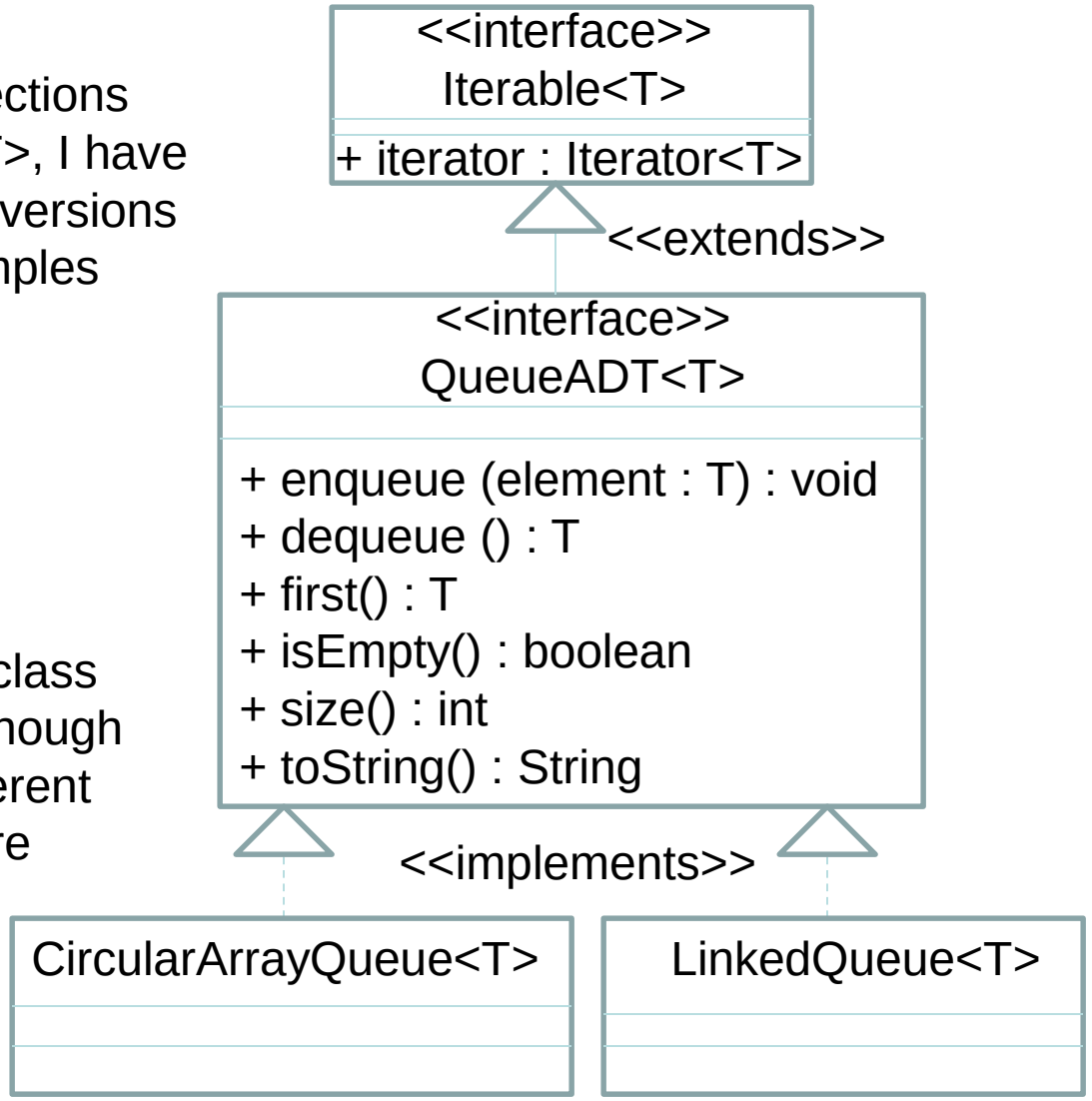
Queues (Continued)

- Queue ADT
- Linked queue implementation
- Array queue implementation
- Circular array queue implementation
- Deque
- Reading L&C 5.1-5.8, 9.3

QueueADT and Queue Classes

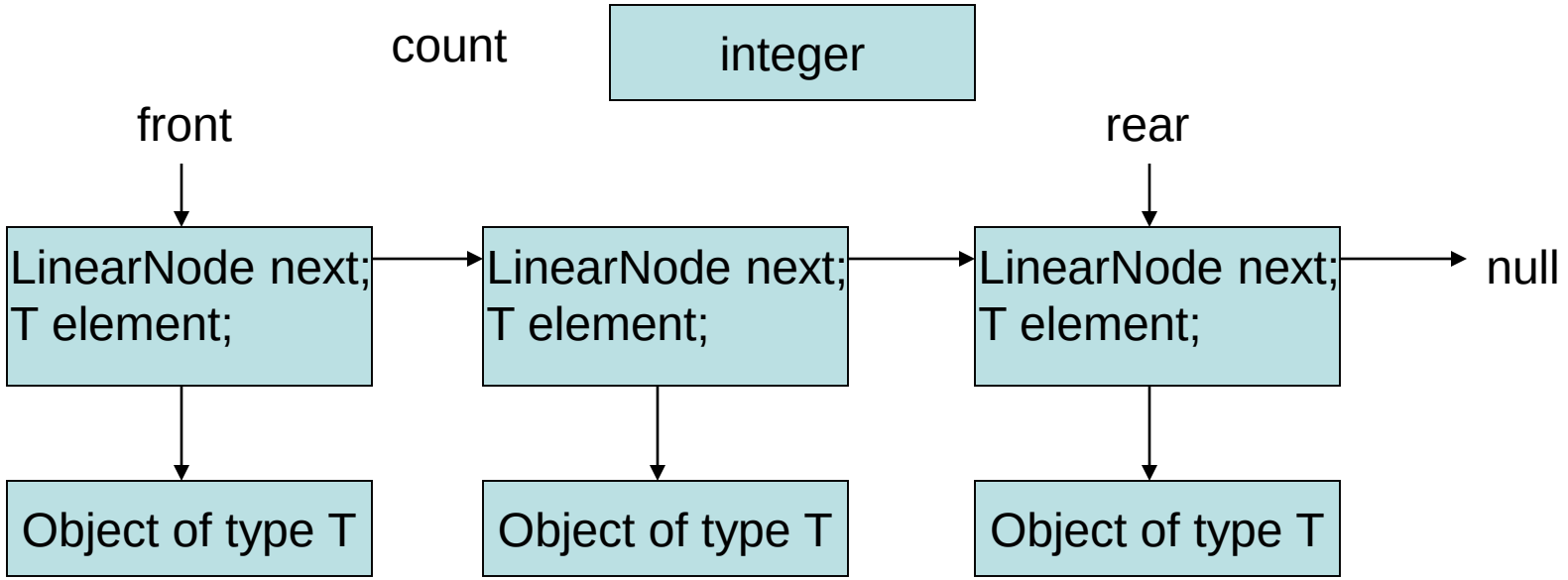
Since the Java Collections all extend Iterable<T>, I have added that to all my versions of the textbook examples



Each implementing class satisfies the ADT although they each use a different internal data structure

Linked Queue Implementation

- We can use the same LinearNode class that we used for LinkedStack implementation
- We use attribute names “front” and “rear” to have a meaning consistent with a queue



Linked Queue Implementation

- enqueue – O(1)

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

- Note the difference between the enqueue method and the stack push method

Linked Queue Implementation

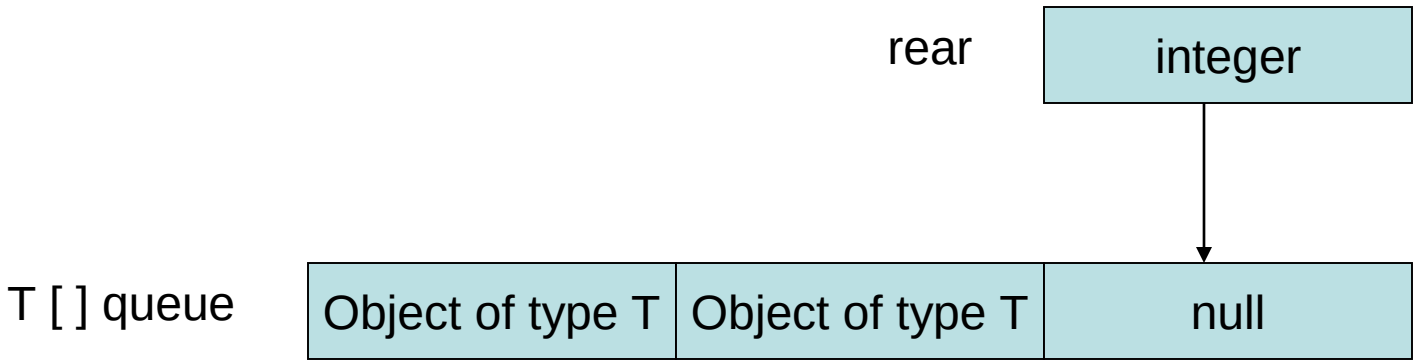
- dequeue – O(1)

```
public T dequeue () throws EmptyQueueException
{
  if (isEmpty())  throw new EmptyQueueException();
  T result = front.getElement();
  front = front.getNext();  // may create garbage
  if (--count == 0)
     rear = null;    // finishes creating garbage
  return result;
}
```

- Note the difference between the dequeue method and the stack pop method

Array Queue Implementation

- We can use an array of elements as a queue
- The front is implicitly index 0 and rear is the index of next available element in the array
- Variable “rear” is also used for count



Array Queue Implementation

- enqueue – O(1)

```
public void enqueue (T element)
{
  if (size() == queue.length)
     expandCapacity();
  queue[rear++] = element;
}
```

- expandCapacity is similar to private helper method used in ArraySet and Stack classes

Array Queue Implementation

- dequeue() – O(n)

```
public T dequeue() throws EmptyQueueException
{
  if (isEmpty())
        throw new EmptyStackException();
  T result = queue[0];
  rear--;
  for (int scan = 0; scan < rear; scan++)
     queue[scan] = queue[scan + 1];
  queue[rear] = null;  // stale alias
  return result;
}
```

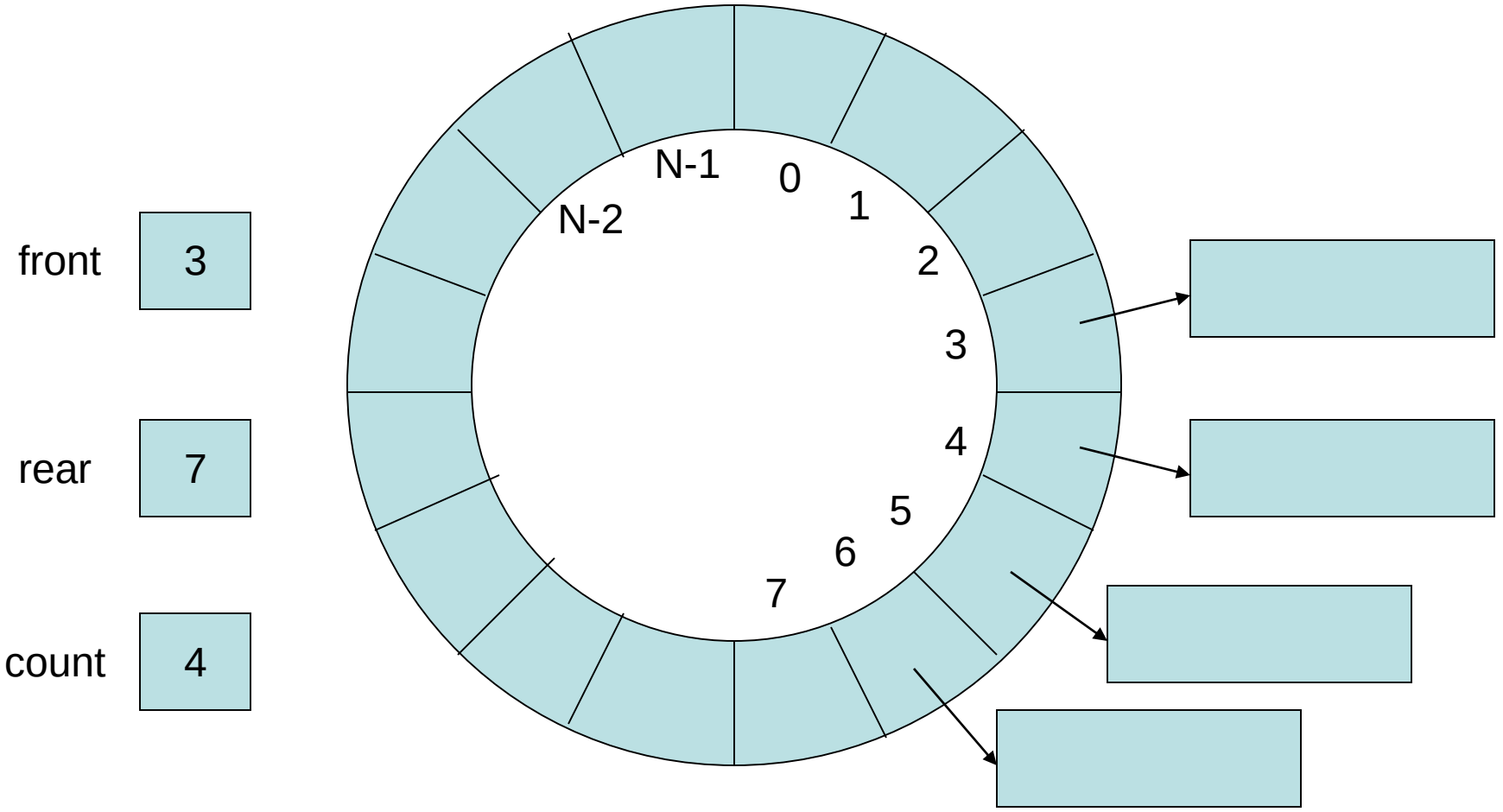
Array Queue Implementation

- Notice that the dequeue is O(n) due to the shifting of the elements in the array queue after the 0^{th} element has been copied out
- This introduces a potential performance problem that we would like to avoid
- Using the 0^{th} element of the array as the rear of the queue doesn't solve the problem – just moves it to the enqueue operation
- With a better design, we can avoid it

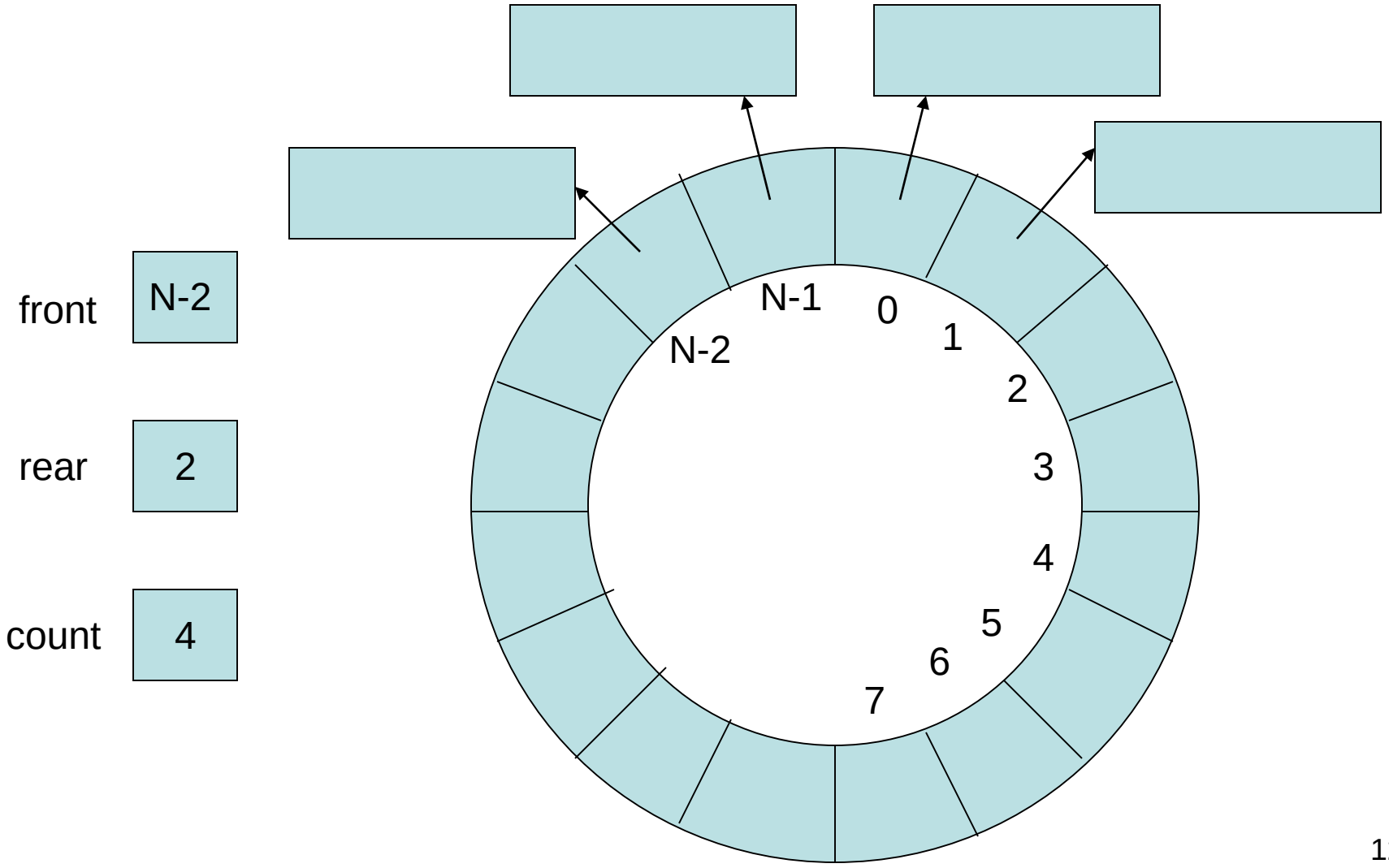
Circular Array Queue Implementation

- This design eliminates the shifting of the elements as part of the dequeue operation
- Commonly called *circular buffering*
- We keep an integer for both the front and rear of the queue in the array and never shift the elements in the array
- When we increment either front or rear to the length of the array, we do not expand the capacity. We set them back to zero to reuse the lower elements in the array

Circular Array Queue Implementation



Circular Array Queue Implementation



Circular Array Queue Implementation

- Method enqueue can not use:

```
rear++;
```

- Method dequeue can not use:

```
front++;
```

- To increment rear, enqueue must use:

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

- To increment front, dequeue must use:

```
front = (front + 1) % queue.length;
```

Circular Array Queue Implementation

- When the front catches up to the rear (a snake eating its own tail?), our code must expand the capacity of the array (replacing the original array with a larger one)
- When our code expands the capacity, it must cycle through the original array from front index to rear index value as it copies from the smaller array to the larger array
- Then, it sets new values for front and rear

Queue Class Iterators

- Again, we need to provide an iterator method and an Iterator class (best implemented as an inner class)
- We want the iterator to provide the elements in the order of the queue from front to rear
- This would be:
  - For a LinkedQueue: The same order as for a LinkedStack's Iterator (Code not shown here)
  - For a CircularArrayQueue: Opposite of the order as for the ArrayStack's Iterator classes

ArrayIterator Class

- The iterator method for each Queue class instantiates and returns a reference to a new ArrayIterator object to its caller
- Any iterator class is closely related to its collection class so it is a good candidate for implementation as an inner class
- As an inner class, the ArrayIterator code can access the array and front/rear variables of the instance of the outer class that instantiated it

ArrayIterator Definition/Attributes

- Class Definition/Attribute Declarations (implemented as an inner class)

```
private class ArrayIterator<T>
                    implements Iterator<T>
{
   private int current;
```

- Constructor:

```
public ArrayIterator()
{
  current = front;  // start at front for FIFO
}
```

ArrayIterator Methods

- hasNext – O(1)

```
public boolean hasNext()
{
  return current != rear;     // outer class variable
}
```

- next – O(1)

```
public T next()
{
  if (!hasNext())
     throw new NoSuchElementException();
  T result = queue[current]; // outer class array
  current = (current + 1) % queue.length;
  return result;
}
```

Deque

- A Deque (pronounced like “deck”) is a data structure that is a double ended queue
- It can be used as either a stack or a queue depending on the methods your code uses
- Look at the Deque class in the Java APIs
- Note the name of each method and what it does to use a Deque data structure correctly (the names are not the traditional ones)

Deque

- If we use a Deque for our traceback stack instead of a Stack, we could add into it as a stack and then remove from it as a queue
- Then we wouldn’t need to use another stack to reverse the order of the elements in order to print them from first to last
- If you want, try a Deque instead of a Stack or Queue in one of our labs or projects