CS2400 - Data Structures and Advanced Programming
Module 11: Queues

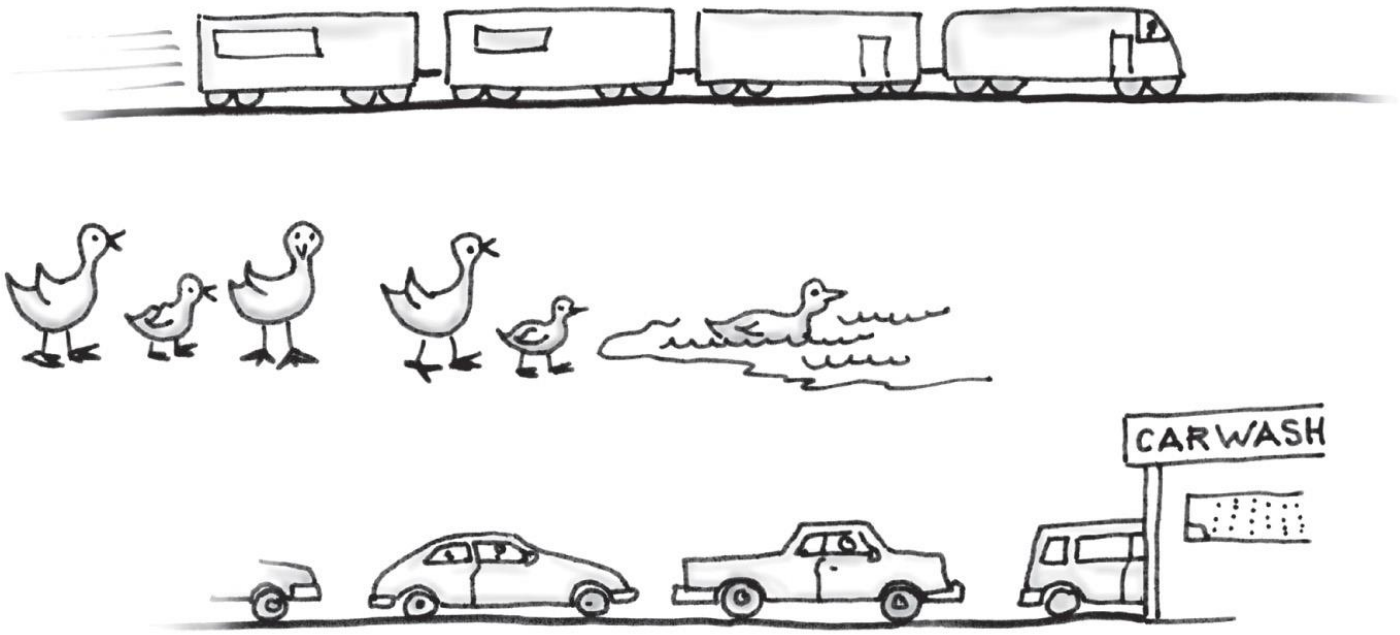
Hao Ji

Computer Science Department

Cal Poly Pomona

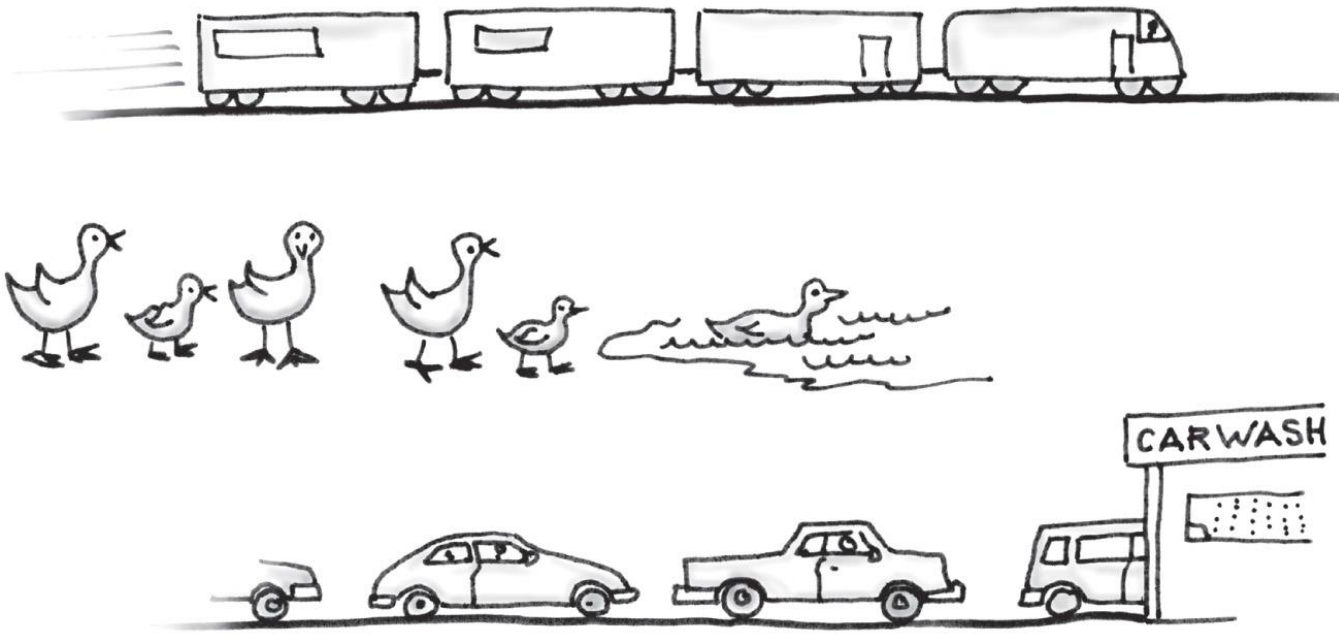
Queue

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



Queue

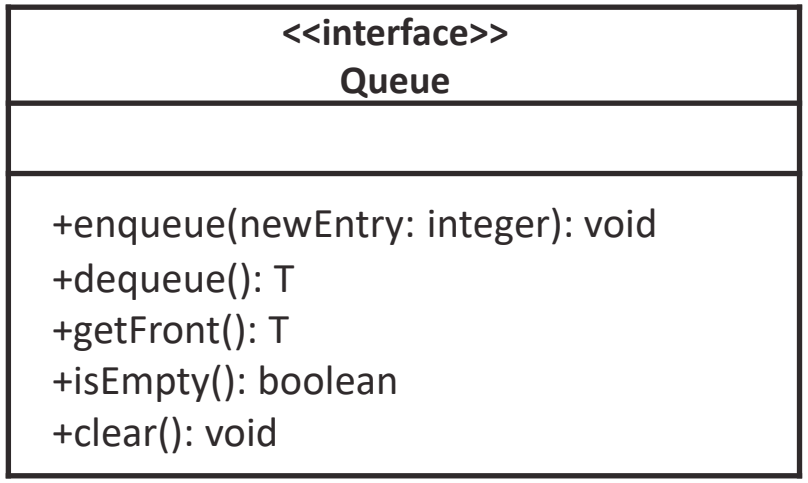
- A way to organize data
  - A collection of objects in chronological order and having the same data type



Entries organized first-in, first-out

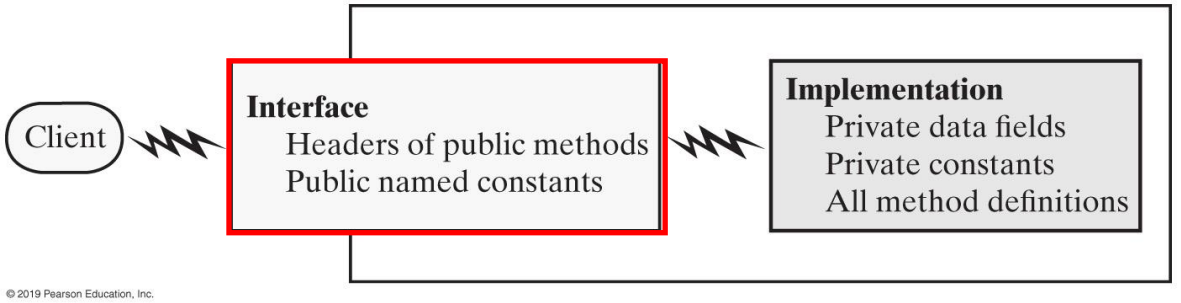
- Item added most recently is at the back of the queue
- Client can look at or remove only the entry at the front of the queue

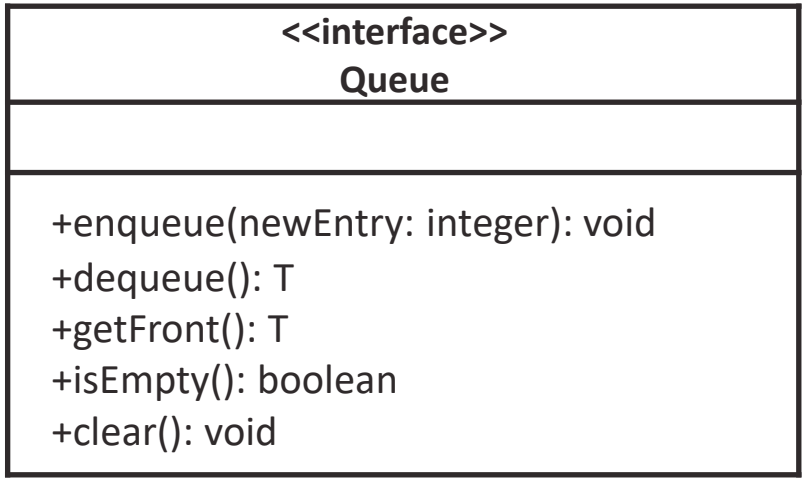
Queue



// Adds a new entry to the back of the queue.
// Removes and returns the entry at the front of the queue
// Retrieves the queue's front entry without changing the queue in any way.
// Detects whether the queue is empty
// Removes all entries from the queue

```java
/** An interface for the ADT queue.  */
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
```

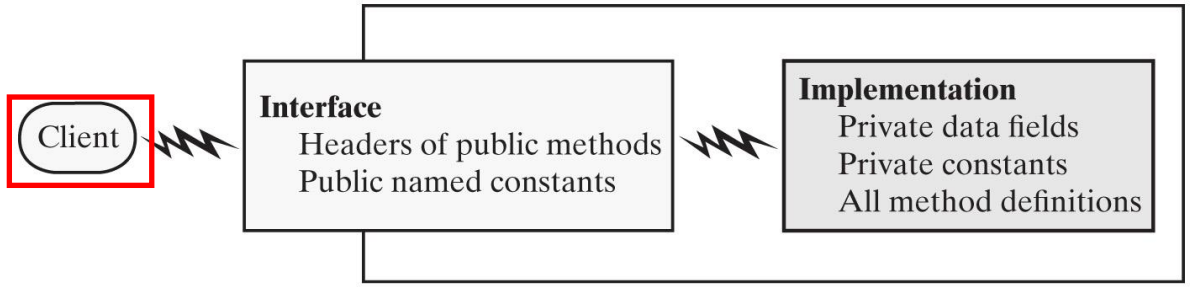


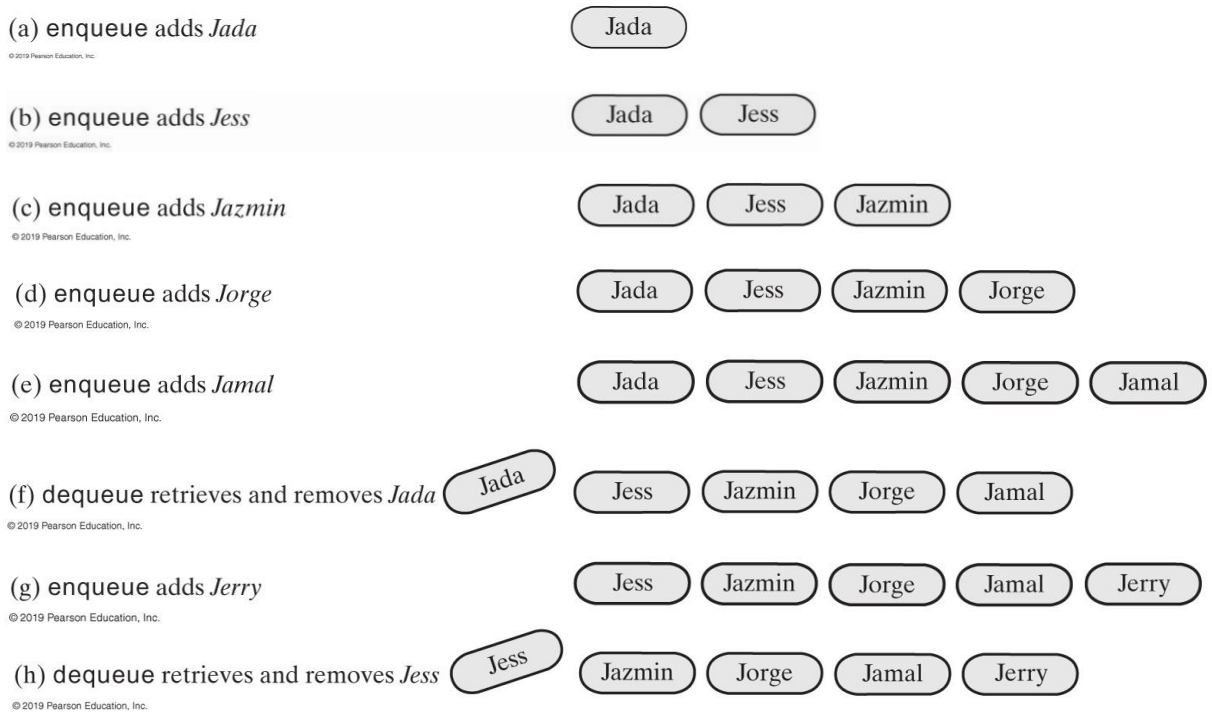


Java Class Library: The Interface Queue

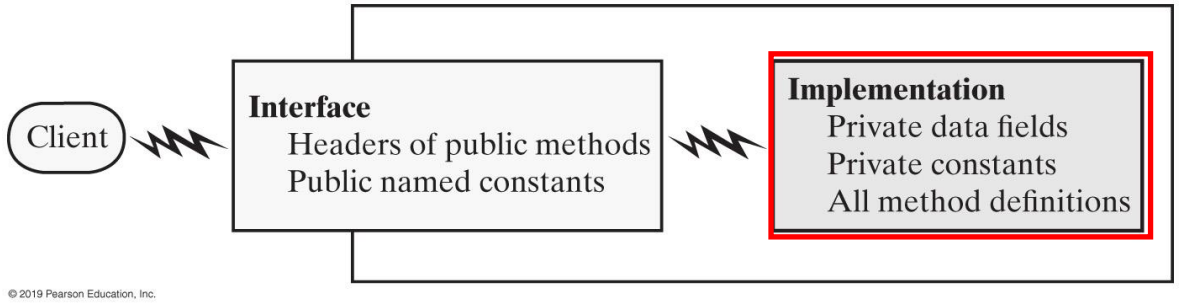
- Methods provided
  - `add`
  - `offer`
  - `remove`
  - `poll`
  - `element`
  - `peek`
  - `isEmpty`
  - `size`

```
QueueInterface<String> myQueue = new LinkedQueue<String>();
myQueue.enqueue("Jim");
myQueue.enqueue("Jess");
myQueue.enqueue("Jill");
myQueue.enqueue("Jane");
myQueue.enqueue("Joe");

String front = myQueue.getFront(); // returns "Jim"
System.out.println(front + " is at the front of the queue.");

front = myQueue.dequeue();         // removes and returns "Jim"
System.out.println(front + " is removed from the queue.");

myQueue.enqueue("Jerry");

front = myQueue.getFront();        // returns "Jess"
System.out.println(front + " is at the front of the queue.");

front = myQueue.dequeue();         // removes and returns "Jess"
System.out.println(front + " is removed from the queue.");
```





Implementations of a Queue



Implementations of a Queue

- An Array-Based Implementation of a Queue
- A Linked Implementation of a Queue

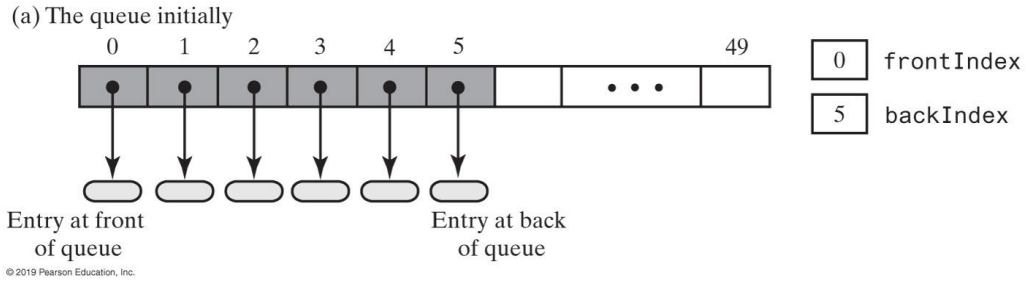
Implementations of a Queue

- **An Array-Based Implementation of a Queue**
- A Linked Implementation of a Queue

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

Implementations of a Queue

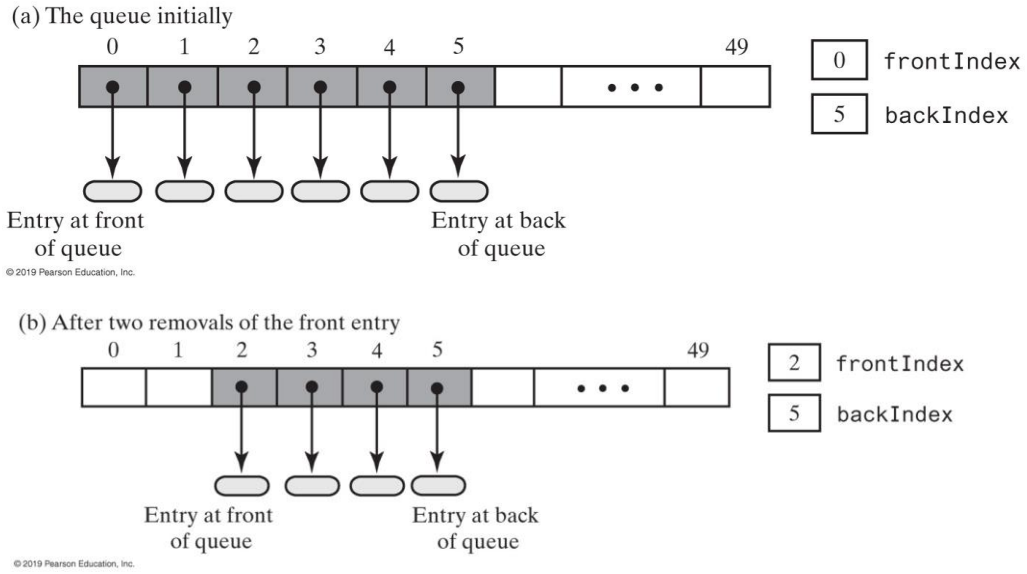
- Problem with letting queue[0] be the queue's front



AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

Implementations of a Queue

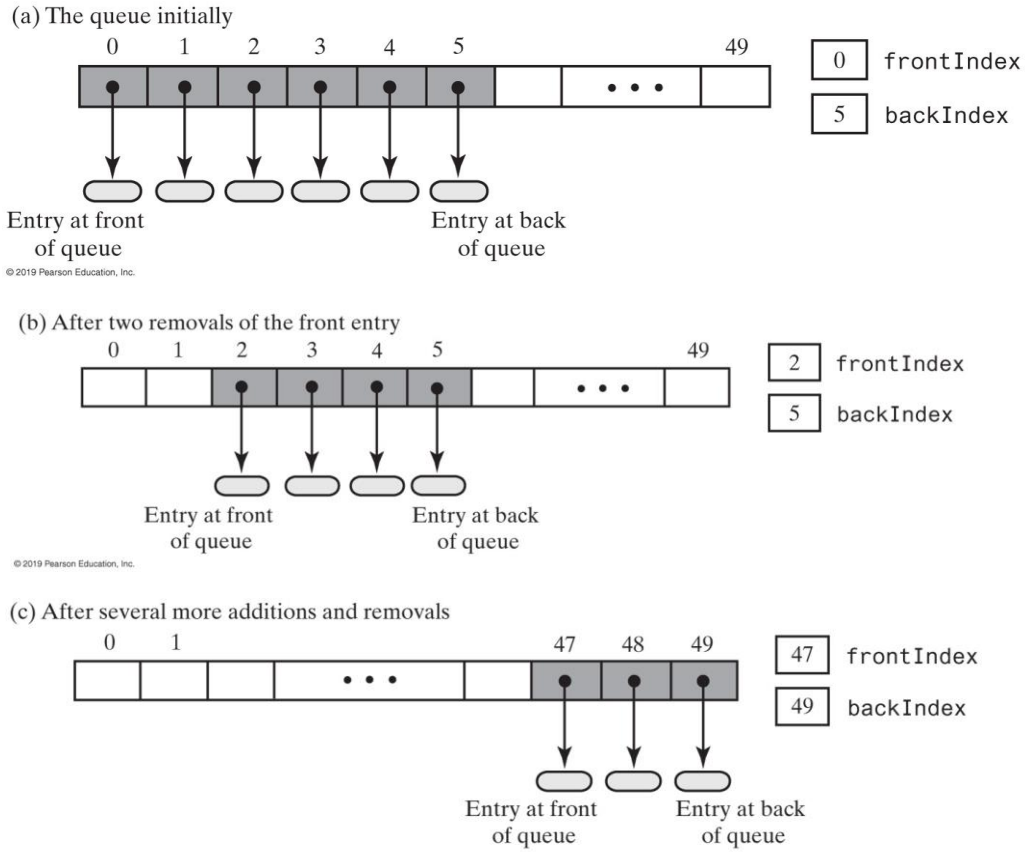
- Problem with letting queue[0] be the queue's front



AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

Implementations of a Queue

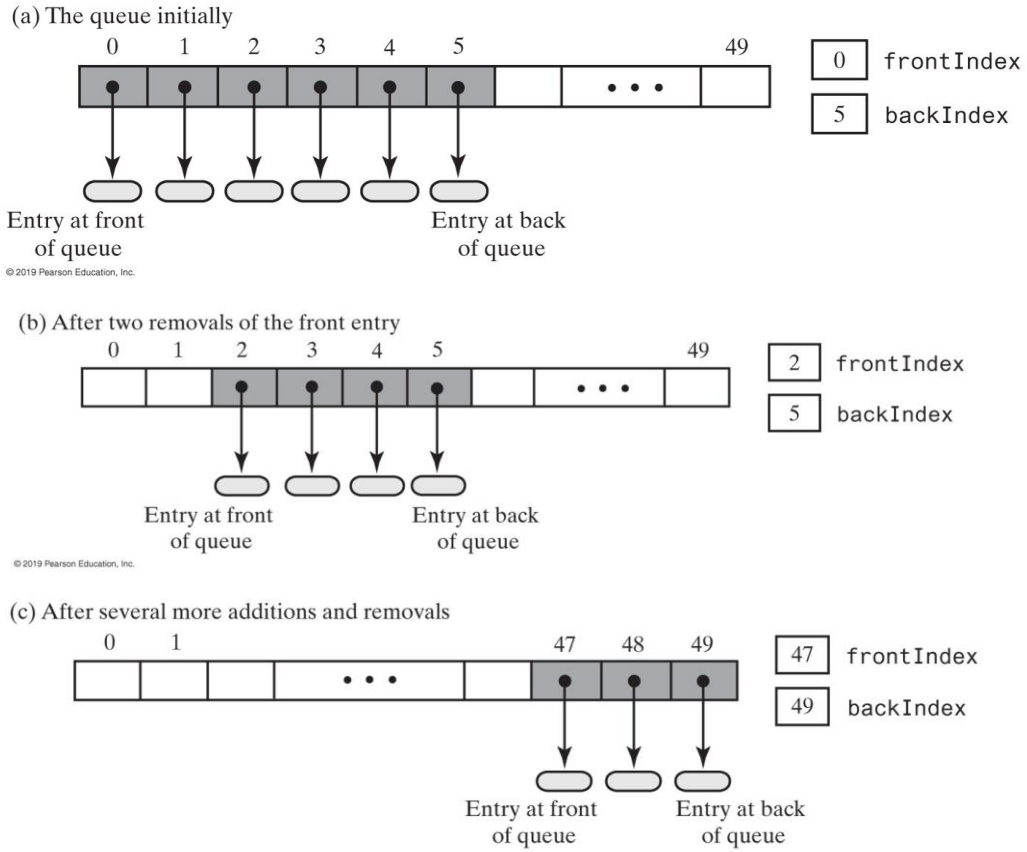
- Problem with letting queue[0] be the queue's front



AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

Implementations of a Queue

- Problem with letting queue[0] be the queue's front

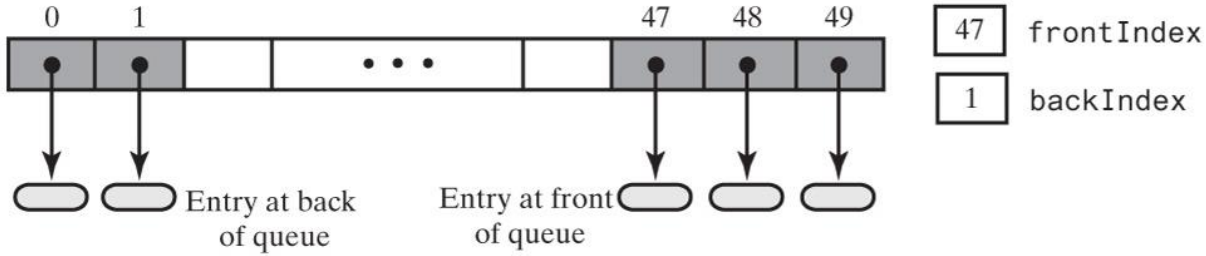


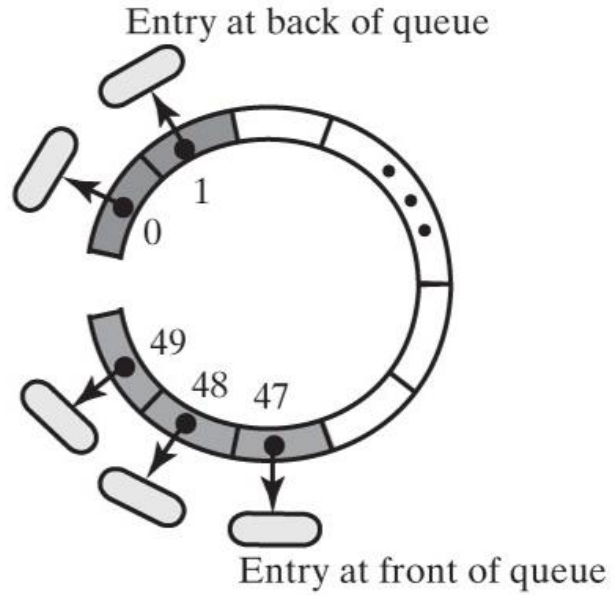
This arrangement would make the operation *dequeue* **inefficient**.

- We would need to shift each array element by one position toward the beginning of the array.

Implementations of a Queue

- **Using a Circular Array**



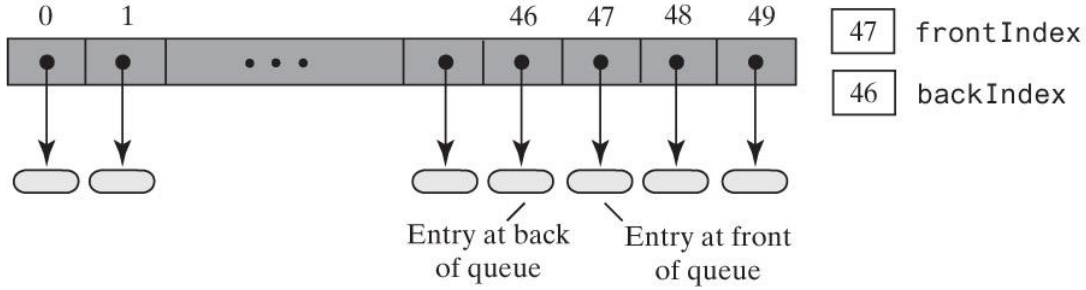


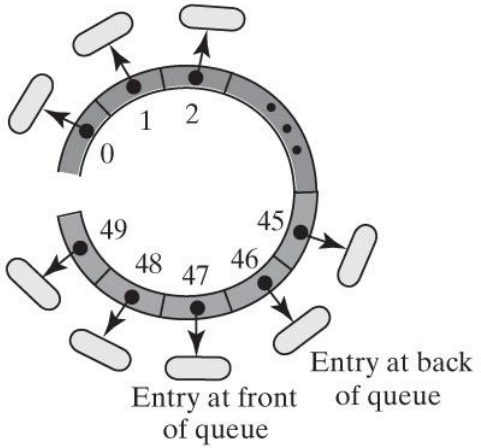

```
backIndex = (backIndex + 1) % queue.length;
```

Implementations of a Queue

- **Using a Circular Array**

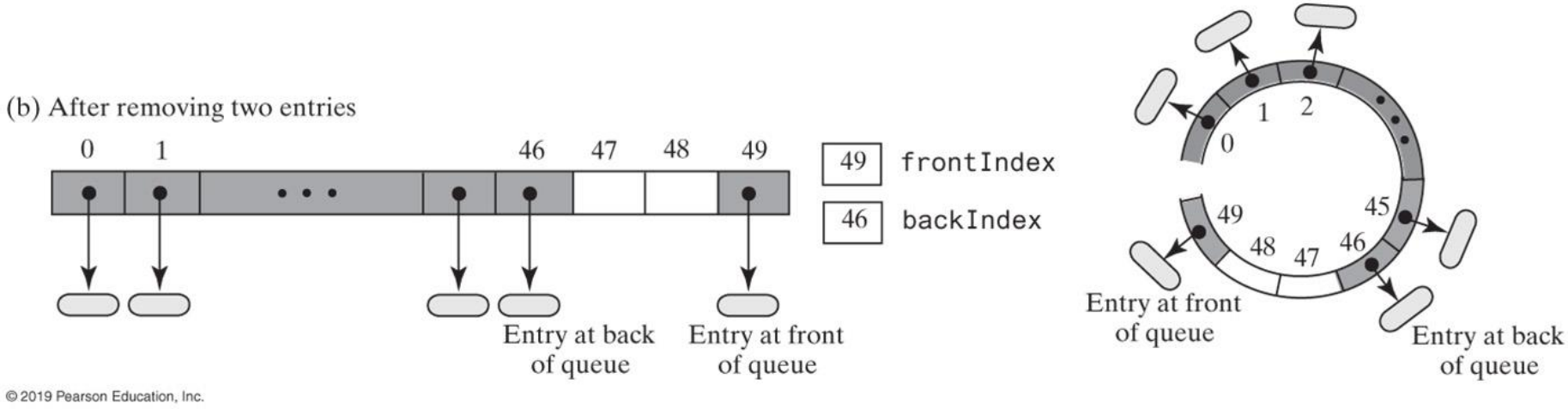
(a) After adding more entries to the queue in Figure 8-7 until it is full





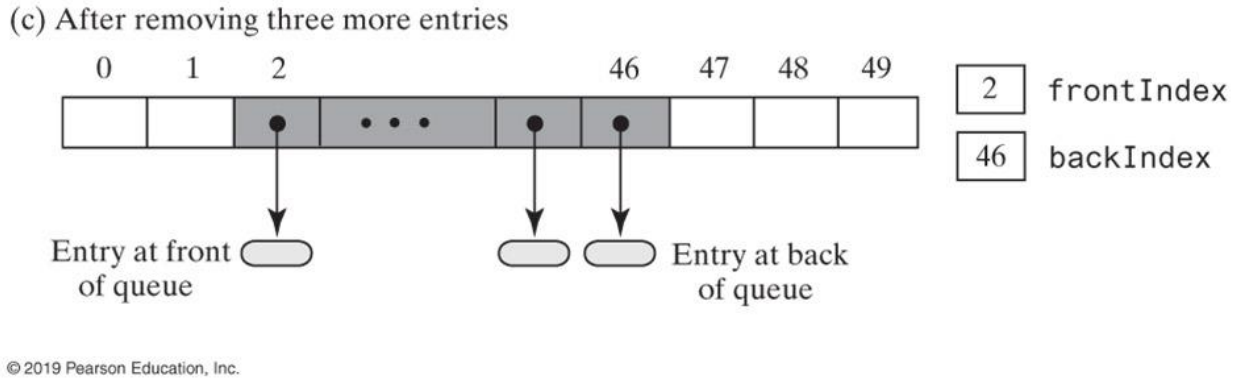
Implementations of a Queue

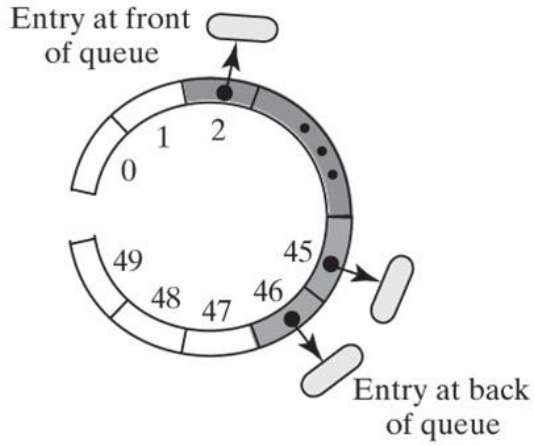
- **Using a Circular Array**



Implementations of a Queue

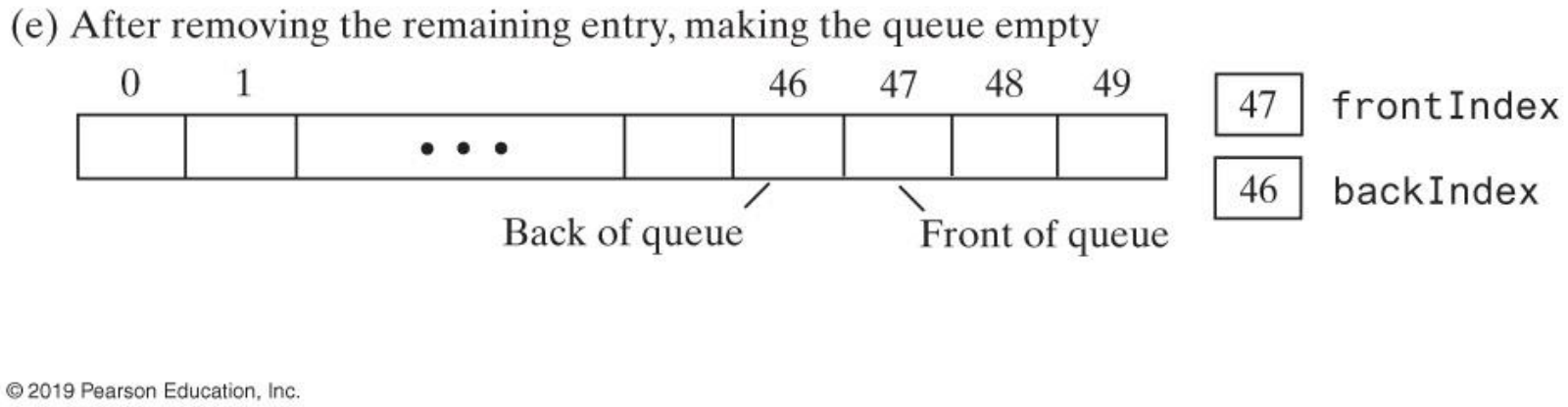
- **Using a Circular Array**

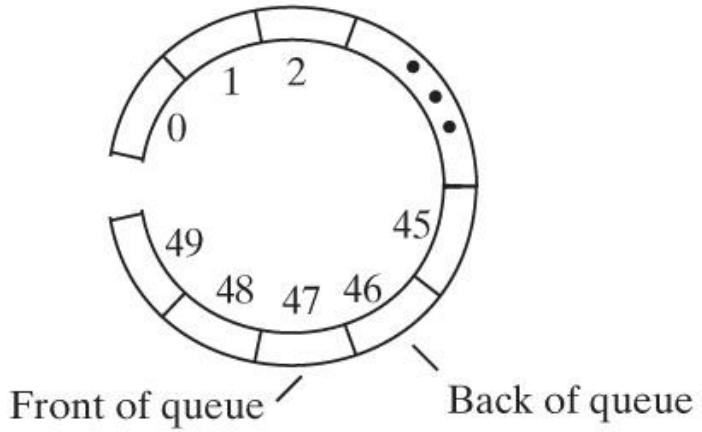




Implementations of a Queue

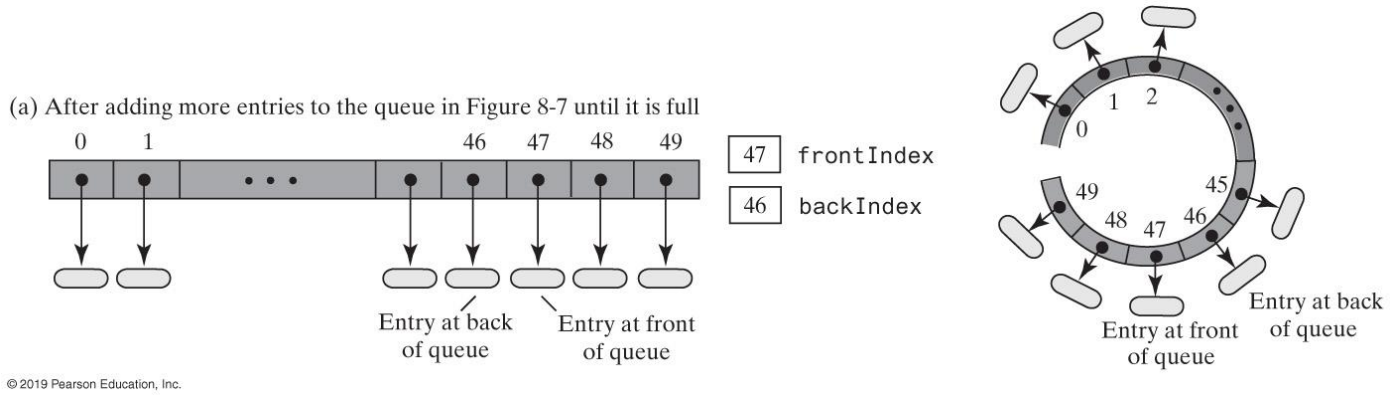
- **Using a Circular Array**

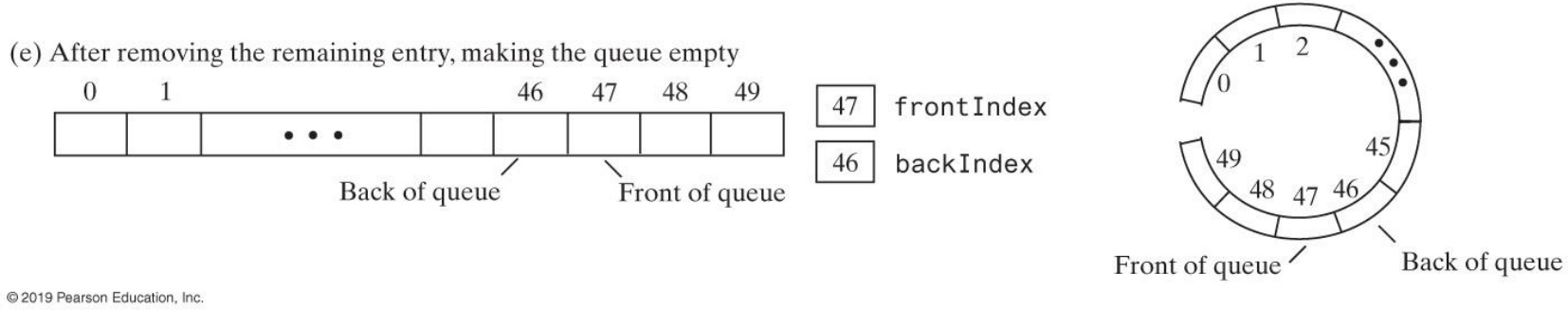




Implementations of a Queue

- We still have an issues in using **a Circular Array**
  - With a circular array, *frontIndex* equals *backIndex + 1* both when the queue is empty and when it is full





Implementations of a Queue

- **Solution**: Circular Array with One Unused Location
  - Leaving unused the array location that follows the back of the queue

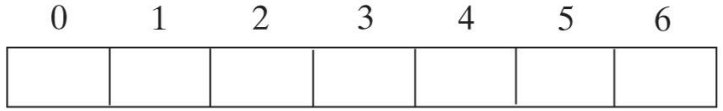
- When the queue is full

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

- When the queue is empty

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

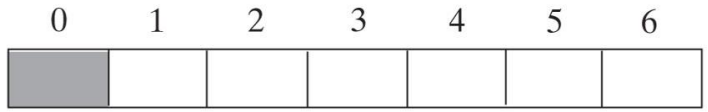
(a) Initially, the queue is empty



0 frontIndex

6 backIndex

(b) After enqueuing one entry



0 frontIndex

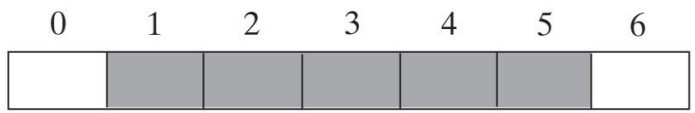
0 backIndex

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

0 frontIndex

5 backIndex

(d) After dequeuing an entry



1 frontIndex

5 backIndex

(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

(i) After dequeuing all but one entry

0 frontIndex

0 backIndex

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

1 frontIndex

0 backIndex

Implementations of a Queue

```
public final class ArrayQueue<T> implements QueueInterface<T>
{
   private T[] queue; // Circular array of queue entries and one unused location
   private int frontIndex;     // Index of front entry
   private int backIndex;      // Index of back entry
   private boolean integrityOK;// true if data structure is created correctly, false if corrupted
   private static final int DEFAULT_CAPACITY = 3;
   private static final int MAX_CAPACITY = 10000;

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

   public ArrayQueue(int initialCapacity)
   {
      integrityOK = false;
      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;
      integrityOK = true;
   } // end constructor
   …
}
```

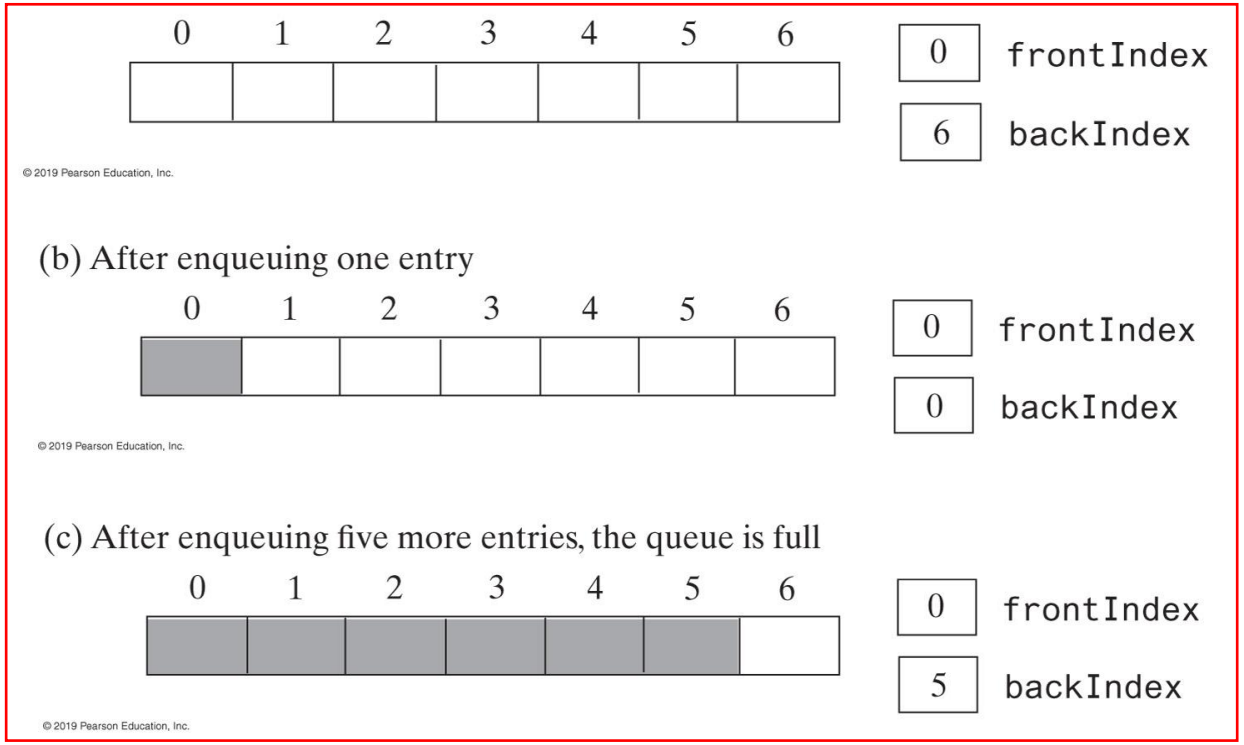
AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

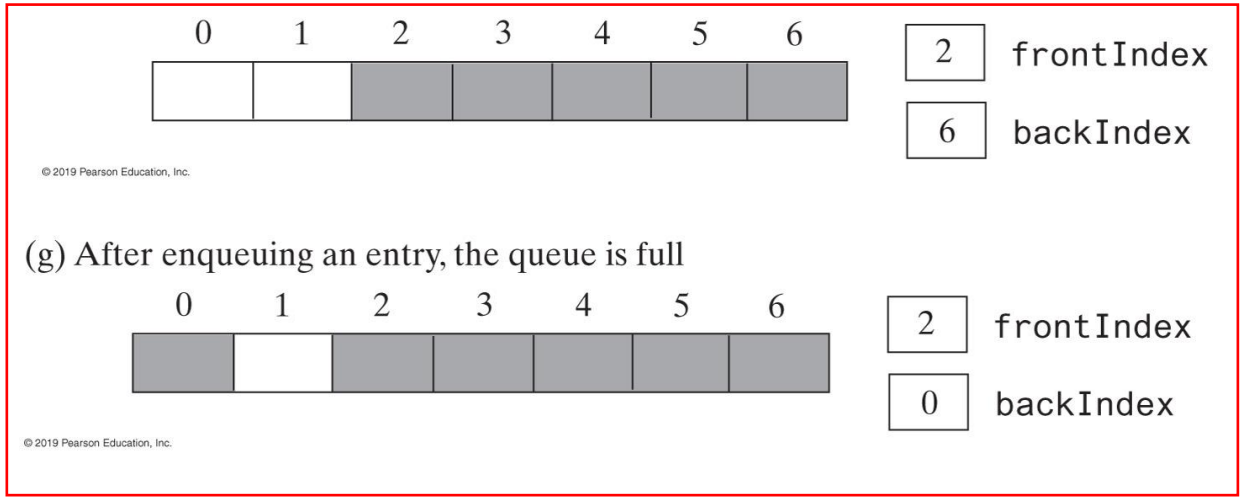
Implementations of a Queue

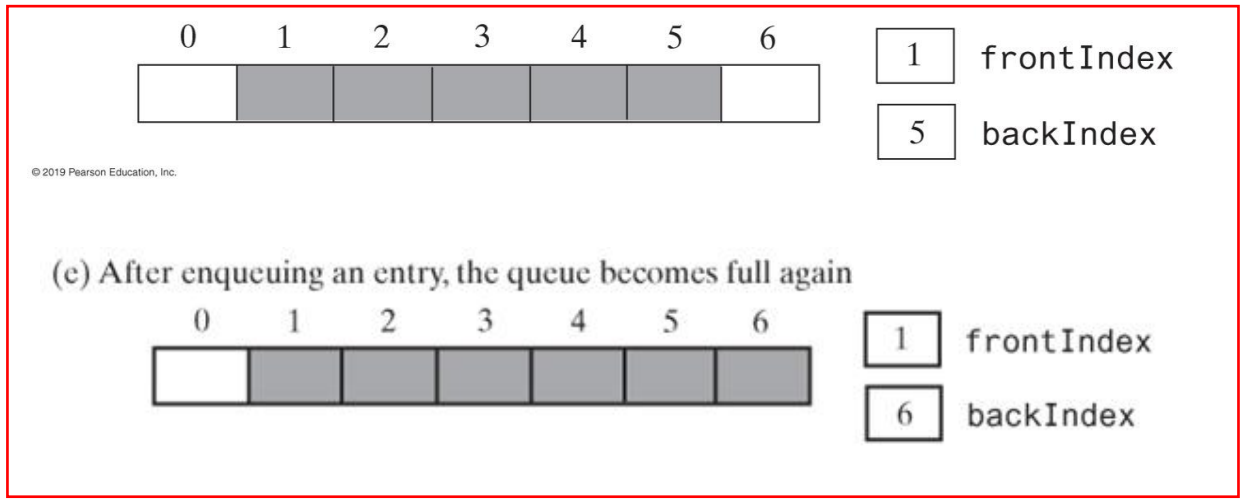
- Adding to the back

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

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void








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

Implementations of a Queue

- Adding to the back

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

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void **-ensureCapacity(): void**

Implementations of a Queue

- Adding to the back

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

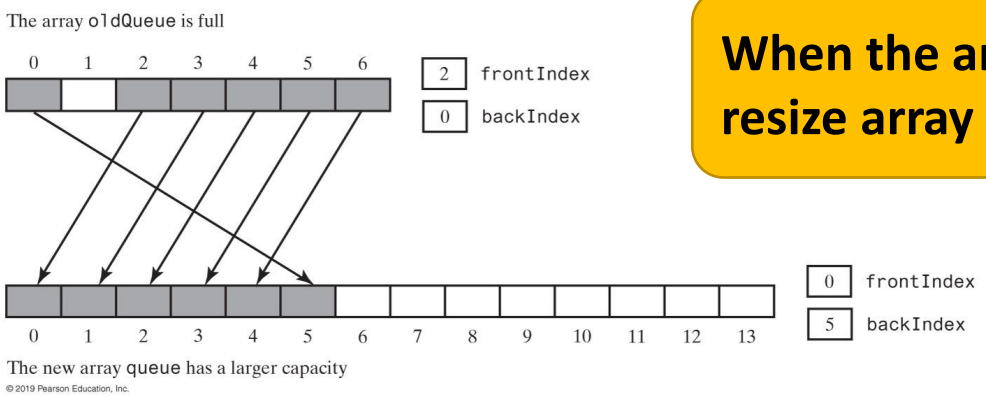
When the array is full, we need to resize array by doubling its size

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void -ensureCapacity(): void

Implementations of a Queue

- Adding to the back

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



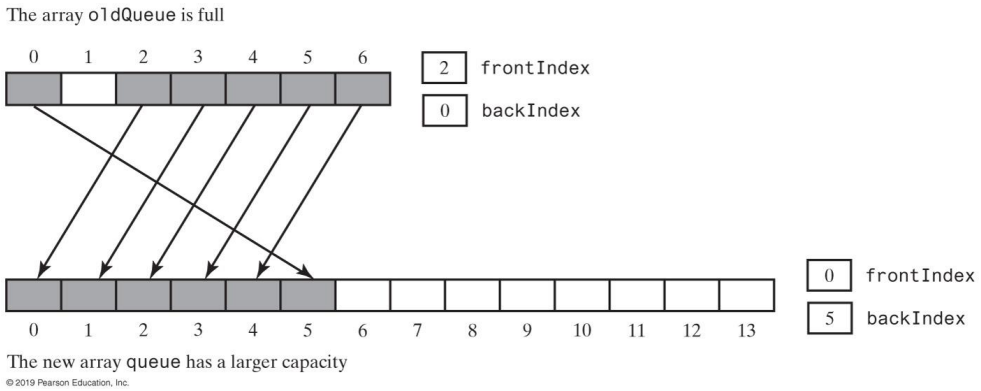
When the array is full, we need to resize array by doubling its size

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void -ensureCapacity(): void

Implementations of a Queue

- Adding to the back

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

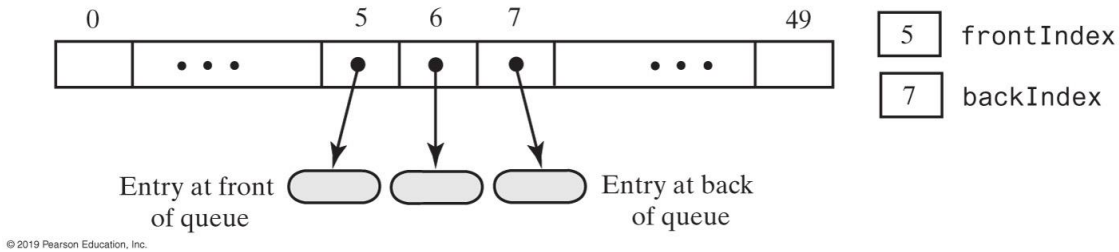



```
// Doubles the size of the array queue if it is full.
// Precondition: checkIntegrity has been called.
private void ensureCapacity()
{
   if (frontIndex == ((backIndex + 2) % queue.length)) // If array is full,
   {                                                   // double size of array
      T[] 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
```

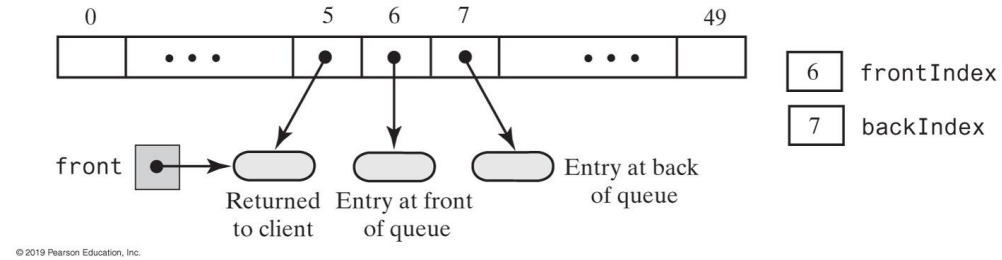
Implementations of a Queue

- Removing the front entry

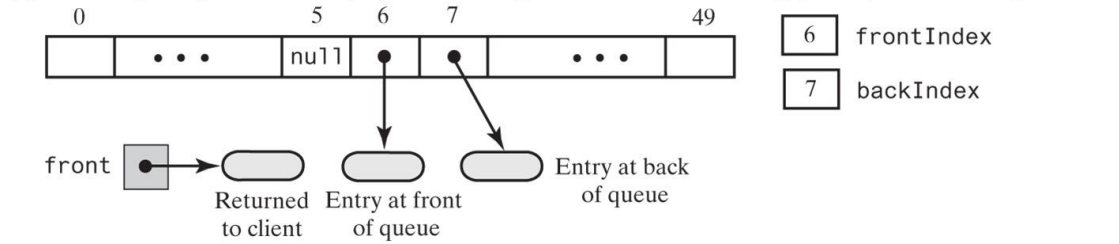
(a) Initially



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



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

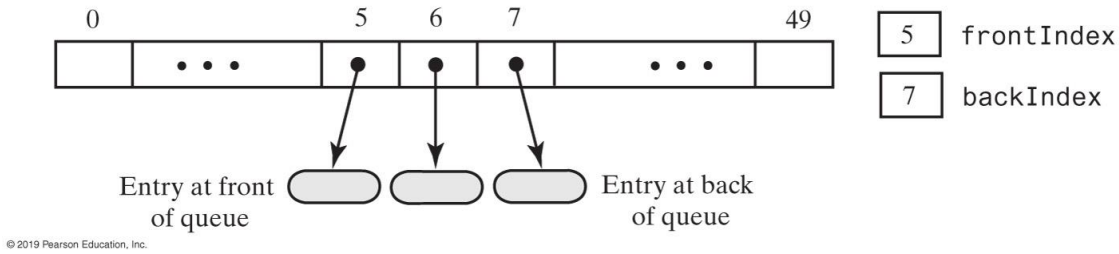


AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void **+dequeue(): T** +getFront(): T +isEmpty(): boolean +clear(): void -ensureCapacity(): void

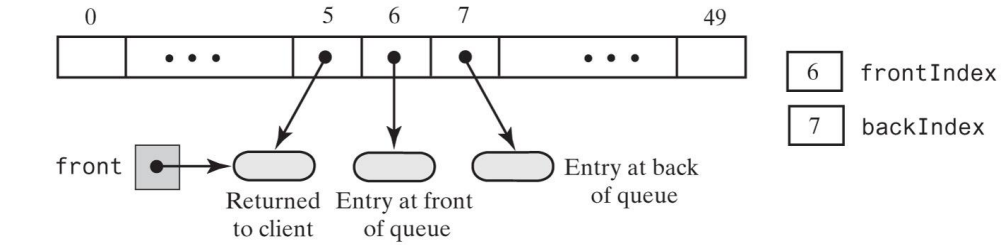
Implementations of a Queue

- Removing the front entry

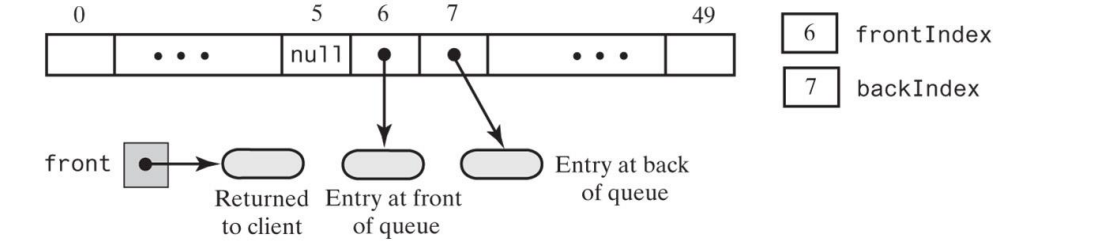
(a) Initially



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



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




```
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
```

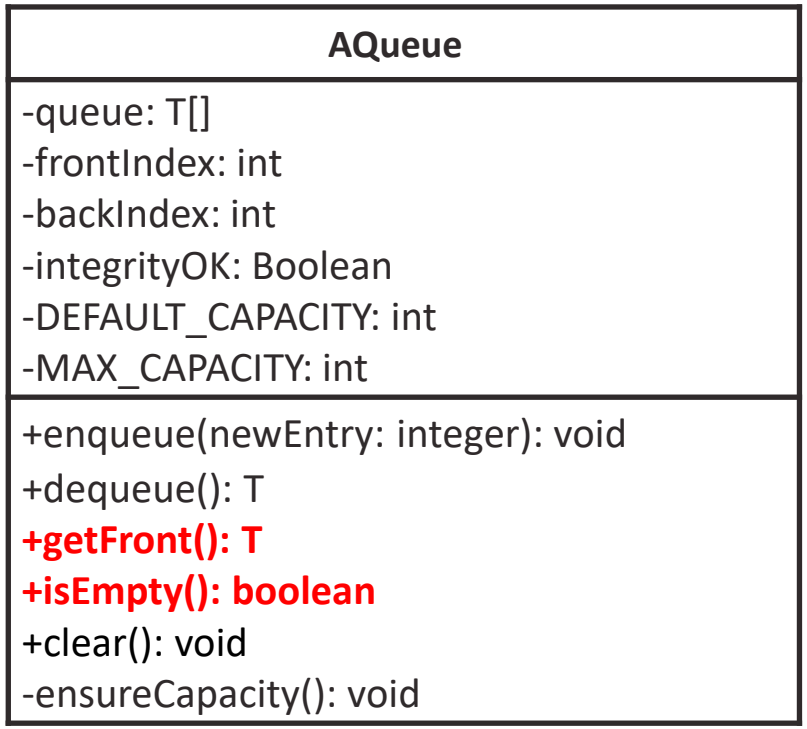
Implementations of a Queue

- The method **getFront**

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

- The method **isEmpty**

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



In-Class Exercises

- Write an implementation of clear that sets to null each array element that was used for the queue

```
public void clear()
{
      checkIntegrity();
      if (!isEmpty())
      { // Deallocates only the used portion
            for (int index = frontIndex; index != backIndex; index = (index + 1) % queue.length)
            {
                  queue[index] = null;
            } // end for

            queue[backIndex] = null;
      } // end if

      frontIndex = 0;
      backIndex = queue.length - 1;
} // end clear
```

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean **+clear(): void** -ensureCapacity(): void

In-Class Exercises

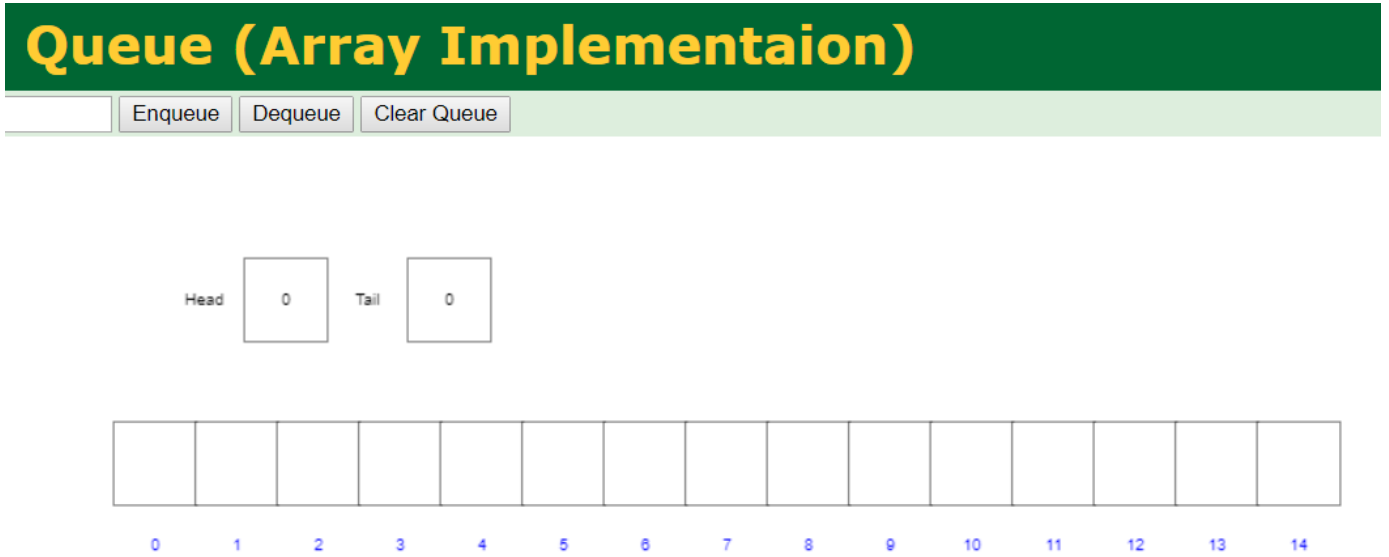
- Write an implementation of clear that sets to null each array element that was used for the queue

```
public void clear()
{
      checkIntegrity();
      if (!isEmpty())
      { // Deallocates only the used portion
            for (int index = frontIndex; index != backIndex; index = (index + 1) % queue.length)
            {
                  queue[index] = null;
            } // end for

            queue[backIndex] = null;
      } // end if

      frontIndex = 0;
      backIndex = queue.length - 1;
} // end clear
```

AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean **+clear(): void** -ensureCapacity(): void

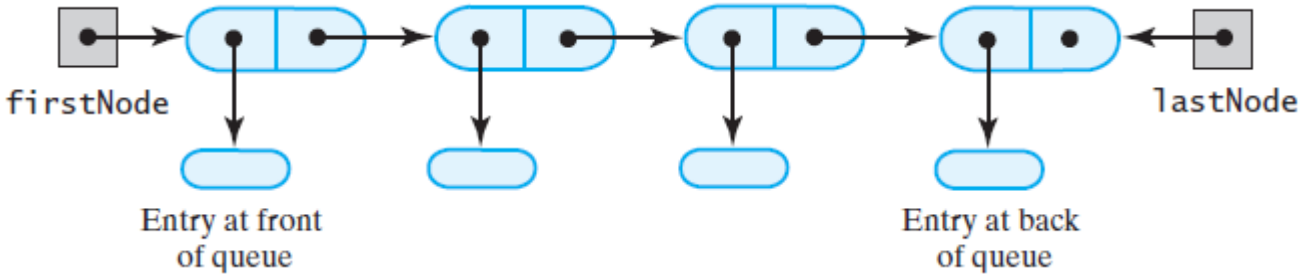
Interactive and Visualization Demo

- https://www.cs.usfca.edu/~galles/visualization/QueueArray.html



Implementations of a Queue

- An Array-Based Implementation of a Queue
- **A Linked Implementation of a Queue**

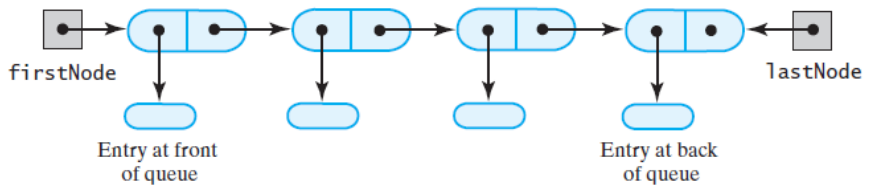


LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

Implementations of a Queue

```
/**
   A class that implements a queue of objects by using
   a chain of linked nodes.
   @author Frank M. Carrano
*/
public 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
   {
      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
```

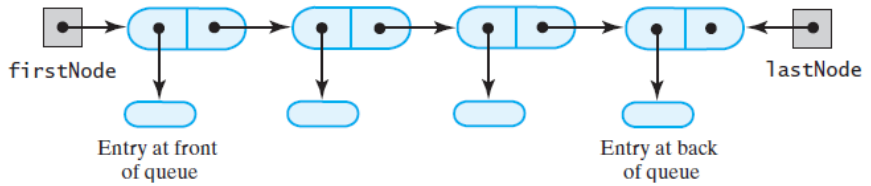
LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void



Implementations of a Queue

- Adding to the back

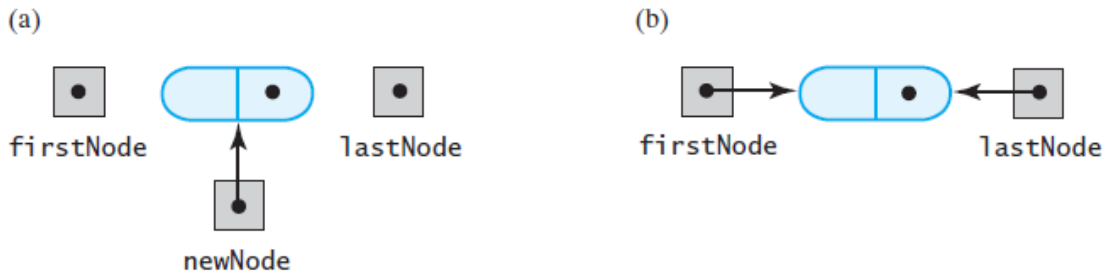
LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void



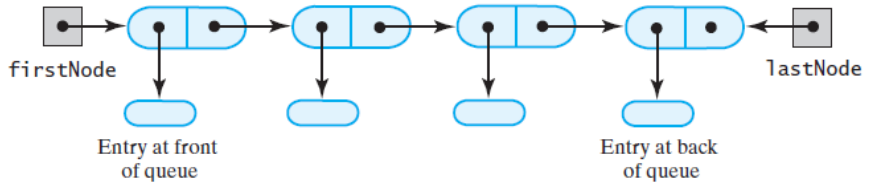
Implementations of a Queue

- Adding to the back
  - **Situation 1: an empty chain**

(a) Before adding a new node to an empty chain; (b) after adding it



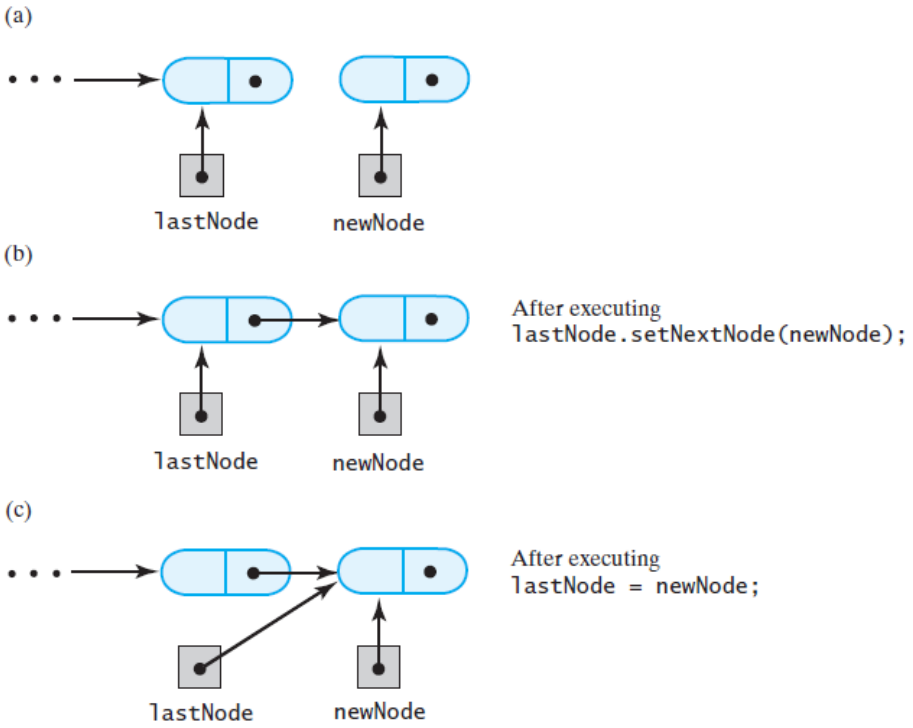
LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void



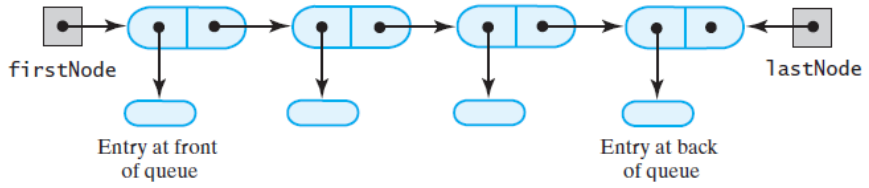
Implementations of a Queue

- Adding to the back
  - **Situation 2: a non-empty chain**

(a) Before, (b) during, and (c) after adding a new node to the end of a nonempty chain that has a tail reference



LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

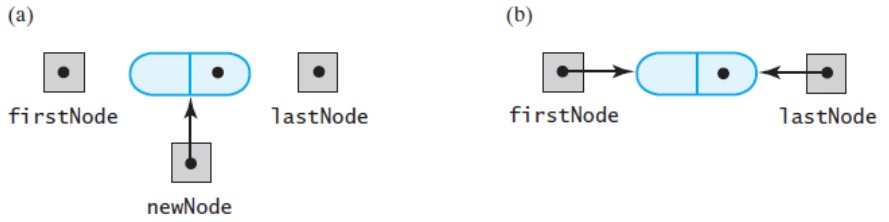


Implementations of a Q

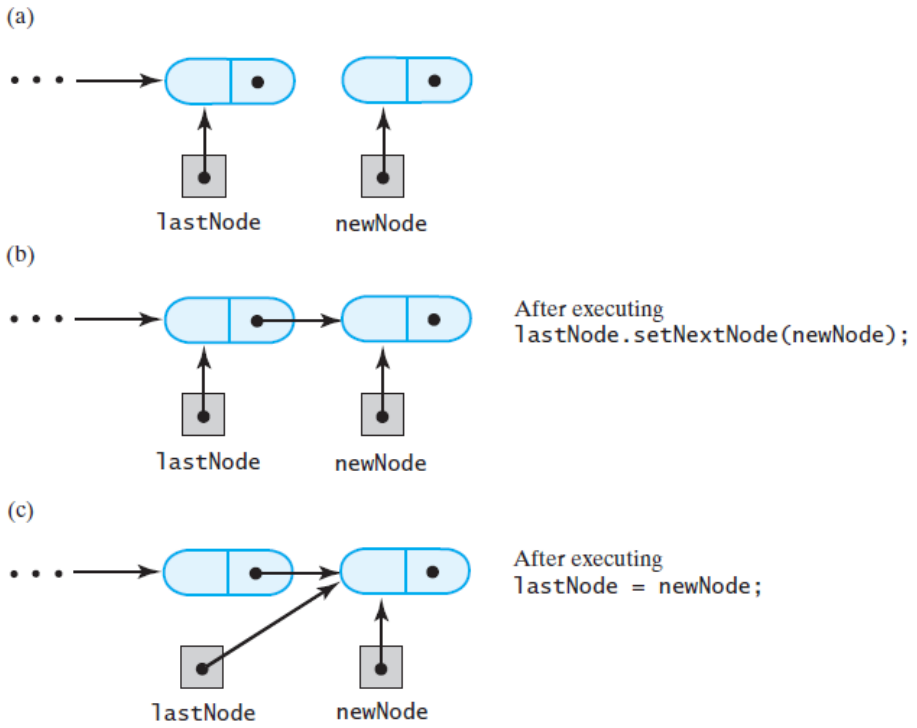
- Adding to the back

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

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

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

(a) Before adding a new node to an empty chain; (b) after adding it



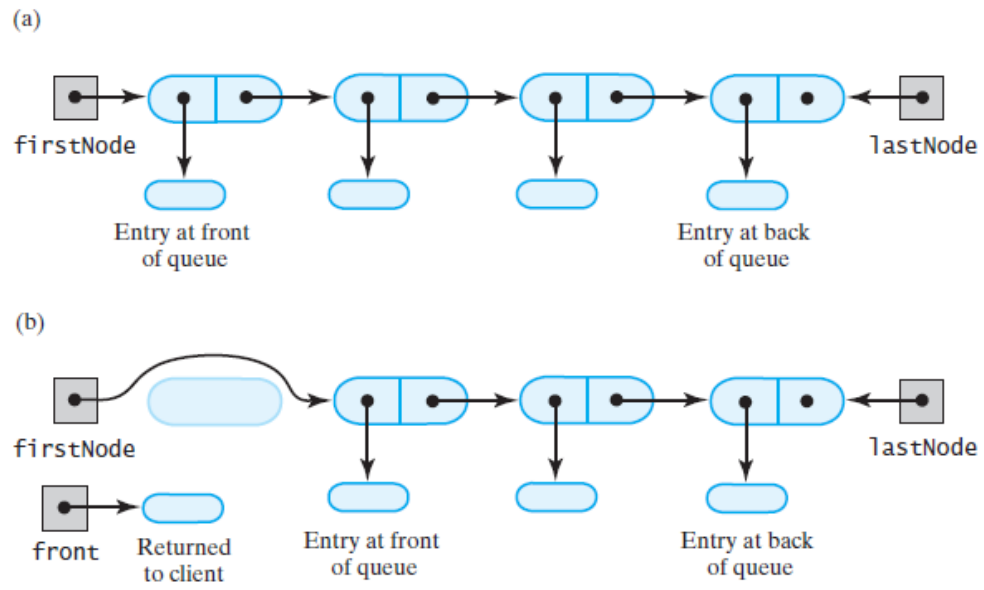
(a) Before, (b) during, and (c) after adding a new node to the end of a nonempty chain that has a tail reference



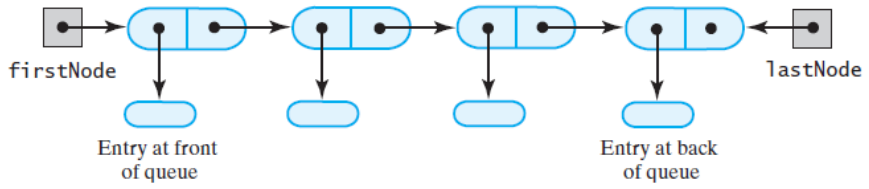
Implementations of a Queue

- Removing the front entry
  - **Situation 1: a chain with more than one entry**

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



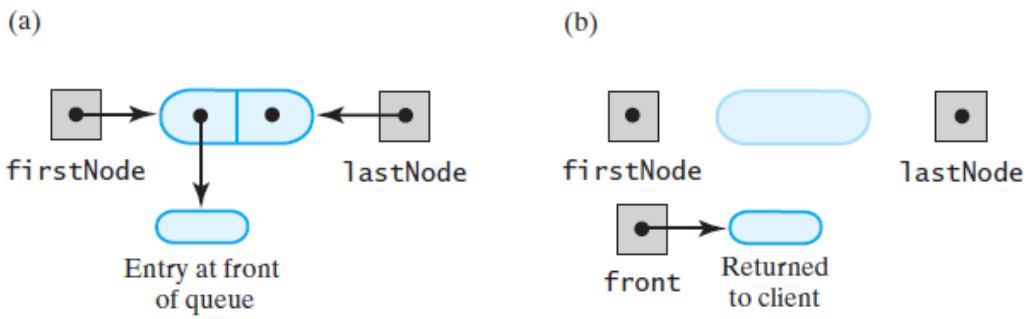
LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void **+dequeue(): T** +getFront(): T +isEmpty(): boolean +clear(): void



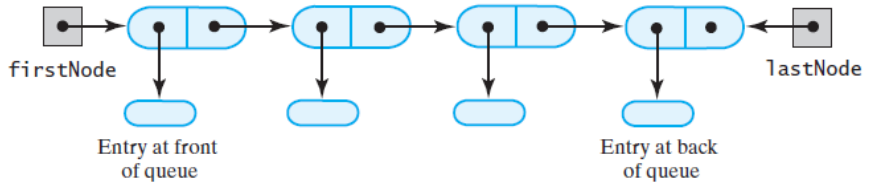
Implementations of a Queue

- Removing the front entry
  - **Situation 2: a chain with only one entry**

(a) A queue of one entry; (b) after removing the entry at the front of the queue



LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void **+dequeue(): T** +getFront(): T +isEmpty(): boolean +clear(): void

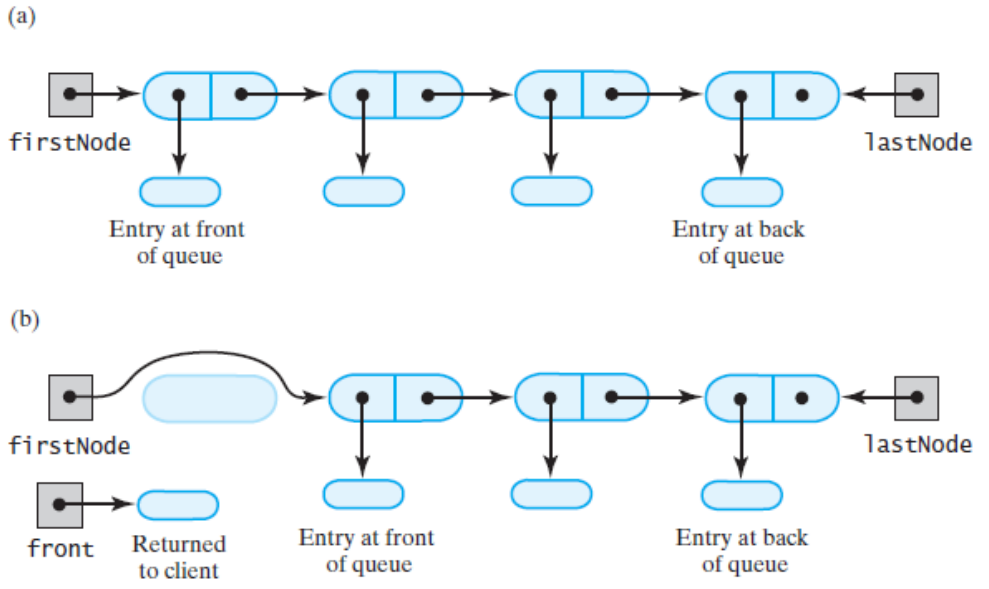


Implementations of a

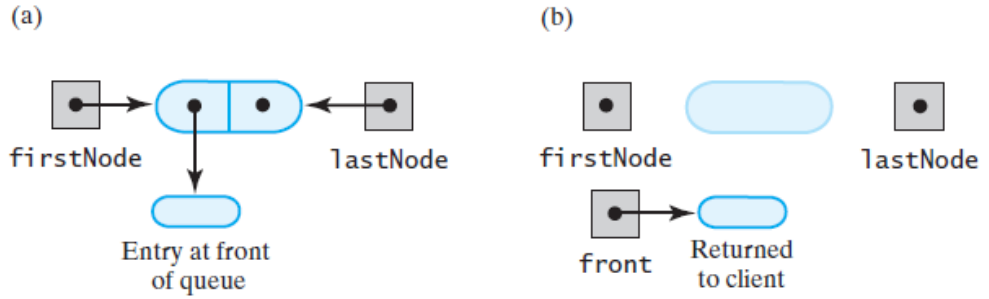
- Removing the front entry

```
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
```

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



(a) A queue of one entry; (b) after removing the entry at the front of the queue



Implementations of a Queue

- The method **getFront**

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

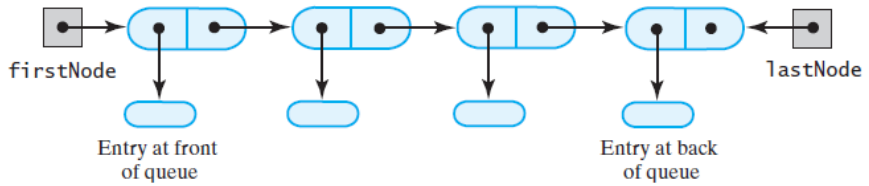
- The method **isEmpty**

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

- The method **clear**

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

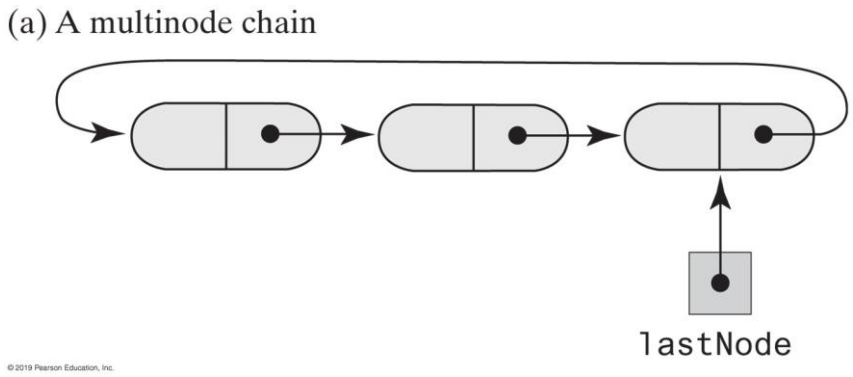
LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T **+getFront(): T** **+isEmpty(): boolean** **+clear(): void**

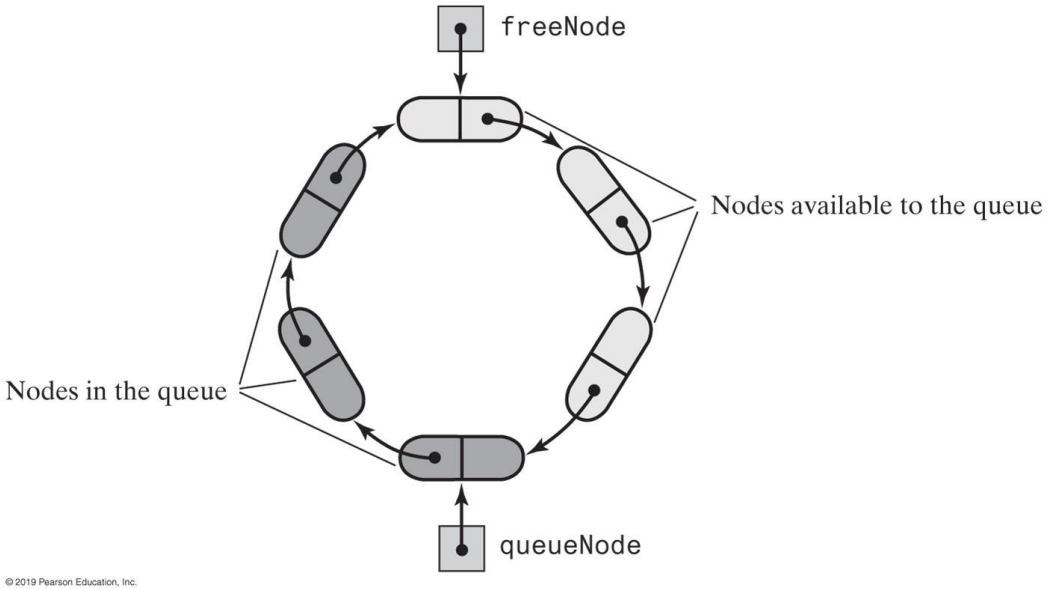


Implementations of a Queue

- An Array-Based Implementation of a Queue
- **A Linked Implementation of a Queue**

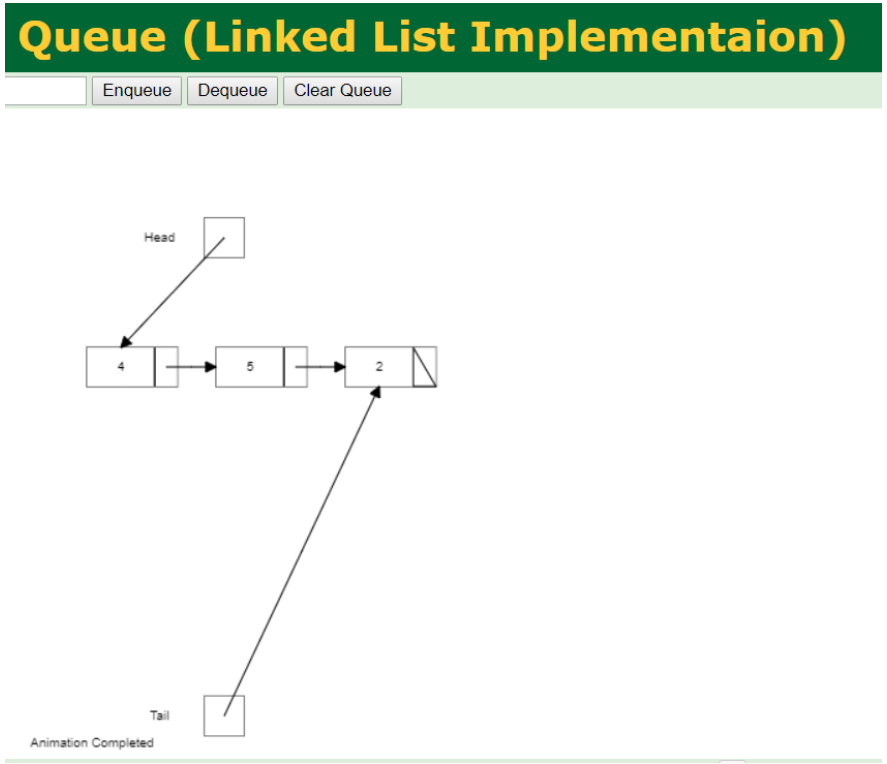
It is possible to use Circular Linked Implementations of a Queue





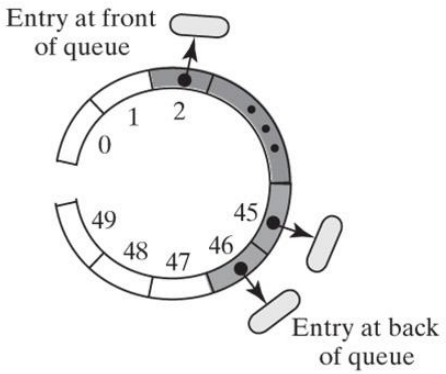
Interactive and Visualization Demo

- https://www.cs.usfca.edu/~galles/visualization/QueueLL.html



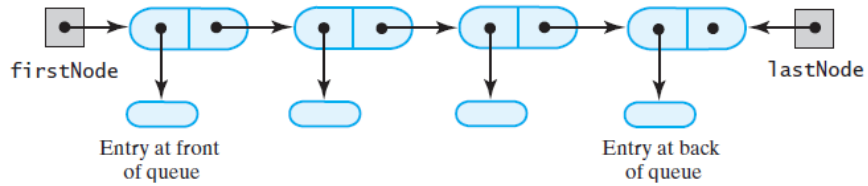
In-Class Exercises: Algorithm Analysis

- What is the **Big Oh** of each queue method in the best case and the worst case?



AQueue
-queue: T[] -frontIndex: int -backIndex: int -integrityOK: Boolean -DEFAULT_CAPACITY: int -MAX_CAPACITY: int
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void -ensureCapacity(): void

LQueue
-firstNode: Node -lastNode: Node
+enqueue(newEntry: integer): void +dequeue(): T +getFront(): T +isEmpty(): boolean +clear(): void

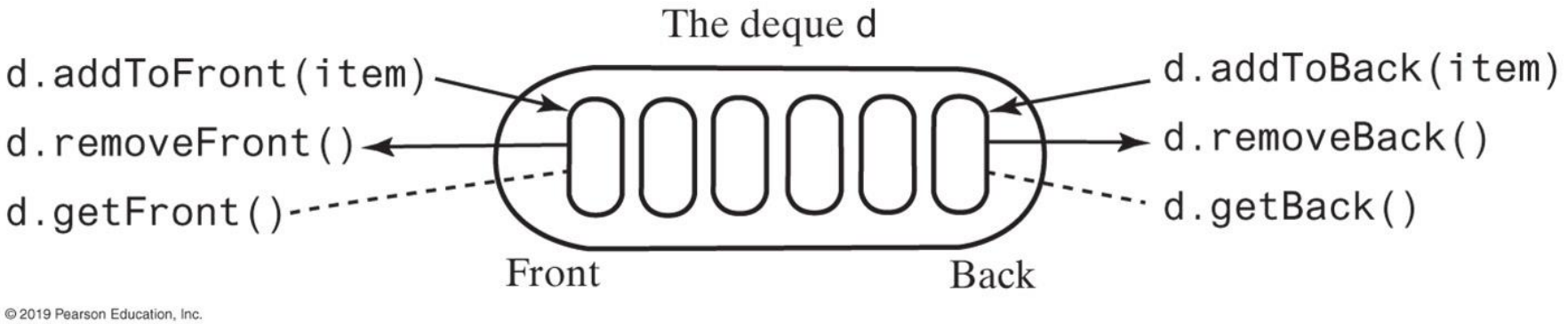


More on Queues

- Queue
- **Deque**
- **Priority Queue**

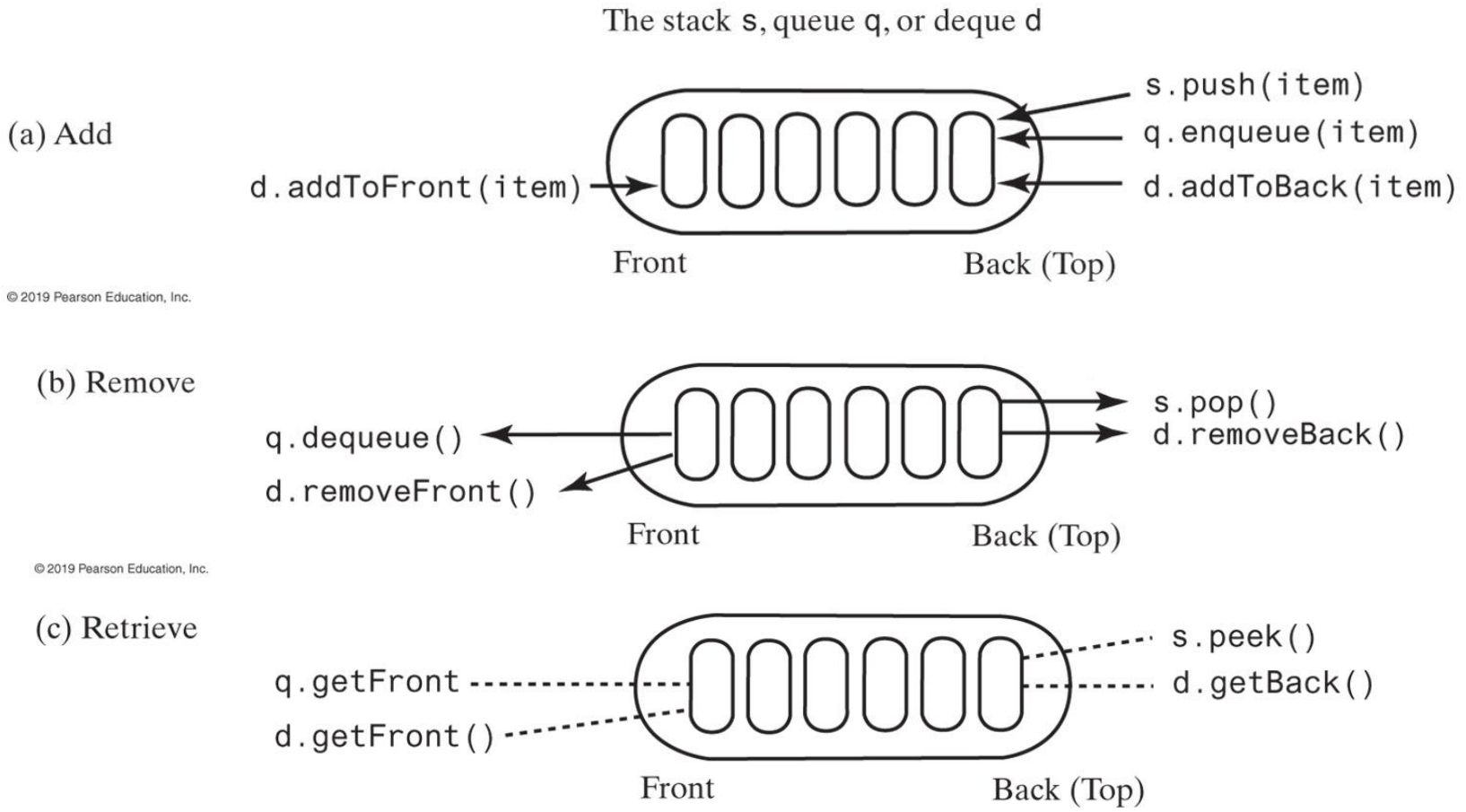
The ADT Deque

- A double ended queue
  - `Deque` pronounced “deck”
- Has both queue-like operations and stack-like operations



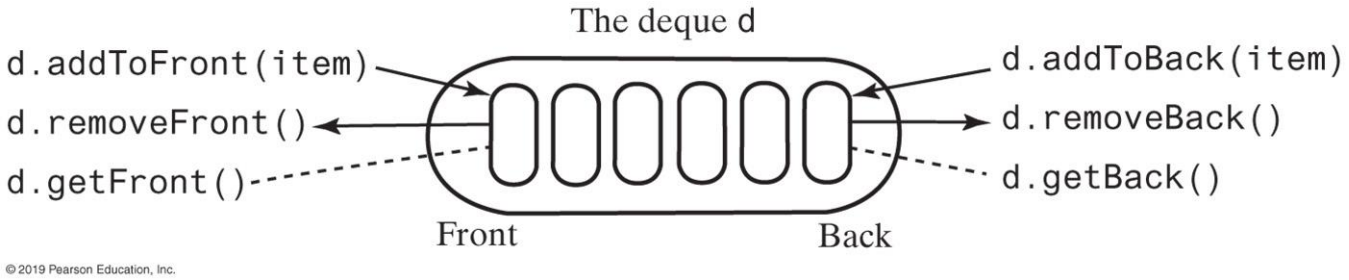
The ADT Deque

- A comparison of operations for a stack s, a queue q, and a deque d



The ADT Deque

```
/** An interface for the ADT deque. */
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.
     @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
```



Java Class Library: The Interface Deque

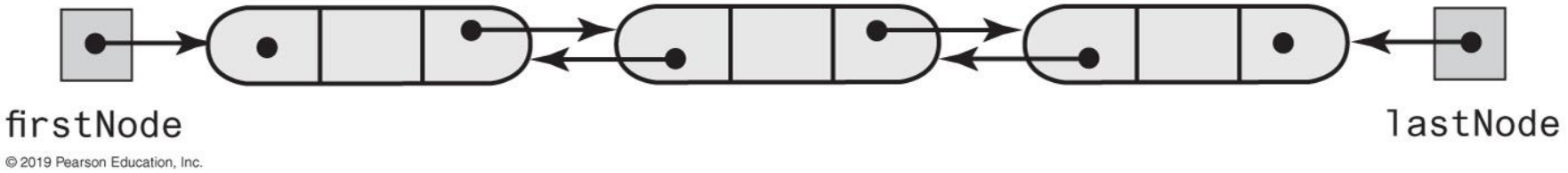
- Methods provided
  - `addFirst, offerFirst`
  - `addLast, offerLast`
  - `removeFirst, pollFirst`
  - `removeLast, pollLast`
  - `getFirst, peekFirst`
  - `getLast, peekLast`
  - `isEmpty, clear, size`
  - `push, pop`

Java Class Library: The Class ArrayDeque

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

Doubly Linked Implementation of a Deque

- A doubly linked chain with head and tail references



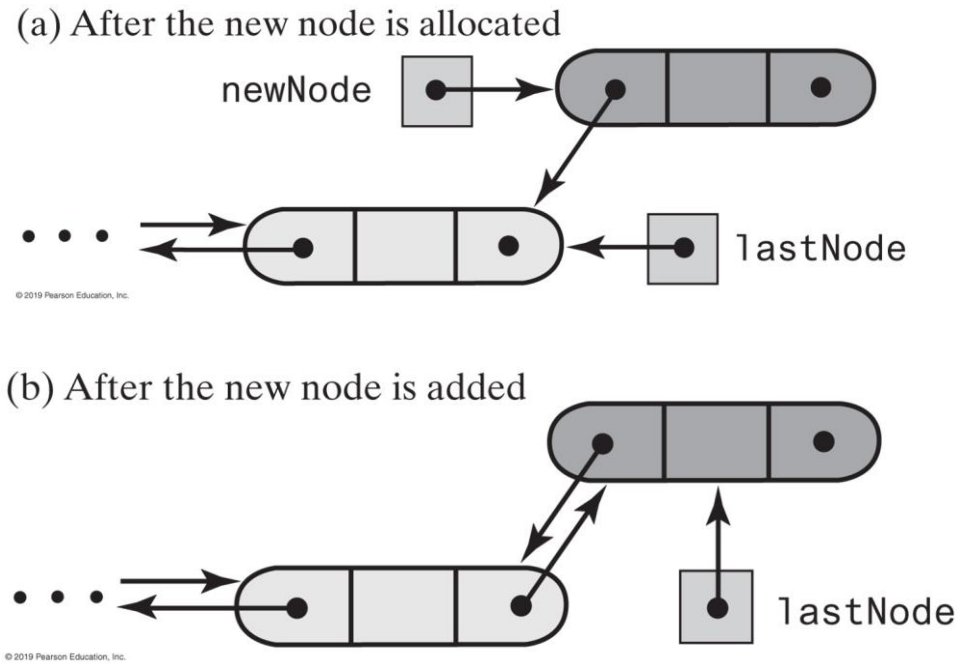
Doubly Linked Implementation of a Deque

```
/**  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
```

LDeque
-firstNode: DLNode -lastNode: DLNode
+addToBack(T newEntry): void +addToFront(T newEntry): void +getBack(): T +getFront(): T +removeFront(): T +removeBack(): T +isEmpty(): boolean +clear(): void

Doubly Linked Implementation of a Deque

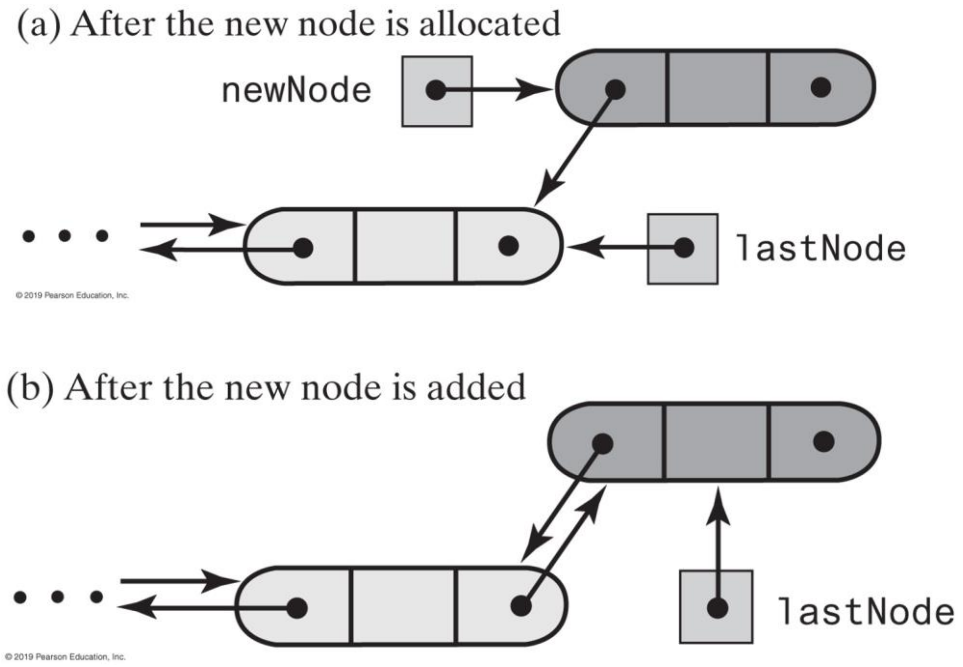
- Adding to the back of a nonempty deque



LDeque
-firstNode: DLNode -lastNode: DLNode
+addToBack(T newEntry): void +addToFront(T newEntry): void +getBack(): T +getFront(): T +removeFront(): T +removeBack(): T +isEmpty(): boolean +clear(): void

Doubly Linked Implementation of a Deque

- Adding to the back of a nonempty deque

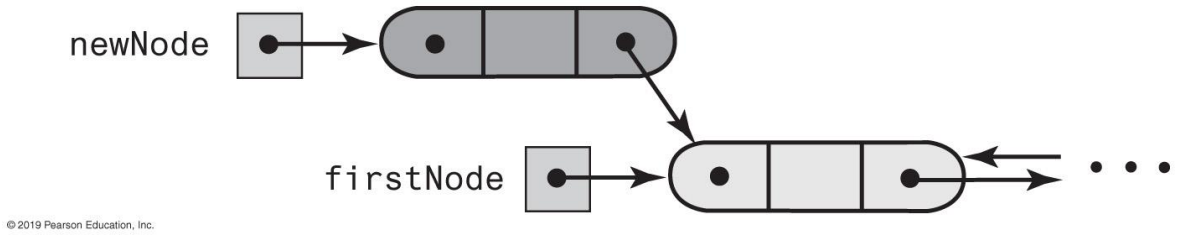



```
public void addToBack(T newEntry)
{
      DLNode newNode = new DLNode(lastNode, newEntry, null);

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

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

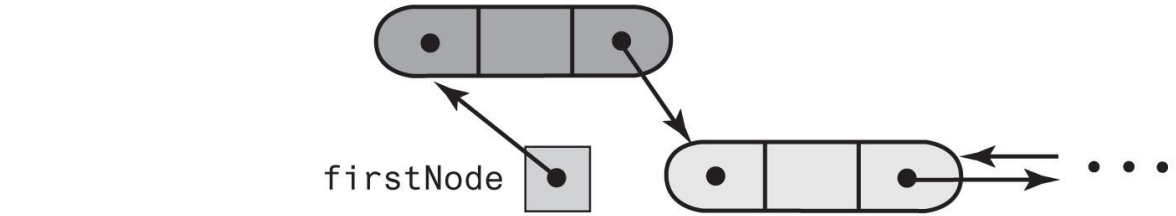
Doubly Linked Implementation of a Deque

- 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

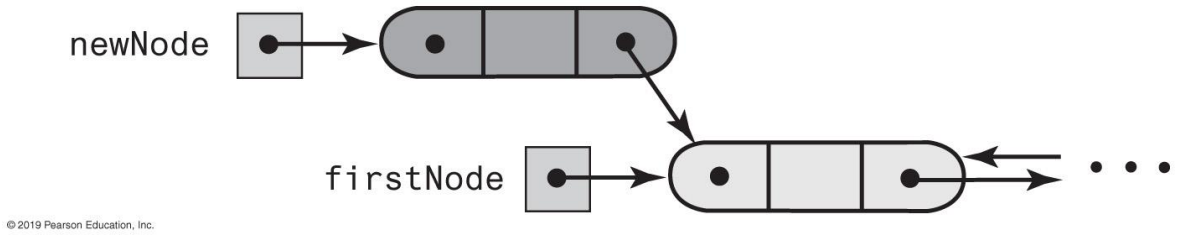


LDeque
-firstNode: DLNode -lastNode: DLNode
+addToBack(T newEntry): void **+addToFront(T newEntry): void** +getBack(): T +getFront(): T +removeFront(): T +removeBack(): T +isEmpty(): boolean +clear(): void

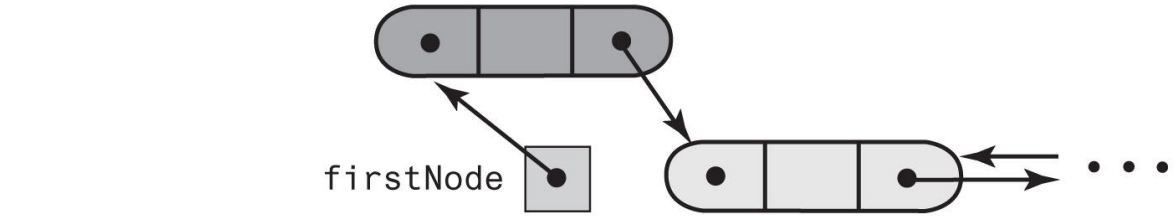
Doubly Linked Implementation of a Deque

- 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




```
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 entry

```
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
```

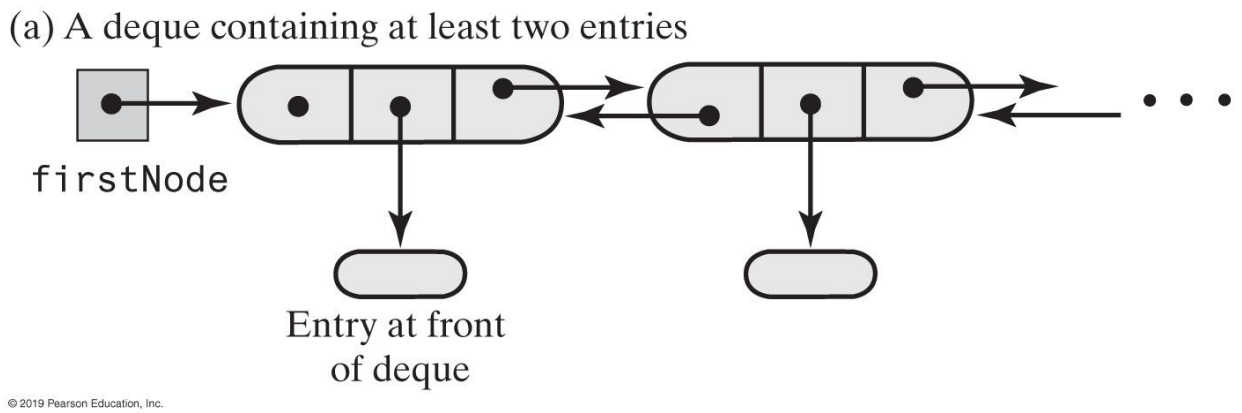
LDeque
-firstNode: DLNode -lastNode: DLNode
+addToBack(T newEntry): void +addToFront(T newEntry): void +getBack(): T +getFront(): T **+removeFront(): T** +removeBack(): T +isEmpty(): boolean +clear(): void

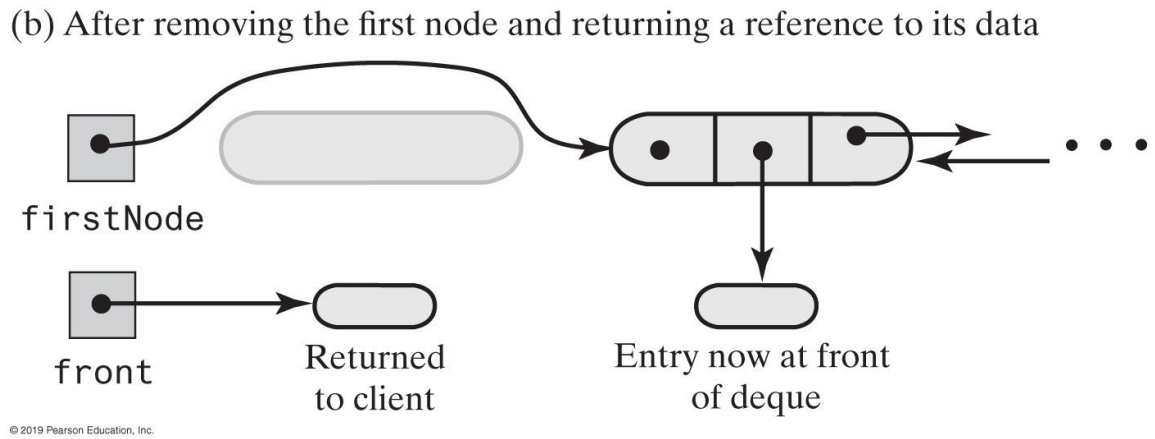
Doubly Linked Implementation of a Deque

- Removing the front entry

```
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
```

Removing the front of a deque containing at least two entries





Doubly Linked Implementation of a Deque

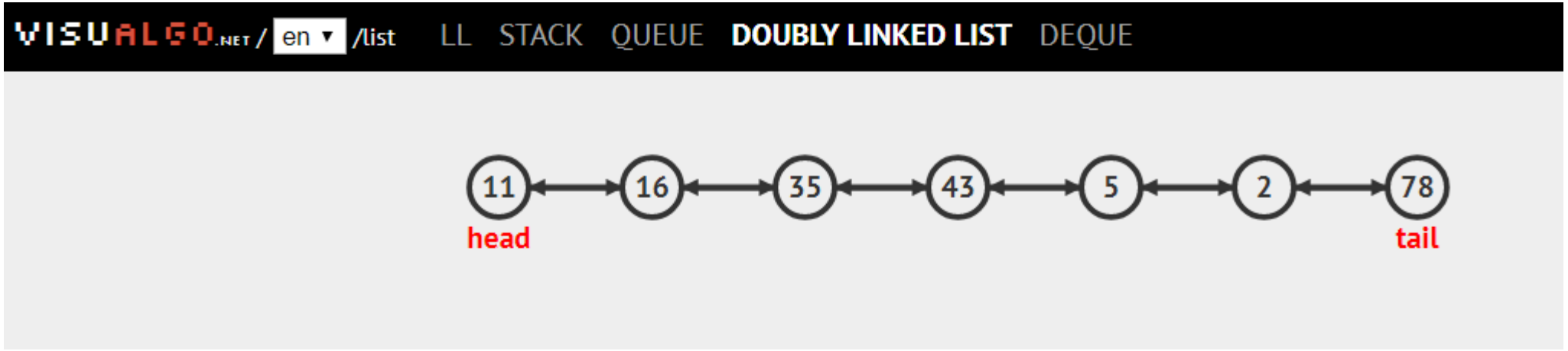
- Removing the back entry

```
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
```

LDeque
-firstNode: DLNode -lastNode: DLNode
+addToBack(T newEntry): void +addToFront(T newEntry): void +getBack(): T +getFront(): T +removeFront(): T **+removeBack(): T** +isEmpty(): boolean +clear(): void

Interactive and Visualization Demo

- https://visualgo.net/en/list



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

```
/** An interface for the 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();

  /** 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
```

Java Class Library: The Class PriorityQueue

- Basic constructors and methods
- PriorityQueue
  - `add`
  - `offer`
  - `remove`
  - `poll`
  - `element`
  - `peek`
  - `isEmpty, clear, size`

Implementation of Priority Queues

- Using queue,
  - Define an array of ordinary queues, called queues[].
  - The items with priority 0 are stored in queues[0]. Items with priority 1 are stored in queues[1]. And so on, up to queues[highest].
  - When an item with priority *i* needs to be added, we insert it to the end of queues[*i*].
  - When an item needs to be removed, we move down through the ordinary queues, starting with the highest priority, until we find a nonempty queue. We then remove the front item from this nonempty queue. For efficiency, we could keep a variable to remember the current highest priority.

Implementation of Priority Queues

- Using heap
  - Each node of the heap contains one element along with the element's priority,
  - The tree is maintained so that it follows the heap storage rules using the element's priorities to compare nodes:
    - The element contained by each node has a priority that is greater than or equal to the priorities of the elements of that node's children.
    - The tree is a complete binary tree.

A heap is the most efficient implementation of priority queues.

Summary

- Queue, Deque, and Priority Queue
- Implementations of a Queue

What I Want You to Do

- Review class slides
- Review Chapters 7 and 8