Queue, Deque, and Priority Queue
Linked List Implementations

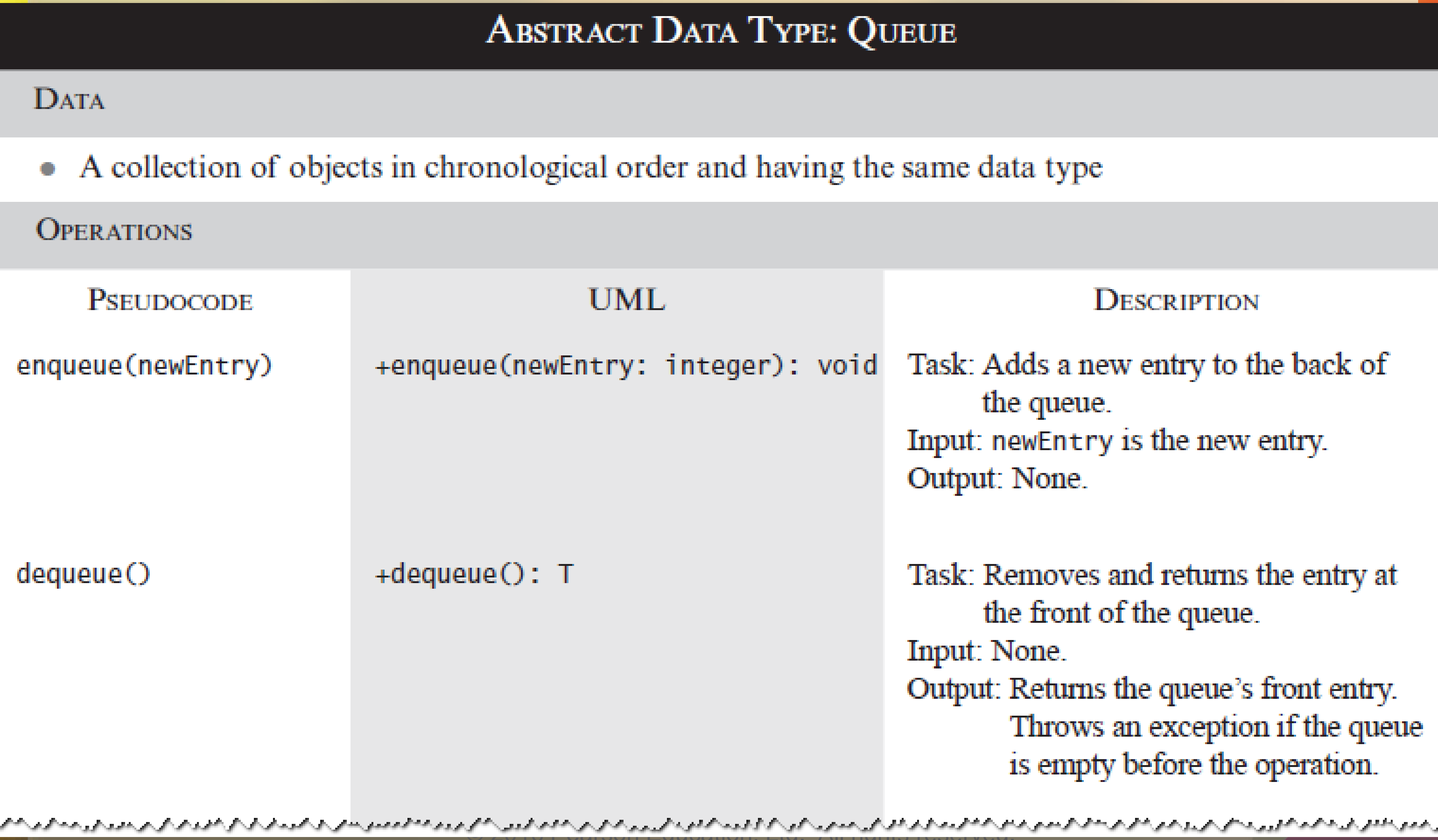
Chapter 11

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

The ADT Queue

- Terminology
  - Item added first, or earliest, is at the front of the queue
  - Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back
- Client can look at or remove only the entry at the front of the queue

The ADT Queue



Abstract Data Type: Queue

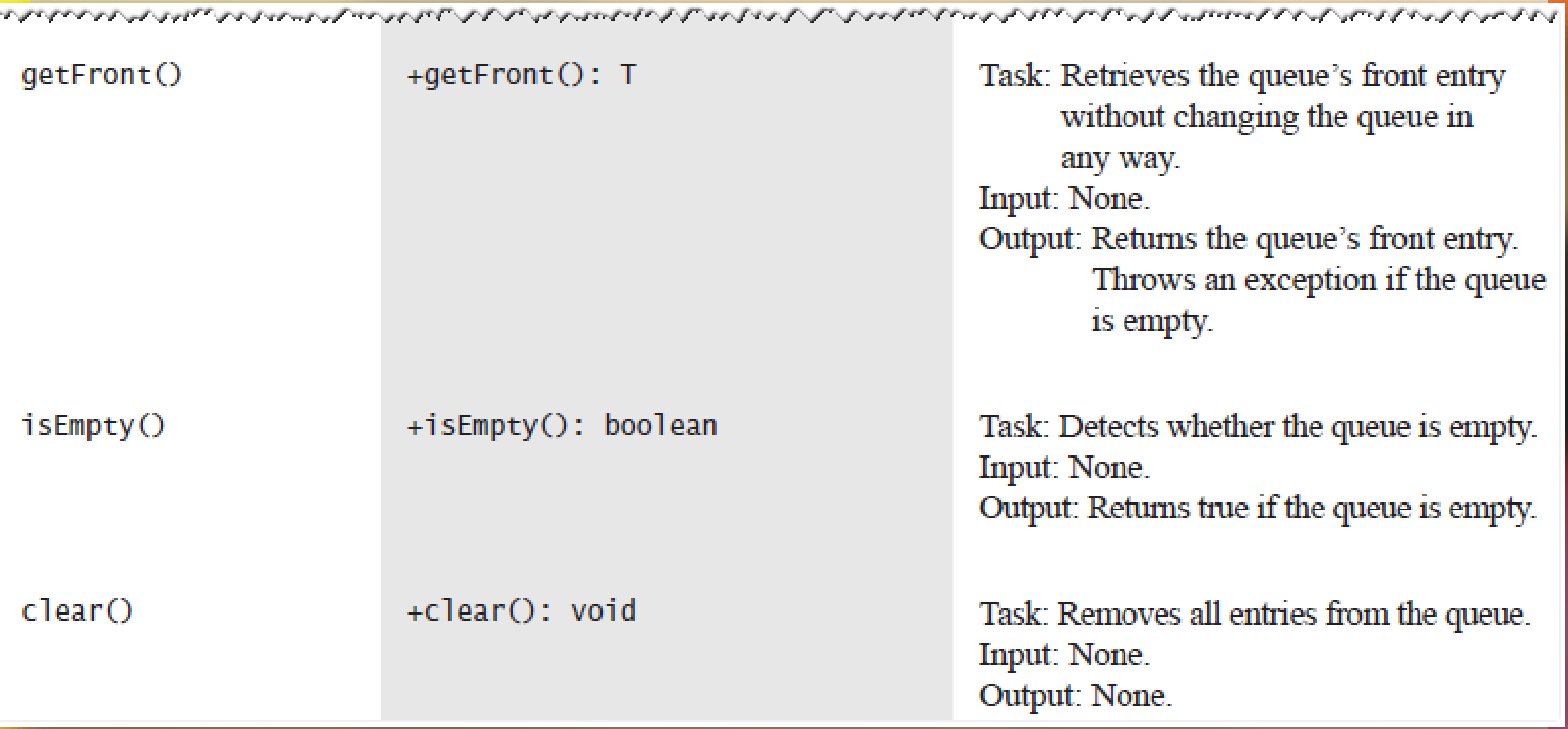
Data

- A collection of objects in chronological order and having the same data type

Operations

Pseudocode	UML	Description
`enqueue(newEntry)`	`+enqueue(newEntry: integer): void`	Task: Adds a new entry to the back of the queue. Input: `newEntry` is the new entry. Output: None.
`dequeue()`	`+dequeue(): T`	Task: Removes and returns the entry at the front of the queue. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty before the operation.

The ADT Queue



`getFront()`	`+getFront(): T`	Task: Retrieves the queue's front entry without changing the queue in any way. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty.
`isEmpty()`	`+isEmpty(): boolean`	Task: Detects whether the queue is empty. Input: None. Output: Returns true if the queue is empty.
`clear()`	`+clear(): void`	Task: Removes all entries from the queue. Input: None. Output: None.

The ADT Queue

```java
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
```

LISTING 10-1 An interface for the ADT queue

Linked Implementation of a Queue

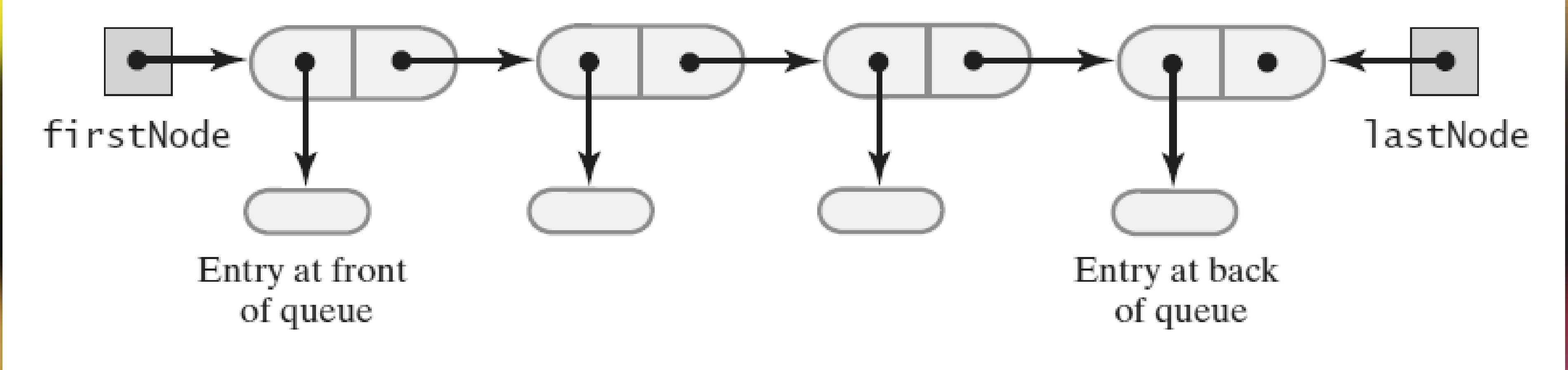
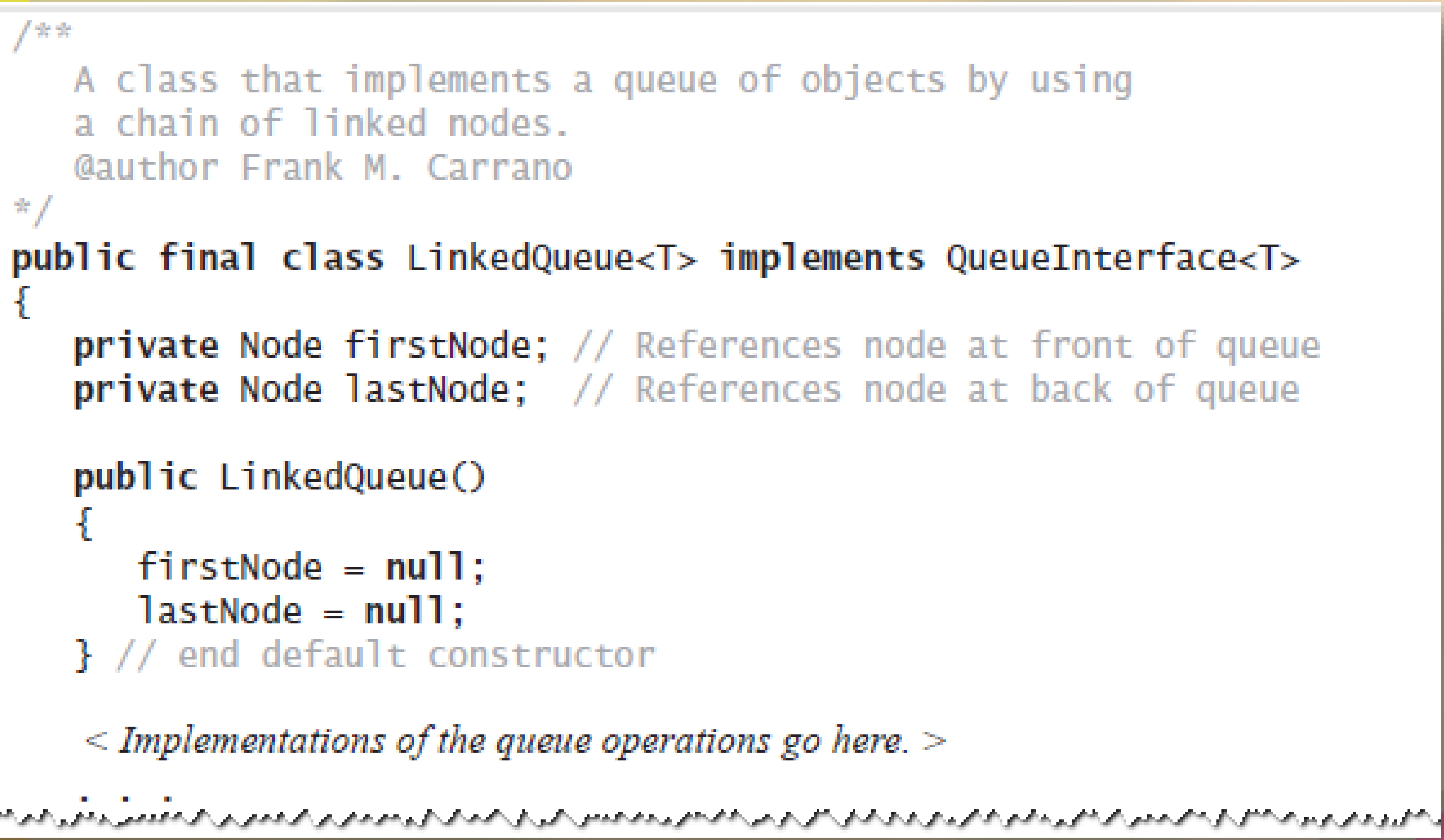


FIGURE 11-1 A chain of linked nodes that implements a queue

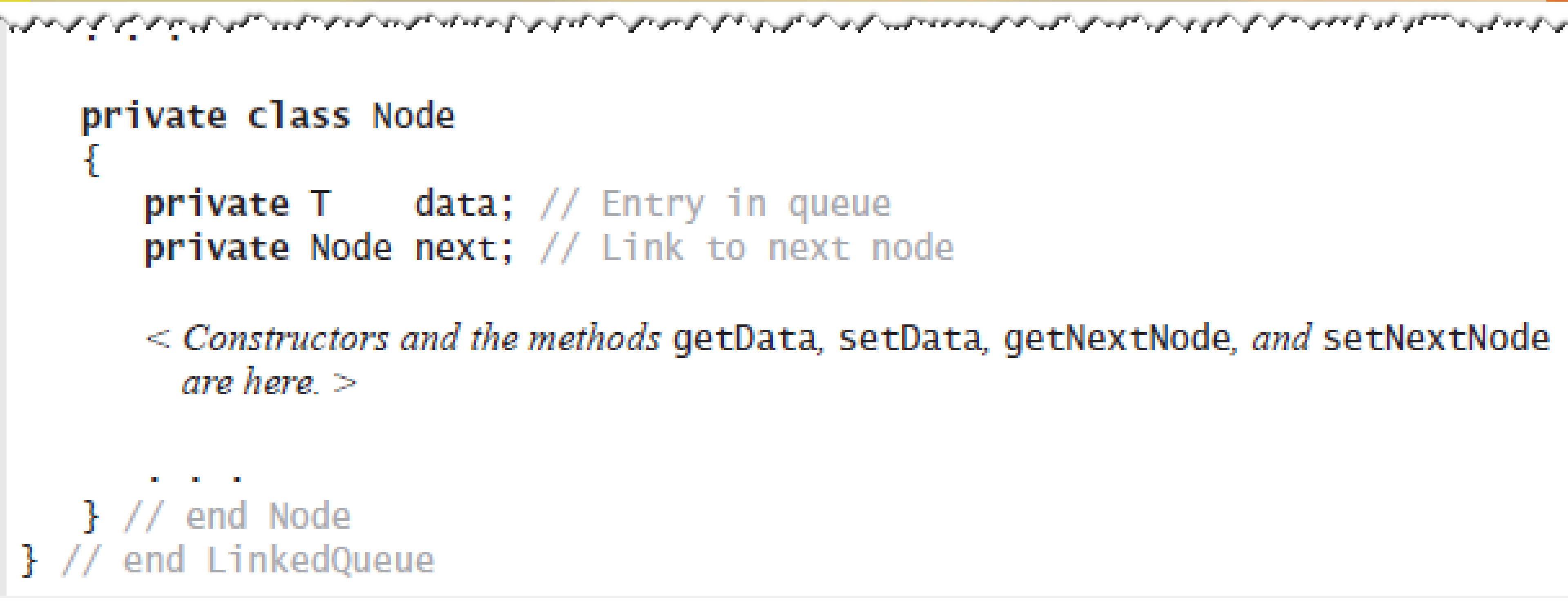
Linked Implementation of a Queue



```
/**
   A class that implements a queue of objects by using
   a chain of linked nodes.
   @author Frank M. Carrano
*/
public final class LinkedQueue<T> implements QueueInterface<T>
{
   private Node firstNode; // References node at front of queue
   private Node lastNode;  // References node at back of queue

   public LinkedQueue()
   {
      firstNode = null;
      lastNode = null;
   } // end default constructor

   < Implementations of the queue operations go here. >

   . . .
```

LISTING 11-1 An outline of a linked implementation of the ADT queue

Linked Implementation of a Queue



```
   . . .

   private class Node
   {
      private T    data; // Entry in queue
      private Node next; // Link to next node

      < Constructors and the methods getData, setData, getNextNode, and setNextNode
        are here. >


      . . .
   } // end Node
} // end LinkedQueue
```

LISTING 11-1 An outline of a linked implementation of the ADT queue

Linked Implementation of a Queue

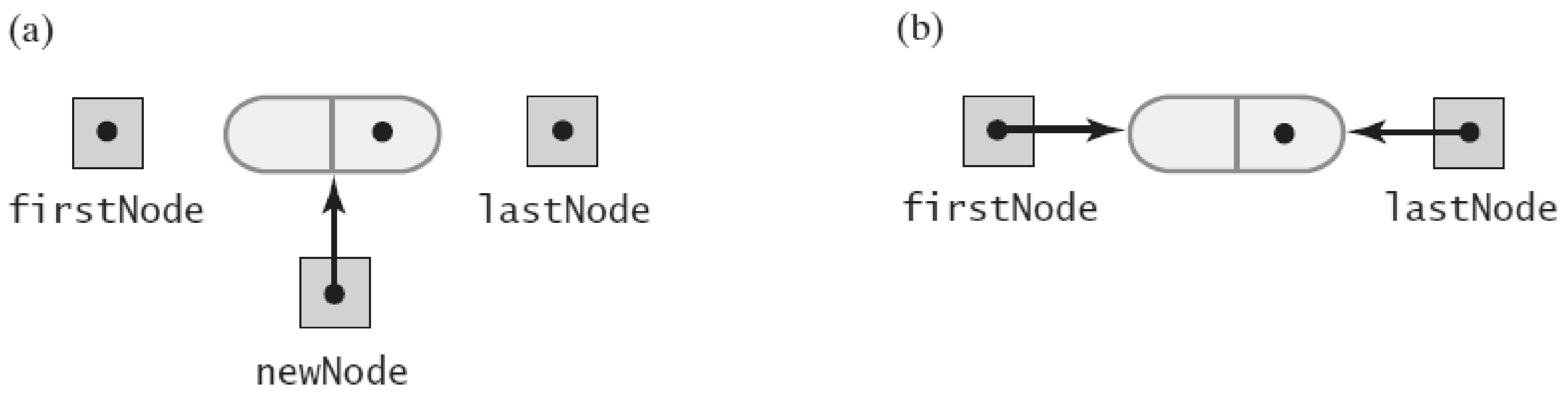


FIGURE 11-2 (a) Before adding a new node to an empty chain; (b) after adding it

Linked Implementation of a Queue

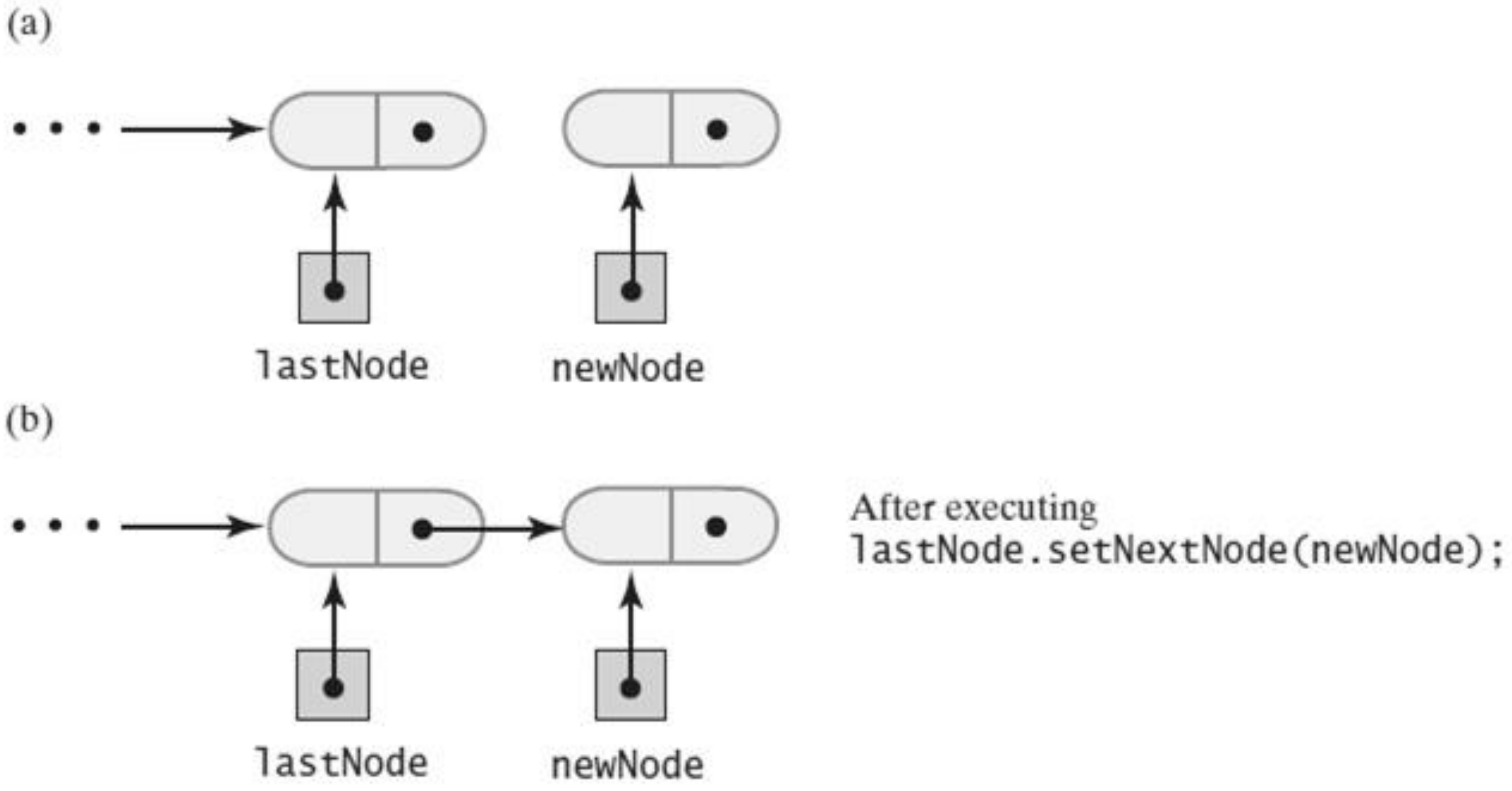


FIGURE 11-3 (a) Before, and (b) during adding a new node to the end of a nonempty chain that has a tail reference

Linked Implementation of a Queue

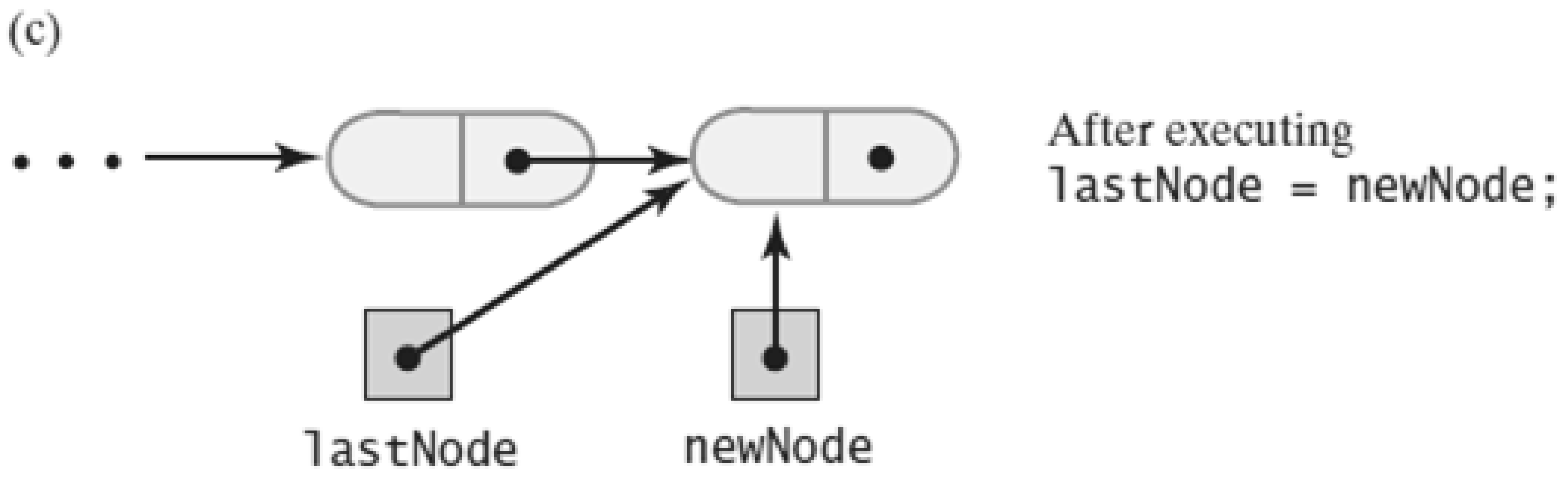
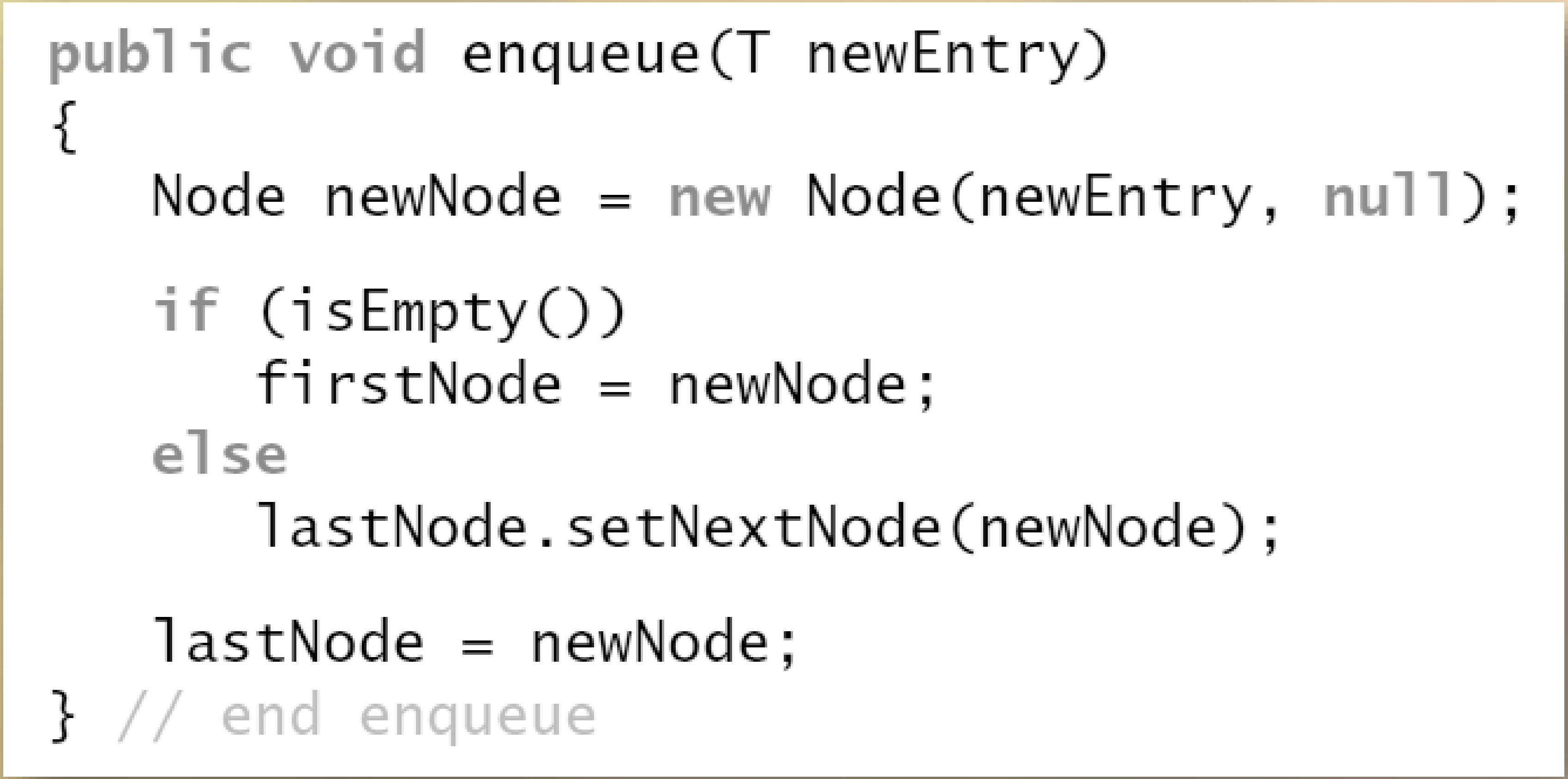


FIGURE 11-3 (c) After adding a new node to the end of a nonempty chain that has a tail reference

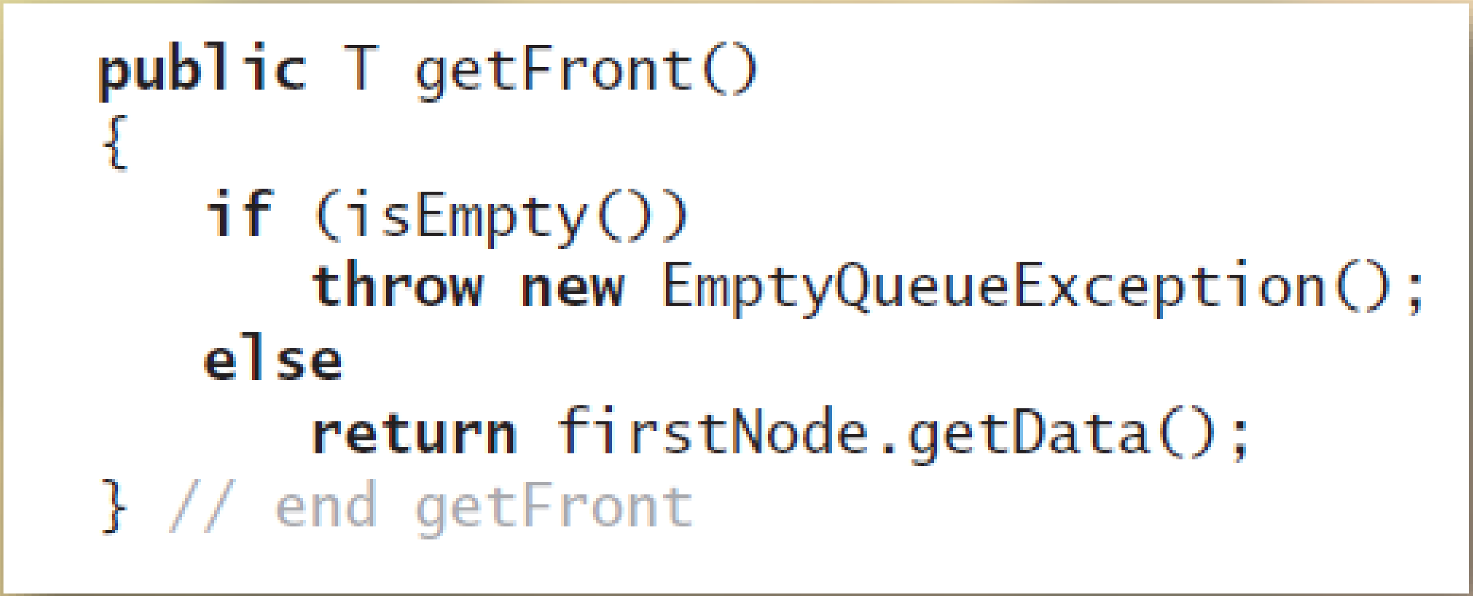
Linked Implementation of a Queue



```java
public void enqueue(T newEntry)
{
   Node newNode = new Node(newEntry, null);

   if (isEmpty())
      firstNode = newNode;
   else
      lastNode.setNextNode(newNode);

   lastNode = newNode;
} // end enqueue
```

The definition of **`enqueue`**

Linked Implementation of a Queue



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

Retrieving the front entry

Linked Implementation of a Queue

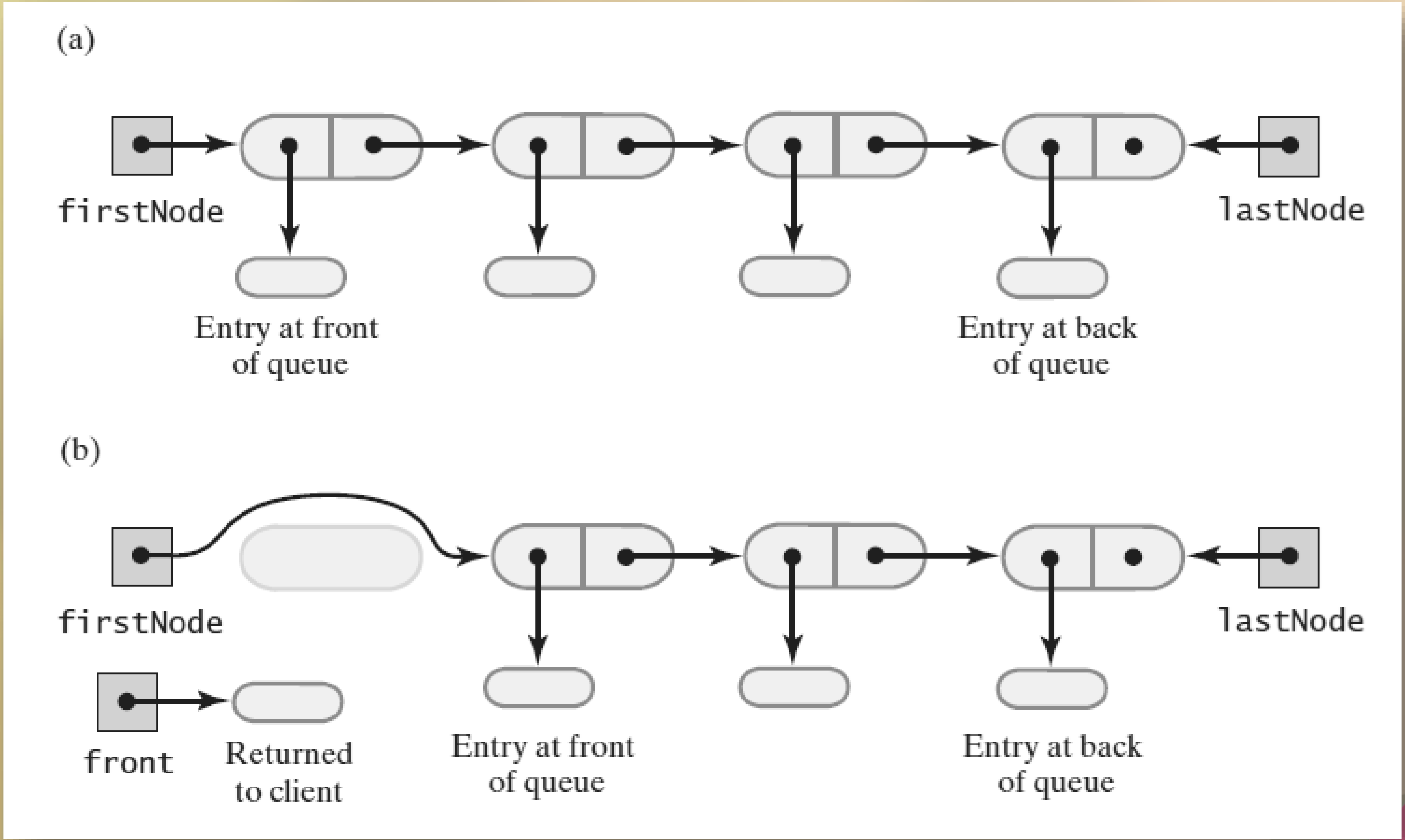


FIGURE 11-4 (a) A queue of more than one entry; (b) after removing the entry at the front of the queue

Linked Implementation of a Queue

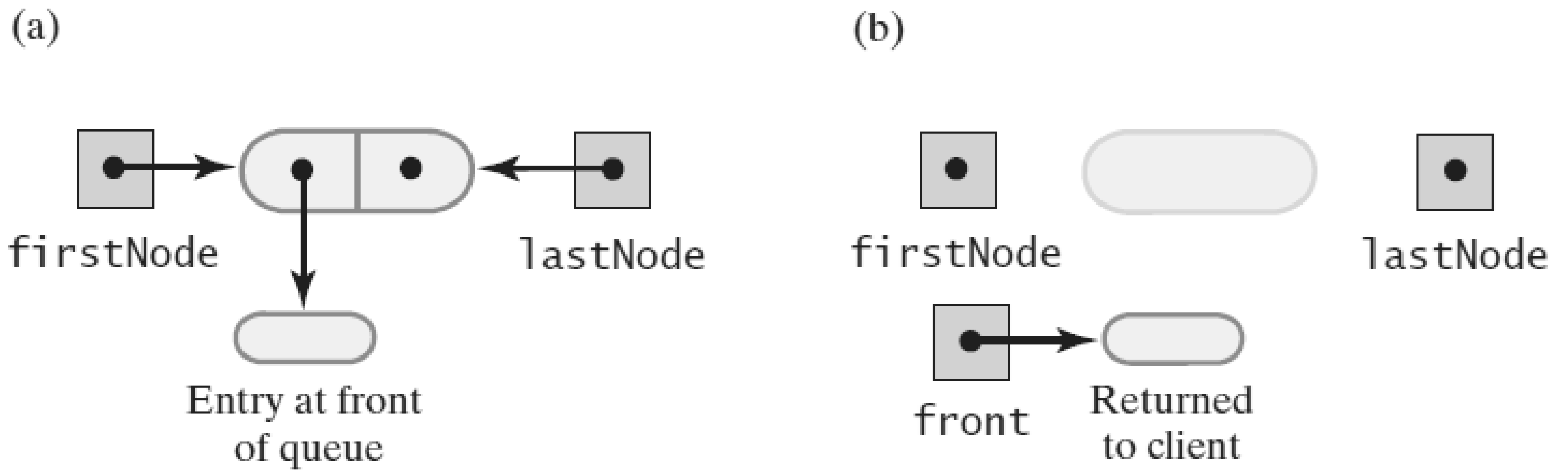


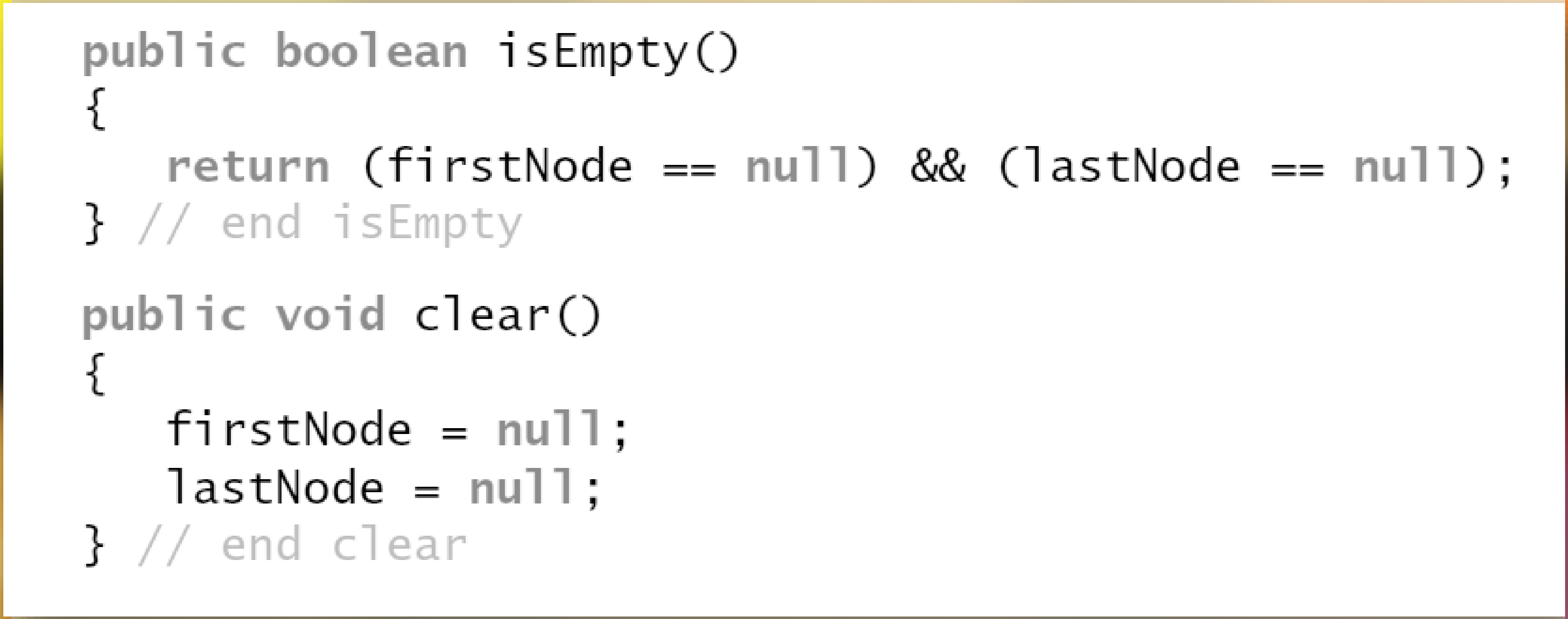
FIGURE 11-5 (a) A queue of one entry; (b) after removing the entry at the front of the queue

Linked Implementation of a Queue

```java
public T dequeue()
{
   T front = getFront(); // Might throw EmptyQueueException
   assert firstNode != null;
   firstNode.setData(null);
   firstNode = firstNode.getNextNode();

   if (firstNode == null)
      lastNode = null;

   return front;
} // end dequeue
```

Removing the front entry

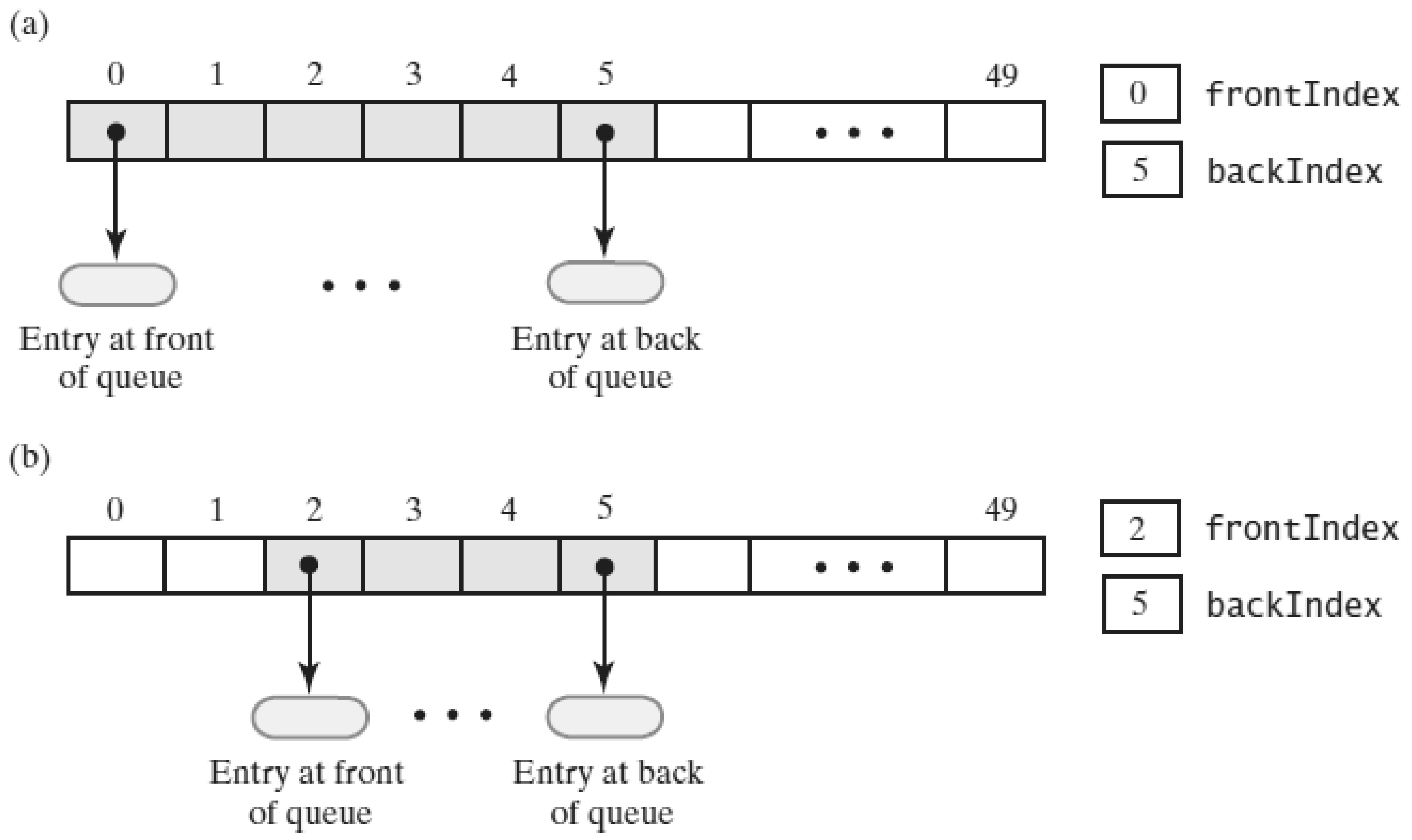
Linked Implementation of a Queue



```
public boolean isEmpty()
{
   return (firstNode == null) && (lastNode == null);
} // end isEmpty

public void clear()
{
   firstNode = null;
   lastNode = null;
} // end clear
```

Public methods `isEmpty` and `clear`

Array-Based Implementation of a Queue: Circular Array



(b) after removing the entry at the front twice;

Circular Array with One Unused Location

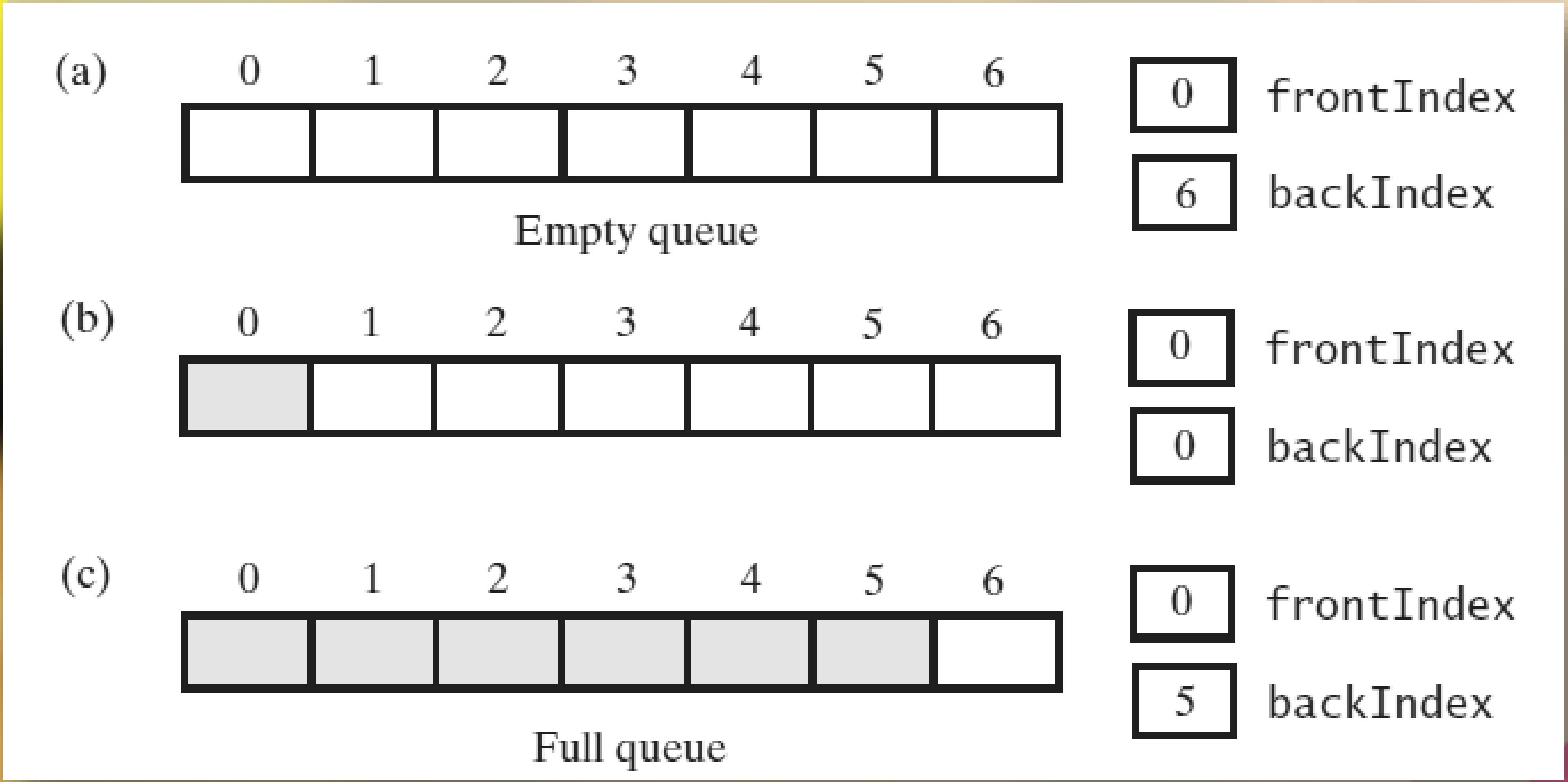


FIGURE 11-8 A seven-location circular array that contains at most six entries of a queue

Circular Linked Implementations of a Queue

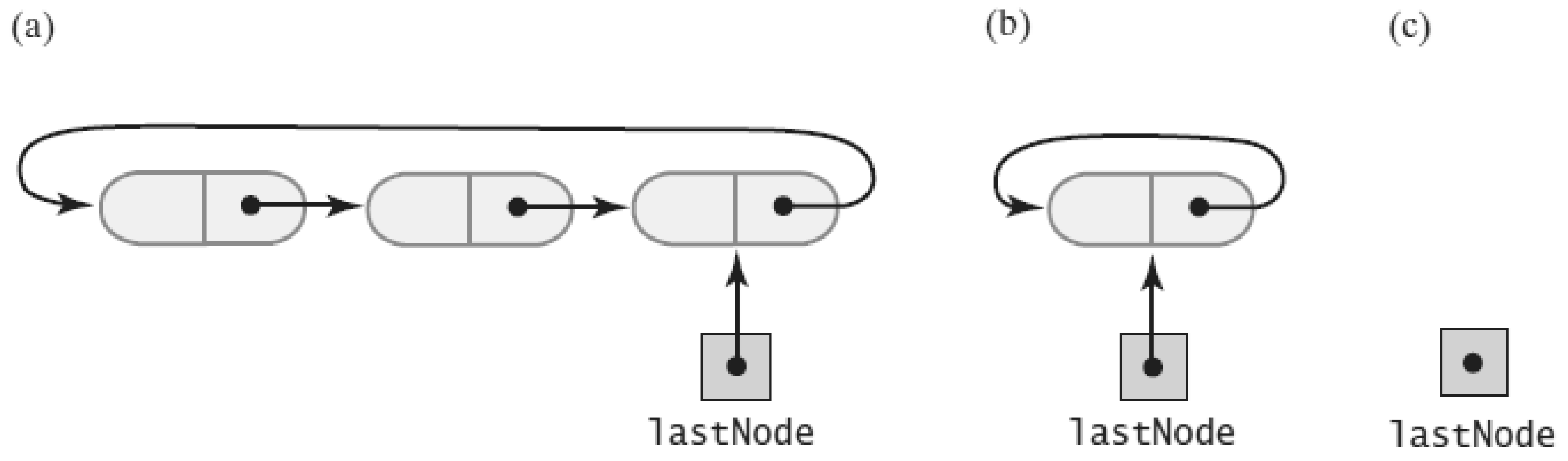


FIGURE 11-11 A circular linked chain with an external reference to its last node that (a) has more than one node; (b) has one node; (c) is empty

Two-Part Circular Linked Chain

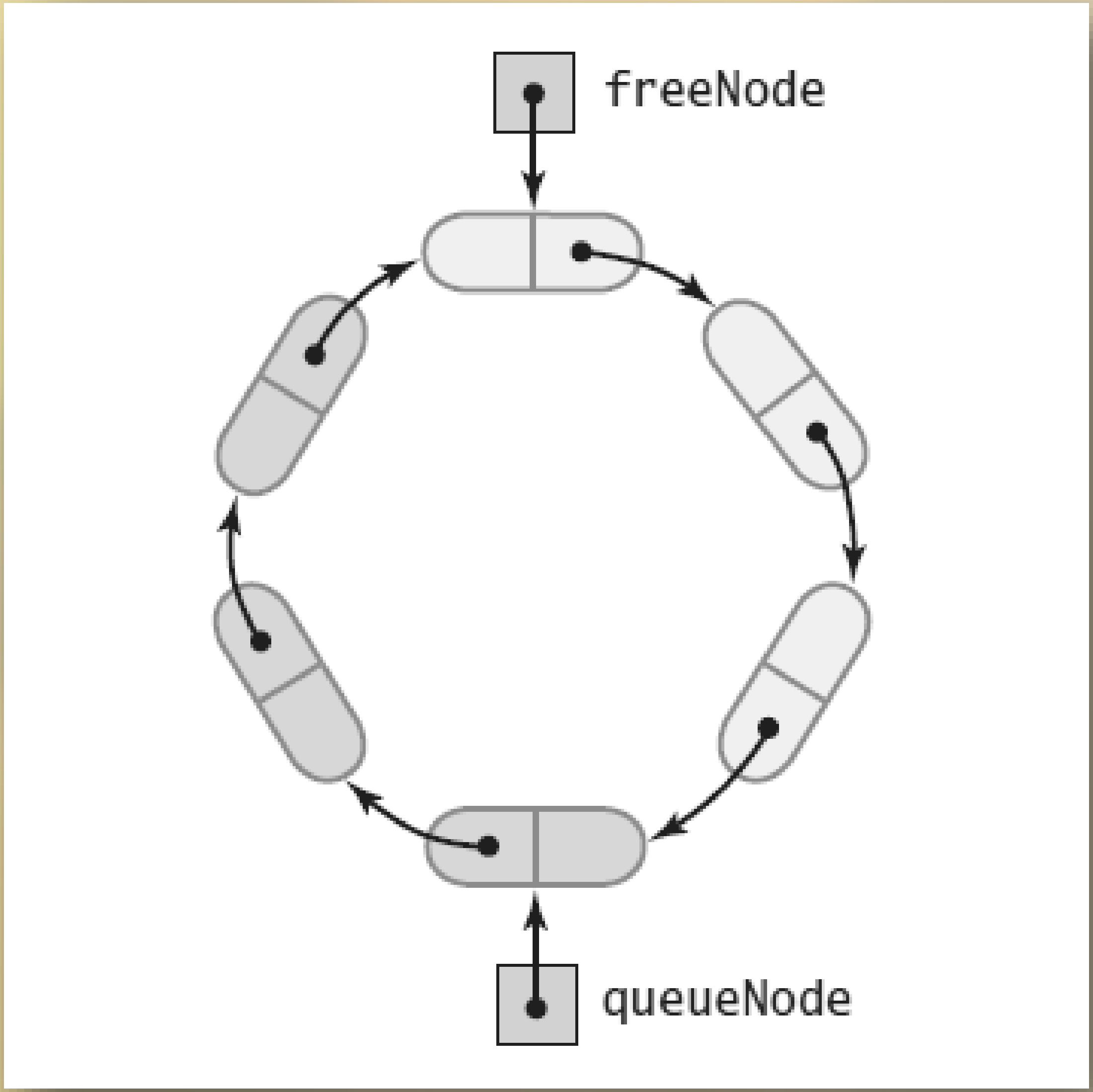


FIGURE 11-12 A two-part circular linked chain that represents both a queue and the nodes available to the queue

Two-Part Circular Linked Chain

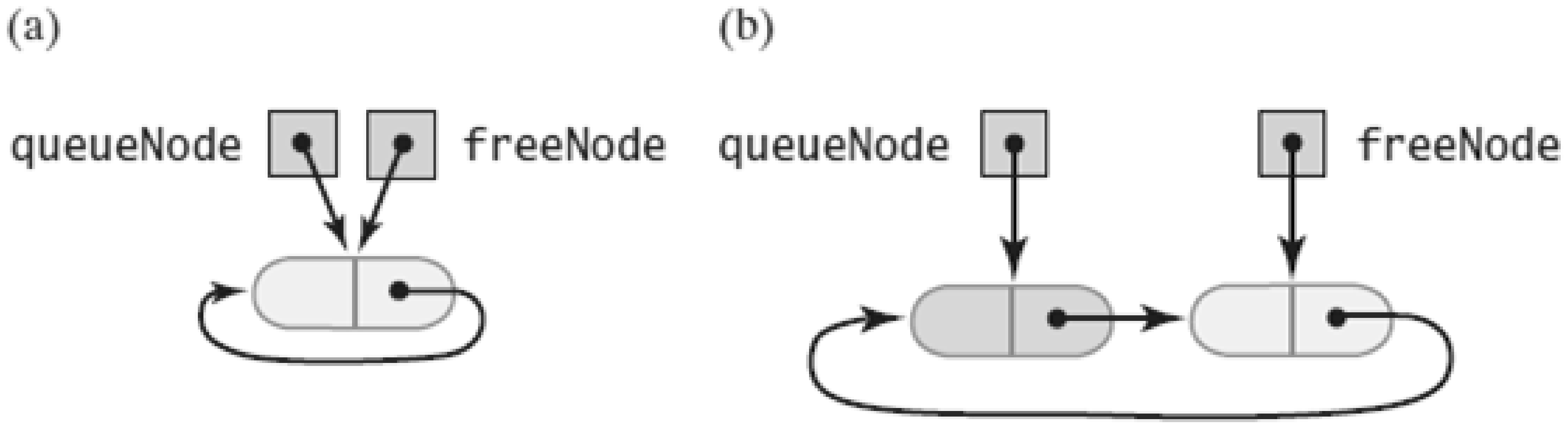


FIGURE 11-14 A two-part circular linked chain that represents a queue: (a) when it is empty; (b) after adding one entry;

Two-Part Circular Linked Chain

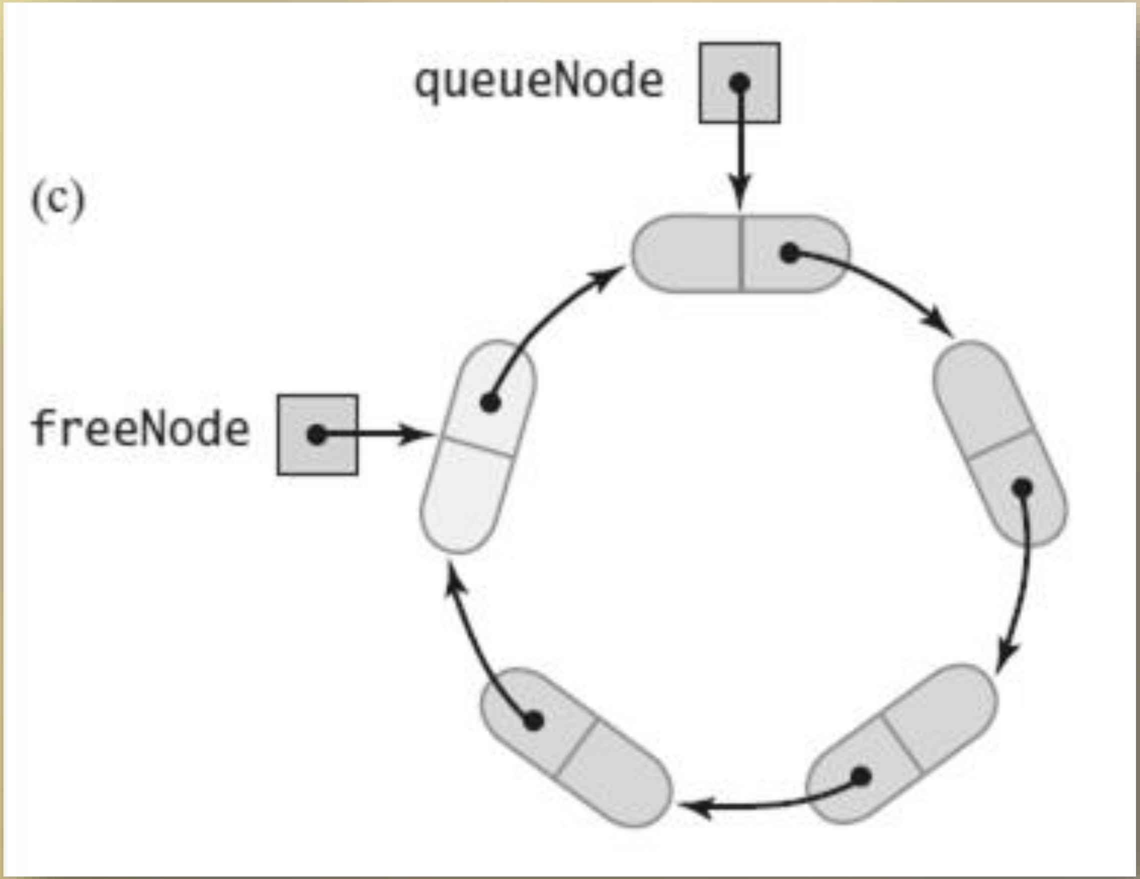


FIGURE 11-14 A two-part circular linked chain that represents a queue: (c) after adding three more entries

Two-Part Circular Linked Chain

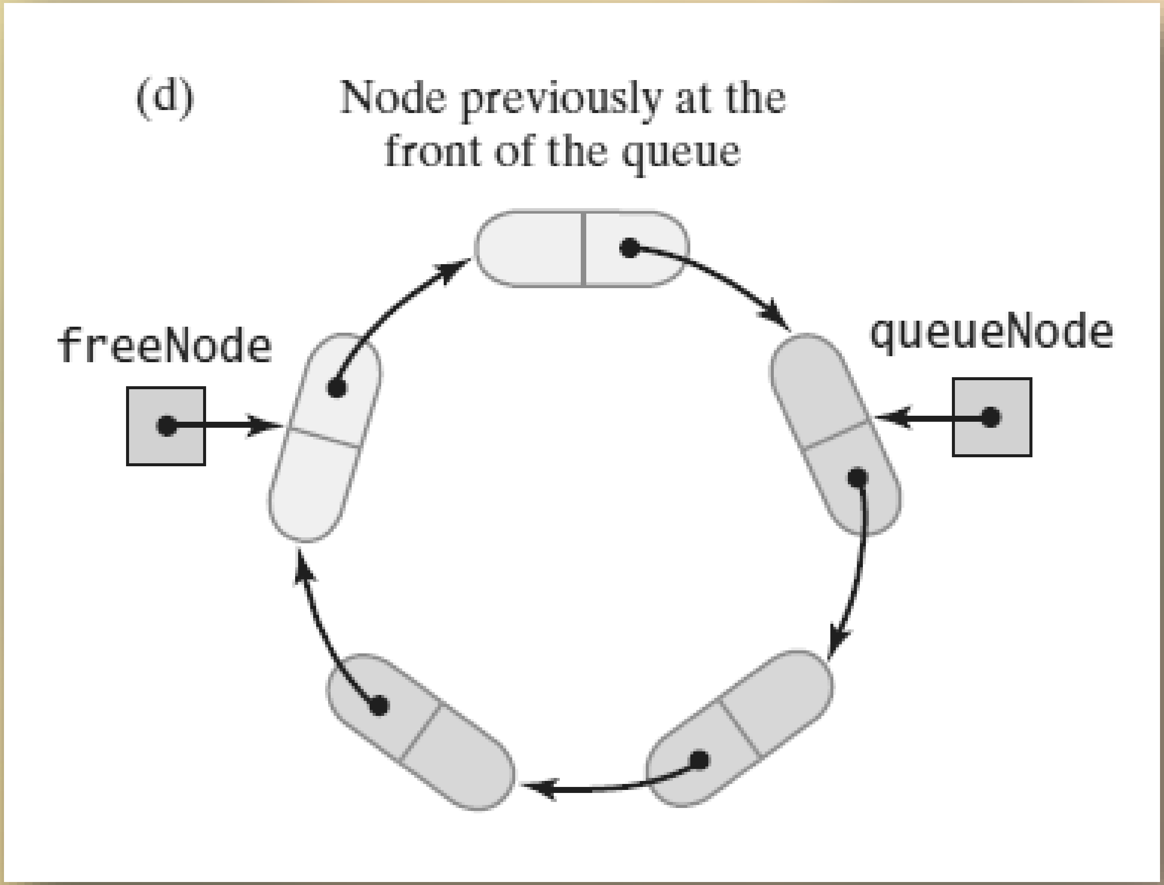


FIGURE 11-14 A two-part circular linked chain that represents a queue: (d) after removing the front entry;

Two-Part Circular Linked Chain

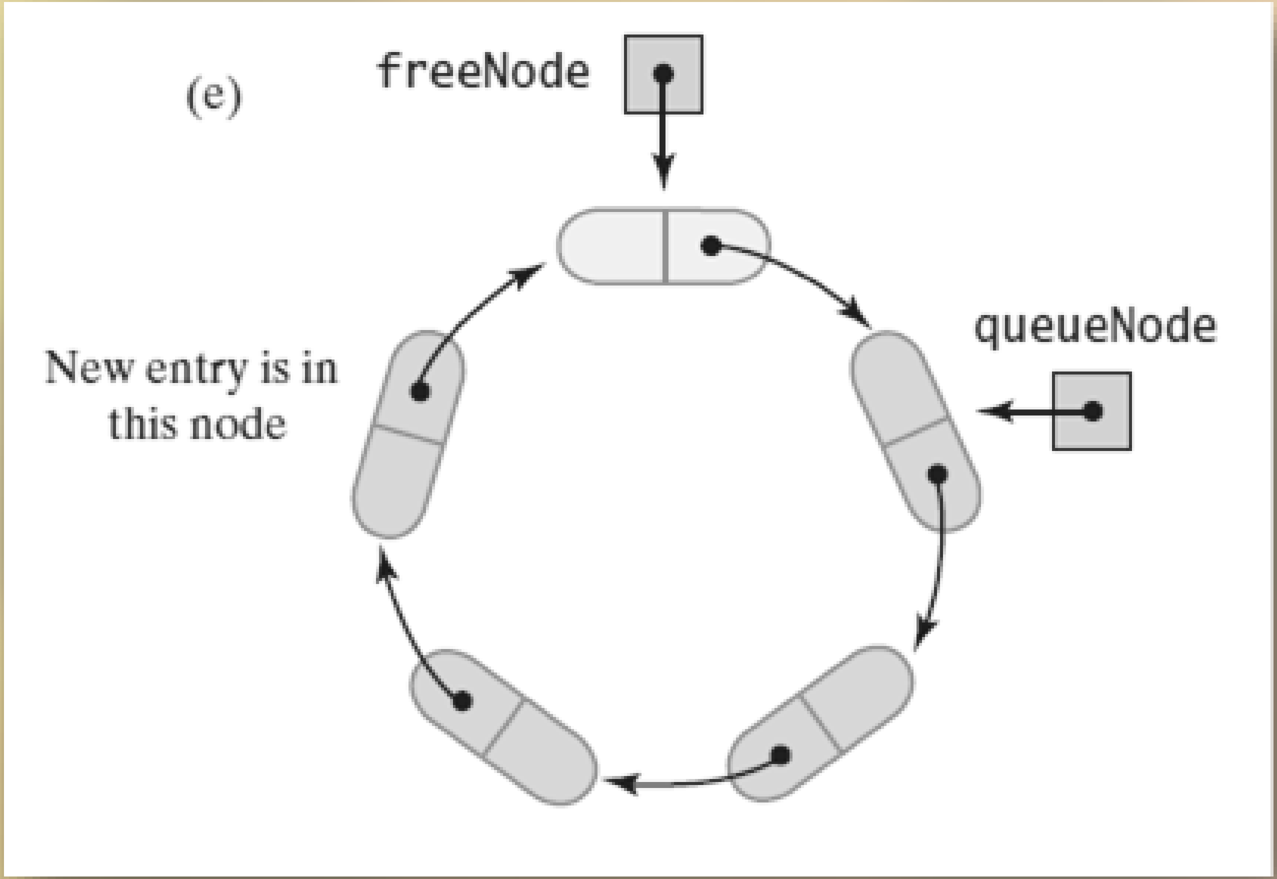
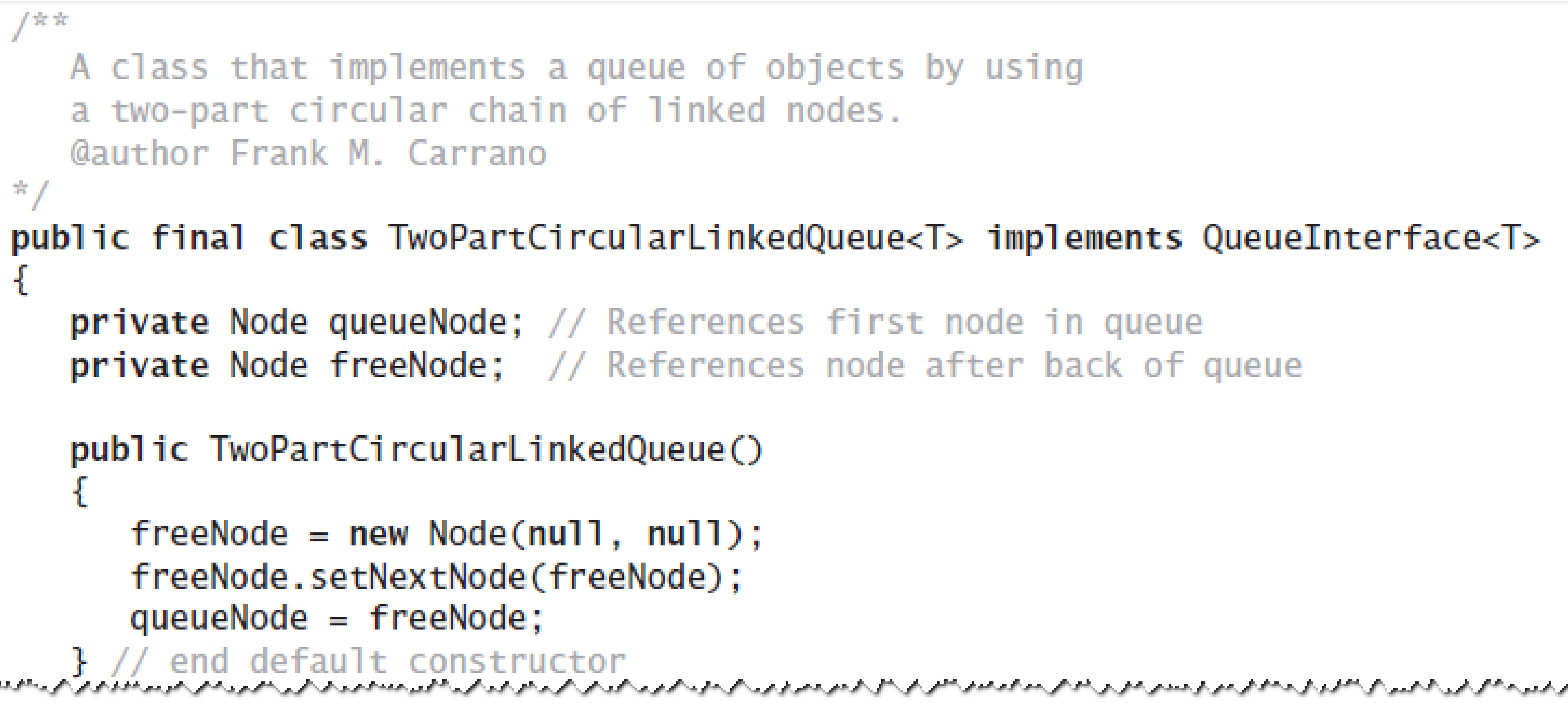


FIGURE 11-14 A two-part circular linked chain that represents a queue: (e) after adding one more entry

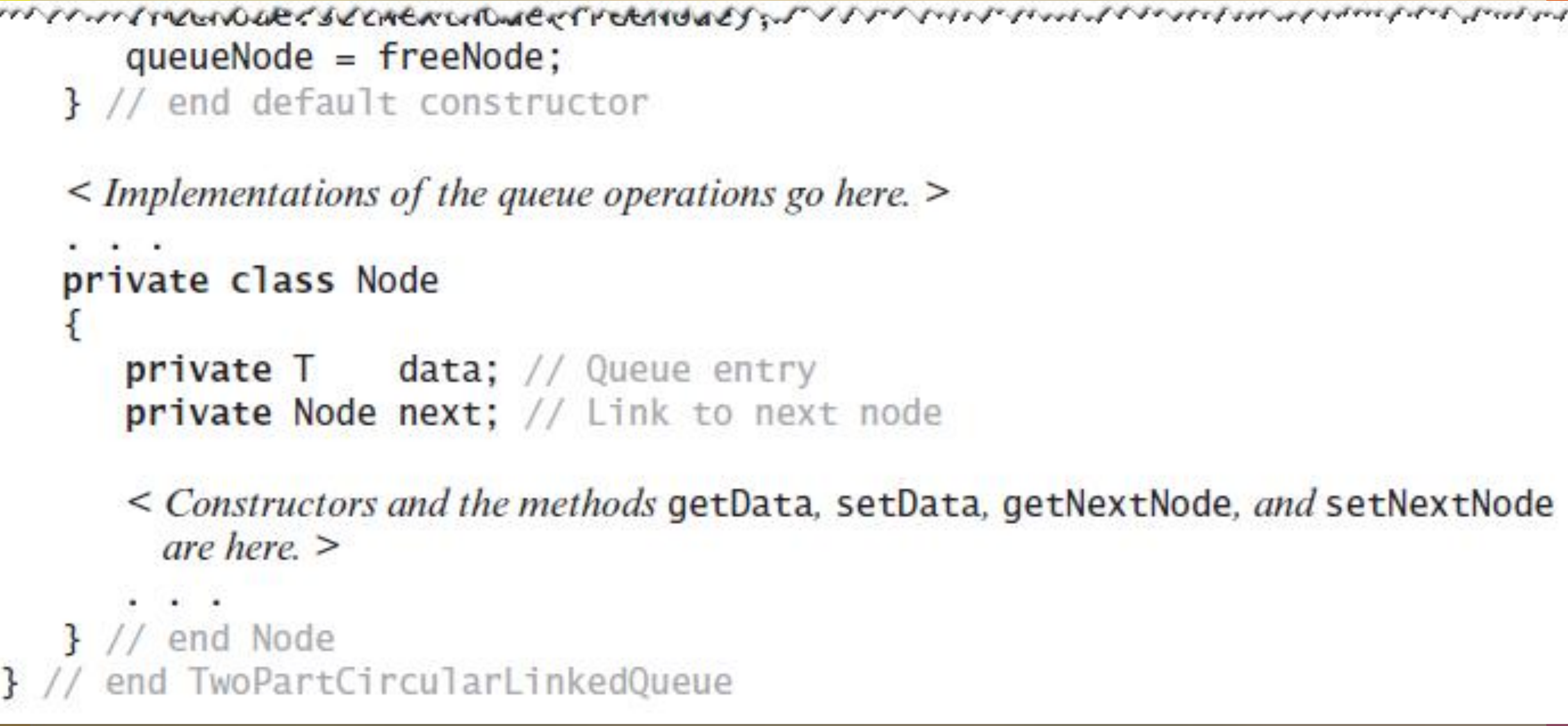
Two-Part Circular Linked Chain



```
/**
   A class that implements a queue of objects by using
   a two-part circular chain of linked nodes.
   @author Frank M. Carrano
*/
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
```

LISTING 11-3 An outline of a two-part circular linked implementation of the ADT queue

Two-Part Circular Linked Chain



```
      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

      < Constructors and the methods getData, setData, getNextNode, and setNextNode
        are here. >
      . . .
   } // end Node
} // end TwoPartCircularLinkedQueue
```

LISTING 11-3 An outline of a two-part circular linked implementation of the ADT queue

Two-Part Circular Linked Chain

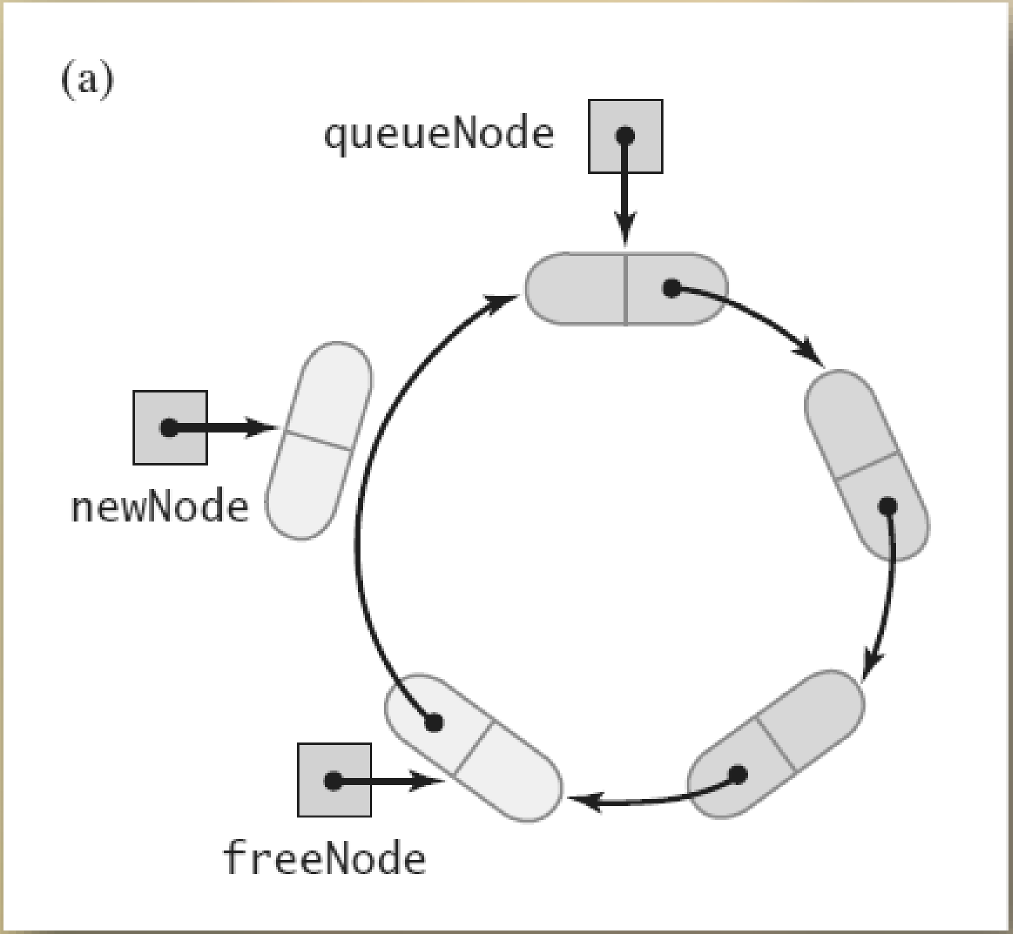


FIGURE 11-14 A chain that requires a new node for an addition to a queue: (a) before the addition;

Two-Part Circular Linked Chain

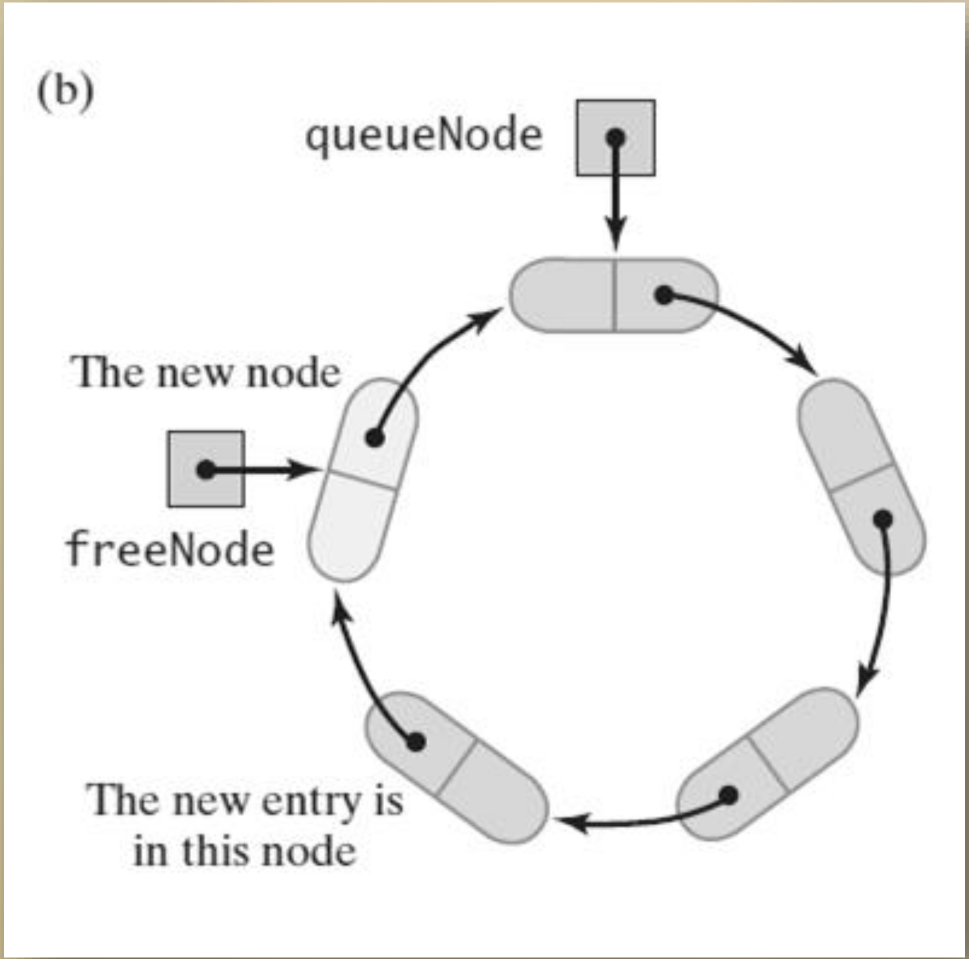


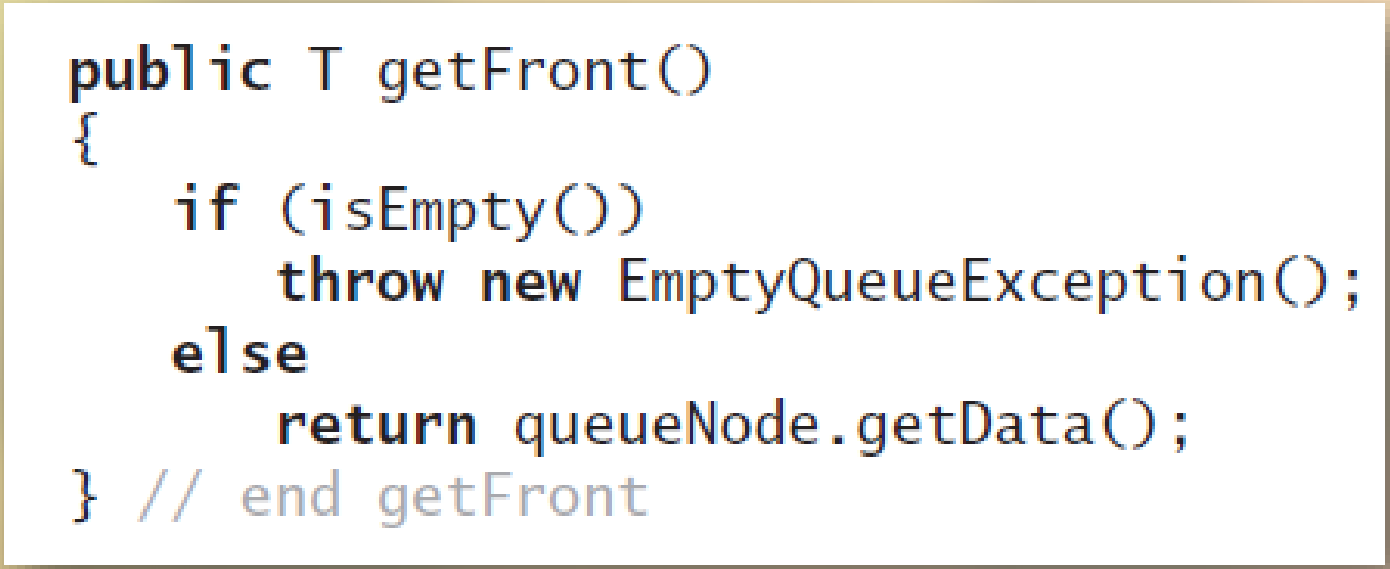
FIGURE 11-14 A chain that requires a new node for an addition to a queue: (b) after the addition

Two-Part Circular Linked Chain

```
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
```

Implementation of `enqueue` is an O(1) operation

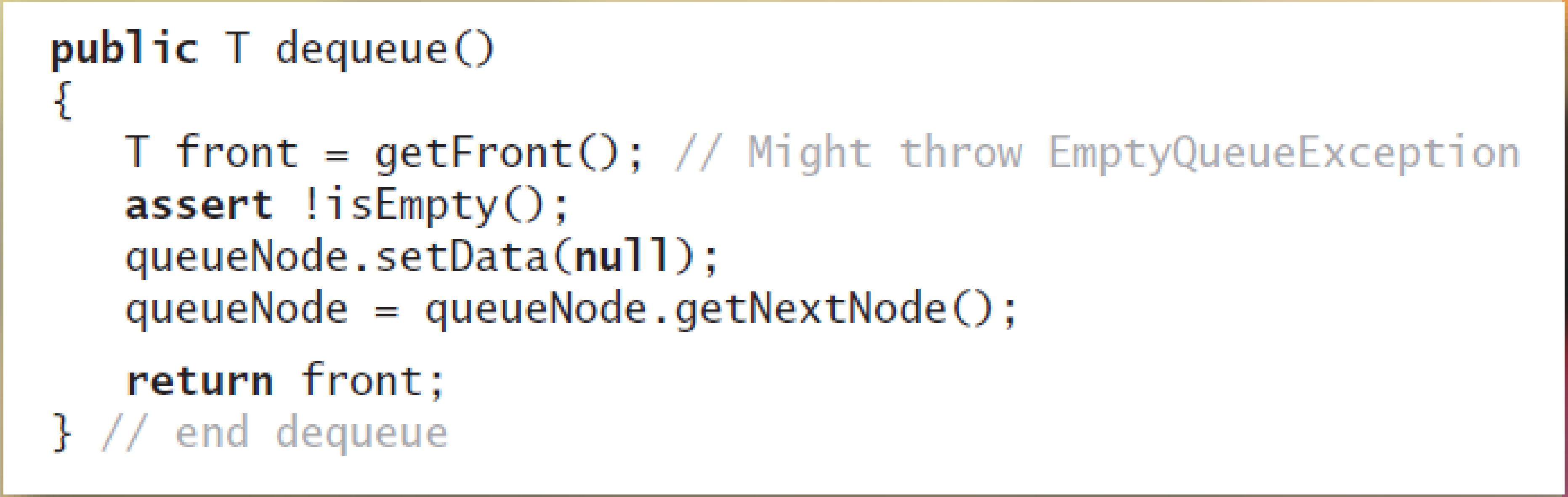
Two-Part Circular Linked Chain



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

Implementation of **`getFront`** is an O(1) operation

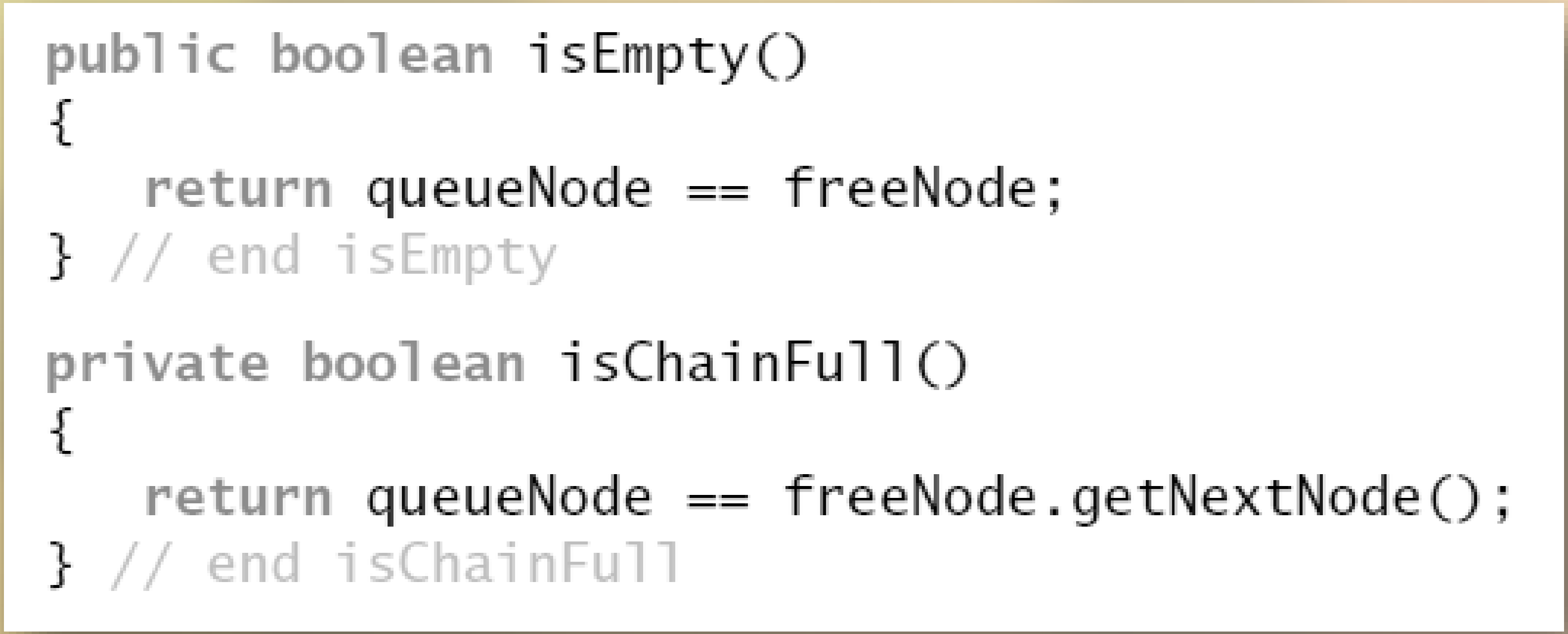
Two-Part Circular Linked Chain



```
public T dequeue()
{
   T front = getFront(); // Might throw EmptyQueueException
   assert !isEmpty();
   queueNode.setData(null);
   queueNode = queueNode.getNextNode();

   return front;
} // end dequeue
```

Implementation of `dequeue` is an O(1) operation

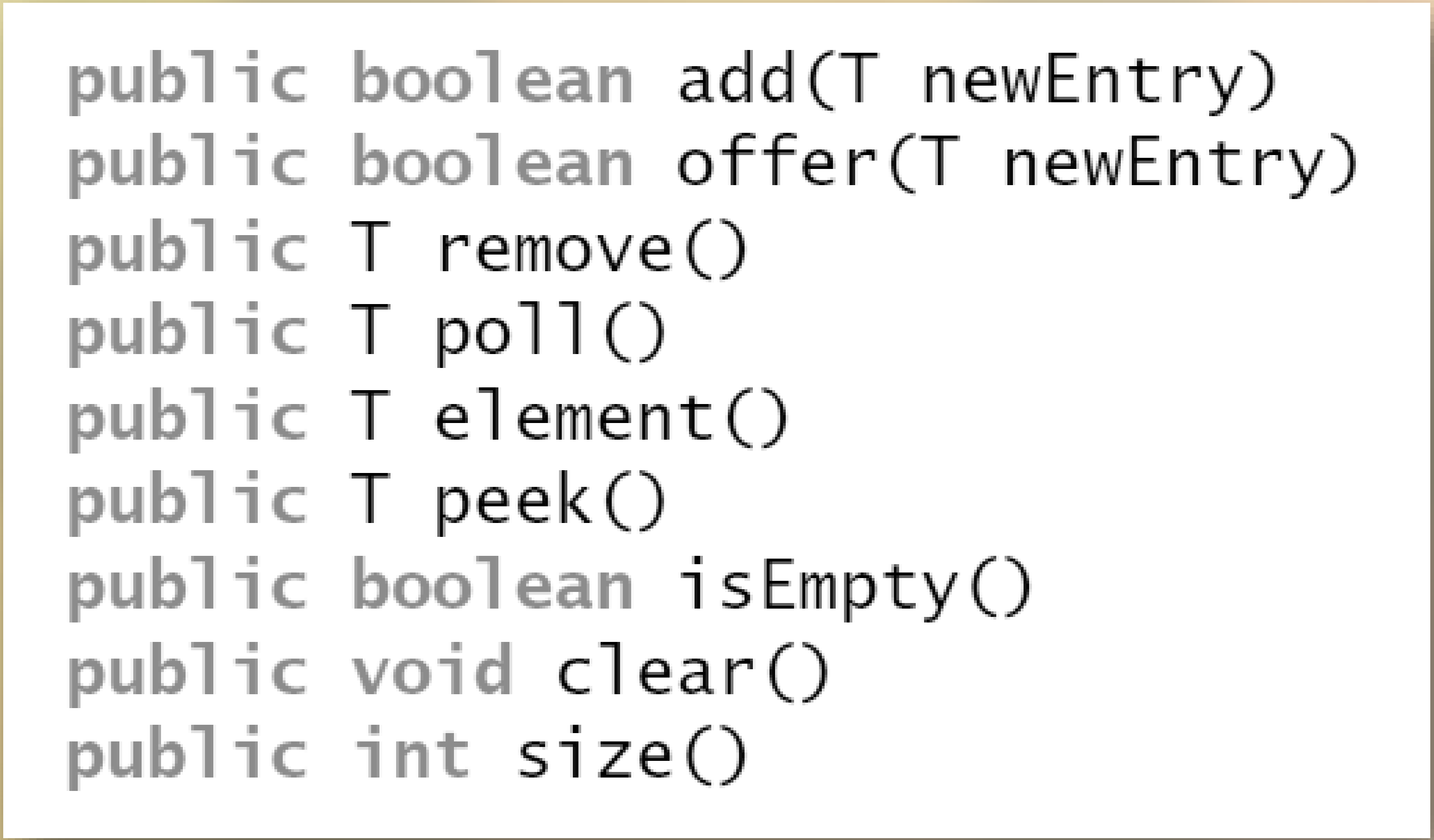
Two-Part Circular Linked Chain



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

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

Methods **`isEmpty`** an **`isChainFull`**

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()
```

Methods in this interface

The ADT Deque

- A double ended queue
- *Deque* pronounced “deck”
- Has both queuelike operations and stacklike operations

The ADT Deque

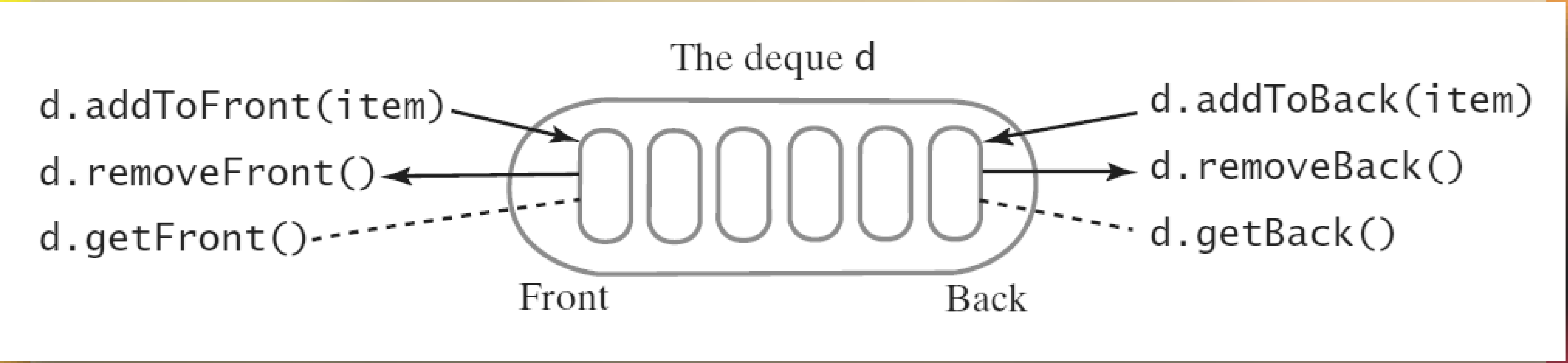
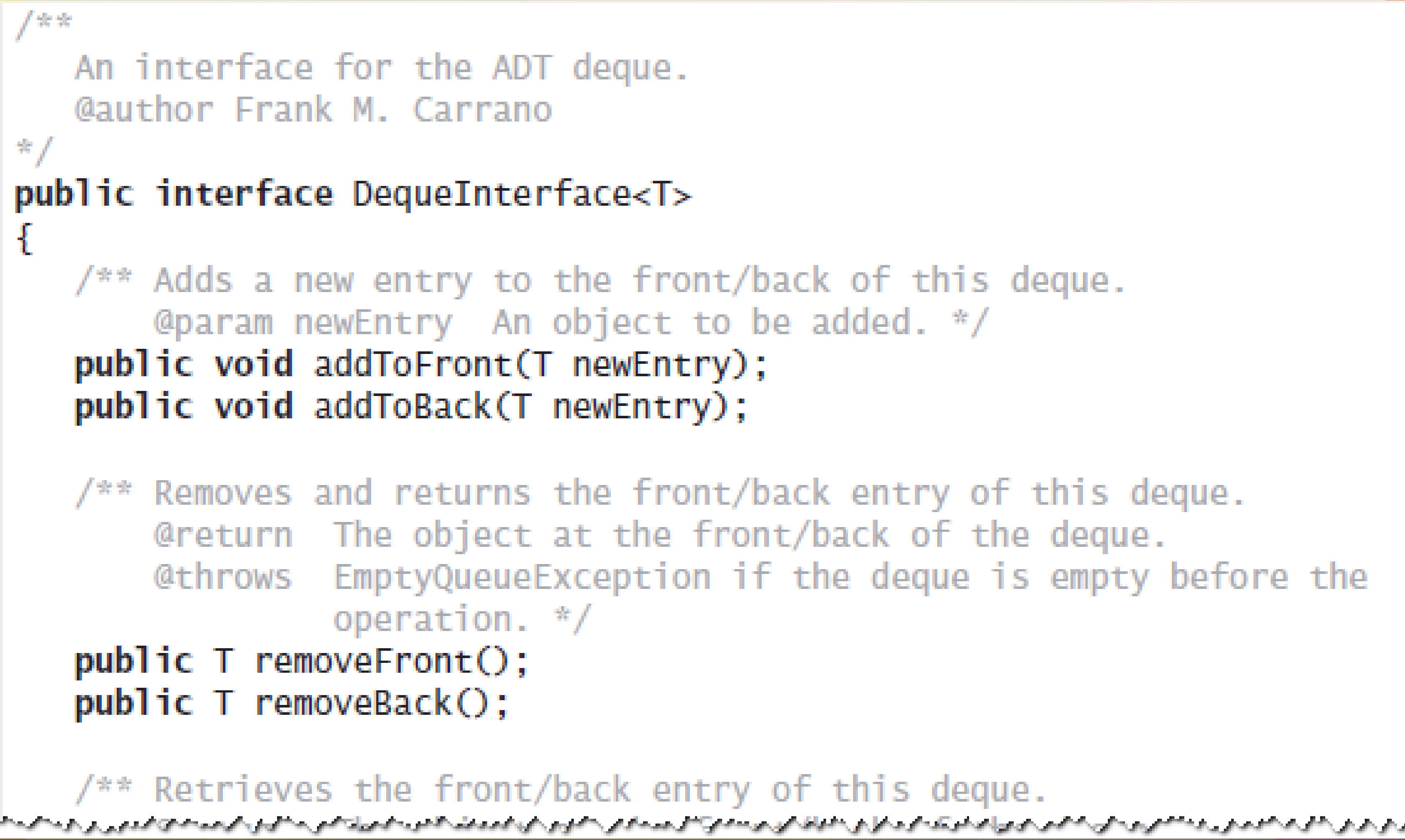


FIGURE 10-10 An instance *d* of a deque

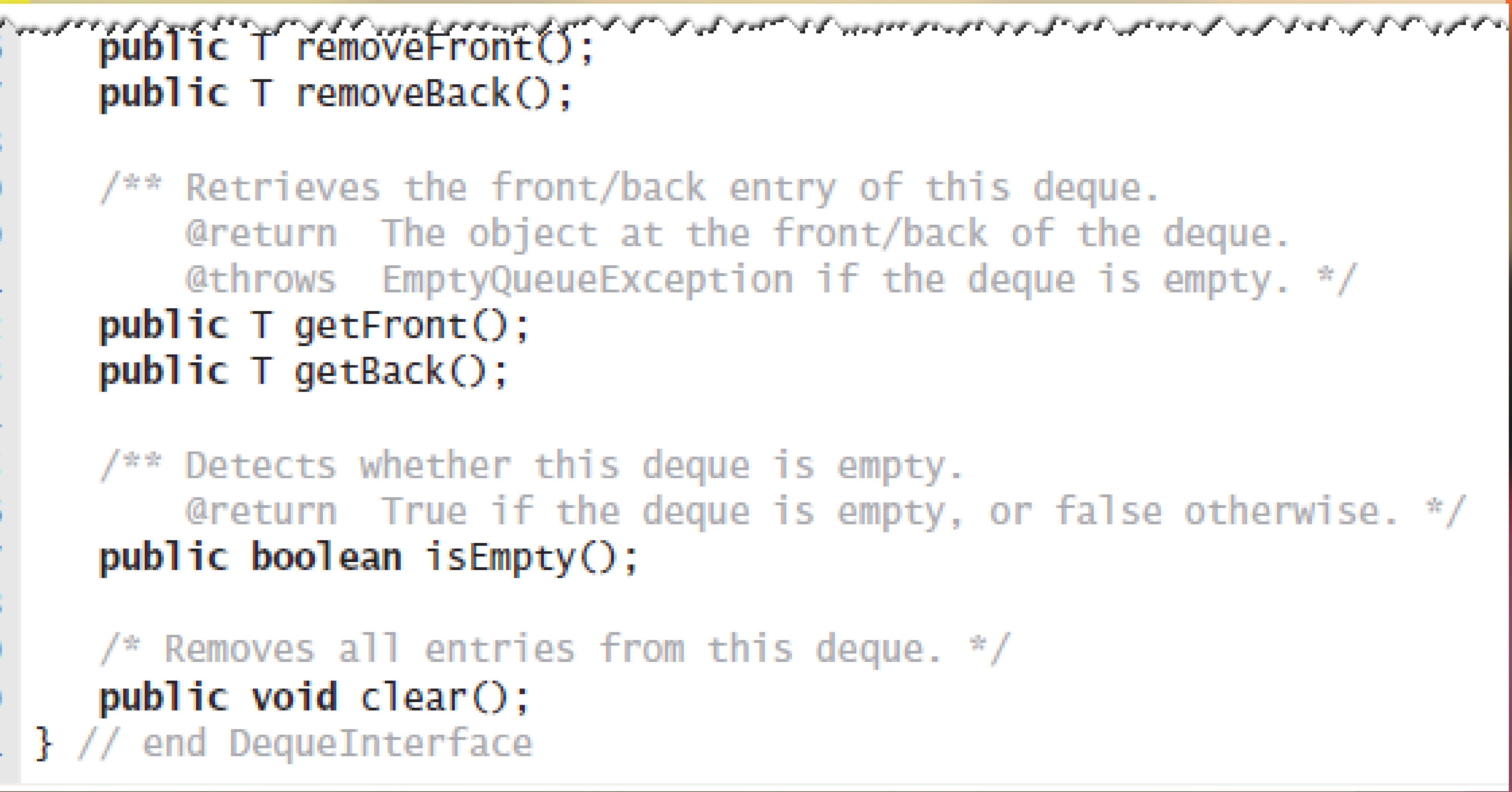
The ADT Deque



```
/**
   An interface for the ADT deque.
   @author Frank M. Carrano
*/
public interface DequeInterface<T>
{
   /** Adds a new entry to the front/back of this deque.
       @param newEntry  An object to be added. */
   public void addToFront(T newEntry);
   public void addToBack(T newEntry);

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

   /** Retrieves the front/back entry of this deque.
```

LISTING 10-4 An interface for the ADT deque

The ADT Deque



```
   public T removeFront();
   public T removeBack();

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

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

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

LISTING 10-4 An interface for the ADT deque

The ADT Deque

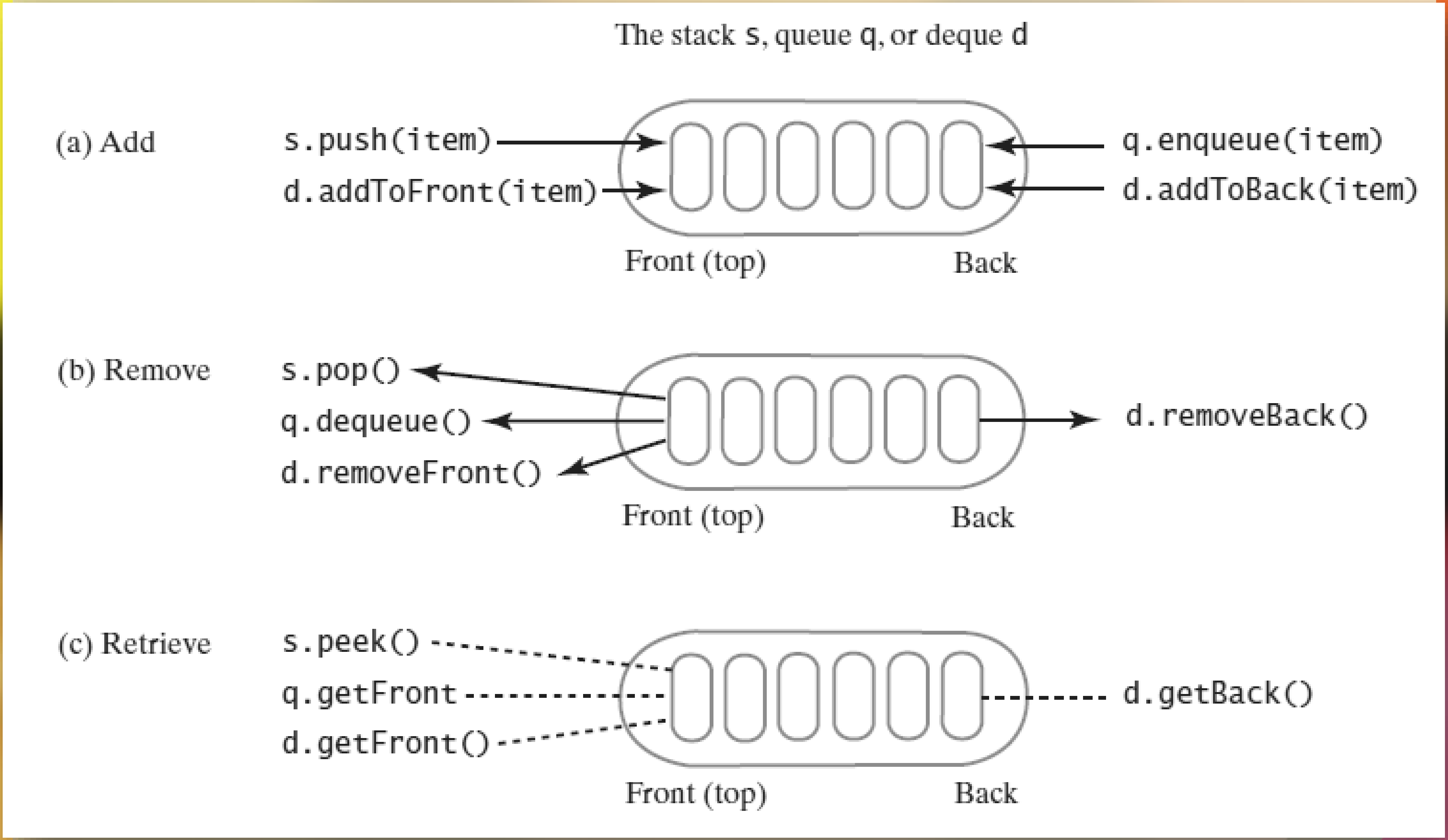
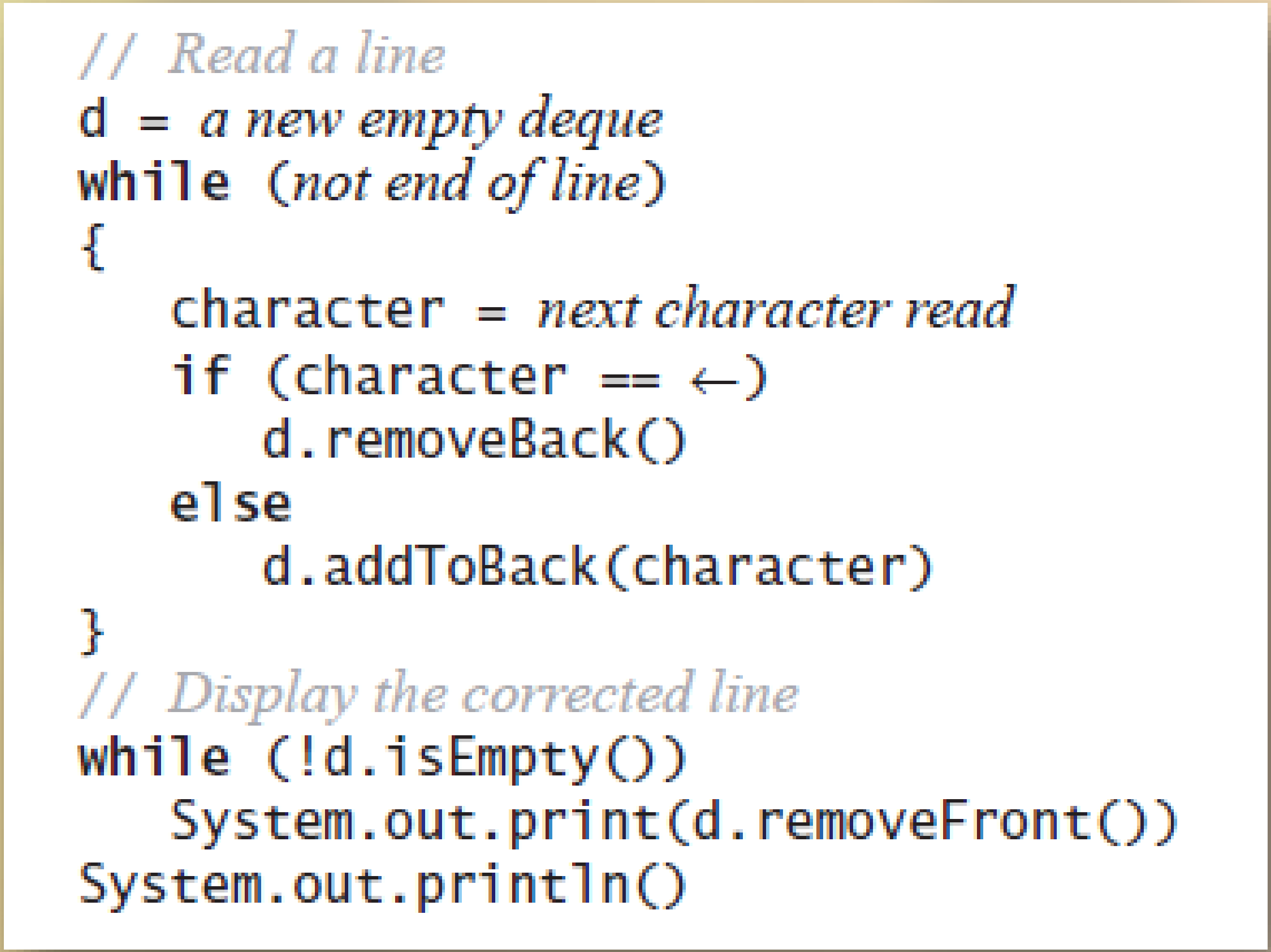


FIGURE 10-11 A comparison of operations for a stack *s*, a queue *q*, and a deque *d*: (a) add; (b) remove; (c) retrieve

The ADT Deque



```
// Read a line
d = a new empty deque
while (not end of line)
{
    character = next character read
    if (character == ←)
        d.removeBack()
    else
        d.addToBack(character)
}
// Display the corrected line
while (!d.isEmpty())
    System.out.print(d.removeFront())
System.out.println()
```

Pseudocode that uses a deque to read and display a line of keyboard input

Doubly Linked Implementation of a Deque

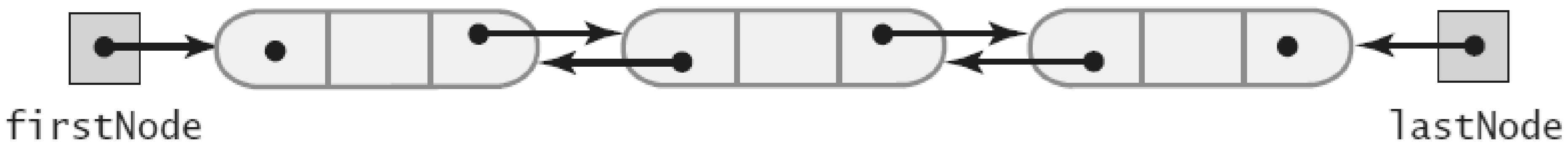
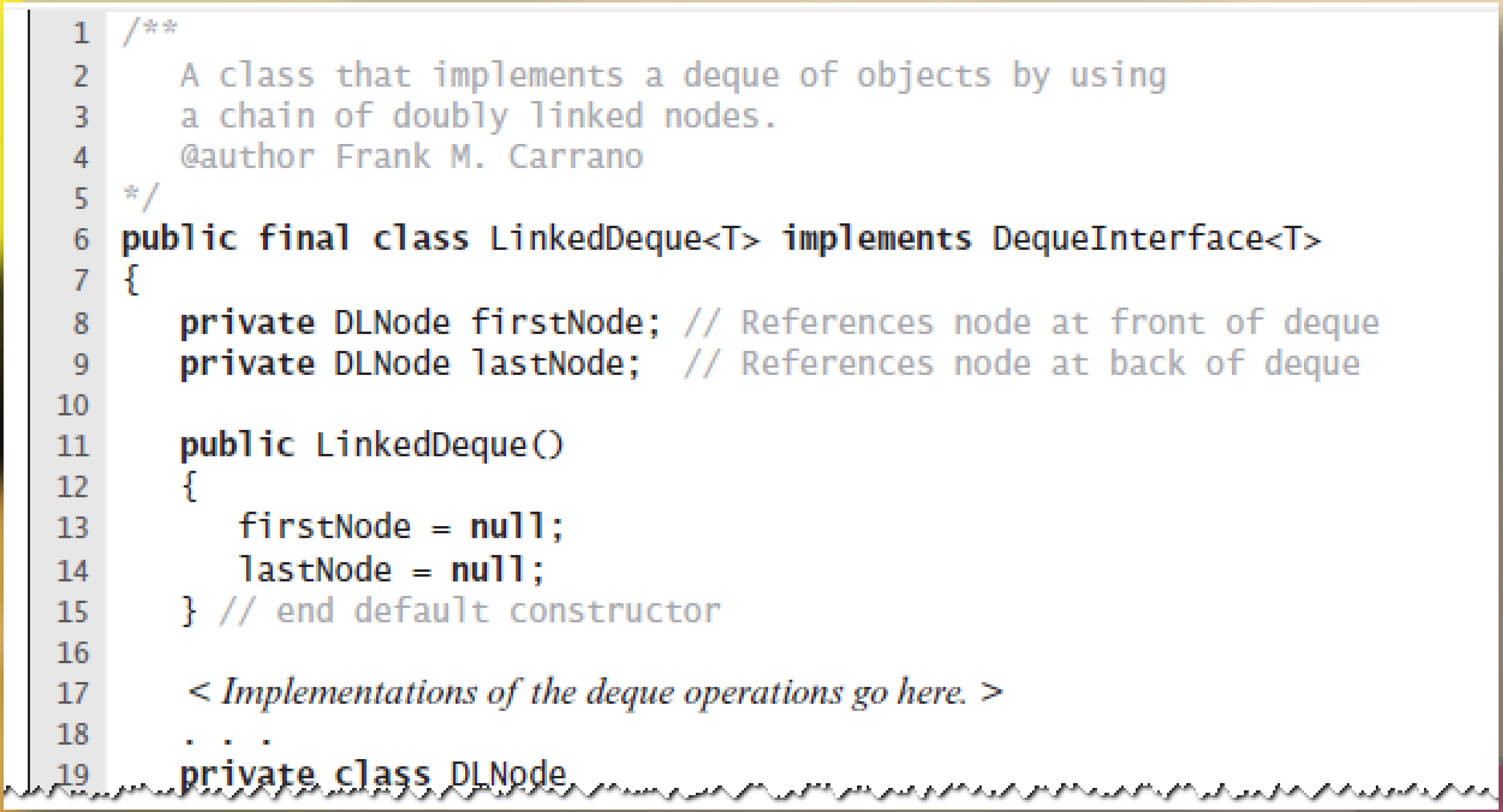


FIGURE 11-16 A doubly linked chain with head and tail references

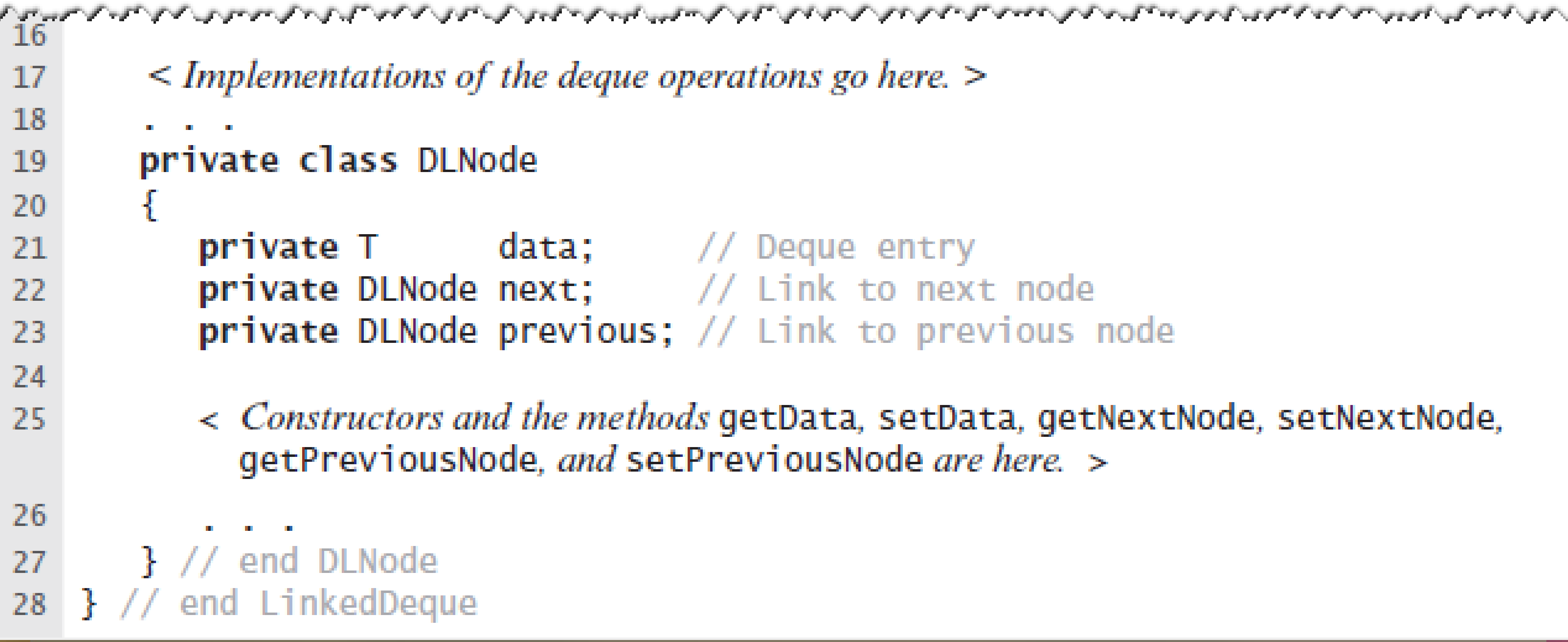
Doubly Linked Implementation of a Deque



```
/**
   A class that implements a deque of objects by using
   a chain of doubly linked nodes.
   @author Frank M. Carrano
*/
public final class LinkedDeque<T> implements DequeInterface<T>
{
   private DLNode firstNode; // References node at front of deque
   private DLNode lastNode;  // References node at back of deque

   public LinkedDeque()
   {
      firstNode = null;
      lastNode = null;
   } // end default constructor

    < Implementations of the deque operations go here. >
   . . .
   private class DLNode
```

LISTING 11-4 An outline of a linked implementation of the ADT deque

Doubly Linked Implementation of a Deque



```
   < Implementations of the deque operations go here. >
   . . .
   private class DLNode
   {
      private T      data;     // Deque entry
      private DLNode next;     // Link to next node
      private DLNode previous; // Link to previous node

      < Constructors and the methods getData, setData, getNextNode, setNextNode,
        getPreviousNode, and setPreviousNode are here. >

      . . .
   } // end DLNode
} // end LinkedDeque
```

LISTING 11-4 An outline of a linked implementation of the ADT deque

Doubly Linked Implementation of a Deque

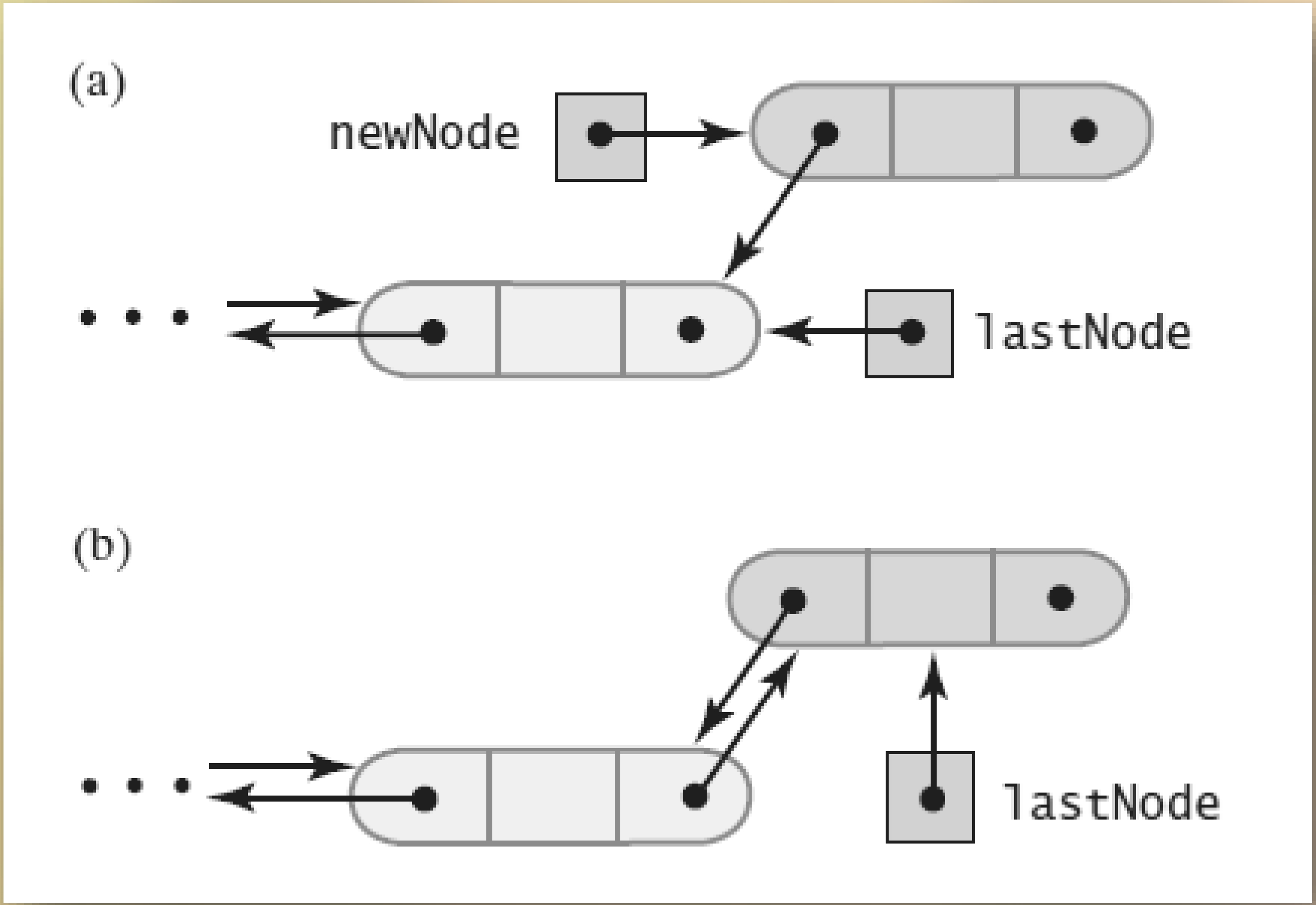
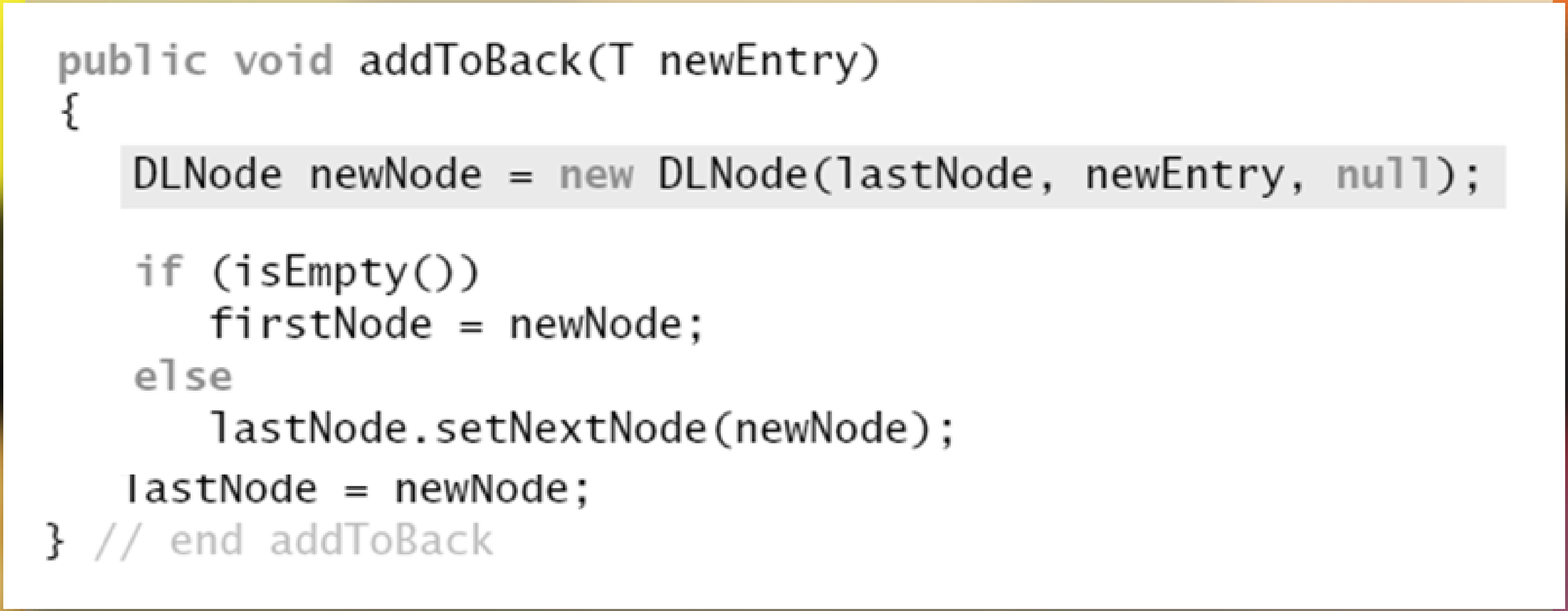


FIGURE 11-17 Adding to the back of a nonempty deque:
(a) after the new node is allocated;
(b) after the addition is complete

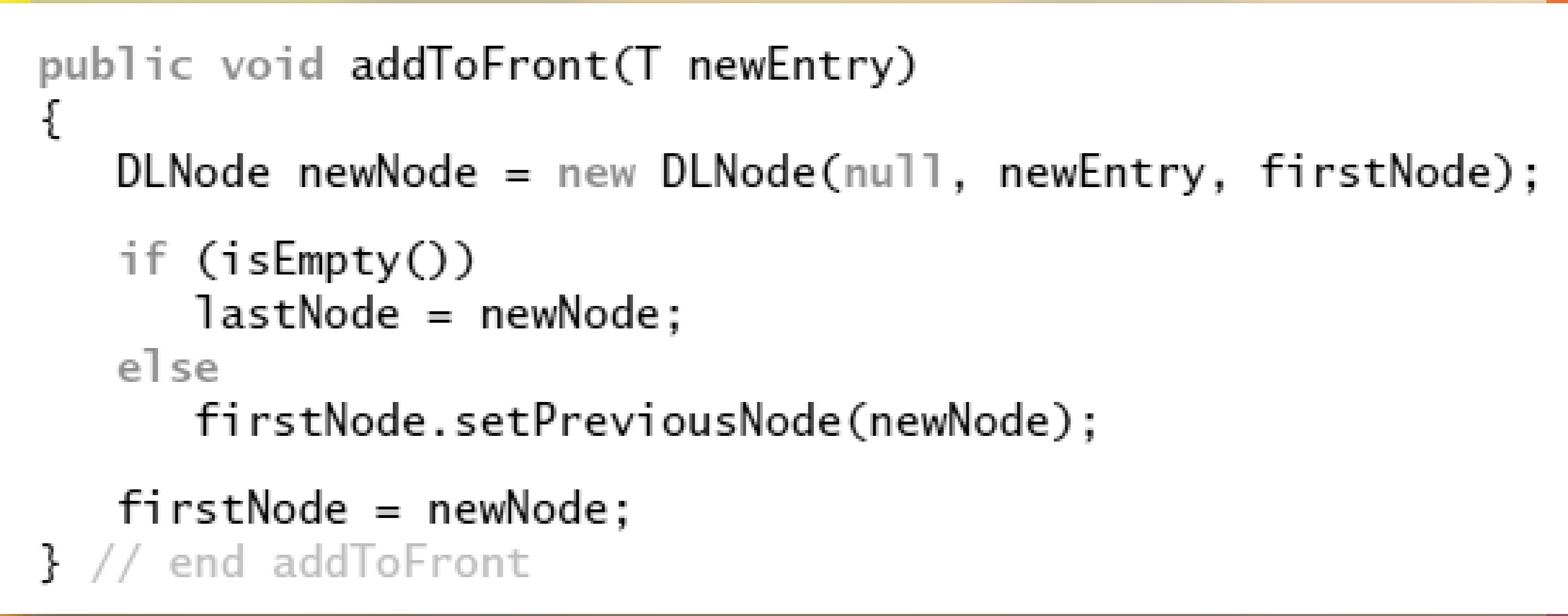
Doubly Linked Implementation of a Deque



```
public void addToBack(T newEntry)
{
   DLNode newNode = new DLNode(lastNode, newEntry, null);

   if (isEmpty())
      firstNode = newNode;
   else
      lastNode.setNextNode(newNode);
   lastNode = newNode;
} // end addToBack
```

LISTING 11-4 An outline of a linked implementation of the ADT deque

Doubly Linked Implementation of a Deque



```
public void addToFront(T newEntry)
{
   DLNode newNode = new DLNode(null, newEntry, firstNode);

   if (isEmpty())
      lastNode = newNode;
   else
      firstNode.setPreviousNode(newNode);

   firstNode = newNode;
} // end addToFront
```

Doubly Linked Implementation of a Deque

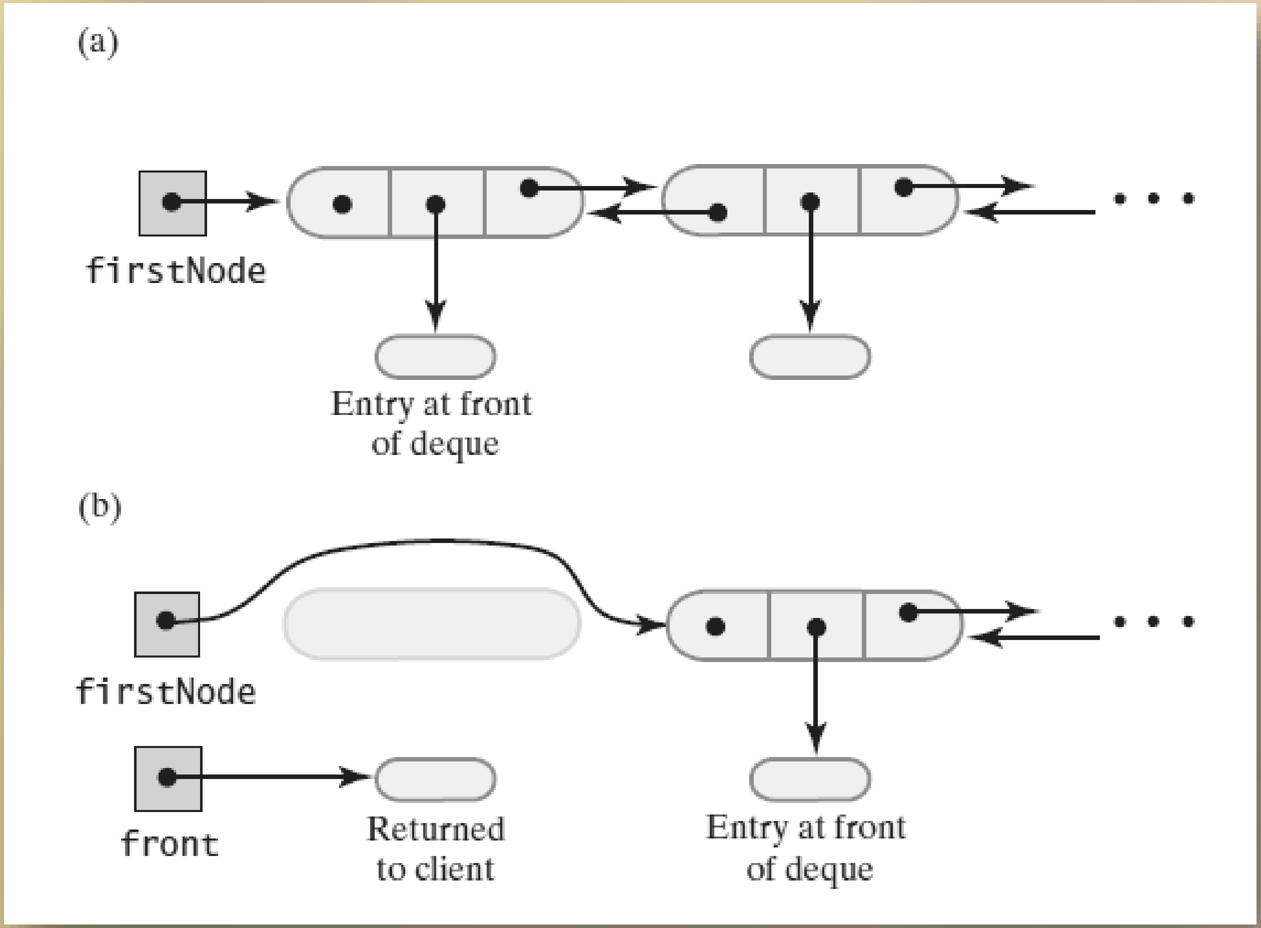
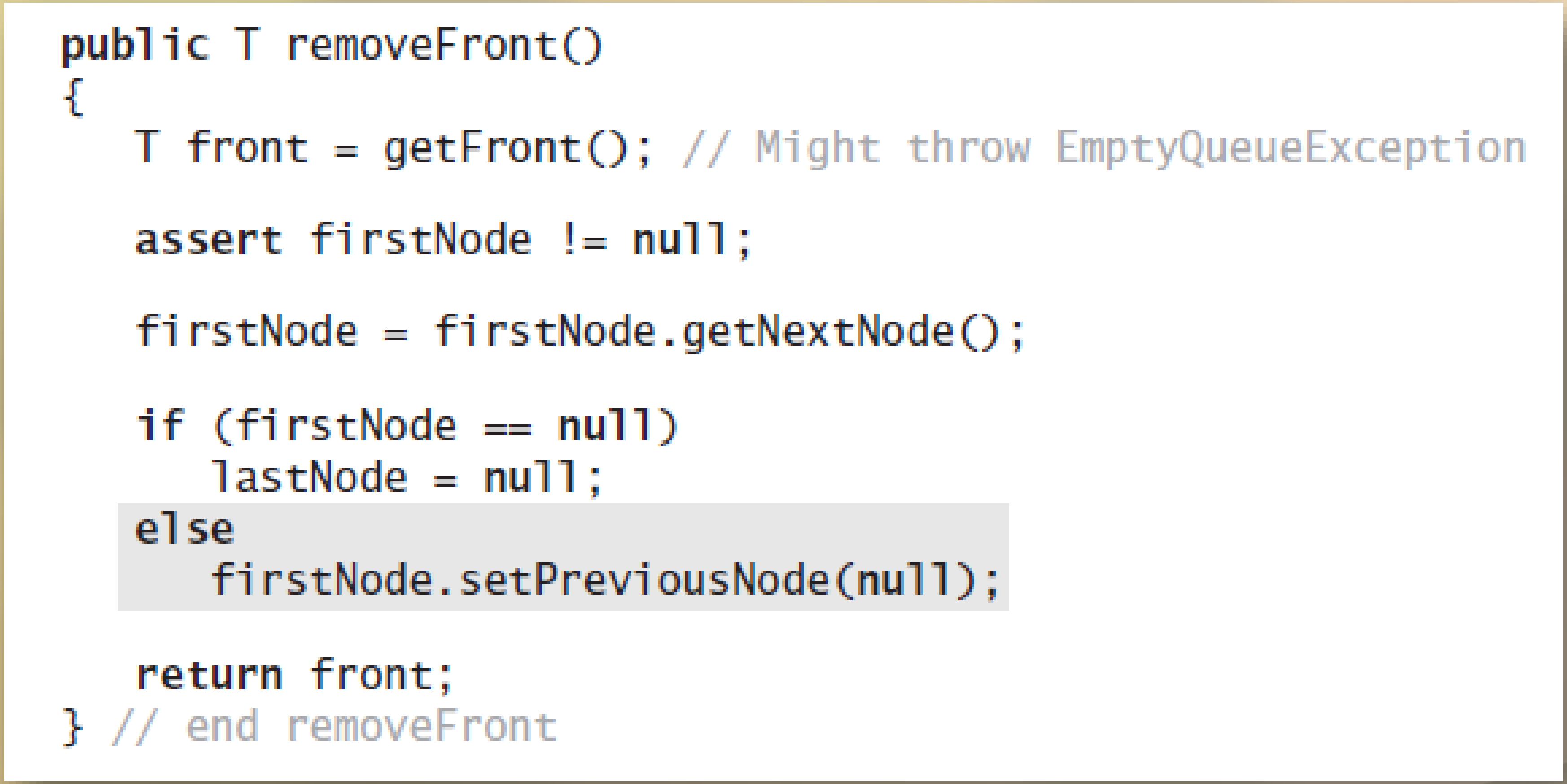


FIGURE 11-18 (a) A deque containing at least two entries; (b) after removing the first node and obtaining a reference to the deque's new first entry.

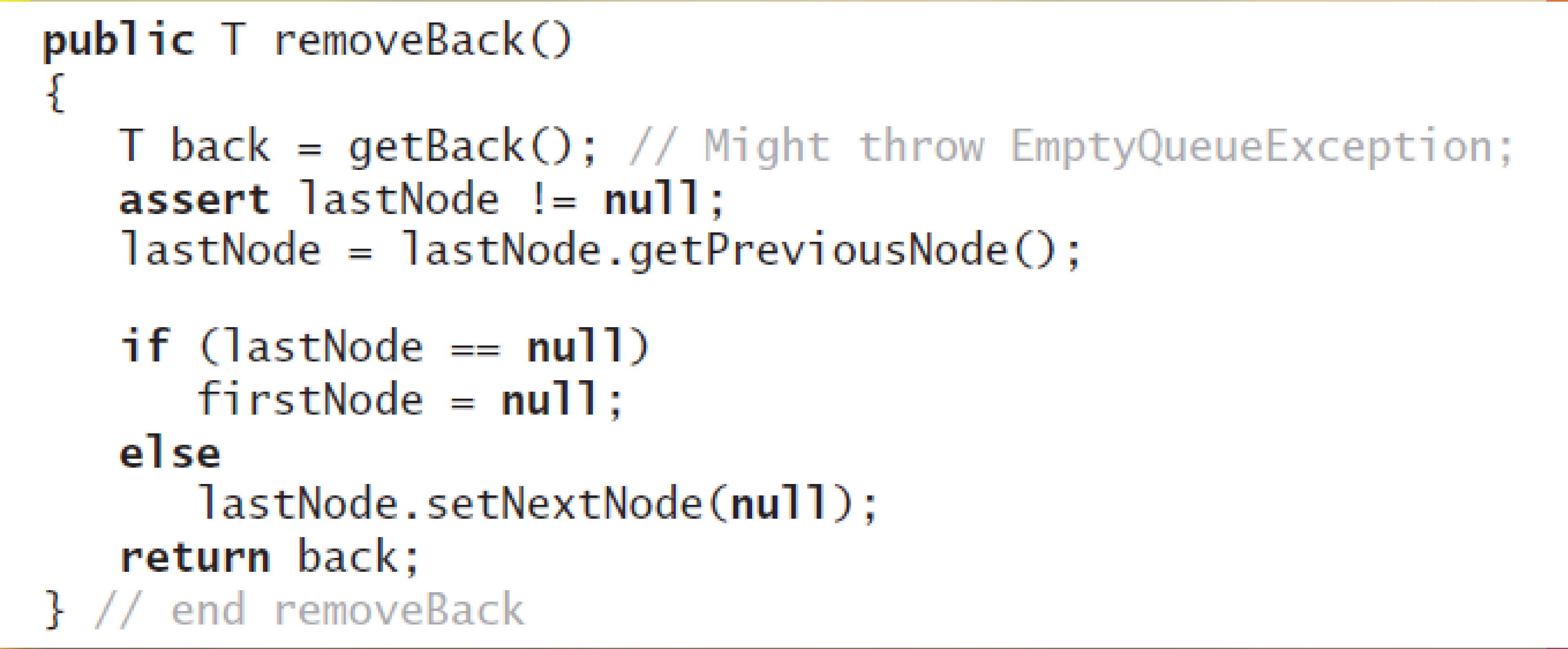
Doubly Linked Implementation of a Deque



```
public T removeFront()
{
   T front = getFront(); // Might throw EmptyQueueException

   assert firstNode != null;

   firstNode = firstNode.getNextNode();

   if (firstNode == null)
      lastNode = null;
   else
      firstNode.setPreviousNode(null);

   return front;
} // end removeFront
```

Implementation of `removeFront`.

Doubly Linked Implementation of a Deque



```
public T removeBack()
{
   T back = getBack(); // Might throw EmptyQueueException;
   assert lastNode != null;
   lastNode = lastNode.getPreviousNode();

   if (lastNode == null)
      firstNode = null;
   else
      lastNode.setNextNode(null);
   return back;
} // end removeBack
```

Implementation of `removeBack`, an O(1) operation.

Java Class Library: The Interface `Deque`

Methods provided

- `addFirst, offerFirst`
- `addLast, offerLast`
- `removeFirst, pollFirst`
- `removeLast, pollLast`
- `getFirst, peekFirst`
- `getLast, peekLast`
- `isEmpty, clear, size`
- `push, pop`

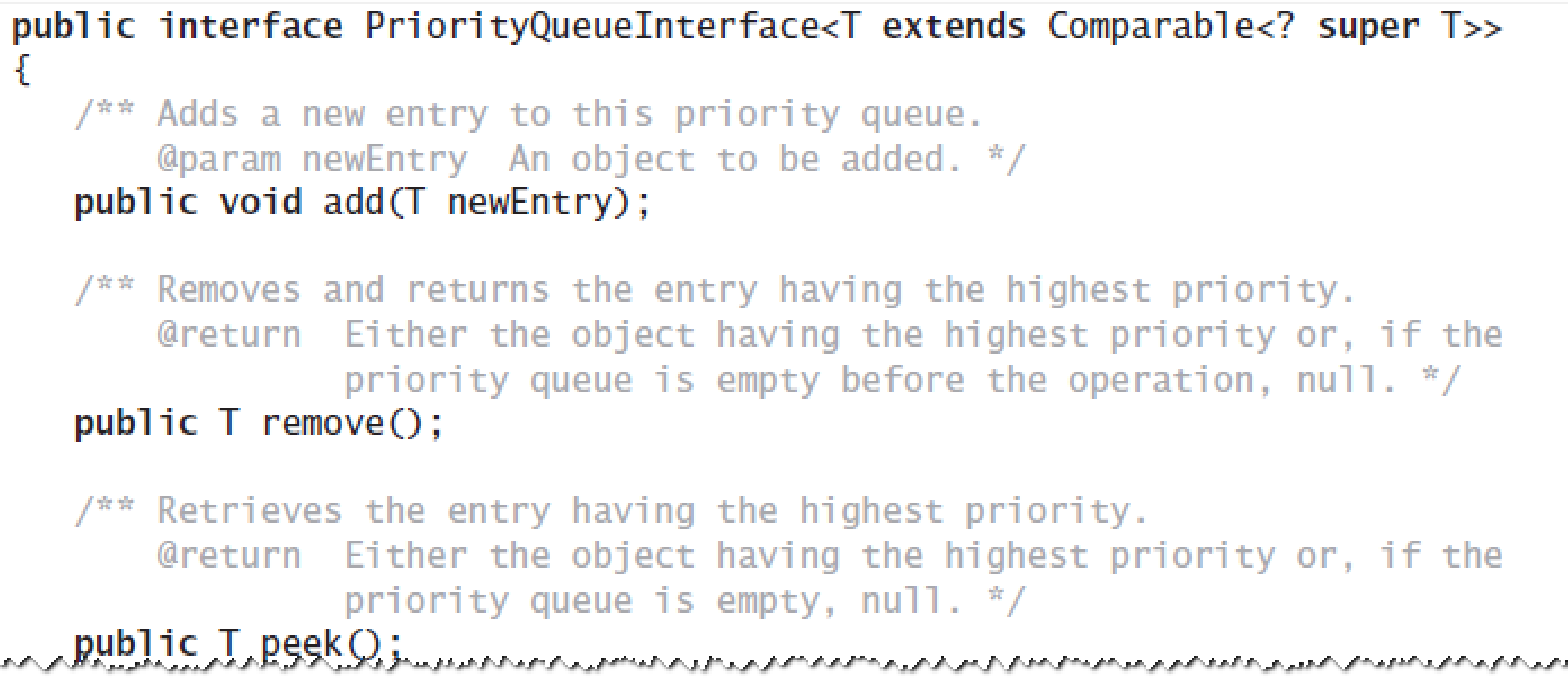
Java Class Library: The Class `ArrayDeque`

- Implements the interface `Deque`
- Constructors provided
  - `ArrayDeque()`
  - `ArrayDeque(int initialCapacity)`

ADT Priority Queue

- Consider how a hospital assigns a priority to each patient that *overrides* time at which patient arrived.
- ADT priority queue organizes objects according to their priorities
- Definition of "priority" depends on nature of the items in the queue

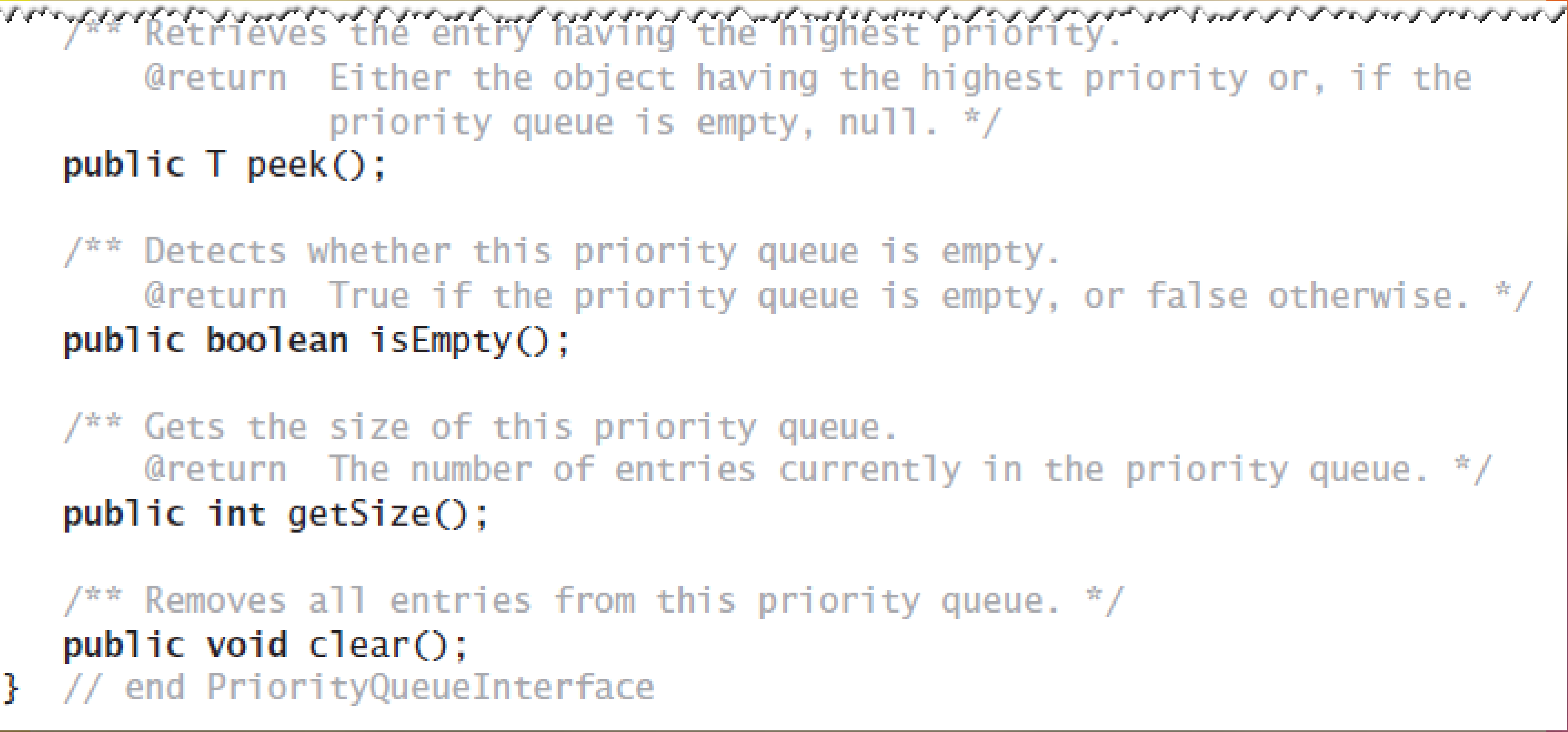
ADT Priority Queue



```
public interface PriorityQueueInterface<T extends Comparable<? super T>>
{
   /** Adds a new entry to this priority queue.
       @param newEntry  An object to be added. */
   public void add(T newEntry);

   /** Removes and returns the entry having the highest priority.
       @return  Either the object having the highest priority or, if the
                priority queue is empty before the operation, null. */
   public T remove();

   /** Retrieves the entry having the highest priority.
       @return  Either the object having the highest priority or, if the
                priority queue is empty, null. */
   public T peek();
```

LISTING 10-5 An interface for the ADT priority queue

ADT Priority Queue



```
   /** Retrieves the entry having the highest priority.
       @return  Either the object having the highest priority or, if the
                priority queue is empty, null. */
   public T peek();

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

   /** Gets the size of this priority queue.
       @return  The number of entries currently in the priority queue. */
   public int getSize();

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

LISTING 10-5 An interface for the ADT priority queue

Possible Implementations of a Priority Queue

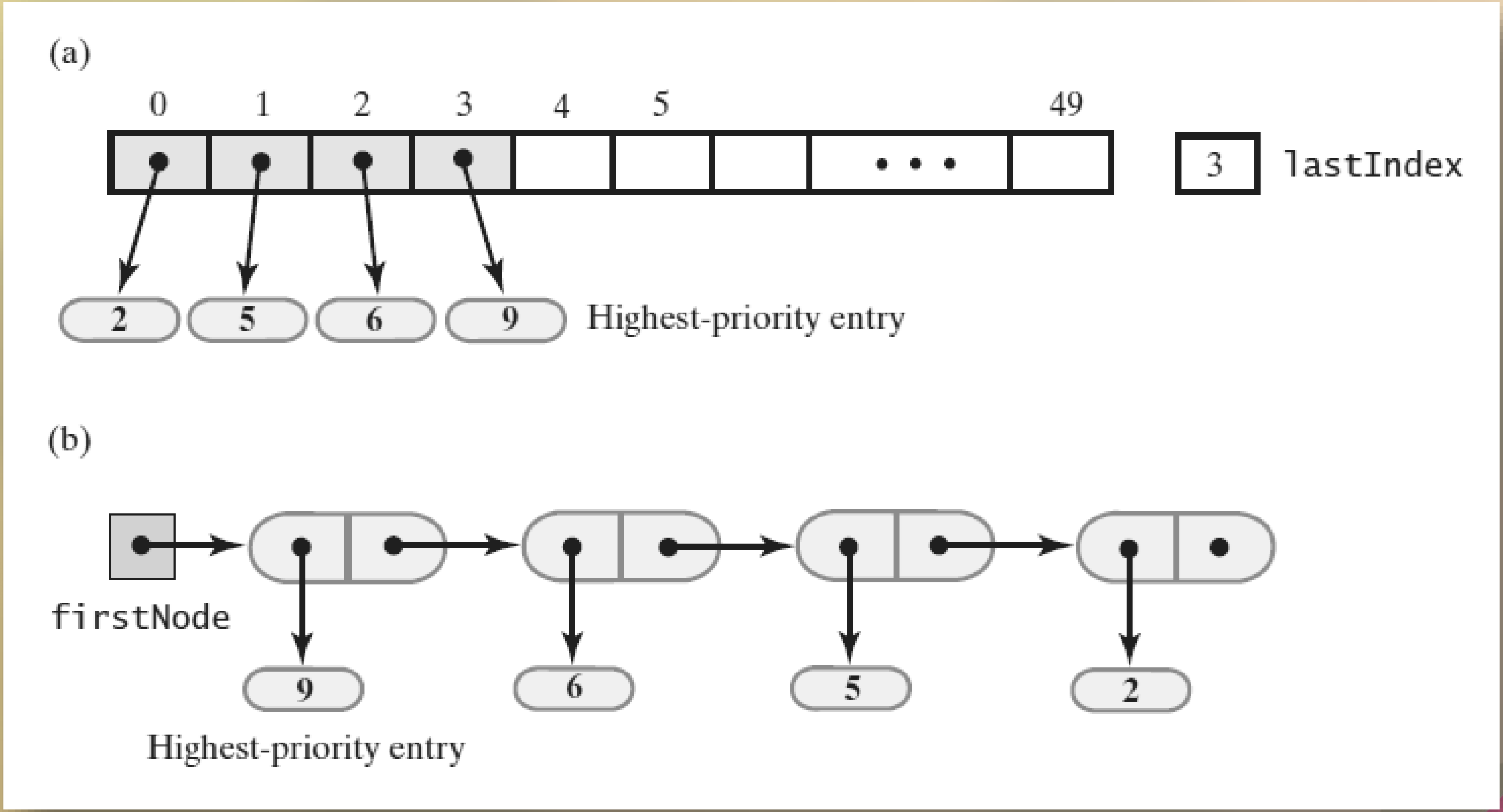


FIGURE 11-19 Two possible implementations of a priority queue using (a) an array; (b) a chain of linked nodes

Java Class Library: The Class `PriorityQueue`

Basic constructors and methods

- `PriorityQueue`
- `add`
- `offer`
- `remove`
- `poll`
- `element`
- `peek`
- `isEmpty`, `clear`, `size`

End

Chapter 11