Review

Queue (FIFO) vs. Stack (LIFO)

Enqueue vs. Push

Dequeue vs. Pop

Array-Based Queue

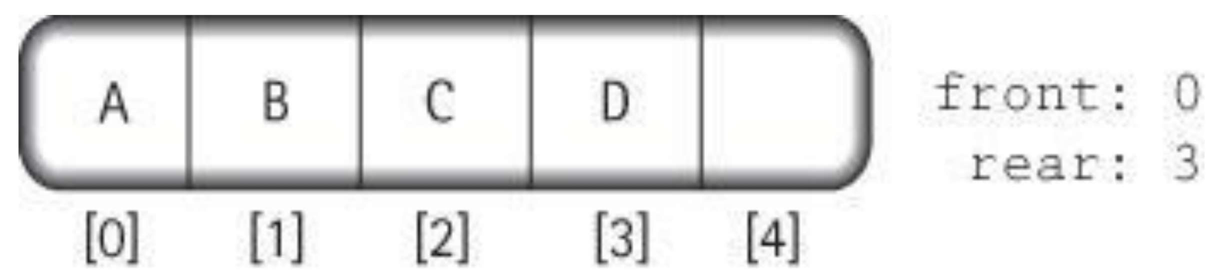
- We've learned to implement the queue using a linked list. Now let's look at an array-based implementation.

- The simplest way is to use an array with fixed capacity to store queue elements. Here **element 0 is always the front**; and we use an **rear index to point to the last element in the queue**.

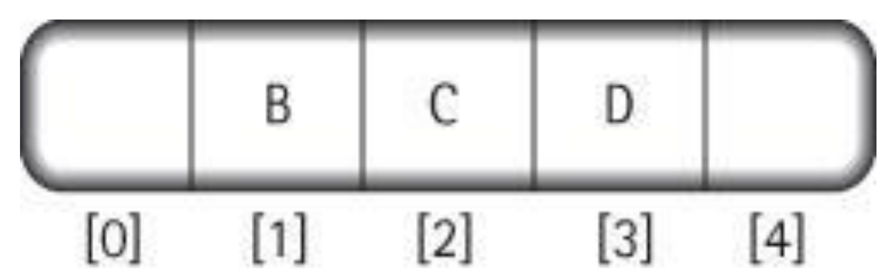
- To **enqueue**, we increment the rear index and append the new element at the end; to **dequeue**, we return element 0, and move all remaining elements to the left by one position, then decrement the rear index.

Array-Based Queue: Fixed Front

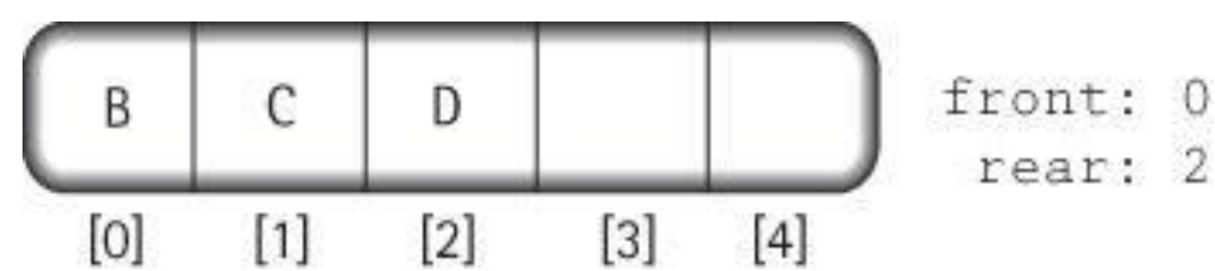
- Start with an empty queue with capacity 5
- After enqueuing 'A', 'B', 'C', and 'D'.



- Dequeue the front element.



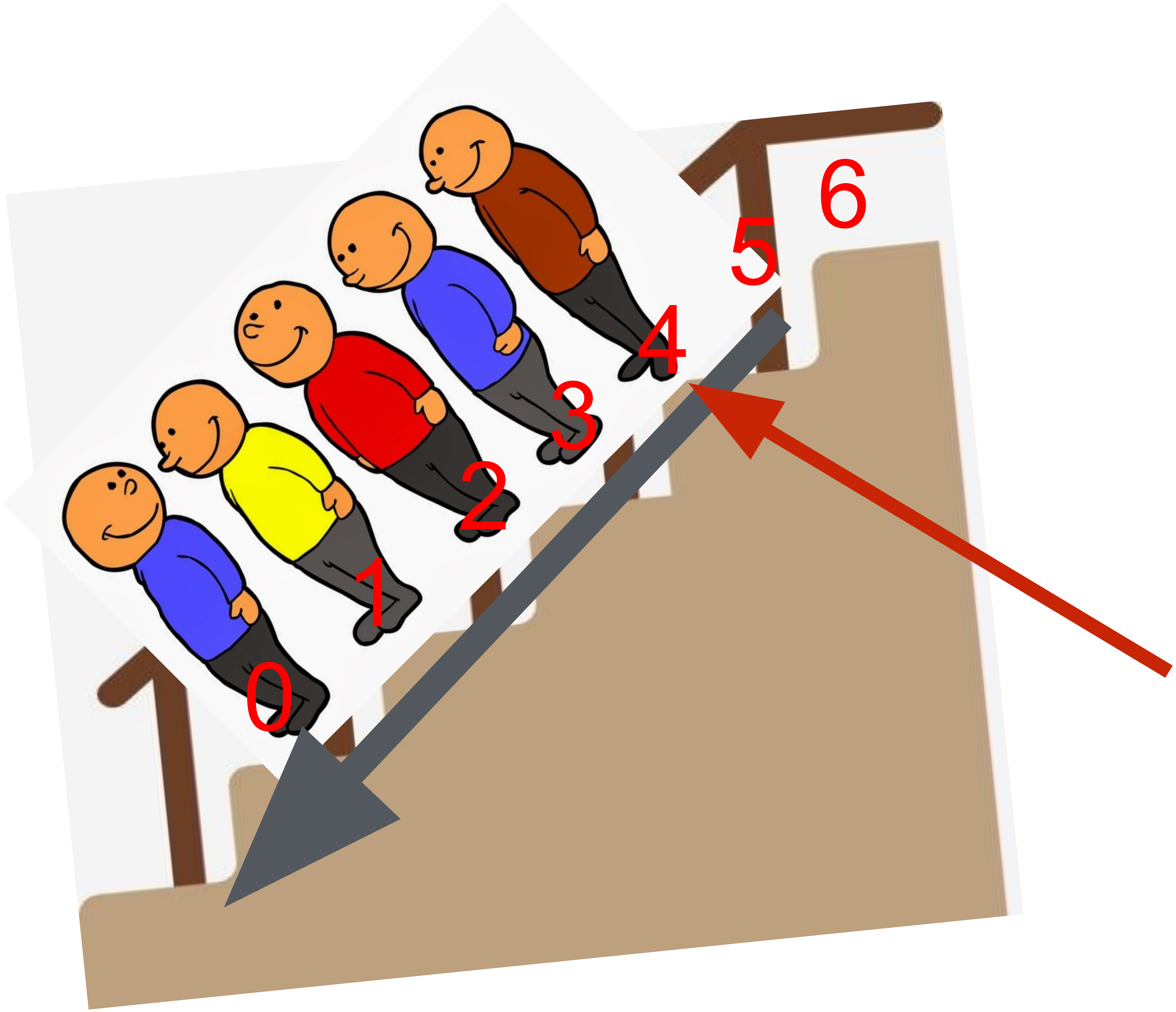
- Move the remaining elements to the left by one spot.

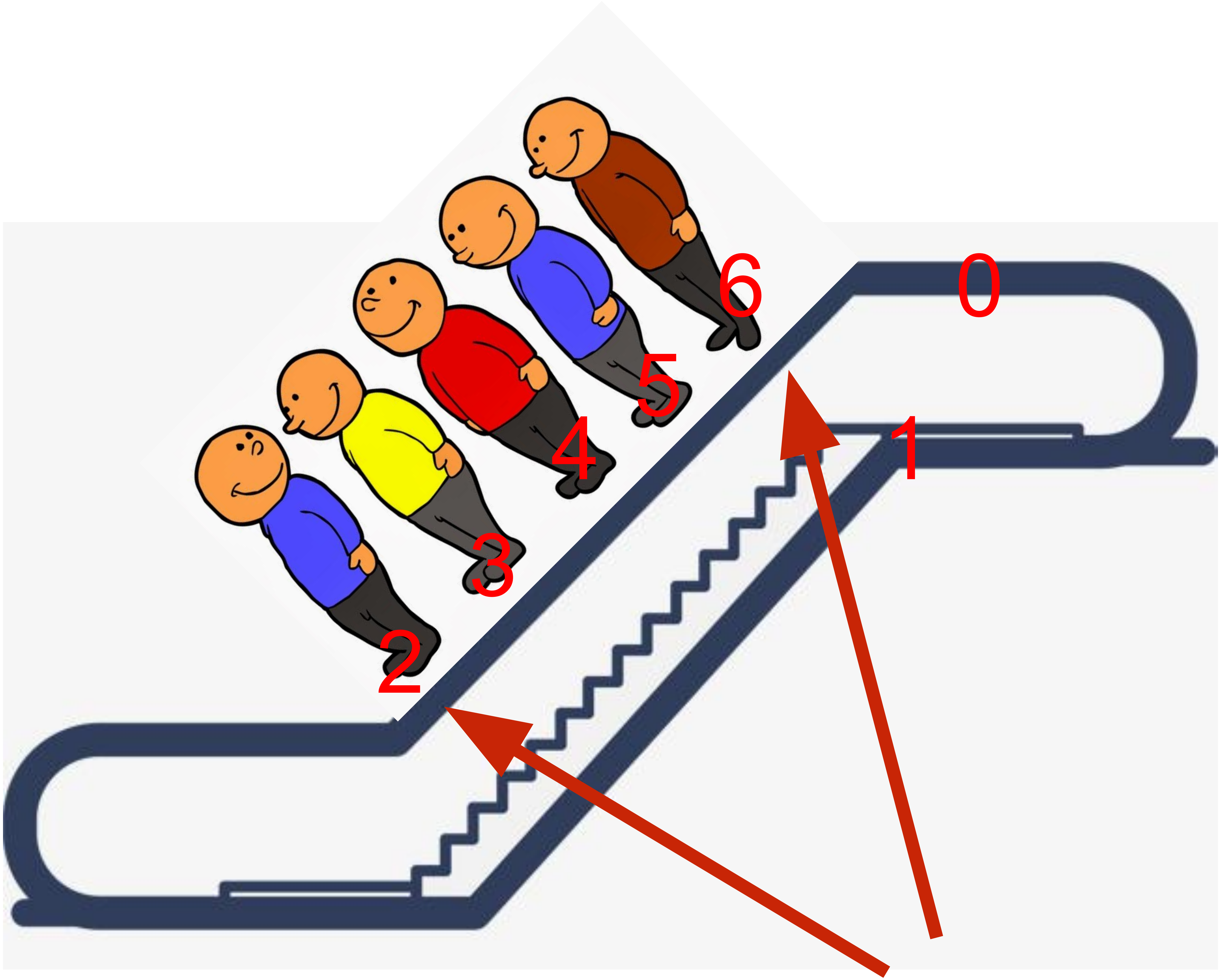


Clicker Question #1

Using the algorithm in the previous slides, what are the big-O costs of the **enqueue** and **dequeue** operations? Assume the queue has N elements, and a capacity of C.

```
(a)  enqueue is O(1), dequeue is O(N)

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

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

(d)  enqueue is O(N), dequeue is O(C)

(e)  enqueue is O(1), dequeue is O(1)
```


0
1
2
3
4
5
6


0
1
2
3
4
5
6

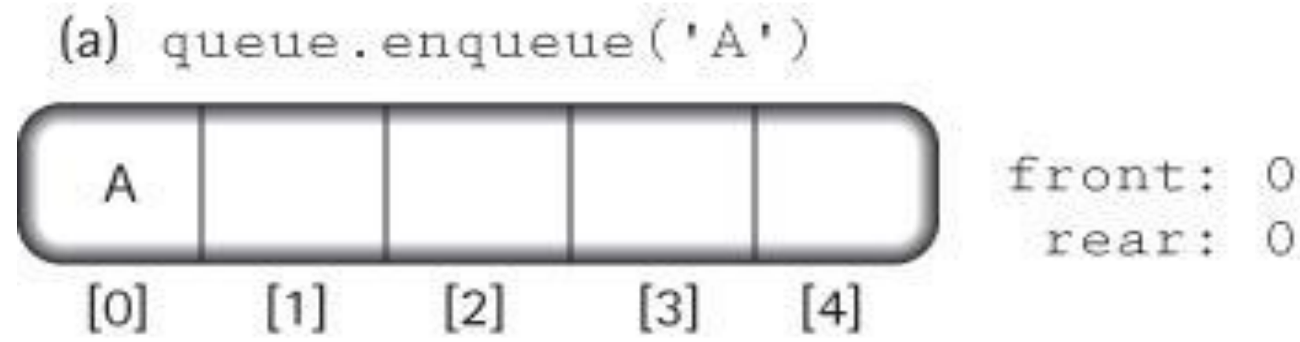
A Circular Queue

- With the fixed frontal element design, **dequeue** takes O(N), which is not efficient. We can make it more efficient by removing the requirement that the front is always at index 0. Instead, we allow it to ‘float’.

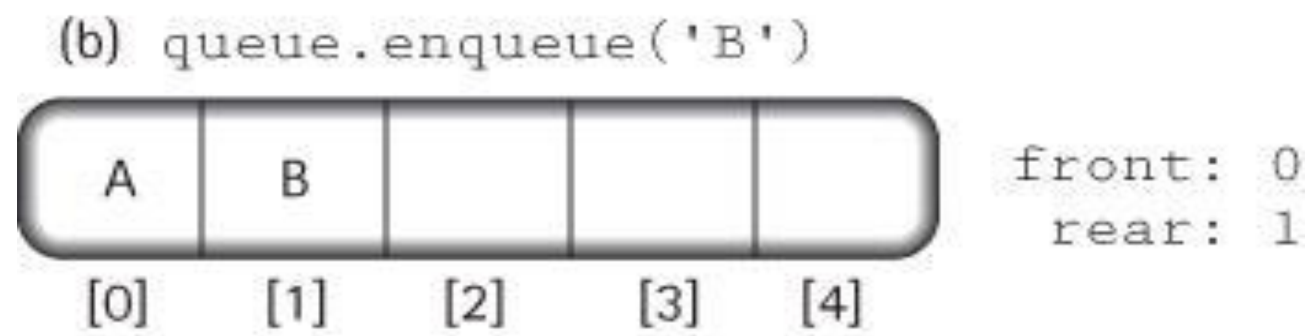
- To do so, we keep a **front index** to point to the current front element, and a **rear index** to point to the current rear element.

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

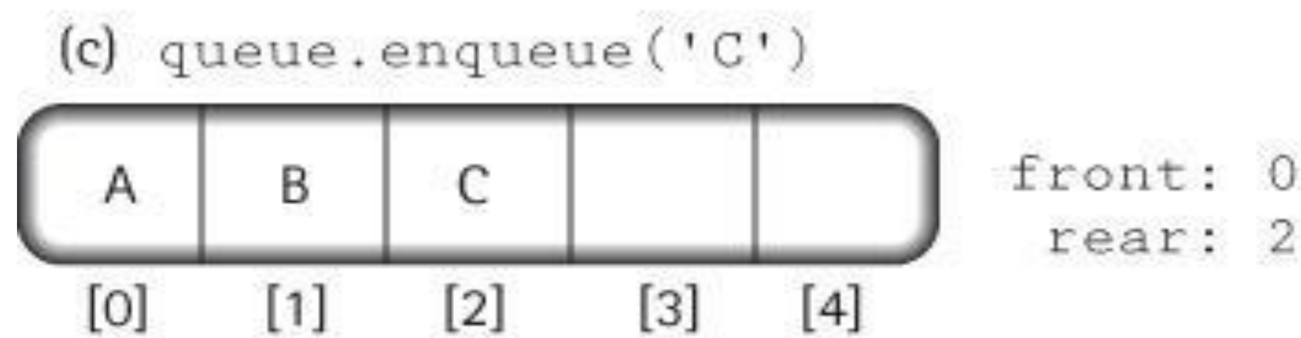
- enqueue A



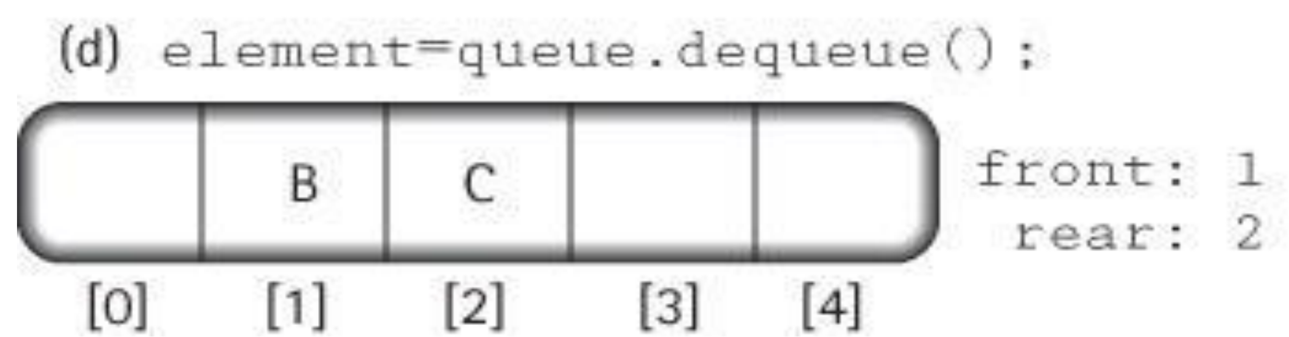
- enqueue B



- enqueue C



- dequeue

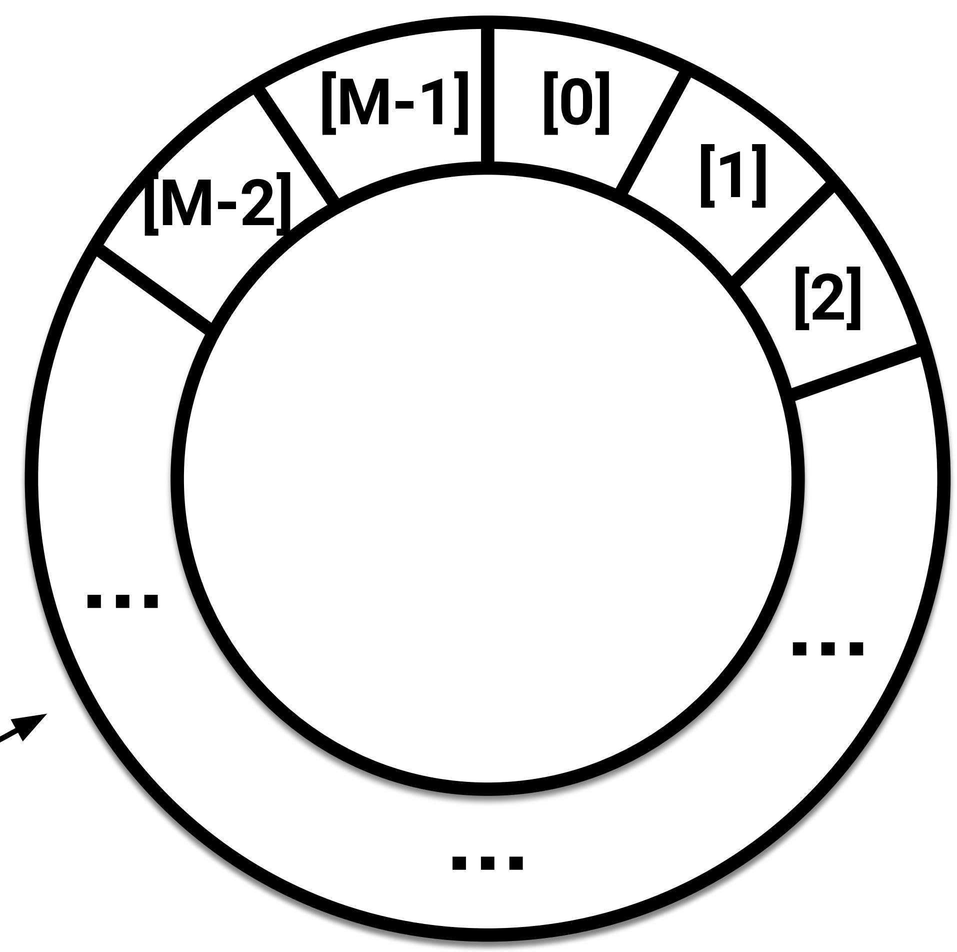


A Circular Queue

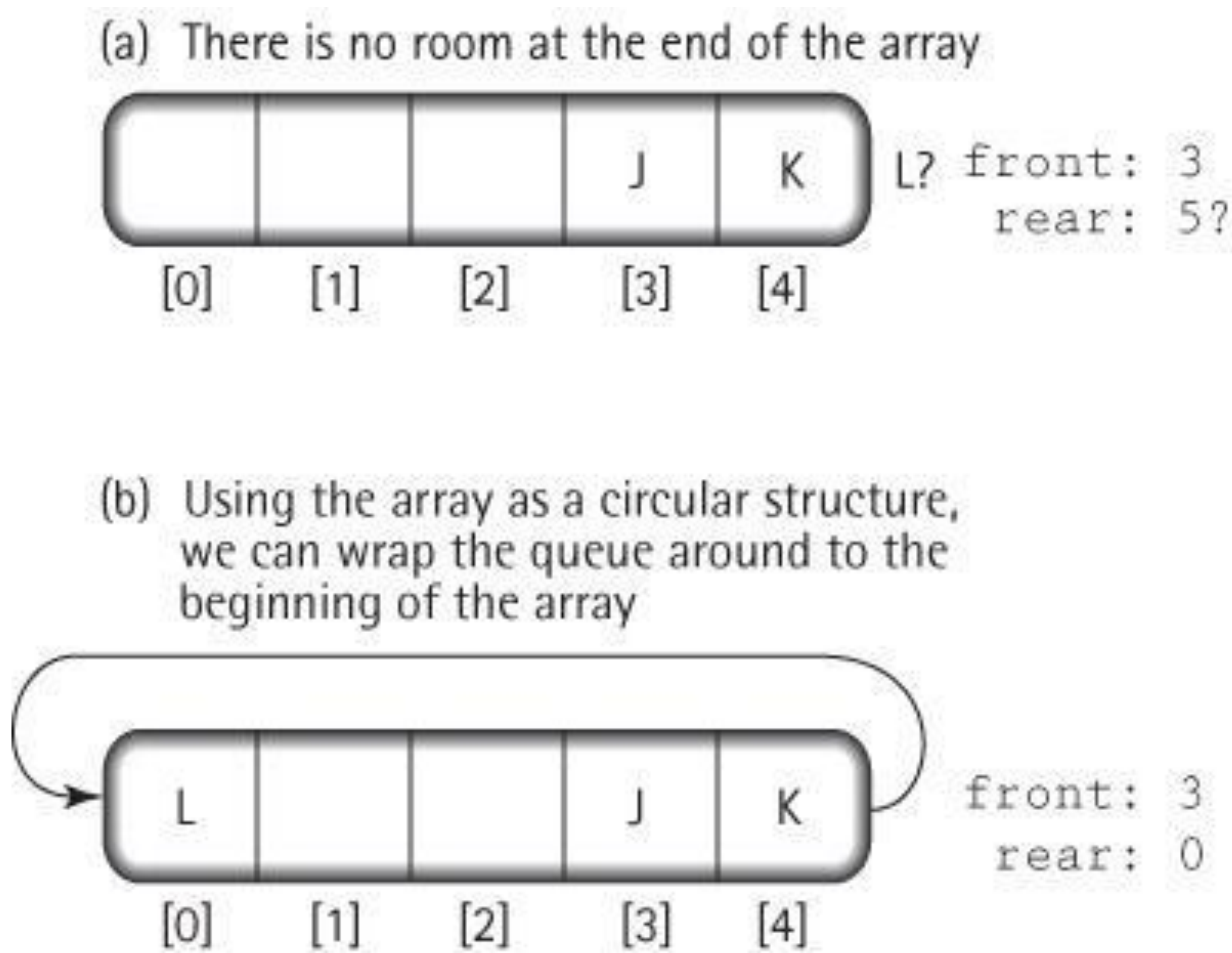
- Note that after dequeuing an element, that spot (e.g. index 0) becomes available again. So if you continue to enqueue, (say D, E, F), element F can be stored at index 0.

- 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



- enqueue L

A Circular Queue

- Considering the capability of ‘wrap-around’, how do we implement this?

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

But is there a more elegant way? Hint: how about use **modulo**?

A Circular Queue

- The following line of code achieves the same:

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

- It's easy to verify the correctness. For example, when `rear` is (`capacity-1`) vs. when it's not.

- When you do (`x % capacity`), as long as x is non-negative, the result is always between `0` and (`capacity - 1`). This makes sure your code does not generate `IndexOutOfBoundsException`.

  - *In Java, (x % capacity) is negative if x is negative.*

Clicker Question #2

In a circular queue of capacity 6, if at the moment, `rear = 2`, and `front = 4`, how many elements are in the queue?

(a) 2

(b) 3

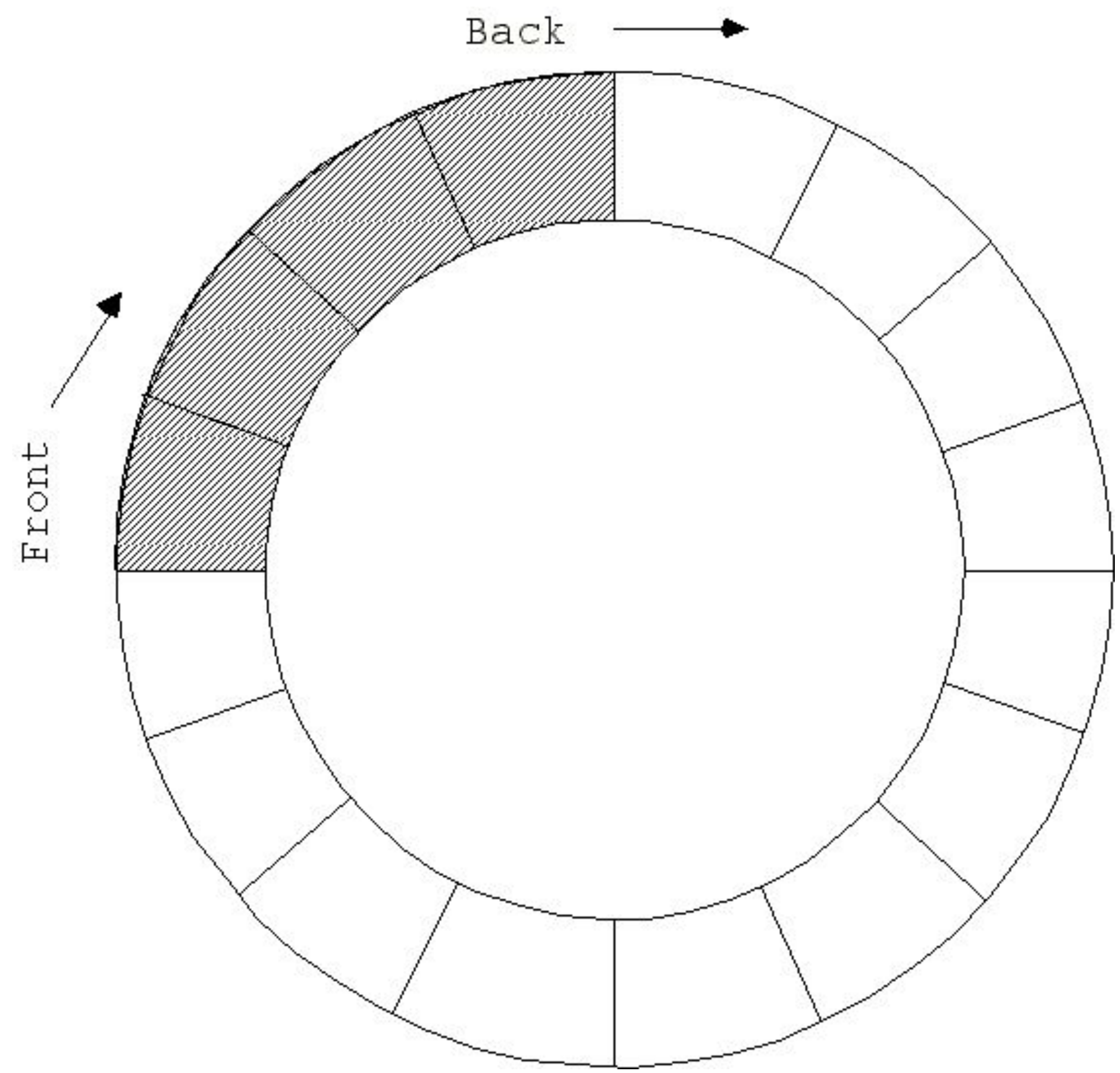
(c) 4

(d) 5

(e) 6

A Circular Queue

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



A Circular Queue

- When the front pointer and rear pointer 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.

- In the beginning, when the queue is just created and hence empty, we should initialize `front = 0`, and `rear = capacity-1`. We should not initialize rear to 0, for the obvious reason as above.

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

Clicker Question #3

In a circular queue, if **front index is 7** and the **queue has 4 elements**, the **rear** index may be:

```
(a) either 2 or 10
(b) 10 or more than 10
(c) any of 0, 1, 2, or 10
(d) any of 0, 1, 2, 3 or 10
(e) anywhere between 2 and 10
```

Coding the Circular Queue

```java
public class ArrayQueue<T> implements QueueInterface<T> {
    protected final int DEFCAP = 100;
    protected T[] queue;
    protected int numElements = 0;
    protected int front = 0, rear;
    public ArrayQueue(int maxSize) {
        queue = (T[]) new Object[maxSize];
        rear = maxSize - 1; // virtual position -1
    }
    public ArrayQueue() {
        this (DEFCAP);  // default capacity
    }
```

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 QueueOverflowException();
    else {
        rear = (rear + 1) % queue.length;
        queue[rear] = element;
        numElements++;
    }
}
```

Coding Queue Operations

```
public T dequeue () {
    if (isEmpty())
        throw new QueueUnderflowException();
    else {
        T toReturn = queue[front];
        queue[front] = null; // release memory
        front = (front + 1) % queue.length;
        numElements--;
        return toReturn;
    }
}
```

Java’s `ArrayList` Class

- Given how arrays are more friendly to work with than linked lists, it would be nice if its capacity is not fixed!

- This can be done by dynamically re-allocating a new, larger array at run-time to accommodate more elements.

- Java provides several variable-length generic array structures, such as **`ArrayList<T>`**. It begins with some fixed capacity, but the capacity is reached, it allocates a new array, twice the size of the original, and copies the elements over. This makes it appear to have unlimited capacity.

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.

Dynamic-size Array

```
public class DynamicArray<T> {
  private T a[];   // a generic array storing elements
  private int nelements;  // number of elements

  public DynamicArray() {
    a = (T[]) new Object[100]; // initial capacity 100
    nelements = 0;
  }

  public void append(T elem) { // append a new element
    if(nelements == a.length) {








}
```

Dynamic-size Array

```
public class DynamicArray<T> {
  private T a[];   // a generic array storing elements
  private int nelements;  // number of elements

  public DynamicArray() {
    a = (T[]) new Object[100]; // initial capacity 100
    nelements = 0;
  }

  public void append(T elem) { // append a new element
    if(nelements == a.length) {
      T[] na = (T[]) new Object[a.length*2]; // double capacity
      for(int i=0; i<a.length; i++) // copy elements over
          na[i] = a[i];
      a = na; // assign new array to a
     }
     a[nelements++] = elem; // append the new element
}
```

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**? (i.e. number of times we allocate new arrays)

(a) O(1)

(b) O(log(N))

(c) O(N)

(d) O(N*log(N))

(d) O($\log^2$(N))

Questions

Queueing

- In **producer-consumer** settings:
  - Each producer generates new tasks (or elements) to be processed: Enqueue!
  - Each consumer serves / consumes the elements: Dequeue!
  - There may be multiple producers and multiple consumers.
  - Elements may be produced / consumed at different rates. A queue serves as a buffer to handle mismatched rates.