Lecture 14: Queues, Deques, and Priority Queues

CS 0445: Data Structures

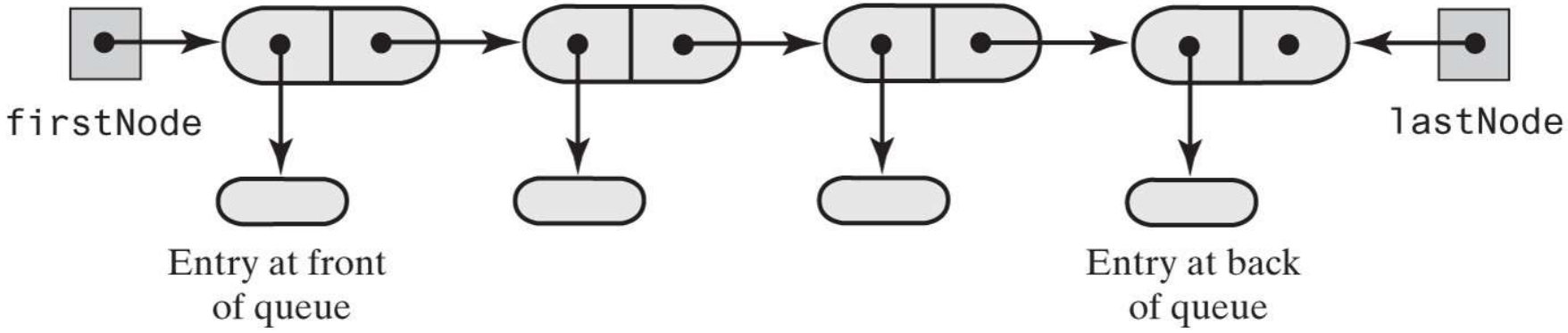
Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Oct 3, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

Linked Implementation of a Queue

- A chain of linked nodes that implements a queue

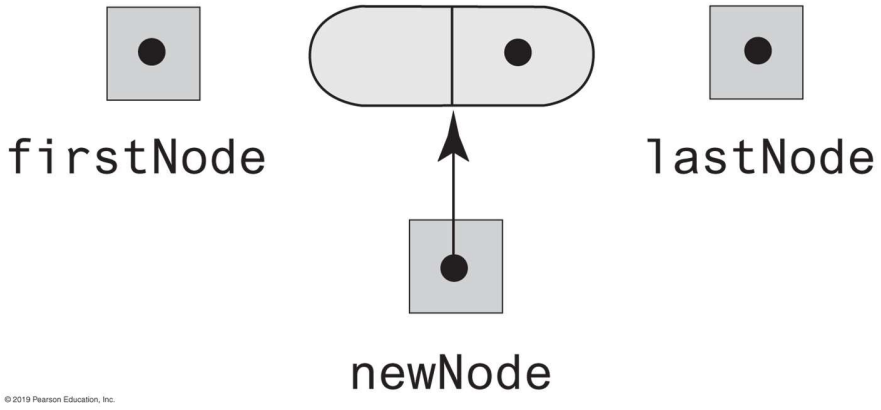


Linked Implementation of a Queue

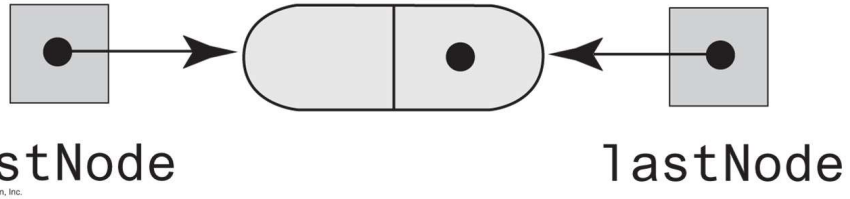
```
/** A class that implements a queue of objects by using
  a chain of linked nodes that has both head and tail references. */
public final class LinkedQueue<T> implements QueueInterface<T>
{
  private Node firstNode; // References node at front of queue
  private Node lastNode;  // References node 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
```

Linked Implementation of a Queue

(a) Before



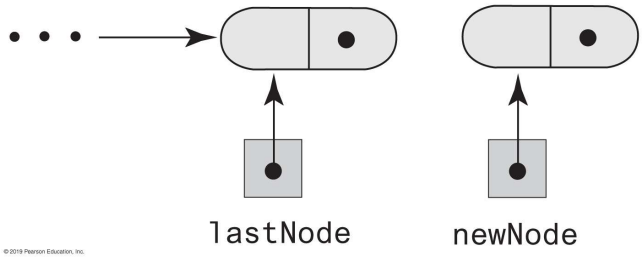
(b) After



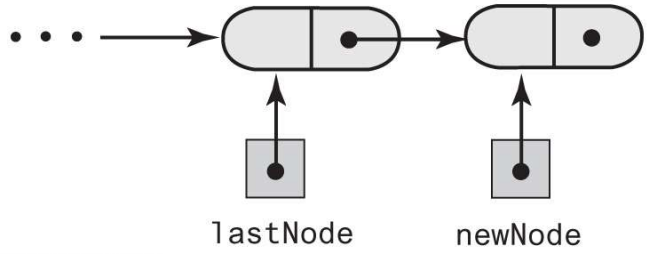
Linked Implementation of a Queue

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

(a) Before the addition

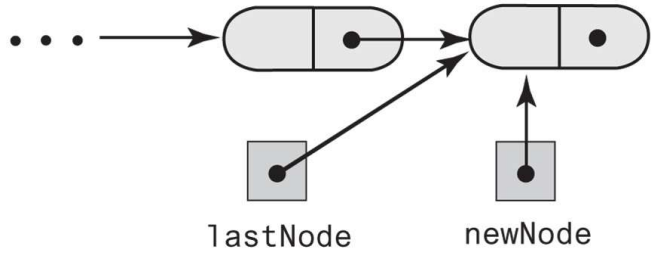


(b) During the addition



After executing
`lastNode.setNextNode(newNode);`

(c) After the addition



After executing
`lastNode = newNode;`

Linked Implementation of a Queue

- The definition of `enqueue` Performance is O(1)

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

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

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

Linked Implementation of a Queue

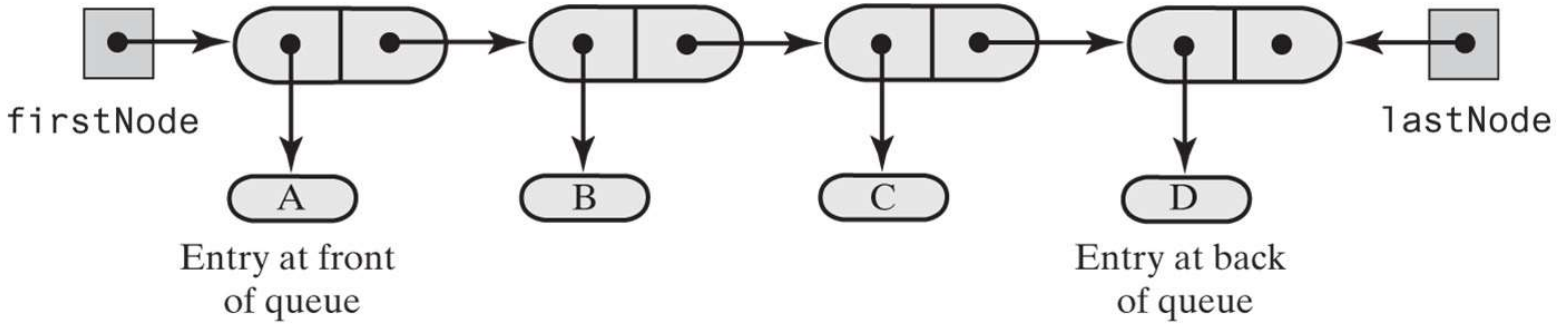
- Retrieving the front entry

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

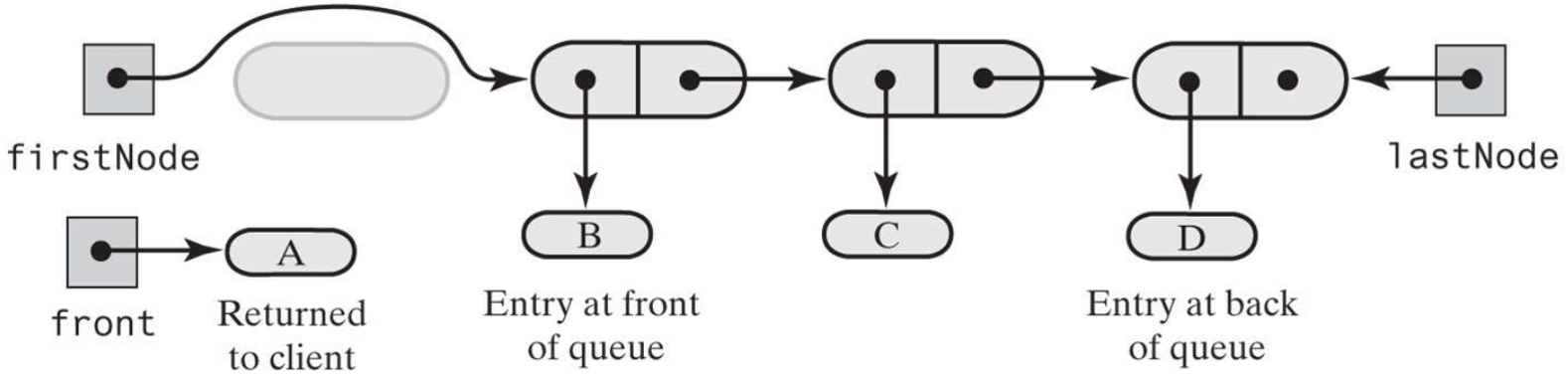
Linked Implementation of a Queue

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

(a) A queue of more than one entry



(b) After removing the entry at the queue's front



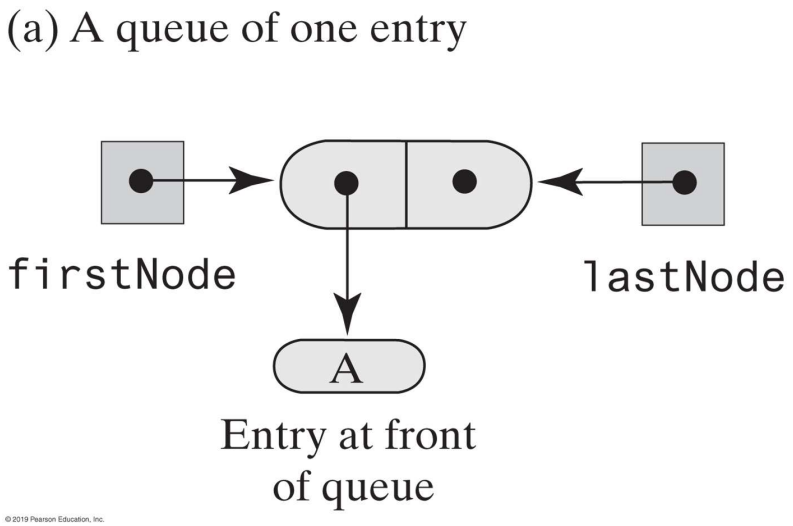
Linked Implementation of a Queue

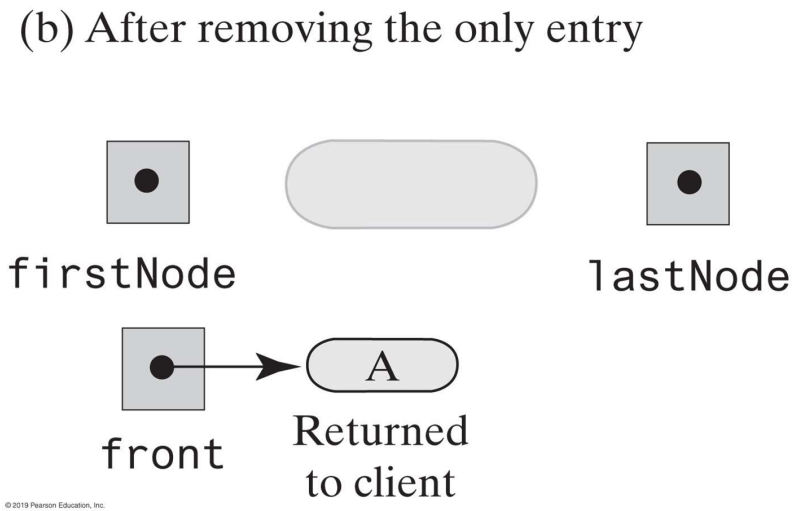
- Removing the front entry

```
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
```

Linked Implementation of a Queue

- Before and after removing the only entry from a queue





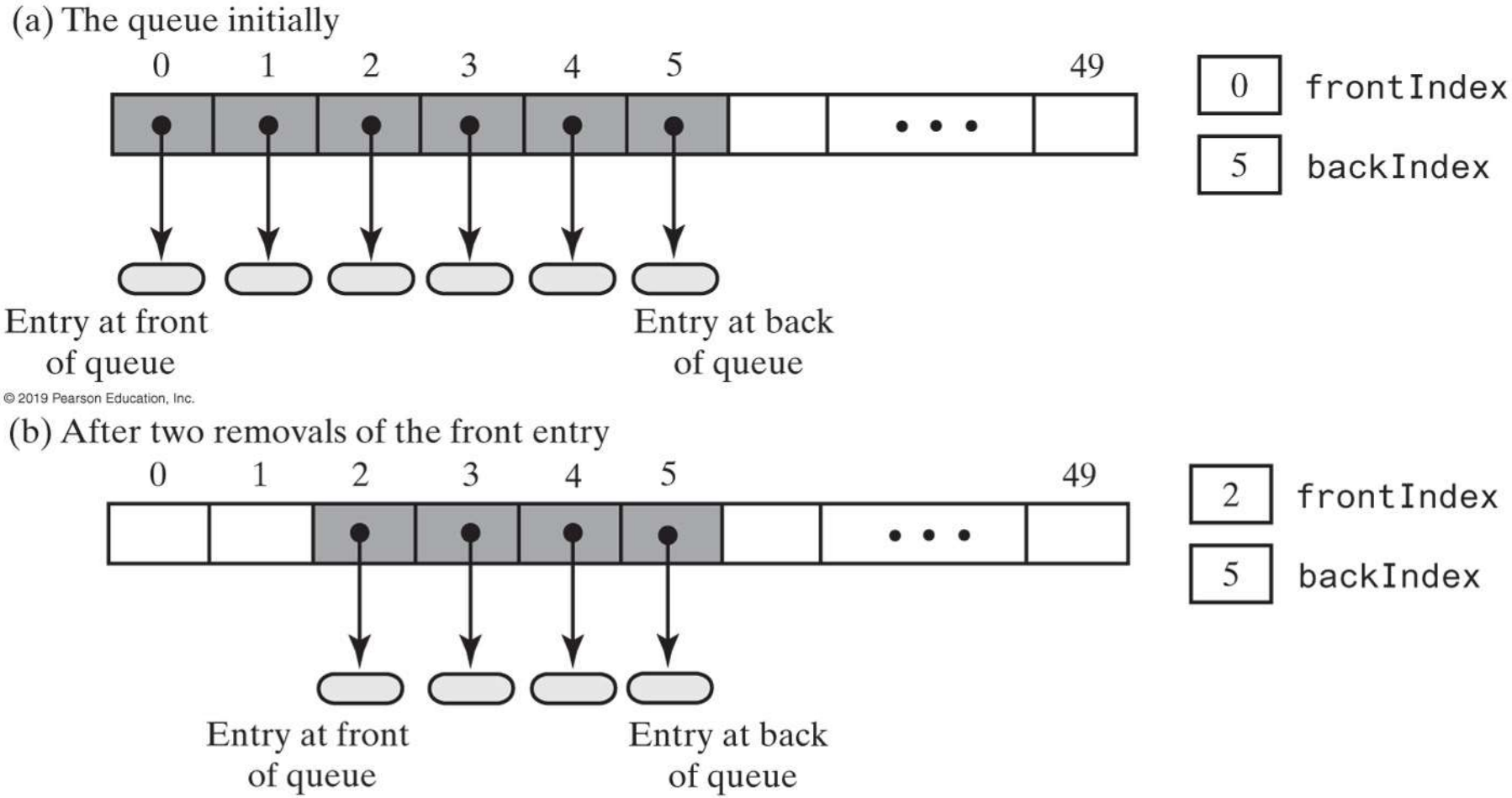
Linked Implementation of a Queue

- Public methods `isEmpty` and `clear`

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

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

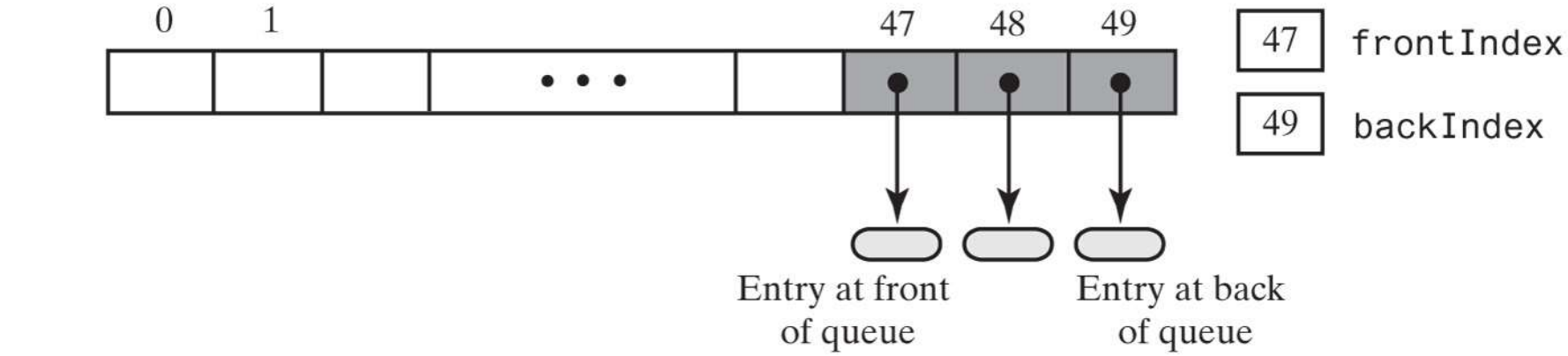
Array-Based Implementation of a Queue: Circular Array

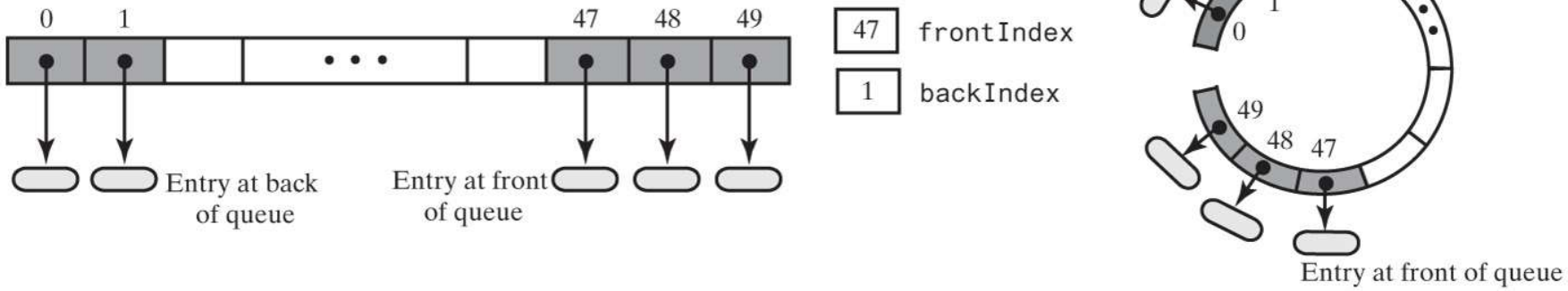
- An array that represents a queue without moving any entries during additions and removals



Circular Array

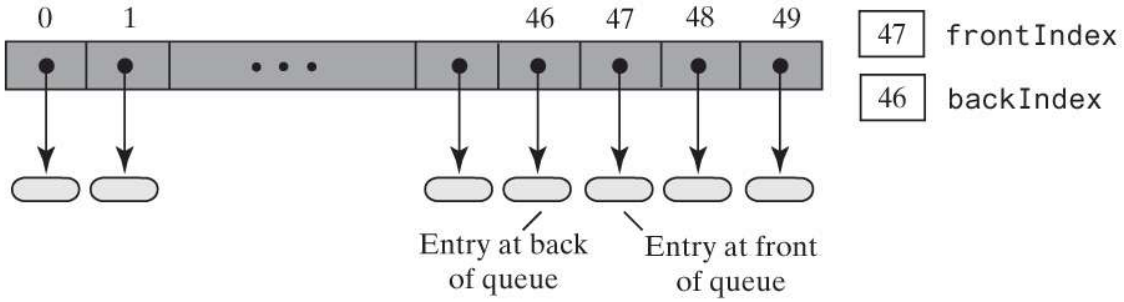
(c) After several more additions and removals

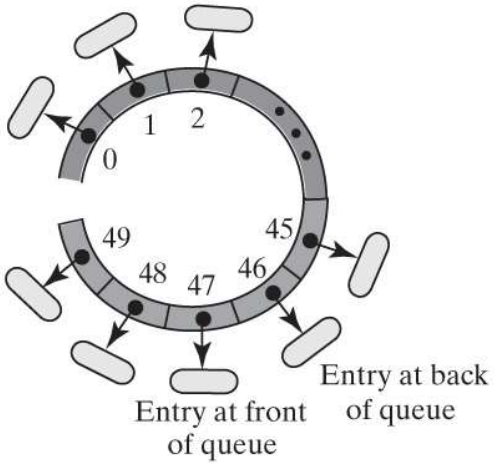




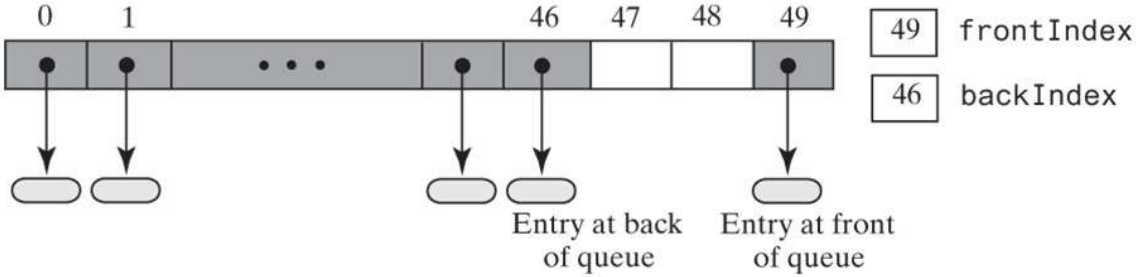
Circular Array

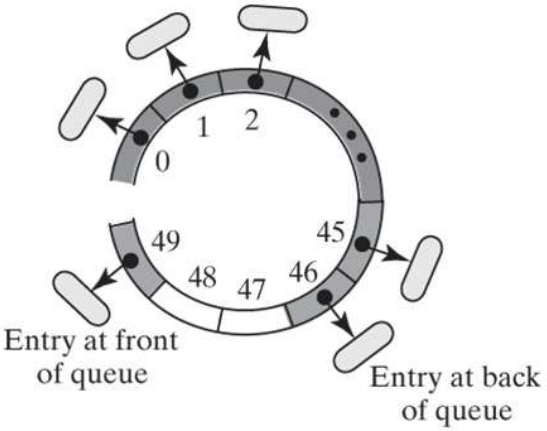
- (a) After adding more entries to the queue in Figure 8-7 until it is full





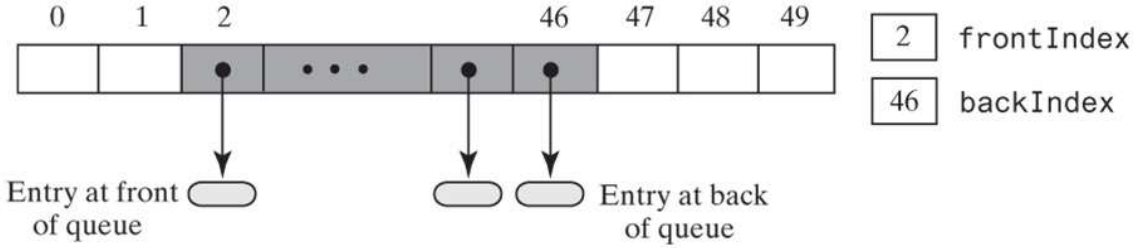
(b) After removing two entries

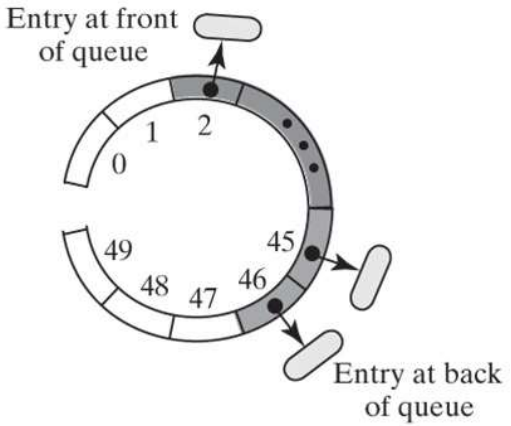




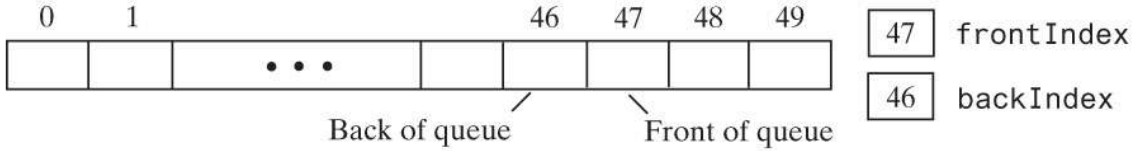
Circular Array

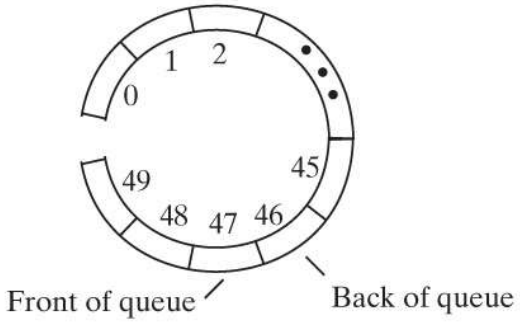
- (c) After removing three more entries





(e) After removing the remaining entry, making the queue empty





Circular Array with One Unused Location

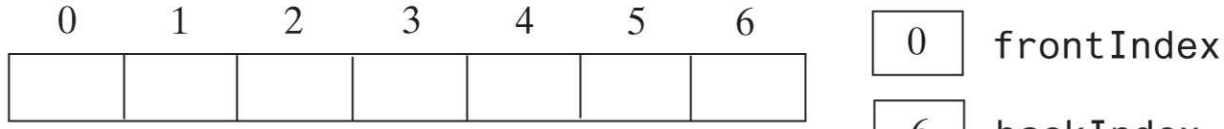
- Retrieving the front entry

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

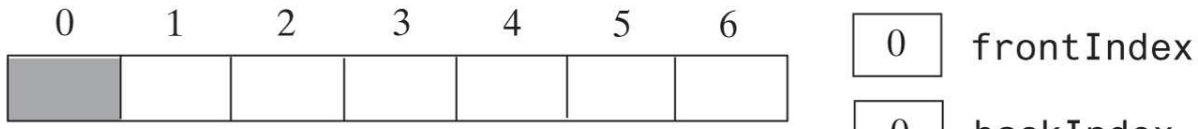
Circular Array (Part 1)

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

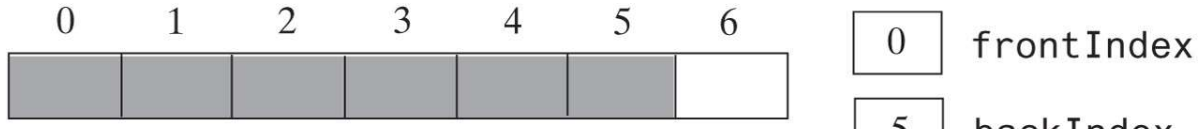
(a) Initially, the queue is empty



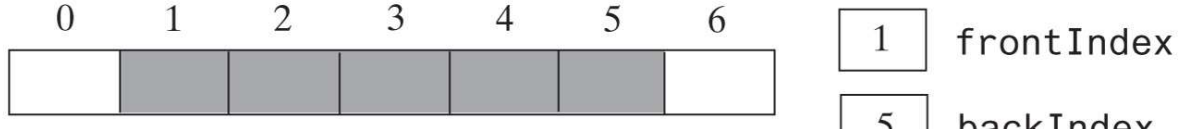
(b) After enqueuing one entry



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



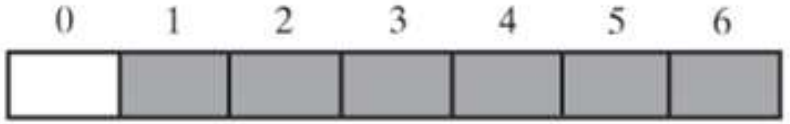
(d) After dequeuing an entry



Circular Array (Part 2)

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

(e) After enqueuing an entry, the queue becomes full again



1 frontIndex

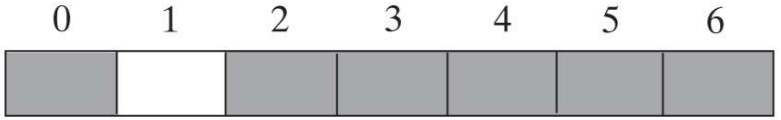
6 backIndex

(f) After dequeuing an entry

2 frontIndex

6 backIndex

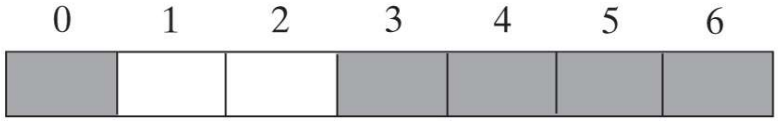
(g) After enqueuing an entry, the queue is full



2 frontIndex

0 backIndex

(h) After dequeuing an entry



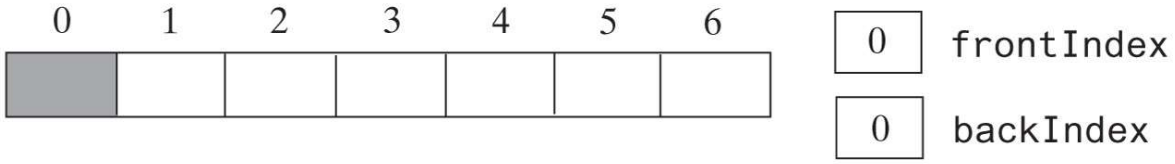
3 frontIndex

0 backIndex

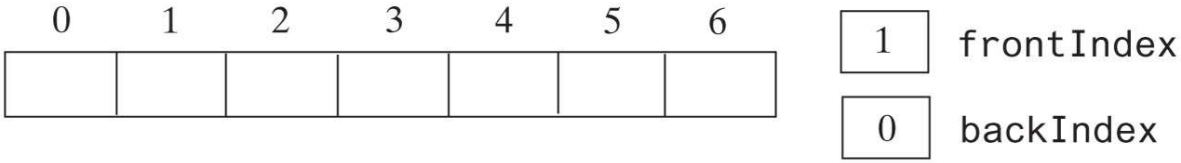
Circular Array (Part 3)

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

(i) After dequeuing all but one entry



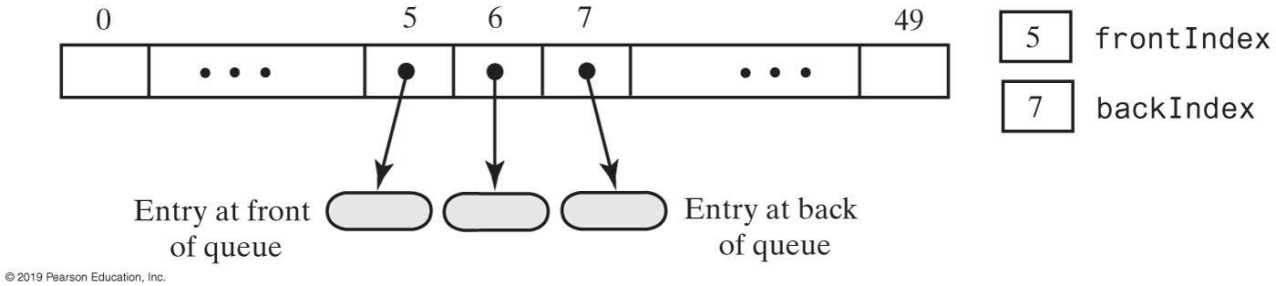
(j) After dequeuing the remaining entry, the queue is now empty



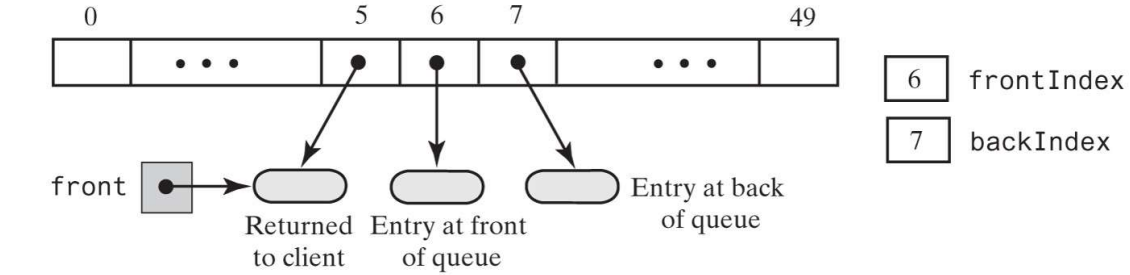
Circular Array with One Unused Location

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

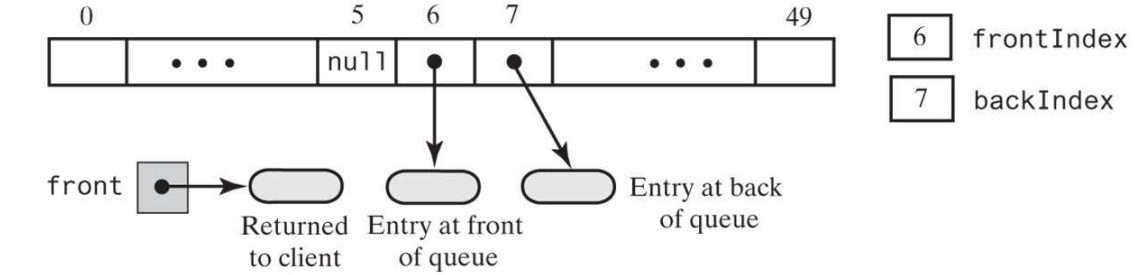
(a) Initially



(b) After dequeuing the front entry by incrementing `frontIndex`



(c) After dequeuing the front entry by incrementing `frontIndex` and setting `queue[frontIndex]` to `null`



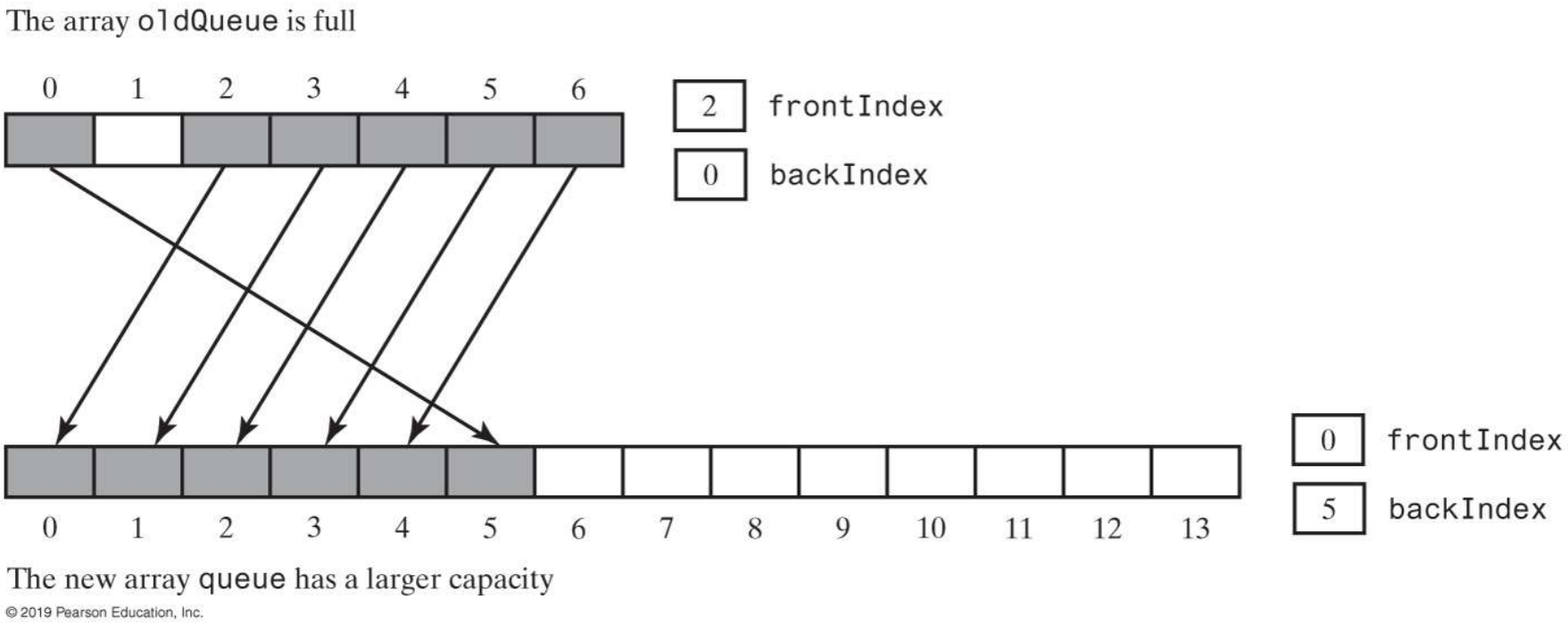
Circular Array with One Unused Location

- Implementation of `dequeue`

```
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
```

Circular Array with One Unused Location

- Doubling the size of an array-based queue



Circular Array with One Unused Location

- Definition of `ensureCapacity`

```
// 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
```

Circular Array with One Unused Location

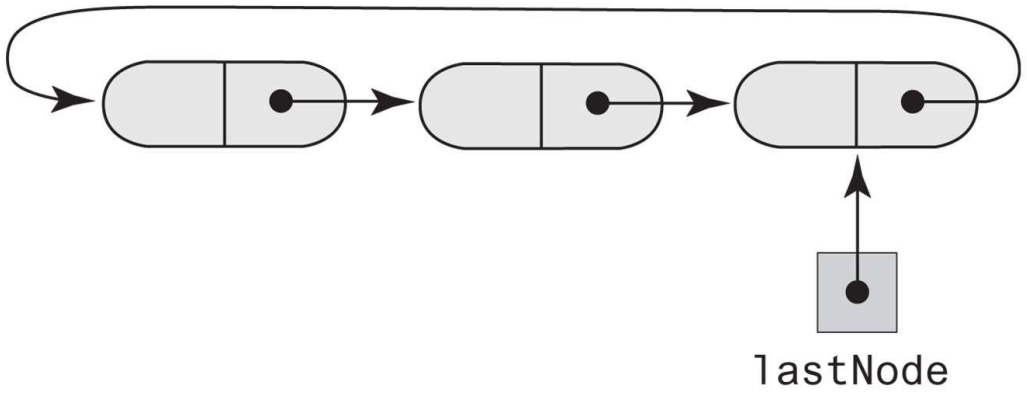
- Implementation of `isEmpty`

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

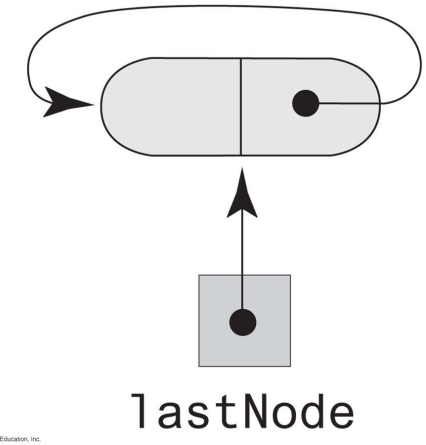
Circular Linked Implementations of a Queue

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

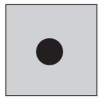
(a) A multinode chain



(b) A one-node chain

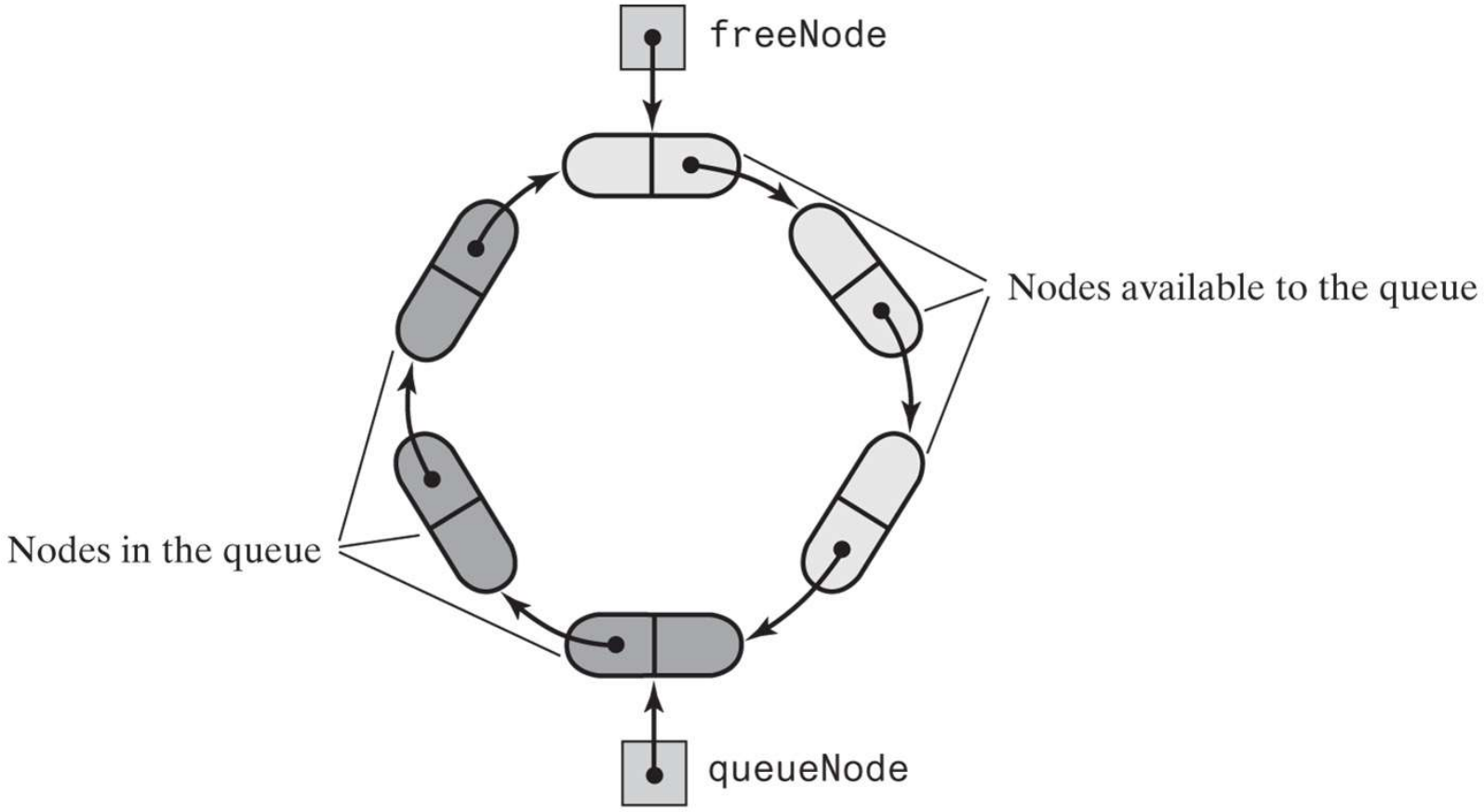


(c) An empty chain



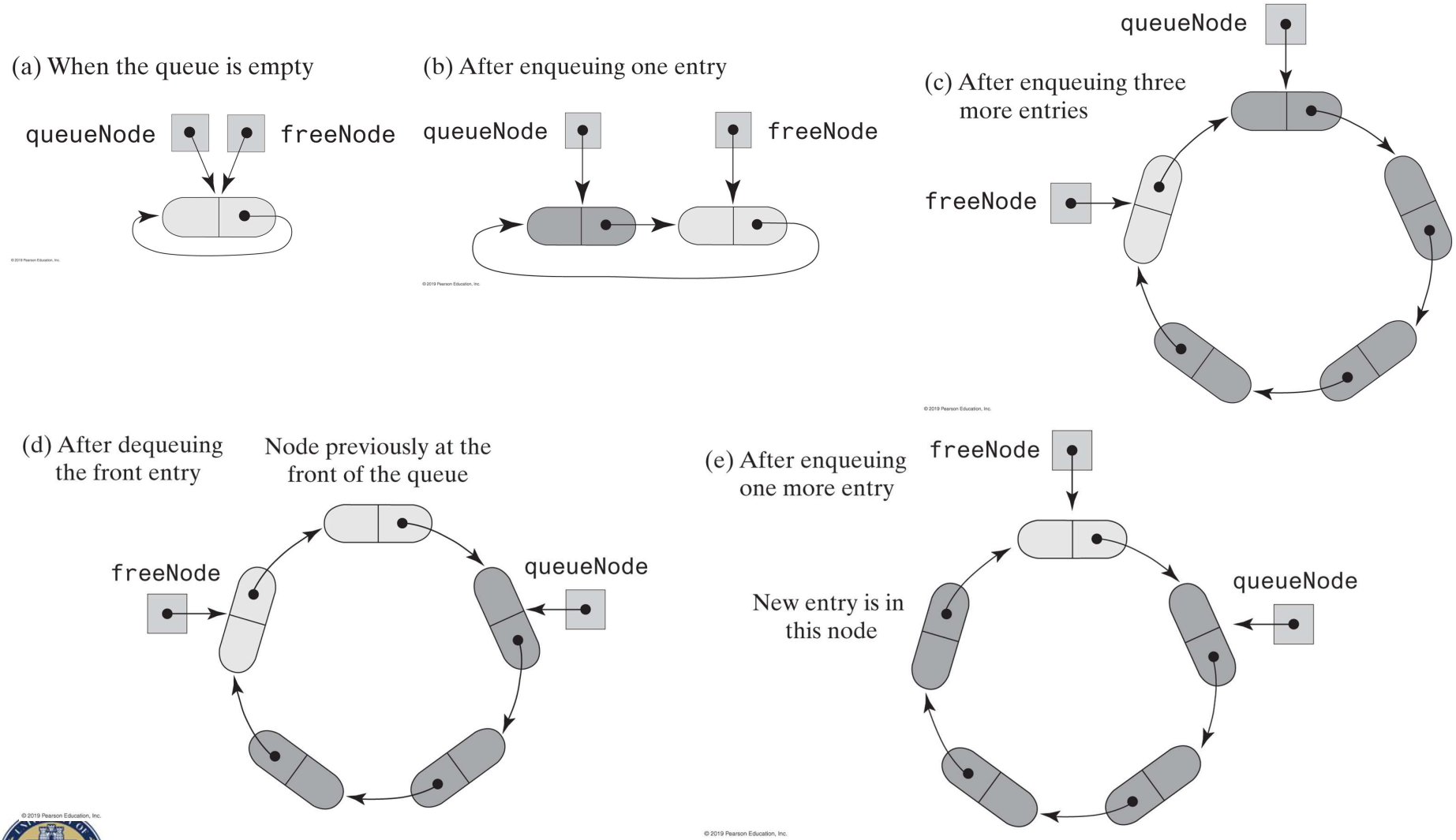
Two-Part Circular Linked Chain

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



Two-Part Circular Linked Chain

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



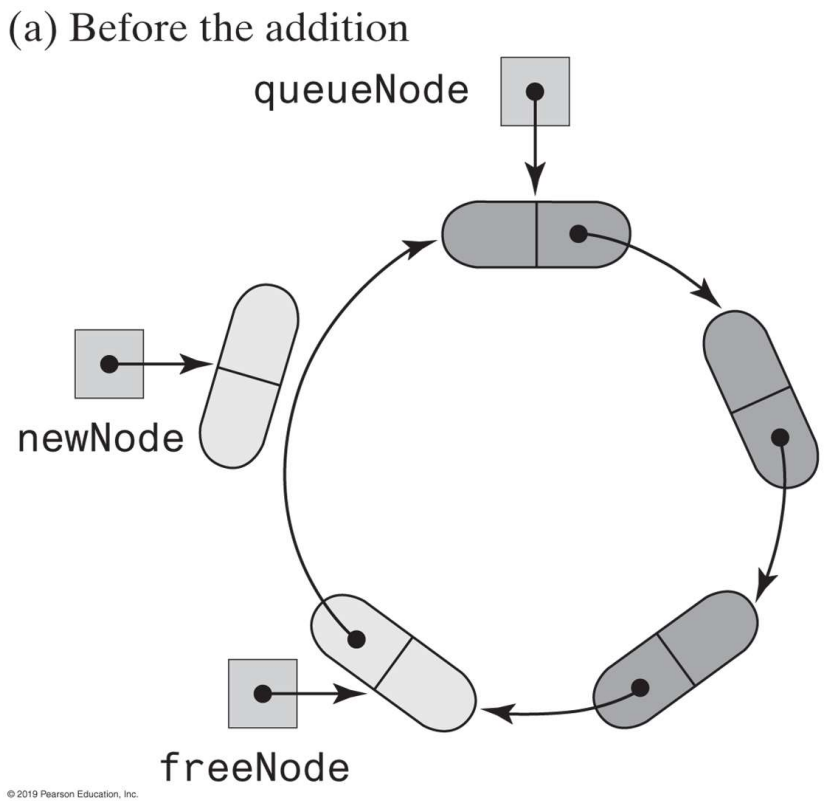
Two-Part Circular Linked Chain

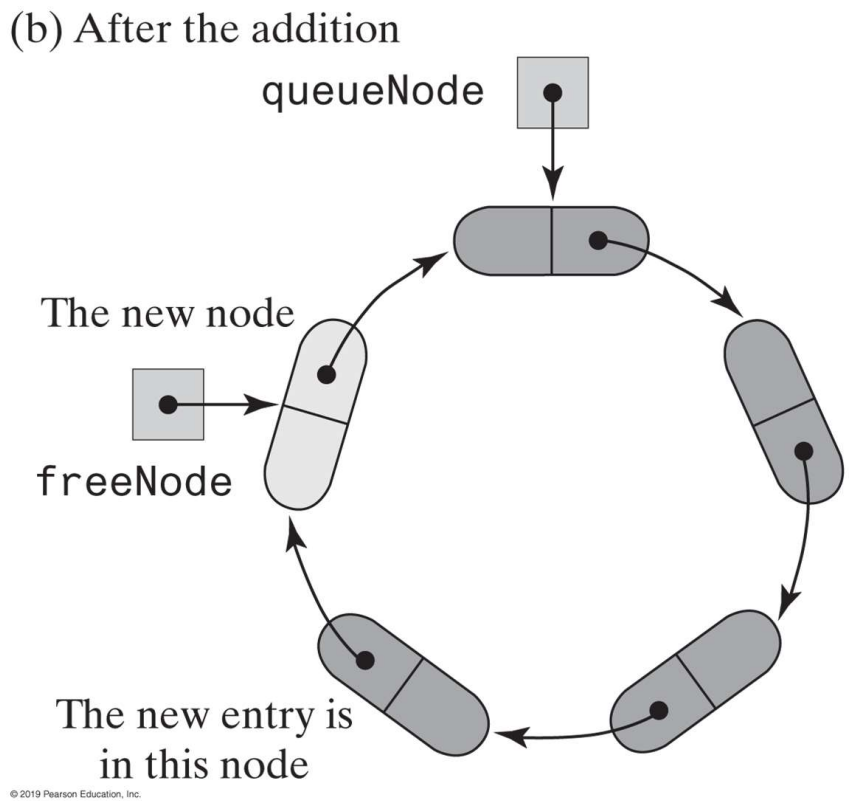
An outline of a two-part circular linked implementation of the ADT queue

```
/**  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
```

Two-Part Circular Linked Chain

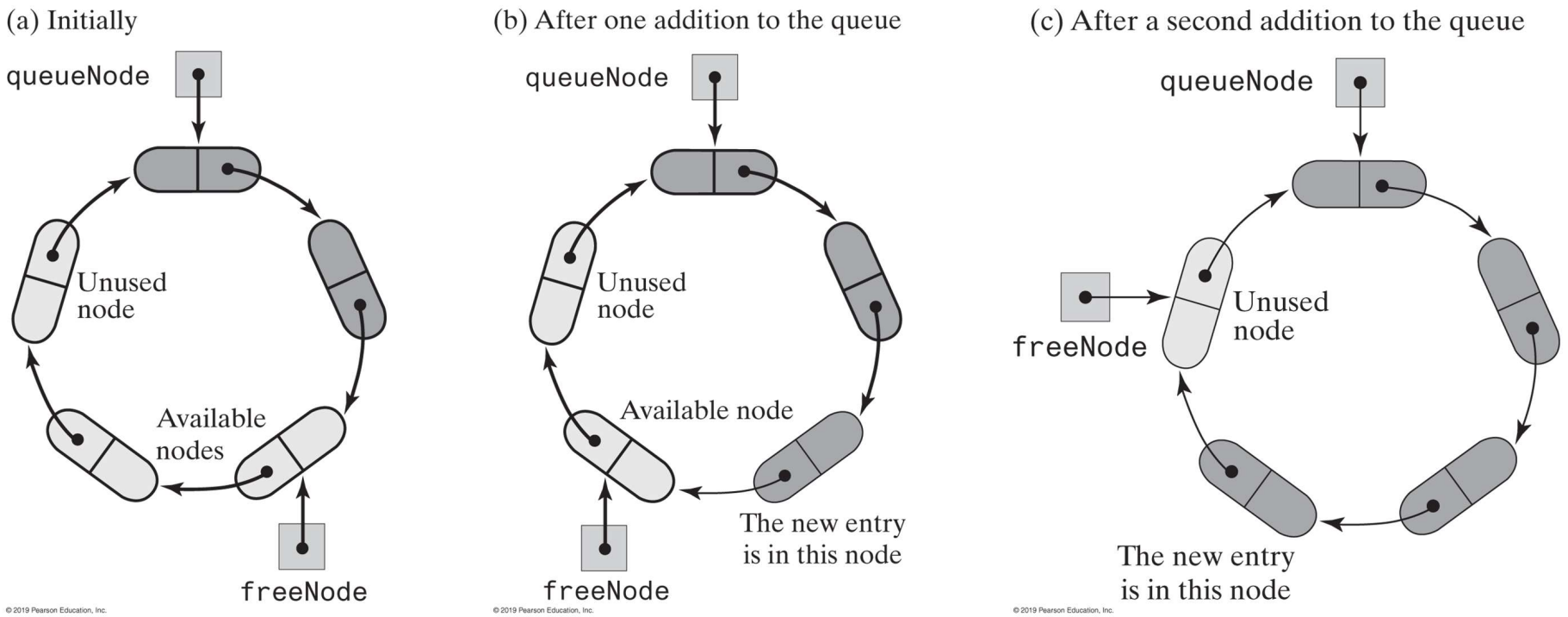
A two-part circular chain that requires a new node for an addition to a queue





Two-Part Circular Linked Chain

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



Two-Part Circular Linked Chain

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

```
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
```

Two-Part Circular Linked Chain

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

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

Two-Part Circular Linked Chain

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

```
public T dequeue()
{
  T front = getFront();  // Might throw EmptyQueueException
  // Assertion: Queue is not empty
  queueNode.setData(null);
  queueNode = queueNode.getNextNode();

  return front;
} // end dequeue
```

Two-Part Circular Linked Chain

- Methods `isEmpty` an `isNewNodeNeeded`

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

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

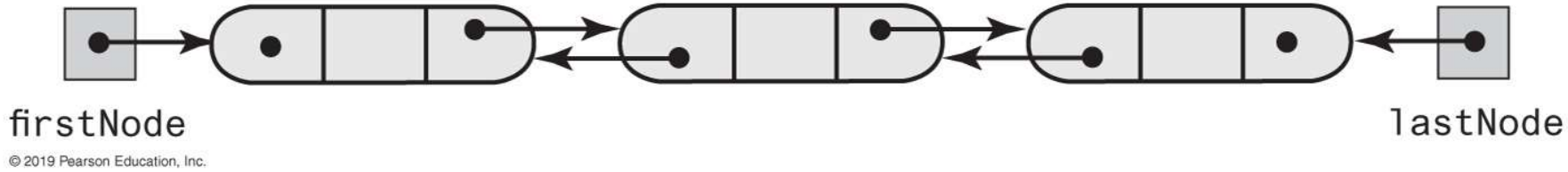
Java Class Library: The Class `AbstractQueue`

- Methods in this interface

```
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()
```

Doubly Linked Implementation of a Deque

- A doubly linked chain with head and tail references

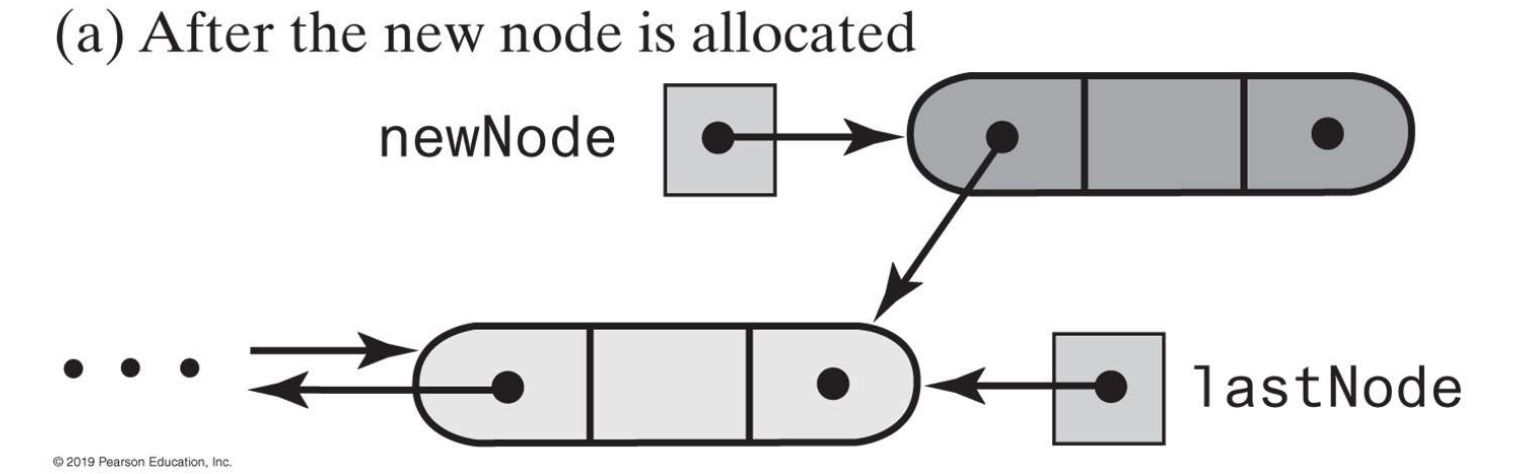


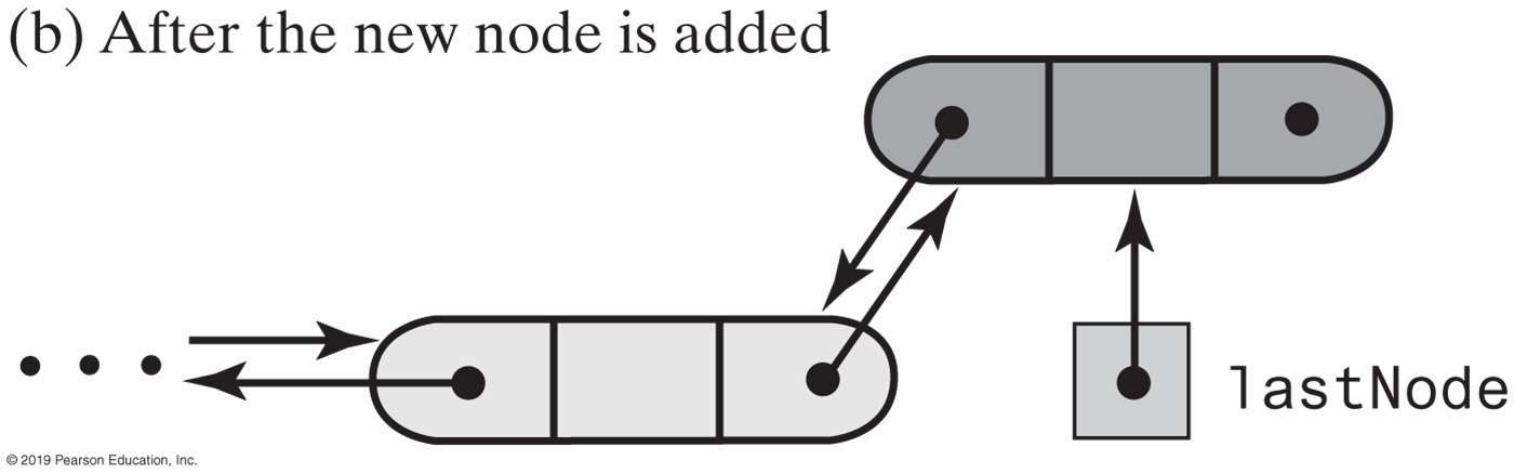
Doubly Linked Implementation of a Deque

```
/**  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
```

Doubly Linked Implementation of a Deque

- Adding to the back of a nonempty deque





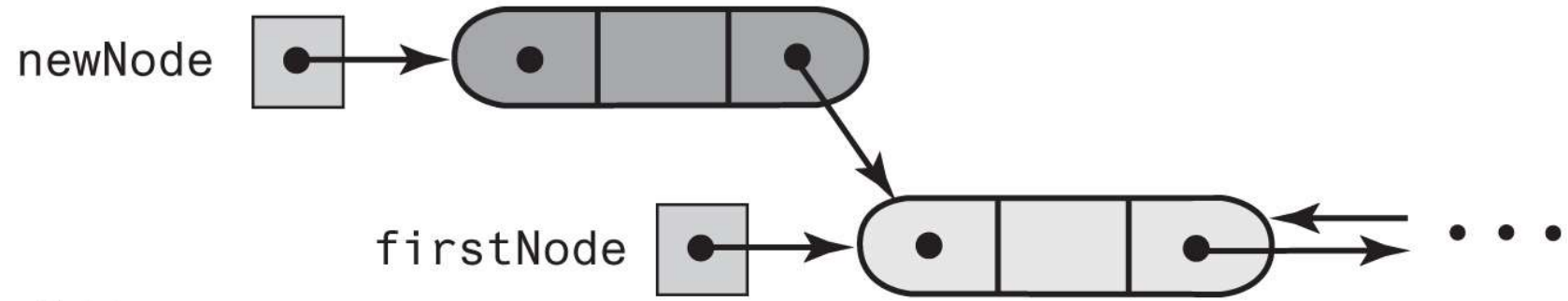
Doubly Linked Implementation of a Deque

An outline of a linked implementation of the ADT deque

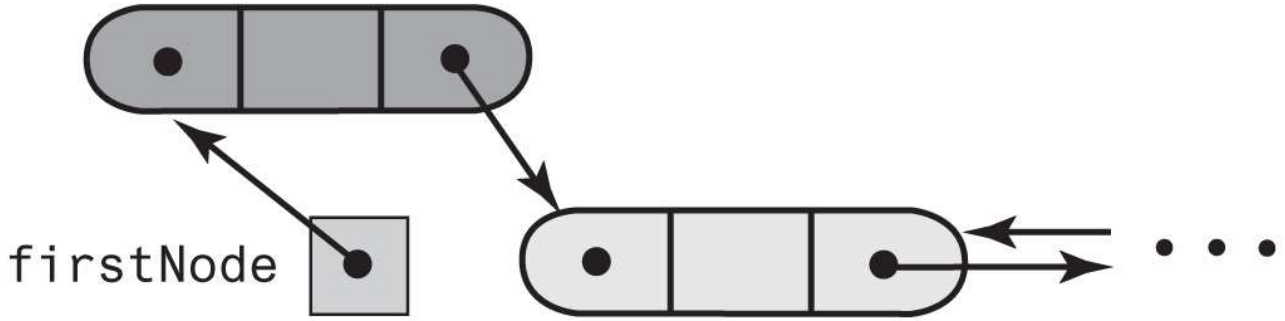
```
public void addToBack(T newEntry)
{
  DLNode newNode = new DLNode(lastNode, newEntry, null);

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

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

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



Doubly Linked Implementation of a Deque

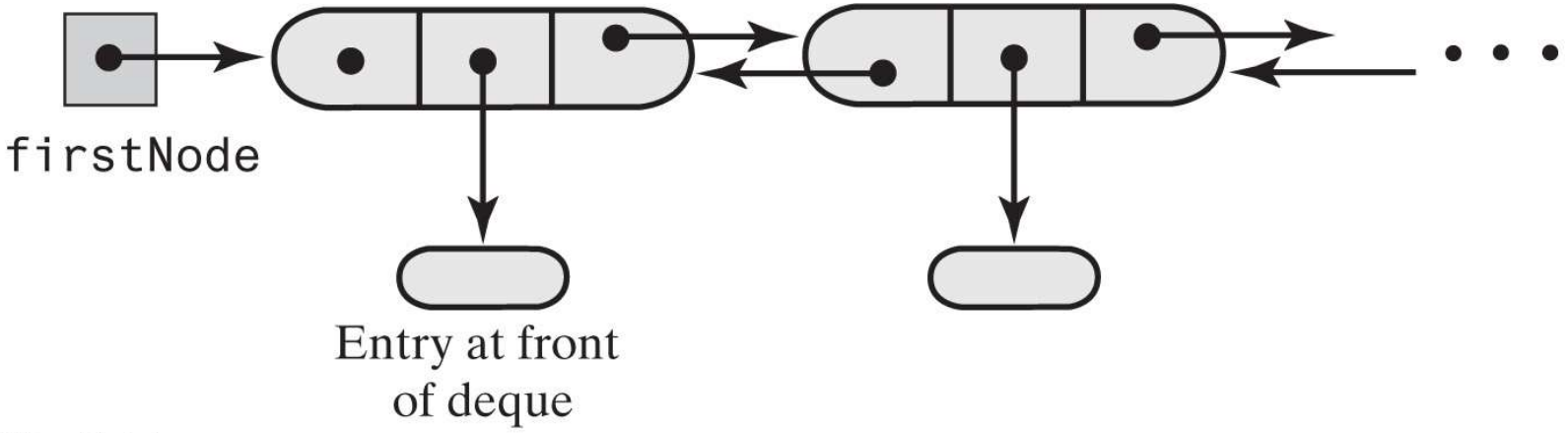
- Implementation of `addToFront`, an O(1) operation.

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

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

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

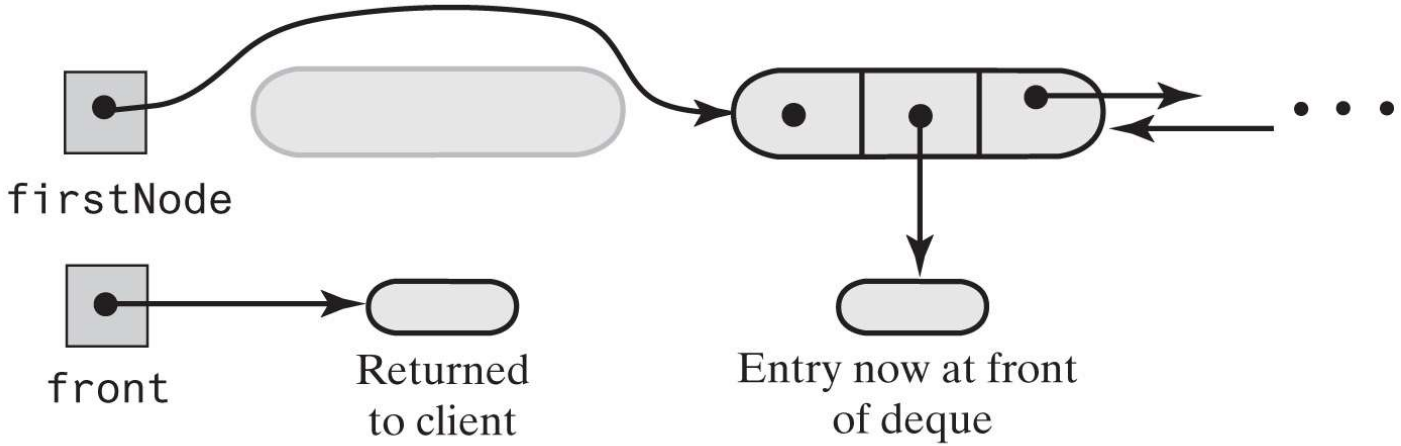
Doubly Linked Implementation of a Deque

Removing the front of a deque containing at least two entries

(a) A deque containing at least two entries



(b) After removing the first node and returning a reference to its data



Doubly Linked Implementation of a Deque

- Implementation of `removeFront`.

```
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
```

Doubly Linked Implementation of a Deque

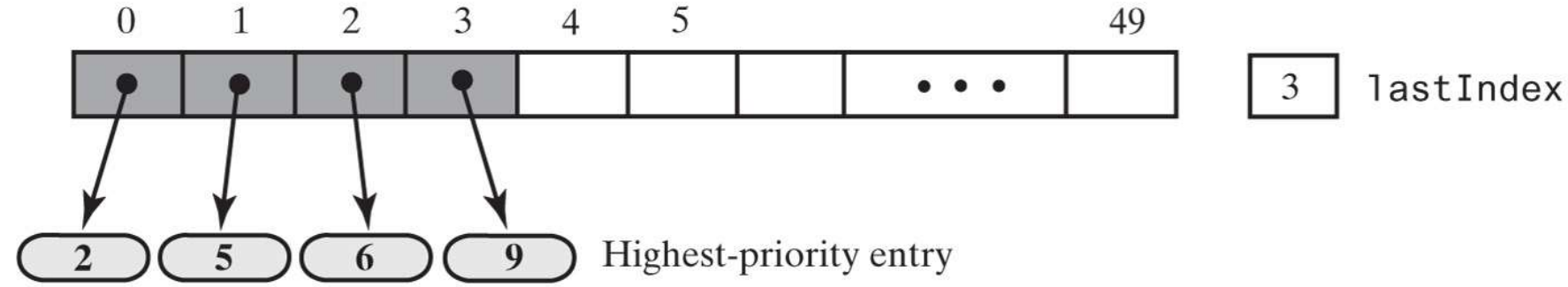
- Implementation of `removeBack`, an O(1) operation.

```
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
```

Possible Implementations of a Priority Queue

Two possible implementations of a priority queue

(a) Array based



(b) Link based