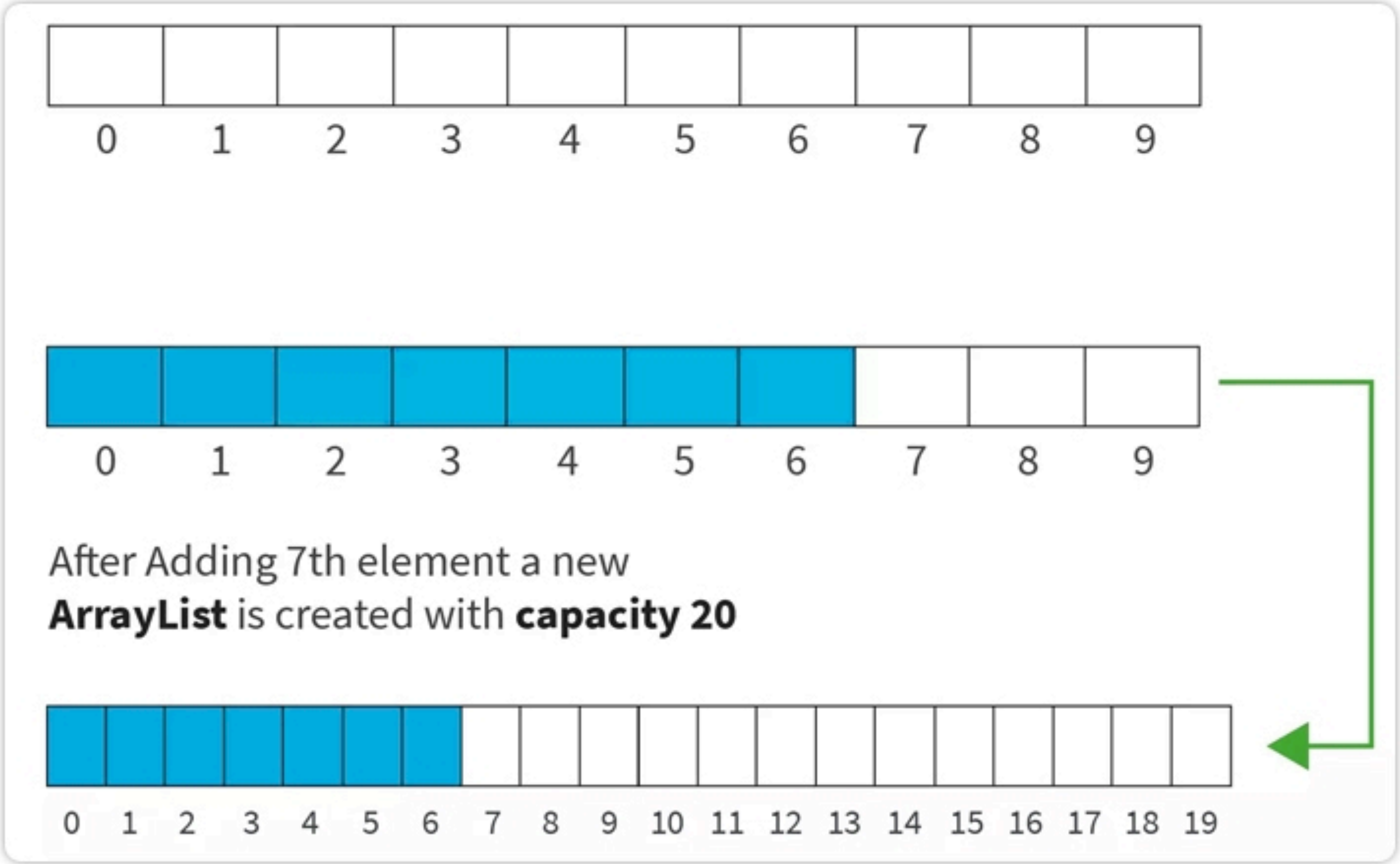
CS62 Class 6: ArrayLists

Basic Data Structures



from https://www.janbasktraining.com/blog/java-print-arraylist/

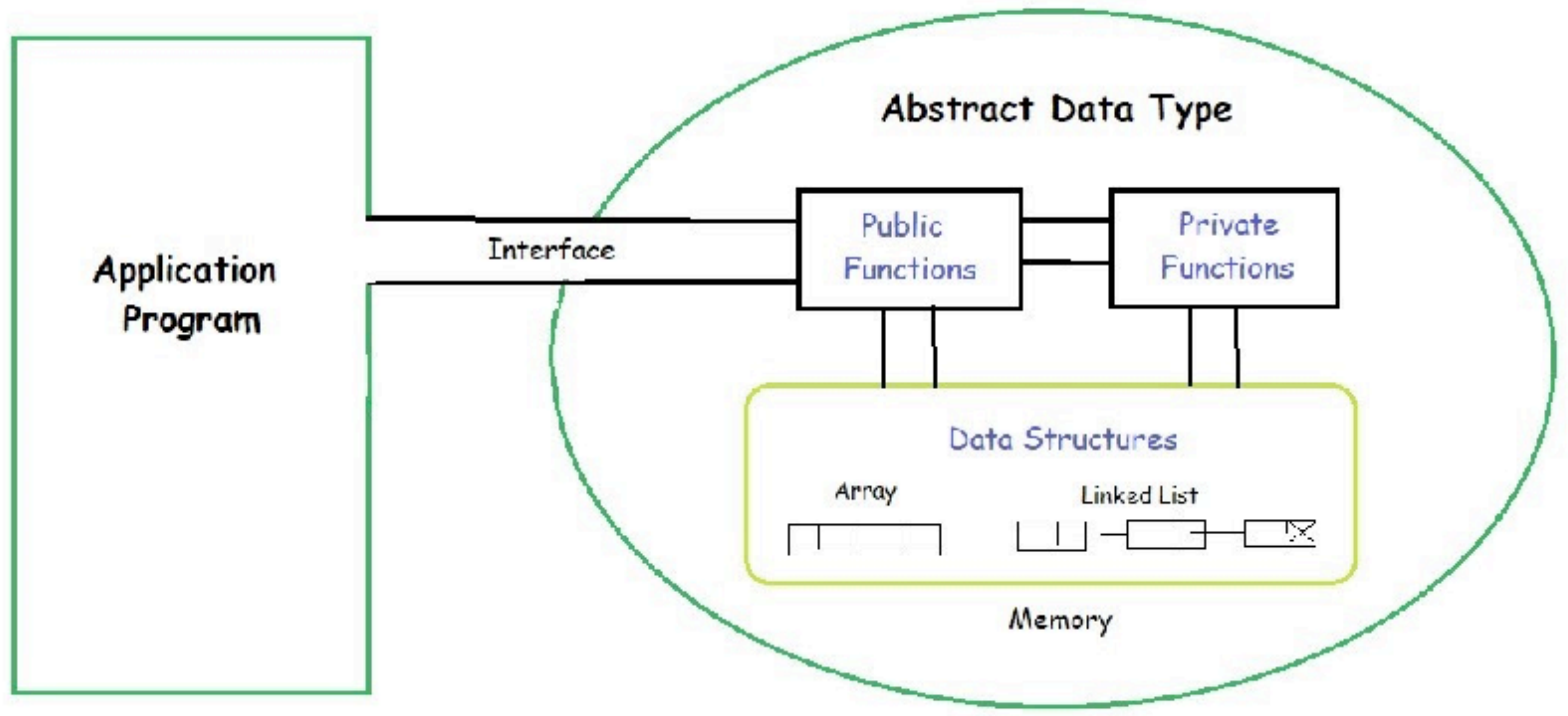
Lecture 6 agenda

- Data structures in general
  - What are abstract data types?
  - Why do we care + history
- ArrayLists behavior
- ArrayLists implementation

Data structures in general

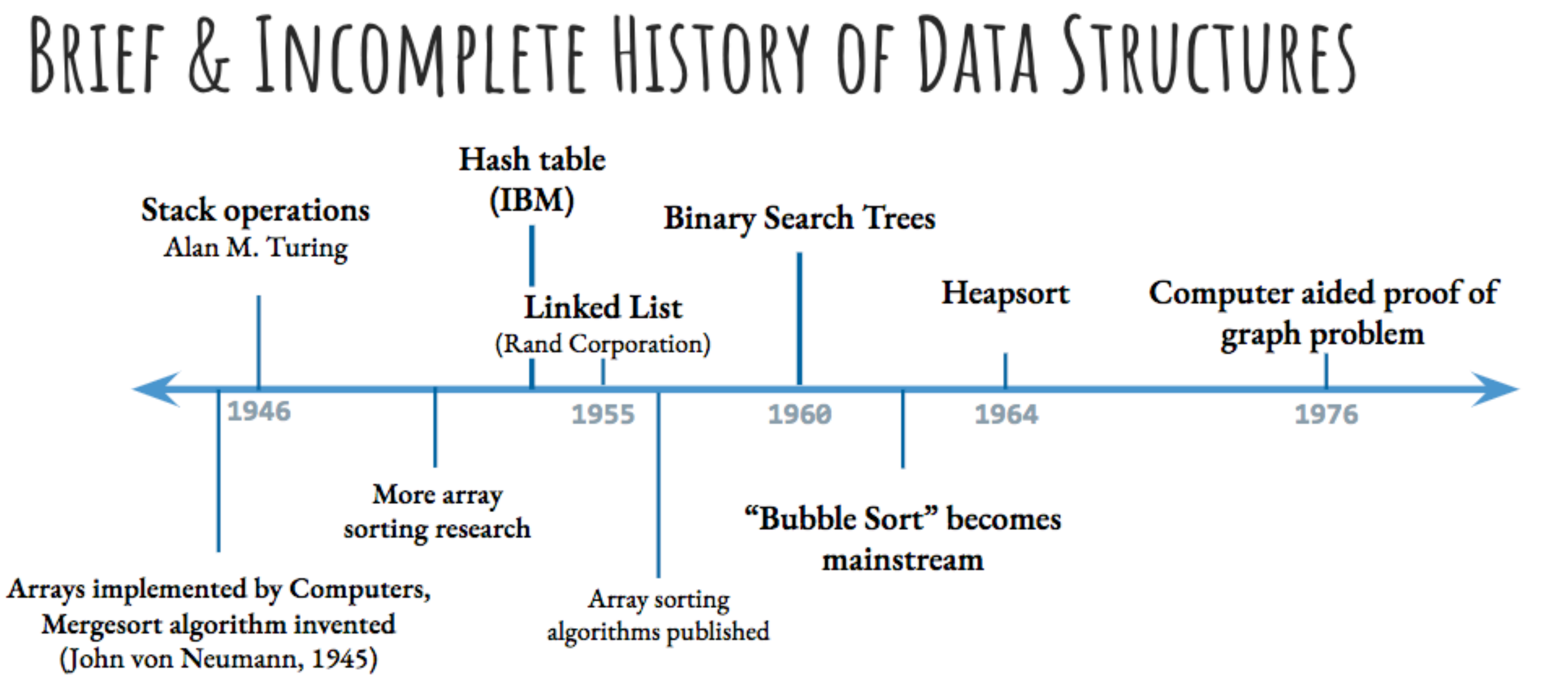
Abstract data type

- Sometimes, you'll hear specific data structures be referred to as "abstract data types". This is a *conceptual model* where users know the behavior of a data structure, but not exactly how it's implemented.
  - How it's implemented = where in memory things are, what algorithms are used…
- For example, the idea of a "list" is an abstract data type. We know we can add, remove, resize a list, but how exactly that happens is not important.
  - ArrayLists, Singly Linked Lists, Doubly Linked Lists are **not** abstract: we have to implement them.
- Basically, the same thing as an interface!
  - …which is also kind of the same thing as an API.



https://www.geeksforgeeks.org/abstract-data-types/

Data structures history



BRIEF & INCOMPLETE HISTORY OF DATA STRUCTURES
Stack operations
Alan M. Turing
Hash table
(IBM)
Linked List
(Rand Corporation)
Binary Search Trees
Heapsort
Computer aided proof of graph problem
1946
1955
1960
1964
1976
More array sorting research
Arrays implemented by Computers, Mergesort algorithm invented (John von Neumann, 1945)
Array sorting algorithms published
"Bubble Sort" becomes mainstream

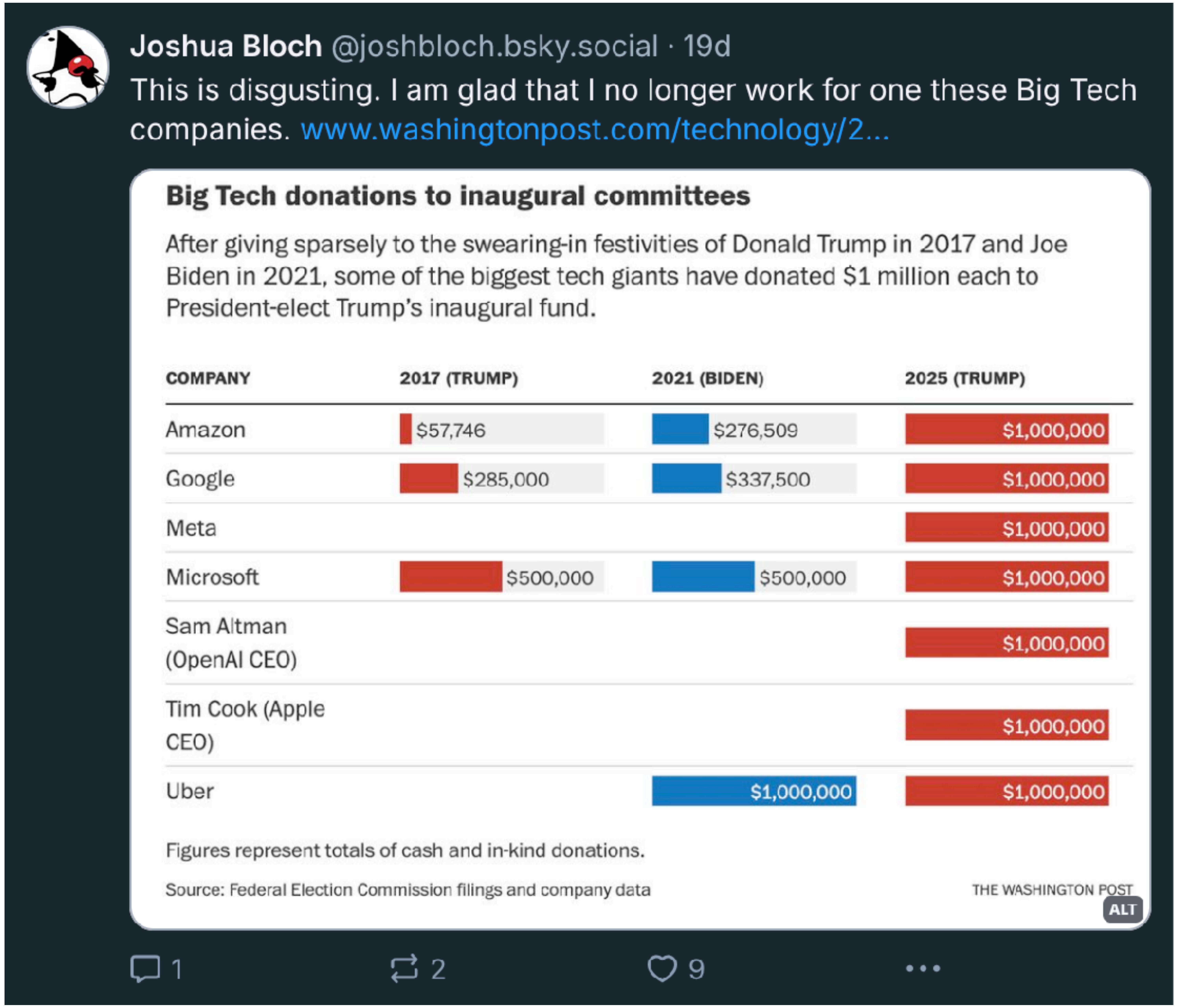
"Tree" was coined in 1857 by British mathematician Arthur Cayley (link)
The origin of graph theory was 1736 on the Bridges of Königsberg (link)

- Lots of concepts in CS came from math (trees, graphs) and have been around for a while
- The invention of many data structures coinciding with World War II is no coincidence
- ArrayLists are not one of these: they are a relatively modern implementation (1998), even though arrays have been around since the 1940s

https://macbookandheels.com/algorithm/2018/10/31/teachingds/

Joshua Bloch: main Java Collections Framework architect

- The person who made ArrayLists isn't some long dead historic figure; he was an engineer at Sun Microsystems (the company that made Java) but now teaches at CMU
- He's currently pretty politically active on BlueSky



Joshua Bloch @joshbloch.bsky.social · 19d
This is disgusting. I am glad that I no longer work for one these Big Tech companies. www.washingtonpost.com/technology/2...

Big Tech donations to inaugural committees

After giving sparsely to the swearing-in festivities of Donald Trump in 2017 and Joe Biden in 2021, some of the biggest tech giants have donated $1 million each to President-elect Trump's inaugural fund.

COMPANY	2017 (TRUMP)	2021 (BIDEN)	2025 (TRUMP)
Amazon	$57,746	$276,509	$1,000,000
Google	$285,000	$337,500	$1,000,000
Meta			$1,000,000
Microsoft	$500,000	$500,000	$1,000,000
Sam Altman (OpenAI CEO)			$1,000,000
Tim Cook (Apple CEO)			$1,000,000
Uber		$1,000,000	$1,000,000

Figures represent totals of cash and in-kind donations.

Source: Federal Election Commission filings and company data

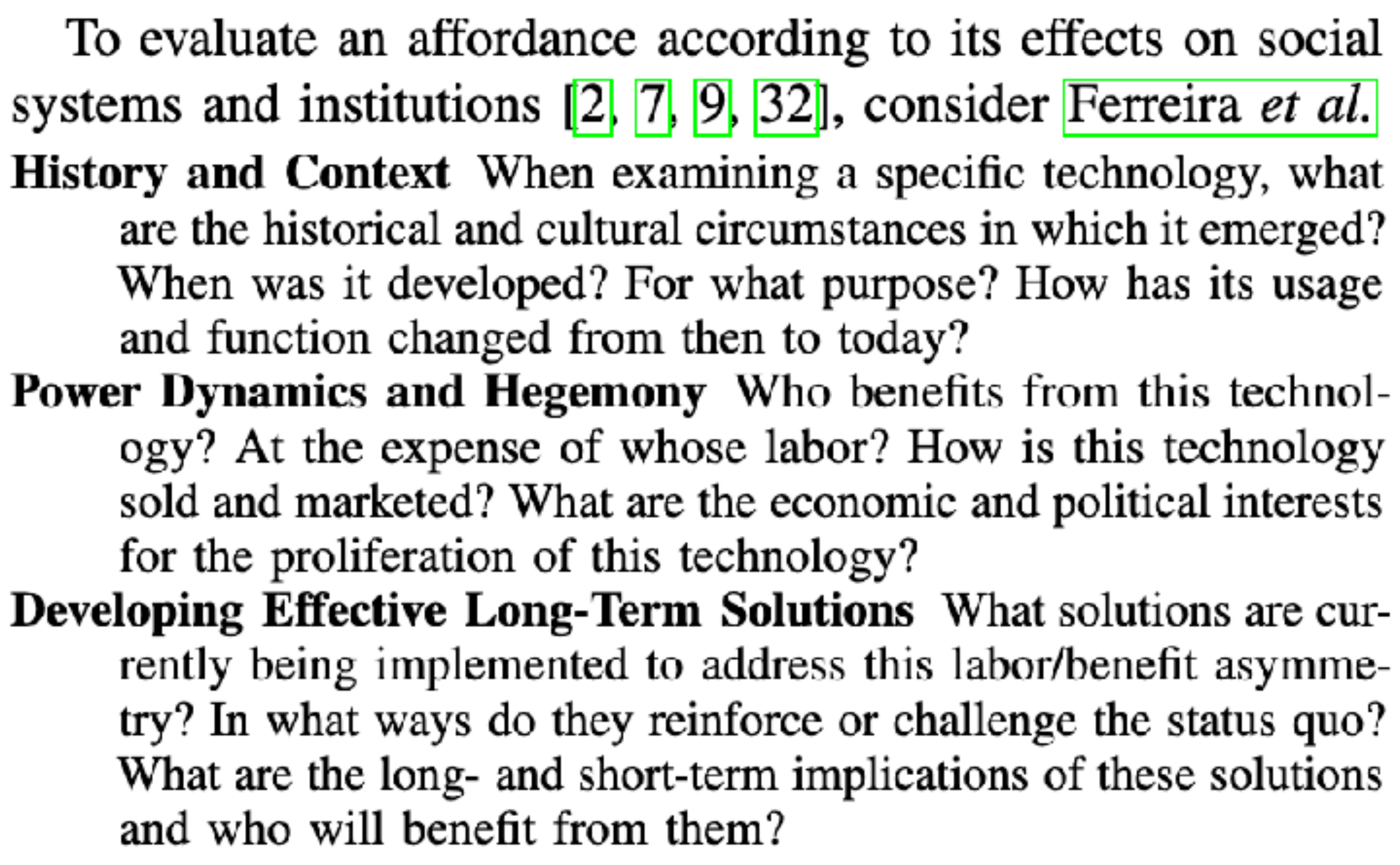
THE WASHINGTON POST

ALT

1 2 9 •••

My (liberal arts-y) wish for you: know history

- It’s one thing to know how data structures work (most data structures courses)
- But it’s another to know when to appropriately and ethically how to use them and understand their history
- STS (science technology society) scholar Bruno Latour calls this “opening the black box of science” - a real person had to create everything we use in this class to solve real problems, and understanding this history gives you a deeper insight into how technology is used by society (and how it can encode bias, etc.)
- Hopefully, this knowledge will equip you for your final project



To evaluate an affordance according to its effects on social systems and institutions [2, 7, 9, 32], consider Ferreira *et al.*

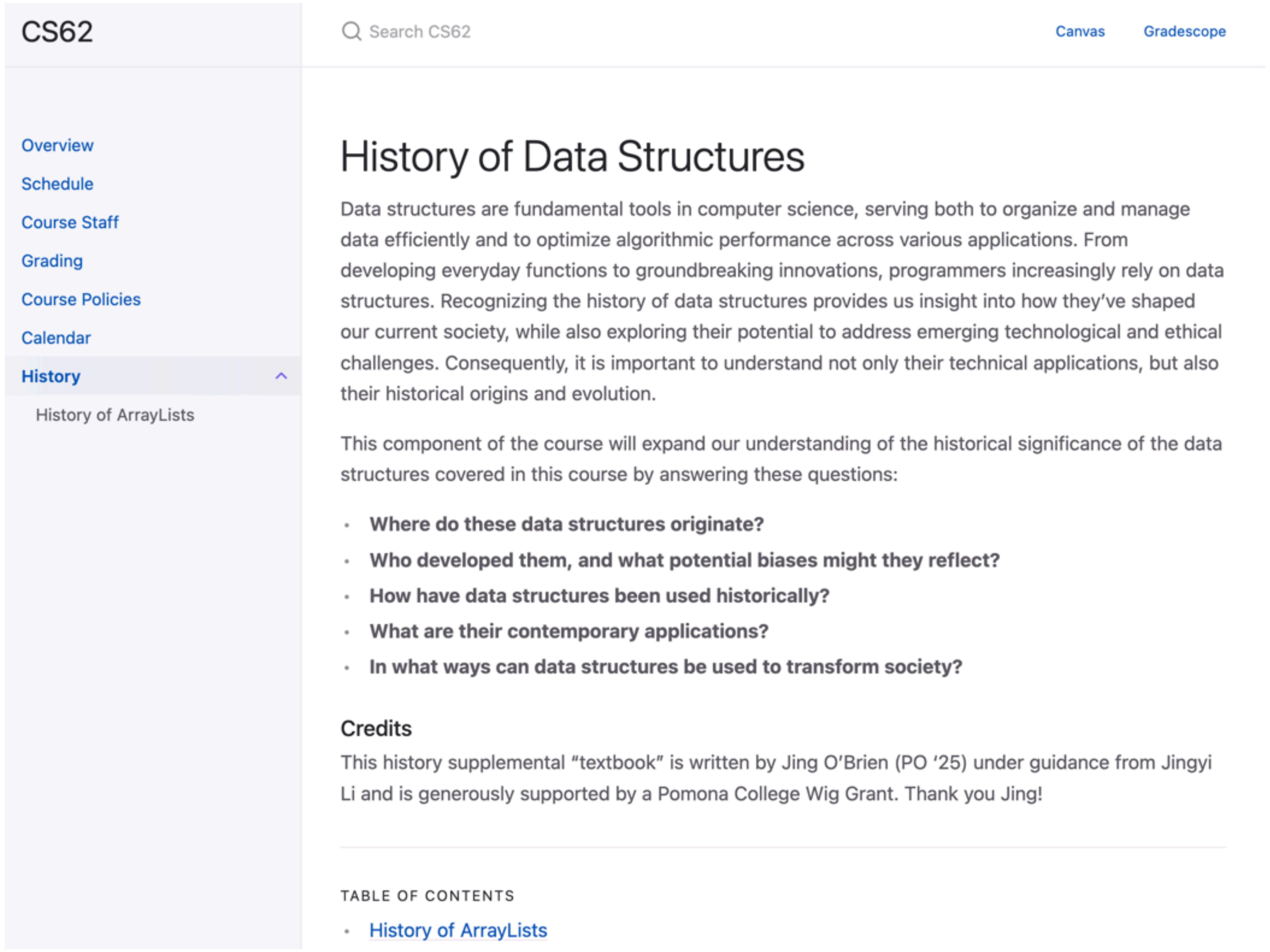
History and Context When examining a specific technology, what are the historical and cultural circumstances in which it emerged? When was it developed? For what purpose? How has its usage and function changed from then to today?

Power Dynamics and Hegemony Who benefits from this technology? At the expense of whose labor? How is this technology sold and marketed? What are the economic and political interests for the proliferation of this technology?

Developing Effective Long-Term Solutions What solutions are currently being implemented to address this labor/benefit asymmetry? In what ways do they reinforce or challenge the status quo? What are the long- and short-term implications of these solutions and who will benefit from them?

See https://kevinl.info/do-abstractions-have-politics/

Thus, we have a history textbook!



History of Data Structures

Data structures are fundamental tools in computer science, serving both to organize and manage data efficiently and to optimize algorithmic performance across various applications. From developing everyday functions to groundbreaking innovations, programmers increasingly rely on data structures. Recognizing the history of data structures provides us insight into how they've shaped our current society, while also exploring their potential to address emerging technological and ethical challenges. Consequently, it is important to understand not only their technical applications, but also their historical origins and evolution.

This component of the course will expand our understanding of the historical significance of the data structures covered in this course by answering these questions:

- **Where do these data structures originate?**
- **Who developed them, and what potential biases might they reflect?**
- **How have data structures been used historically?**
- **What are their contemporary applications?**
- **In what ways can data structures be used to transform society?**

Credits

This history supplemental "textbook" is written by Jing O'Brien (PO '25) under guidance from Jingyi Li and is generously supported by a Pomona College Wig Grant. Thank you Jing!

TABLE OF CONTENTS

- My RA Jing O'Brien is doing research on the history of data structures as we go along. Read the pages to supplement learning purely "technical" material!

https://cs.pomona.edu/classes/cs62/history/

Why do we need data structures?

- To organize and store data so that we can perform efficient operations on them based on our needs: imagine walking to an unorganized library and trying to find your favorite title or books from your favorite author.
- We can define efficiency in different ways.
  - Time: How fast can we perform certain operations on a data structure?
  - Space: How much memory do we need to organize our data in a data structure?
  - Affordances: How does this data structure bake in assumptions for how we should think about the problem? What can we and can we not do with it?
- There is no data structure that fits all needs.
  - That's why we're spending a semester looking at different data structures.
  - So far, the only data structure we have encountered is arrays.
- The goal of this class is for you to understand trade offs between data structures, to choose the appropriate data structure for the appropriate problem

Types of operations on data structures

- Insertion: adding a new element in a data structure.
- Deletion: Removing (and possibly returning) an element.
- Searching: Searching for a specific data element.

- Replacement: Replacing an existing element with a new one (and possibly returning old).
- Traversal: Going through all the elements.
- Sorting: Sorting all elements in a specific way.
- Check if empty: Check if data structure contains any elements.

- Not a single data structure does all these things efficiently.
- You need to know both the kind of data you have, the different operations you will need to perform on them, and any technical limitations to pick an appropriate data structure.

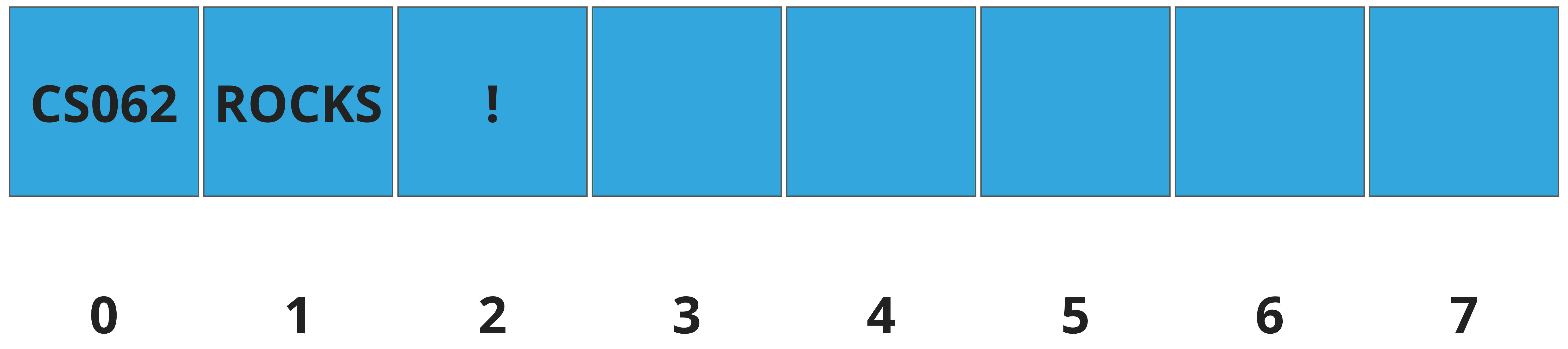
Linear vs non-linear data structures

- Linear: elements arranged in a linear sequence based on a specific order.
  - E.g., Arrays, ArrayLists, linked lists, stacks, queues.
  - Linear memory allocation: all elements are placed in a contiguous block of memory. E.g., arrays and ArrayLists.
  - Use of pointers/links: elements don’t need to be placed in contiguous blocks. The linear relationship is formed through pointers. E.g., singly and doubly linked lists.
- Non-linear: elements arranged in non-linear, mostly hierarchical relationship.
  - E.g., trees and graphs.

ArrayLists & their behaviors

ArrayLists

Remember, an ArrayList is implemented using Arrays



Capacity = 8

Size = 3

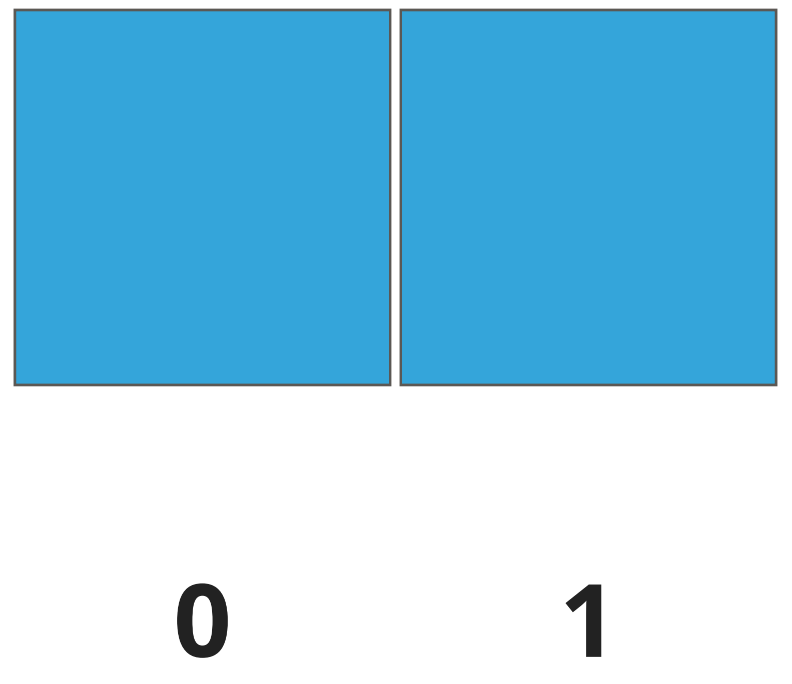
`ArrayList()`: Constructs an ArrayList

What should happen?

```
ArrayList<String> al = new ArrayList<String>();
```

`ArrayList()`: Constructs an ArrayList

When we first make an ArrayList, it is with a size 2 Array.



```
ArrayList<String> al = new ArrayList<String>();
```

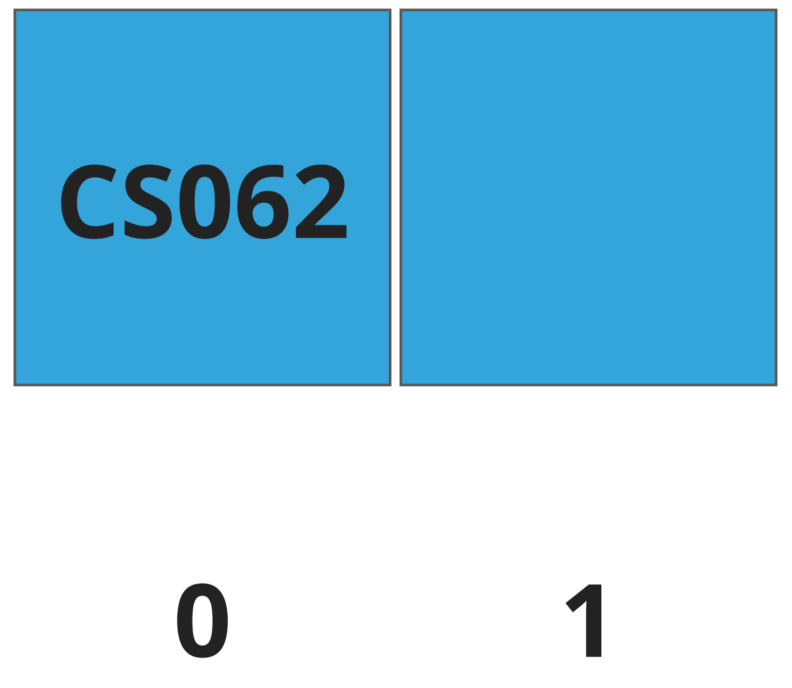
Capacity = 2

Size = 0

What should happen?

```
al.add("CS062");
```

`add(E element)`: Appends the element to the end of the ArrayList



```
al.add("CS062");
```

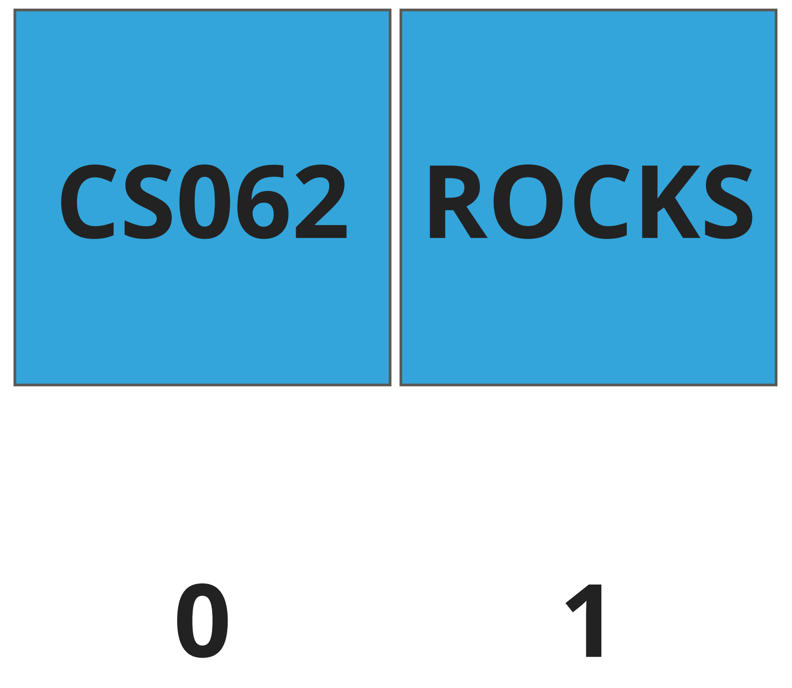
Capacity = 2

Size = 1

What should happen?

```
al.add("ROCKS");
```

`add(E element)`: Appends the element to the end of the ArrayList



```
al.add("ROCKS");
```

Capacity = 2

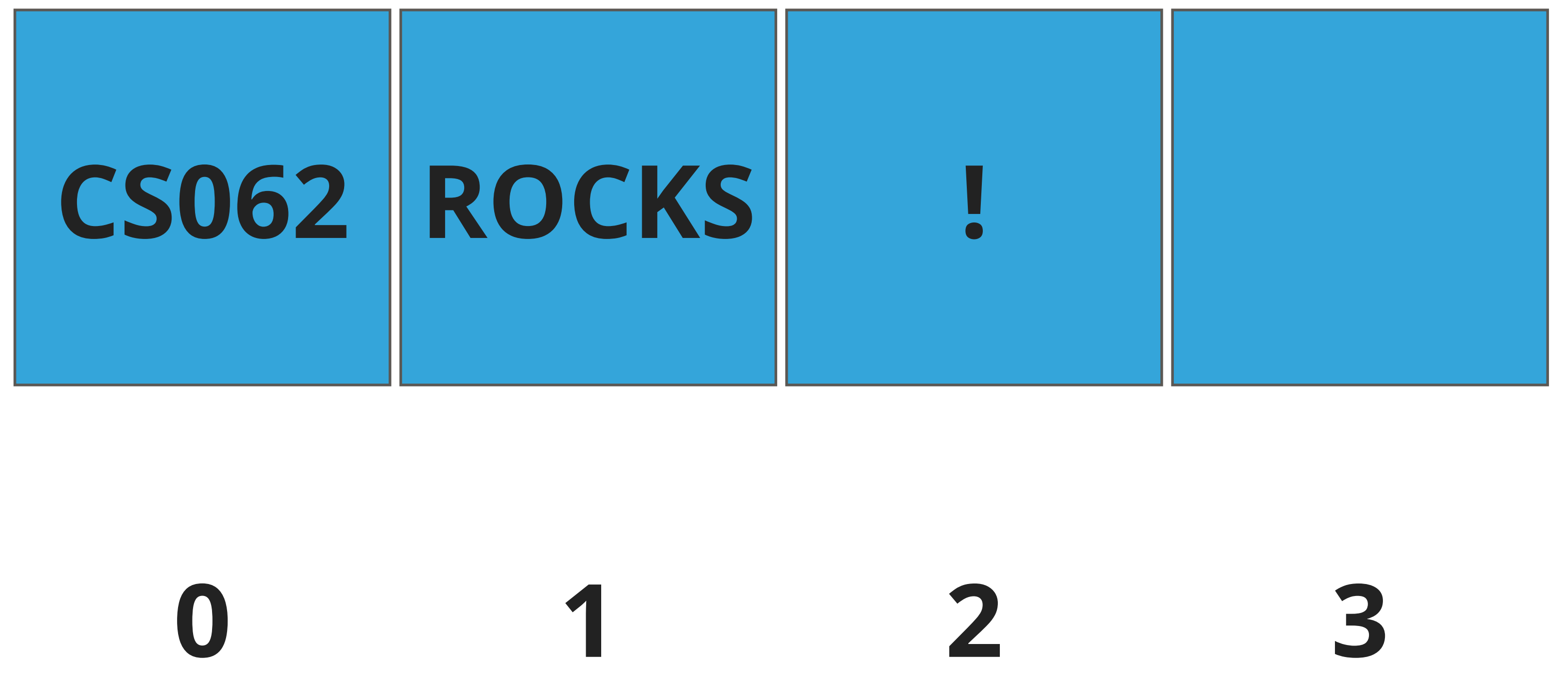
Size = 2

What should happen?

```
al.add("!");
```

`add(E element)`: **Appends the element to the end of the ArrayList**

- Double capacity since it's full and then add new element



```
al.add("!");
```

Capacity = 4

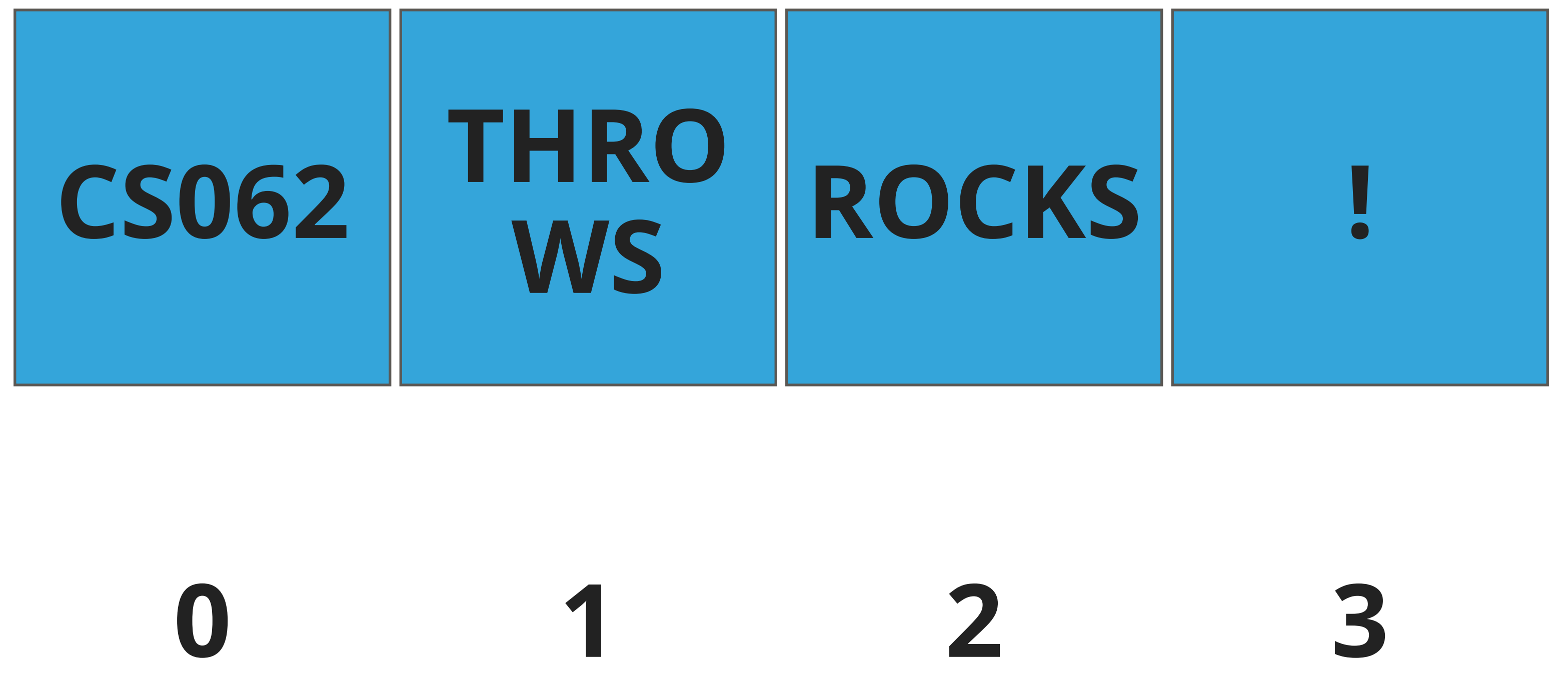
Size = 3

What should happen?

```
al.add(1, "THROWS");
```

`add(int index, E element)`: **Adds element at the specified index**

- Shift elements to the right



`al.add(1, “THROWS”);`

Capacity = 4

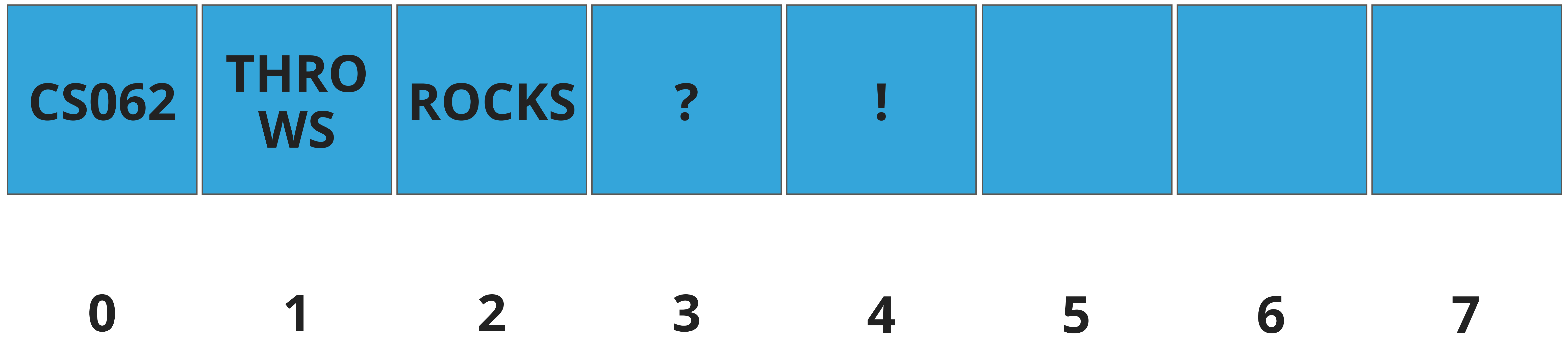
Size = 4

What should happen?

`al.add(3, “?”);`

`add(int index, E element)`: **Adds element at the specified index**

- Double capacity since full
- Shift elements to the right



Capacity = 8

Size = 5

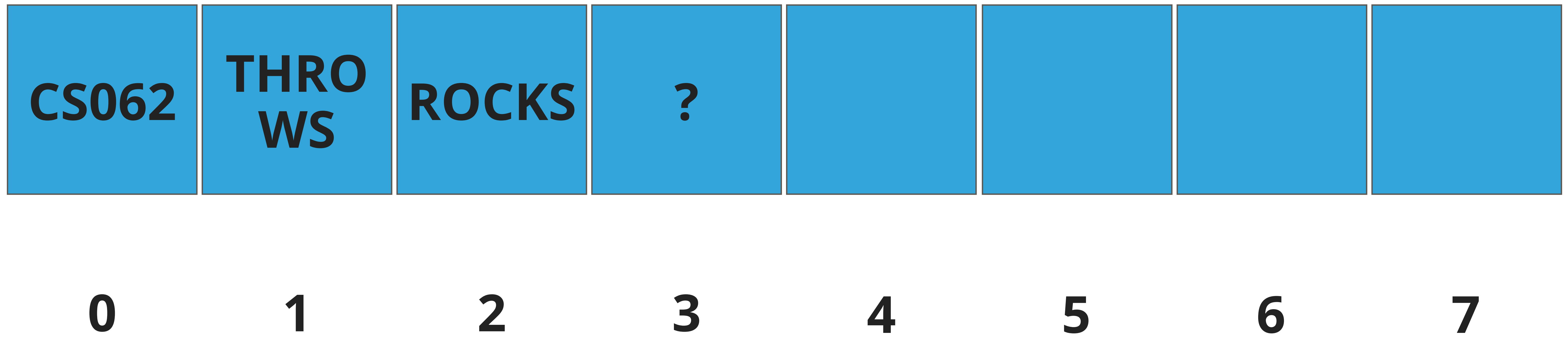
```
al.add(3, “?”);
```

What should happen?

```
al.remove();
```

`remove()`: Removes and returns element from the end of ArrayList

- Remove and Return last element



Capacity = 8

Size = 4

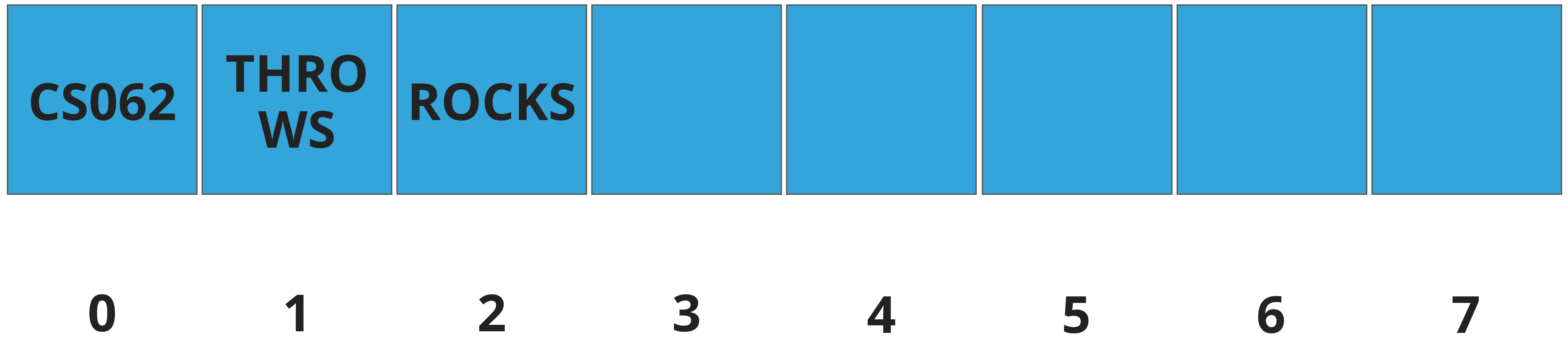
```
al.remove();
```

What should happen?

```
al.remove();
```

`remove()`: **Removes and returns element from the end of ArrayList**

- Remove and Return last element



Capacity = 8

Size = 3

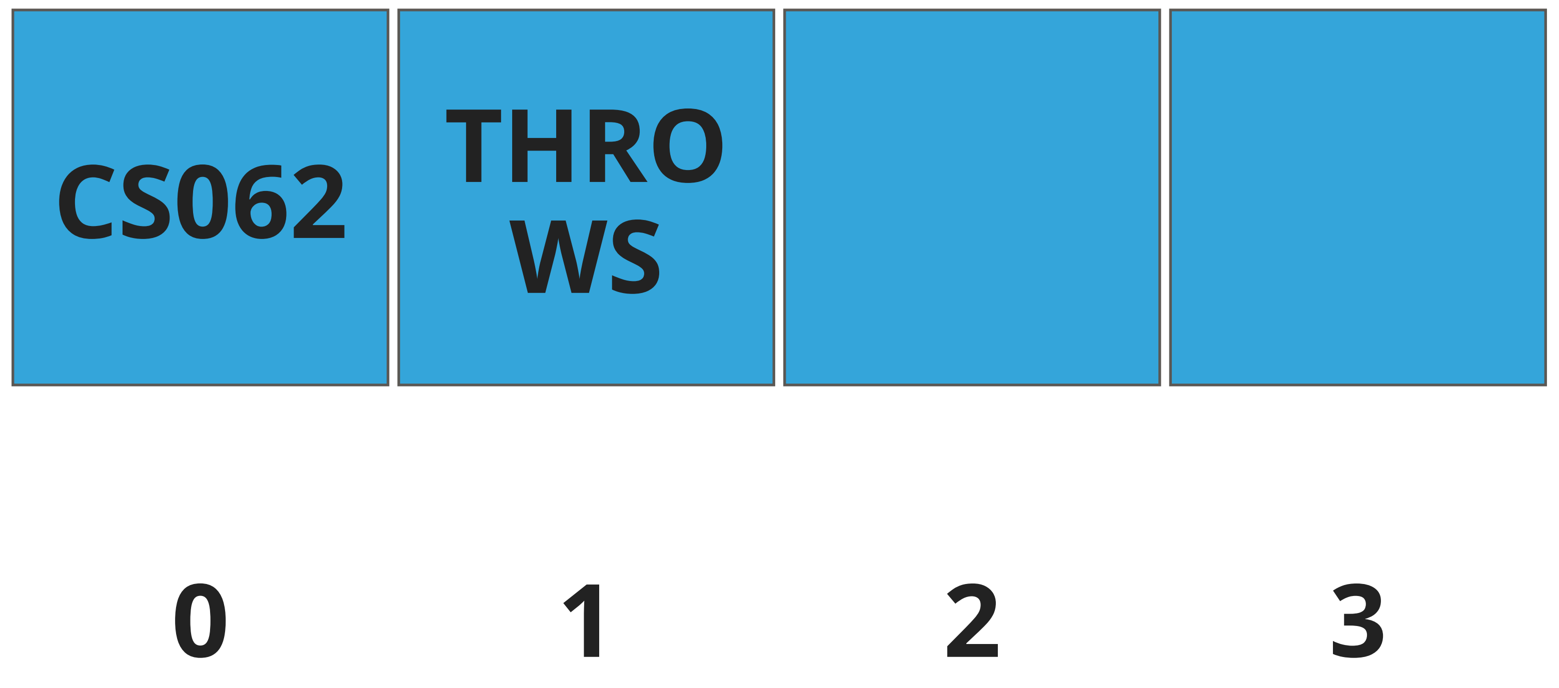
```
al.remove();
```

What should happen?

```
al.remove();
```

`remove()`: Removes and returns element from the end of ArrayList

- Remove and return last element
- Halve capacity when 1/4 full



```
al.remove();
```

Capacity = 4

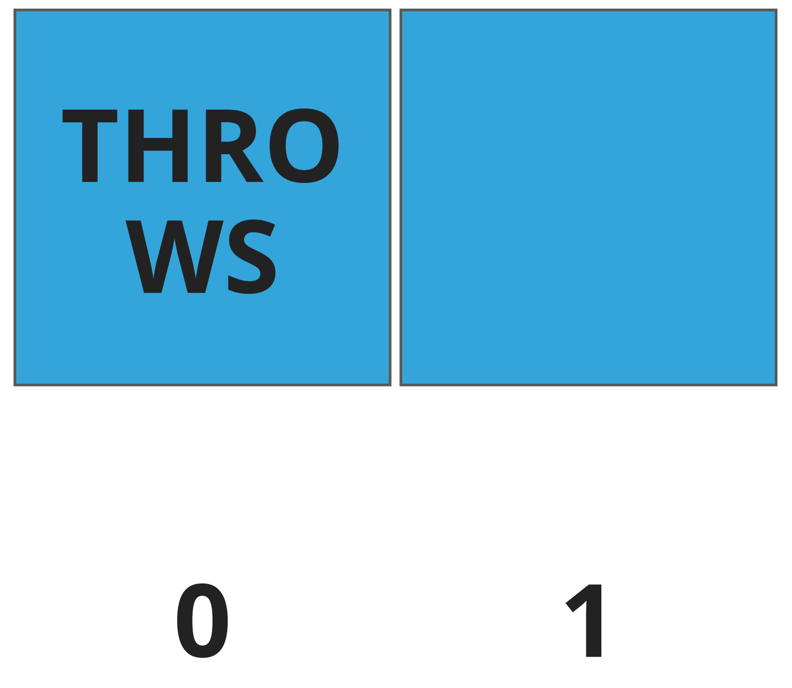
Size = 2

What should happen?

```
al.remove(0);
```

`remove(int index)`: Removes and returns element from specified index

- Remove element from index
- Shift elements to the left
- Halve capacity when 1/4th full



```
al.remove(0);
```

Capacity = 2

Size = 1

Reminder: Interface List

```java
interface List<E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

Standard Operations of `ArrayList<E>` class

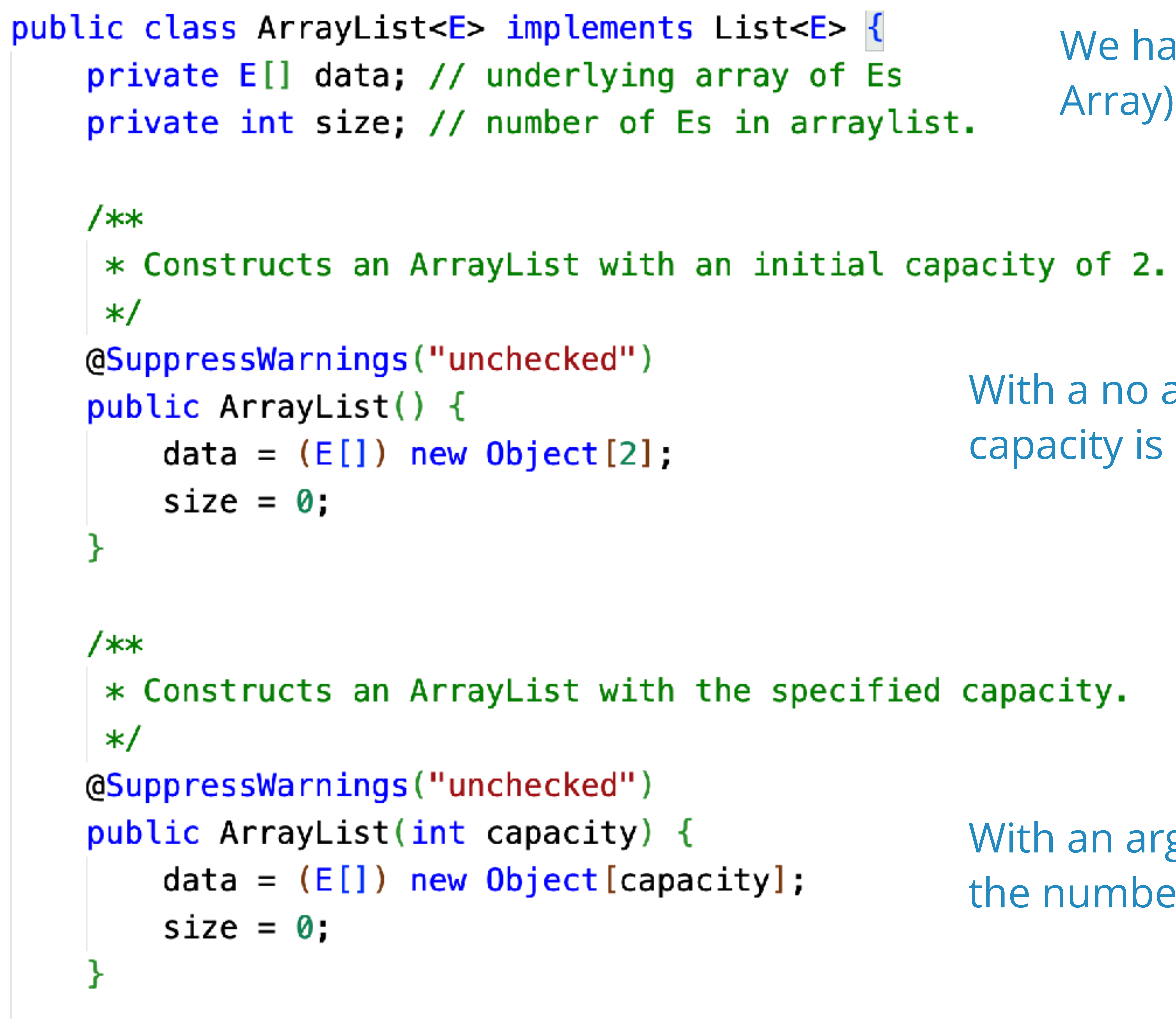
- `ArrayList()`: Constructs an empty ArrayList with an initial capacity of 2 (can vary across implementations, another common initial capacity is 10).
- `ArrayList(int capacity)`: Constructs an empty ArrayList with the specified initial capacity.
- `isEmpty()`: Returns true if the ArrayList contains no elements.
- `size()`: Returns the number of elements in the ArrayList.
- `get(int index)`: Returns the element at the specified index.
- `add(E element)`: Appends the element to the end of the ArrayList.
- `add(int index, E element)`: Inserts the element at the specified index and shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- `E remove()`: Removes and returns the element at the end of the ArrayList.
- `E remove(int index)`: Removes and returns the element at the specified index. Shifts any subsequent elements to the left (subtracts one from their indices).
- `E set(int index, E element)`: Replaces the element at the specified index with the specified element and returns the olde element.
- `clear()`: Removes all elements.

ArrayList Implementation

Our own implementation of ArrayLists

- We will implement the List interface we defined in the last lecture.
- We will work with generics because we want arrayLists to hold objects of an type.
- We will use an array and we will keep track of how many elements we have in our ArrayList.

Instance variables & constructors



```java
public class ArrayList<E> implements List<E> {
    private E[] data; // underlying array of Es
    private int size; // number of Es in arraylist.

    /**
     * Constructs an ArrayList with an initial capacity of 2.
     */
    @SuppressWarnings("unchecked")
    public ArrayList() {
        data = (E[]) new Object[2];
        size = 0;
    }

    /**
     * Constructs an ArrayList with the specified capacity.
     */
    @SuppressWarnings("unchecked")
    public ArrayList(int capacity) {
        data = (E[]) new Object[capacity];
        size = 0;
    }
```

We have 2 instance variables: our data (in an Array) and size (number of elements present)

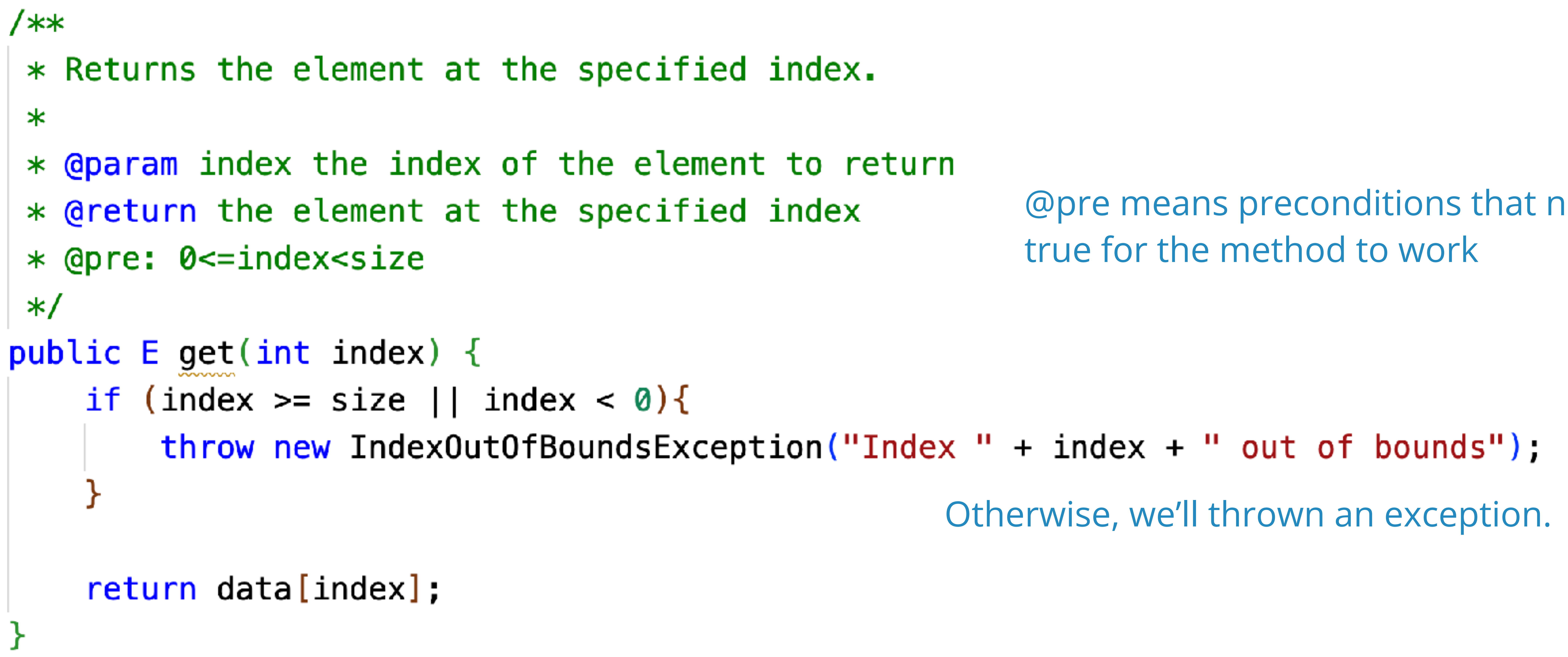
With a no argument constructor, the default capacity is 2.

With an argument, we just set the capacity to the number passed in.

isEmpty(), size()

```java
/**
 * Returns true if the ArrayList contains no Es.
 *
 * @return true if the ArrayList does not contain any E
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns the number of Es in the ArrayList.
 *
 * @return the number of Es in the ArrayList
 */
public int size() {
    return size;
}
```

Getting an element at index i



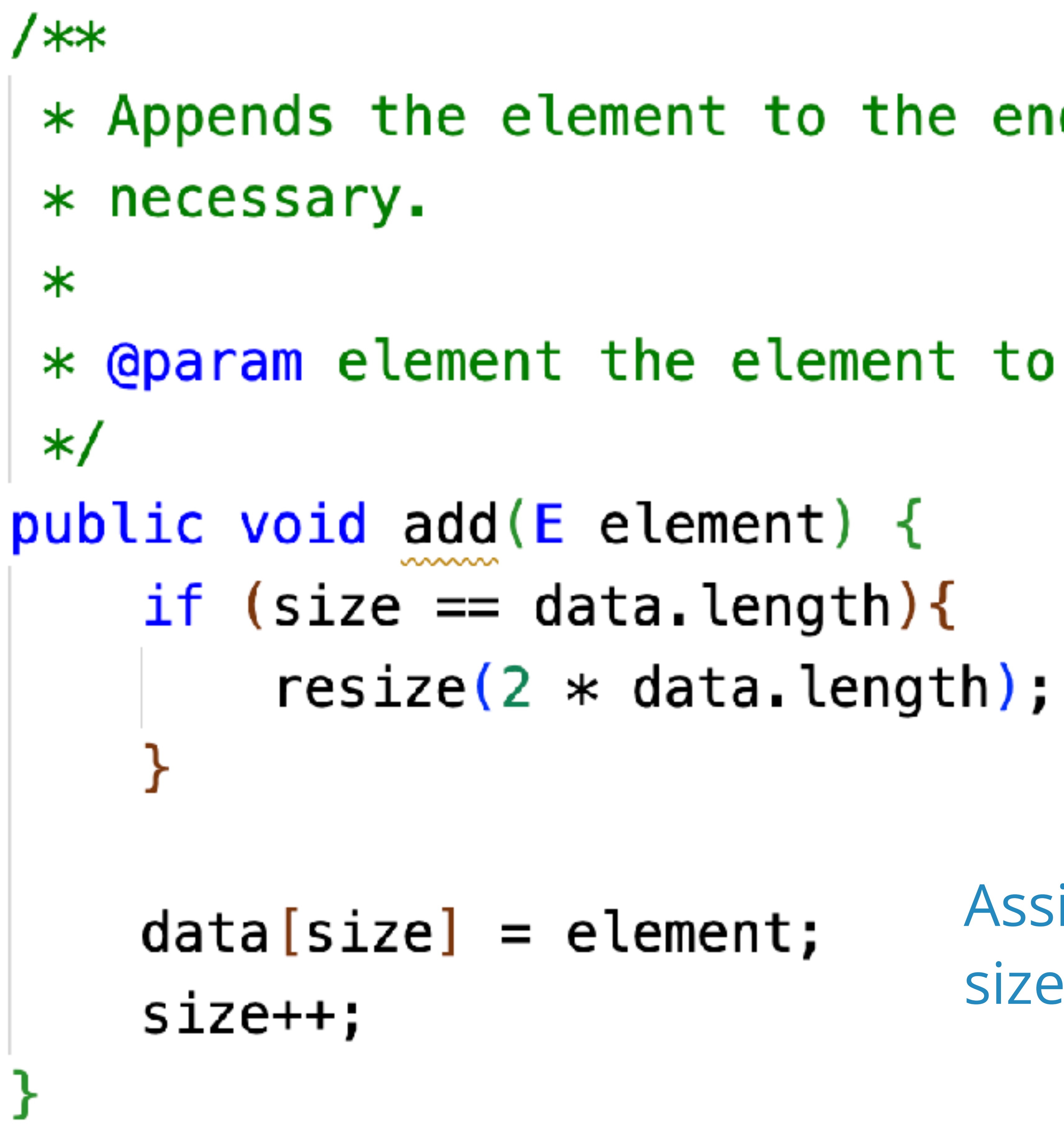
```java
/**
 * Returns the element at the specified index.
 *
 * @param index the index of the element to return
 * @return the element at the specified index
 * @pre: 0<=index<size
 */
public E get(int index) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

    return data[index];
}
```

@pre means preconditions that need to be true for the method to work

Otherwise, we'll thrown an exception.

Remember, data is just a plain old Java Array, so we access elements by indexing into it.

Adding an element to the ArrayList



```
/**
 * Appends the element to the end of the ArrayList. Doubles its capacity if
 * necessary.
 *
 * @param element the element to be inserted
 */
public void add(E element) {
    if (size == data.length){
        resize(2 * data.length);
    }

    data[size] = element;
    size++;
}
```

First, double if the Array is full

(calling a mysterious resize method...which you will write soon as practice :))

Assign element to index "size" (we know that size will always be the last empty index)

Increment size

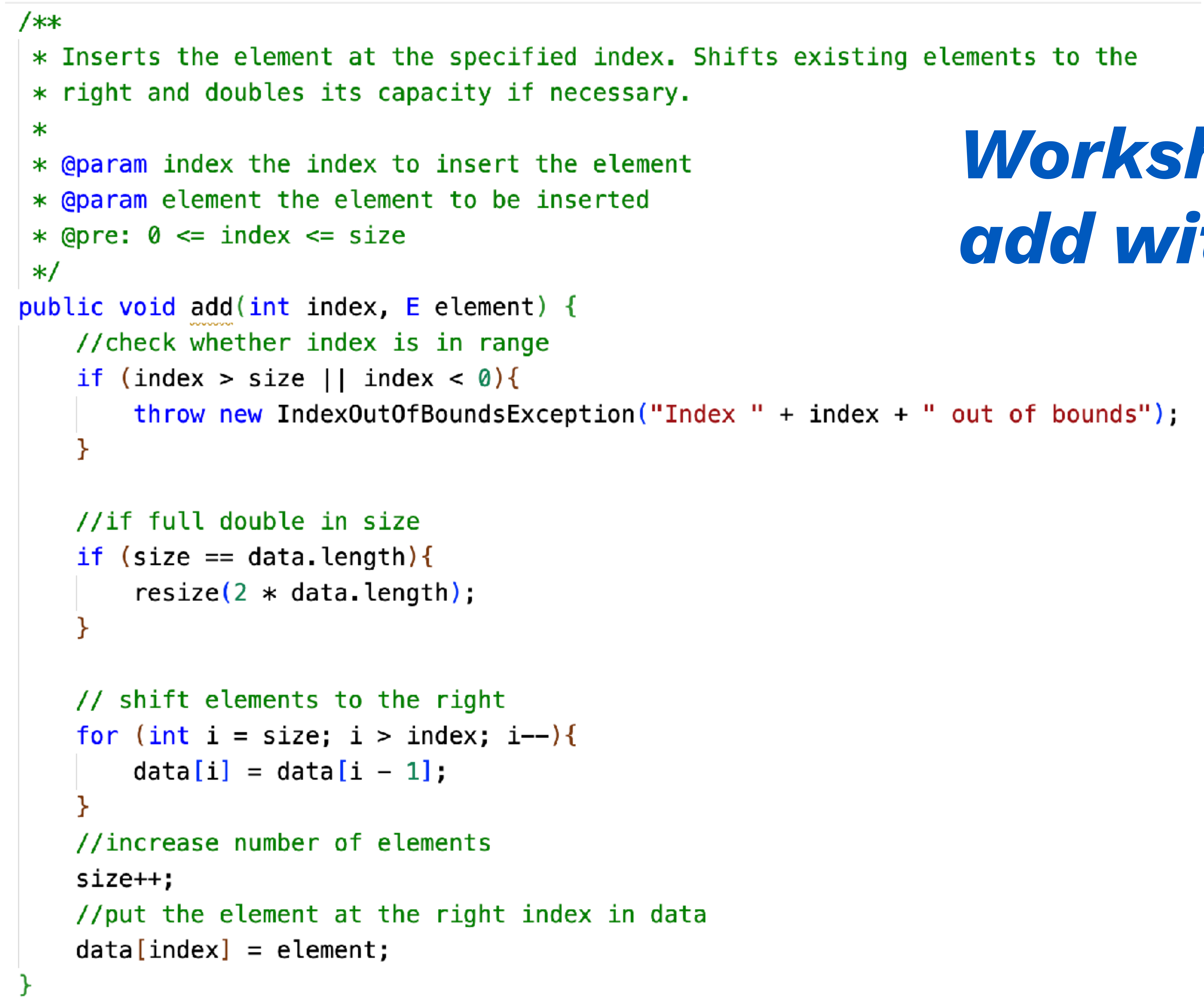
Worksheet time!

- Divide into 3 groups. Each group will write on the whiteboard together an implementation for
  - `void resize(int capacity) // resizing the arrayList to a new capacity`
  - `void add(int i, E element) // inserting element at specific index`
  - `E set(int index, E element) //replacing an element at an index, and returning the old one that was changed`
- We'll look over and correct the code together as a class. If your group finishes early, try doing the other methods together :)

Worksheet answer: resize

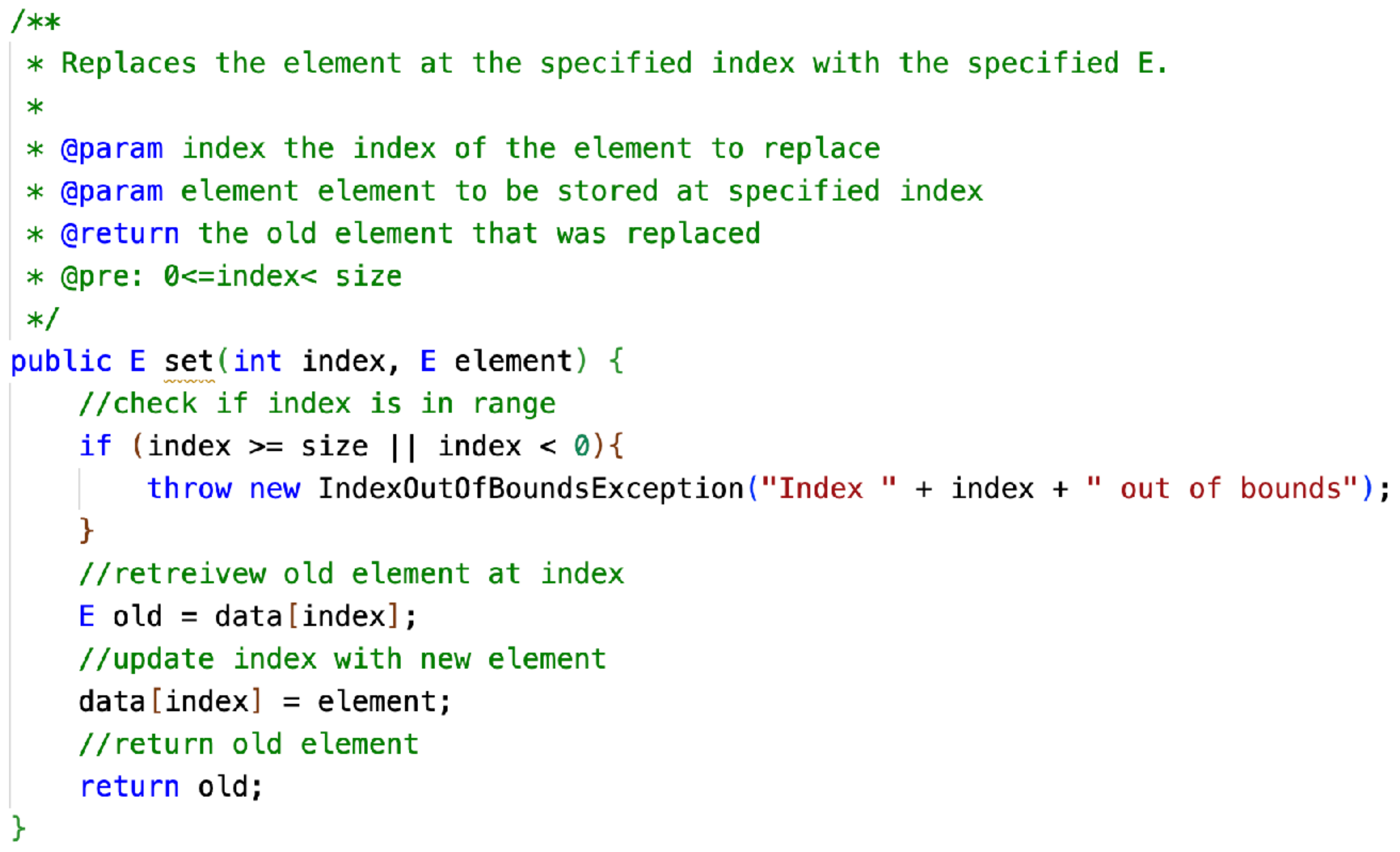
```java
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array of Es with the provided
    capacity
    E[] temp = (E[]) new Object[capacity];

    //copy all elements from old array (data) to temp array
    for (int i = 0; i < size; i++){
        temp[i] = data[i];
    }

    //point data to the new temp array
    data = temp;
}
```

Worksheet answer: add with index



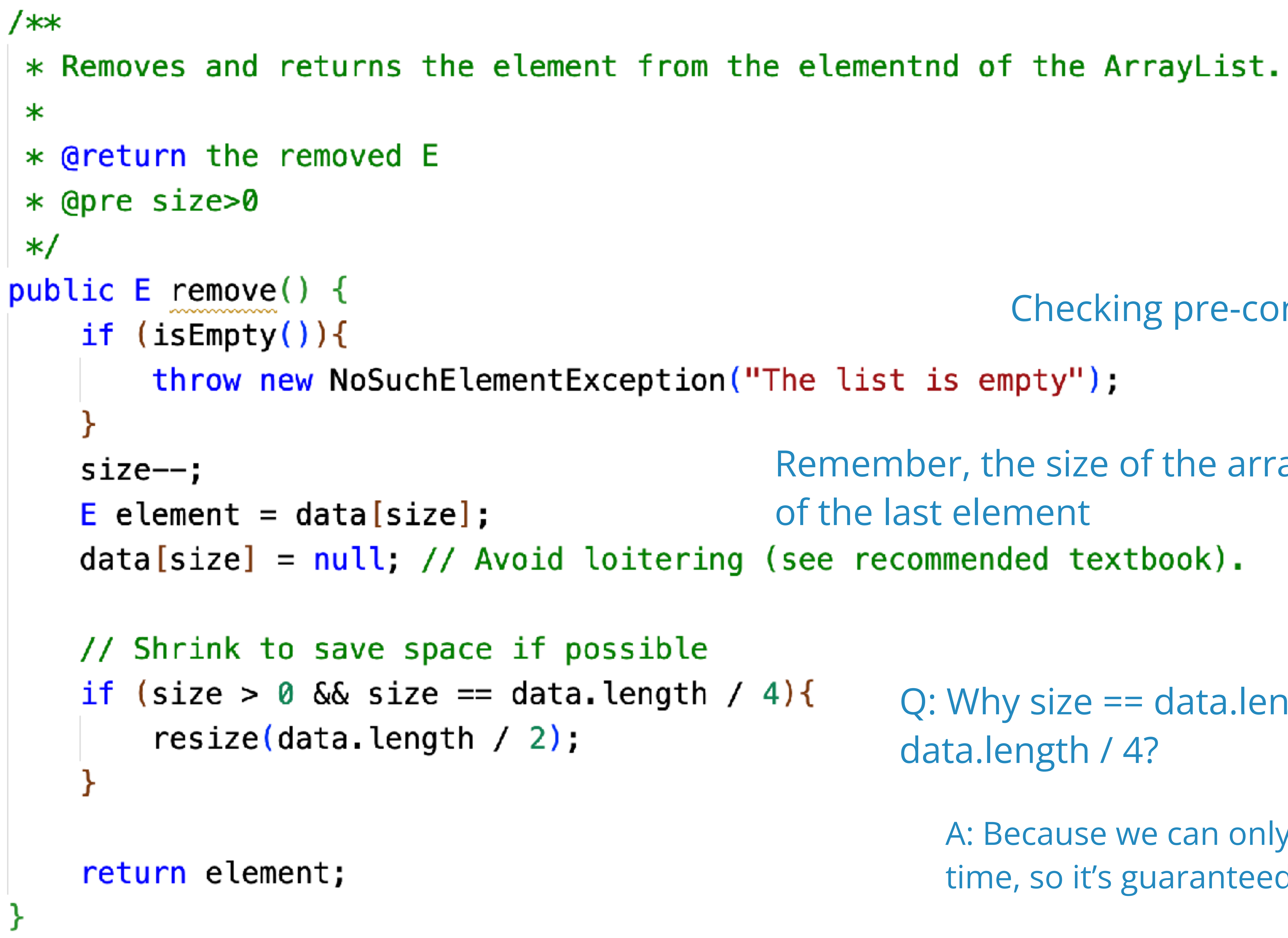
```java
/**
 * Inserts the element at the specified index. Shifts existing elements to the
 * right and doubles its capacity if necessary.
 *
 * @param index the index to insert the element
 * @param element the element to be inserted
 * @pre: 0 <= index <= size
 */
public void add(int index, E element) {
    //check whether index is in range
    if (index > size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

    //if full double in size
    if (size == data.length){
        resize(2 * data.length);
    }

    // shift elements to the right
    for (int i = size; i > index; i--){
        data[i] = data[i - 1];
    }
    //increase number of elements
    size++;
    //put the element at the right index in data
    data[index] = element;
}
```

Worksheet answer: set



```
/**
 * Replaces the element at the specified index with the specified E.
 *
 * @param index the index of the element to replace
 * @param element element to be stored at specified index
 * @return the old element that was replaced
 * @pre: 0<=index< size
 */
public E set(int index, E element) {
    //check if index is in range
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    //retreivew old element at index
    E old = data[index];
    //update index with new element
    data[index] = element;
    //return old element
    return old;
}
```

Removing (and returning) the last element



```java
/**
 * Removes and returns the element from the elementnd of the ArrayList.
 *
 * @return the removed E
 * @pre size>0
 */
public E remove() {
    if (isEmpty()){
        throw new NoSuchElementException("The list is empty");
    }
    size--;
    E element = data[size];
    data[size] = null; // Avoid loitering (see recommended textbook).

    // Shrink to save space if possible
    if (size > 0 && size == data.length / 4){
        resize(data.length / 2);
    }

    return element;
}
```

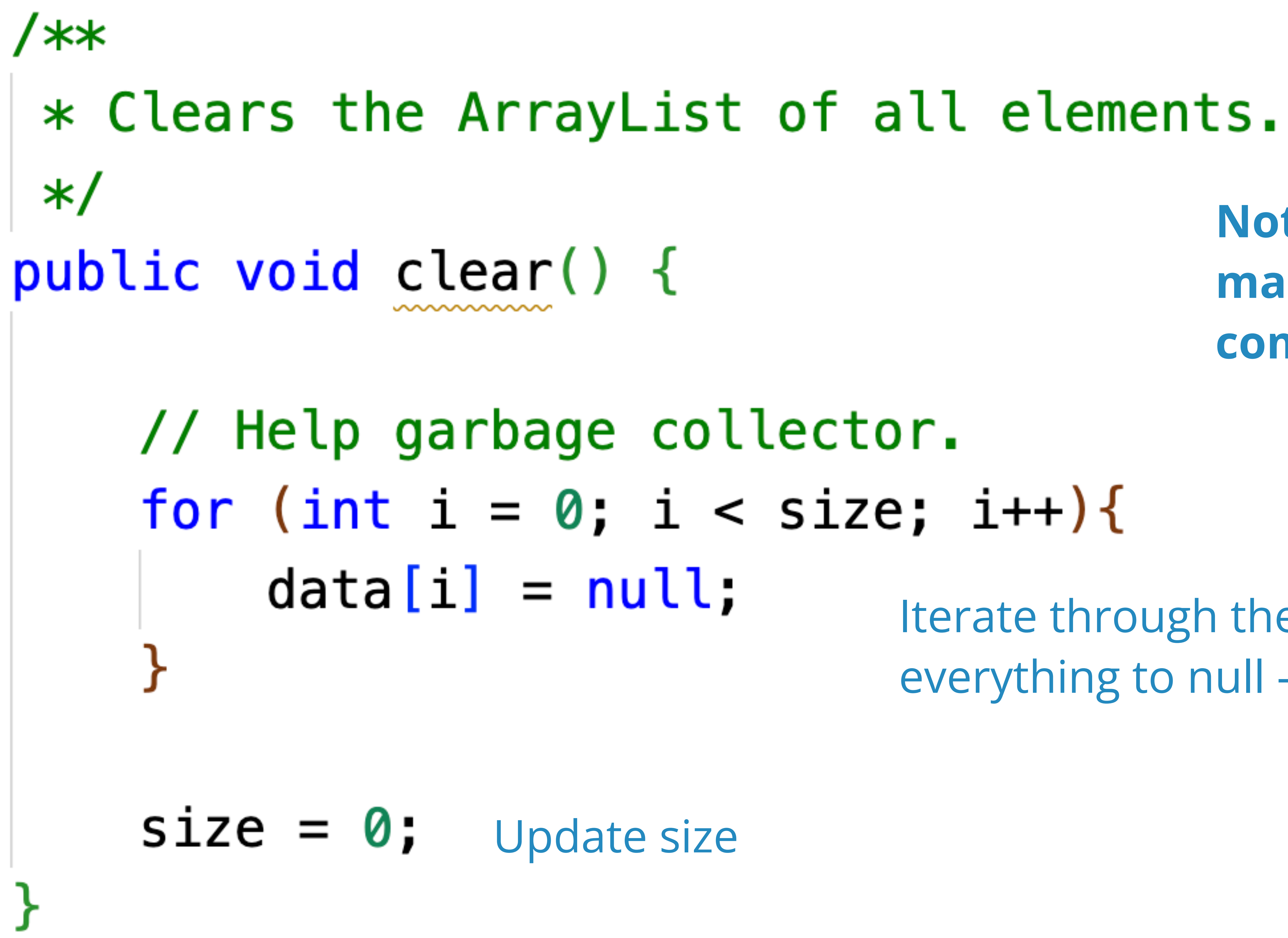
Checking pre-condition

Remember, the size of the array - 1 is the index of the last element

Q: Why size == data.length / 4? Why not size <= data.length / 4?

A: Because we can only remove one element at a time, so it's guaranteed to eventually be equal

Clearing the ArrayList



```
/**
 * Clears the ArrayList of all elements.
 */
public void clear() {

    // Help garbage collector.
    for (int i = 0; i < size; i++){
        data[i] = null;
    }

    size = 0;
}
```

Note that we don't need to call remove() many times - let's avoid unnecessary computation.

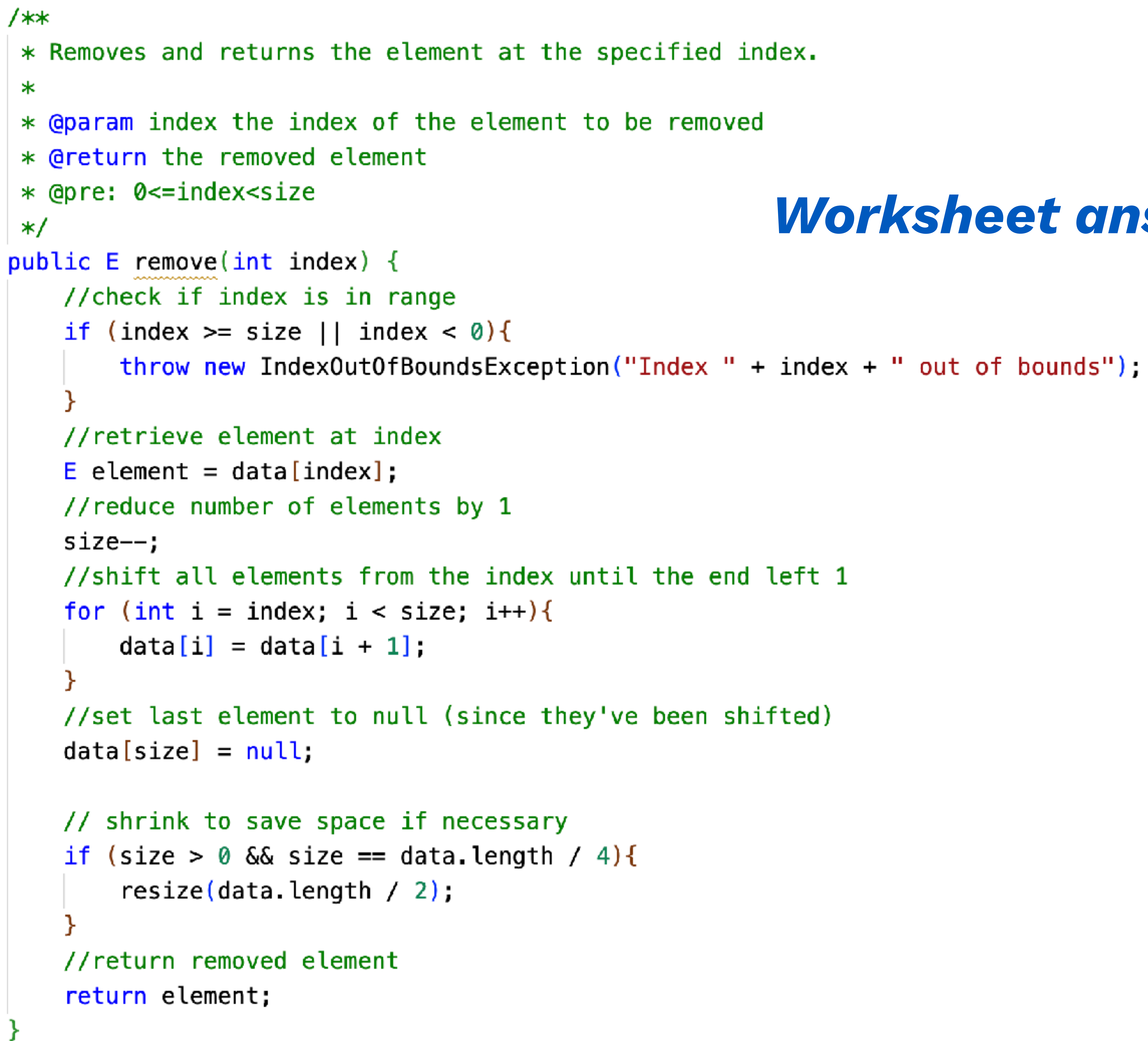
Iterate through the underlying Array and set everything to null - prevent "loitering"

Update size

Worksheet time!

- With a partner, write the implementation for
  - `E remove(int index) // returns and removes the element at the specified index`

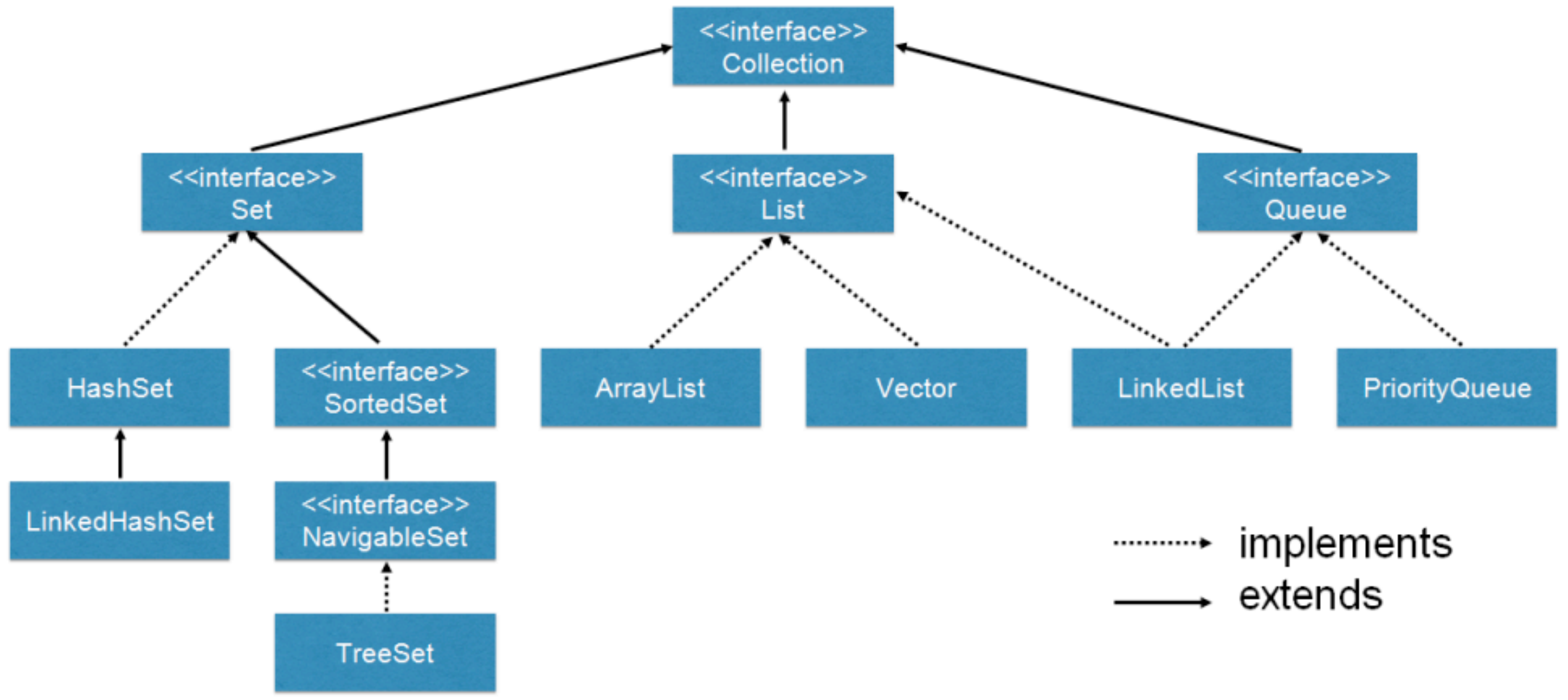
Worksheet answer: remove with index



```java
/**
 * Removes and returns the element at the specified index.
 *
 * @param index the index of the element to be removed
 * @return the removed element
 * @pre: 0<=index<size
 */
public E remove(int index) {
    //check if index is in range
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    //retrieve element at index
    E element = data[index];
    //reduce number of elements by 1
    size--;
    //shift all elements from the index until the end left 1
    for (int i = index; i < size; i++){
        data[i] = data[i + 1];
    }
    //set last element to null (since they've been shifted)
    data[size] = null;

    // shrink to save space if necessary
    if (size > 0 && size == data.length / 4){
        resize(data.length / 2);
    }
    //return removed element
    return element;
}
```

ArrayLists vs Vectors

Collection Interface



- Honestly, in the real world, not many people use ArrayLists. They prefer Vectors (e.g., most Leetcode problems in Java will use Vectors as “lists”)
- Vectors are slower, but *synchronized*, so they are memory safe.
- .push(), .pop() methods…we won’t learn them in this class, but telling you so you’re familiar in case they show up!

`ArrayList` in Java Collections

- Resizable list that increases by 50% when full and does NOT shrink.
- Not thread-safe (more in CS105).

```
java.util.ArrayList;

public class ArrayList<E> extends AbstractList<E> implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

`Vector` in Java Collections

- Java has one more class for resizable arrays.
- Doubles when full.
- Is synchronized (more in CS105).

```
java.util.Vector;

public class Vector<E> extends AbstractList<E> implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Lecture 6 wrap-up

- Exit ticket: https://forms.gle/UmwLXKqEEHpz9j2S7
- Part I of Darwin (first two classes) due Tues 11:59pm

Resources

- Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
- ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Resizable arrays in our textbook: https://algs4.cs.princeton.edu/13stacks/
- History of ArrayLists (made just for you all!) https://cs.pomona.edu/classes/cs62/history/arraylists/
- Note: the code has some Iterator<E> stuff for toString(), we'll go over it in a later lecture!