

COSC 222 Data Structure

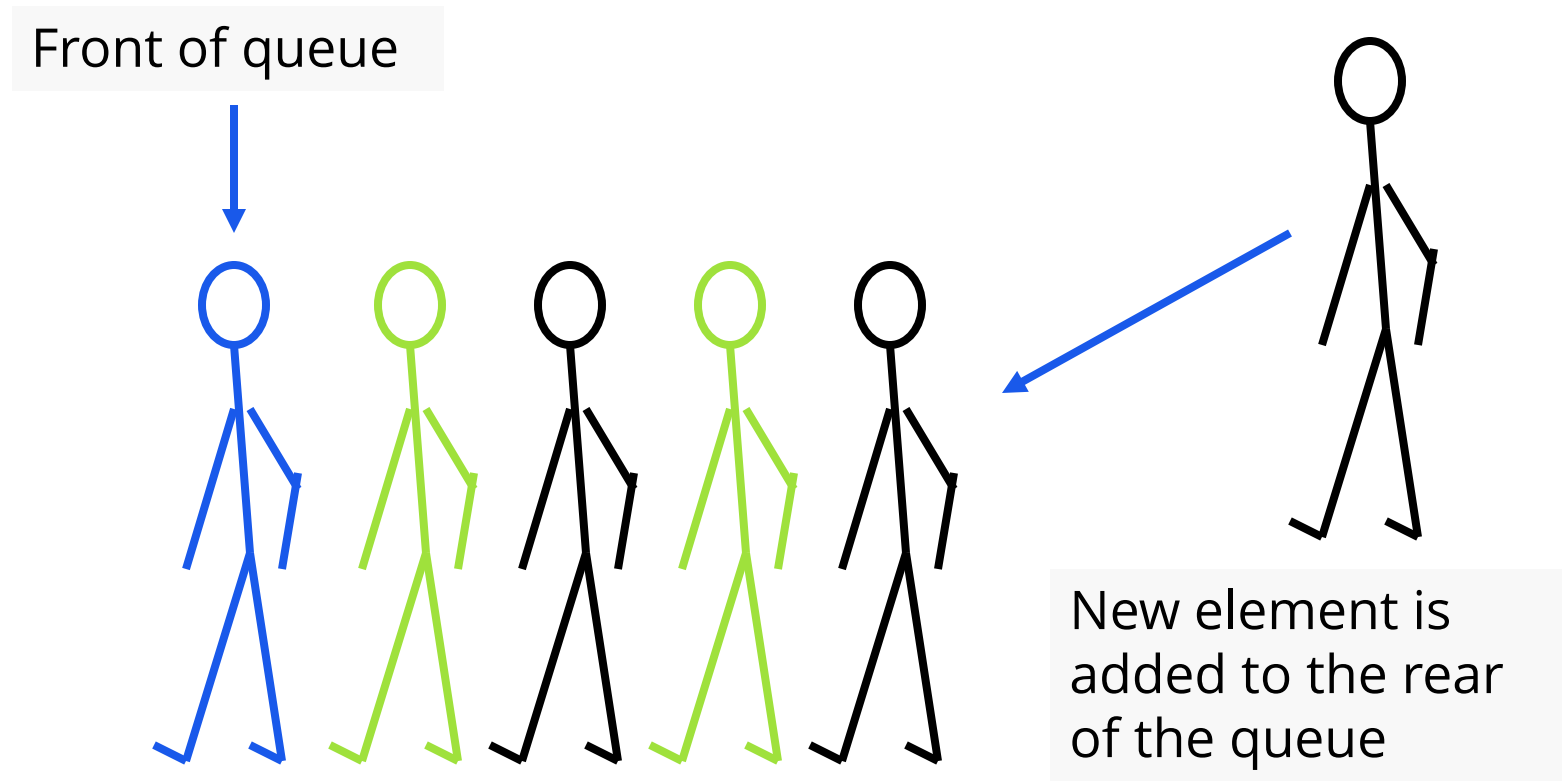
Queue

Queues

- **Queue**: a collection whose elements are
 - added at one end (the **rear** or **tail** of the queue) and
 - removed from the other end (the **front** or **head** of the queue)
- A queue is a **FIFO** (first in, first out) data structure

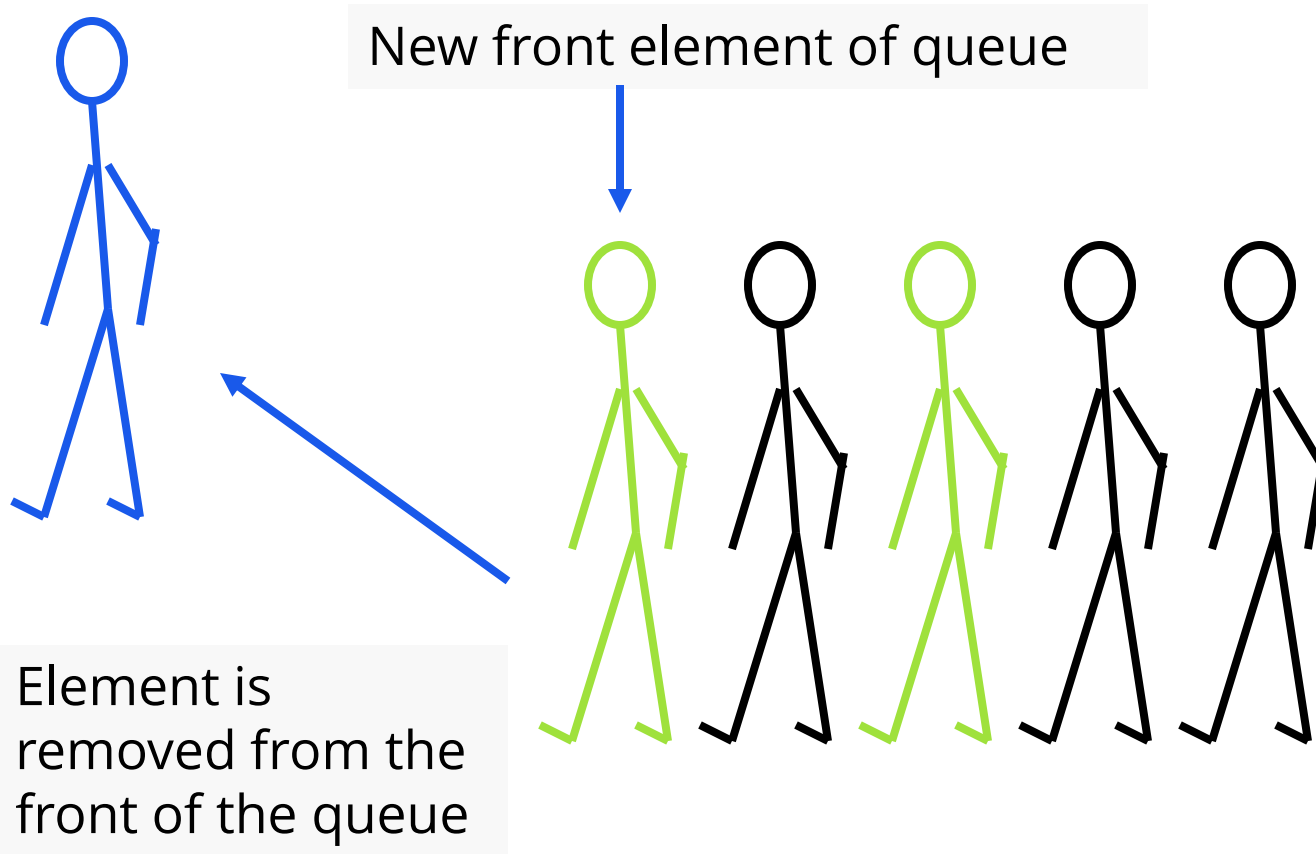
Conceptual View of a Queue

- Adding an element



Conceptual View of a Queue

- Removing an element



Queues Applications

- Operating systems / Computing:
 - Queue of processes to be run
 - Access to shared resources (e.g., printer)
 - Keyboard input buffer
- Real world examples:
 - People waiting in a line
 - Cars at a gas station
 - etc.



Queue operations

Operation	Description
enqueue	Adds an element to the rear of the queue
dequeue	Removes an element from the front of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue
toString	Returns a string representation of the queue

Interface to a Queue in Java

```
public interface QueueADT<T> {  
    // Adds one element to the rear of the queue  
    public void enqueue (T element);  
    // Removes and returns the element at the front of the  
    queue  
    public T dequeue( );  
    // Returns without removing the element at the front of  
    the queue  
    public T first( );  
    // Returns true if the queue contains no elements  
    public boolean isEmpty( );  
    // Returns the number of elements in the queue  
    public int size( );  
    // Returns a string representation of the queue  
    public String toString( );  
}
```

Queue Implementation Issues

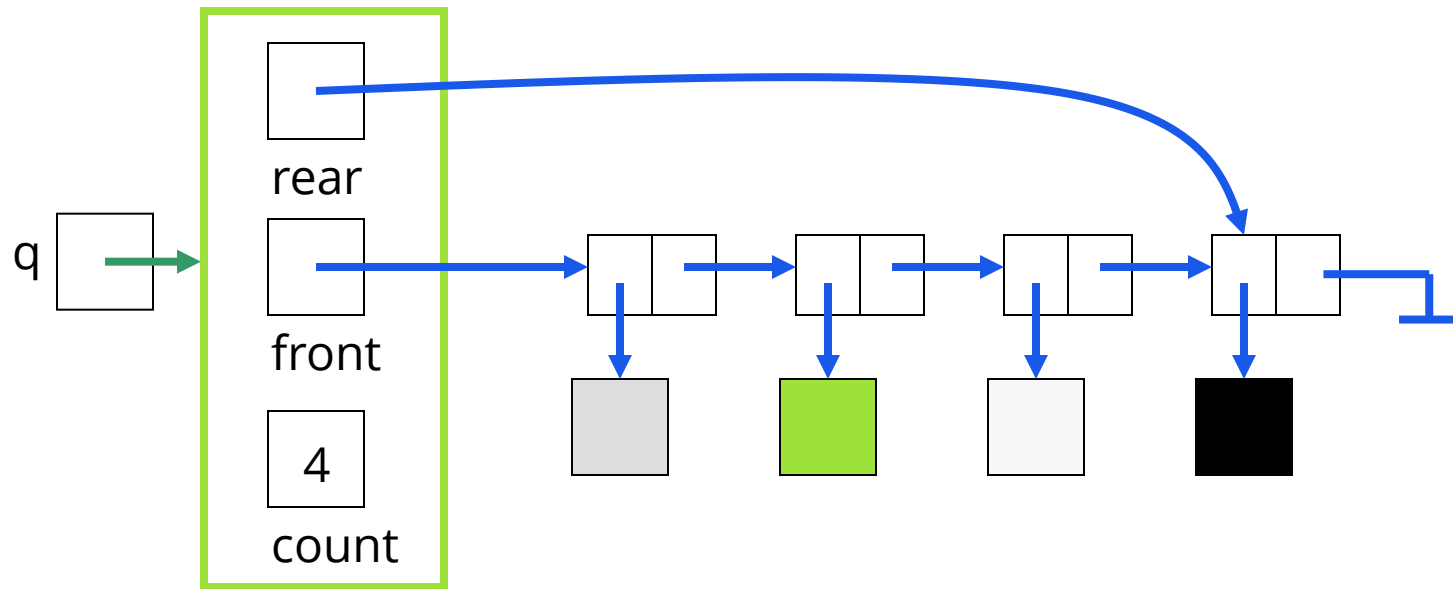
- What do we need to implement a queue?
 - A data structure (**container**) to hold the data elements
 - Something to indicate the **front** of the queue
 - Something to indicate the **end** of the queue

Queue Implementation Using a Linked List

- Queue can be represented as a **linked list of nodes**, with each node containing a data element
- We need two pointers for the linked list
 - A pointer to the beginning of the linked list (**front** of queue)
 - A pointer to the end of the linked list (**rear** of queue)
- We will also have a **count** of the number of items in the queue

Linked Implementation of a Queue

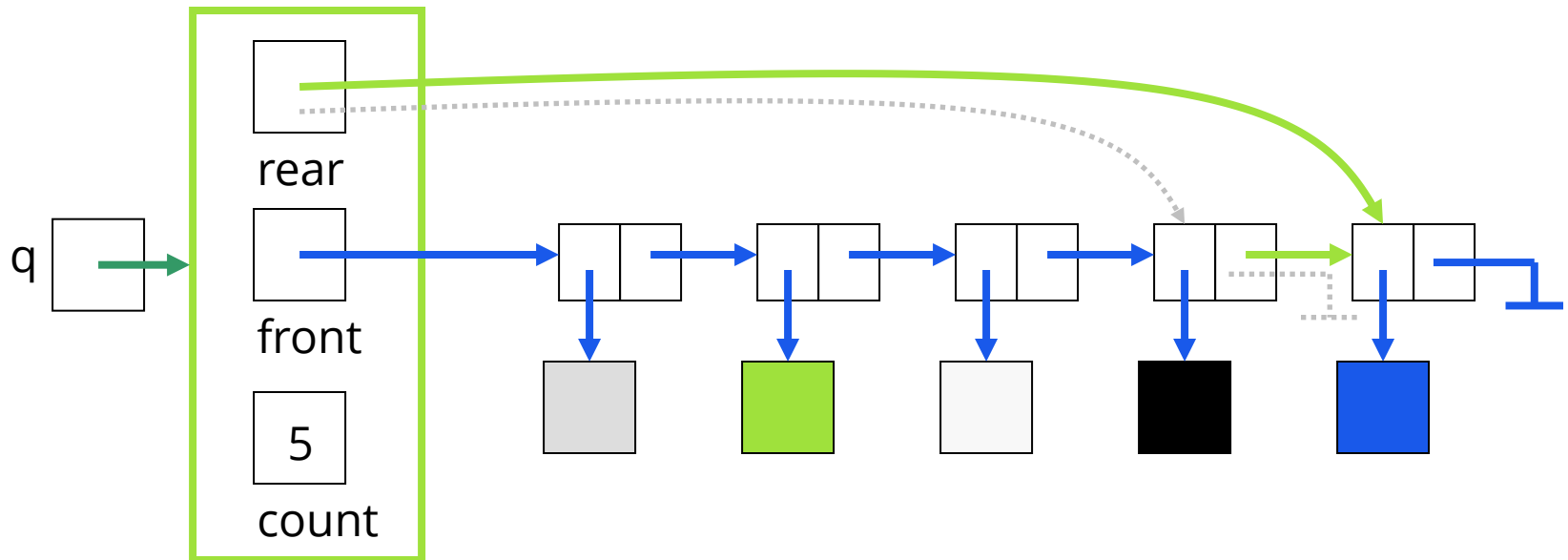
A queue *q* containing four elements



Queue After Adding Element




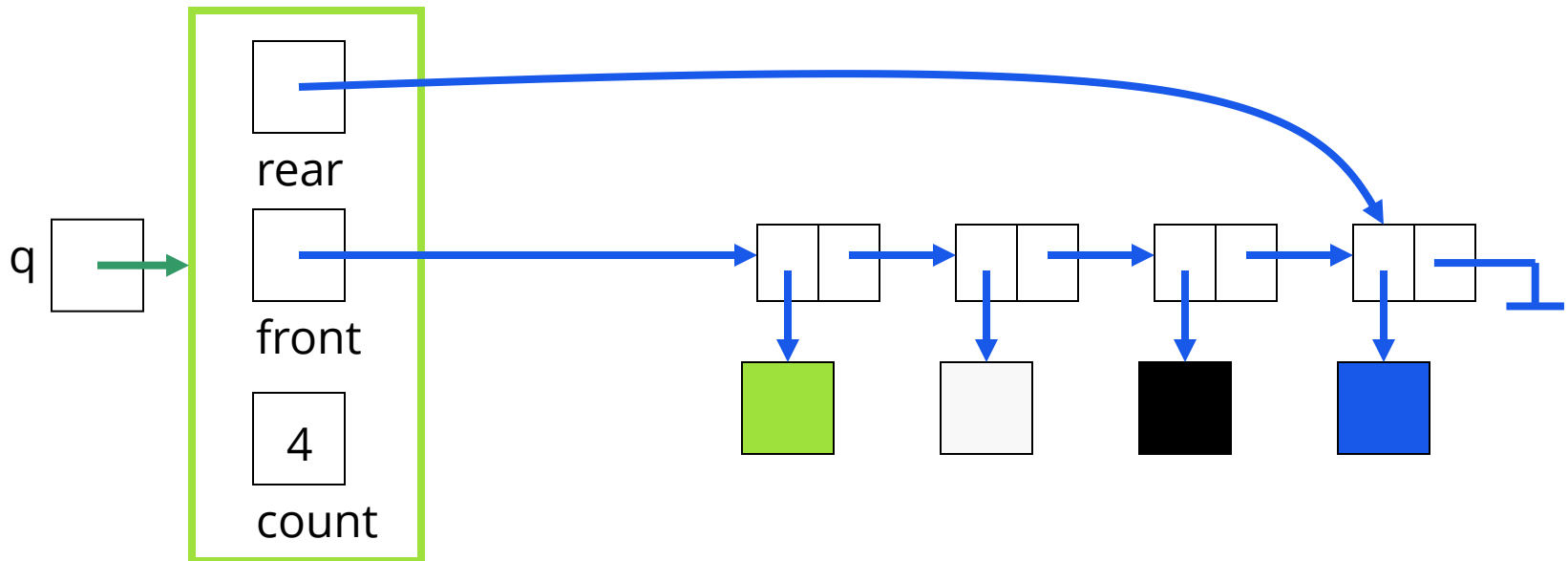
New element is added in a node at the end of the list, rear points to the new node, and count is incremented



Queue After a dequeue Operation



Node containing  is removed from the front of the list (see previous slide), front now points to the node that was formerly second, and count has been decremented.



Java Implementation

- The queue is represented as a linked list of nodes:
 - Example implementation: **QNode** class

```
public class QNode<T> {  
    private T data;  
    private QNode<T> link;  
  
    public QNode(T initData, QNode<T> initLink){  
        data = initData;  
        link = initLink;  
    }  
  
    public T getData() {return data;}  
    public QNode<T> getLink() {return link;}  
    public void setData(T data) {this.data = data;}  
    public void setLink(QNode<T> n) {link = n;}  
}
```

Java Implementation

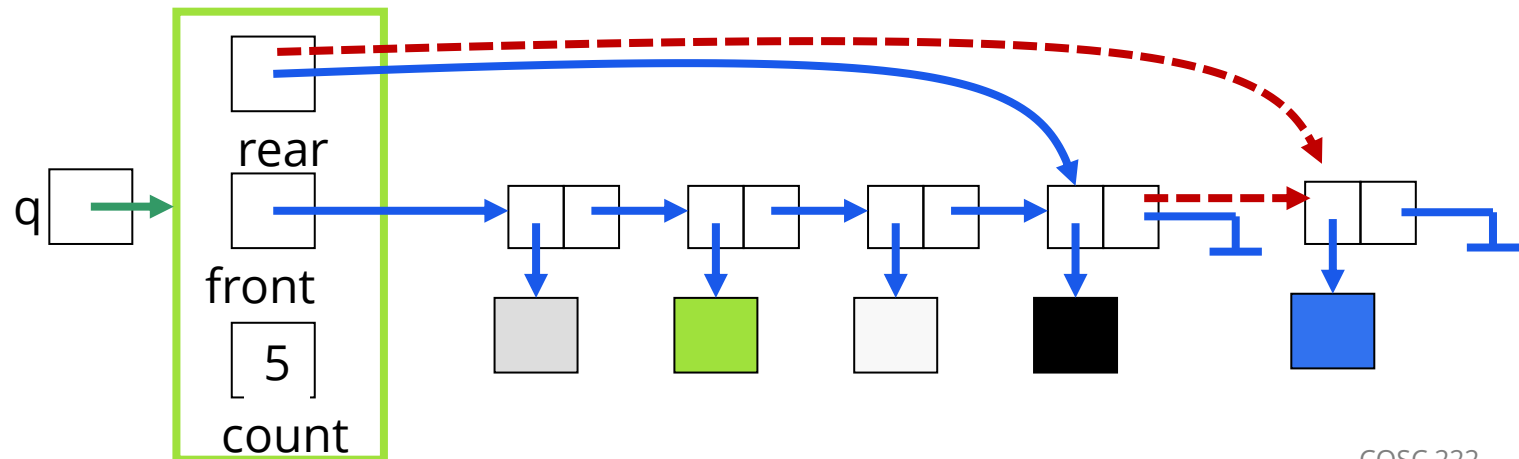
- The queue is represented as a linked list of nodes:
 - **front** is a reference to the head of the queue (beginning of the linked list)
 - **rear** is a reference to the tail of the queue (end of the linked list)
 - The integer **count** is the number of nodes in the queue

Creating a Queue

```
public class LinkedQueue<T> implements QueueADT<T> {  
    /**  
     * Attributes  
     */  
    private int count;  
    private QNode<T> front, rear;  
  
    /**  
     * Creates an empty queue.  
     */  
    public LinkedQueue() {  
        count = 0;  
        front = rear = null;  
    }  
}
```

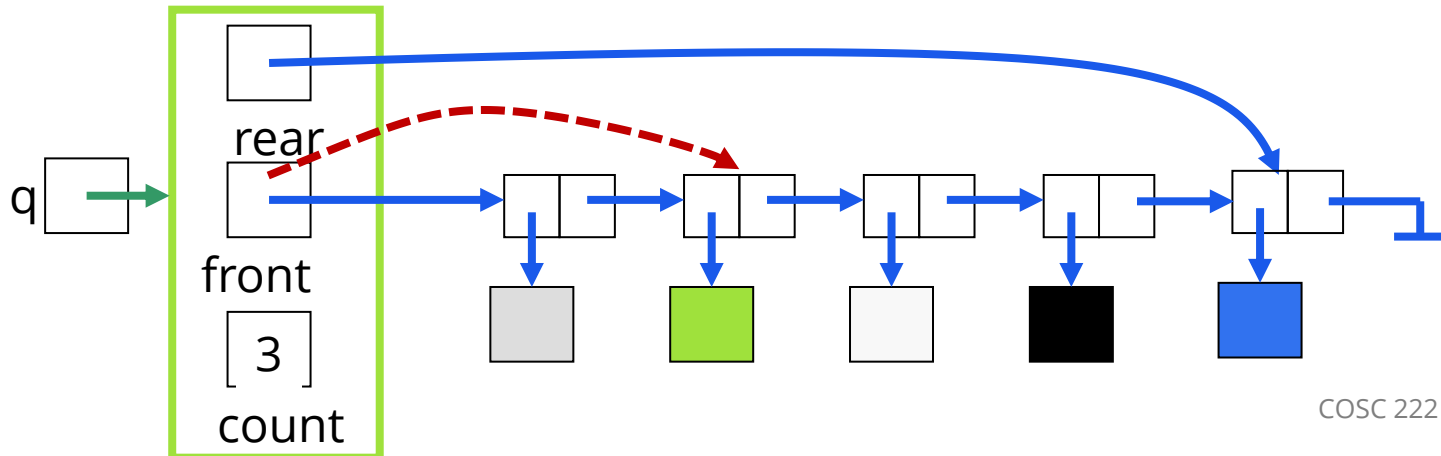
Add an Element

```
public void enqueue (T element) {  
    QNode<T> node = new QNode<T> (element, null);  
  
    if (isEmpty( ))  
        front = node;  
    else  
        rear.setLink(node);  
  
    rear = node;  
    count++;  
}
```



Removing an Element

```
public T dequeue ( ) {  
    if (isEmpty( )) {  
        System.out.println("Queue is empty.");  
        return null;  
    }  
    T result = front.getData( );  
    front = front.getLink();  
    count--;  
    if (isEmpty( ))  
        rear = null;  
    return result;  
}
```



Other methods

```
public boolean isEmpty()  
{  
    return (count == 0);  
}
```

```
public T first()  
{  
    if (isEmpty()){  
        System.out.println("Queue is empty.");  
        return null;  
    }  
  
    return front.getData();  
}
```

Other methods

```
public int size()  
{  
    return count;  
}
```

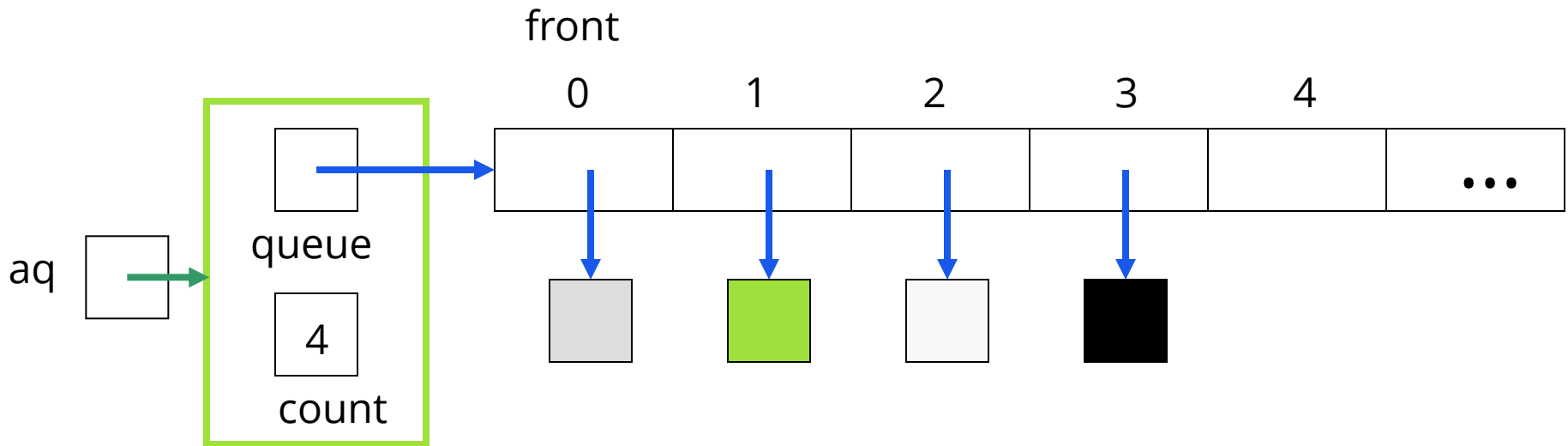
```
public String toString()  
{  
    String result = "";  
    QNode<T> current = front;  
    while (current != null)  
    {  
        result = result + (current.getData()).toString() + "\t";  
        current = current.getLink();  
    }  
    return result;  
}
```

Array Implementation of a Queue

- First Approach:
 - Use an array in which index 0 represents one end of the queue (the front)
 - Integer value count represents the number of elements in the array (so the element at the rear of the queue is in position $\text{count} - 1$)

An Array Implementation of a Queue

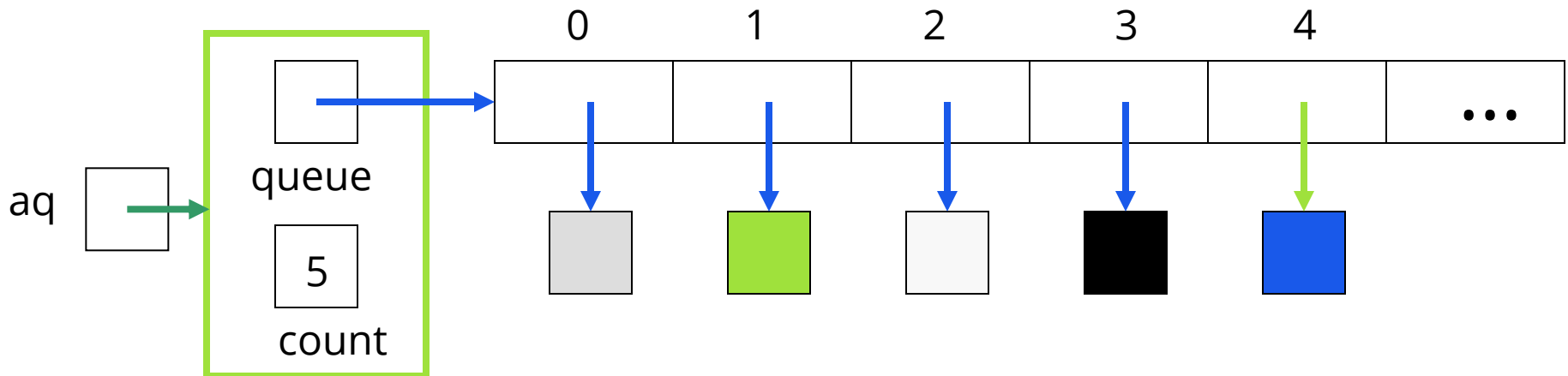
A queue `aq` containing four elements




Queue After Adding an Element

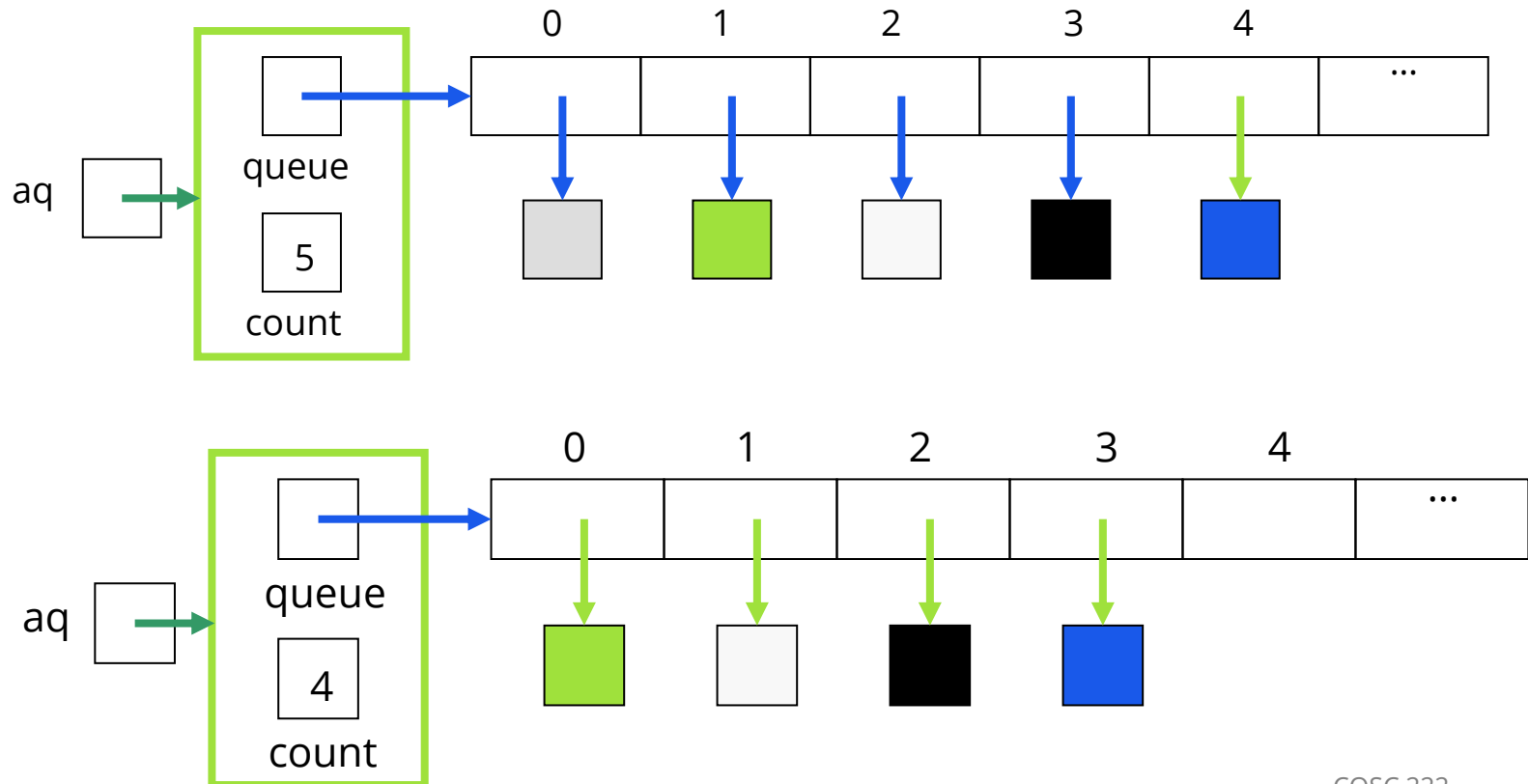


The element is added at the array location given by the value of count and then count is increased by 1.



Queue After Removing an Element

Element  is removed from array location 0, remaining elements are shifted forward one position in the array, and then count is decremented.



Add an Element

```
public void enqueue (T element)
{
    if (size() == queue.length)
        expandCapacity();

    queue[rear] = element;
    rear++;
}
```

```
private void expandCapacity()
{
    T[] larger = (T[])(new Object[queue.length*2]);

    for (int scan=0; scan < queue.length; scan++)
        larger[scan] = queue[scan];

    queue = larger;
}
```


Remove an Element

```
public T dequeue() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException ("Queue is empty");

    T result = queue[0];

    rear--;

    // shift the elements
    for (int scan=0; scan < rear; scan++)
        queue[scan] = queue[scan+1];

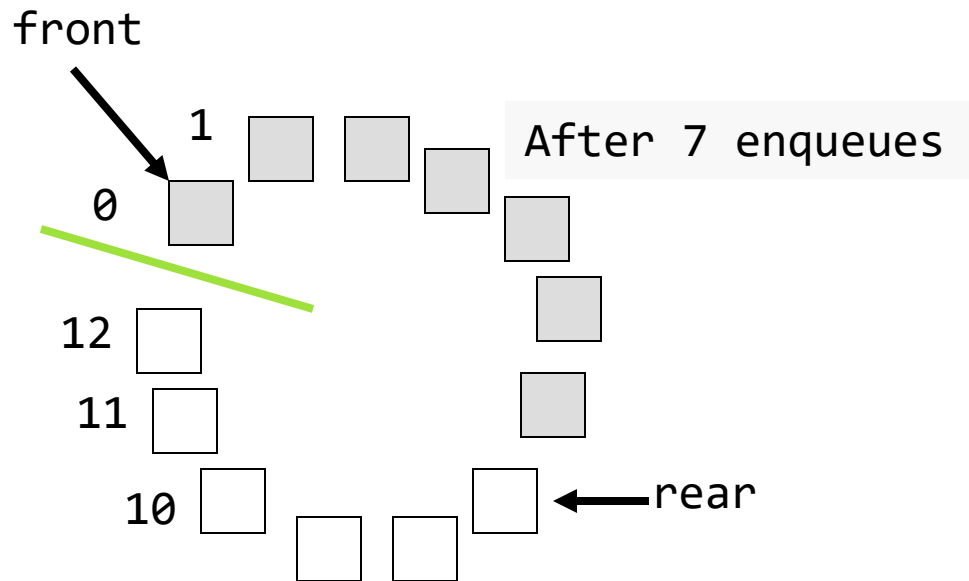
    queue[rear] = null;

    return result;
}
```

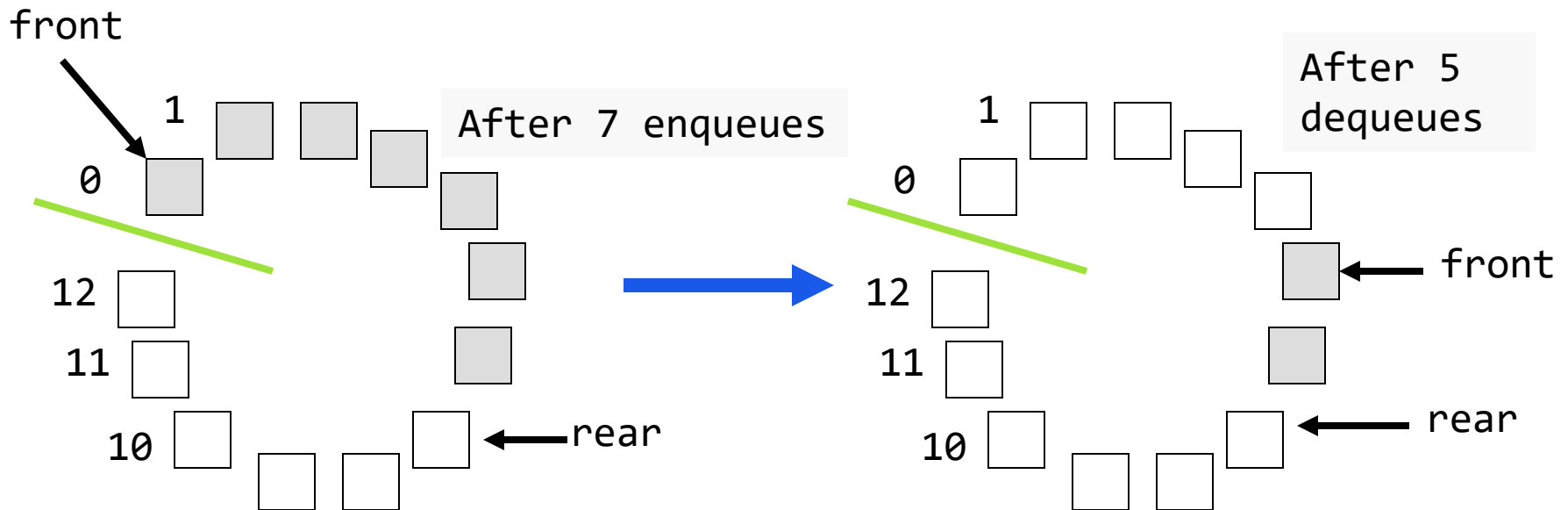
Second Approach: Queue as a Circular Array

- If we don't fix one end of the queue at index 0, we won't have to shift elements
- **Circular array** is an array that conceptually loops around on itself
- The last index is thought to “precede” index 0
- In an array whose last index is n , the location “**before**” index 0 is index n ; the location “**after**” index n is index 0
- Need to keep track of where the front as well as the **rear** of the queue are at any given time

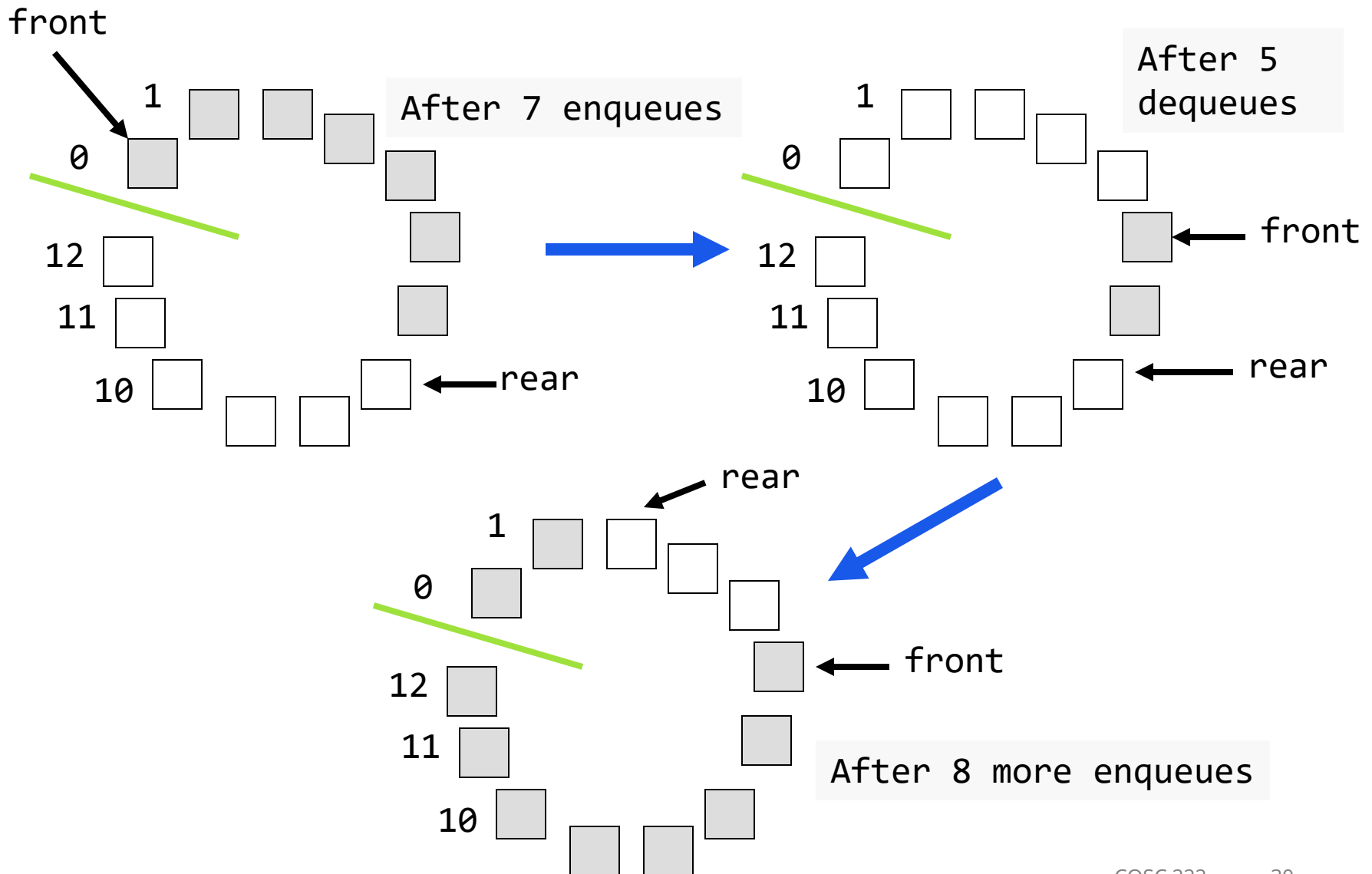
Conceptual Example of a Circular Queue



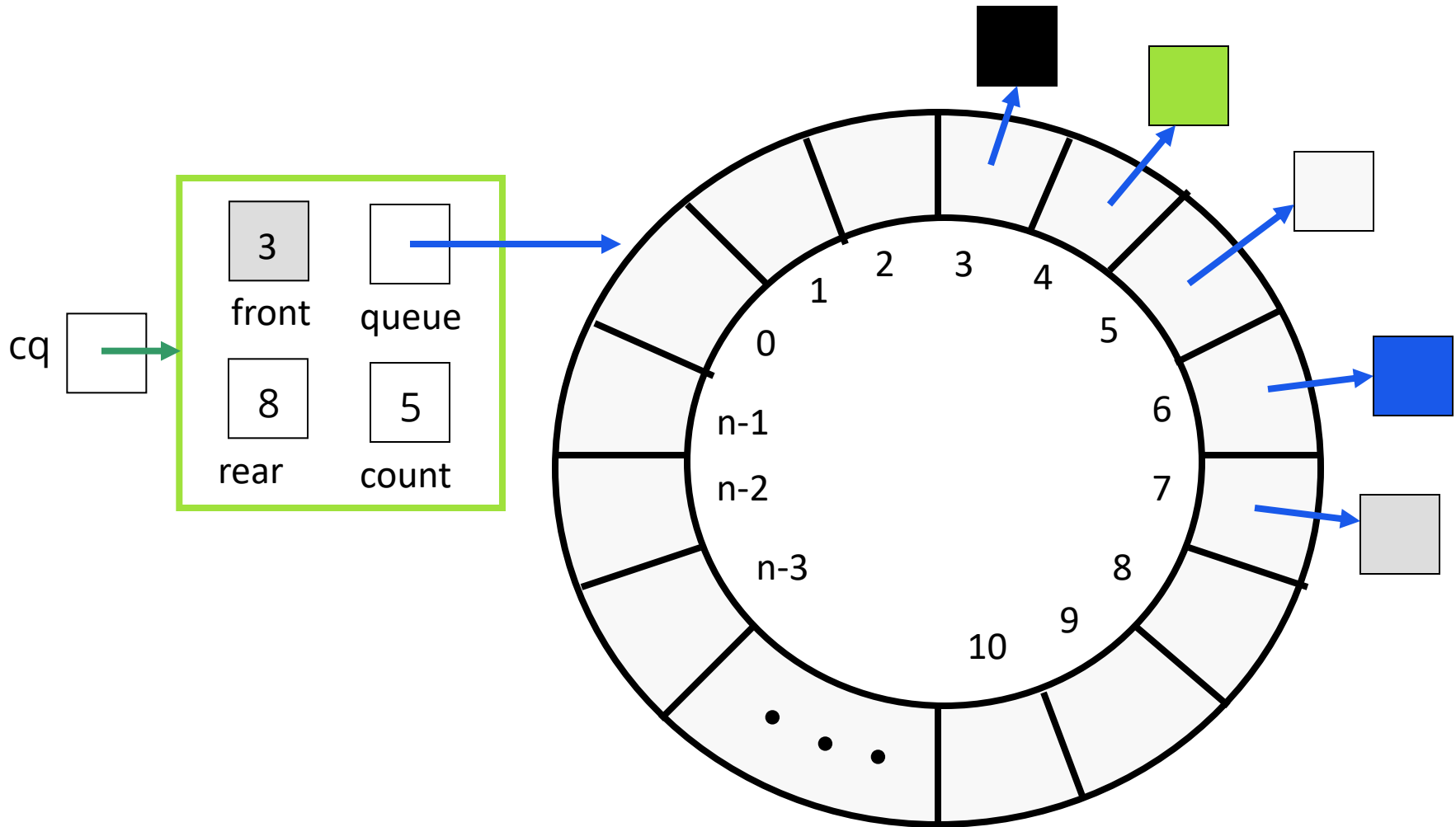
Conceptual Example of a Circular Queue



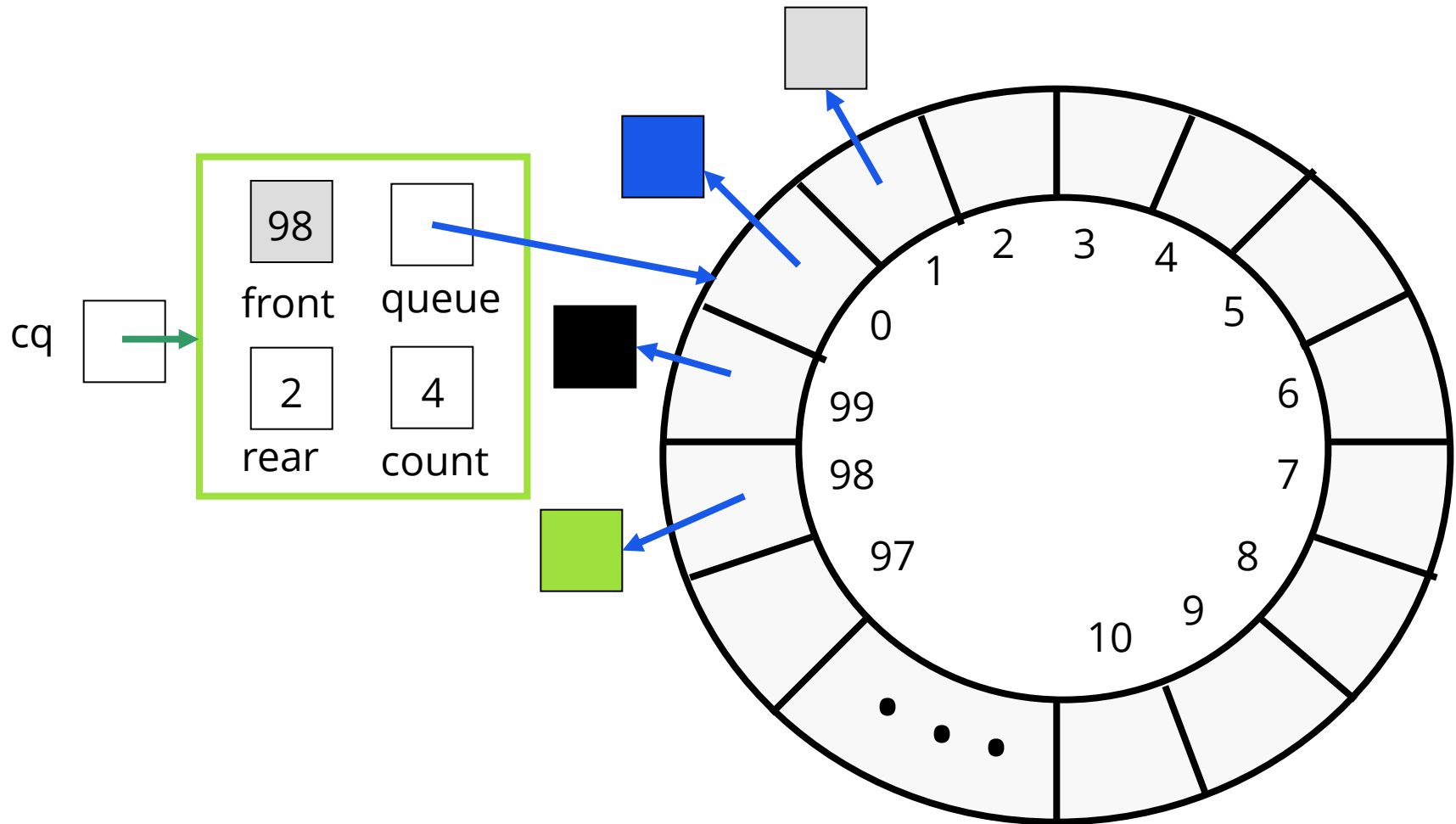
Conceptual Example of a Circular Queue



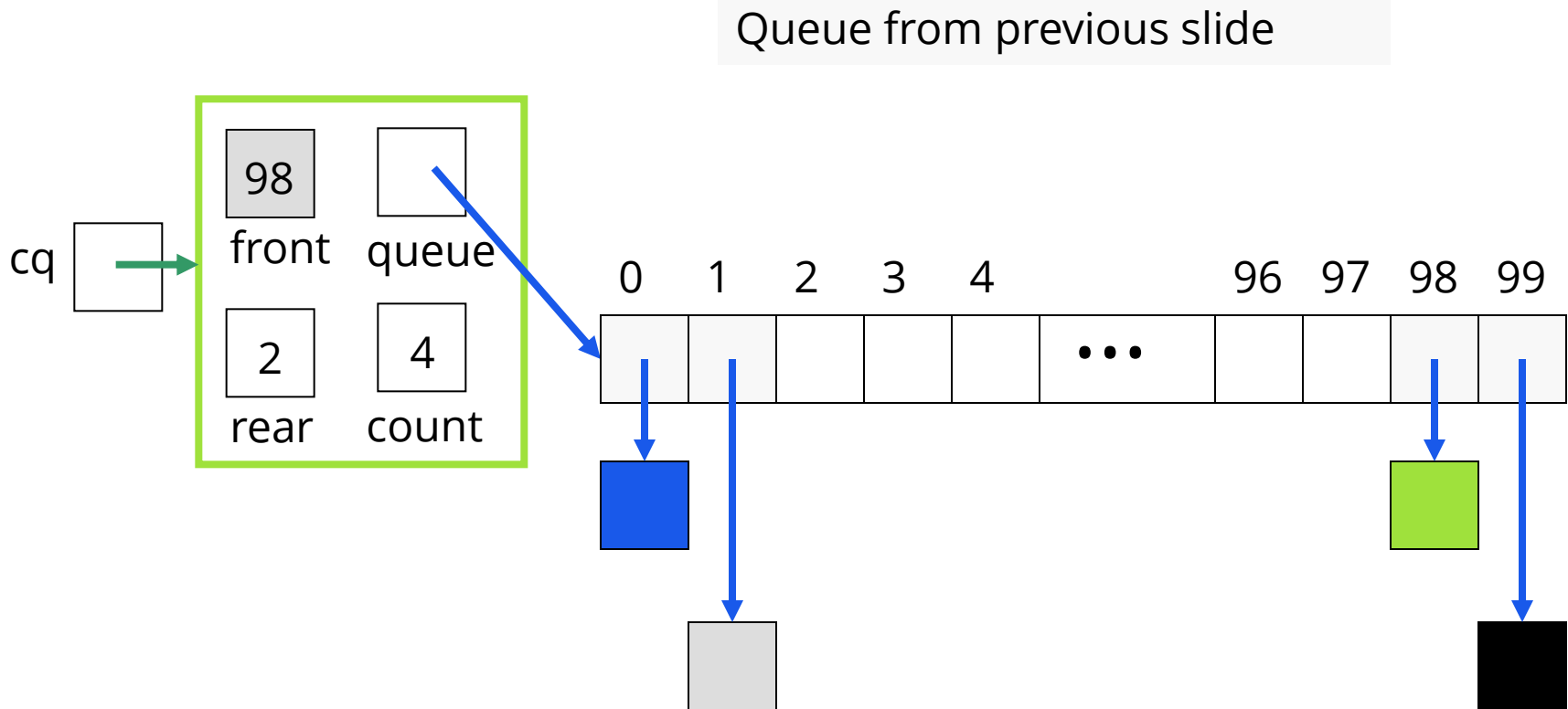
Circular Array Implementation of a Queue



A Queue Straddling the End of a Circular Array



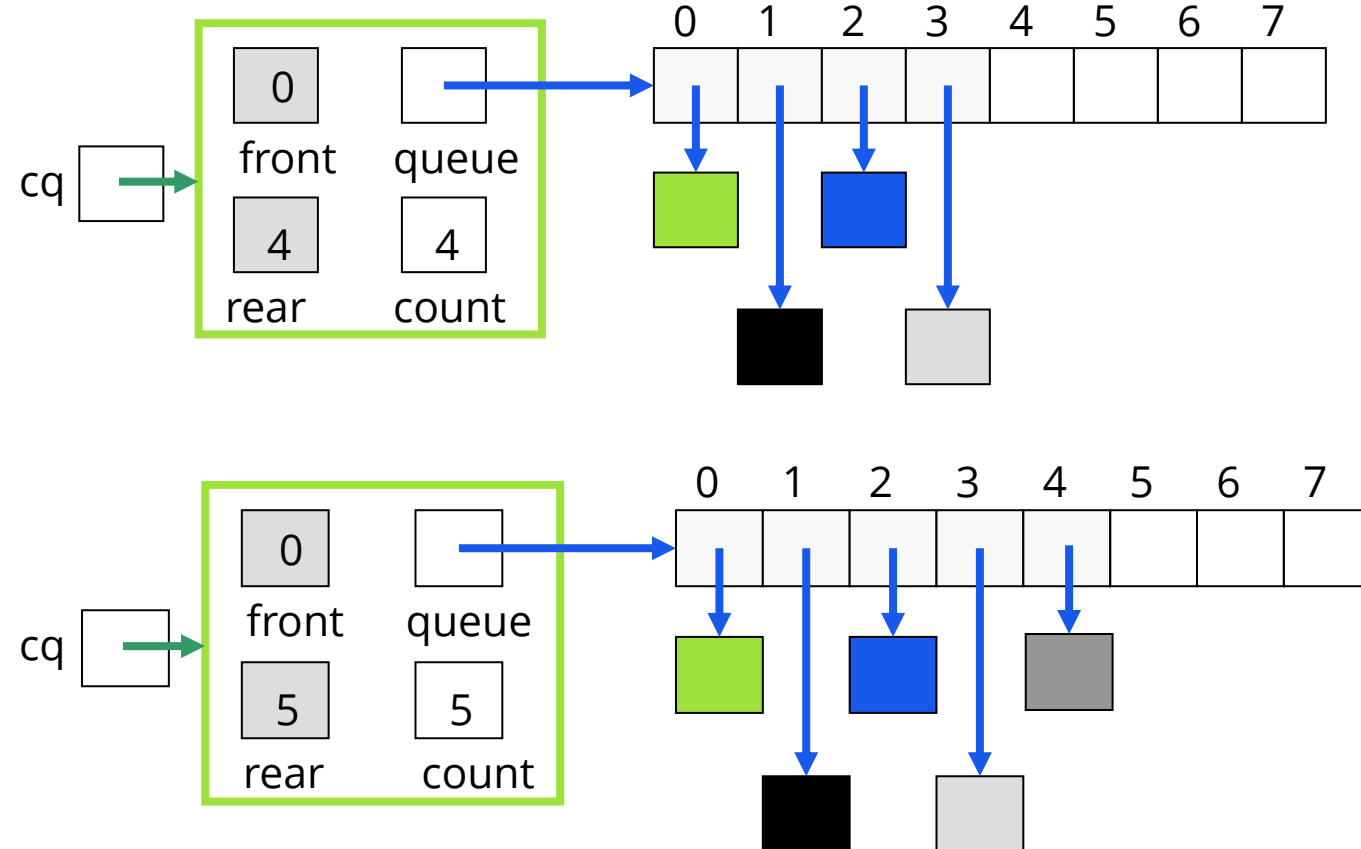
Circular Queue Drawn Linearly



Circular Array Implementation

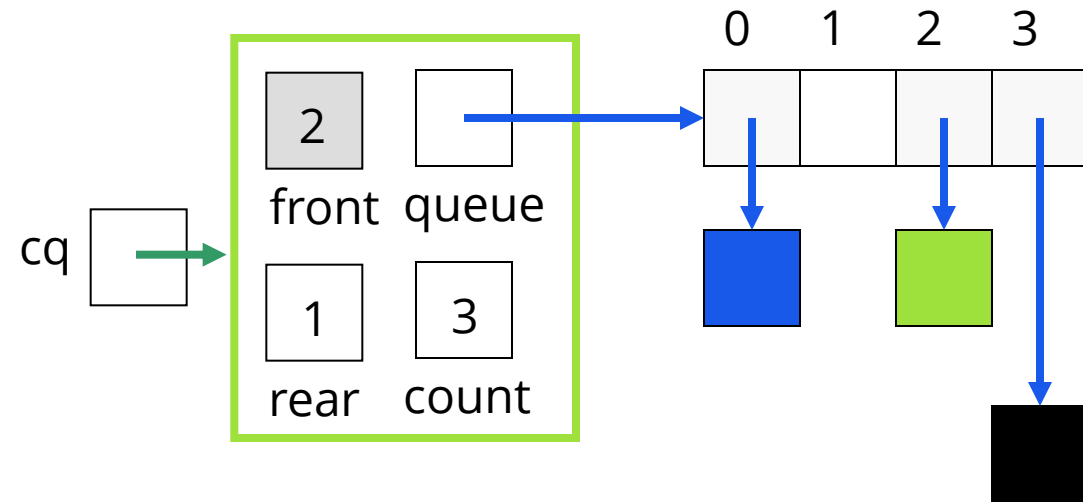
- When an element is enqueued, the value of **rear** is incremented
- But it must take into account the need to loop back to index 0:

$\text{rear} = (\text{rear} + 1) \% \text{queue.length};$

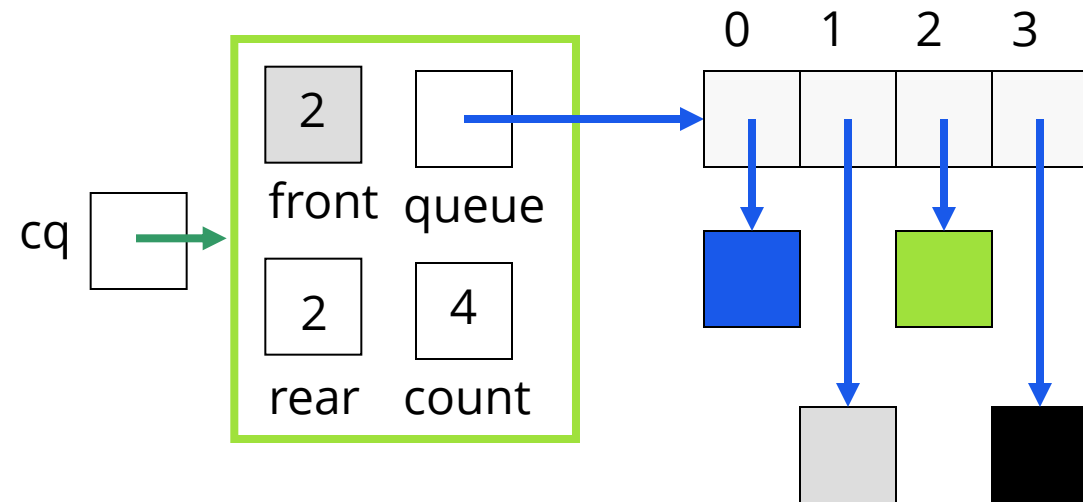


$$\text{rear} = (4 + 1) \% 8 = 5$$

Example: array of length 4 What happens?

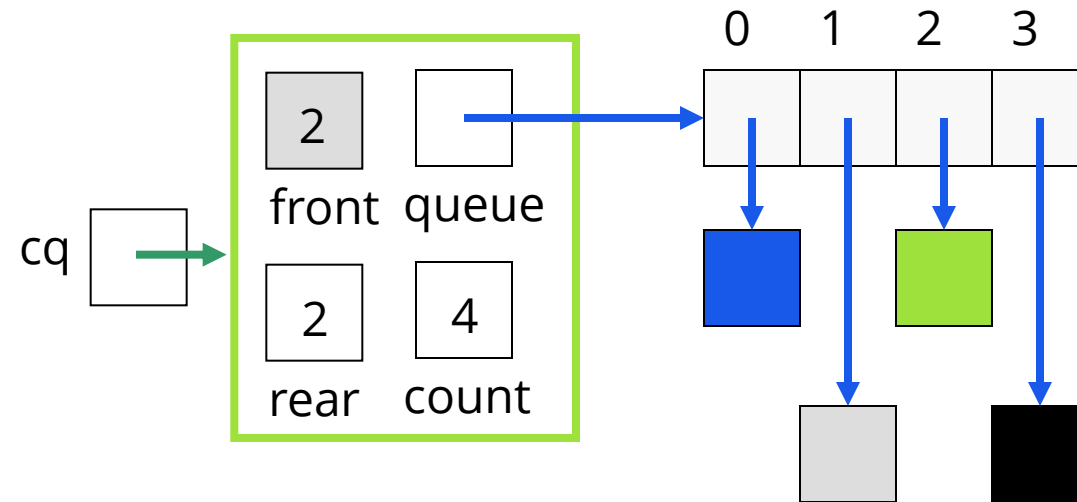


Suppose we try to add one more item to a queue implemented by an array of length 4

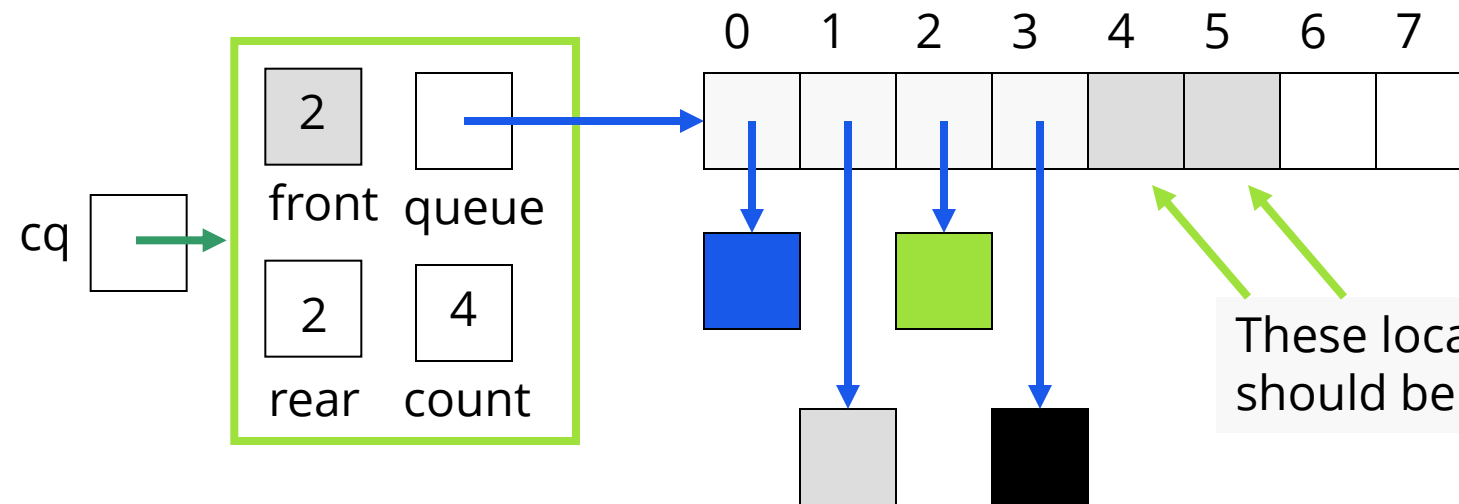


The queue is now full.

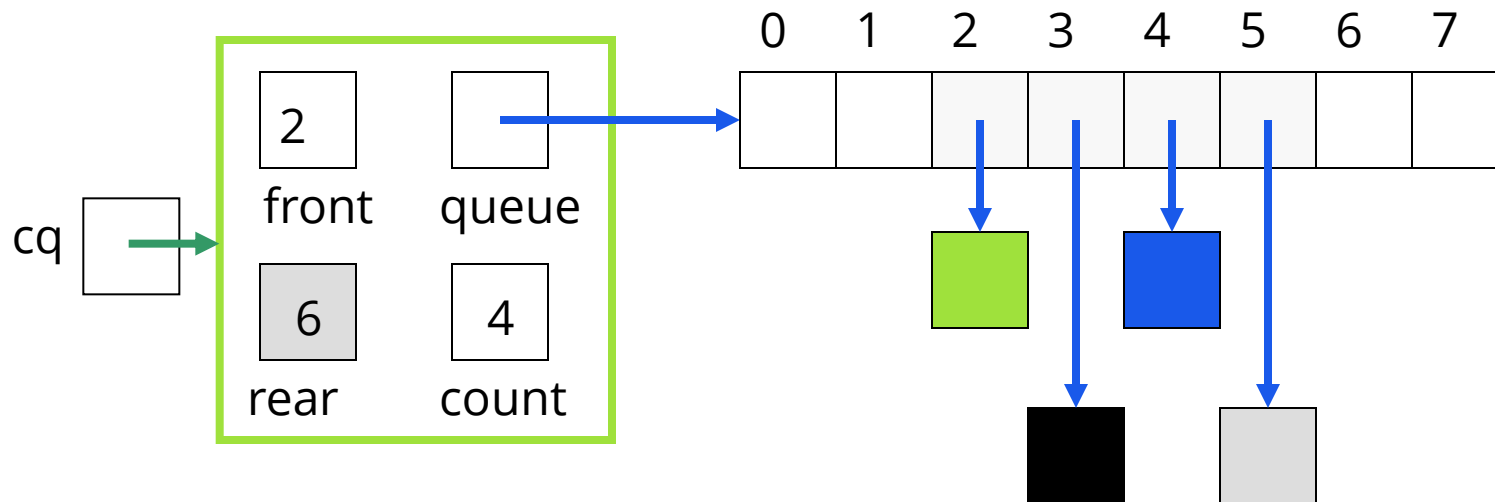
Add another item! Need to expand capacity...



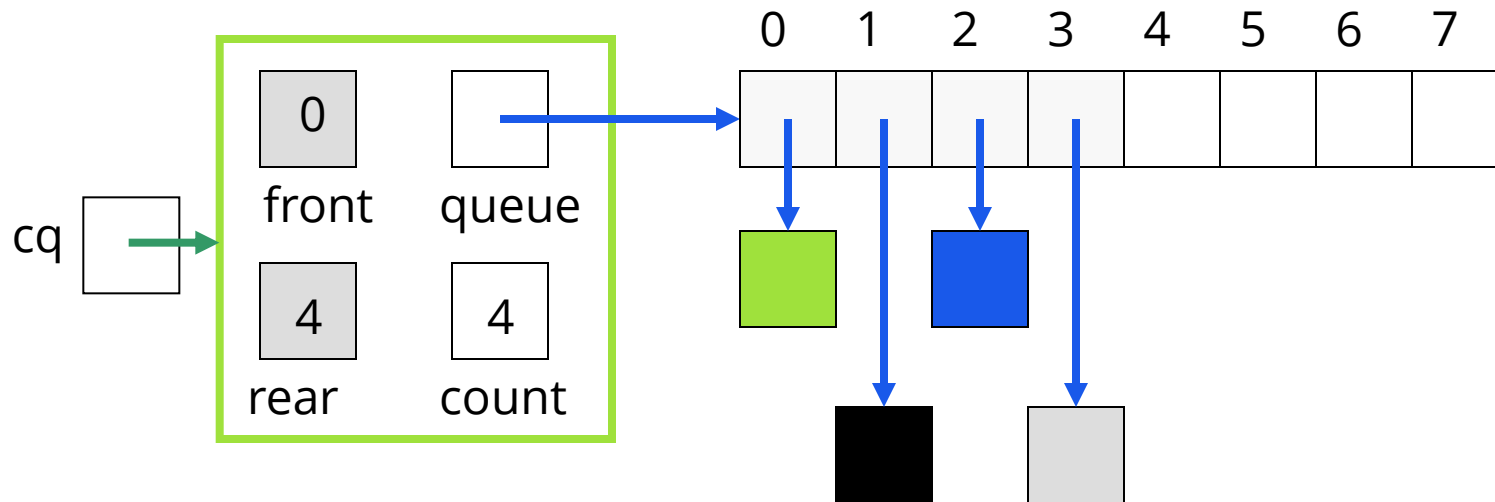
We can't just double the size of the array and copy values to the same positions as before: circular properties of the queue will be lost



We *could* build the new array, and copy the queue elements into contiguous locations beginning at location front:



Better: copy the queue elements in order to the *beginning* of the new array



Questions?