Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Array-Based Queue

- We’ve learned to implement the queue using a linked list. Now let’s look at an array-based implementation.

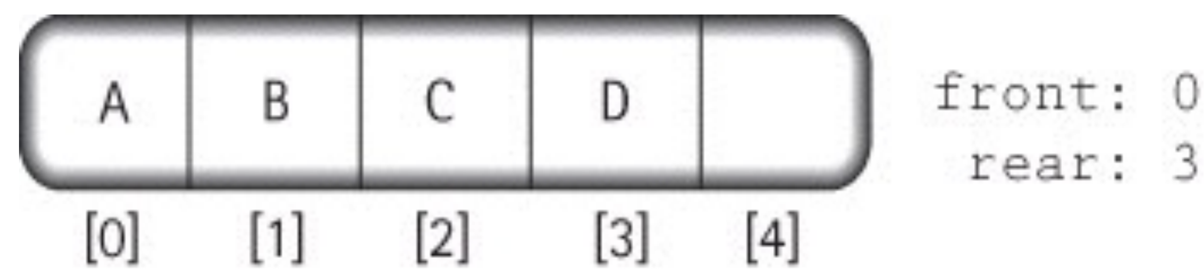
- To begin, we use an array with fixed capacity to store queue elements. We assume **element 0 is always the front**, and use an **rear index to point to the last element in the queue**.

- To **enqueue**, we append the new element at the end, and increment the rear index; to **dequeue**, we return element 0, and move all remaining elements to the left by one position, then decrement the rear index.

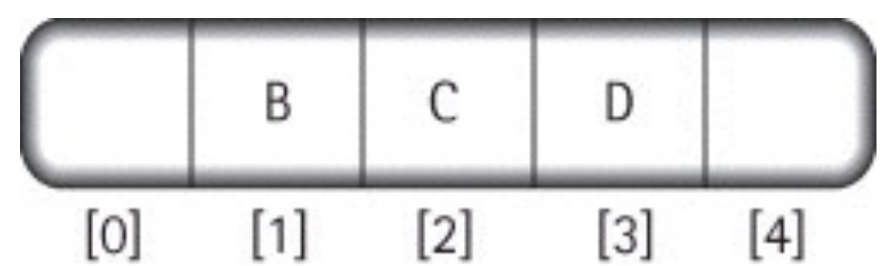
Array-Based Queue: Fixed Front

- Start with an empty queue with capacity 5

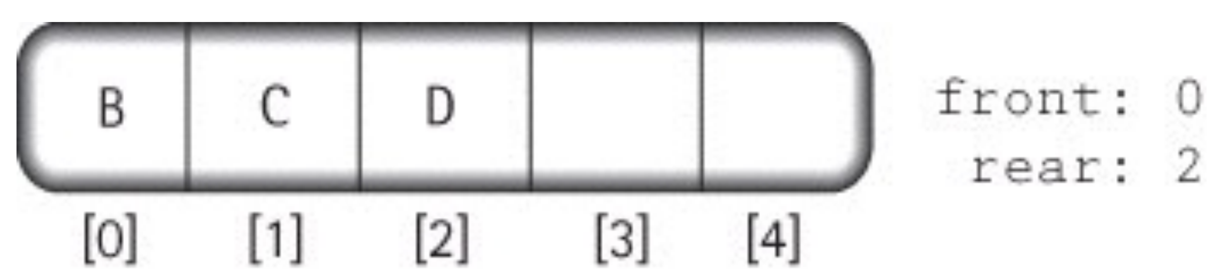
- After enqueuing ‘A’, ‘B’, ‘C’, and ‘D’.



- Dequeue the front element.



- Move the remaining elements to the left by one spot.



Clicker Question #1

Using the algorithm in the previous slides (i.e. fixed front array-based queue), what’s the running cost of the enqueue and dequeue operations. Assume the queue has N elements, and a capacity of M?

(a) enqueue is O(1), dequeue is O(M)

(b) enqueue is O(1), dequeue is O(N)

(c) enqueue is O(N), dequeue is O(N)

(d) enqueue is O(M), dequeue is O(1)

(e) enqueue is O(M), dequeue is O(N)

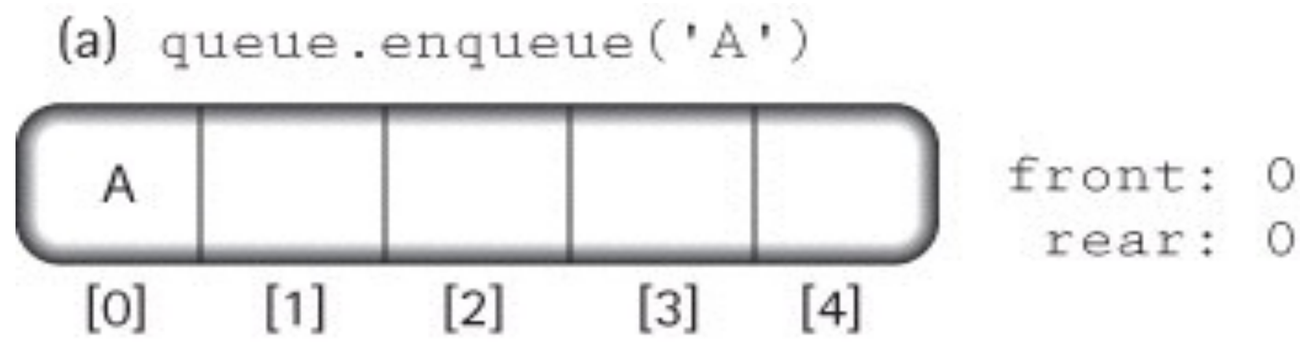
A Circular Queue

- With fixed front design, dequeue takes O(N), which is inefficient. To make it more efficient, we remove the requirement that the front is always at index 0. Instead, we will allow it to 'float'.

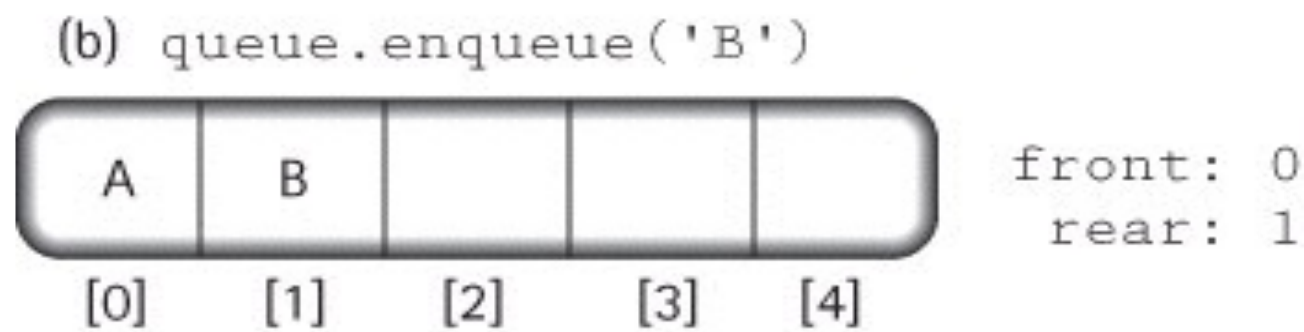
- We keep a **front index** to point to the current front element, and a **rear index** to point to the current rear.

- To **enqueue**, we add a new item at the rear and increment the rear index. To **dequeue**, we remove the front item and **increment the front index**.

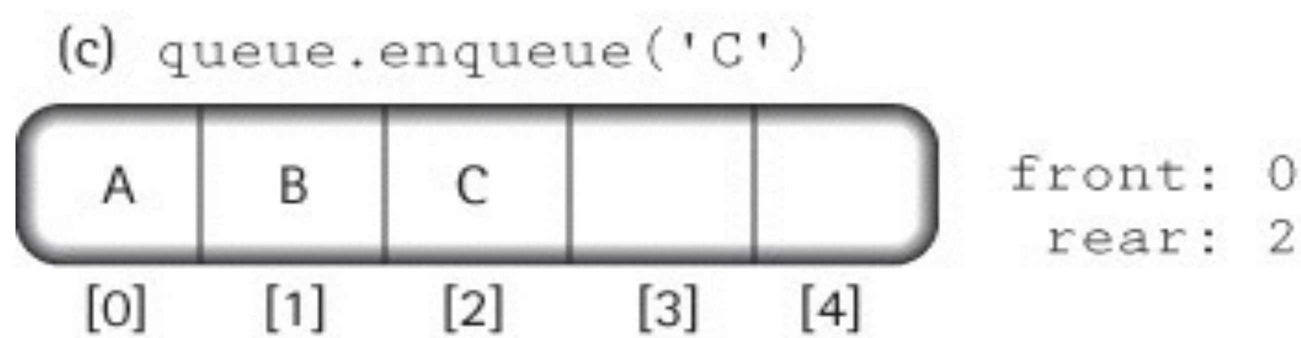
- enqueue A



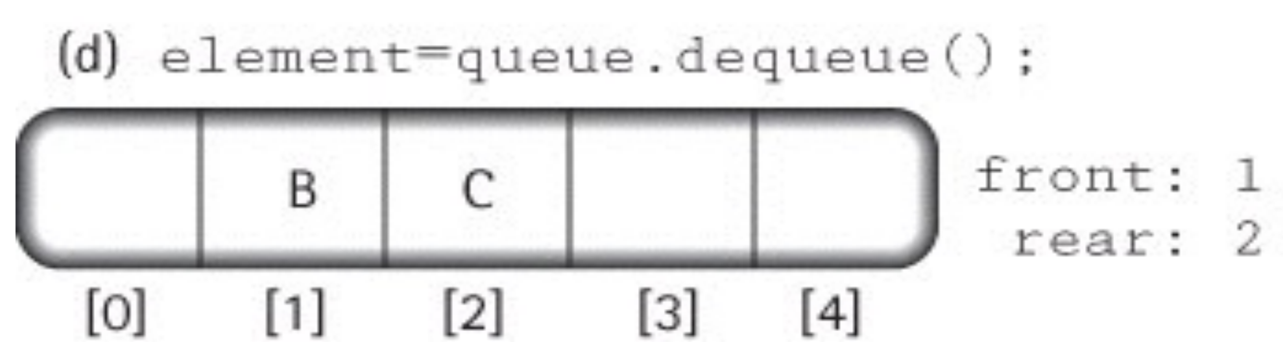
- enqueue B



- enqueue C



- dequeue

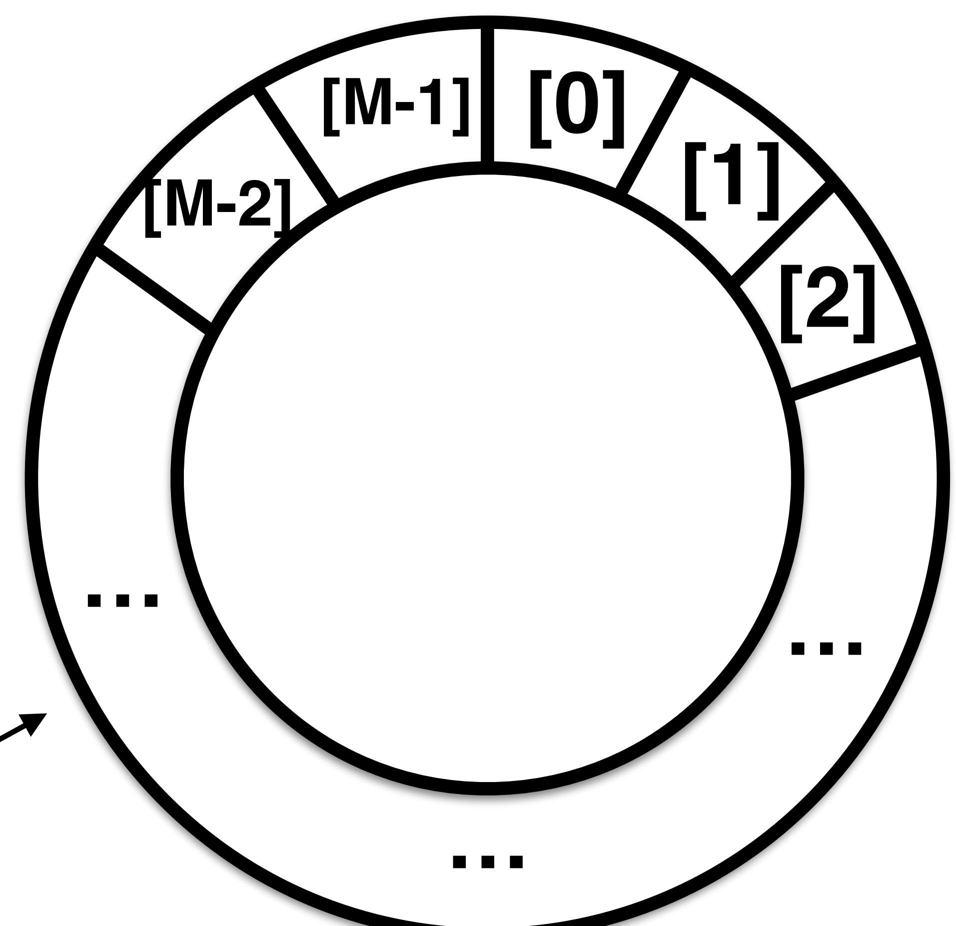


A Circular Queue

- Note that after dequeuing an element, that spot (e.g. index 0) becomes available. So if you continue to enqueue, (say D, E, F), element F can be stored at index 0.

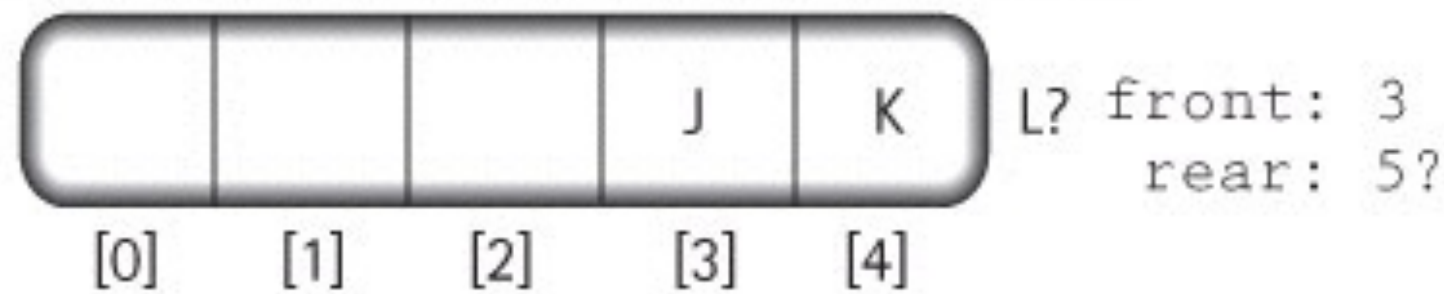
- Imagine the array is **circular** (i.e. the end of the array wraps back to the beginning of the array). Hence it's called a **circular queue**.

A circular queue of capacity M

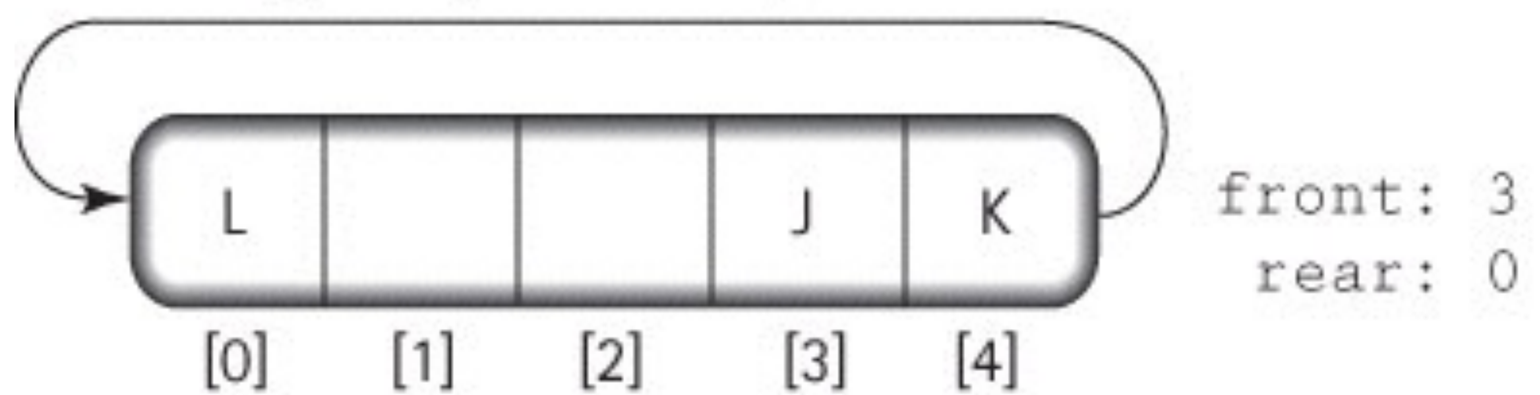


A Circular Queue

(a) There is no room at the end of the array



(b) Using the array as a circular structure, we can wrap the queue around to the beginning of the array



- enqueue L

A Circular Queue

- Considering the ‘wrap-around’, how do we increment the rear index when enqueuing?

```
if (rear == capacity - 1)
  rear = 0;
else
  rear = rear + 1;
```

- Is there a simpler way? Hint: using modulo?

A Circular Queue

- Considering the ‘wrap-around’, how do we increment the rear index when enqueuing?

```
rear = (rear + 1) % capacity;
```

- Verify if it’s correct when `rear` is (`capacity - 1`).
- In general, when you do `x % capacity`, as long as `x` is non-negative, the result is always between `0` and (`capacity - 1`).
- Careful: **if x is negative, the result will be negative**!
  - this is not true in other languages (python etc).

Clicker Question #2

If we want to **<u>decrement</u>** the `rear` index, which of the following code is correct, assuming a circular queue?

```
(a) rear = (rear - 1);
(b) rear = (rear - 1) % capacity;
(c) rear = (rear - 1 + capacity) % capacity;
(d) rear = (rear - 1) % capacity + 1;
(e) rear = (rear - 1) % capacity + capacity;
```

A Circular Queue

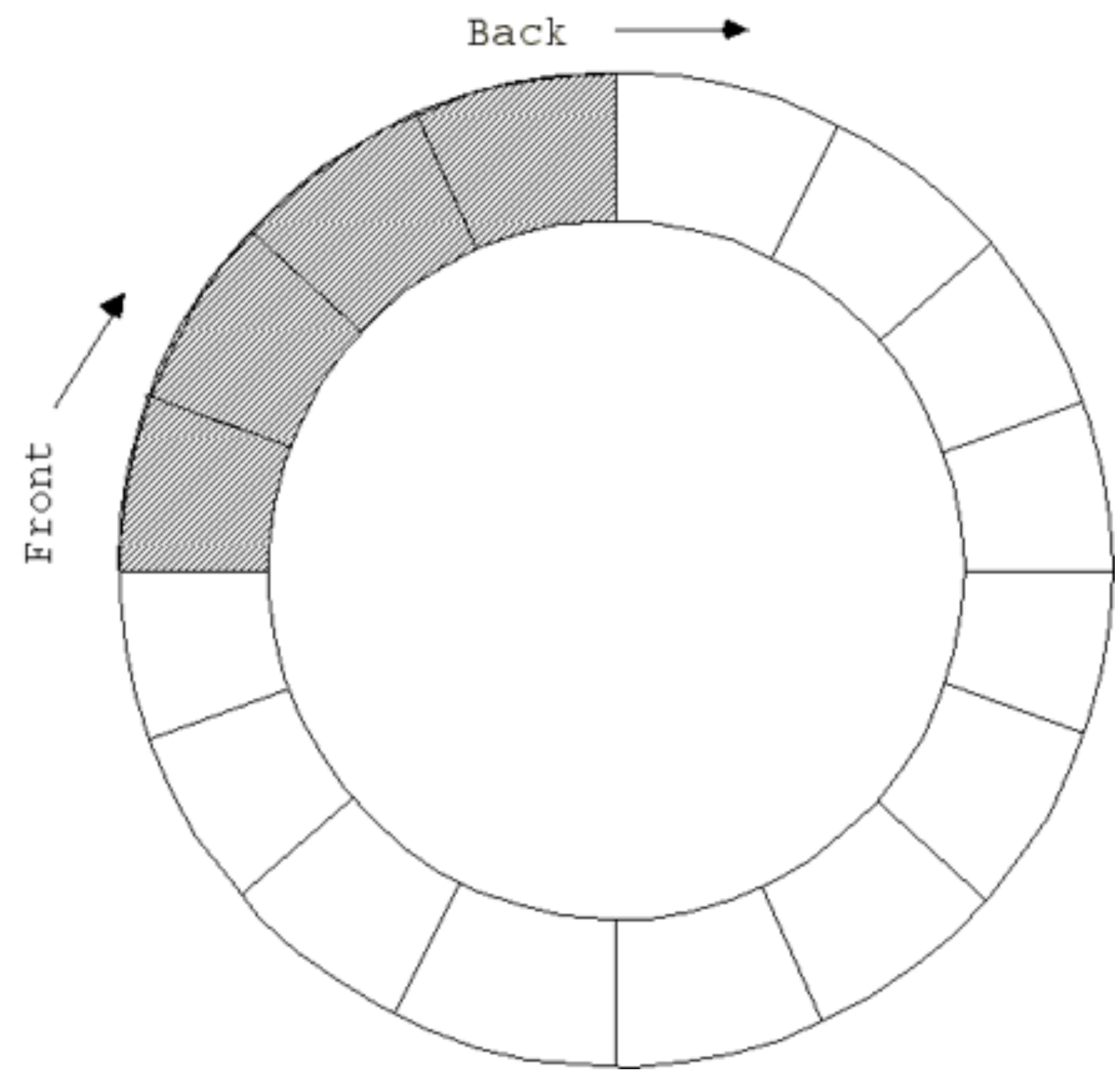
- As `front` index points to the front element, and `rear` index points to the rear element, when they are equal, there is exactly one element. For example, when they are both equal to 0 (or both equal to 1 and so on), there is one element.

- At the beginning, when the queue is empty, we initialize `front = 0`, and `rear = capacity-1`.

- This creates ambiguity, why? Thank about the values of `front` and `rear` when the queue is full. How do we address this?

A Circular Queue

- At any point, the elements starting from the `front` index to the `rear` index (in a circular fashion) constitute the queue elements.



Clicker Question #3

- Assuming the number of queue elements is less than `capacity` (i.e. ignoring the ambiguity), which of the following expressions correctly calculates the number of elements in the queue? (Hint: consider some different combinations of `front` and `rear`).

```
(a) (rear - front) % capacity

(b) (rear - front + 1) % capacity

(c) (front - rear + 1) % capacity

(d) none of the above
```

Coding Queue Operations

```
public boolean isEmpty() {
    return (numElements == 0);
}
public boolean isFull() {
    return (numElements == queue.length);
}
```

Coding Queue Operations

```
public void enqueue (T element) {
    if (isFull())
        throw new QOE(“add to full queue”);
    else {
        rear = (rear + 1) % queue.length;
        queue[rear] = element;
        numElements++;
    }
}
```

Coding Queue Operations

```
public T dequeue () {
    if (isEmpty())
        throw new QUE(“dequeue from empty”);
    else {
        T toReturn = queue[front];
        queue[front] = null;
        front = (front + 1) % queue.length;
        numElements--;
        return toReturn;
    }
}
```

Java's `ArrayList` Class

- It would be nice if the array capacity is not fixed!
- At the end of the Linked List lecture, we briefly mentioned arrays that can dynamically expand, thus overcoming the 'fixed capacity' limitation.
- Java provides several variable-length generic array structures, such as **`ArrayList<T>`**. It begins with some fixed capacity, but the capacity is reached, it copies itself into another array of twice the size, thus *appears* to be an array of unlimited size.

Java's `ArrayList` Class

- What's this doubling-the-capacity business?
  - Let's say we have an initial ArrayList of capacity 10, and we keep adding elements to it.
  - Adding the 11th element causes the ArrayList to allocate a new array of capacity 20, copy the existing 10 elements and the 11th element to the new array, then release the old array.

Java's `ArrayList` Class

- What's this doubling-the-capacity business?
  - Let's say we have an initial ArrayList of capacity 10, and we keep adding elements to it.
  - Adding the 11th element causes the ArrayList to allocate a new array of capacity 20, copy the existing 10 elements and the 11th element to the new array, then release the old array.
  - Same thing happens as the 21th element is added.
  - By the time we have 81 elements we have done four capacity 'expansions': to 20, 40, 80, and 160.

Unbounded Array-Based Queue

- You can use Java's `ArrayList` to implement Array-based Queue. Then it won't be bounded and hence can implement the `UnboundedQueueInterface`.

- Or you can write an array **expand** method yourself and call it inside the enqueue method. This way, the array will be expanded if enqueue is called when the array is already full.

Clicker Question #4

- Assume a dynamic array has a capacity of 1 to begin with. Each time it expands, we double its capacity. Starting from an empty array, we add one element at a time, until there are N elements. **How many times would the array have expanded**? (number of times the expand method were called)

(a) O(1)

(b) O(N)

(c) O(log N)

(d) O(N^2)