DATA STRUCTURES USING 'C'

Lecture-14

Data Structures

The Queue ADT

Lecture Objectives

- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be used to solve problems
- Examine various queue implementations
- Compare queue implementations

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
- Any waiting line is a queue:
  - The check-out line at a grocery store
  - The cars at a stop light
  - An assembly line

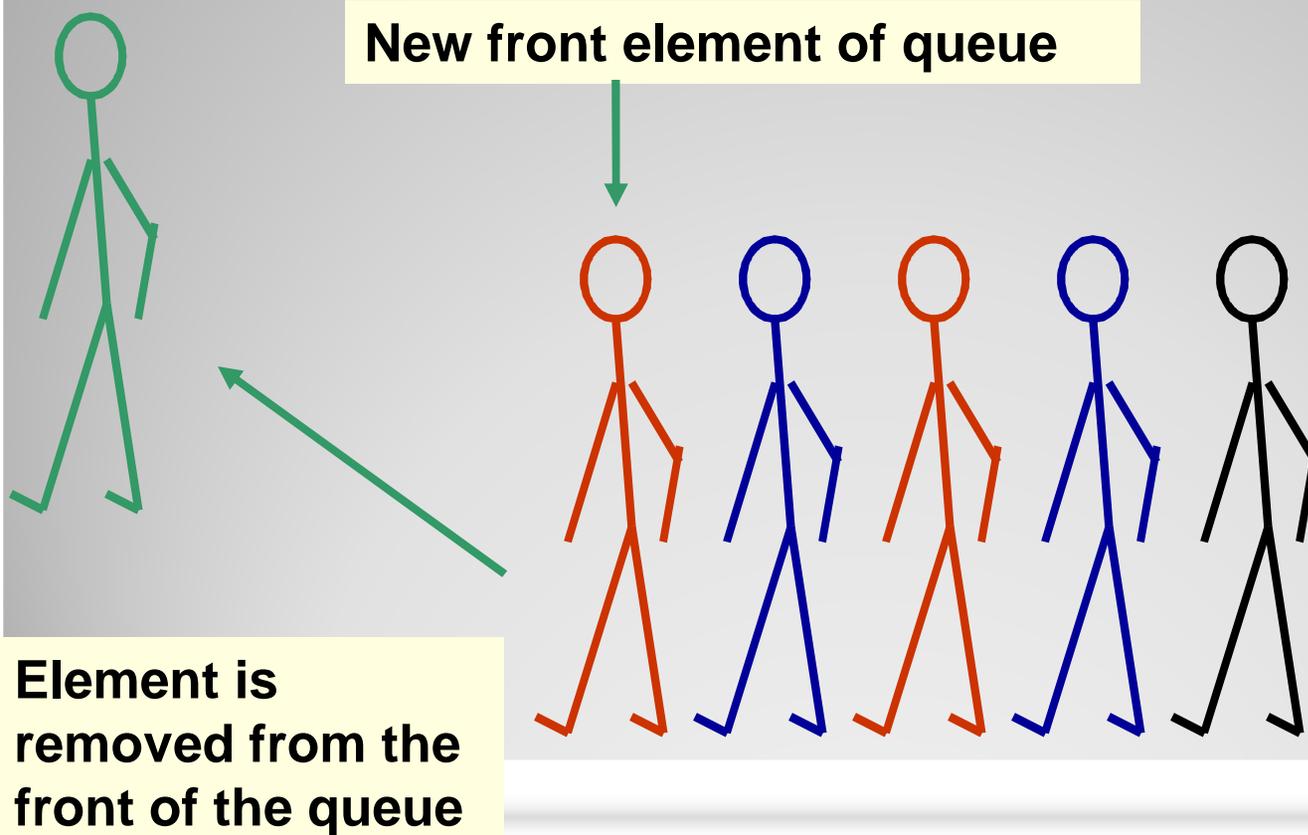# Conceptual View of a Queue

Adding an element

Front of queue
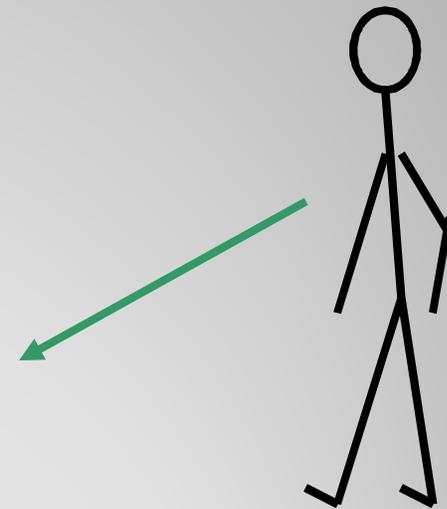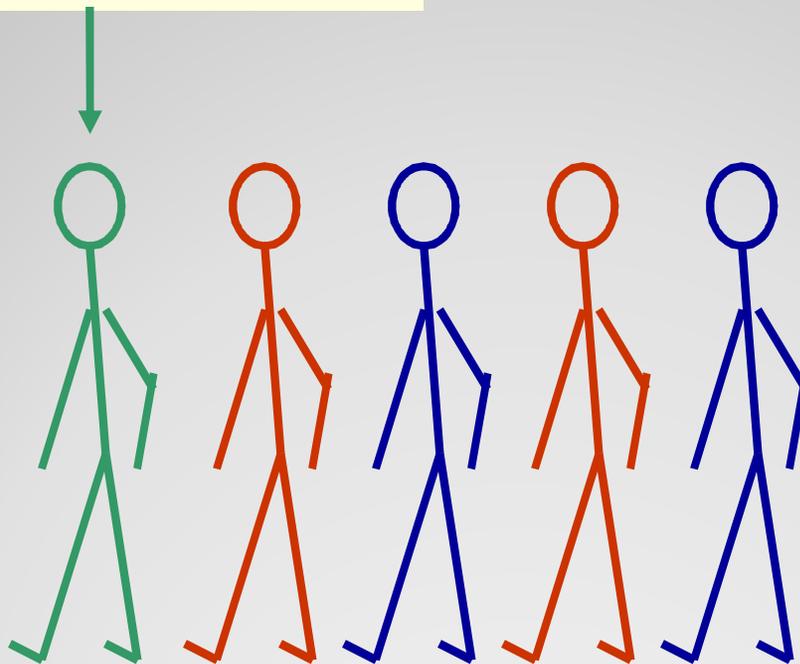
New element is added to the rear of the queue

Conceptual View of a Queue

Removing an element

New front element of queue

Element is removed from the front of the queue

Uses of Queues in Computing

- For any kind of problem involving FIFO data
- Printer queue (e.g. printer in MC 235)
- Keyboard input buffer
- GUI event queue (click on buttons, menu items)
- To encode messages (more on this later)

Uses of Queues in Computing

- In *simulation studies*, where the goal is to reduce waiting times:
  - Optimize the flow of traffic at a traffic light
  - Determine number of cashiers to have on duty at a grocery store at different times of day
  - Other examples?

# Queue Operations

- *enqueue* : add an element to the tail of a queue
- *dequeue* : remove an element from the head of a queue
- *first* : examine the element at the head of the queue ("peek")
- Other useful operations (e.g. is the queue empty)
- It is *not* legal to access the elements in the middle of the queue!

Operations on a Queue

Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear 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

The QueueADT interface in UML

<<interface>> QueueADT
dequeue( ) enqueue( ) first( ) isEmpty( ) size( ) toString( )
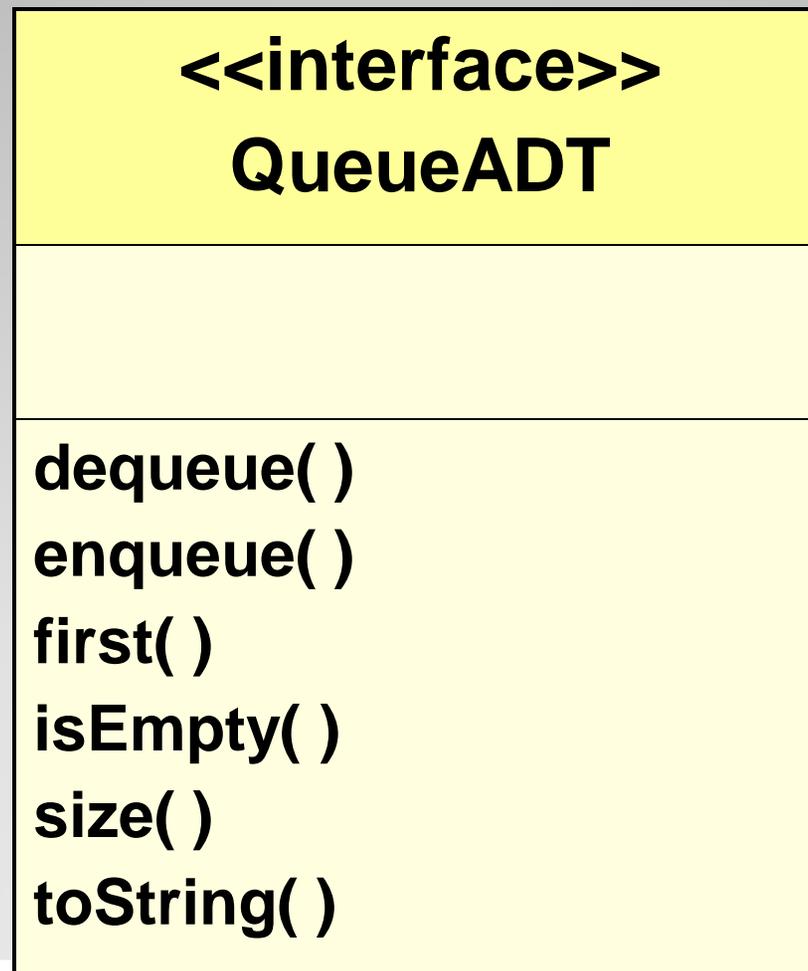
Interface to a Queue in Java

```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( );
}
```

Using Queues: Coded Messages

- A *Caesar cipher* is a *substitution code* that encodes a message by shifting each letter in a message by a constant amount **k**

  - If **k** is **5**, **a** becomes **f**, **b** becomes **g**, etc.

    - *Example*: *n qtaj ofaf*

  - Used by Julius Caesar to encode military messages for his generals (around 50 BC)

  - This code is fairly easy to decode

Using Queues: Coded Messages

- *Modern version*: ROT13
  - Each letter is shifted by 13
  - "used in online forums as a means of hiding spoilers, punchlines, puzzle solutions, and offensive materials from the casual glance" (*Wikipedia*)
  - What is the advantage of shifting 13?
    - Hint: ROT13(A) = N, ROT13(N) = A

Using Queues: Coded Messages

- *An improvement*: change how much a letter is shifted depending on where the letter is in the message

- A *repeating key* is a sequence of integers that determine how much each character is shifted

  - Example: consider the repeating key

    3 1 7 4 2 5

  - The first character in the message is shifted by 3, the next by 1, the next by 7, and so on

  - When the key is exhausted, start over at the beginning of the key

An Encoded Message Using a Repeated Key

Encoded message

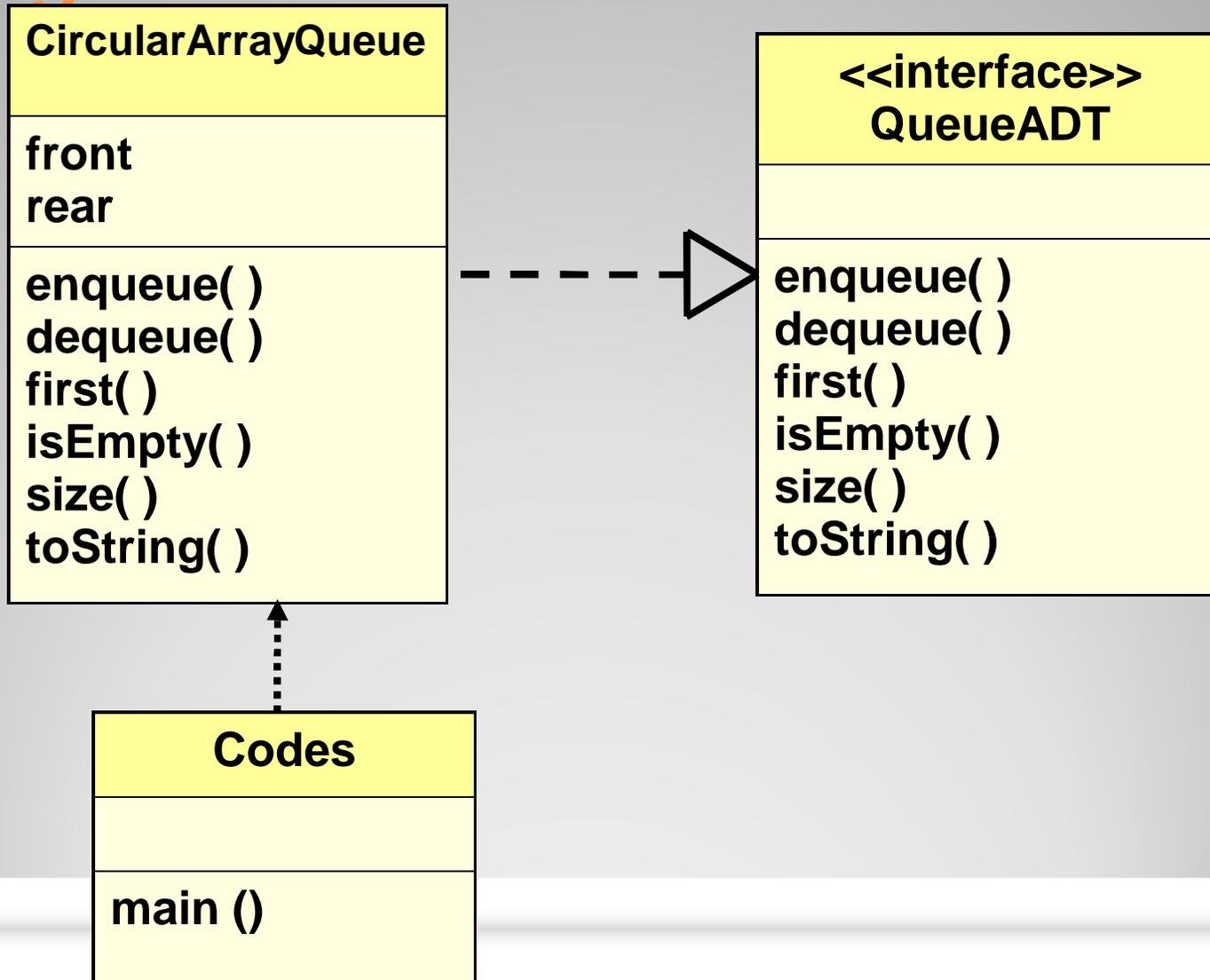| n	o	v	a	n	g	j	h	l		m	u		u	r	x	l	v
3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

Key

Decoded message

Using Queues: Coded Messages

- We can use a queue to store the values of the key
    - *dequeue* a key value when needed
    - *enqueue* it back onto the end of the queue

- So, the queue represents the constantly cycling values in the key

Using Queues: Coded Messages

- See *Codes.java*
  - Note that there are *two* copies of the key, stored in two separate queues
    - The encoder has one copy
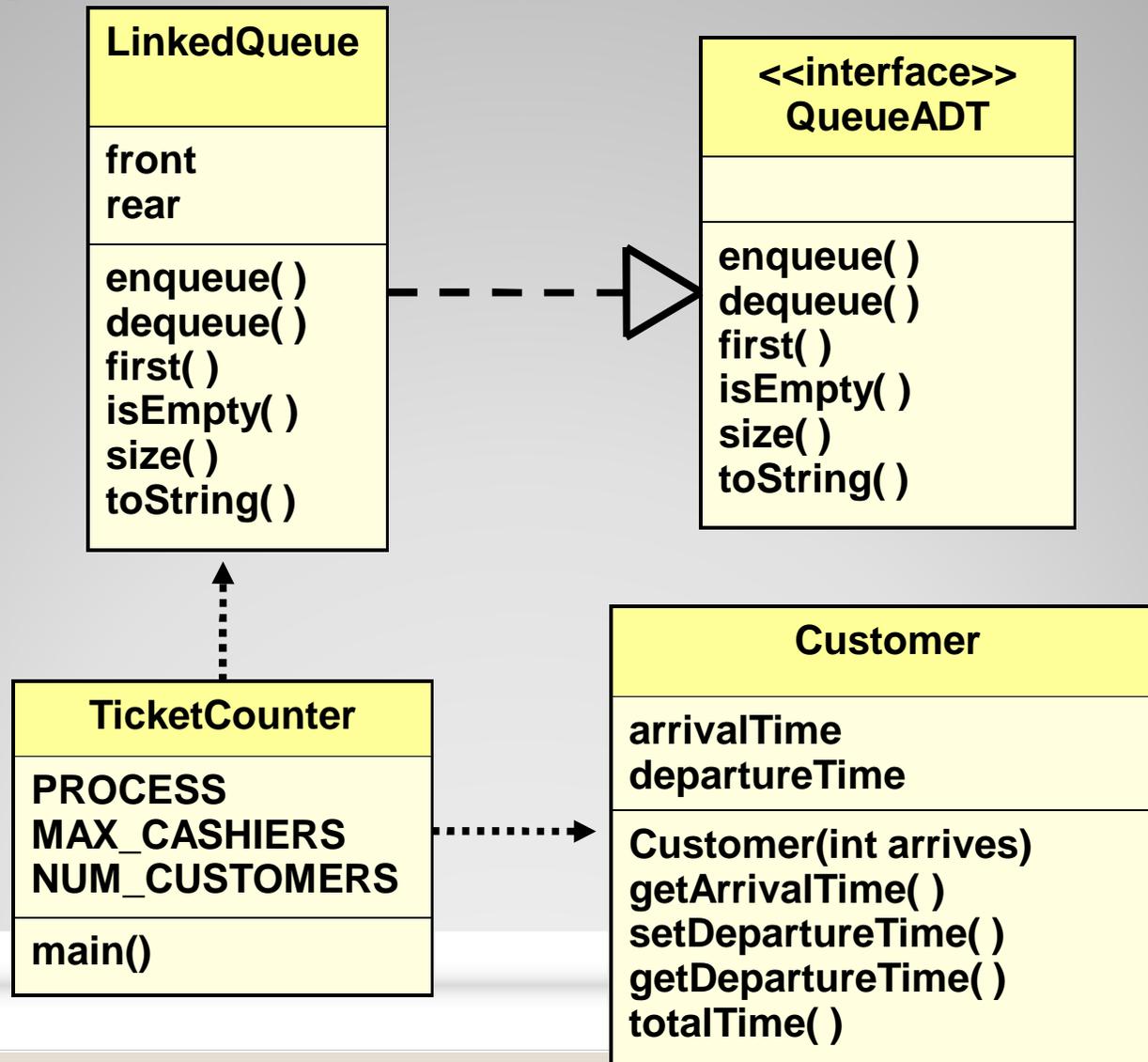   - The decoder has a separate copy

    - Why?

UML Description of Codes Program

CircularArrayQueue

front
rear

enqueue( )
dequeue( )
first( )
isEmpty( )
size( )
toString( )

<<interface>>
QueueADT

enqueue( )
dequeue( )
first( )
isEmpty( )
size( )
toString( )

Codes

main ()

Using Queues: Ticket Counter Simulation

- Simulate the waiting line at a movie theatre:
  - Determine how many cashiers are needed to keep the customer wait time under 7 minutes
- *Assume*:
  - Customers arrive on average every 15 seconds
  - Processing a request takes two minutes once a customer reaches a cashier
- See *Customer.java*, *TicketCounter.java*

UML Description of TicketCounter Program

LinkedQueue

front
rear

enqueue( )
dequeue( )
first( )
isEmpty( )
size( )
toString( )

<<interface>>
QueueADT

enqueue( )
dequeue( )
first( )
isEmpty( )
size( )
toString( )

TicketCounter

PROCESS
MAX_CASHIERS
NUM_CUSTOMERS

main()

Customer

arrivalTime
departureTime

Customer(int arrives)
getArrivalTime( )
setDepartureTime( )
getDepartureTime( )
totalTime( )

Results of Ticket Counter Simulation

Number of Cashiers

Average time (in seconds)

1	2	3	4	5	6	7	8	9	10
5317	2325	1332	840	547	355	219	120	120	120

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

- Internally, the queue is 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

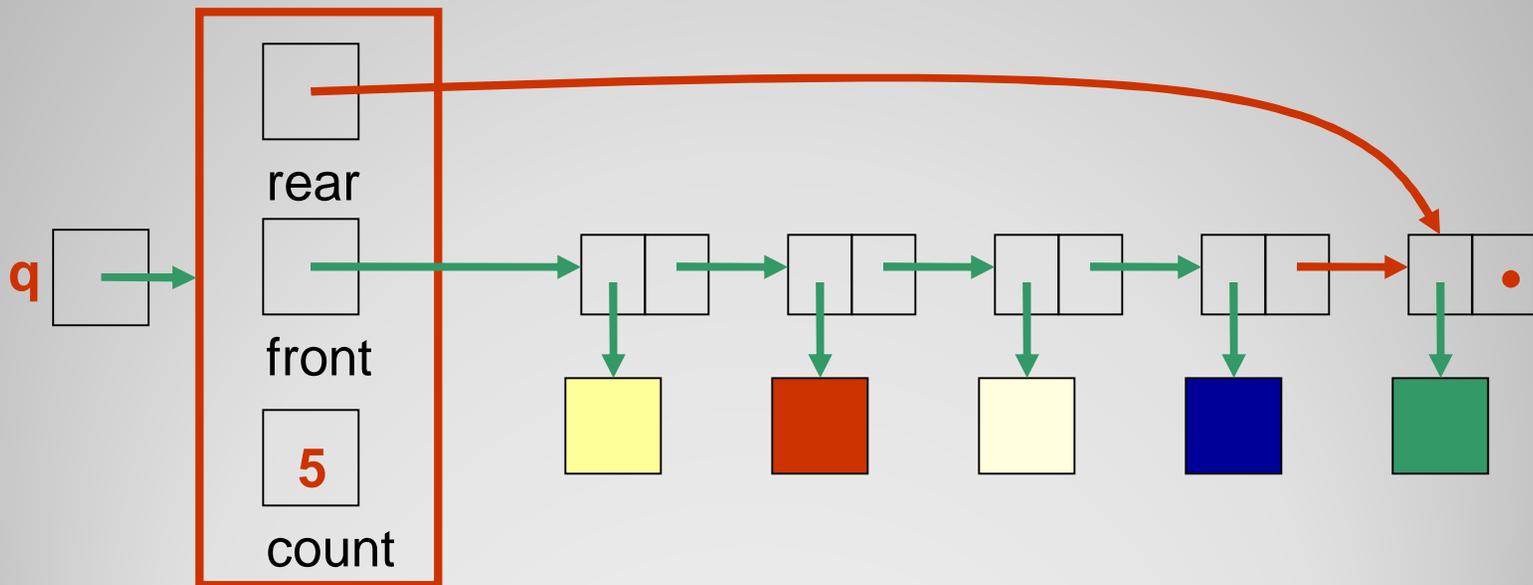# Linked Implementation of a Queue

A queue **q** containing four elements
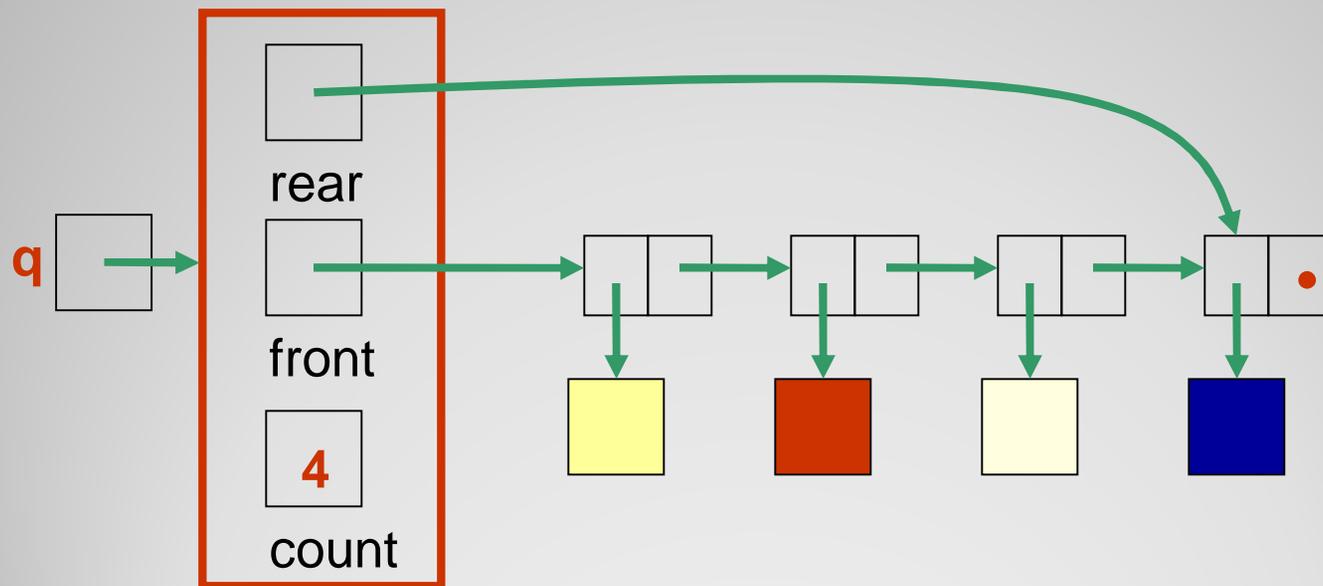
Discussion

- What if the queue is empty?

- What if there is only 1 element?

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.
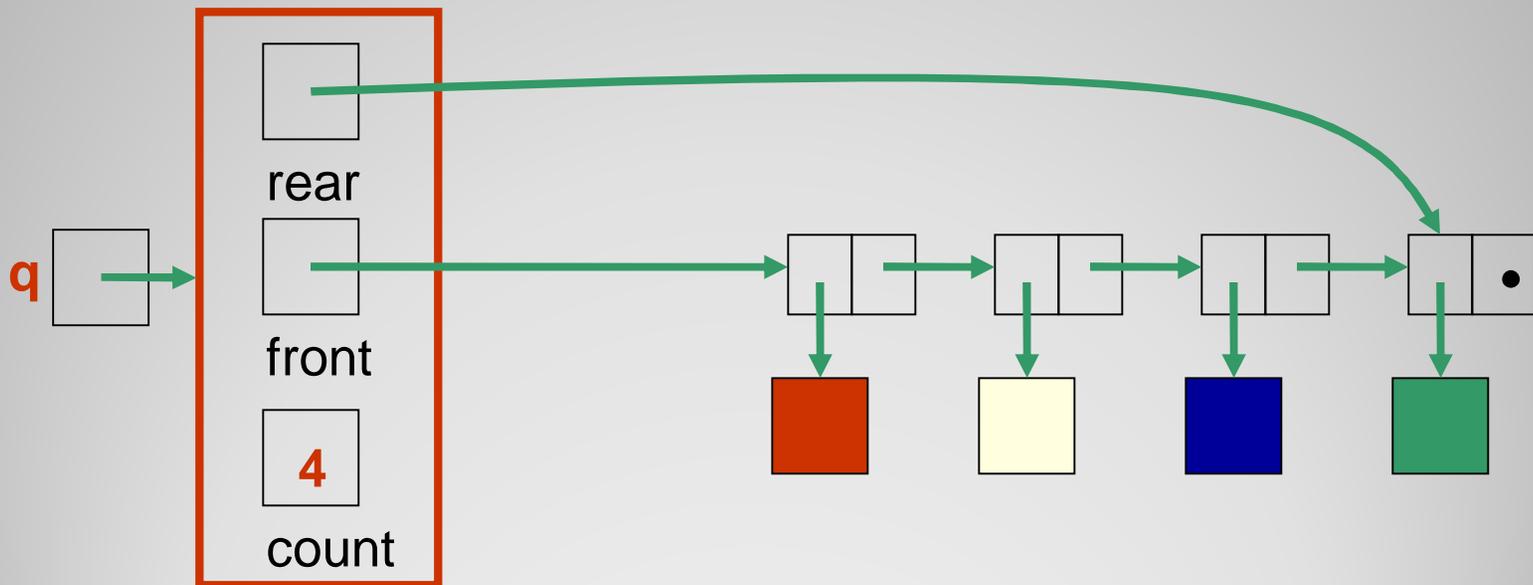
Java Implementation

- The queue is represented as a linked list of nodes:
  - We will again use the **LinearNode** class
  - **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

```java
public class LinkedQueue<T> implements QueueADT<T>
{
  /**
   * Attributes
   */
  private int count;
  private LinearNode<T> front, rear;


  /**
   * Creates an empty queue.
   */
  public LinkedQueue()
  {
    count = 0;
    front = rear = null;
  }
```

The LinkedQueue class

```java
//--------------------------------------------------------------------
//  Adds the specified element to the rear of the queue.
//--------------------------------------------------------------------
public void enqueue (T element)
{
   LinearNode<T> node = new LinearNode<T> (element);

   if (isEmpty( ))
      front = node;
   else
      rear.setNext (node);

   rear = node;
   count++;
}
```

The enqueue( ) operation

```
//--------------------------------------------------------------------
//  Removes the element at the front of the queue and returns a
//  reference to it. Throws an EmptyCollectionException if the
//  queue is empty.
//--------------------------------------------------------------------
public T dequeue ( ) throws EmptyCollectionException
{
   if (isEmpty( ))
     throw new EmptyCollectionException ("queue");
   T result = front.getElement( );
   front = front.getNext( );
   count--;
   if (isEmpty( ))
     rear = null;
   return result;
}
```

The dequeue( ) operation

Array Implementation of a Queue

- *First Approach*:
  - Use an array in which index 0 represents one end of the queue (the *front*)
  - Integer value **rear** represents the next open slot in the array (and also the number of elements currently in the queue)
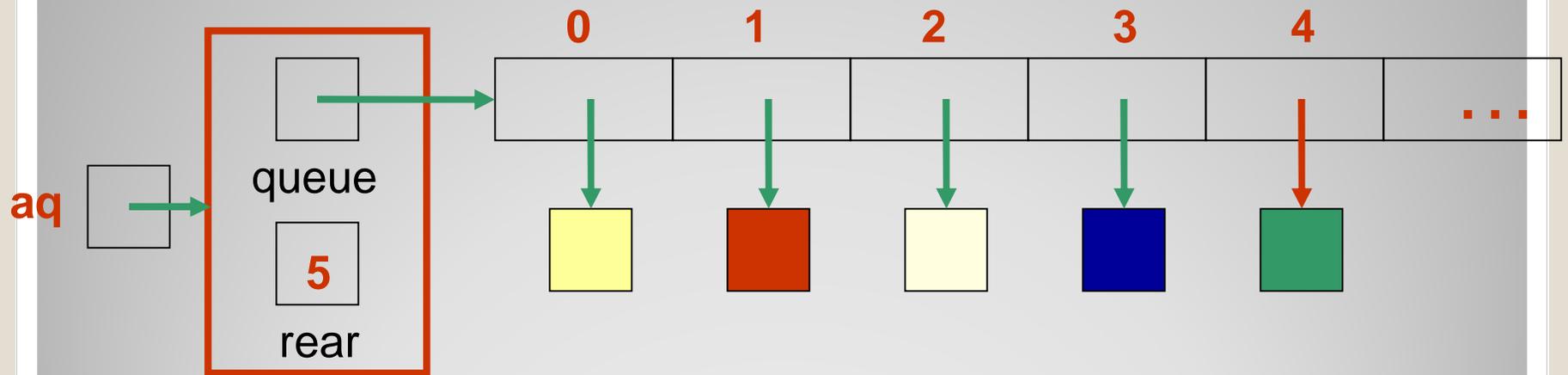- *Discussion*: What is the challenge with this approach?

An Array Implementation of a Queue

A queue **aq** containing four elements
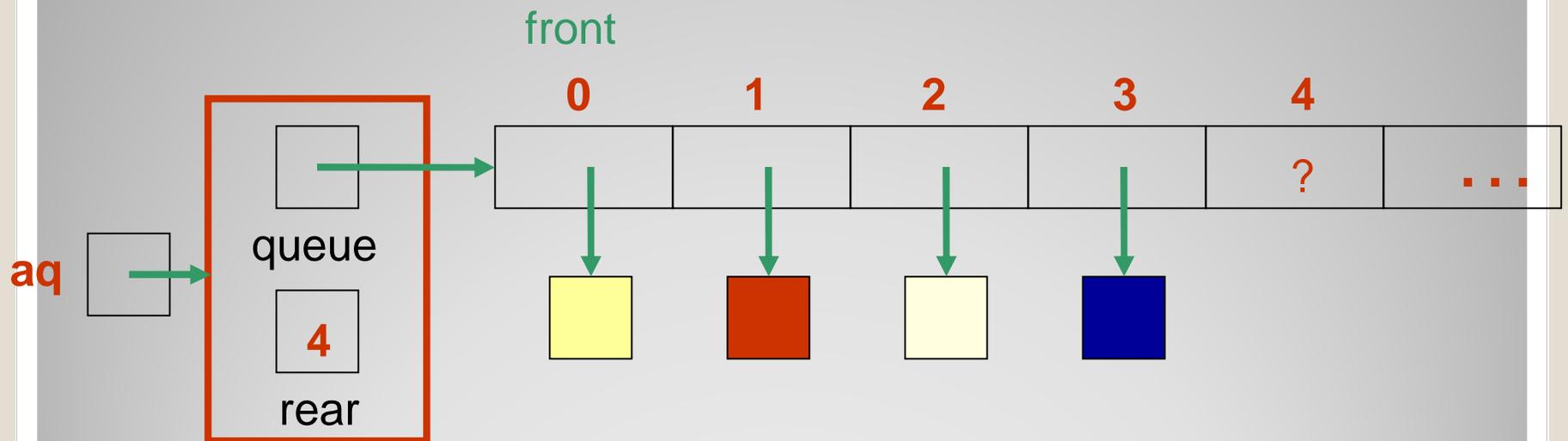
Queue After Adding an Element

Element is added at the array location given by the (old) value of rear, and then rear is incremented.

aq

queue

5

rear

0   1   2   3   4

. . .

Queue After Removing an Element

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

aq

queue

4

rear

0   1   2   3   4
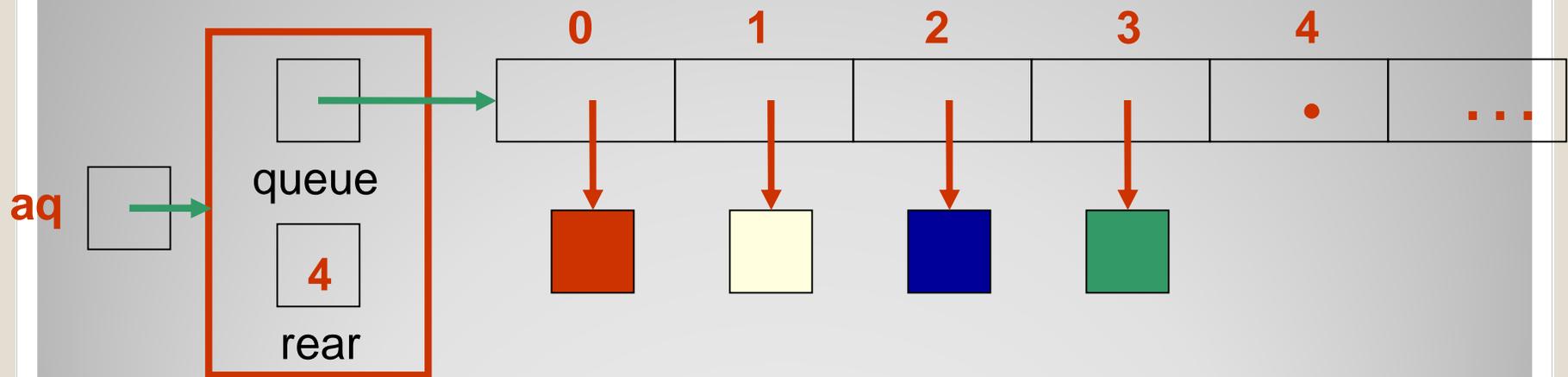
Java Array Implementation

- See *ArrayQueue.java*

```java
public class ArrayQueue<T> implements QueueADT<T>
{
   private final int DEFAULT_CAPACITY = 100;
   private int rear;
   private T[] queue;

public ArrayQueue()
   {
      rear = 0;
      queue = (T[])(new Object[DEFAULT_CAPACITY]);
   }
public ArrayQueue (int initialCapacity)
   {
      rear = 0;
      queue = (T[])(new Object[initialCapacity]);
   }
```

The ArrayQueue class

```
//--------------------------------------------------------------
//  Adds the specified element to the rear of the queue,
//  expanding the capacity of the queue array if
//  necessary.
//--------------------------------------------------------------
public void enqueue (T element)
{
   if (size() == queue.length)
     expandCapacity( );

   queue[rear] = element;
   rear++;
}
```
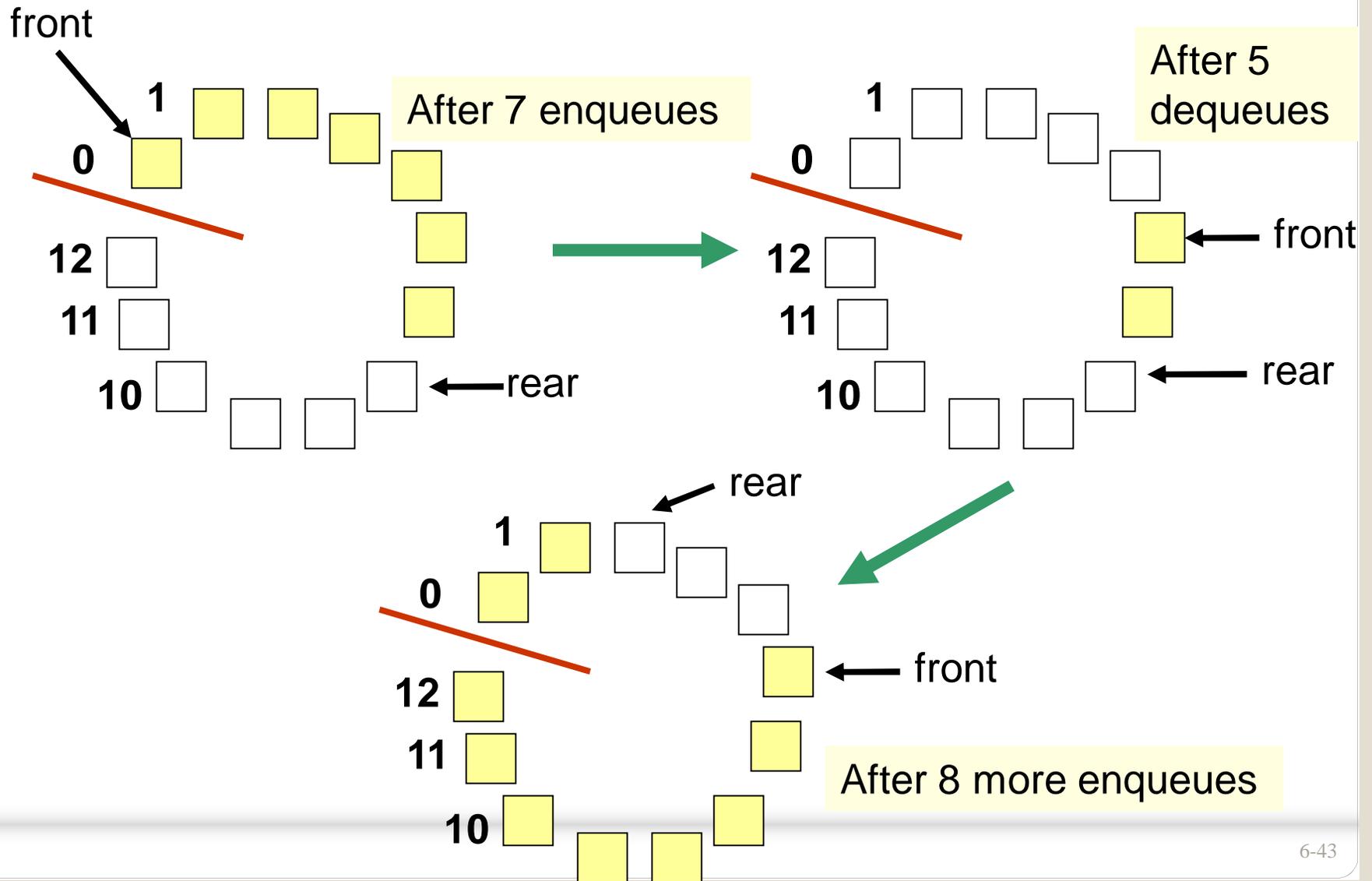
The enqueue( ) operation

```java
//------------------------------------------------------------------
//  Removes the element at the front of the queue and returns
//  a reference to it. Throws anEmptyCollectionException if the
//  queue is empty.
//------------------------------------------------------------------
public T dequeue ( ) throws EmptyCollectionException
{
   if (isEmpty( ))
      throw new EmptyCollectionException ("queue");
   T result = queue[0];
   rear--;
   // shift the elements
   for (int i = 0; i < rear; i++)
      queue[i] = queue[i+1];
   queue[rear] = null;
   return result;
}
```

The dequeue( ) operation

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**
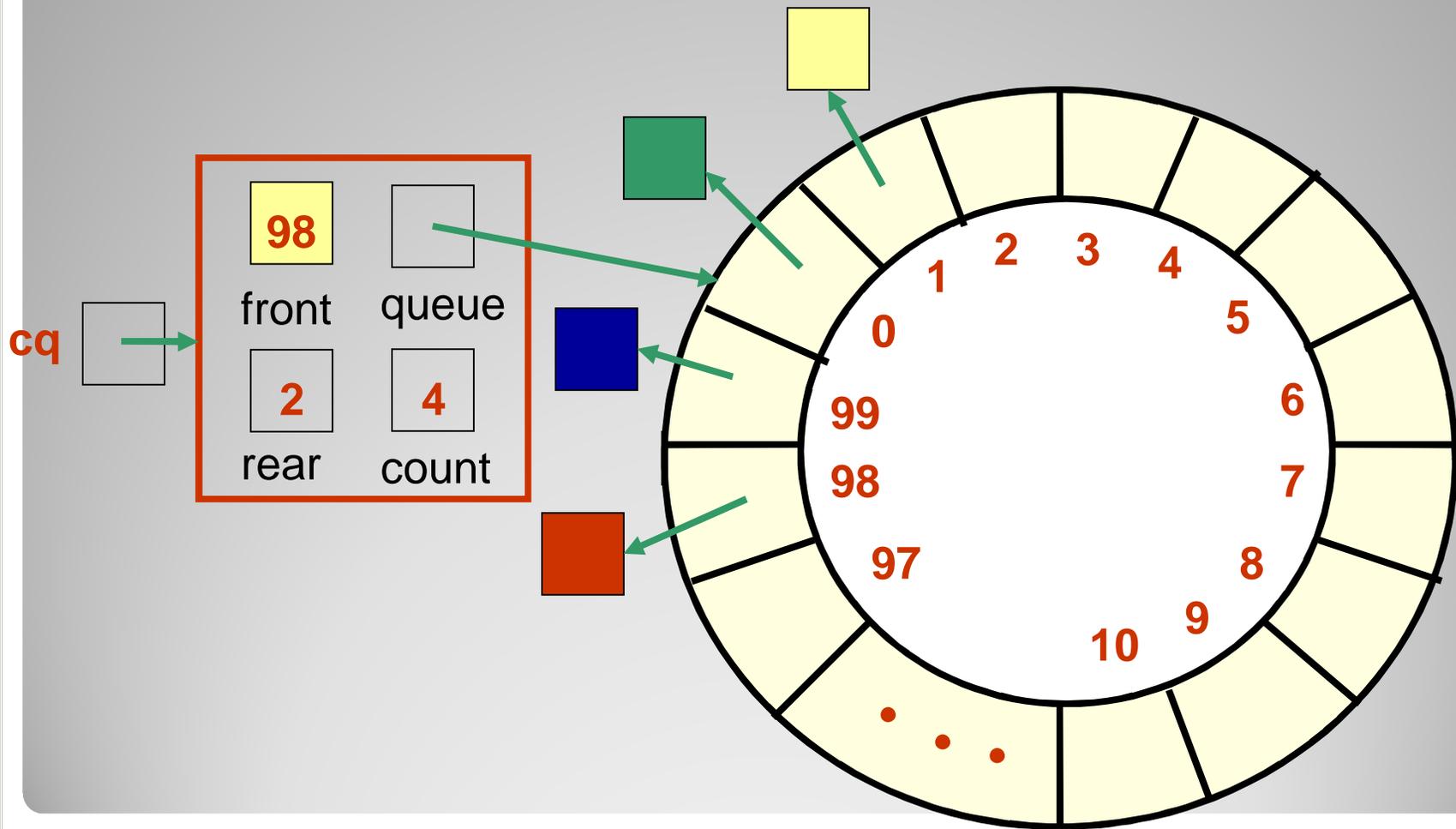- Need to keep track of where the *front* as well as the *rear* of the queue are at any given time

Conceptual Example of a Circular Queue

front

1

0

12

11

10

After 7 enqueues

rear

1

0

12

11

10

After 5 dequeues

front

rear

rear

1

0

12

11

10

front

After 8 more enqueues
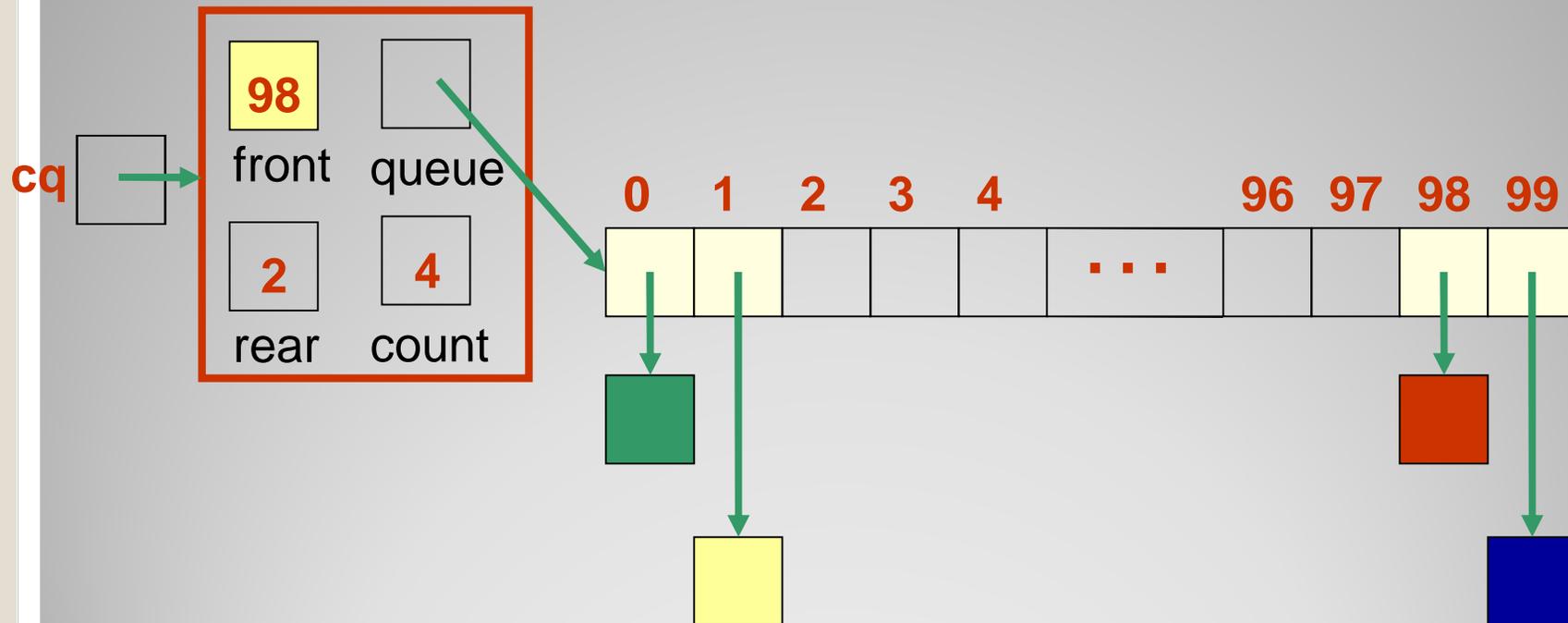
Circular Array Implementation of a Queue
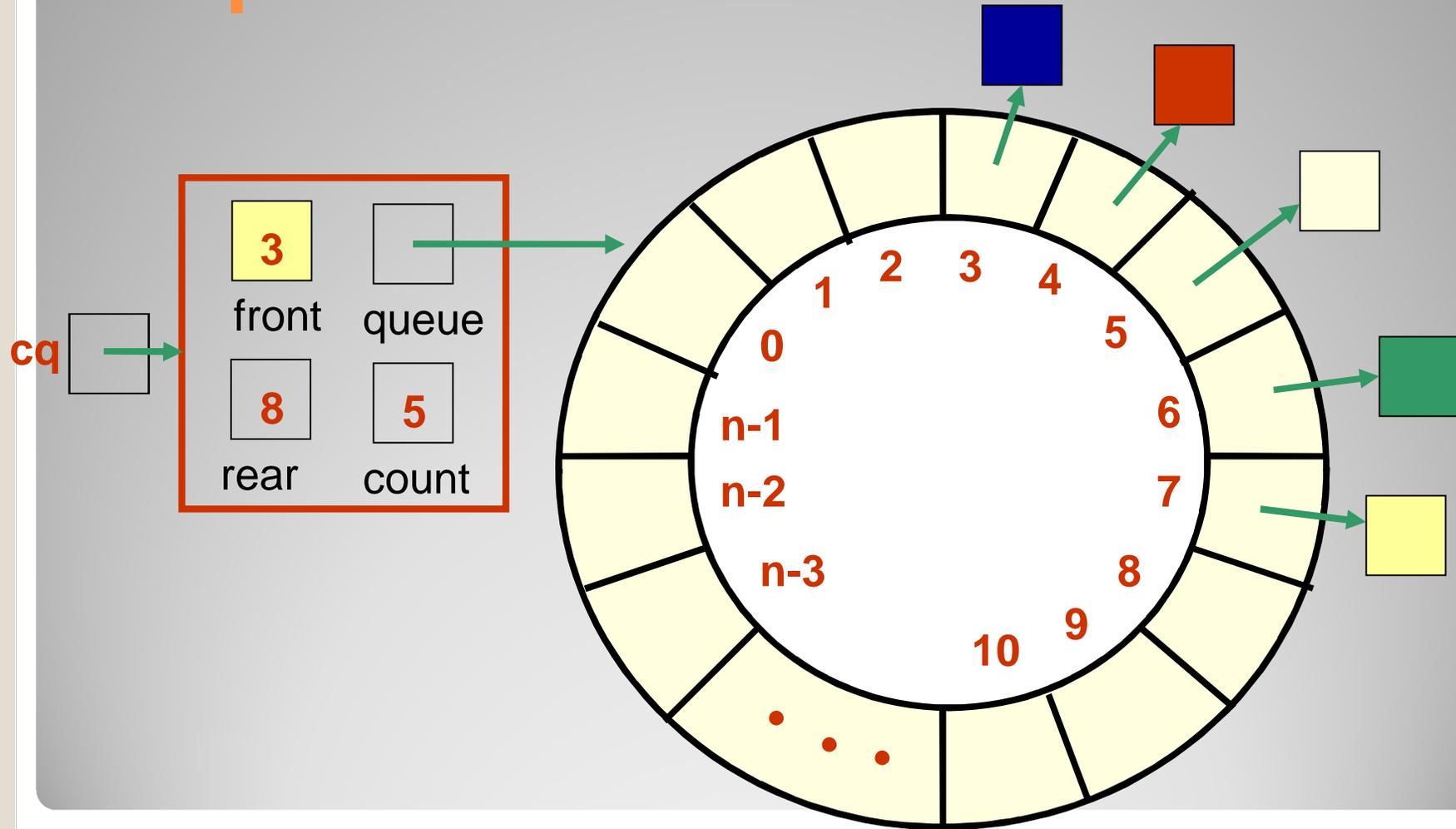
A Queue Straddling the End of a Circular Array

Circular Queue Drawn Linearly

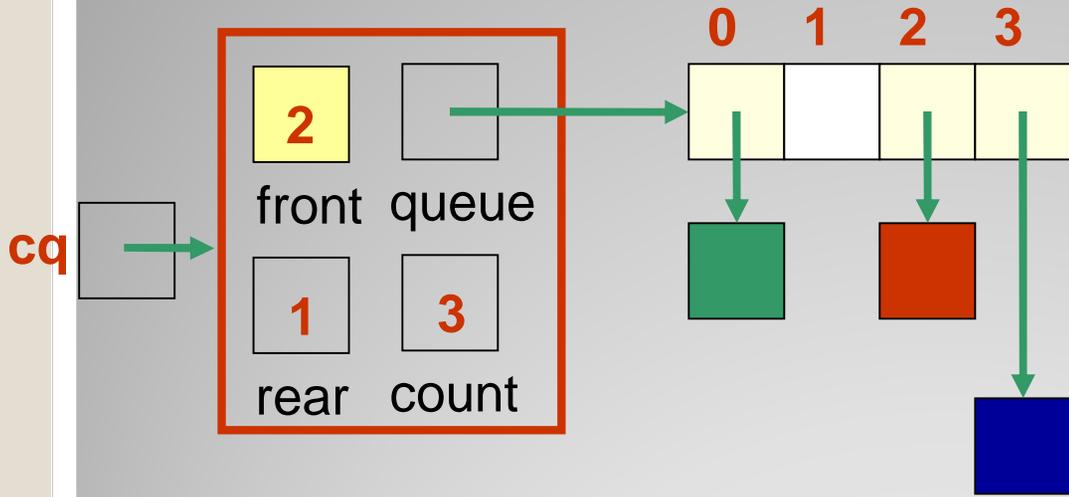**Queue from previous slide**

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:
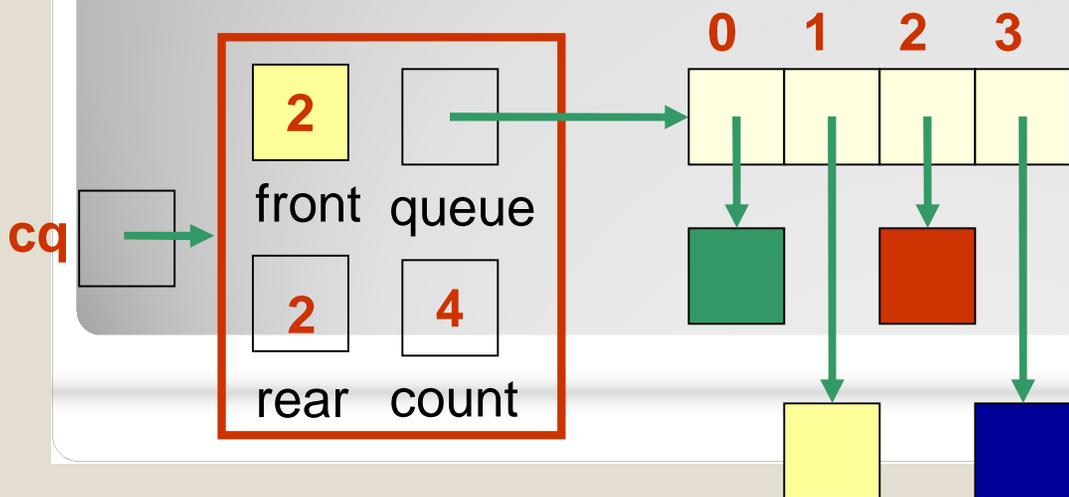
    rear = (rear+1) % queue.length;

- Can this array implementation also reach capacity?
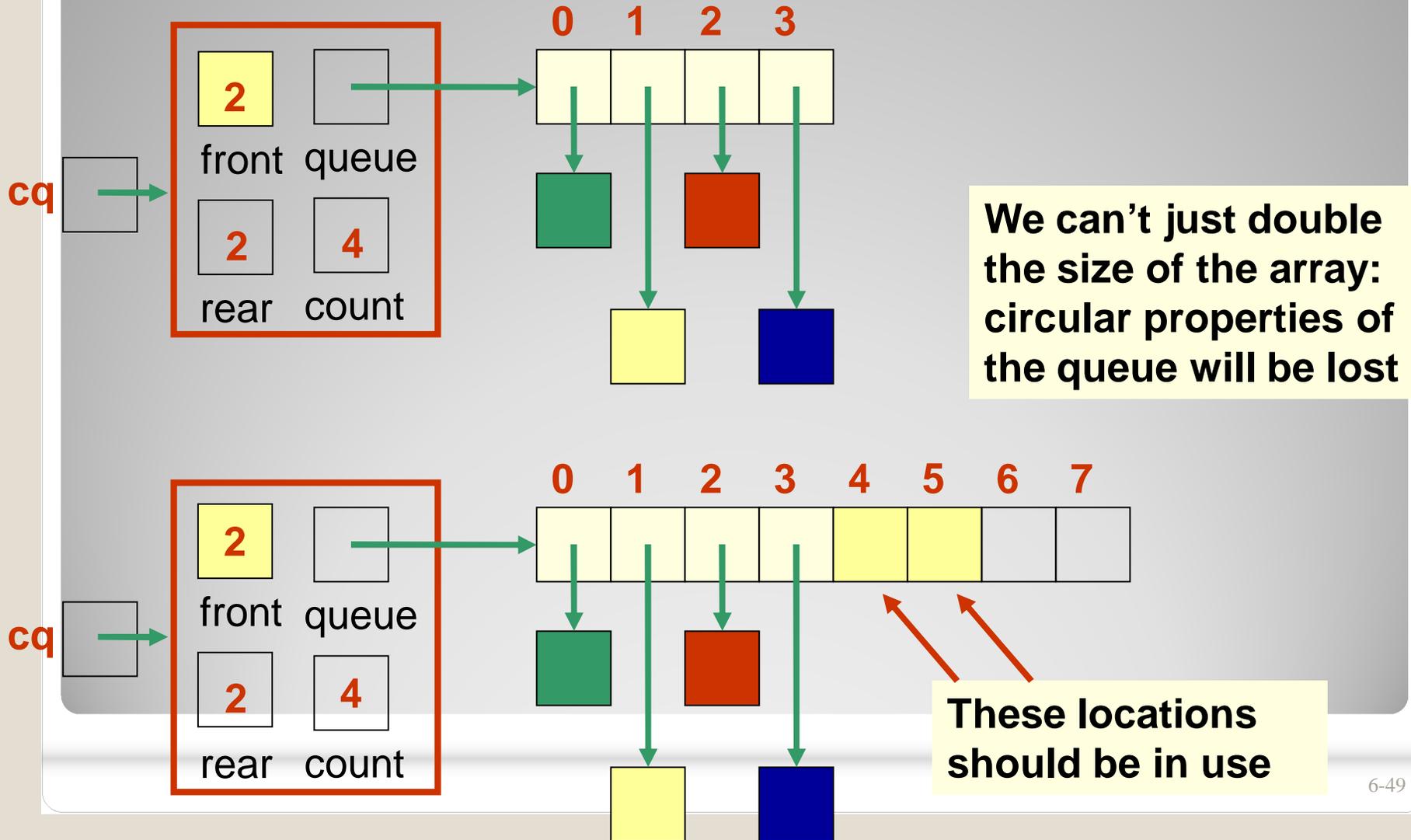
Example: array of length 4 What happens?

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

The queue is now full. How can you tell?

Add another item!
Need to expand capacity…

We can't just double the size of the array: circular properties of the queue will be lost

These locations should be in use

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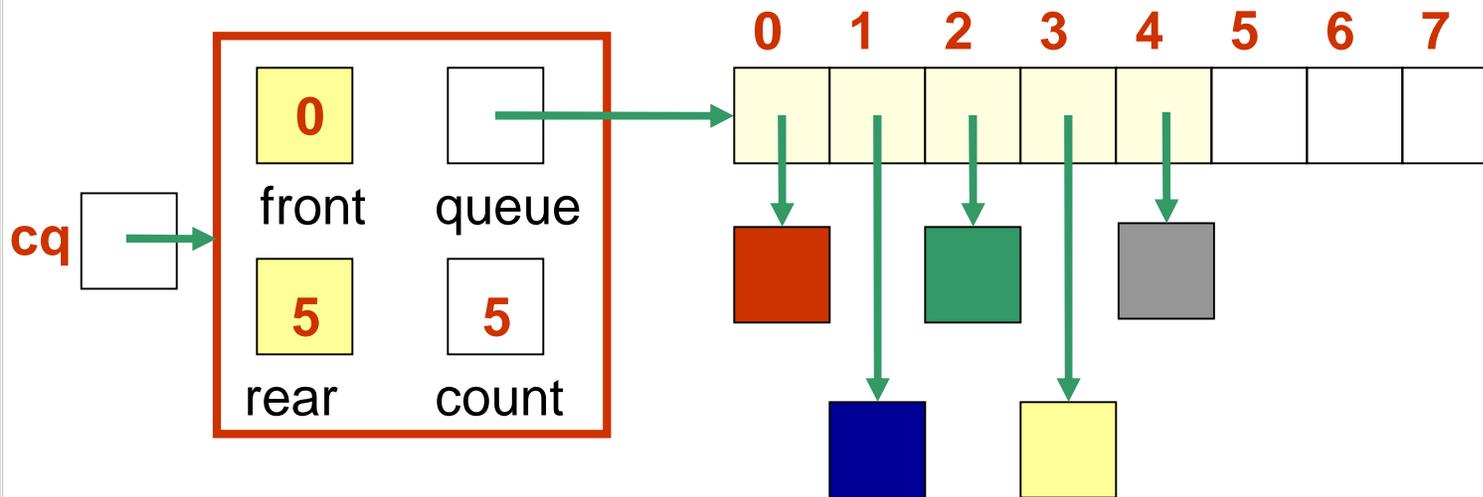
cq

2
front

queue

6
rear

4
count

0 1 2 3 4 5 6 7
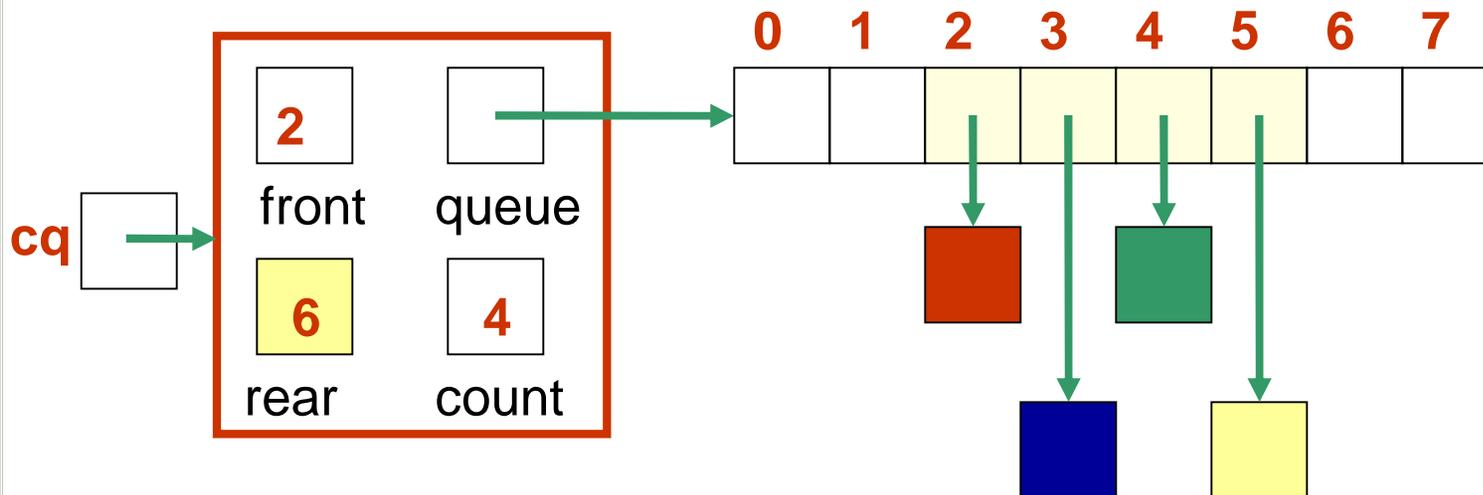
Better: copy the queue elements in order to the *beginning* of the new array
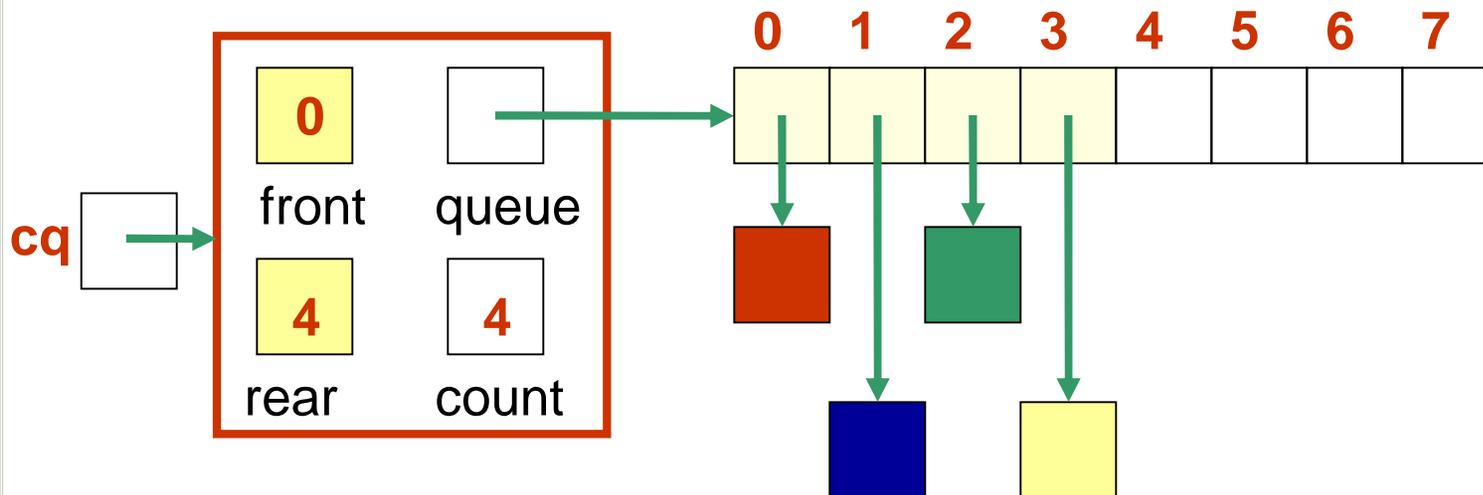
New element is added at rear = (rear+1) % queue.length

See *expandCapacity()* in *CircularArrayQueue.java*

Analysis of Queue Operations

- The linked implementation of a queue does not suffer because of the need to operate on both ends of the queue (why not?)

- **enqueue** operation:
  - **O(1)** for linked implementation
  - **O(n)** for circular array implementation if need to expand capacity, **O(1)** otherwise
  - What about the noncircular array implementation?

Analysis of Queue Operations

- **dequeue** operation:
    - **O(1)** for linked implementation
    - **O(1)** for circular array implementation
    - **O(n)** for noncircular array implementation (why?)