implementation

One option: Implement the methods in the ADT from scratch.

```
public class MyQueue<E>{
  Object[] data;
  int size;
  int front;
  int back;
  public boolean enqueue(E elem){
       //if full resize
       //change front
       //add in the new element
  }
}
```

Another option:
Lazy Greg needs to implement the Queue Interface with the following methods:

- void enqueue(E element) – add elements to the back of the queue.
- E dequeue() – remove element from front of the queue.
- int size() – return the size of the queue.

Let's see what he can do to be as lazy as possible.

Greg has access to a data structure implementation that supports the following methods.

- void add(int index, E value)
- E remove(int index)
- int size()

Let's call this data structure .....ArrayList

Inheritance?

Greg realizes that he can just make Queue extend the ArrayList and write the additional methods by using other existing methods.

```
public Queue<E> extends ArrayList<E>
{
  …
  public E dequeue() {
    E toReturn = this.contents.get(0);
    this.contents.remove(0);
    return toReturn;
  }
  …
}
```

Such an implementation comes with strings attached.

The other methods in the Queue are public and accessible by anyone. But a Queue does not expose such methods!

So Inheritance is not the best design pattern to use here.

A queue has-a ArrayList!

Adapter Pattern

Making the ArrayList variable private makes sure that users of the Queue cannot access the ArrayList or its methods.

Only the Queue methods are public and therefore usable by clients.

You can happily use ArrayList within Queue and pass on operations to it.

Adapter Pattern

```
public class Queue<E> implements QueueInterface<E> {
  private ArrayList<E> container;
  …
  public void enqueue(E element) {
    this.contents.add(this.contents.size(), element);
  }
}
```

This is called '**delegation**'. The enqueue method of Queue is delegating the task to add method of ArrayList

And no one needs to know..

Everyone thinks I implemented Queue from scratch

Implementing Stack

Mapping Attributes
Before deciding on what methods to use, one needs to map the corresponding attributes.

For example: To use the ArrayList as a Stack, we need to map the Top of the stack to some position in the list (front or back—our choice, but how to choose?)

Once this is done, we can map the methods on top of the stack to methods operating on the back of the List.

If we choose the front…

- push -> add
- pop -> remove
- peek -> get

Adapter Pattern Summary

You would like to implement an Interface A.

You have an implementation B that implements another interface C which defines methods very much similar to the methods in A but differ slightly (like name).

You use an instance of B inside your class that implements A and delegate tasks to it.

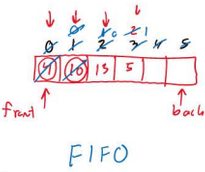
Your class A "has a" class B.

- This is called composition

A **queue** has two operations, **enqueue** and **dequeue**.

Enquing adds an element to the **back** of the queue, and **dequeue** removes the **front** element and returns it.

```
Queue<Integer> q = new ALQueue<>();
q.enqueue(4);
q.enqueue(10);
q.enqueue(13);
Integer i = q.dequeue();
q.enqueue(5);
Integer i2 = q.dequeue();
```



What number is stored in i?

4

What number is stored in i2?

10

FIFO

What is the contents of the queue? (starting at the **front**)

front → [13, 5] ← back

```
import java.util.ArrayList;

public interface Queue<E> {
  void enqueue(E element);
  E dequeue();
  int size();
}

class ALQueue<E> implements Queue<E> {

        ArrayList<E> contents;

        ALQueue() {
            this.contents = new ArrayList<>();
        }

        @Override
        public void enqueue(E element) {
            this.contents.add(element);
        }

        @Override
        public E dequeue() {
            E temp = this.contents.remove(0);
            return temp;
        }

        @Override
        public int size() {
            return this.contents.size();
        }
}
```

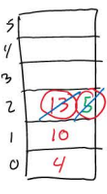
```
interface Queue<E> {
    void enqueue(E element);
    E dequeue();
    int size();
    E peek();
}
```

```
@Override
public E peek() {
    E temp = this.contents.get(0);
    return temp;
}

@Override
public String toString() {
    return "front -> " + this.contents.toString() + " <- back";
}
```

A **stack** has two operations, **push** and **pop**.

Pushing adds an element to the **top** of the stack, and **pop** removes the **top** element and returns it.

```
Stack<Integer> s = new ALStack<>();
s.push(4);
s.push(10);
s.push(13);
Integer i = s.pop();
s.push(5);
Integer i2 = s.pop();
```



What number is stored in i?

13

What number is stored in i2?

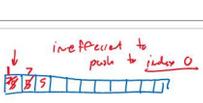
5

What is the contents of the stack? (starting at the top)

top → [10, 4]

```
import java.util.ArrayList;

public interface Stack<E> {
  void push(E element);
  E pop();
  int size();
}

class ALStack<E> implements Stack<E>


        ArrayList<E> contents;

        ALStack() {
            this.contents = new ArrayList<>();
        }

        public void push(E element) {
            this.contents.add(element);
        }

        public E pop() {
            E temp = this.contents.remove(this.contents.size() - 1);
            return temp;
        }

        public int size() {
            return this.contents.size();
        }

}
```

```
public E peek() {
    E temp = this.contents.get(this.contents.size() - 1);
    return temp;
}

@Override
public String toString() {
    return this.contents.toString() + " <- top";
}
```



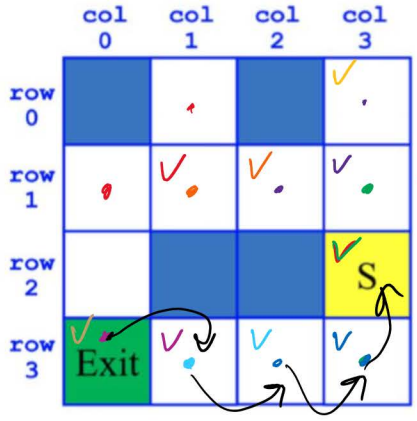
if index == 0 → this.contents.add(0, element);
this.remove(0)

Breadth-First Search (BFS) Queue

Guaranteed shortest path

```
class Square {
  boolean visited;
  Square previous;
}
```

class Node {
boolean visited
Next next
}




```
SearchForTheExit

Initialize a Queue to hold Squares as we search
Mark starting square as visited
Enqueue starting square on Queue
While Queue is not empty
    Dequeue square sq from Queue
    Mark sq as visited
    If sq is the Exit, we're done!
    For each of square's unvisited neighbors (S, W, N, E):
        Set neighbor's previous to sq
        Enqueue neighbor to Queue
```

path?

Run through the SearchForTheExit algorithm. Draw the queue.

front → (2,3) (3,3) (1,3) (3,2) (1,2) (0,3) (3,1) (1,1) (3,0) (1,0) (0,1)

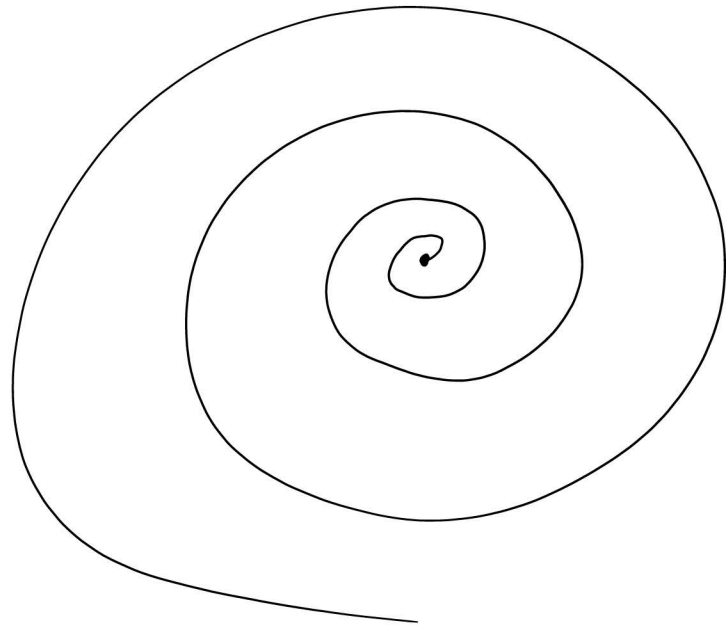
(3,0) → (3,1) → (3,2) → (3,3) → (2,3)

exit start

How many nodes were visited? 9

How many total squares were added to the queue? 11

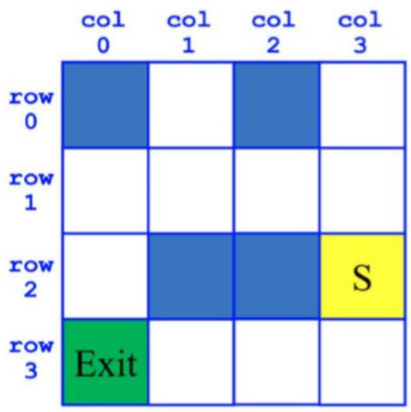
Was this the shortest path? true



Depth-First Search (DFS)

Will always find a path. Possibly faster.

```
class Square {
  boolean visited;
  Square previous;
}
```




```
SearchForTheExit

Initialize a Stack to hold Squares as we search
Mark starting square as visited
Push starting square on Stack
While Stack is not empty
    Pop square sq from Stack
    Mark sq as visited
    If sq is the Exit, we're done!
    For each of square's unvisited neighbors (S, W, N, E):
        Set neighbor's previous to sq
        Push neighbor to Stack
```

Run through the SearchForTheExit algorithm. Draw the stack.

How many nodes were visited?

How many total squares were added to the stack?

Was this the shortest path?