BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

6: Resizable Arrays

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his

Lecture 6: Resizable Arrays

- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

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 on a data structure?
  - Space: How much memory do we need to organize our data in a data structure?
- 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.
    - And ArrayList, but informally.

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.

Lecture 6: Resizable Arrays

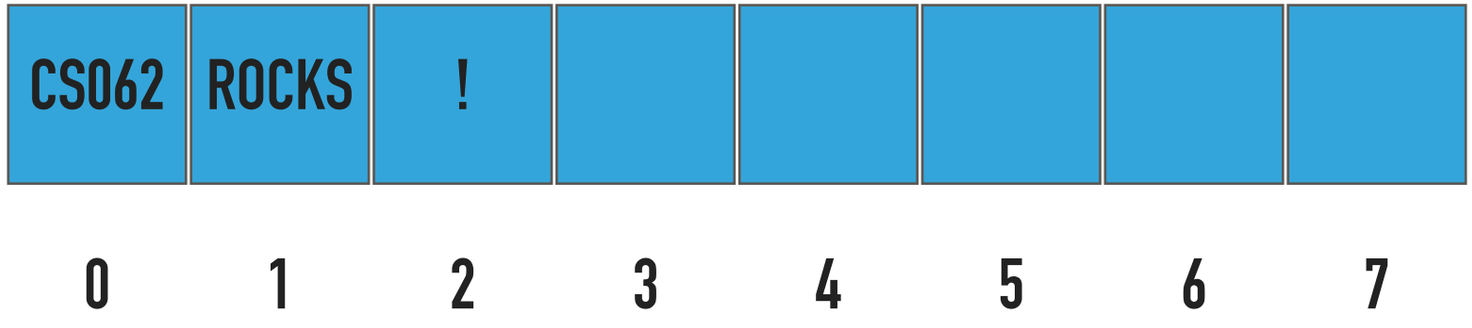
- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

Limitations of arrays

- Fixed-size.
- Do not work well with Generics.
  - `E[] myArray = new (E[]) Object[capacity];`
- Adding or removing from the middle is hard.
- Limited functionality (Java requires the use of `Arrays` class for manipulating arrays, such as sorting and searching).

Resizable array or ArrayList

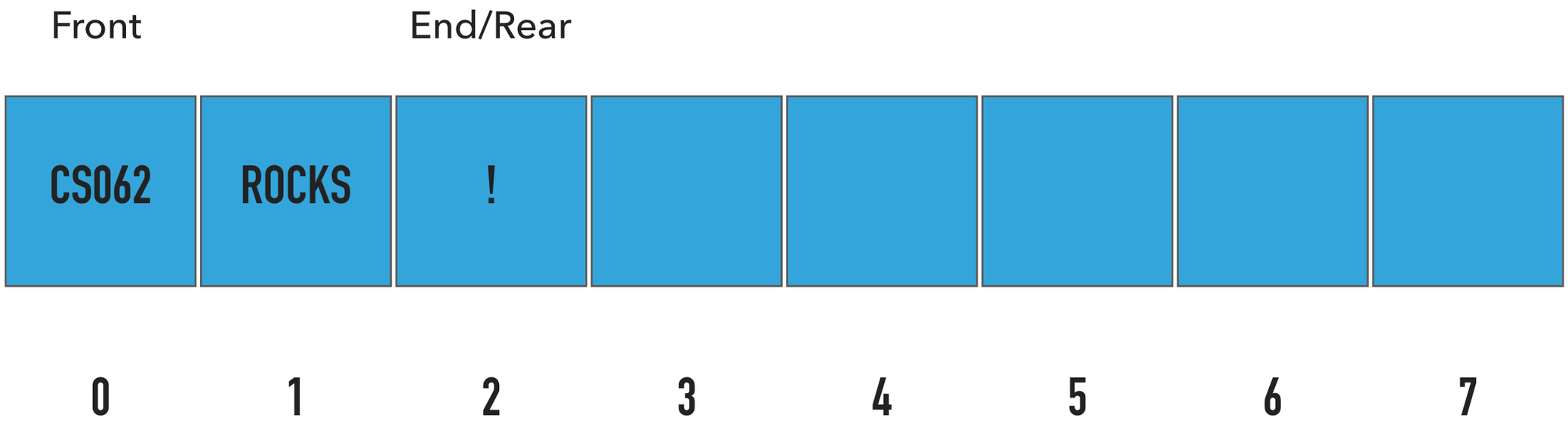
- Dynamic linear data structure that is zero-indexed.
- Sequential data structure that requires consecutive memory cells.
- Implemented with an underlying array of a specific capacity.
- But the client does not see that!



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 and shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- `Item remove()`: Removes and returns the item at the end of the ArrayList.
- `Item remove(int index)`: Retrieves and removes the item at the specified index. Shifts any subsequent elements to the left (subtracts one from their indices).
- `set(int index, Item item)`: Replaces the item at the specified index with the specified item.

ArrayLists



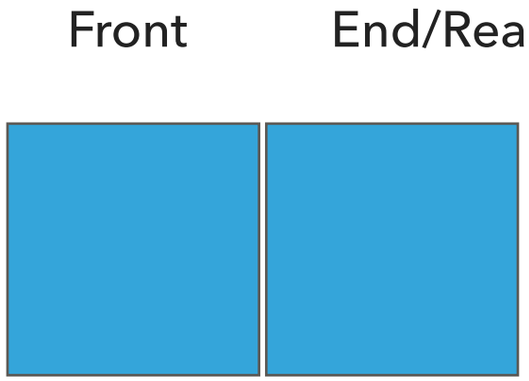
Capacity = 8

Number of items = 3

What should happen?

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

ArrayList(): Constructs an ArrayList




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

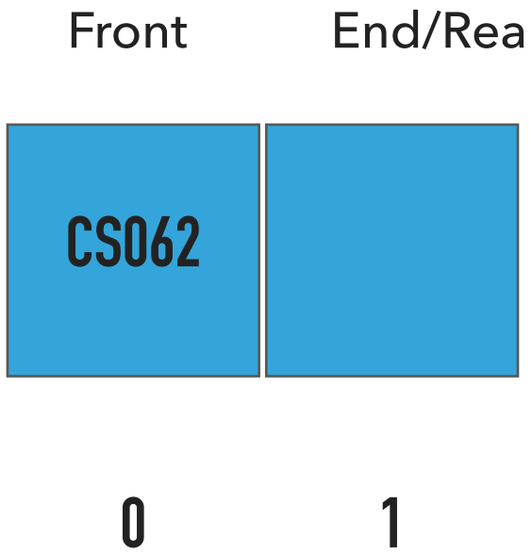
Capacity = 2

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");
```

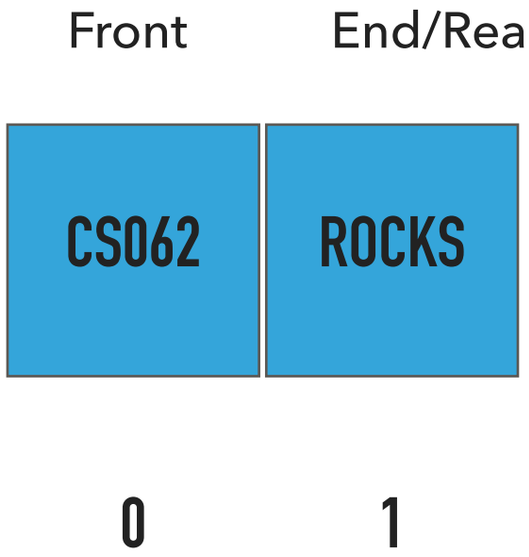
Capacity = 2

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");
```

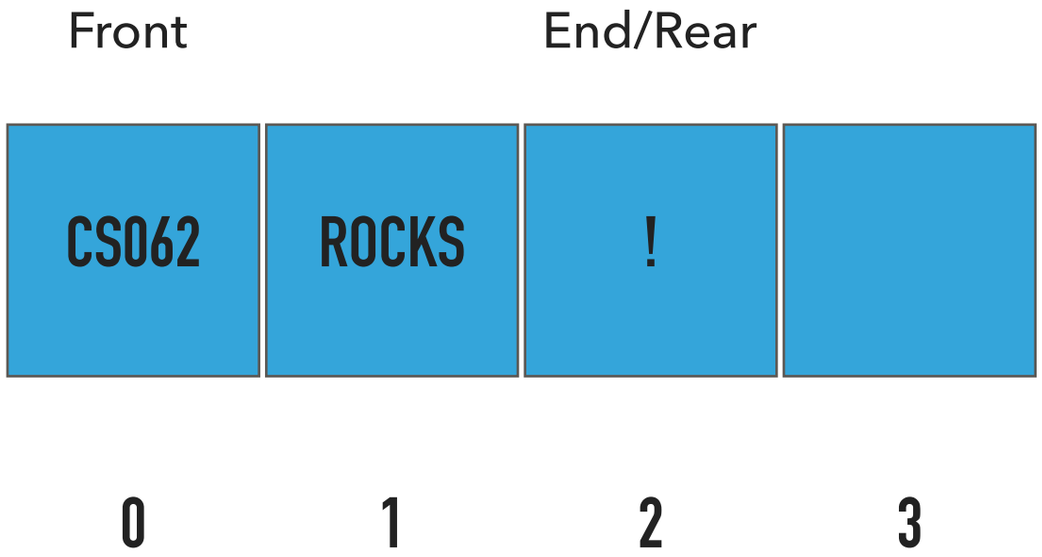
Capacity = 2

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

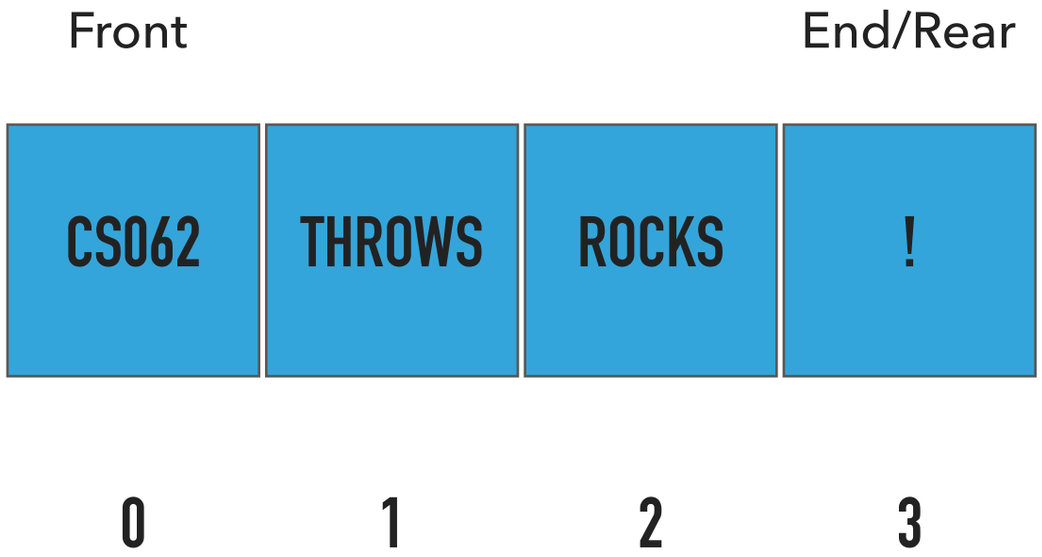
DOUBLE CAPACITY SINCE IT'S FULL AND THEN ADD NEW ITEM INCREASE NUMBER OF ITEMS

```
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
INCREASE NUMBER OF ITEMS

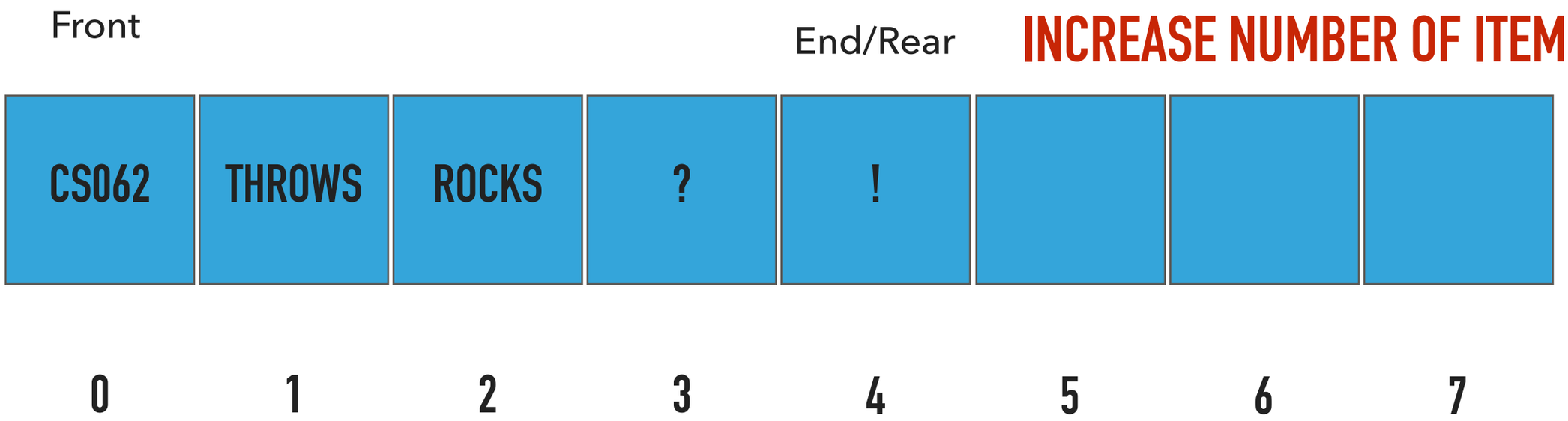
```
al.add(1, “THROWS”);
```

What should happen?

```
al.add(3, “?”);
```

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

DOUBLE CAPACITY SINCE IT'S FULL
SHIFT ELEMENTS TO THE RIGHT
ADD NEW ITEM
INCREASE NUMBER OF ITEMS



Capacity = 8

Number of items = 5

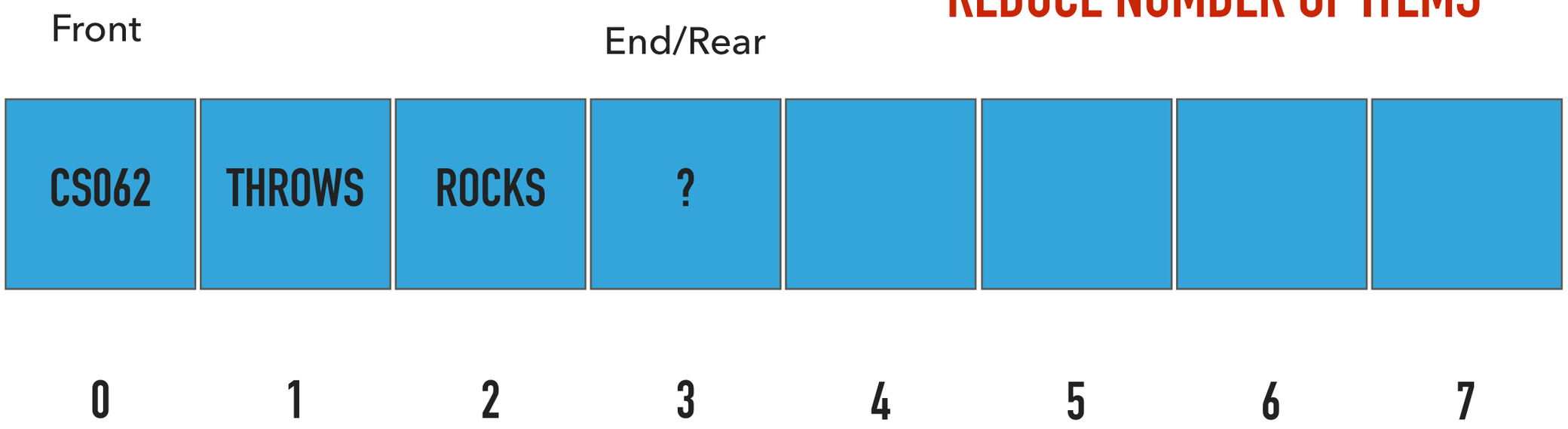
```
al.add(3, "?");
```

What should happen?

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

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

RETURN LAST ELEMENT
REDUCE NUMBER OF ITEMS



Capacity = 8

Number of items = 4

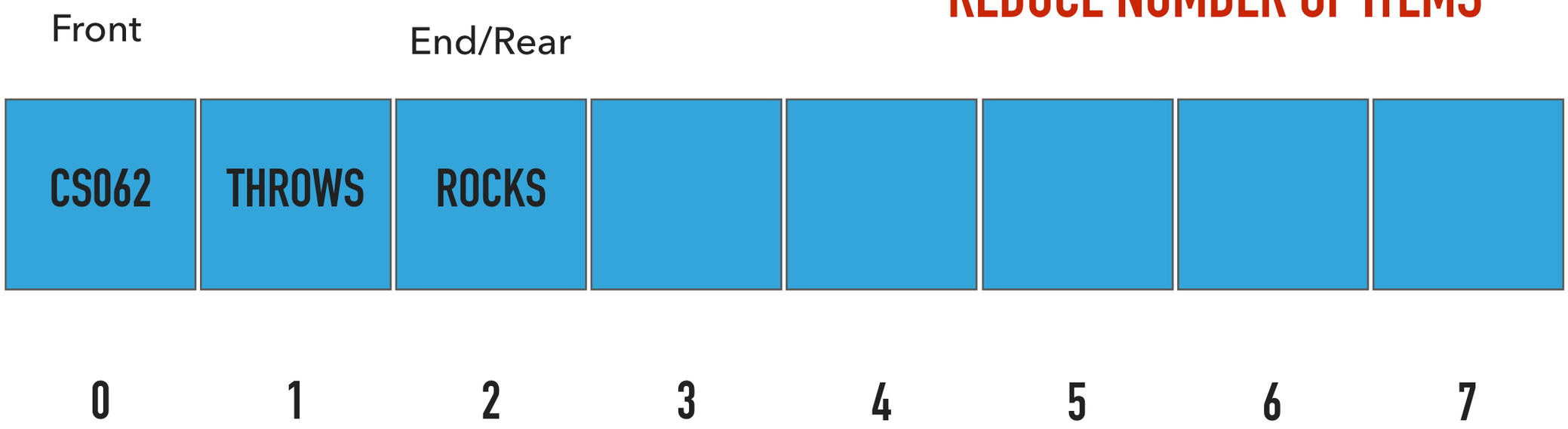
```
al.remove();
```

What should happen?

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

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

RETURN LAST ELEMENT
REDUCE NUMBER OF ITEMS



Capacity = 8

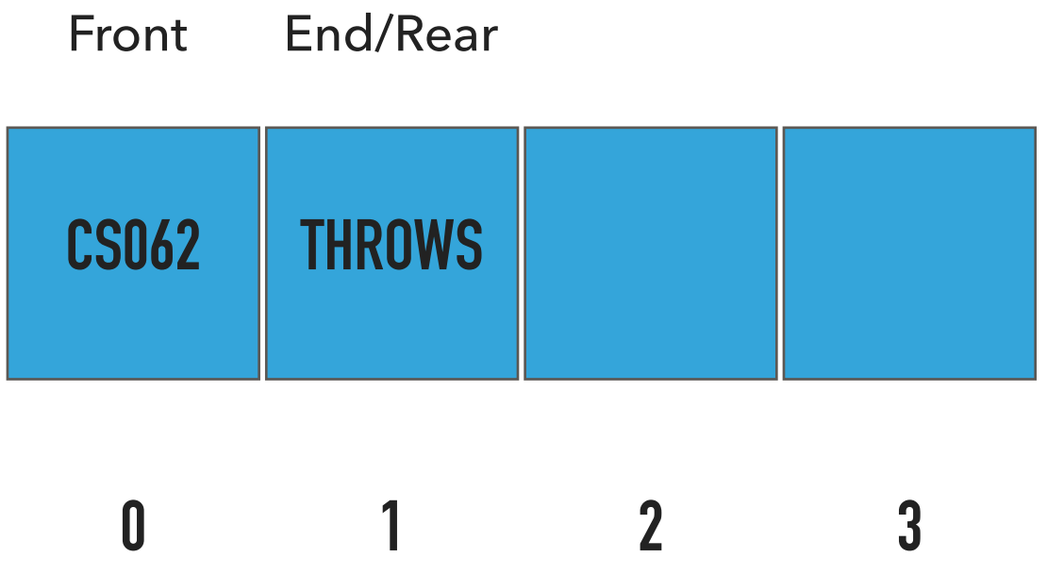
Number of items = 3

```
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

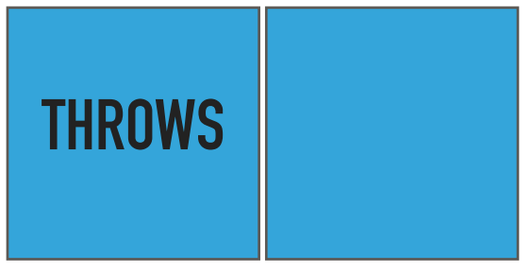
REMOVE ITEM FROM THE END
HALVE CAPACITY WHEN 1/4 FULL

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. Yesterday, we got to test a very similar 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.

PRACTICE TIME: 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() {


    }

    /**
     * Constructs an ArrayList with the specified capacity.
     */
    @SuppressWarnings("unchecked")
    public ArrayList(int capacity) {


    }
```

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;
    }
```

PRACTICE TIME: Check if is empty and how many items

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

}
```

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

}
```

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;
}
```

PRACTICE TIME: Resize underlying array's capacity

```
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array with the provided capacity

    //copy all the elements from the old array (a) into the temporary array

    //point a to the new temporary array


}
```

Resize underlying array's capacity

```
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array of Items with the provided capacity
    Item[] temp = (Item[]) new Object[capacity];
    //copy all the elements from the old array (a) into the temporary array
    for (int i = 0; i < n; i++)
        temp[i] = a[i];
    //point a to the new temporary array
    a = temp;

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

PRACTICE TIME: 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) {
     //check whether ArrayList is full
     //if yes, double in size

     //add the item at the end of the ArrayList and increase the counter by 1
}
```

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) {
     //check whether ArrayList is full
    if (n == a.length){
     //if yes, double in size
        resize(2 * a.length);
    }
     //add the item at the end of the ArrayList and increase the counter by 1
    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
 * @throws IndexOutOfBoundsException if not in range
 */
private void rangeCheck(int index) {
    if (index > n || index < 0)
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
}
```

PRACTICE TIME: 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) {
    //check whether index in range

    //if full double size

    //shift elements to the right


    //set item to position index

    //increase number of items

}
```

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) {
     //check whether index in range
     rangeCheck(index);

     //if full double size
     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];

     //set item to position index
     a[index] = item;
}
```

PRACTICE TIME: 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) {
    //check whether index in range

    //retrieve old item at index

    //update index with new item

    //return old 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) {
    //check whether index in range
    rangeCheck(index);
    //retrieve old item at index
    Item old = a[index];
    //update index with new item
    a[index] = item;
    //return old item
    return old;
}
```

PRACTICE TIME: Retrieve and remove item from the end of ArrayList

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

    //retrieve last item and reduce number of items by 1

    //set the position where the removed item is to null

    //shrink in half to save space if number of items in ArrayList is 1/4 of its size

    //return the removed item
}
```

Retrieve and remove item from the end of ArrayList

```
 /**
 * Retrieves and removes the item from the end of the ArrayList.
 * @return the removed item
 * @throws NoSuchElementException if ArrayList is empty
 * @pre n>0
 */
public Item remove() {
    //if ArrayList is empty throw NoSuchElementException
    if (isEmpty())
        throw new NoSuchElementException("The list is empty");
    //retrieve last item and reduce number of items by 1
    Item item = a[--n];
    //set the position where the removed item is to null
    a[n] = null; // Avoid loitering (see text).

    //shrink in half to save space if number of items in ArrayList is 1/4 of its size
    if (n > 0 && n == a.length / 4)
        resize(a.length / 2);
    //return the removed item
    return item;
}
```

PRACTICE TIME: 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) {
        //check whether index in range

        //retrieve Item at index

        //reduce number of items by 1

        //shift all items from index till the end one position to the left


        //set the last item (since they have been shifted to the left), to null

        //shrink in half to save space if number of items in ArrayList is 1/4 of its size

        //return removed 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) {
        //check whether index in range
        rangeCheck(index);

        //retrieve Item at index
        Item item = a[index];
        //reduce number of items by 1
        n--;
         //shift all items from index till the end one position to the left
        for (int i = index; i < n; i++)
            a[i] = a[i + 1];
        //set the last item (since they have been shifted to the left), to null
        a[n] = null; // Avoid loitering (see text).

        //shrink in half to save space if number of items in ArrayList is 1/4 of its size
        if (n > 0 && n == a.length / 4)
            resize(a.length / 2);
        //return removed item
        return item;
}
```

Clear all elements

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

    // Go through all elements of the array and set them to null

    // Set number of items to 0

}
```

Clear all elements

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

    // Go through all elements of the array and set them to null
    for (int i = 0; i < n; i++)
        a[i] = null;
    // Set number of items to 0
    n = 0;
}
```

Lecture 6: Resizable Arrays

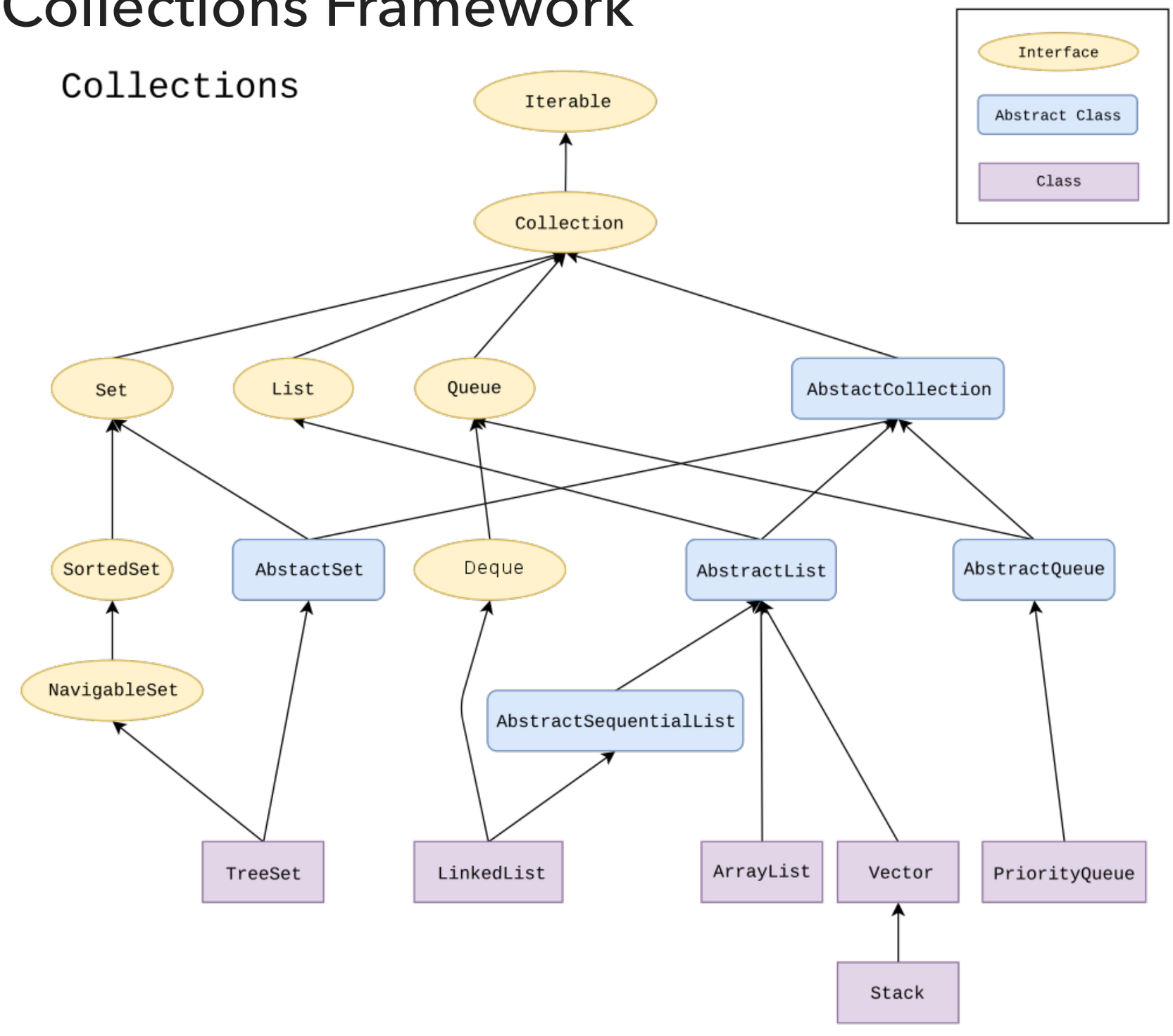
- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

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 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: Resizable Arrays

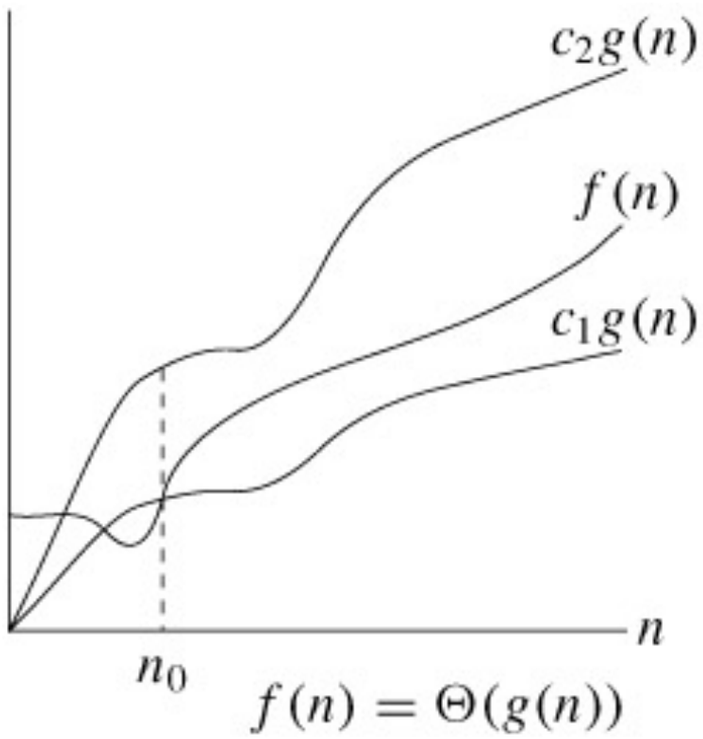
- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

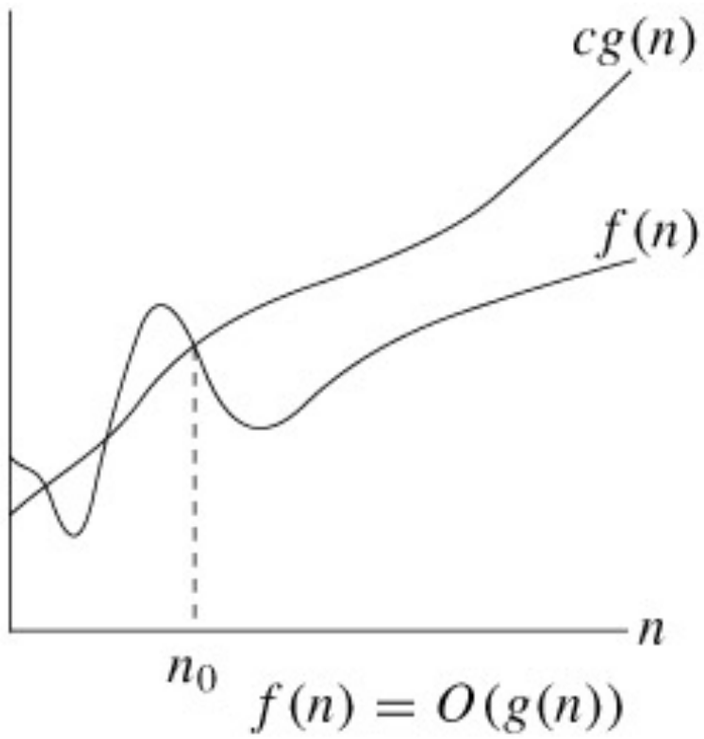
Type of analyses

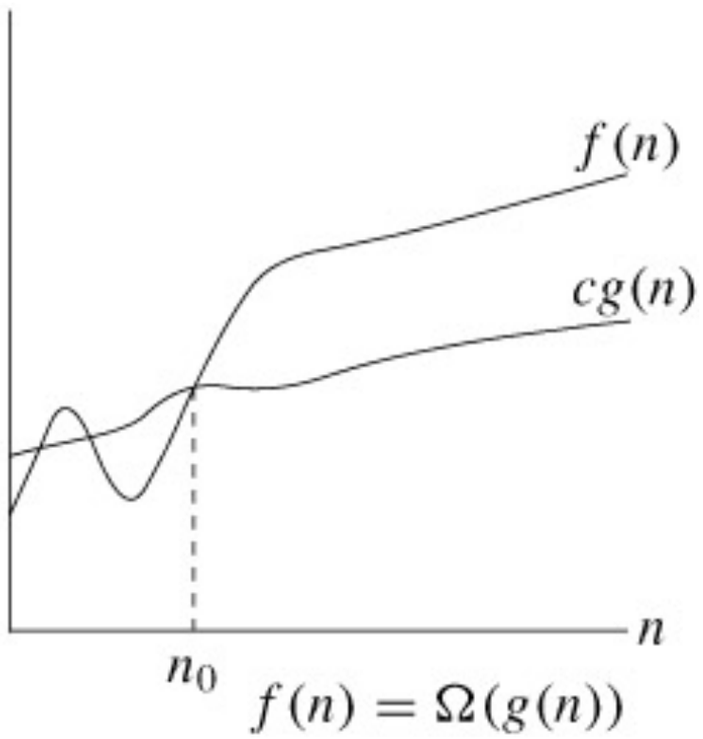
- Best case: lower bound on cost.
  - What the goal of all inputs should be.
  - Often not realistic, only applies to “easiest” input.
- Worst case: upper bound on cost.
  - Guarantee on all inputs.
  - Calculated based on the “hardest” input.
- Average case: expected cost for random input.
  - A way to predict performance.
  - Not straightforward how we model random input.

Asymptotic Notations

- Θ notation: bounds function from above and below.
- O notation: bounds function from above.
- Ω notation: bounds function from below.

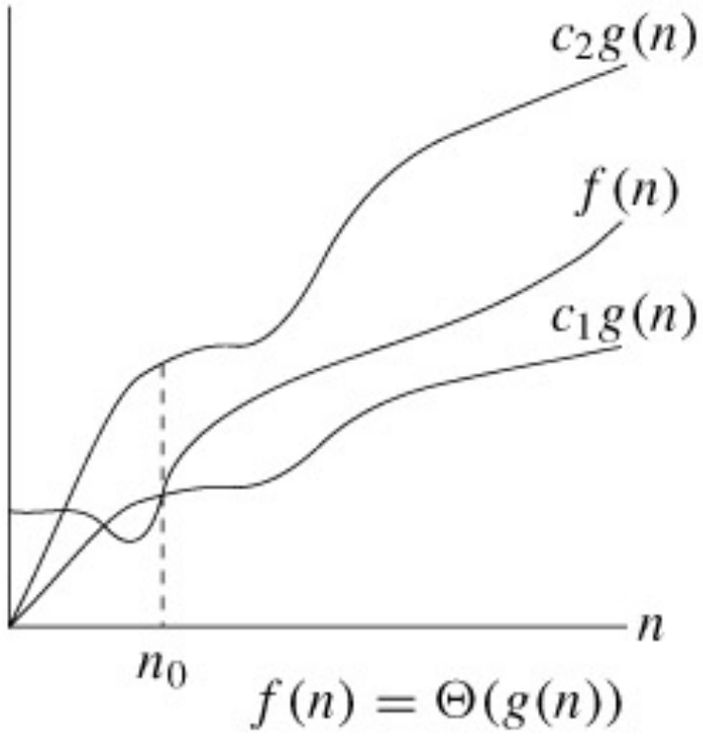




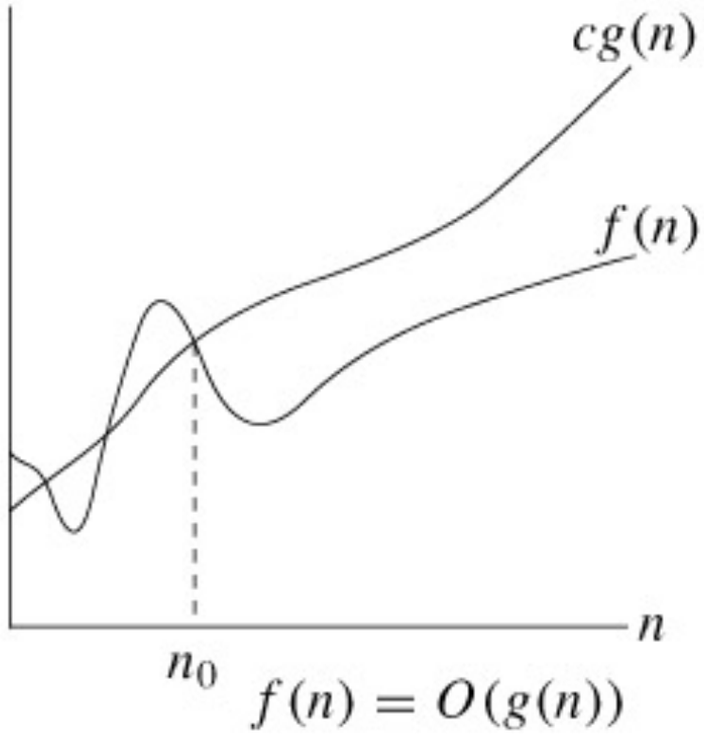


Big O - asymptotic upper bound

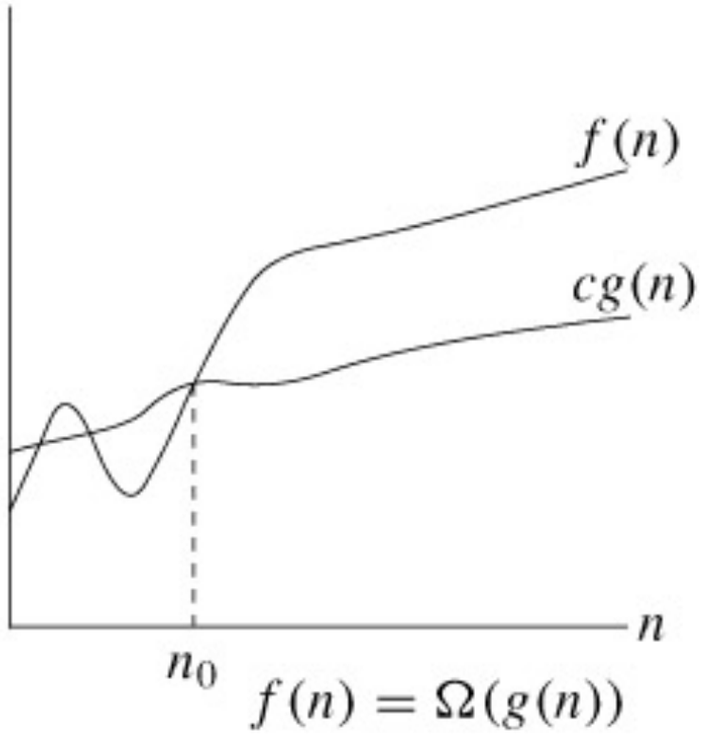
- For a given function $f(n)$, the order of growth is $g(n)$ ($O(g(n))$) if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n > n_0$}



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Asymptotic analysis simplifies analyzing worst-case performance

- We will be dropping constants. For example:
  - $3n^3 + 2n + 7 = O(n^3)$
  - $2^n + n^2 = O(2^n)$
  - $1000 = O(1)$
- Yes, $3n^3 + 2n + 7 = O(n^6)$, but that's a rather useless bound.
- Sorting them by increasing rate of growth:
  - $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!)$

How to interpret Big O

- $O(1)$ or "order one”: running time does not change as size of the problem changes, that is running time stays constant and independent of problem size.
- $O(\log n)$ or "order log n”: running time increases as problem size grows. Whenever problem size doubles, running time increases by a constant.
- $O(n)$ or "order n”: time increases proportionally to the the rate of growth of the size of the problem, that is in a linear rate. Double the problem size, you get double running time.
- $O(n^2)$ or "order n squared”: Double the problem size you get quadruple running time.

Lecture 6: Resizable Arrays

- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

Worst-case performance of `add()` is $O(n)$

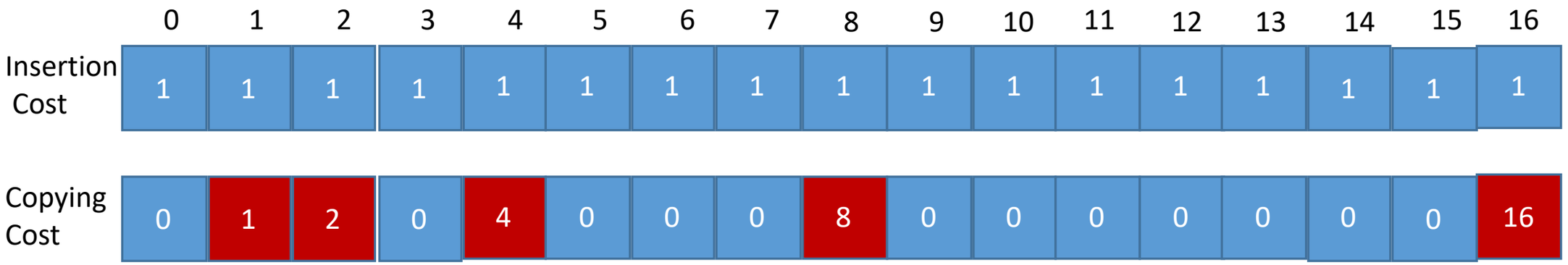
- Cost model: 1 for insertion, n for copying n items to a new array.
- Worst-case: If ArrayList is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- Total cost: $n + 1 = O(n)$.

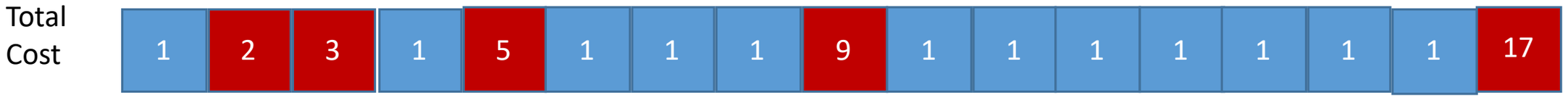
- Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

Amortized analysis

- Amortized cost per operation: for a sequence of n operations, it is the total cost of operations divided by n.
  - Simplest form of amortized analysis called aggregate method. More complicated methods exist, such as accounting (banking) and potential (physicist's).

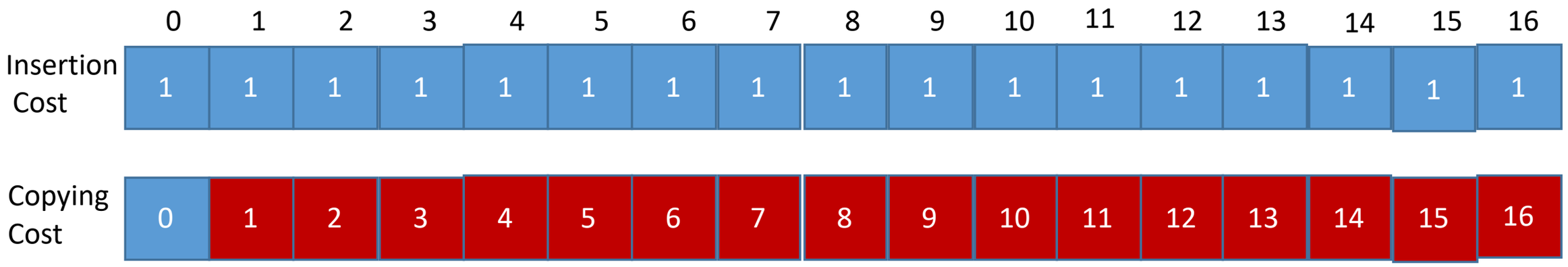
Amortized analysis for n add() operations

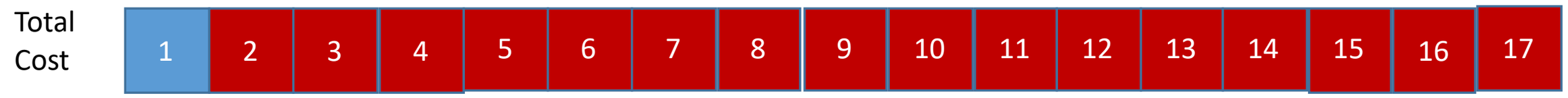




- As the ArrayList increases, doubling happens *half as often* but *costs twice as much.*
- O(total cost)= $\sum$(“cost of insertions”) + $\sum$(“cost of copying”)
- $\sum$(“cost of insertions”) $= n.$
- $\sum$(“cost of copying”) = $1 + 2 + 2^2 + \ldots 2^{\lfloor \log 2^n \rfloor} \leq 2n.$
- O(total cost) $\leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O(1)$, but “lumpy”.

Amortized analysis for n add() operations when increasing ArrayList by 1.





- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") = $1 + 2 + 3 + \ldots + n - 1 = n(n-1)/2$.
- O(total cost) $= n + n(n-1)/2 = n(n+1)/2$, therefore amortized cost is $(n+1)/2$ or $O(n)$.
- Same idea when increasing ArrayList size by a constant.
  - This is why in the lab yesterday, we saw that doubling was the fastest and linear(1) the slowest

Lecture 6: Resizable Arrays

- Background
- ArrayList
- Java Collections
- Theory of Algorithms
- Running Time of ArrayList operations

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)
  - Chapter 1.4 (pages 197–199)
- Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Practice Problems:

- 1.4.1, 1.4.5 - 1.4.7, 1.4.32, 1.4.35-1.4.36.

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)
  - Chapter 1.4 (pages 197–199)
- Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Practice Problems:

- 1.4.1, 1.4.5 - 1.4.7, 1.4.32, 1.4.35-1.4.36.