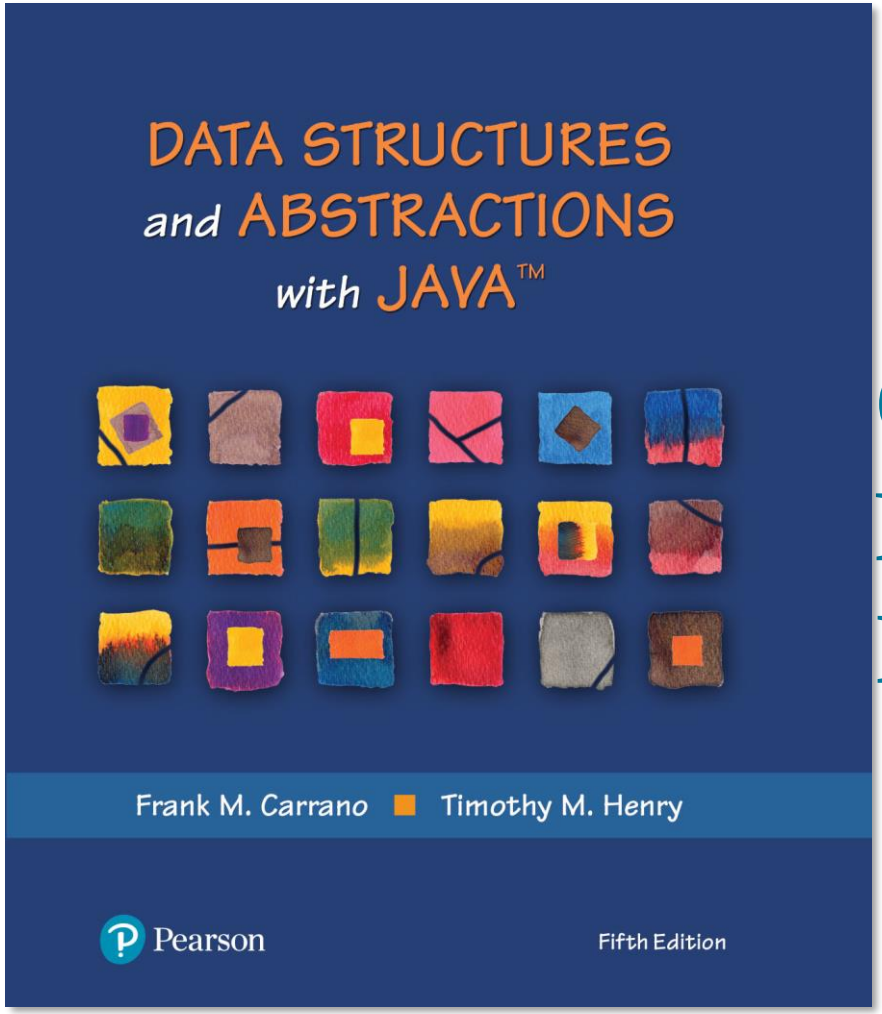
Data Structures and Abstractions with Java™

5th Edition



Chapter 8

Queue, Deque, and Priority Queue Implementations

Linked Implementation of a Queue

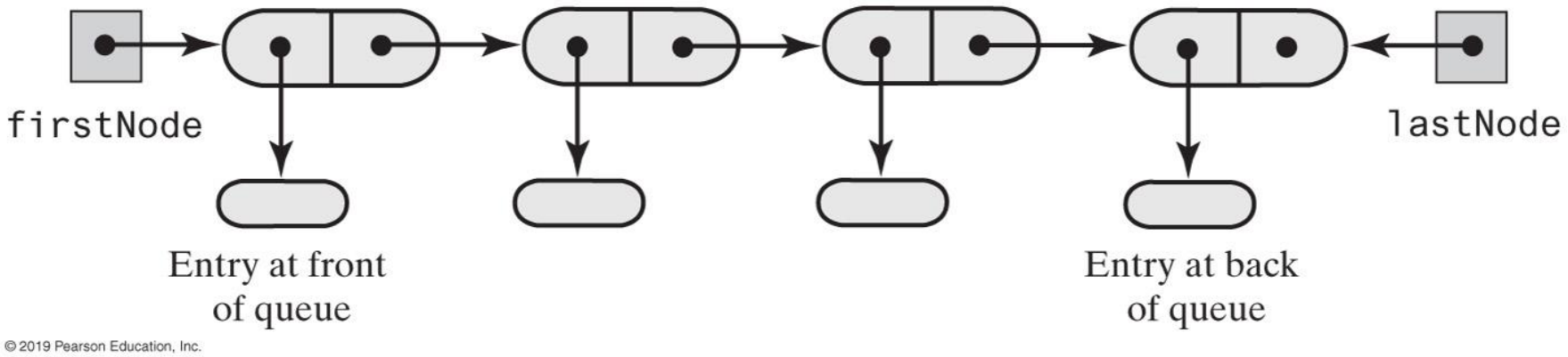


FIGURE 8-1 A chain of linked nodes that implements a queue

Linked Implementation of a Queue

```
/** A class that implements a queue of objects by using
  a chain of linked nodes that has both head and tail references. */
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. >
//  . . .

      private class Node
      {
        //  < Implementation of the inner class Node goes here. >

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

LISTING 8-1 An outline of a linked implementation of the ADT queue

Linked Implementation of a Queue

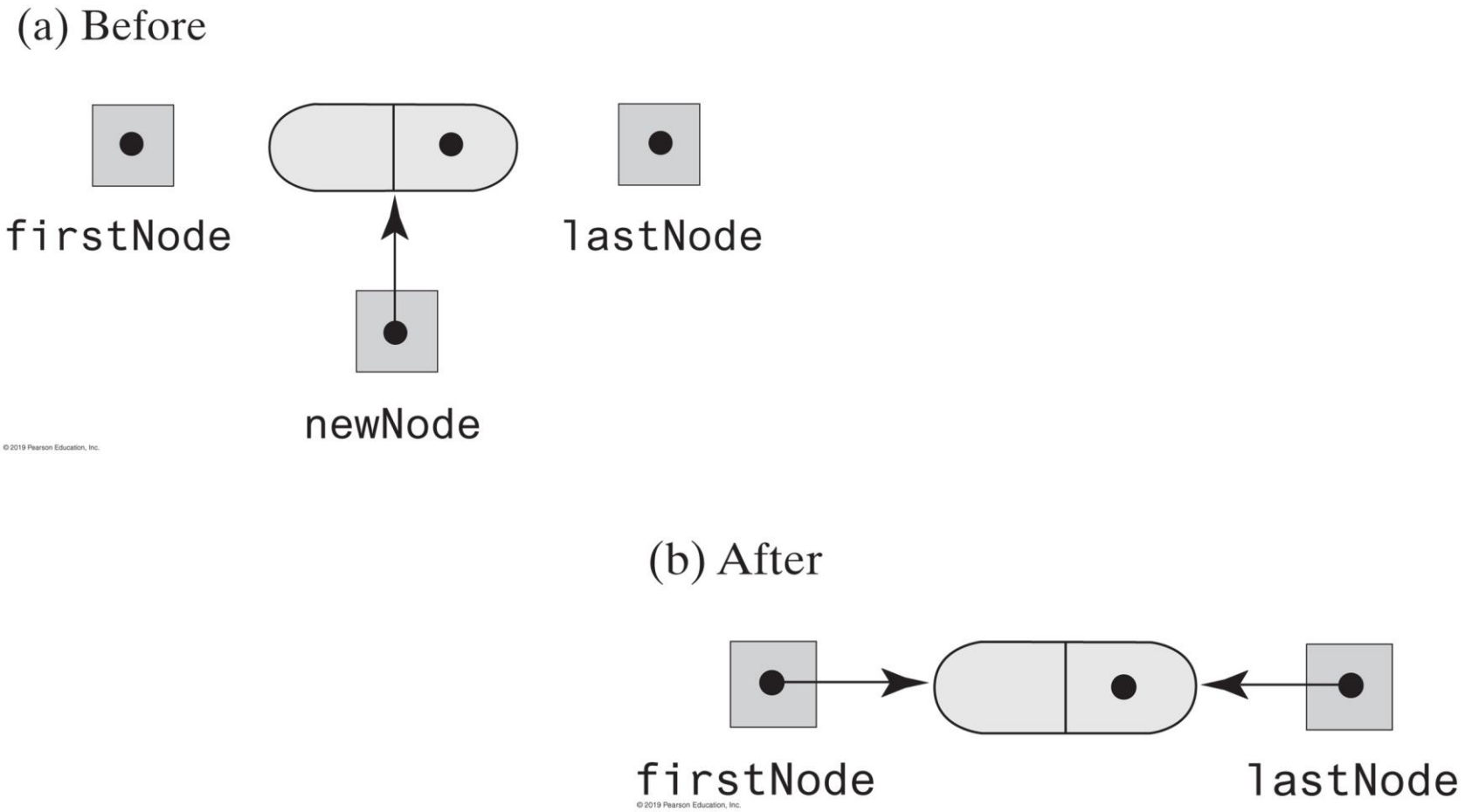


FIGURE 8-2 Before and after adding a new node to an empty chain

FIGURE 8-3 Adding a new node to the end of a nonempty chain that has a tail reference

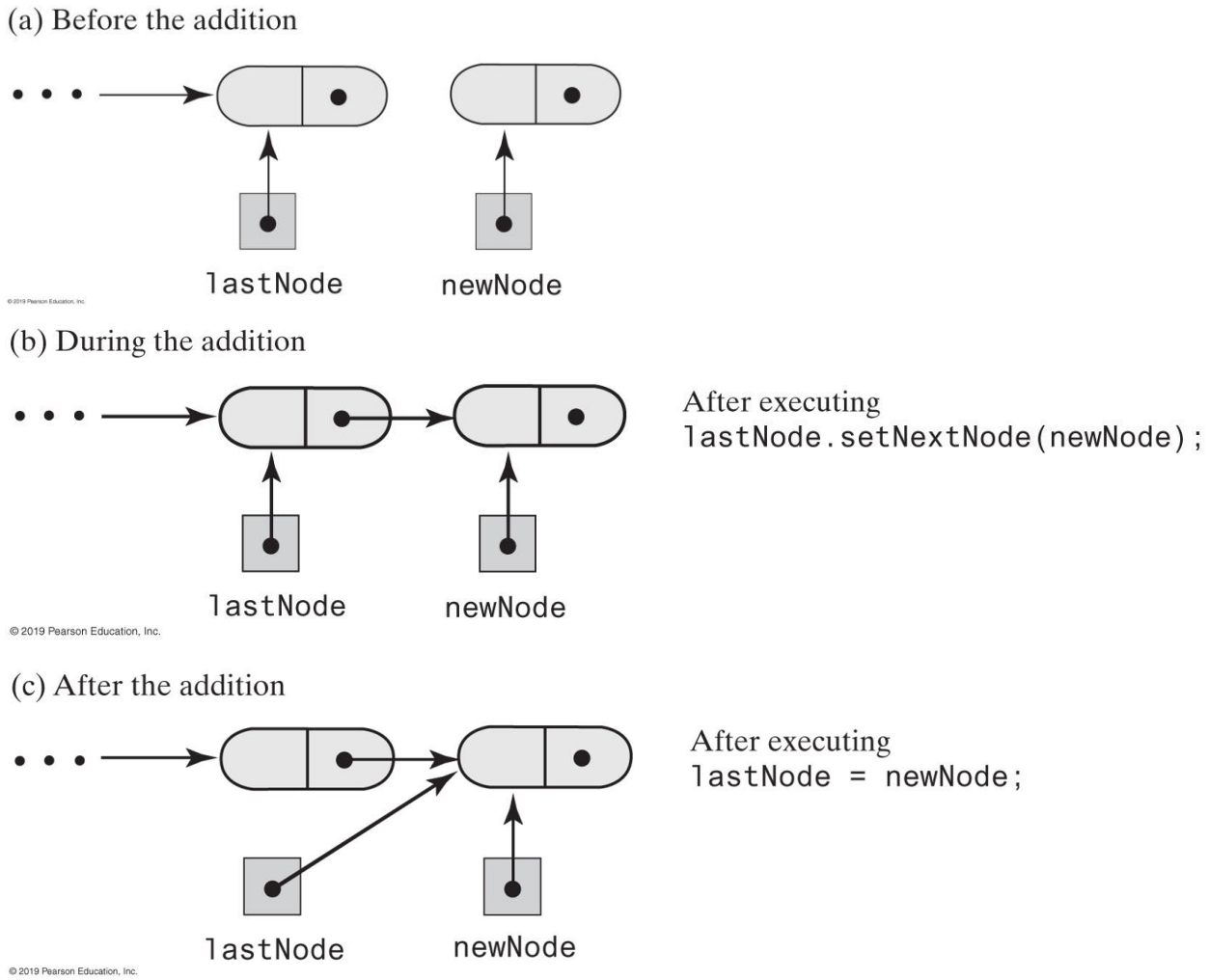


FIGURE 8-3 Adding a new node to the end of a nonempty chain that has a tail reference

Linked Implementation of a Queue

```
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` Performance is O(1)

Linked Implementation of a Queue

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

Retrieving the front entry

FIGURE 8-4 Before and after removing the entry at the front of a queue that has more than one entry

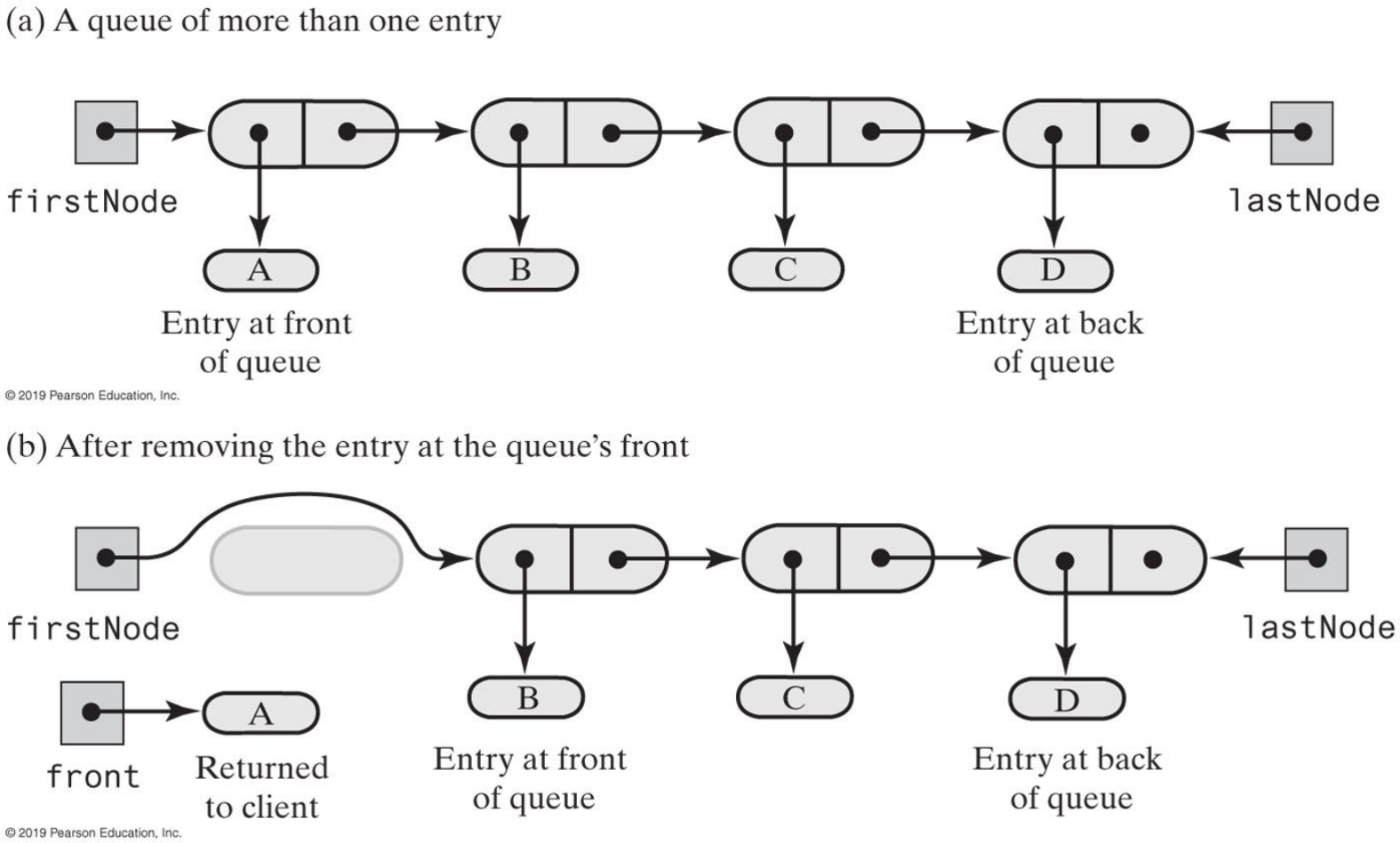


FIGURE 8-4 Before and after removing the entry at the front of a queue that has more than one entry

Linked Implementation of a Queue

```
public T dequeue()
{
  T front = getFront();  // Might throw EmptyQueueException
  // Assertion: 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

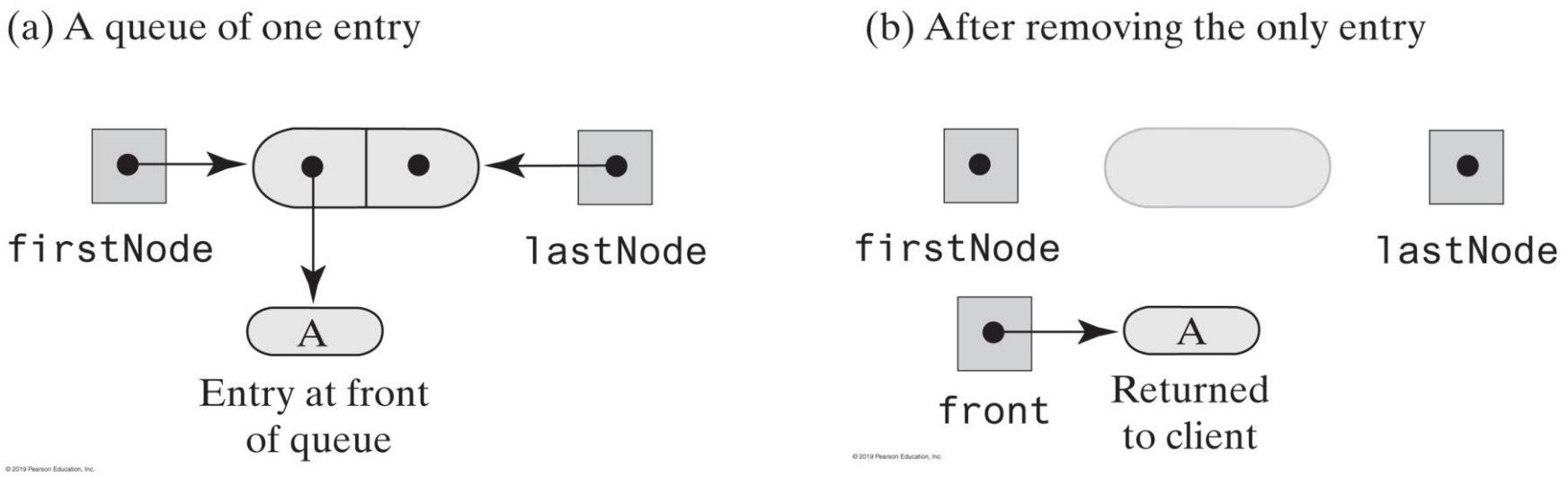


FIGURE 8-5 Before and after removing the only entry from a queue

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

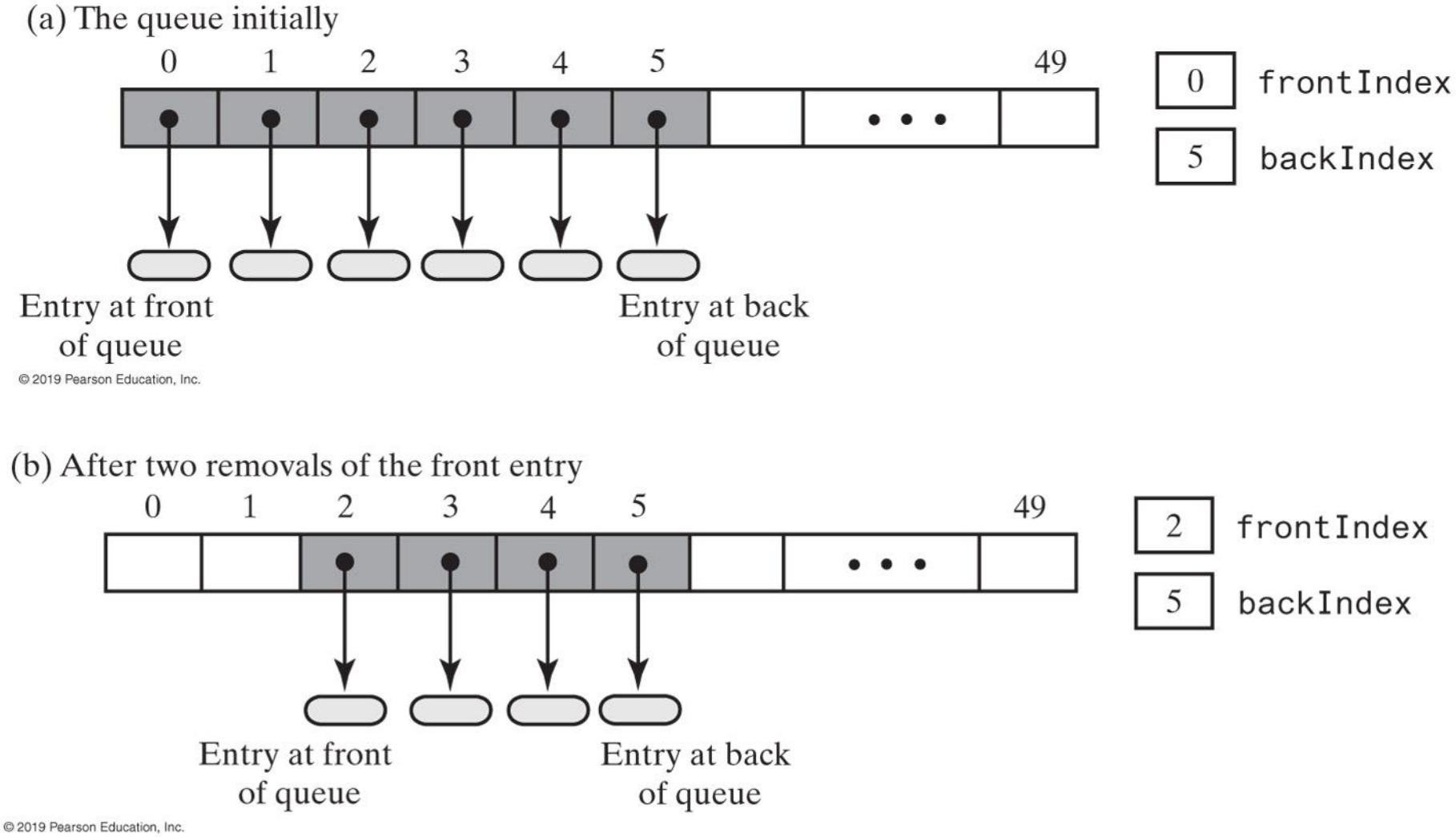


FIGURE 8-6 An array that represents a queue without moving any entries during additions and removals

Circular Array

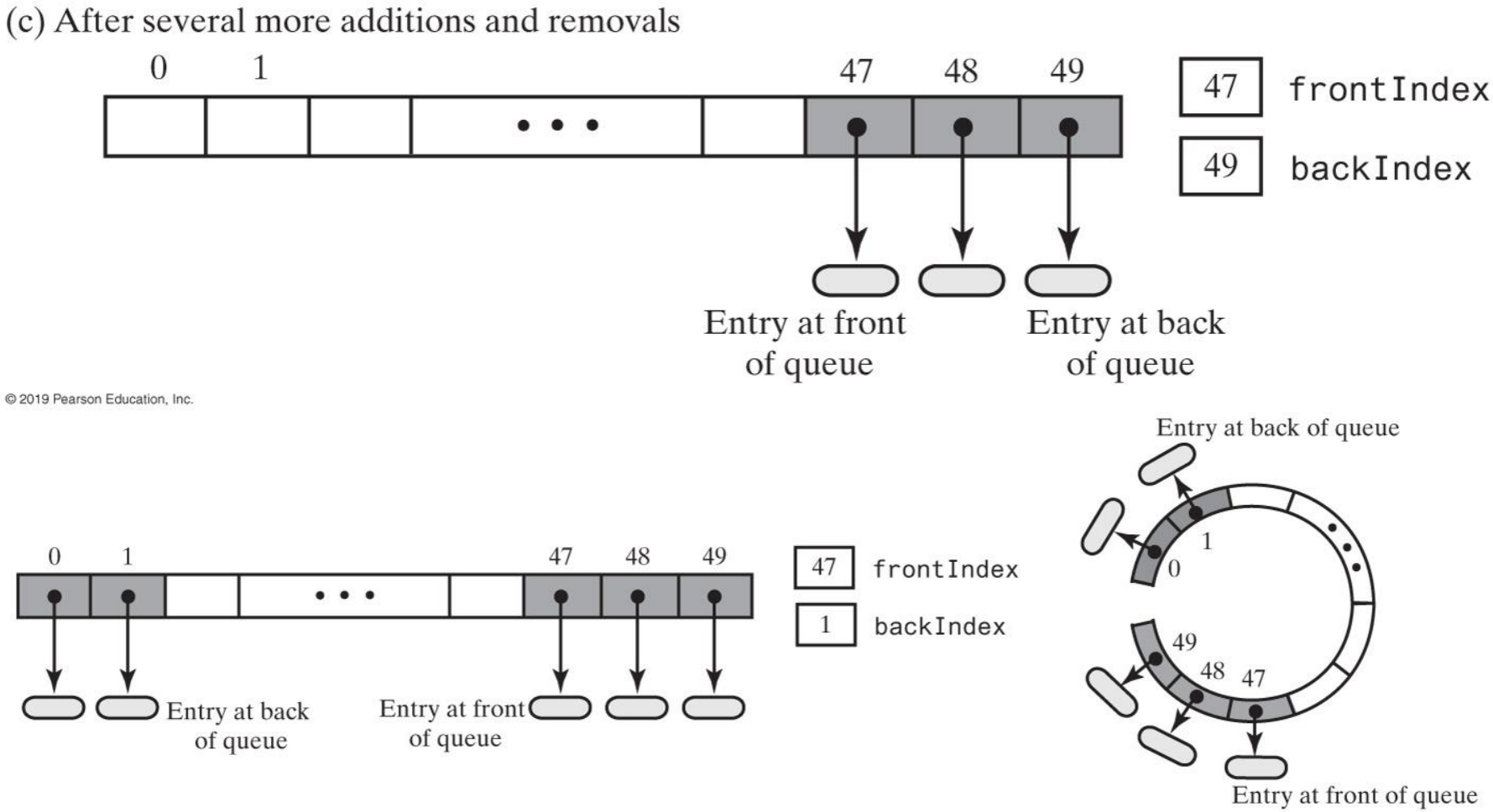


FIGURE 8-7 A circular array that represents the queue in Figure 8-6c after adding two more entries

Circular Array

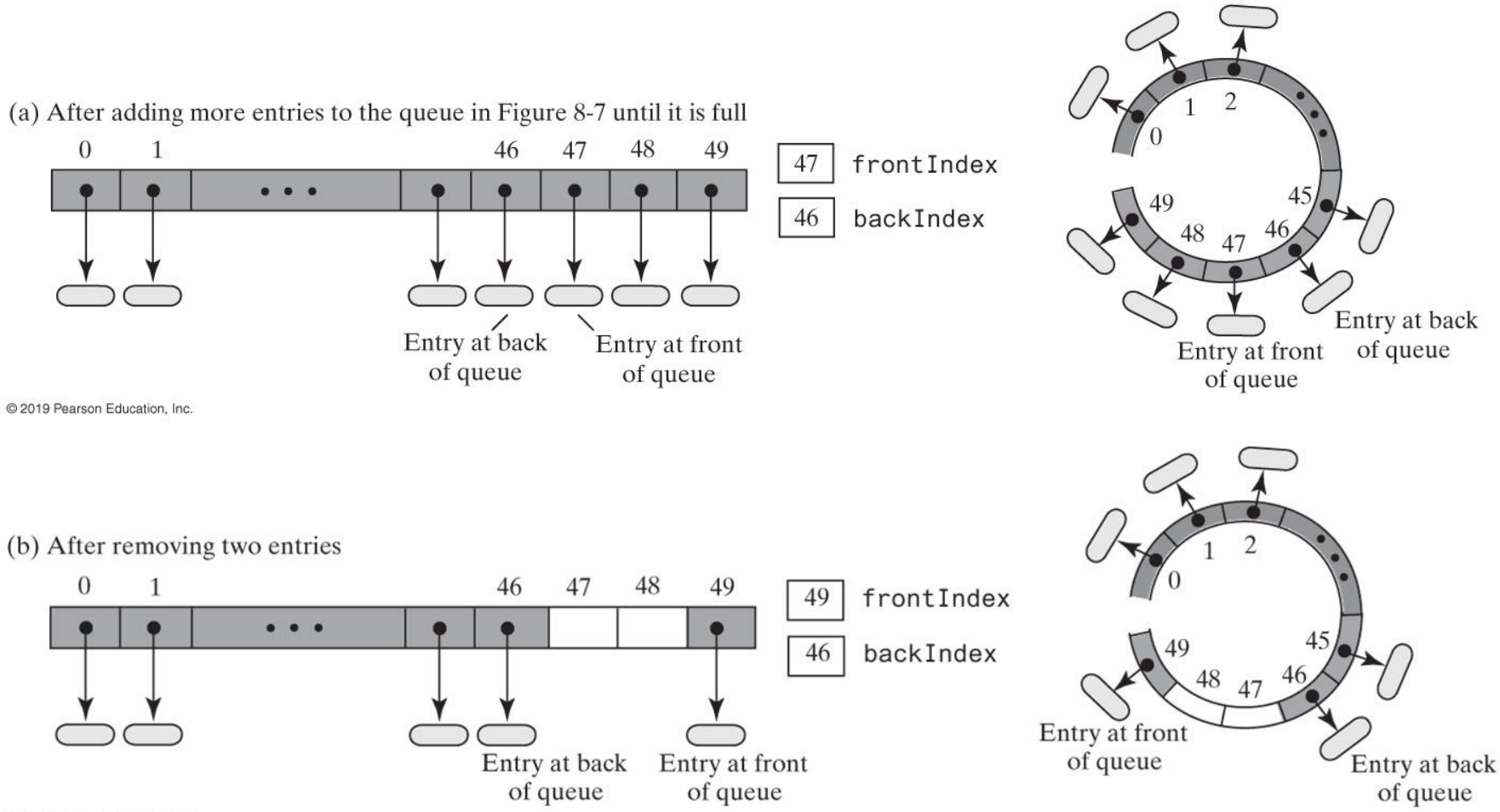


FIGURE 8-8a&b A circular array representation of a queue as entries are removed

Circular Array

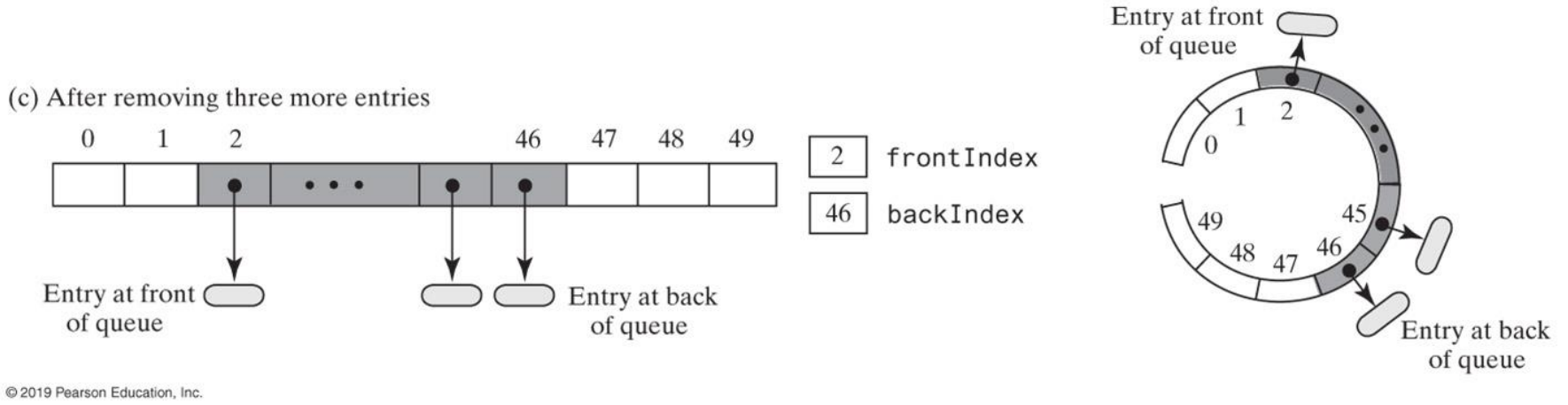


FIGURE 8-8c&d A circular array representation of a queue as entries are removed

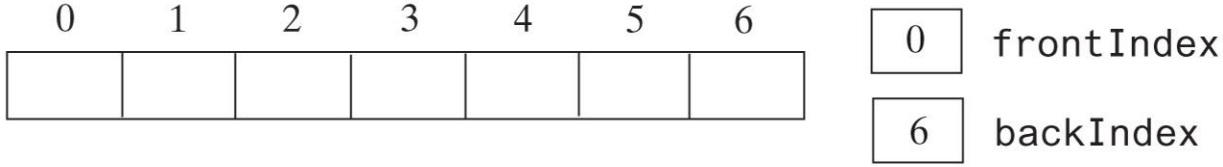
Circular Array with One Unused Location

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

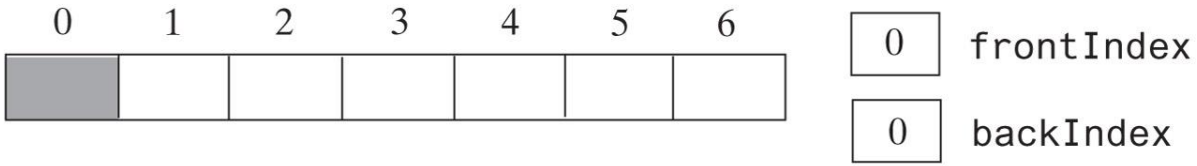
Retrieving the front entry

Circular Array (Part 1)

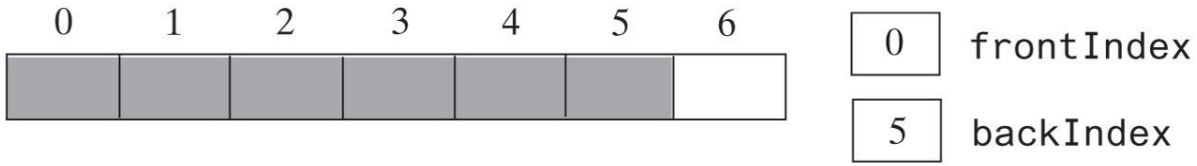
(a) Initially, the queue is empty



(b) After enqueuing one entry



(c) After enqueuing five more entries, the queue is full



(d) After dequeuing an entry

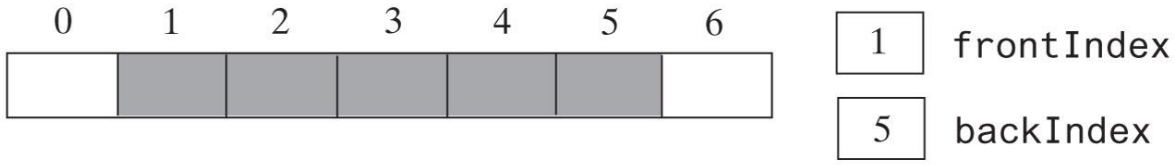


FIGURE 8-9 A seven-element circular array that contains at most six entries of a queue

Circular Array (Part 2)

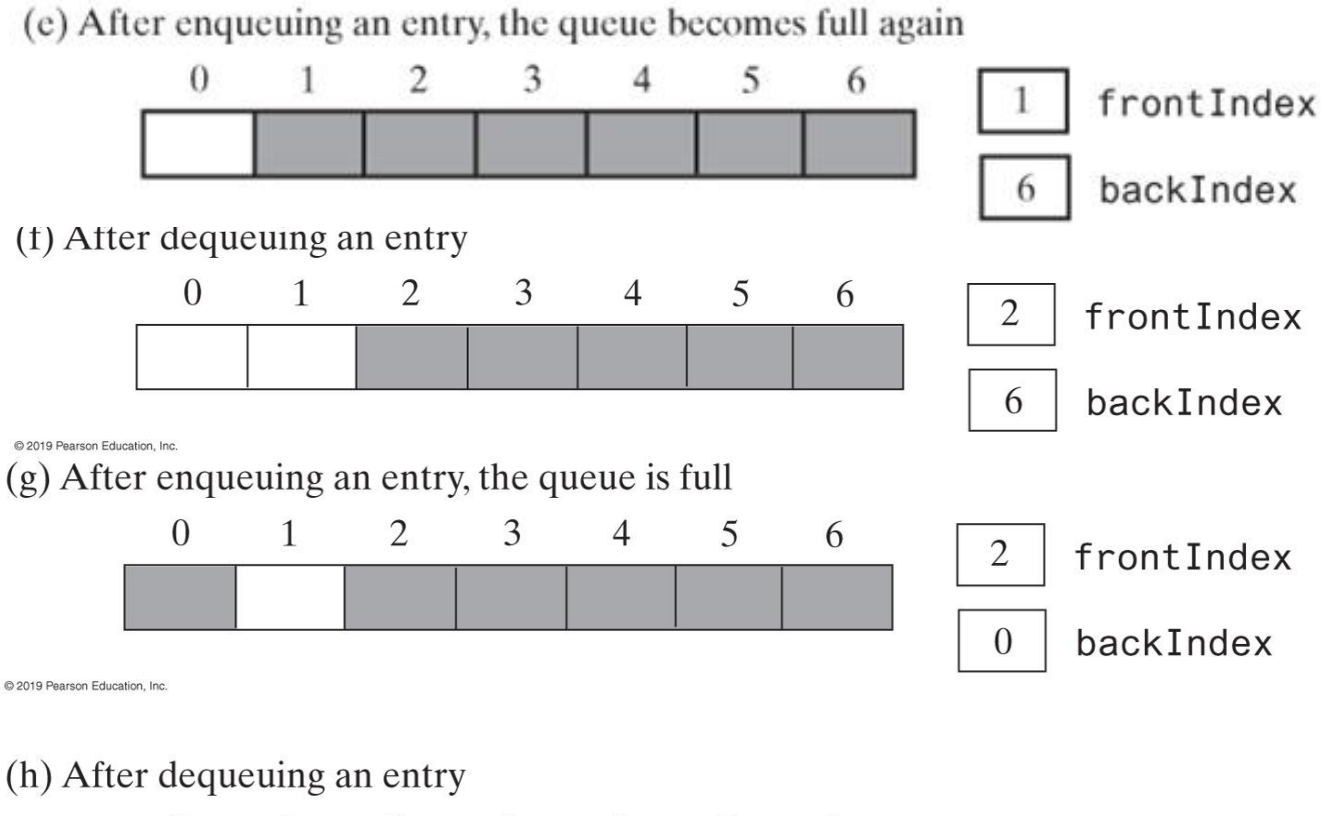
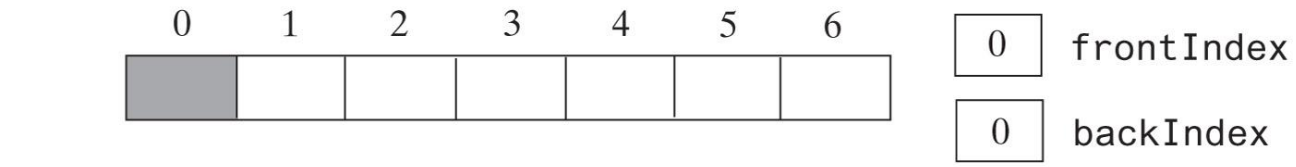


FIGURE 8-9 A seven-element circular array that contains at most six entries of a queue

Circular Array (Part 3)

(i) After dequeuing all but one entry



(j) After dequeuing the remaining entry, the queue is now empty

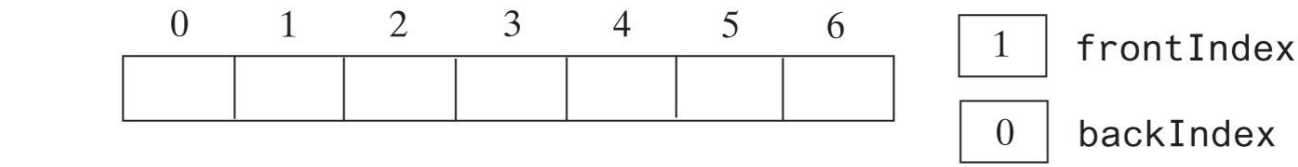


FIGURE 8-9 A seven-element circular array that contains at most six entries of a queue

Circular Array with One Unused Location

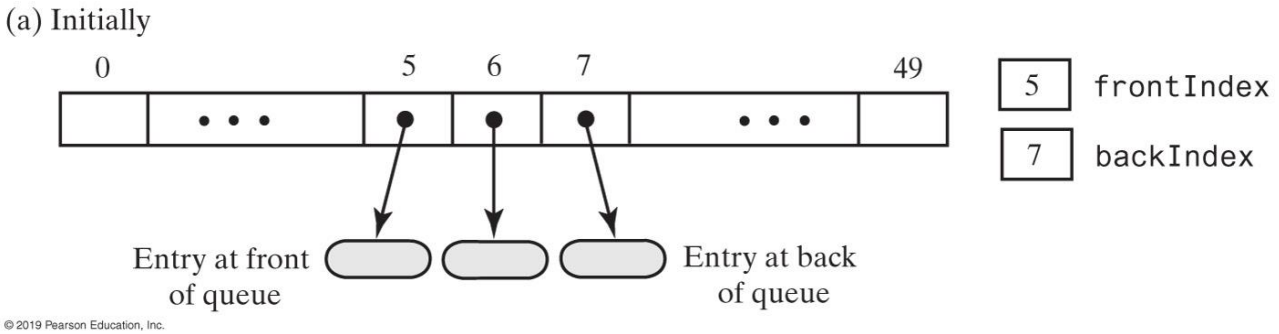


FIGURE 8-10 An array-based queue and two ways to remove its front entry

Circular Array with One Unused Location

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

Implementation of dequeue

Circular Array with One Unused Location

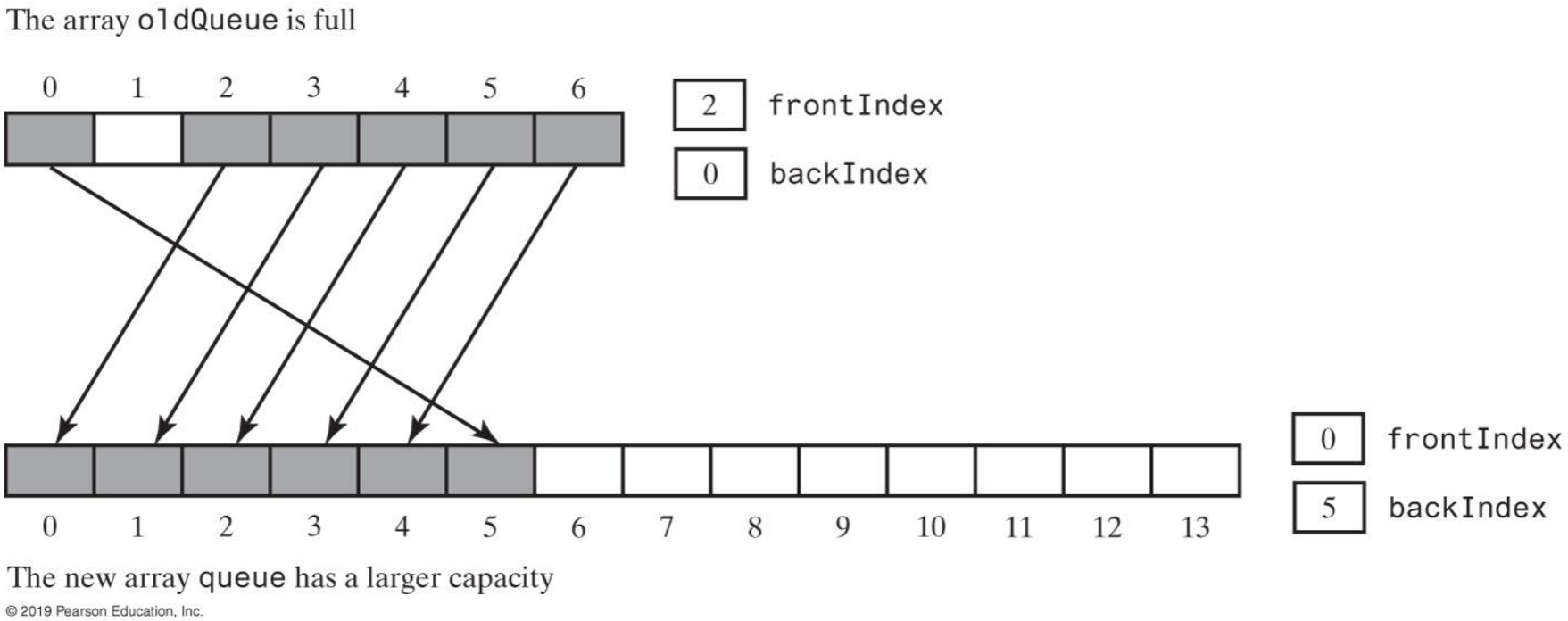


FIGURE 8-11 Doubling the size of an array-based queue

Circular Array with One Unused Location

```
// Doubles the size of the array queue if it is full.
// Precondition: checkIntegrity 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);
    integrityOK = false;

    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] tempQueue = (T[]) new Object[newSize];
    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;
    integrityOK = true;
  } // end if
} // end ensureCapacity
```

Definition of `ensureCapacity`

Circular Array with One Unused Location

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

Implementation of `isEmpty`

Circular Linked Implementations of a Queue

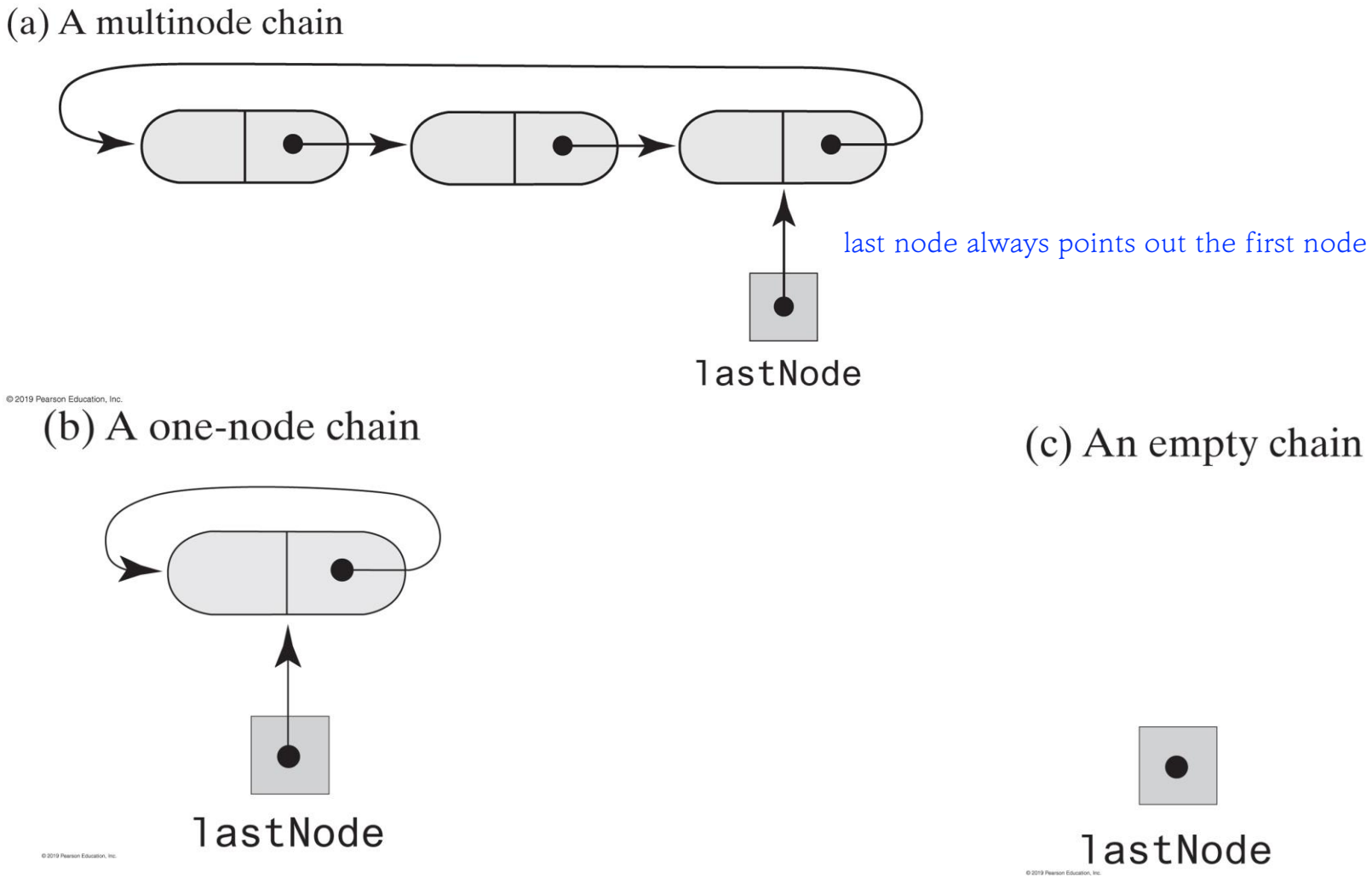


FIGURE 8-12 Circular linked chains, each with an external reference to its last node

Two-Part Circular Linked Chain

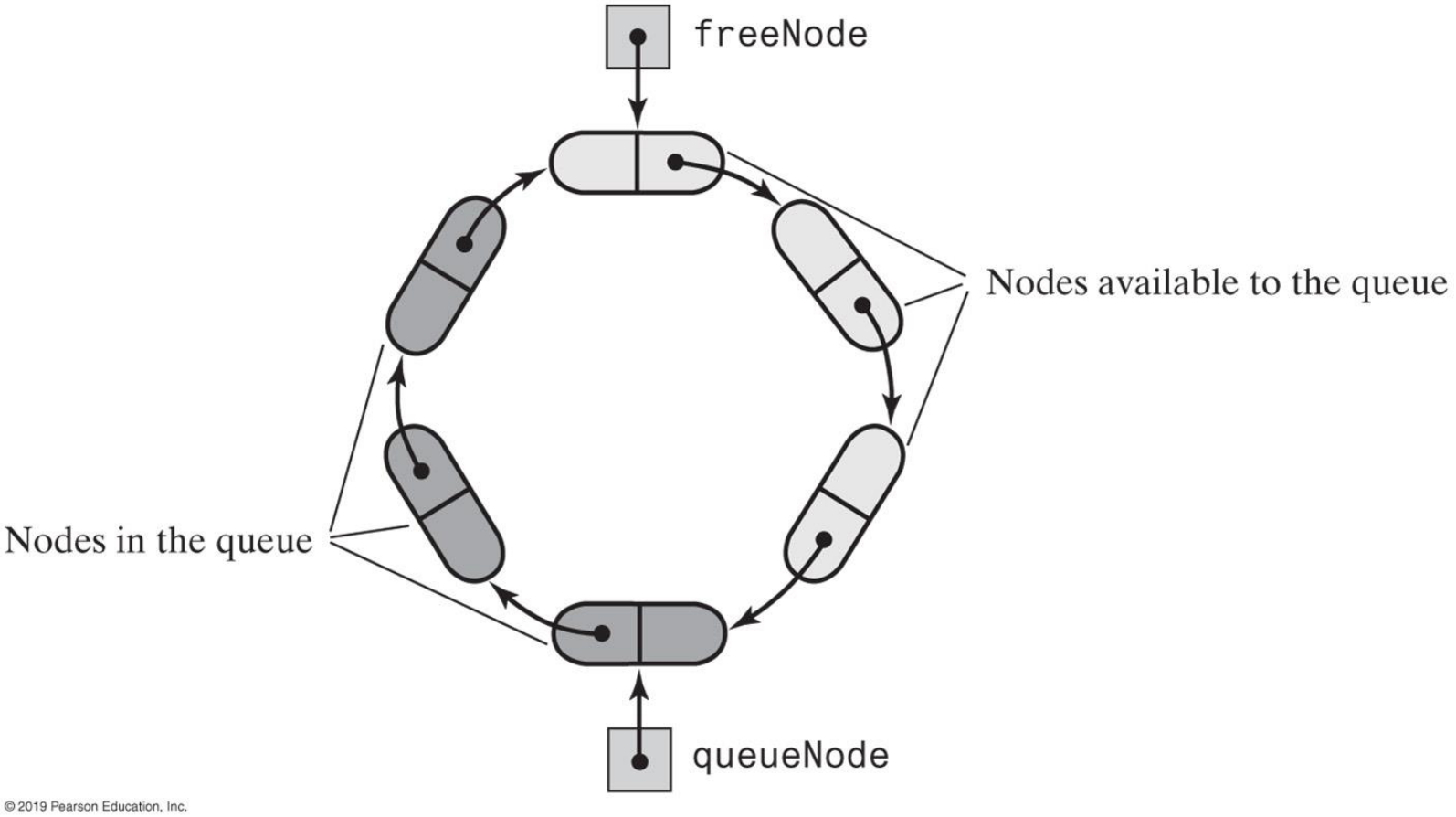


FIGURE 8-13 A two-part circular linked chain that represents both a queue and the nodes available to the queue

Two-Part Circular Linked Chain

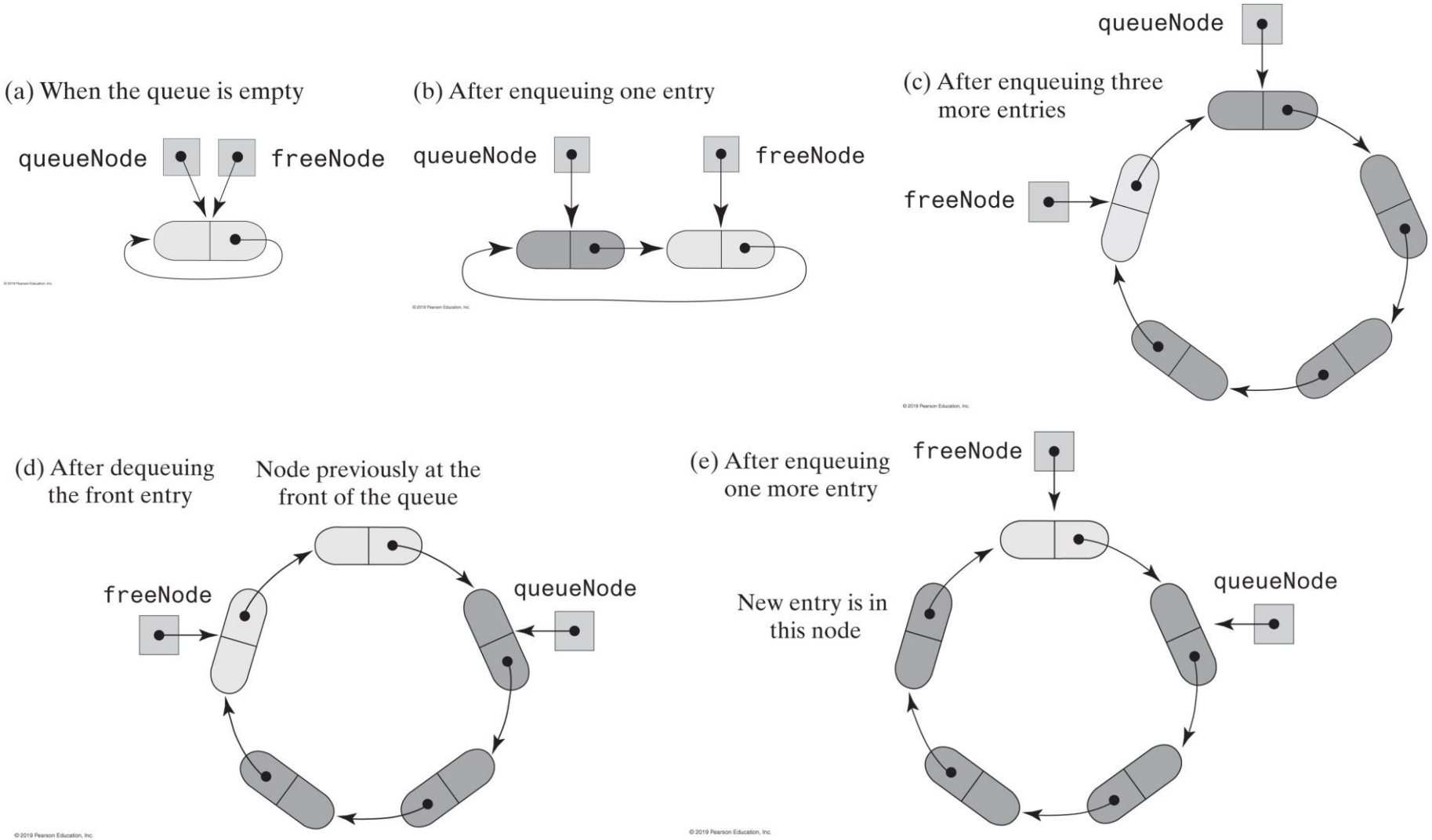


FIGURE 8-14 Various states of a two-part circular linked chain that represents a queue

Two-Part Circular Linked Chain

```
/**  A class that implements the ADT queue by using
  a two-part circular chain of linked nodes. */
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
     {
          //  < Implementation of the nine Node class god here. >
     } // end Node
} // end TwoPartCircularLinkedQueue
```

LISTING 8-3 An outline of a two-part circular linked implementation of the ADT queue

Two-Part Circular Linked Chain

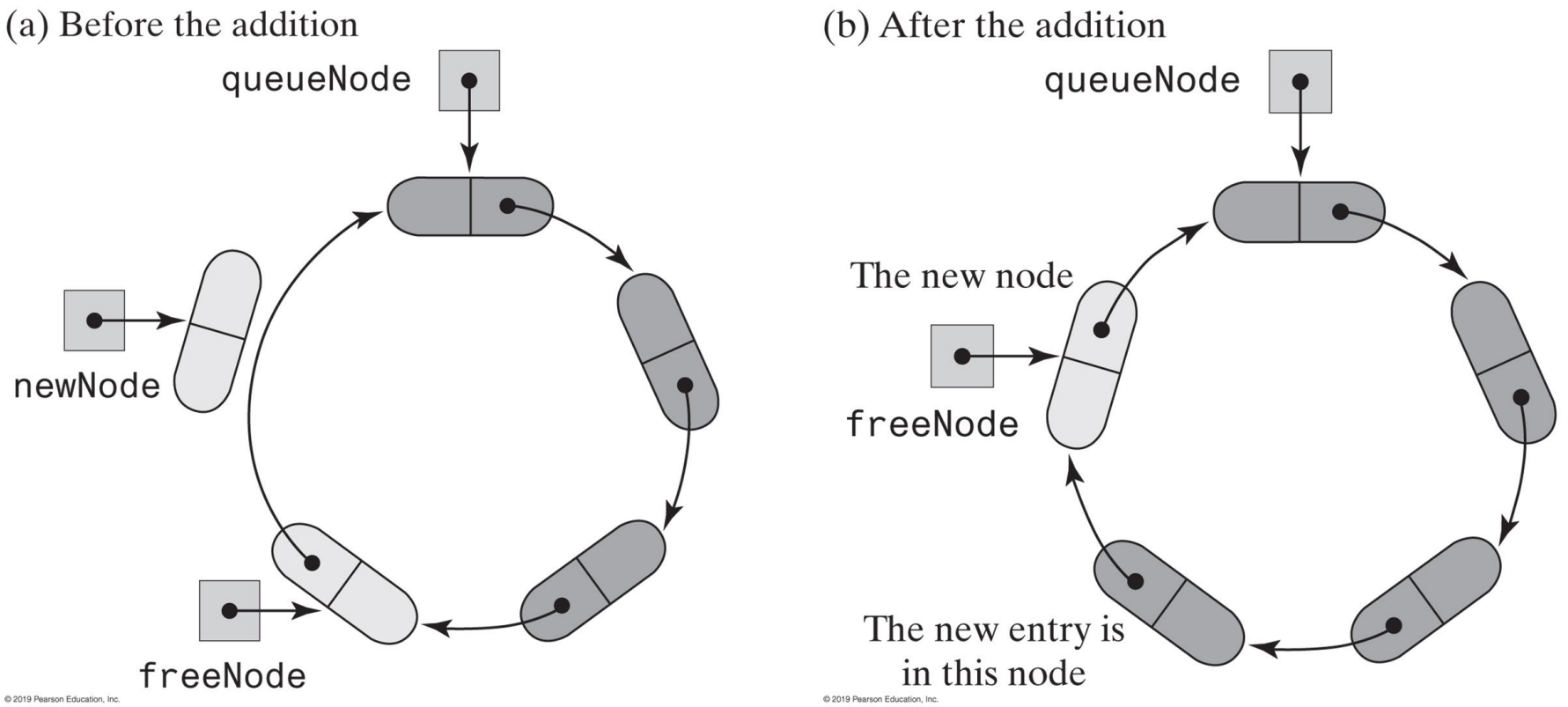


FIGURE 8-15a A two-part circular chain that requires a new node for an addition to a queue

Two-Part Circular Linked Chain

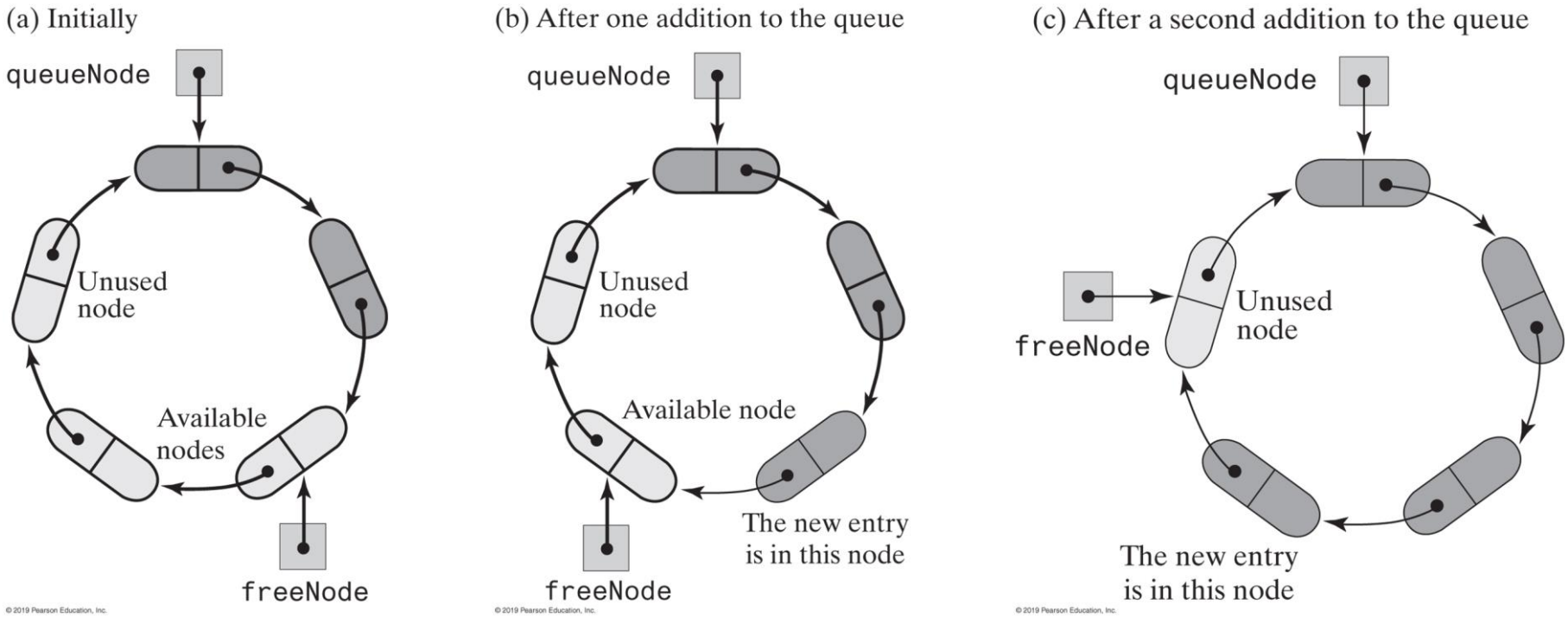


FIGURE 8-16 A two-part circular linked chain with nodes available for addition to a queue

Two-Part Circular Linked Chain

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

  if (isNewNodeNeeded())
  {
    // 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
   // Assertion: Queue is not empty
   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 isNewNodeNeeded()
{
  return queueNode == freeNode.getNextNode();
} // end isNewNodeNeeded
```

Methods `isEmpty` an `isNewNodeNeeded`

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

Doubly Linked Implementation of a Deque

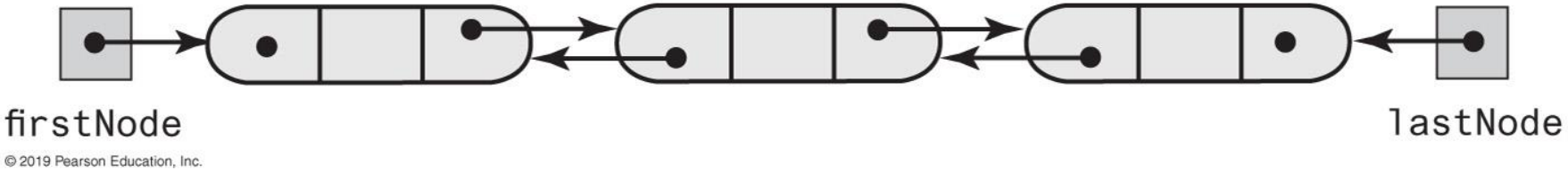


FIGURE 8-17 A doubly linked chain with head and tail references

Doubly Linked Implementation of a Deque

```
/**  A class that implements the a deque of objects by using
   a chain of doubly linked nodes. */
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
  {
      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 8-4 An outline of a linked implementation of the ADT deque

Doubly Linked Implementation of a Deque

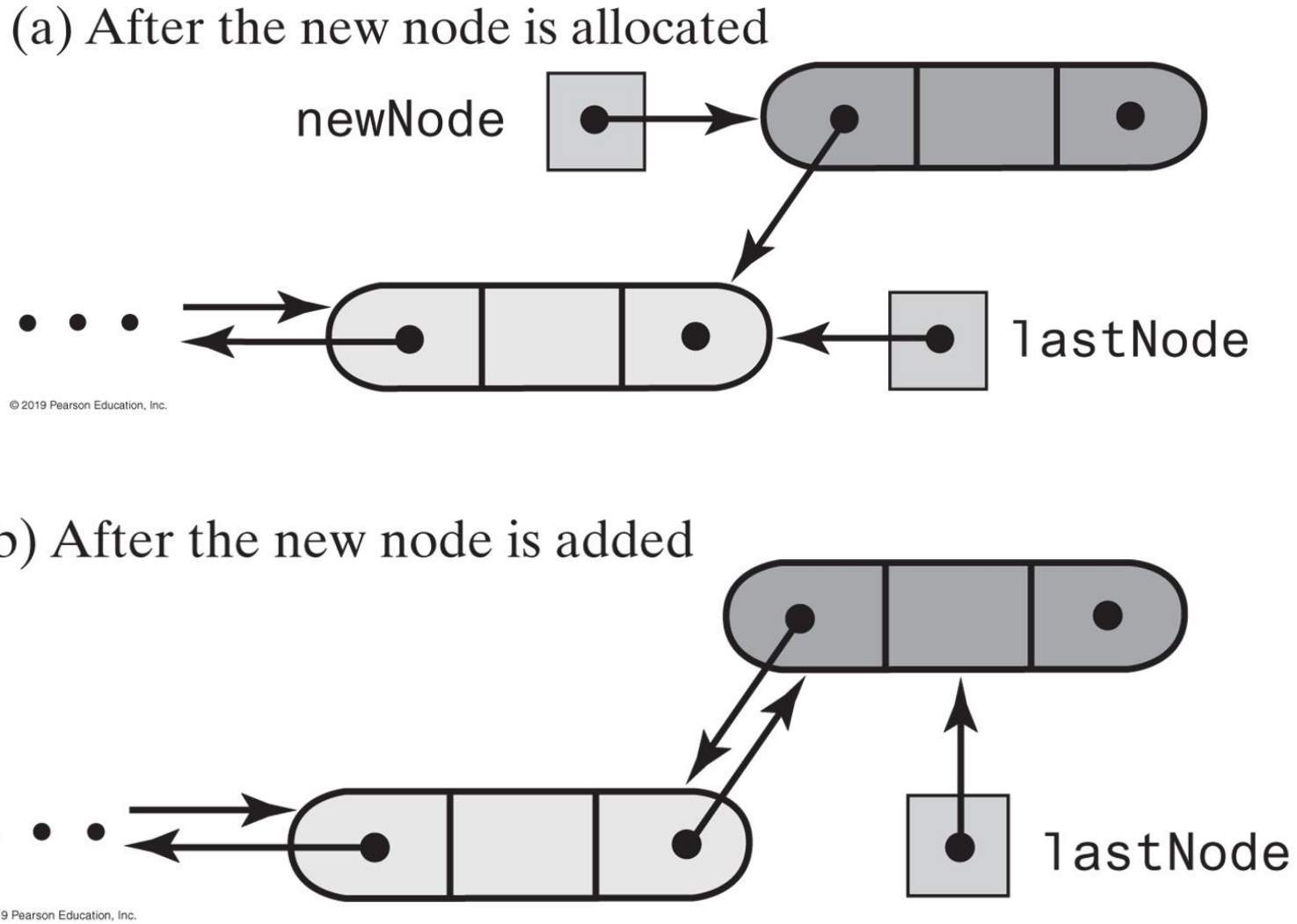


FIGURE 8-18 Adding to the back of a nonempty deque

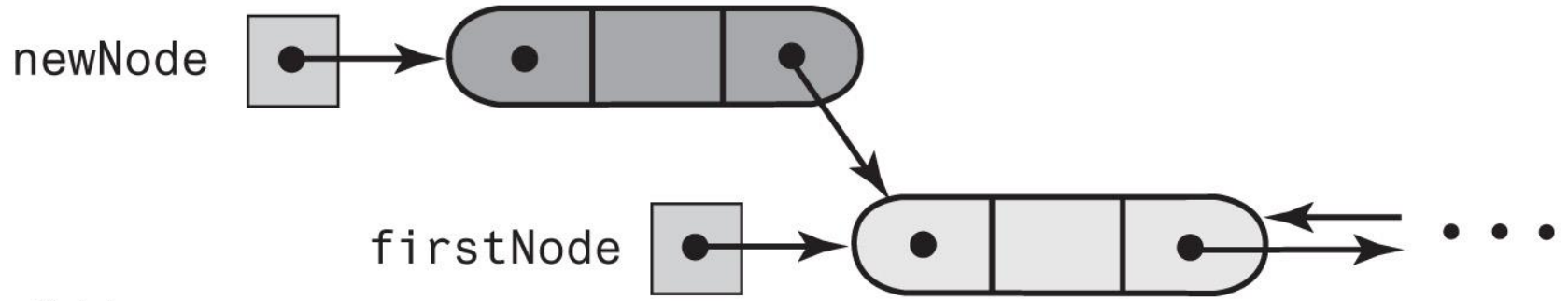
Doubly Linked Implementation of a Deque

```
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 8-4 An outline of a linked implementation of the ADT deque

FIGURE 8-19a Adding to the front of a nonempty deque

(a) After the new node is allocated



(b) After the new node is added to the front

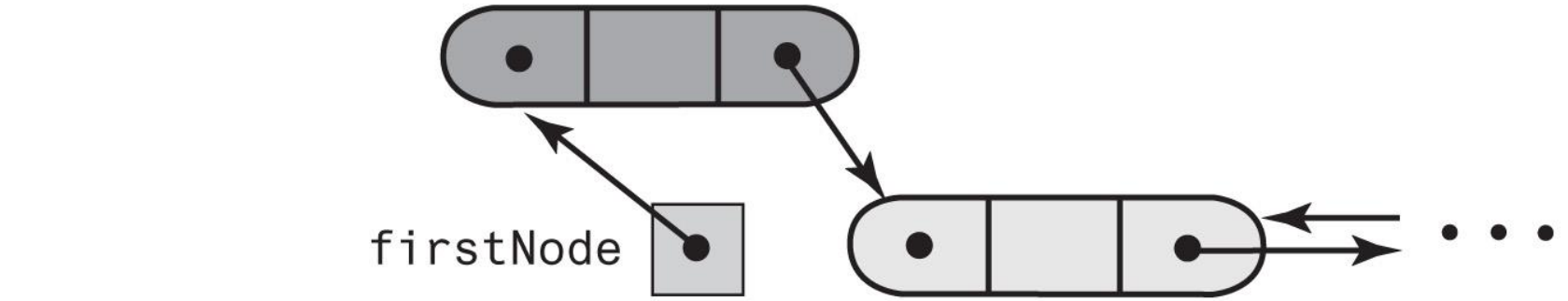


FIGURE 8-19 Adding to the front of a nonempty deque

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
```

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

Doubly Linked Implementation of a Deque

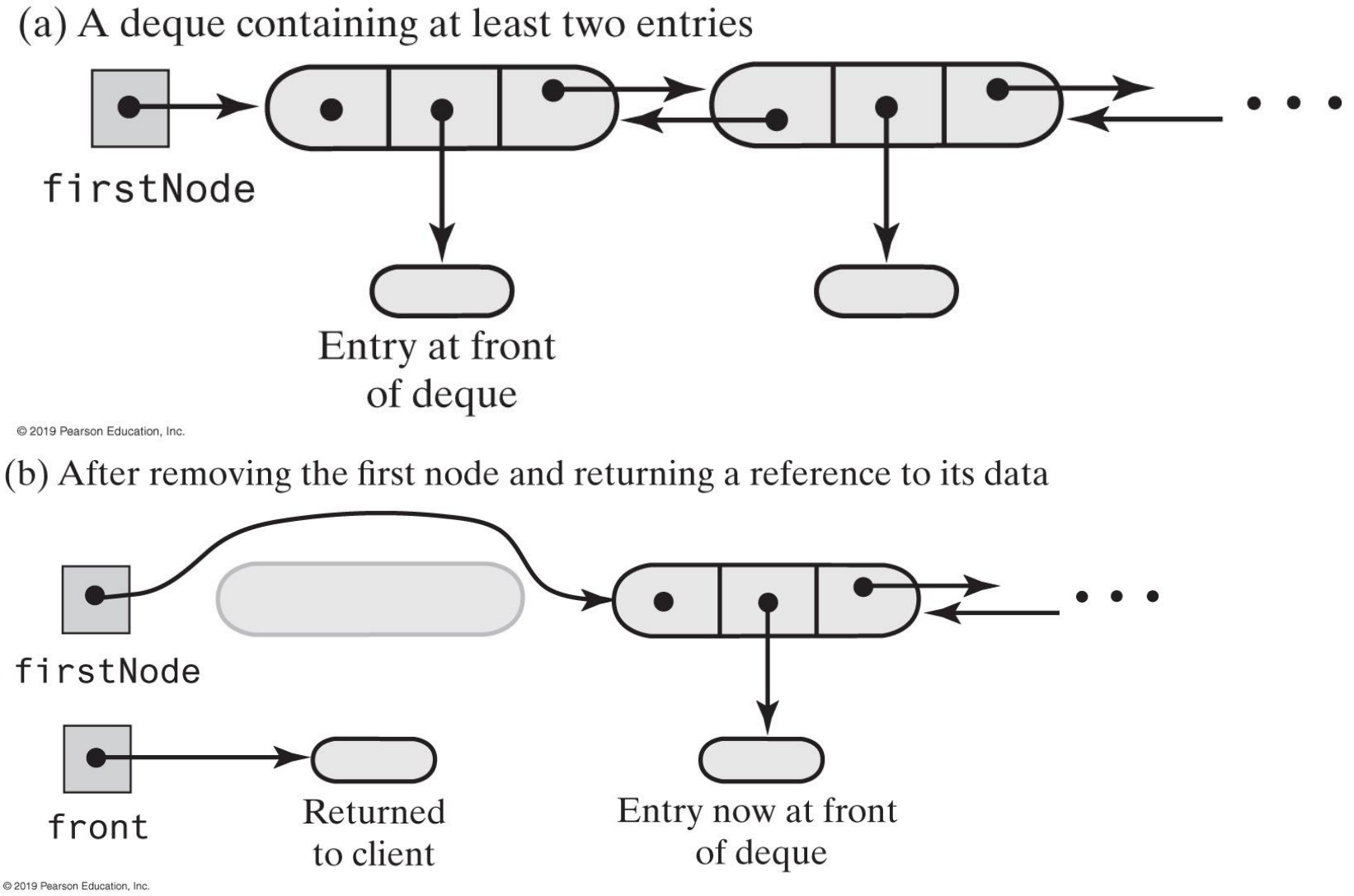


FIGURE 8-20 Removing the front of a deque containing at least two entries

Doubly Linked Implementation of a Deque

```
public T removeFront()
{
  T front = getFront();  // Might throw EmptyQueueException
  // Assertion: 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
  // Assertion: lastNode != null
  lastNode = lastNode.getPreviousNode();

  if (lastNode == null)
    firstNode = null;
  else
    lastNode.setNextNode(null);
  } // end if

  return back;
} // end removeBack
```

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

Possible Implementations of a Priority Queue

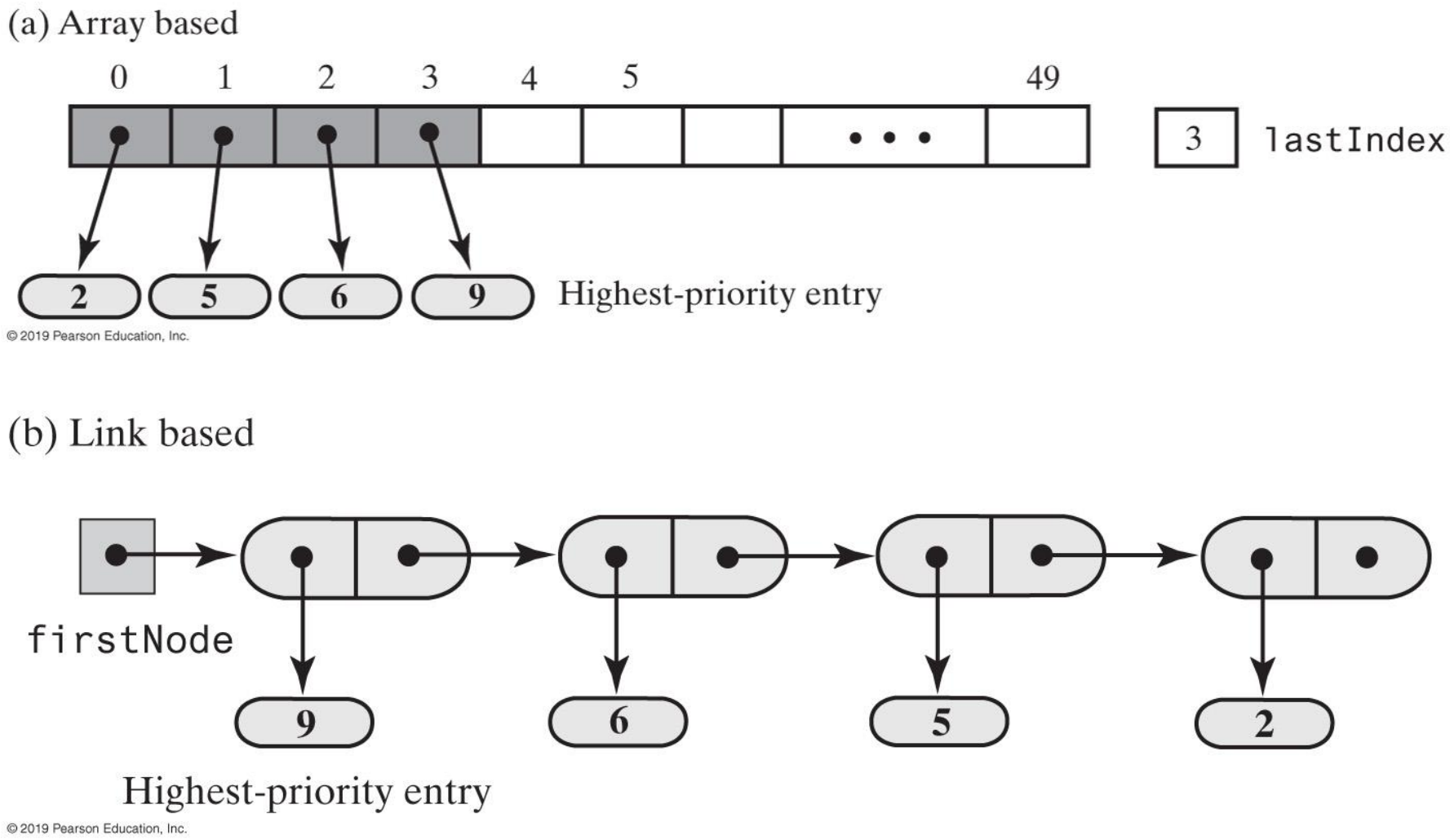


FIGURE 8-21 Two possible implementations of a priority queue

End

Chapter 8