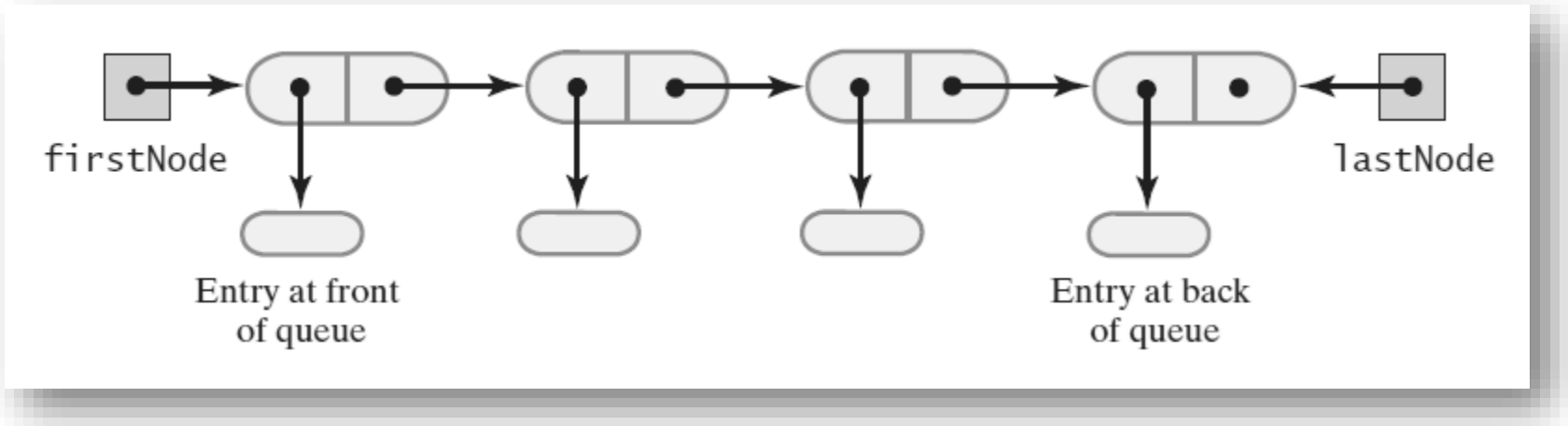
Last Class

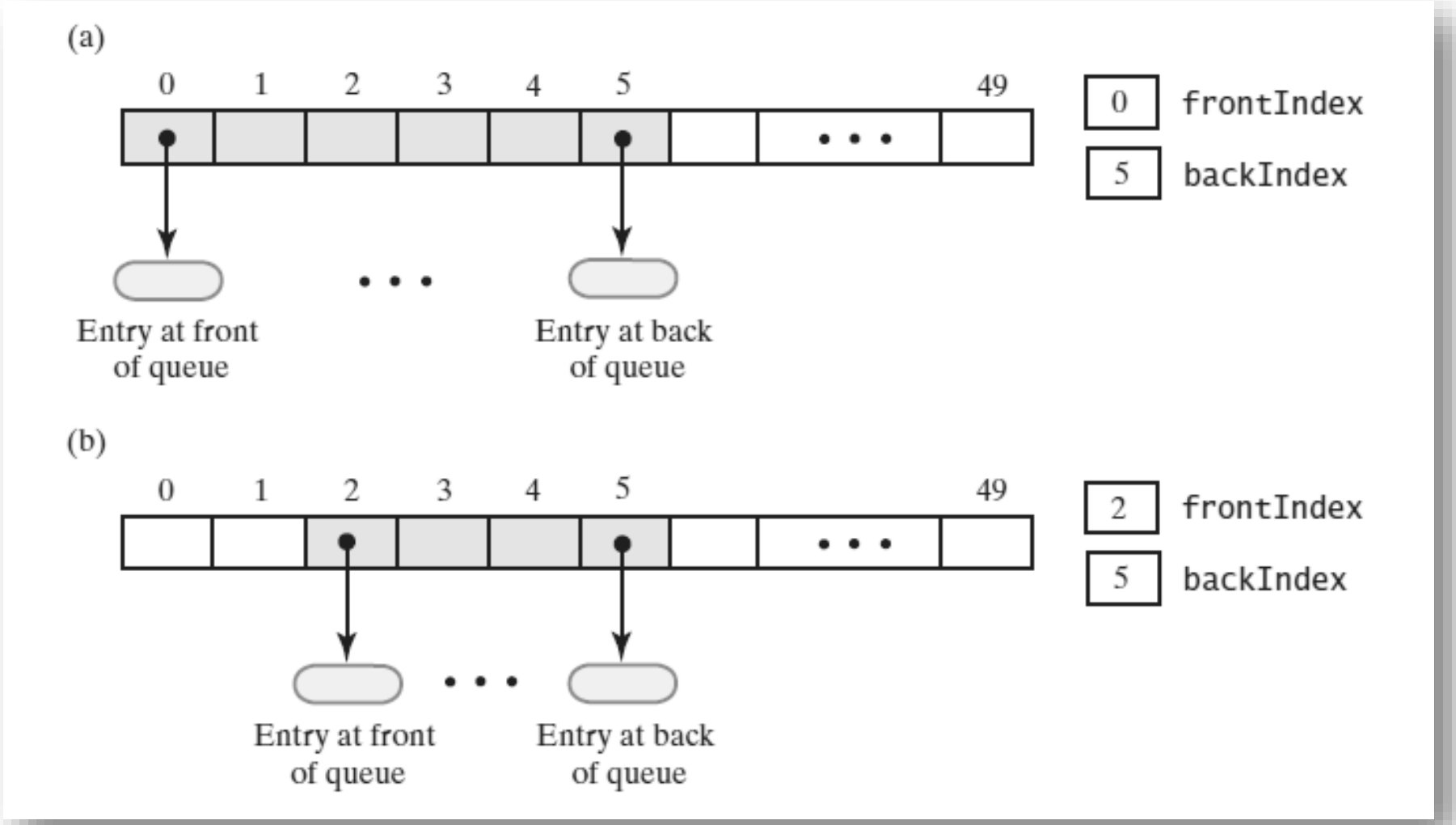
```
public interface QueueInterface<T>
{
 /** Adds a new entry to the back of this queue.
     @param newEntry  An object to be added. */
 public void enqueue(T newEntry);

 /** Removes and returns the entry at the front of this queue.
     @return  The object at the front of the queue.
     @throws  EmptyQueueException if the queue is empty before the operation. */
 public T dequeue();

 /**  Retrieves the entry at the front of this queue.
     @return  The object at the front of the queue.
     @throws  EmptyQueueException if the queue is empty. */
 public T getFront();

 /** Detects whether this queue is empty.
     @return  True if the queue is empty, or false otherwise. */
 public boolean isEmpty();

 /** Removes all entries from this queue. */
 public void clear();
} // end QueueInterface
```

Last Class:
A Linked Implementation of a Queue

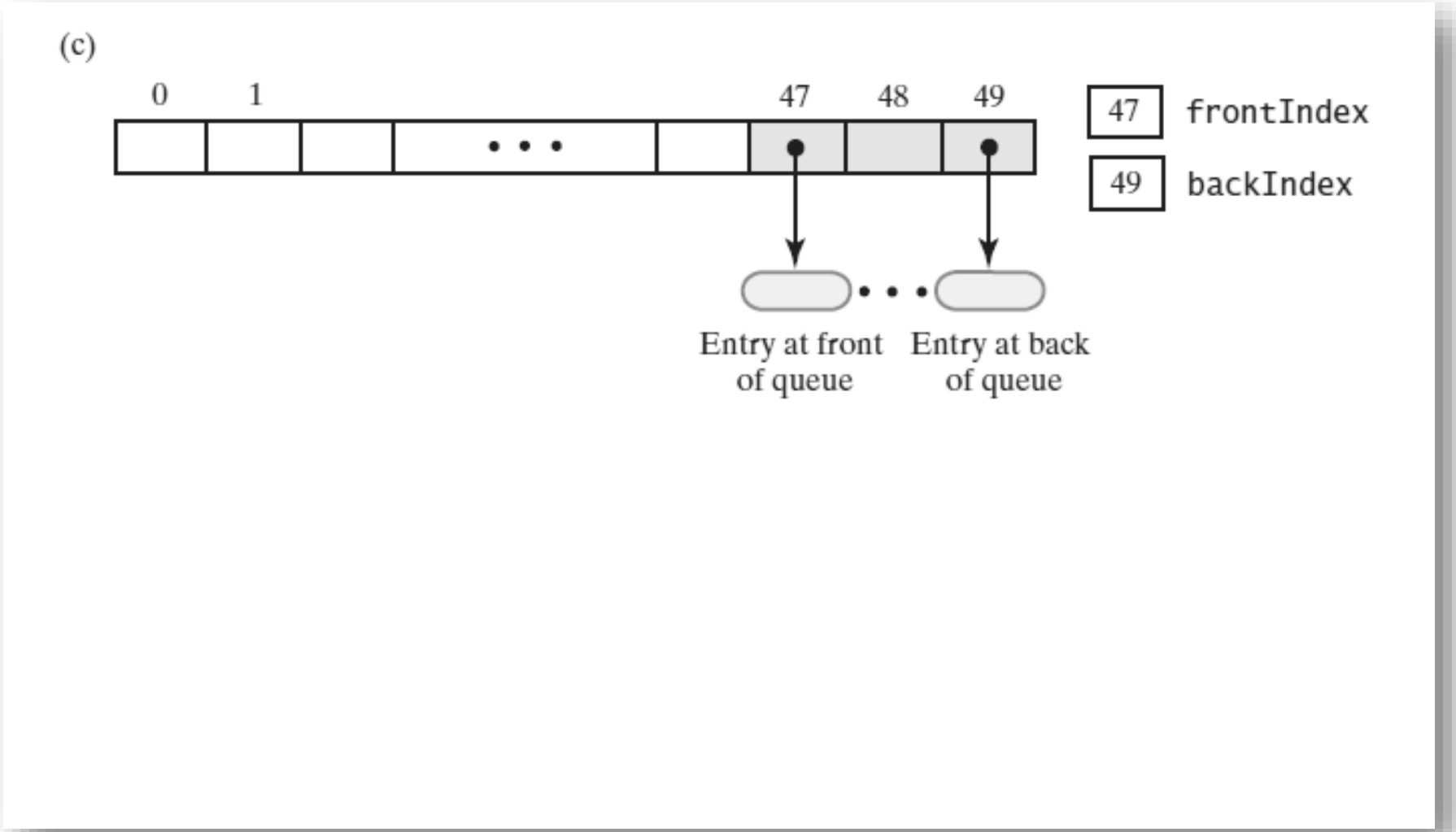


Array-based implementation of a Queue using Circular Array

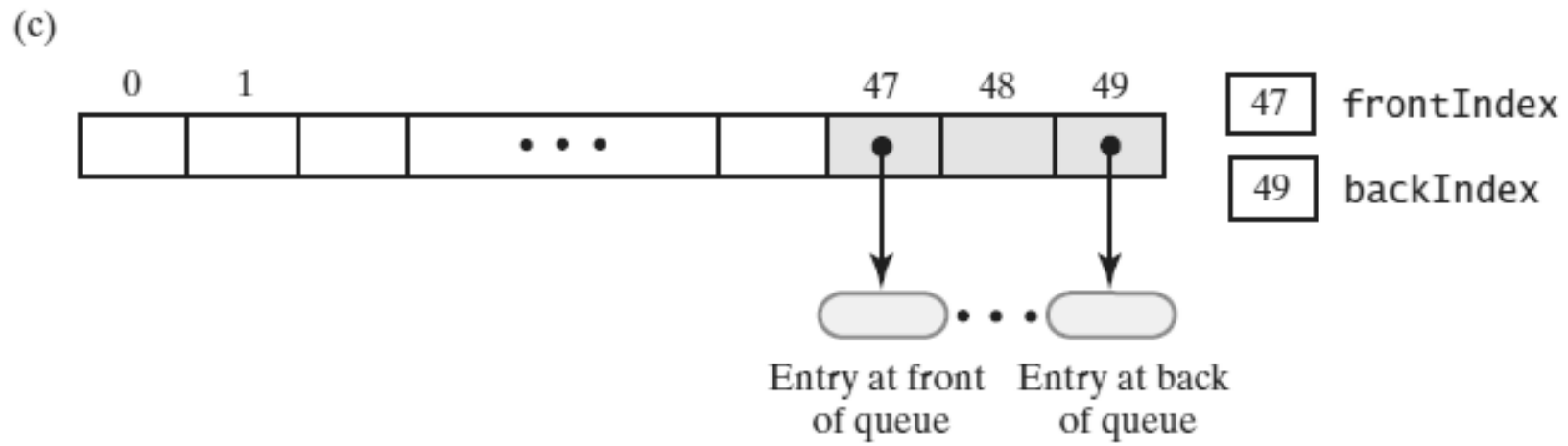
An Array-Based Implementation of a Queue: Circular Array



An Array-Based Implementation of a Queue: Circular Array (cont.)

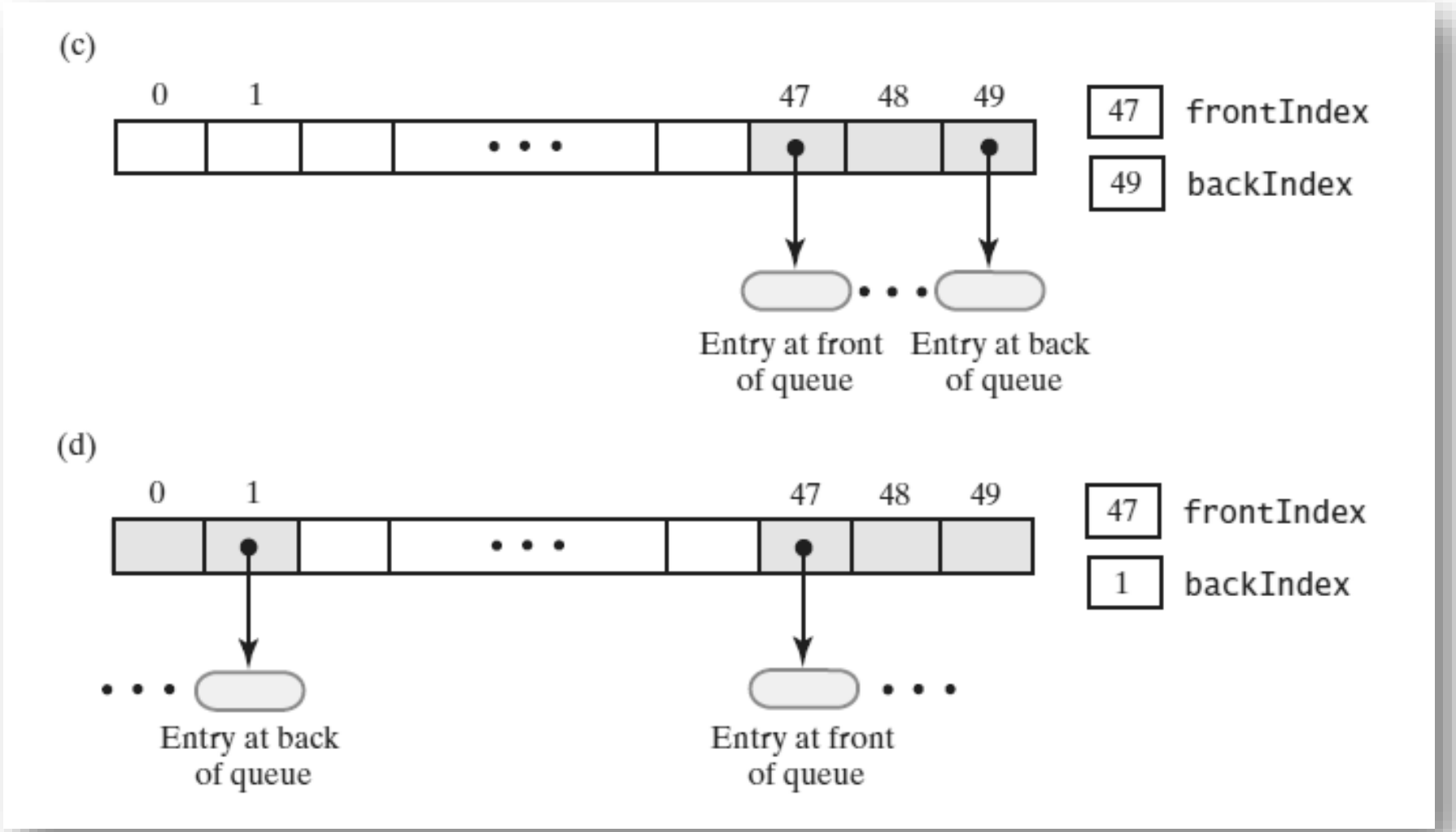


An Array-Based Implementation of a Queue: Circular Array (cont.)

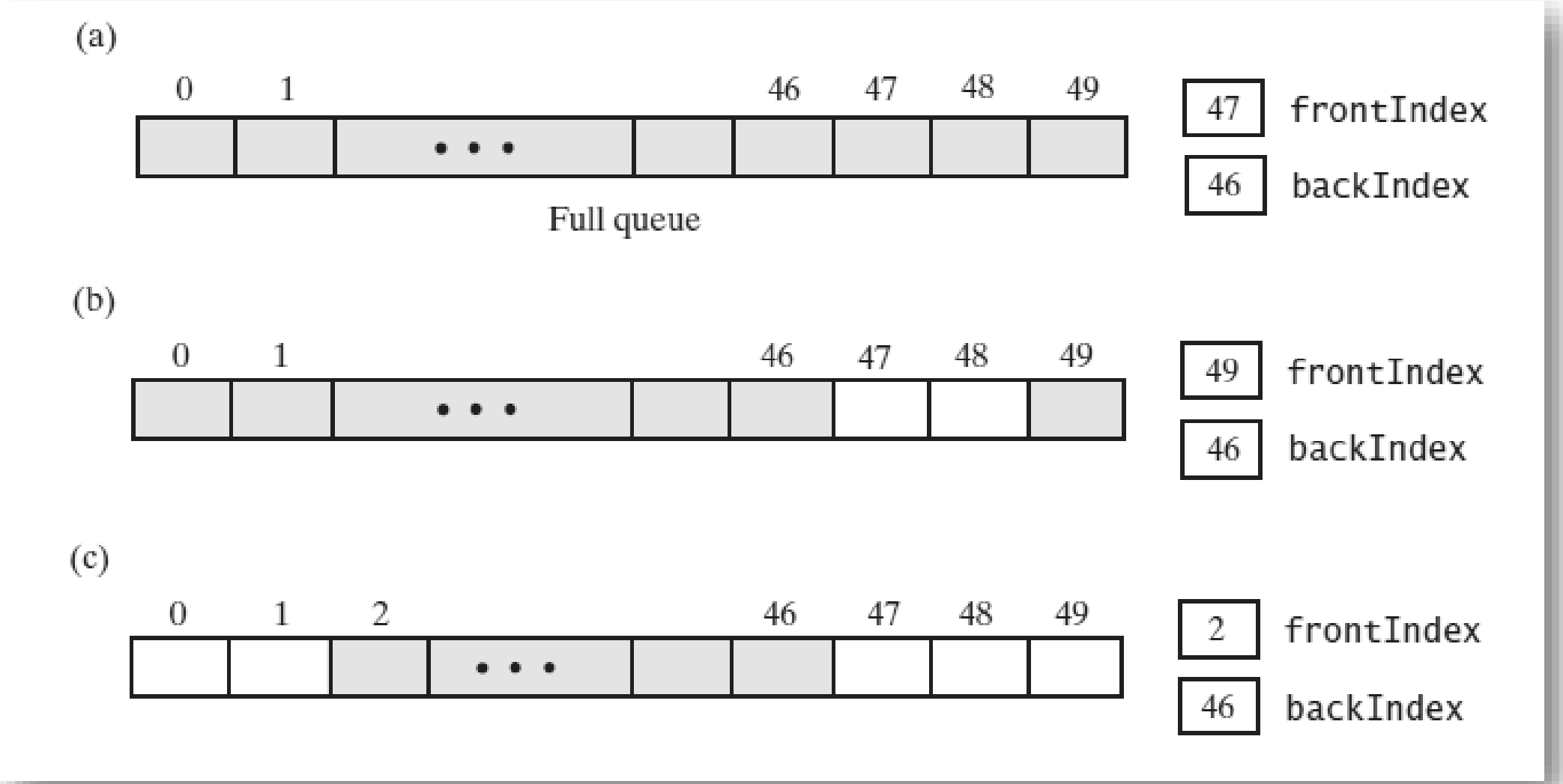



```
backIndex = (backIndex + 1) % queue.length();
```

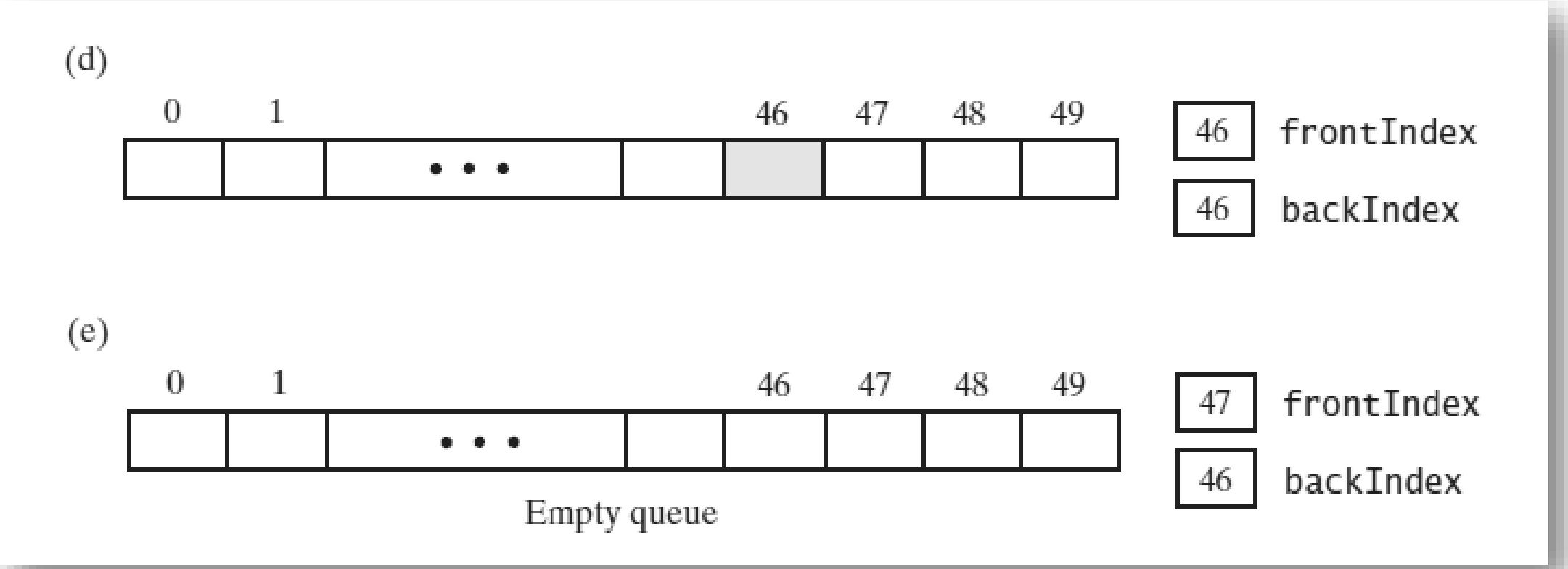
An Array-Based Implementation of a Queue: Circular Array (cont.)



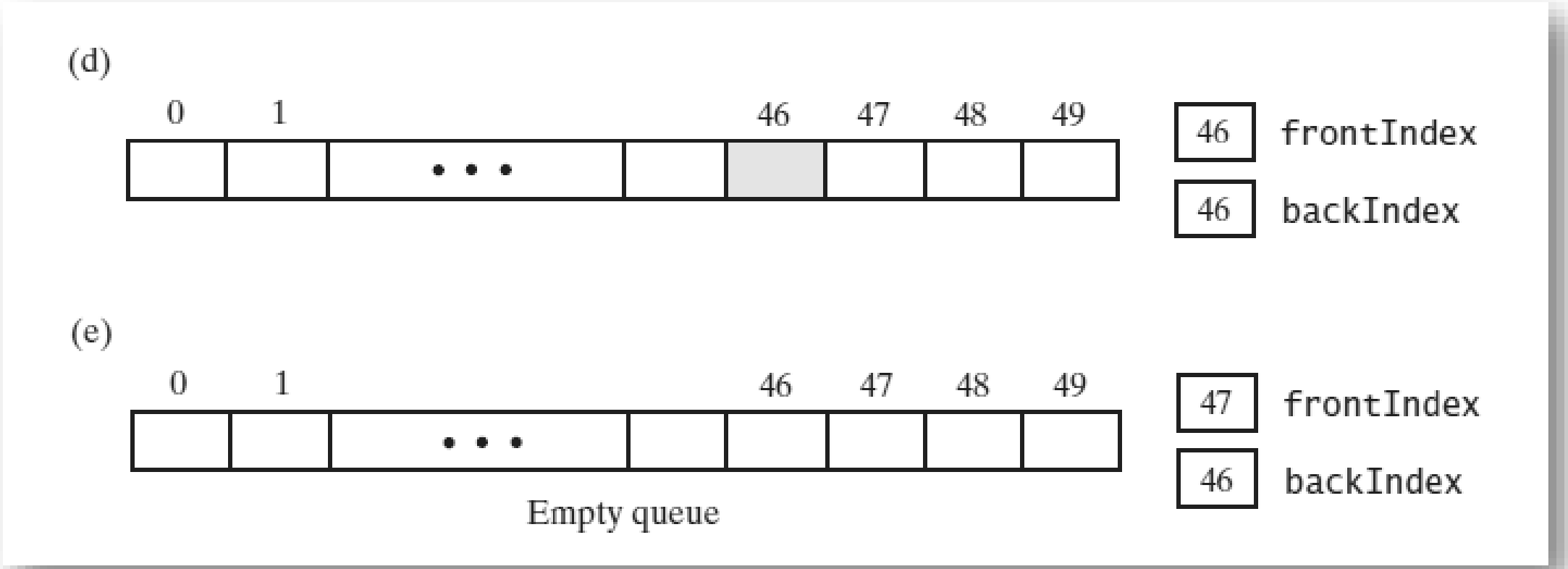
Complications



Complications (cont.)



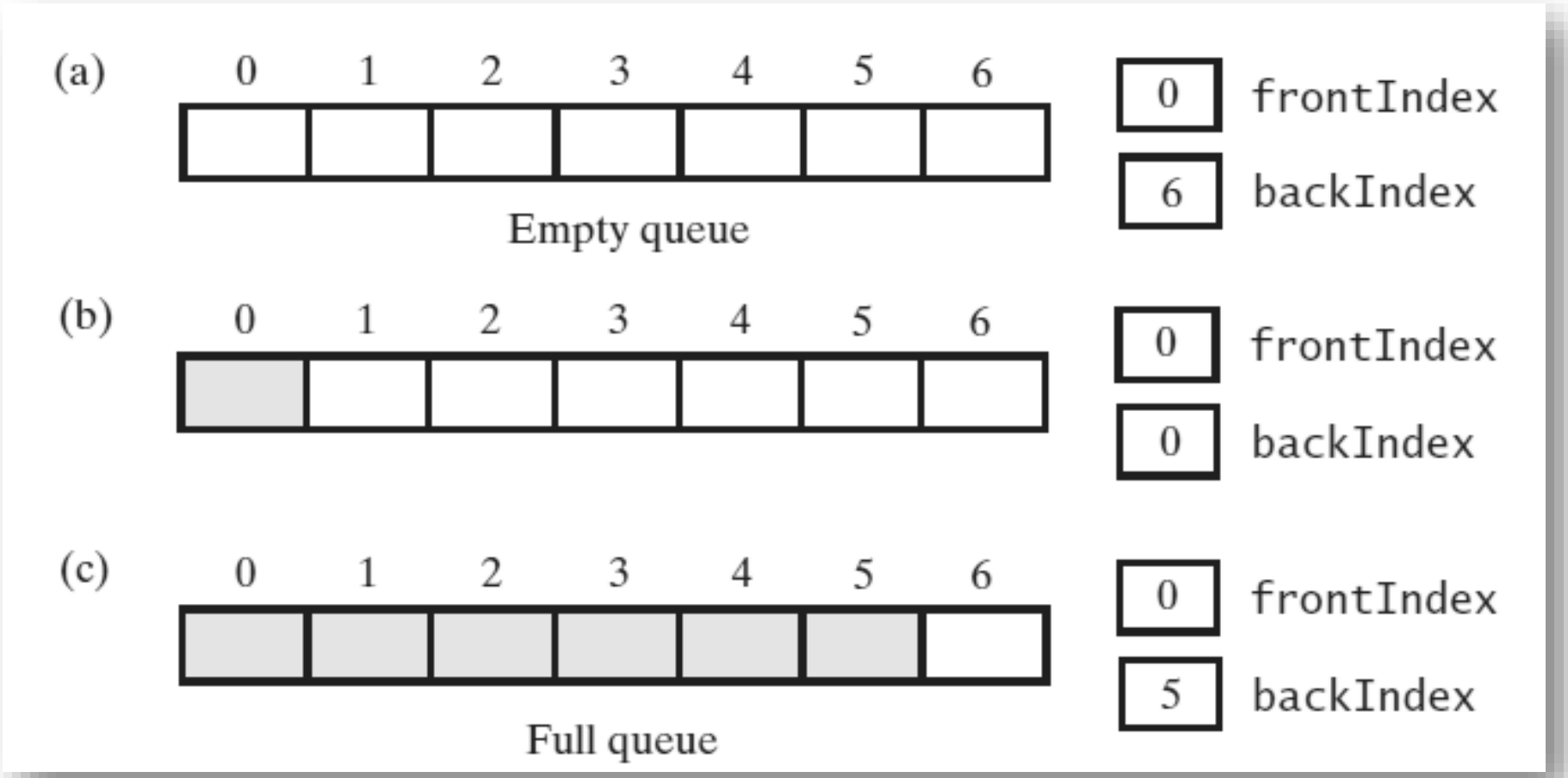
Complications (cont.)



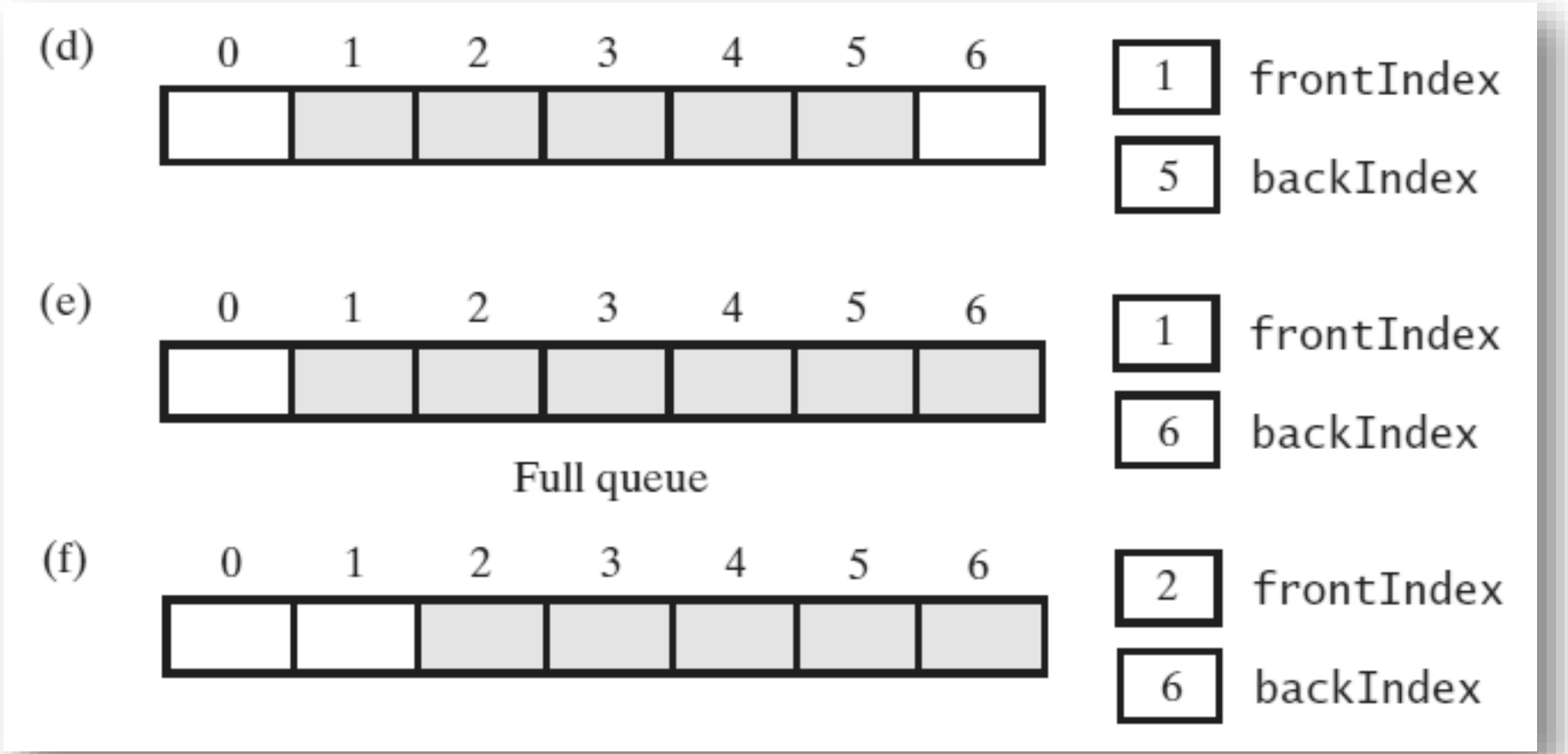
Note: With a circular array, **frontIndex** equals **backIndex+1** both when the queue is empty and when it is full.

A different approach for array-based implementation

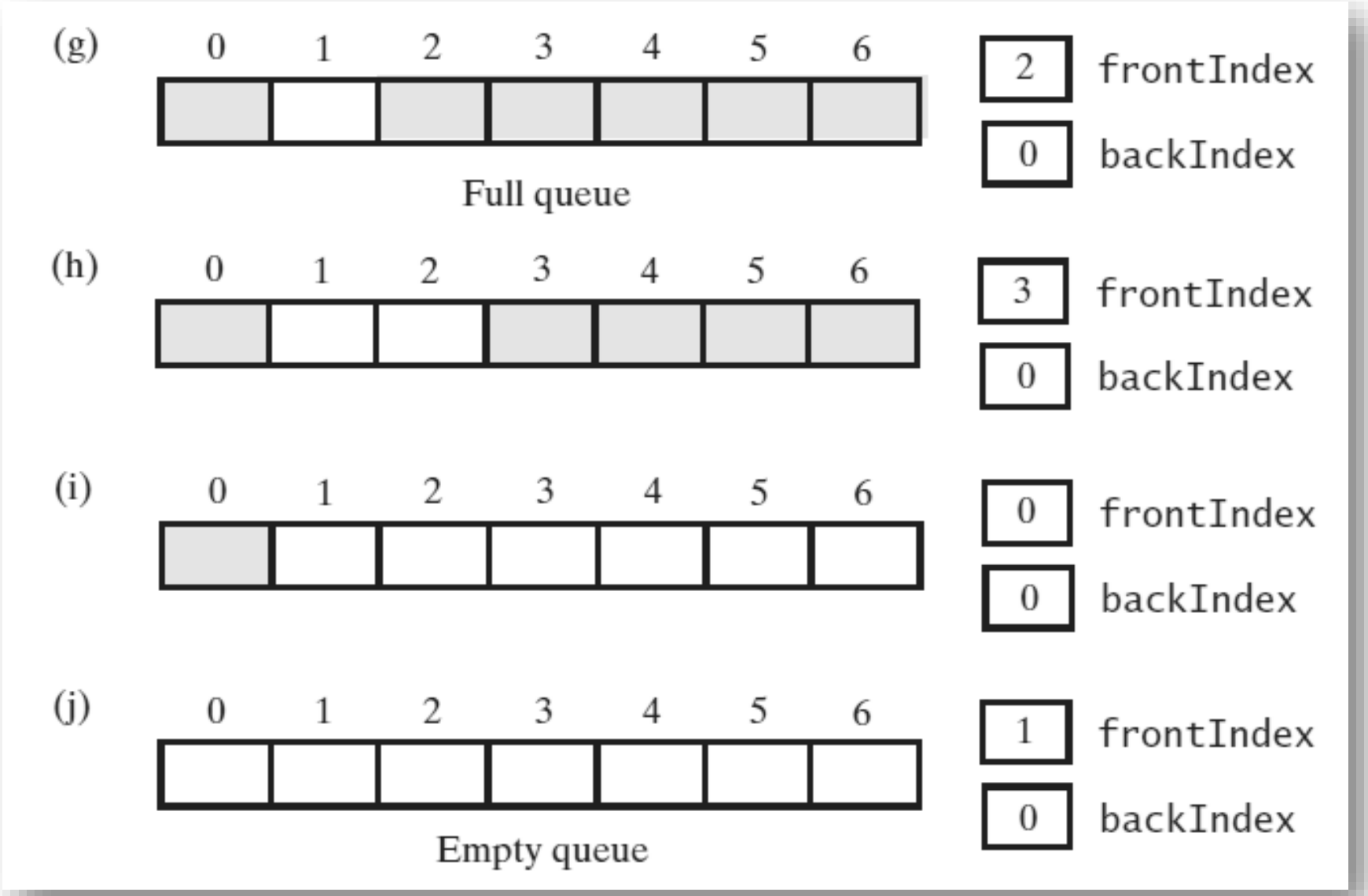
A Circular Array with One Unused Location



A Circular Array with One Unused Location (cont.)



A Circular Array with One Unused Location (cont.)



A Circular Array with One Unused Location (cont.)

- To summarize, the queue is full when

  frontIndex equals **(backIndex+2) % queue.length**

- and the queue is empty when

  frontIndex equals **(backIndex+1) % queue.length**

```java
public final class ArrayQueue<T> implements QueueInterface<T>
{
 private T[] queue; // Circular array of queue entries and one unused location
 private int frontIndex;
 private int backIndex;
 private boolean initialized = false;
 private static final int DEFAULT_CAPACITY = 50;
 private static final int MAX_CAPACITY = 10000;

 public ArrayQueue() { this(DEFAULT_CAPACITY); } // end default constructor

 public ArrayQueue(int initialCapacity)
 {
  checkCapacity(initialCapacity);

  // The cast is safe because the new array contains null entries
  @SuppressWarnings("unchecked")
  T[] tempQueue = (T[]) new Object[initialCapacity + 1];
  queue = tempQueue;
  frontIndex = 0;
  backIndex = initialCapacity;
  initialized = true;
 } // end constructor

 //  < Implementations of the queue operations go here. >
 //  . . .
} // end ArrayQueue
```

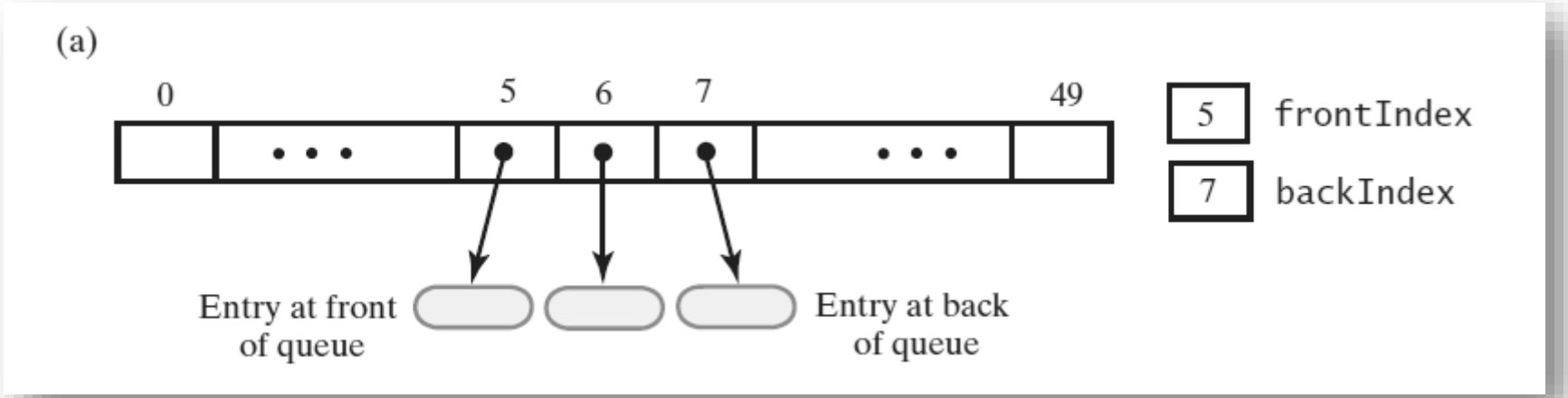
Adding to the back

```
public void enqueue(T newEntry)
{
 checkInitialization();
 ensureCapacity();
 backIndex = (backIndex + 1) % queue.length;
 queue[backIndex] = newEntry;
} // end enqueue
```

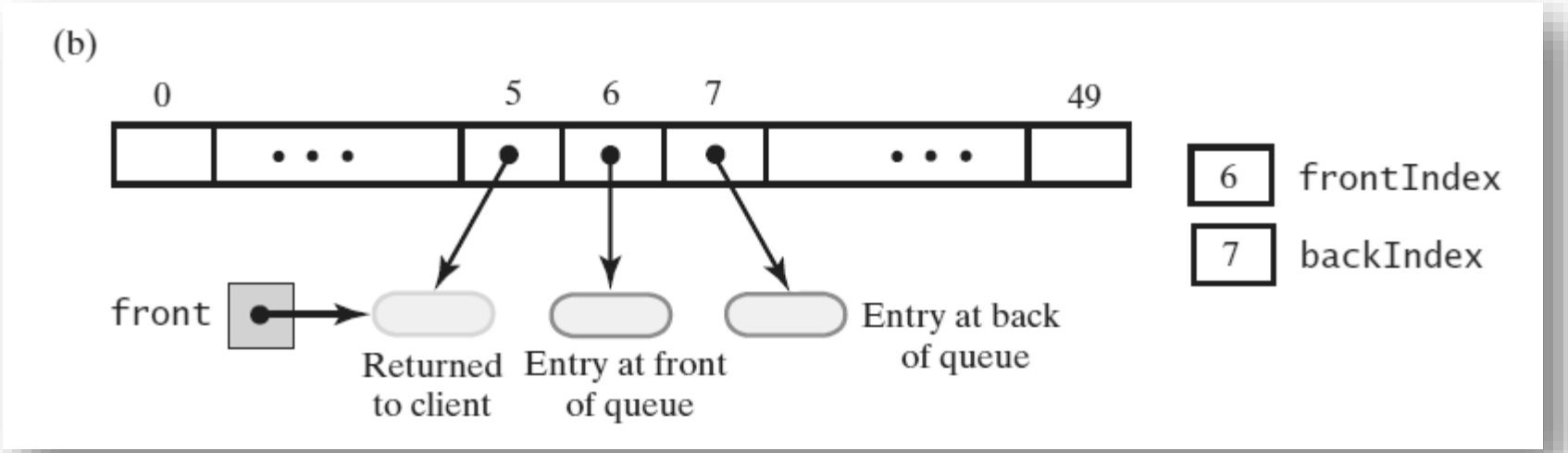
Retrieving the front entry

```
public T getFront()
{
 checkInitialization();
 if (isEmpty())
  throw new EmptyQueueException();
 else
  return queue[frontIndex];
} // end getFront
```

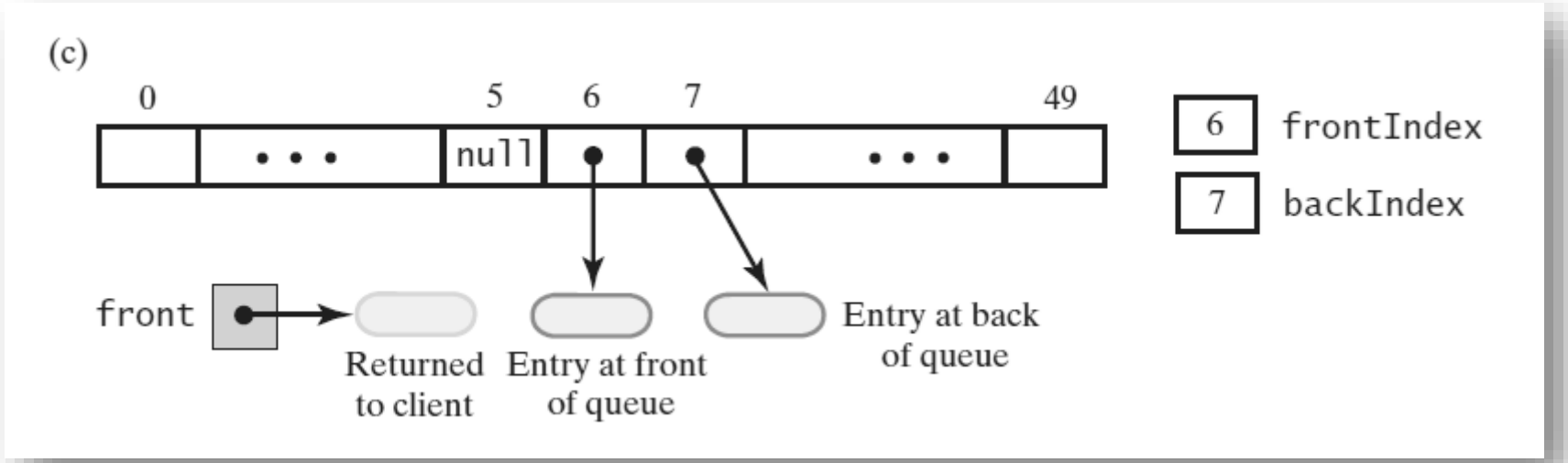
Removing the front entry



Removing the front entry (cont.)



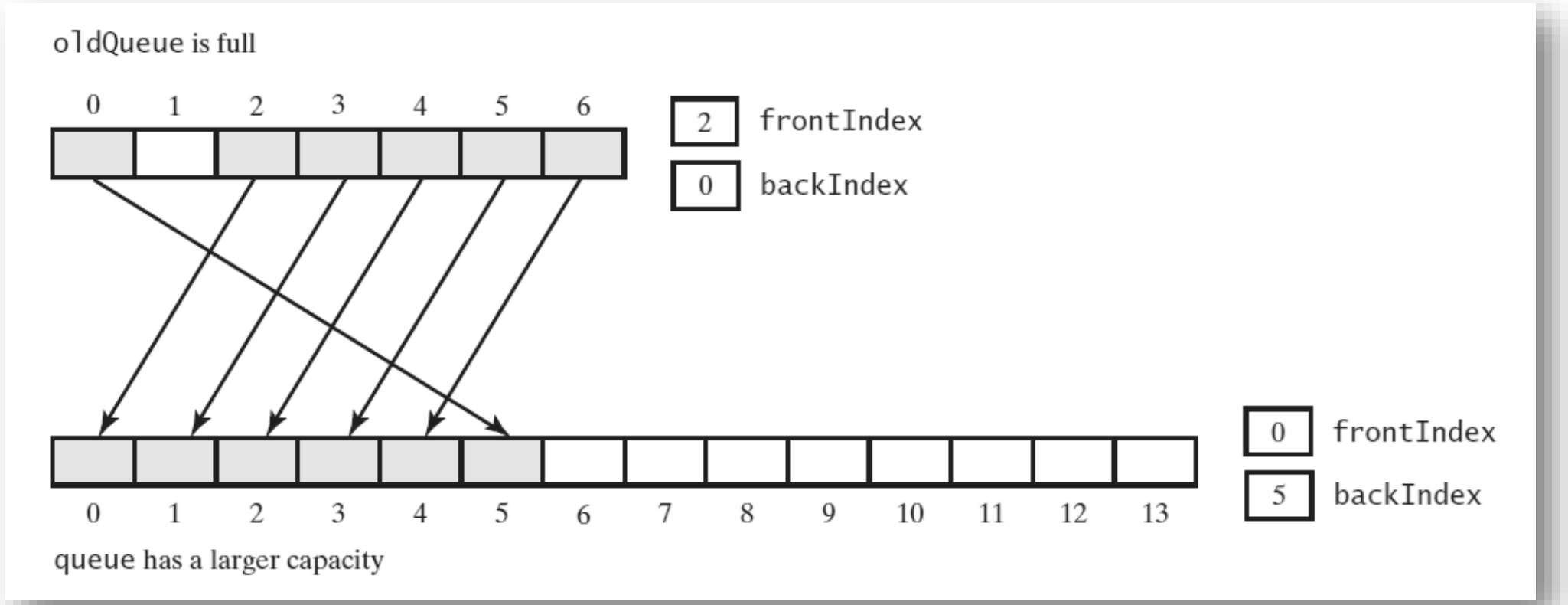
Removing the front entry (cont.)



Removing the front entry (cont.)

```
public T dequeue()
{
 checkInitialization();
 if (isEmpty())
  throw new EmptyQueueException();
 else
 {
  T front = queue[frontIndex];
  queue[frontIndex] = null;
  frontIndex = (frontIndex + 1) % queue.length;
  return front;
 } // end if
} // end dequeue
```

The private method **ensureCapacity**



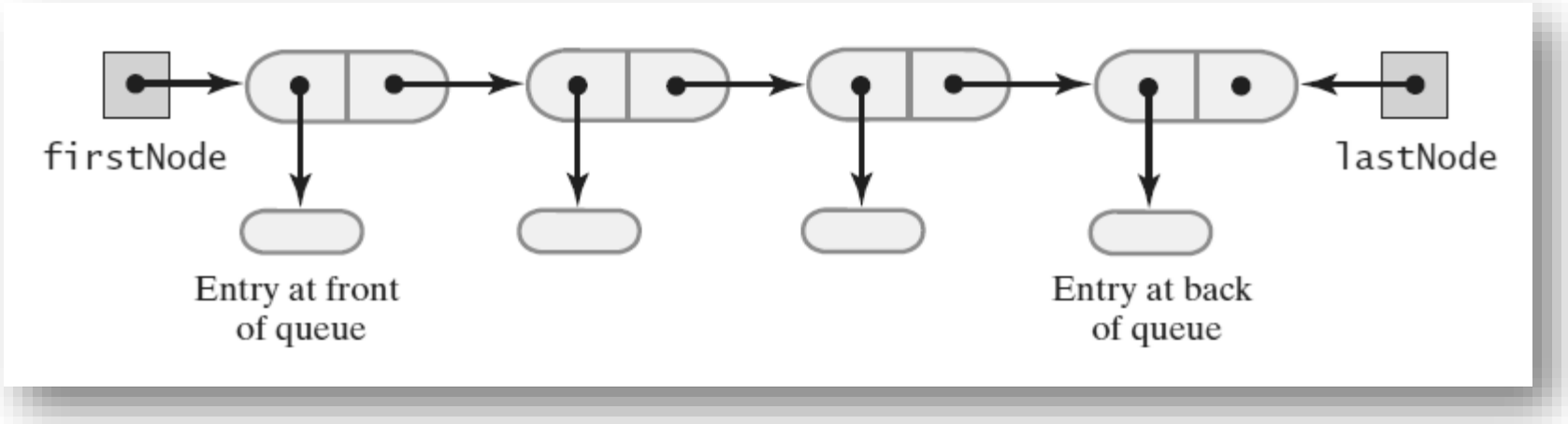
```java
// Doubles the size of the array queue if it is full
// Precondition: checkInitialization has been called.
private void ensureCapacity()
{
 if (frontIndex == ((backIndex + 2) % queue.length)) // if array is full,
 {                                                   // double size of array
  T[] oldQueue = queue;
  int oldSize = oldQueue.length;
  int newSize = 2 * oldSize;
  checkCapacity(newSize);

  // The cast is safe because the new array contains null entries
  @SuppressWarnings("unchecked")
  T[] tempQueue = (T[]) new Object[2 * oldSize];
  queue = tempQueue;
  for (int index = 0; index < oldSize - 1; index++)
  {
   queue[index] = oldQueue[frontIndex];
   frontIndex = (frontIndex + 1) % oldSize;
  } // end for

  frontIndex = 0;
  backIndex = oldSize - 2;
 } // end if
} // end ensureCapacity
```

The rest of the class

```
public boolean isEmpty()
{
 return frontIndex == ((backIndex + 1) % queue.length);
} // end isEmpty
```

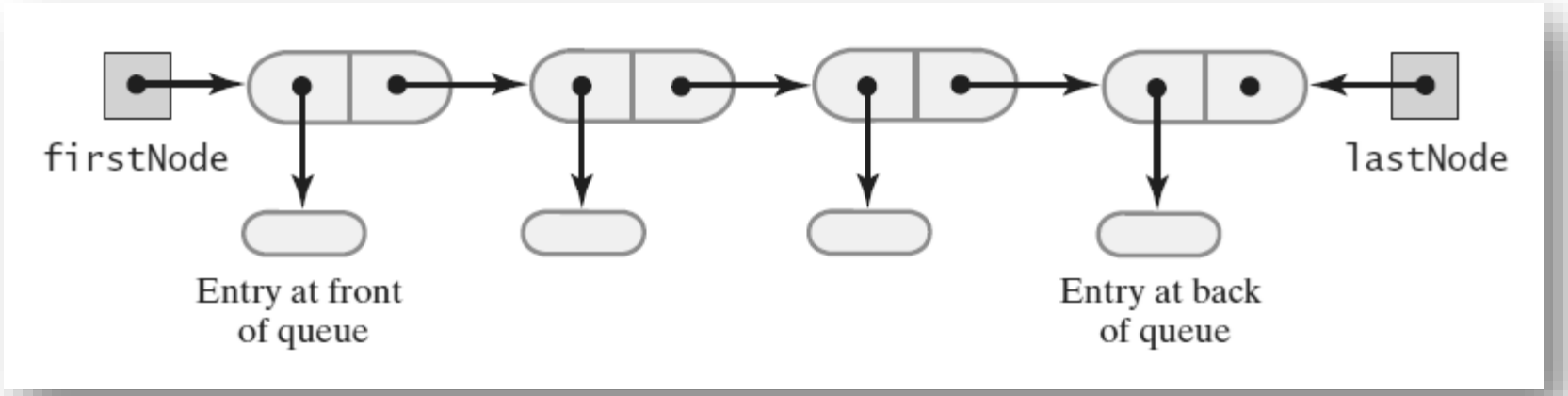
Circular Implementation of a Queue using Linked Chains

Circular Linked Implementations of a Queue

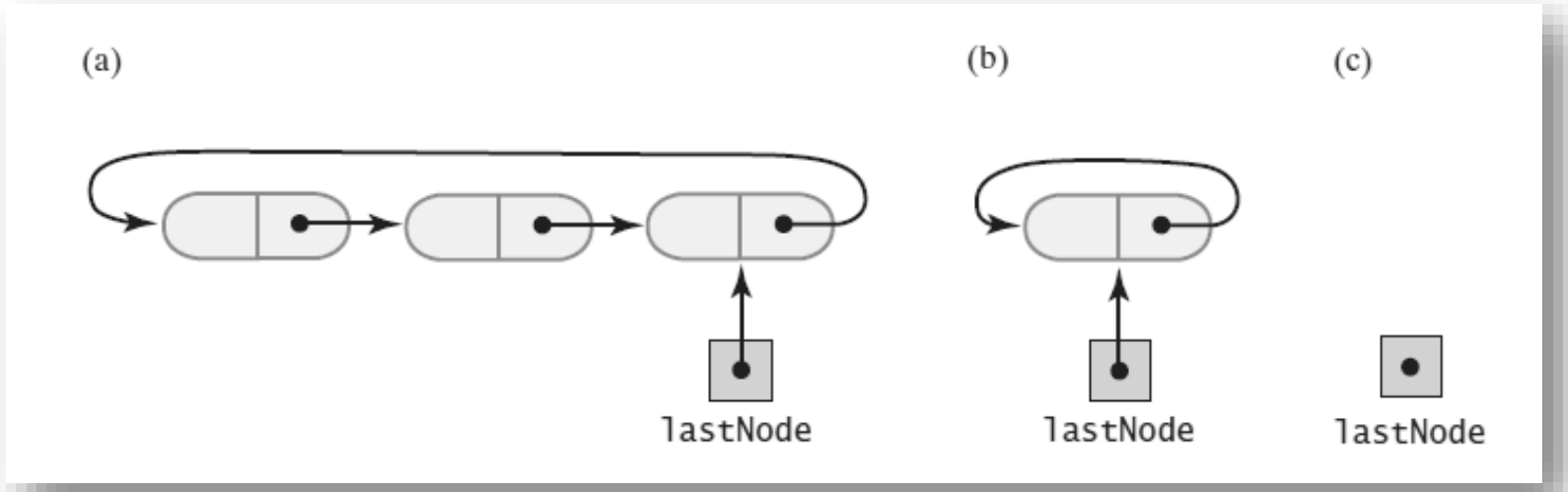


Linear linked chain

Circular Linked Implementations of a Queue



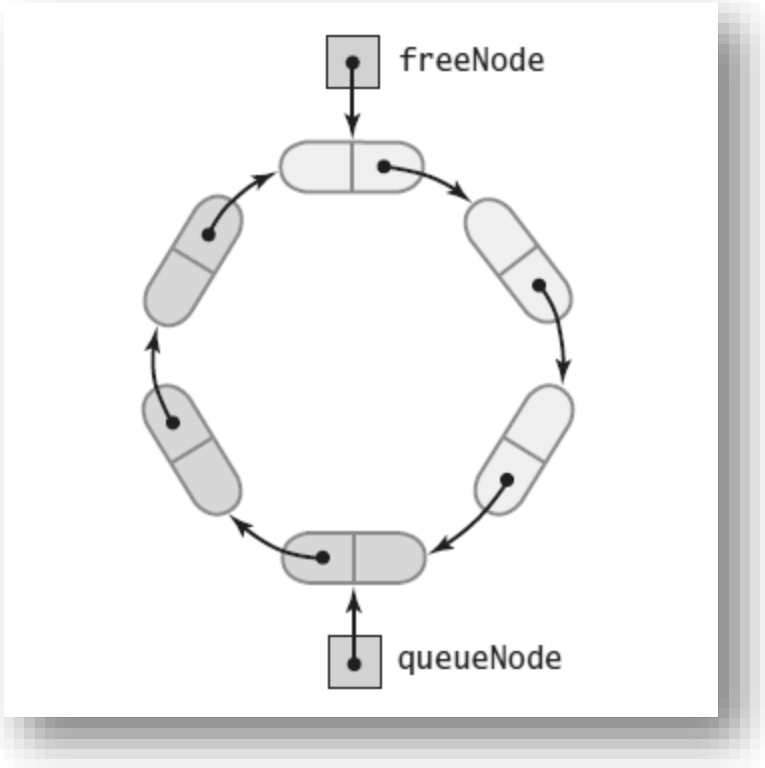
Linear linked chain



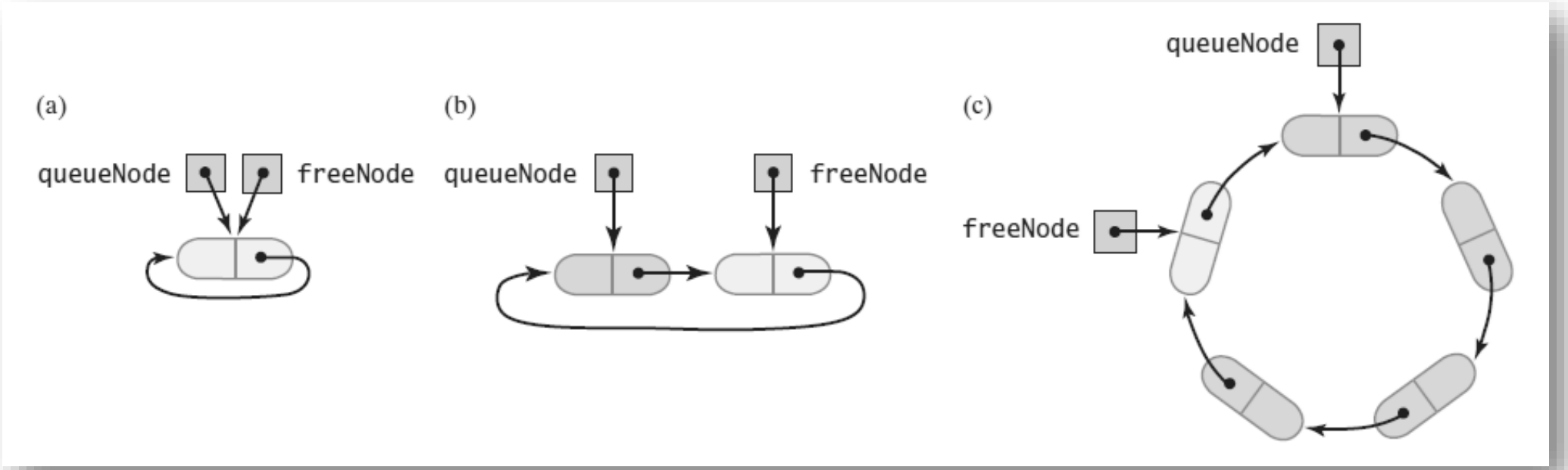
Circular linked chain

A Two-Part Circular Implementation of a Queue using Linked Chains

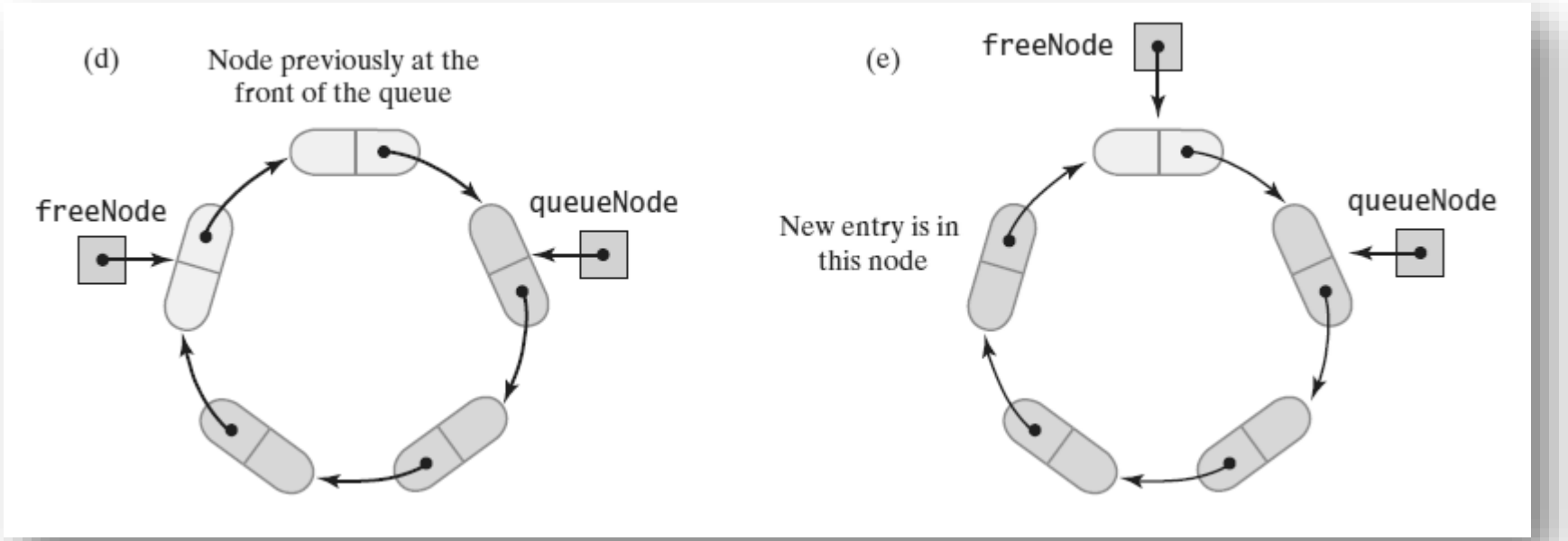
A Two-Part Circular Linked Chain



A Two-Part Circular Linked Chain (cont.)



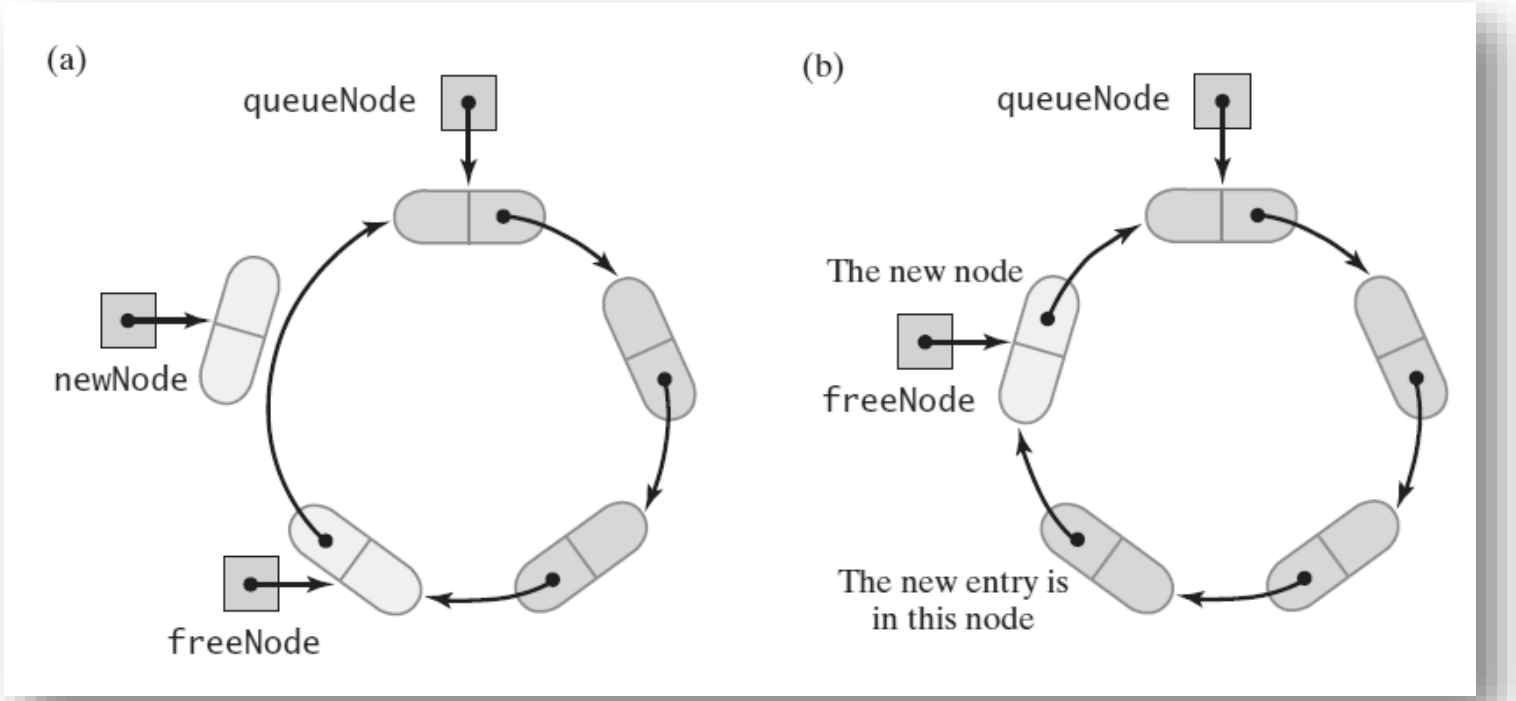
A Two-Part Circular Linked Chain (cont.)



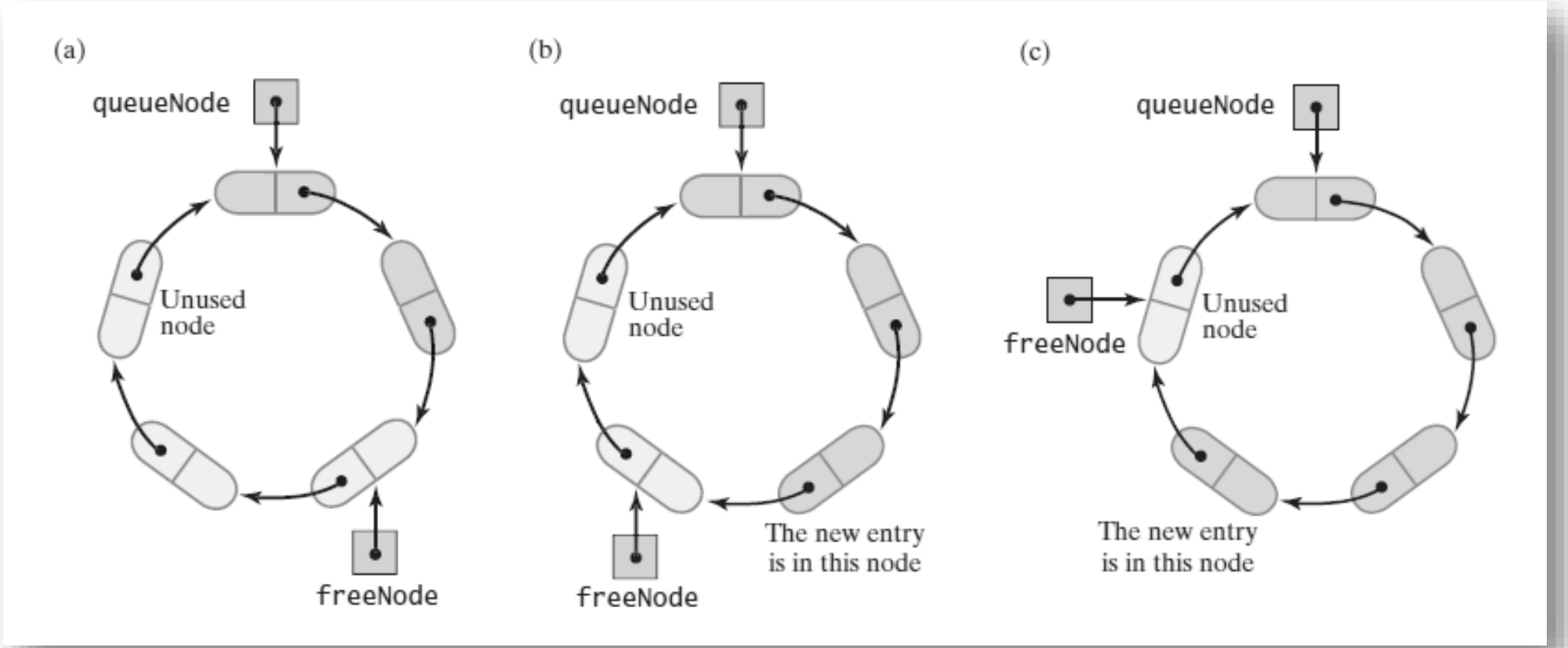
```java
public final class TwoPartCircularLinkedQueue<T> implements QueueInterface<T>
{
 private Node queueNode; // References first node in queue
 private Node freeNode;  // References node after back of queue

 public TwoPartCircularLinkedQueue()
 {
  freeNode = new Node(null, null);
  freeNode.setNextNode(freeNode);
  queueNode = freeNode;
 } // end default constructor

 //  < Implementations of the queue operations go here. >
 //  . . .
```

```
  private class Node
  {
   private T    data;  // Queue entry
   private Node next;  // Link to next node

   private Node(T dataPortion)
   {
    data = dataPortion;
    next = null;
   } // end constructor

   private Node(T dataPortion, Node linkPortion)
   {
    data = dataPortion;
    next = linkPortion;
   } // end constructor

   private T getData() { return data; } // end getData

   private void setData(T newData) { data = newData; } // end setData

   private Node getNextNode() { return next; } // end getNextNode

   private void setNextNode(Node nextNode)
   { next = nextNode; } // end setNextNode
  } // end Node
} // end TwoPartCircularLinkedQueue
```

Adding to the back



Adding to the back (cont.)



Adding to the back (cont.)

```
public void enqueue(T newEntry)
{
 freeNode.setData(newEntry);

 if (isChainFull())
 {
  // Allocate a new node and insert it after the node that
  // freeNode references
  Node newNode = new Node(null, freeNode.getNextNode());
  freeNode.setNextNode(newNode);
 } // end if

 freeNode = freeNode.getNextNode();
} // end enqueue
```

```
private boolean isChainFull()
{
 return queueNode == freeNode.getNextNode();
} // end isChainFull
```

Retrieving the front

```
public T getFront()
{
 if (isEmpty())
  throw new EmptyQueueException();
 else
  return queueNode.getData();
} // end getFront
```

Removing the front

```
public T dequeue()
{
 T front = getFront();  // Might throw EmptyQueueException
 assert !isEmpty();

 queueNode.setData(null);
 queueNode = queueNode.getNextNode();

 return front;
} // end dequeue
```

The rest of the class

```
public boolean isEmpty()
{
 return queueNode == freeNode;
} // end isEmpty
```

Java Class Library:
The Class **AbstractQueue**

```
public boolean add(T newEntry)
public boolean offer(T newEntry)
public T remove()
public T poll()
public T element()
public T peek()
public boolean isEmpty()
public void clear()
public int size()
```

References

- F. M. Carrano and T. M. Henry, “Data Structures and Abstractions”, 4th edition. Pearson Education.