CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

7–8: Resizable Arrays

Alexandra Papoutsaki
LECTURES

Mark Kampe
LABS

Lecture 7: Resizable Arrays

- Background
- ArrayList
- Java Collections

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

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?
  - Space: How much memory do we need to organize our data?
- 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.

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.
  - 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.

Lecture 7: Resizable Arrays

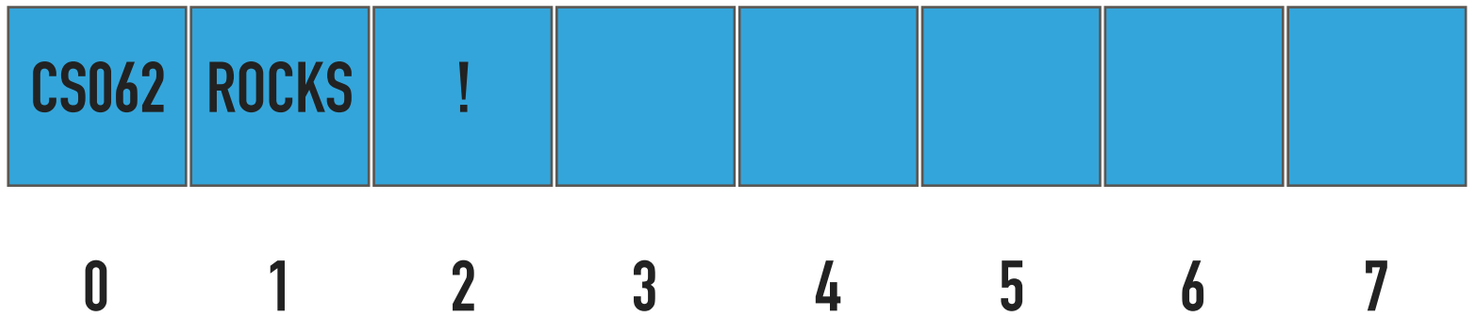
- Background
- ArrayList
- Java Collections

Limitations of arrays

- Fixed-size.
- Do not work with Generics.
- Adding or removing from the middle is hard.
- Limited functionality (in Java, requires the use of `Arrays` class)

Resizable array or ArrayList

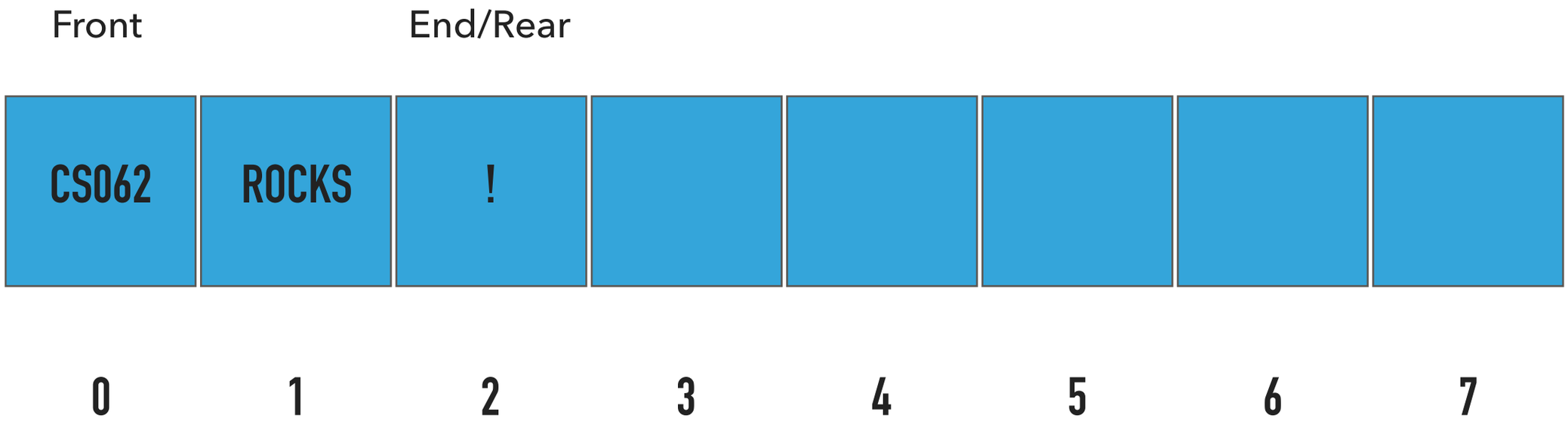
- Dynamic linear data structure that is zero-indexed.
- Sequential data structure that requires consecutive memory cells.
- Implemented with an underlying array of a specific capacity.



Standard Operations of `ArrayList<Item>` class

- `ArrayList()`: Constructs an empty ArrayList with an initial capacity of 2 (can vary across implementations).
- `ArrayList(int capacity)`: Constructs an empty ArrayList with the specified initial capacity.
- `isEmpty()`: Returns true if the ArrayList contains no items.
- `size()`: Returns the number of items in the ArrayList.
- `get(int index)`: Returns the item at the specified index.
- `add(Item item)`: Appends the item to the end of the ArrayList.
- `add(int index, Item item)`: Inserts the item at the specified index.
- `Item remove()`: Retrieves and removes the item from the end of the ArrayList.
- `Item remove(int index)`: Retrieves and removes the item at the specified index.
- `set(int index, Item item)`: Replaces the item at the specified index with the specified item.

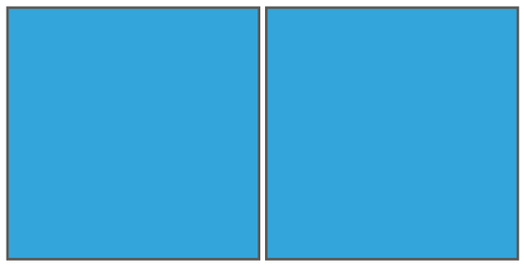
ArrayLists



Capacity = 8

Number of items = 3

`ArrayList()`: Constructs an ArrayList




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

Capacity = 2

Number of items = 0

What should happen?

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

`add(Item item)`: Appends the item to the end of the ArrayList


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

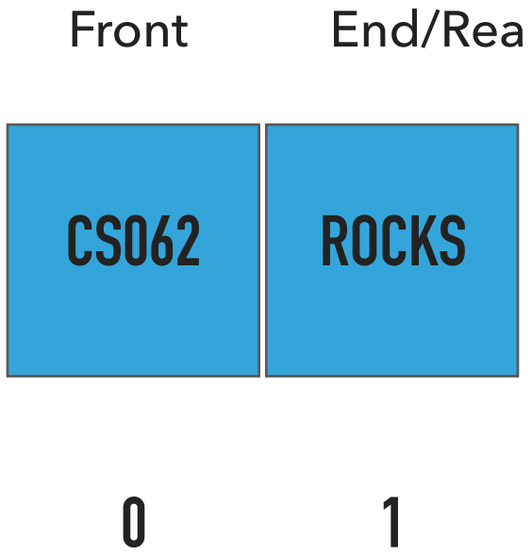
Capacity = 2

Number of items = 1

What should happen?

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

`add(Item item)`: Appends the item to the end of the ArrayList




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

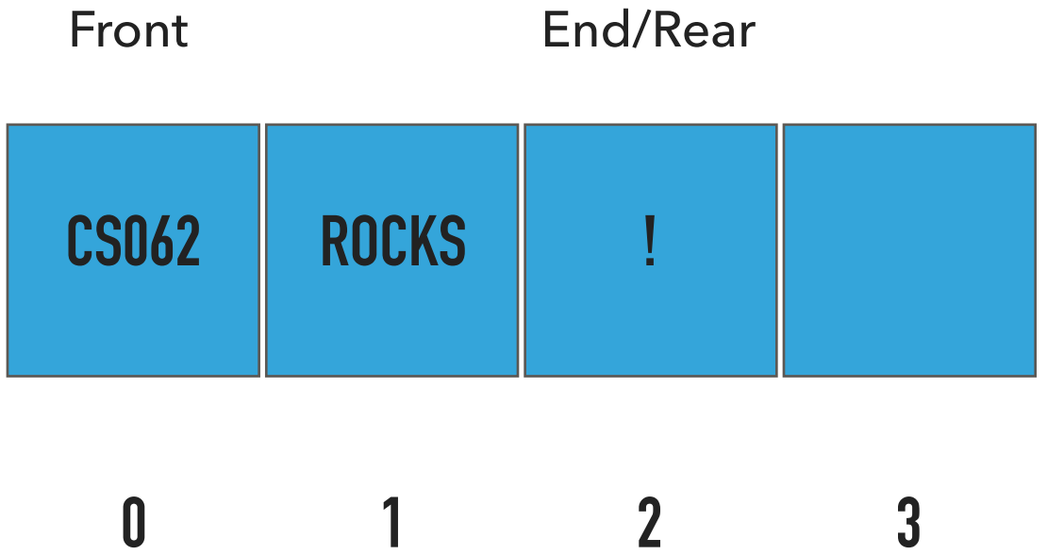
Capacity = 2

Number of items = 2

What should happen?

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

`add(Item item)`: Appends the item to the end of the ArrayList



Capacity = 4

Number of items = 3

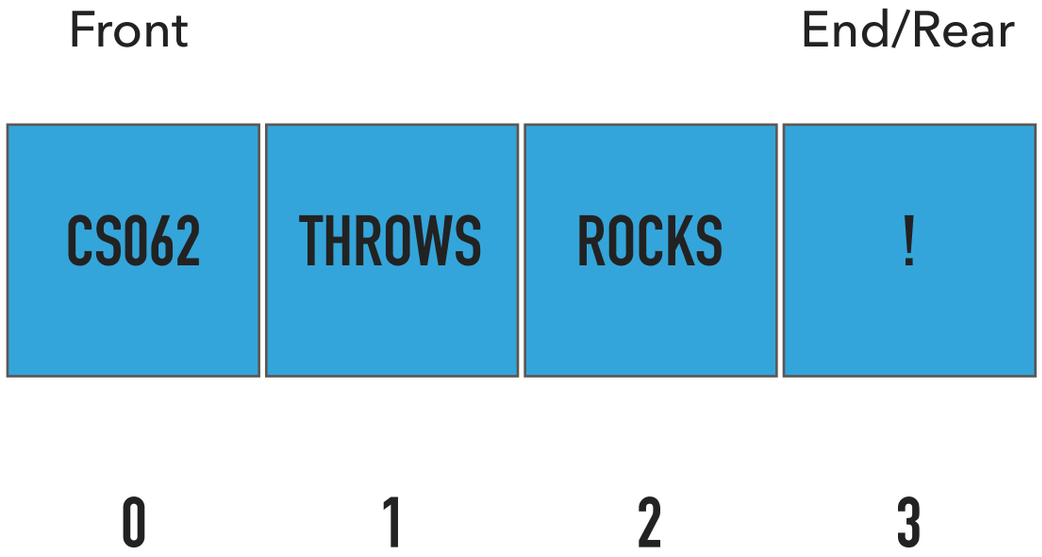
DOUBLE CAPACITY SINCE IT'S FULL AND THEN ADD NEW ITEM

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

What should happen?

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

`add(int index, Item item)`: Adds item at the specified index



Capacity = 4

Number of items = 4

SHIFT ELEMENTS TO THE RIGHT
ADD NEW ITEM

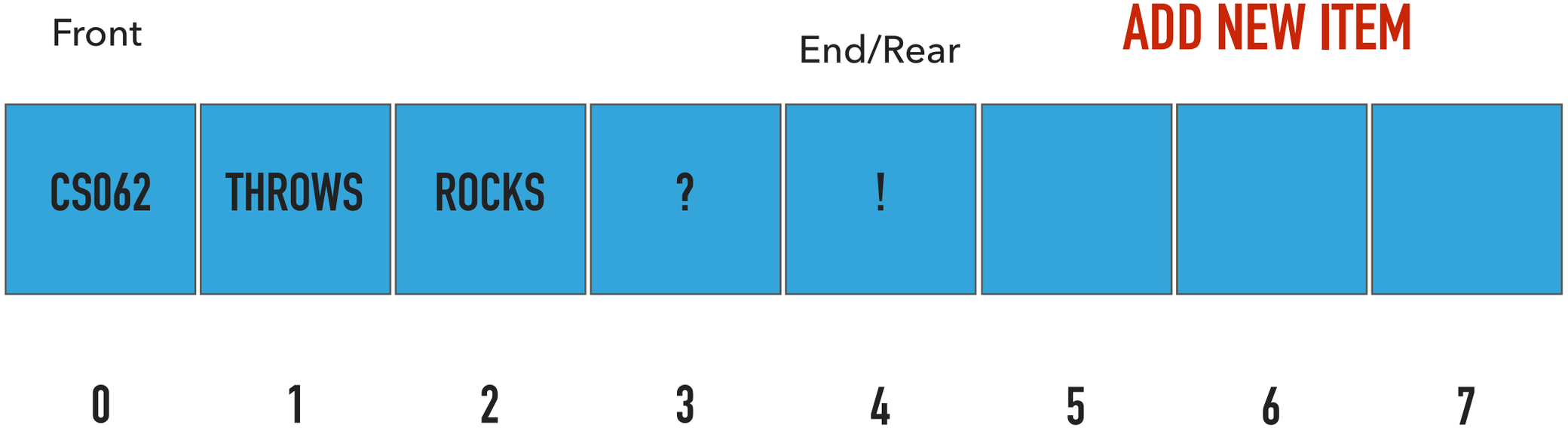
```
al.add(1, “THROWS”);
```

What should happen?

```
al.add(3, “?”);
```

`add(int index, Item item)`: Adds item at the specified index

RESIZE
SHIFT ELEMENTS TO THE RIGHT
ADD NEW ITEM



Capacity = 8

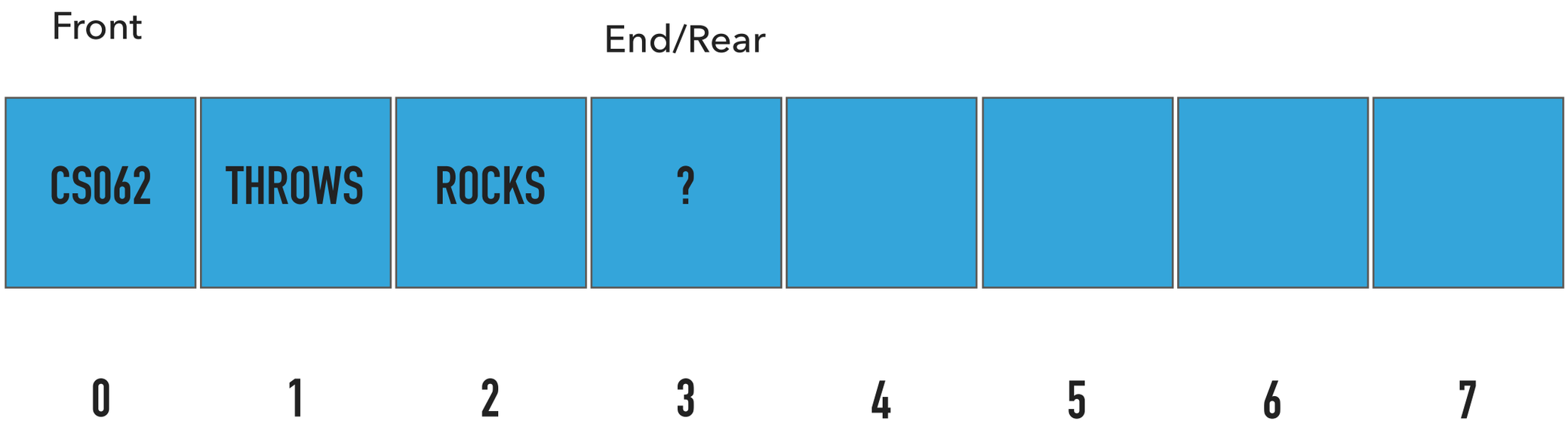
Number of items = 5

```
al.add(3, “?”);
```

What should happen?

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

`remove()`: Retrieves and removes item from the end of ArrayList



Capacity = 8

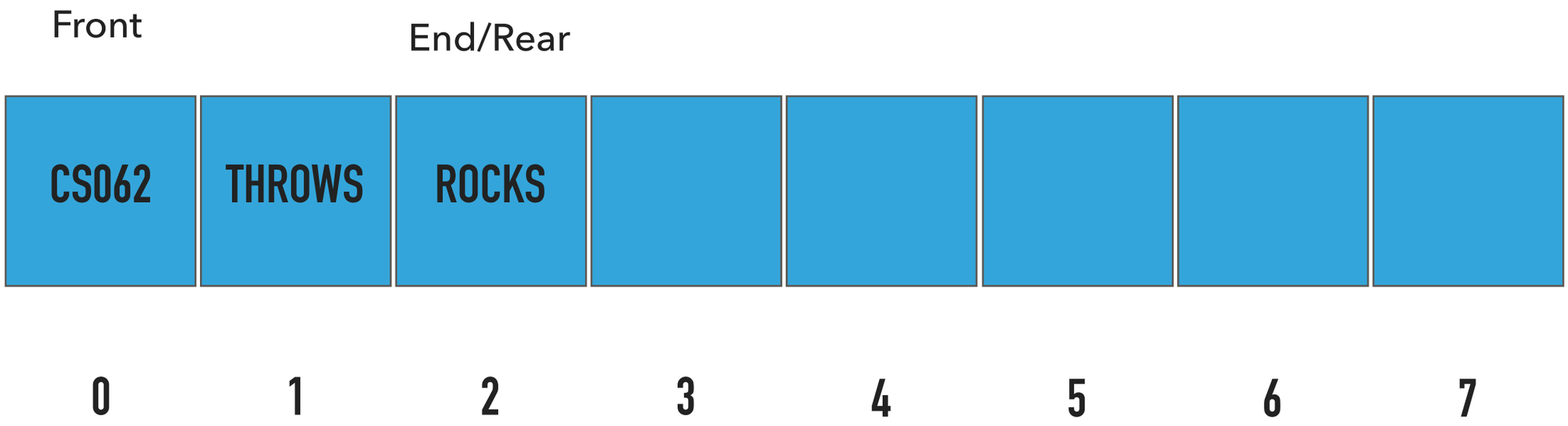
Number of items = 4

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

What should happen?

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

`remove()`: Retrieves and removes item from the end of ArrayList



Capacity = 8

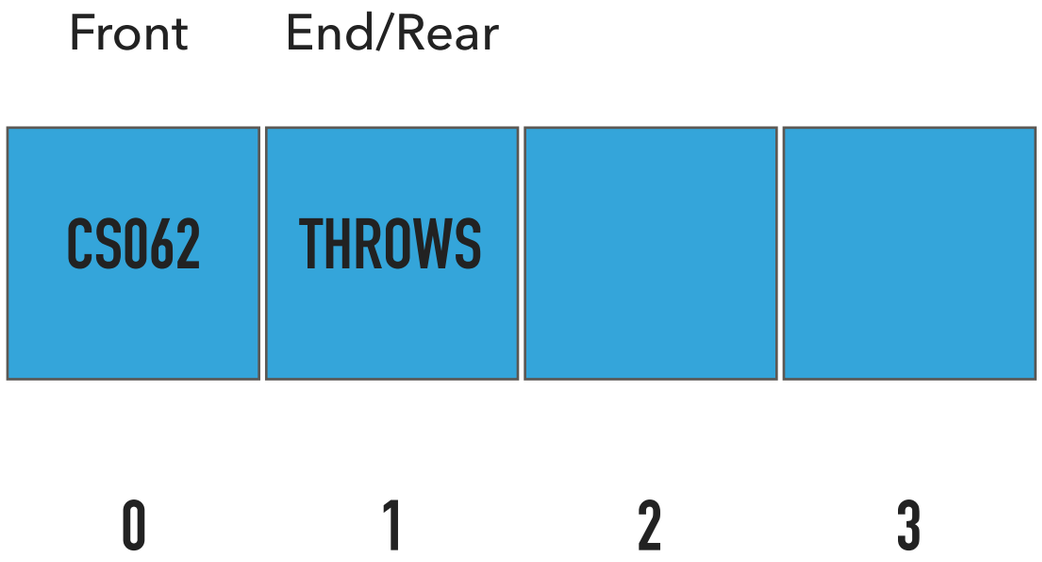
Number of items = 3

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

What should happen?

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

`remove()`: Retrieves and removes item from the end of ArrayList




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

Capacity = 4

Number of items = 2

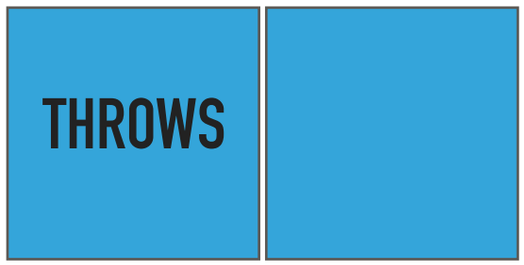
REMOVE ITEM FROM THE END
HALVE CAPACITY WHEN 1/4 FULL

What should happen?

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

`remove(int index)`: Retrieves and removes item from specified index

Front End/Rear



0 1

Capacity = 2

Number of items = 1

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

REMOVE ITEM FROM INDEX
SHIFT ELEMENTS TO THE LEFT
HALVE CAPACITY WHEN 1/4 FULL

Our own implementation of ArrayLists

- We will follow the textbook style.
  - It does not offer a class for this so we will build our own. We will also test our implementation in lab!
- We will work with generics because we don't want to offer multiple implementations.
- We will use an array and we will keep track of how many elements we have in our arrayList.

Instance variables and constructors

```
public class ArrayList<Item> implements Iterable<Item> {
    private Item[] a; // underlying array of items
    private int n; // number of items in arraylist

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

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

Check if is empty and how many items

```
/**
 * Returns true if the ArrayList contains no items.
 *
 * @return true if the ArrayList does not contain any item
 */
public boolean isEmpty() {
    return n == 0;
}
```

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

Resize underlying array's capacity

```
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    assert capacity >= n;
    // textbook implementation.
    Item[] temp = (Item[]) new Object[capacity];
    for (int i = 0; i < n; i++)
        temp[i] = a[i];

    a = temp;

    // alternative implementation
    // a = java.util.Arrays.copyOf(a, capacity);
}
```

Append an item to the end of ArrayList

```
/**
 * Appends the item to the end of the ArrayList. Doubles its capacity if necessary.
 *
 * @param item
 *            the item to be inserted
 */
public void add(Item item) {
    if (n == a.length)
        resize(2 * a.length);

    a[n++] = item;
}
```

Check if index is >=0 and <n

```
/**
 * A helper method to check if an index is in range 0<=index<n
 *
 * @param index
 *            the index to check
 */
private void rangeCheck(int index) {
    if (index > n || index < 0)
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
}
```

Add an item at a specified index

```
/**
 * Inserts the item at the specified index. Shifts existing elements
 * to the right and doubles its capacity if necessary.
 *
 * @param index
 *            the index to insert the item
 * @param item
 *            the item to be inserted
 */
public void add(int index, Item item) {
    rangeCheck(index);

    if (n == a.length)
        resize(2 * a.length);

    // shift elements to the right
    for (int i = n++; i > index; i--)
        a[i] = a[i - 1];

    a[index] = item;
}
```

Replace an item at a specified index

```
/**
     * Replaces the item at the specified index with the specified item.
     *
     * @param index
     *            the index of the item to replace
     * @param item
     *            item to be stored at specified index
     * @return the old item that was changed.
     */
    public Item set(int index, Item item) {
        rangeCheck(index);

        Item old = a[index];
        a[index] = item;

        return old;
    }
```

Retrieve and remove item from the end of ArrayList

```
/**
     * Retrieves and removes the item from the end of the ArrayList.
     * @return the removed item
     * @pre n>0
     */
    public Item remove() {
        if (isEmpty())
            throw new NoSuchElementException("The list is empty");

        Item item = a[--n];
        a[n] = null; // Avoid loitering (see text).

        // Shrink to save space if possible
        if (n > 0 && n == a.length / 4)
            resize(a.length / 2);

        return item;
}
```

Retrieve and remove item from a specific index

```
/**
     * Retrieves and removes the item at the specified index.
     *
     * @param index
     *            the index of the item to be removed
     * @return the removed item
     */
    public Item remove(int index) {
        rangeCheck(index);

        Item item = a[index];
        n--;

        for (int i = index; i < n; i++)
            a[i] = a[i + 1];

        a[n] = null; // Avoid loitering (see text).

        // shrink to save space if necessary
        if (n > 0 && n == a.length / 4)
            resize(a.length / 2);

        return item;
}
```

Clear all elements

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

    // Help garbage collector.
    for (int i = 0; i < n; i++)
        a[i] = null;

    n = 0;
}
```

Lecture 7: Resizable Arrays

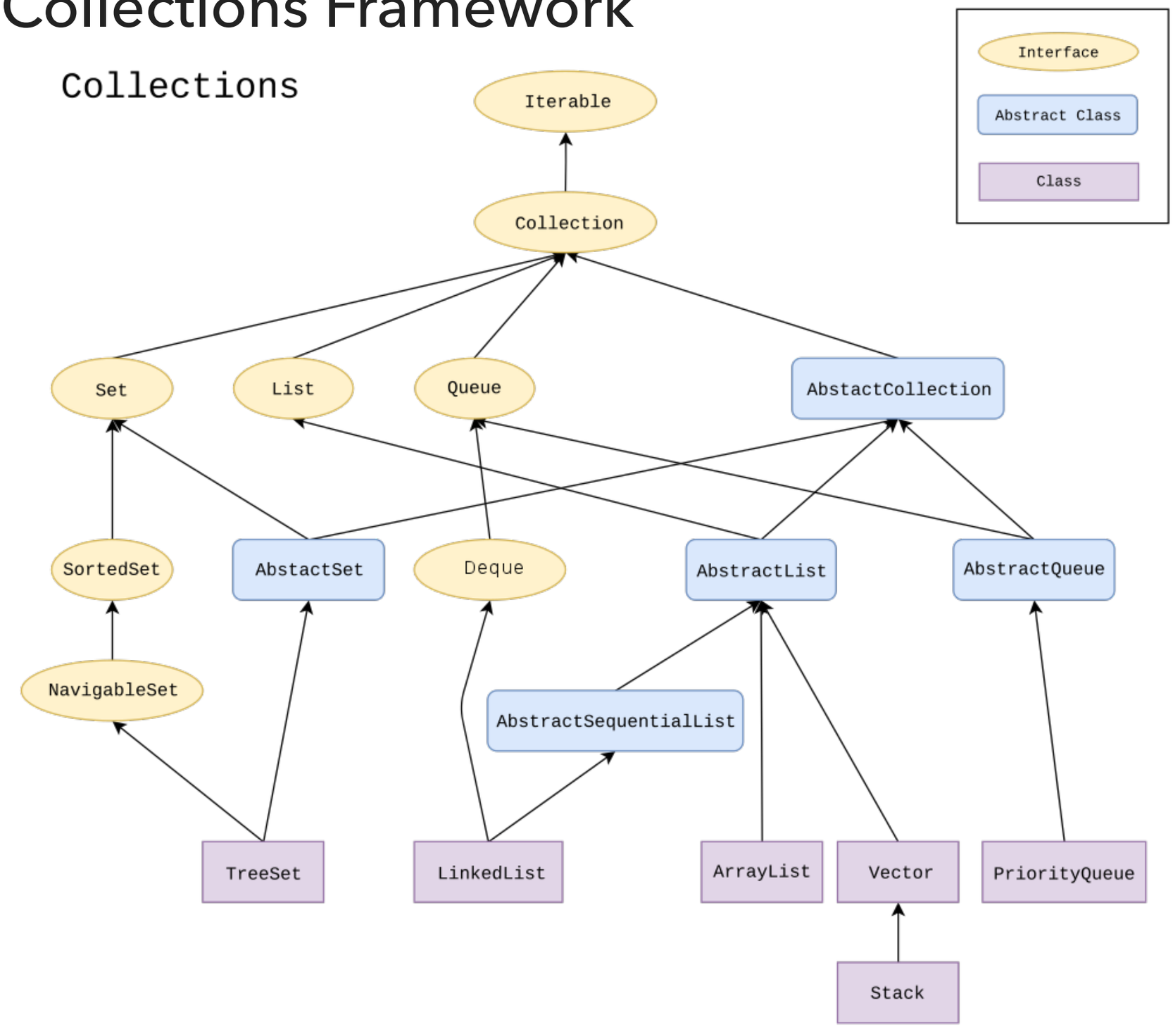
- Background
- ArrayList
- Java Collections

The Java Collections Framework

- Collection: an object that groups multiple elements into a single unit, allowing us to store, retrieve, manipulate data.
- Collections Framework:
  - Interfaces: ADTs that represent collections.
  - Implementations: The actual data structures.
  - Algorithms: methods that perform useful computations, such as searching and sorting.

https://docs.oracle.com/javase/tutorial/collections/intro/index.html

The Java Collections Framework



https://en.wikipedia.org/wiki/Java_collections_framework

List ADT

- A collection storing elements in an ordered fashion.
- Elements are accessed in a zero-based fashion.
- Typically allow duplicate elements and null values but always check the specifications of implementation.

https://en.wikipedia.org/wiki/Java_collections_framework

`ArrayList` in Java Collections

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

```
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 later in the course).

```
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 7: Resizable Arrays

- Background
- ArrayList
- Java Collections

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Textbook:
  - Chapter 1.3 (Page 136–137)
- Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/