Class 04 – Queues

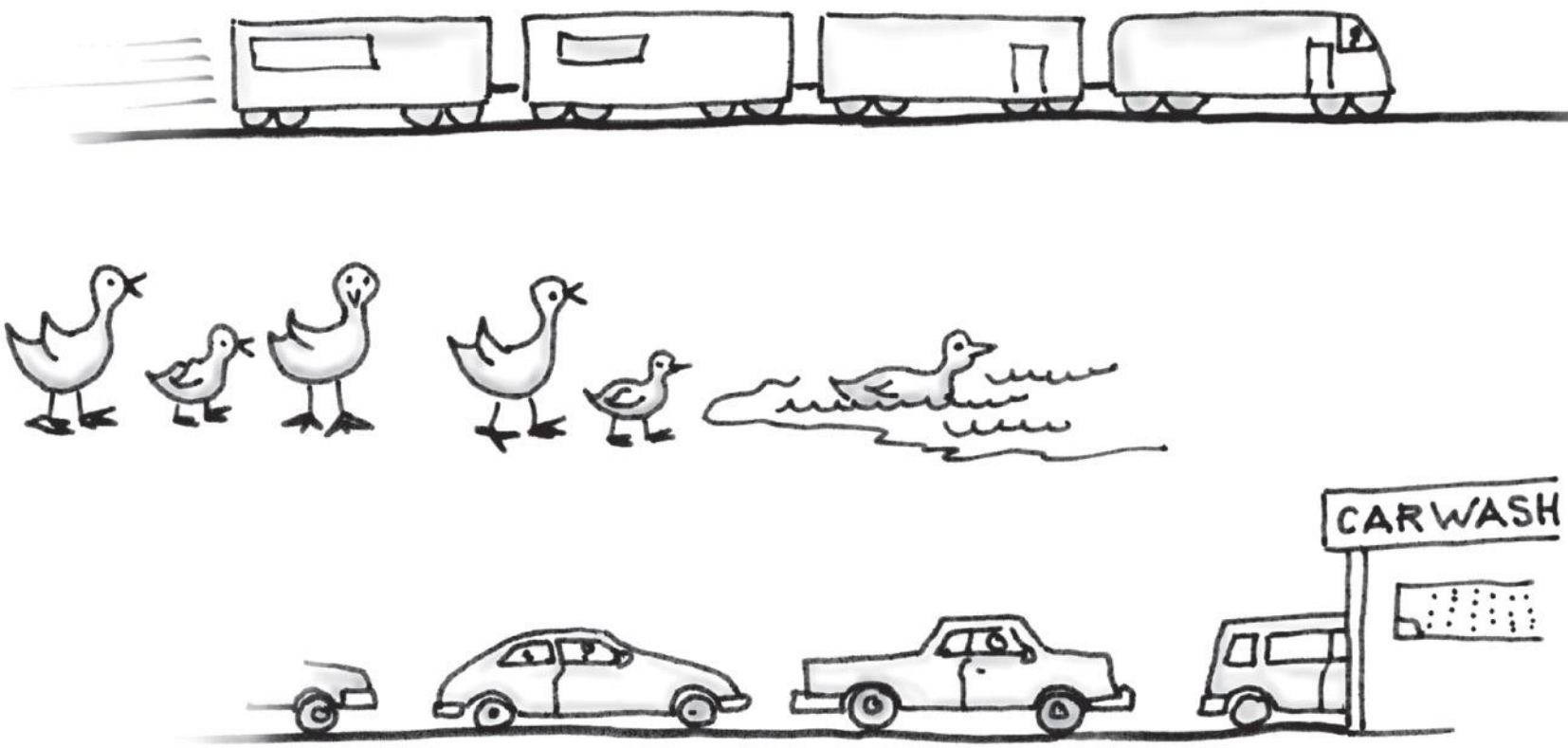
CSIS 3475 Data Structures and Algorithms

The ADT Queue

- A queue is another name for a waiting line
- Used within operating systems and to simulate real-world events
  - Come into play whenever processes or events must wait
- Entries organized first-in, first-out

The ADT Queue

- Some everyday queues



The ADT Queue

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

The ADT Queue

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

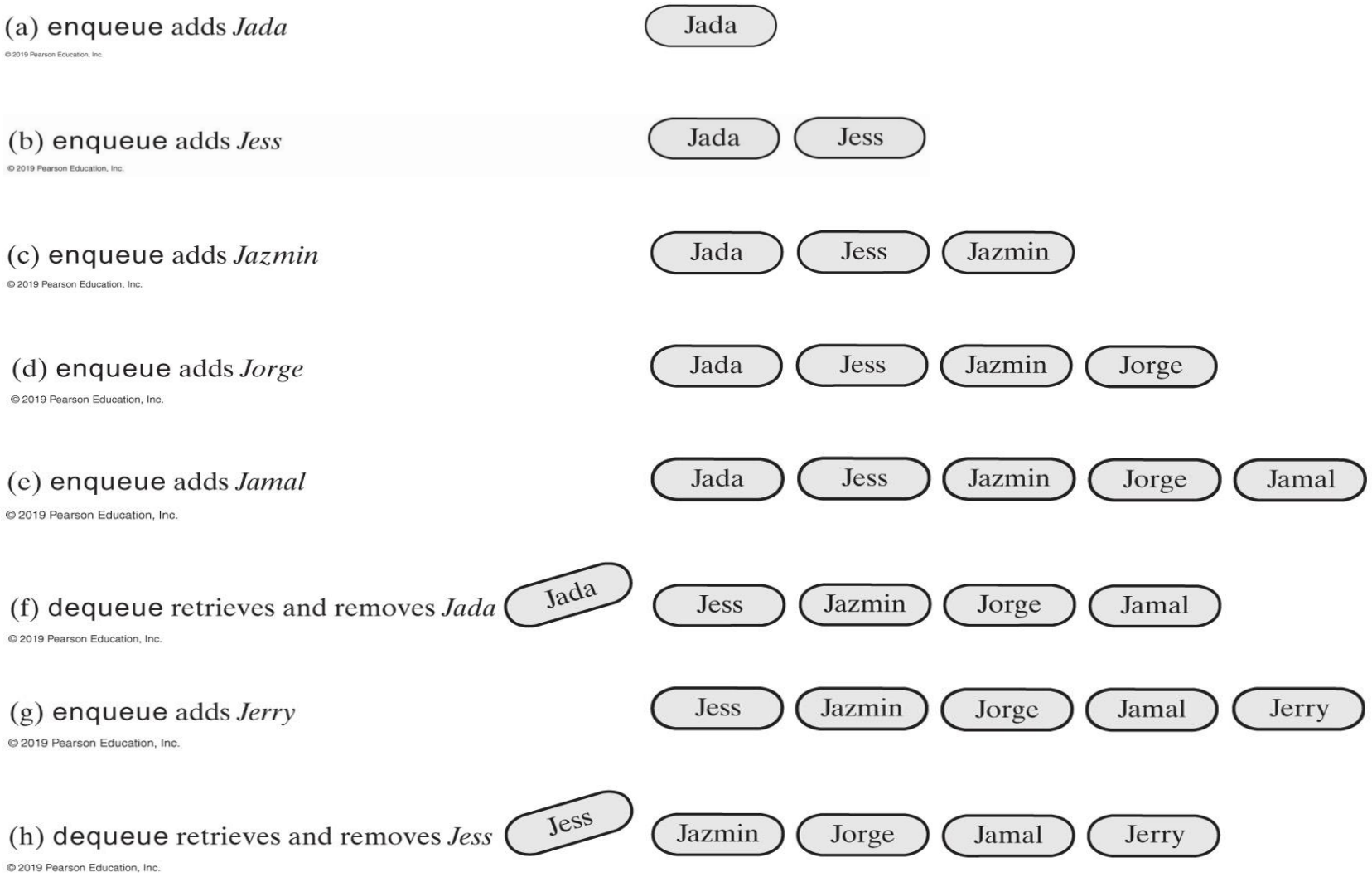
Pseudocode	UML	Description
enqueue(newEntry)	+enqueue(newEntry: integer): void	Task: Adds a new entry to the back of the queue. Input: newEntry is the new entry. Output: None.
dequeue()	+dequeue(): T	Task: Removes and returns the entry at the front of the queue. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty before the operation.
getFront()	+getFront(): T	Task: Retrieves the queue's front entry without changing the queue in any way. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the queue is empty. Input: None. Output: Returns true if the queue is empty.
clear()	+clear(): void	Task: Removes all entries from the queue. Input: None. Output: None.

+ isEmpty()
+ size().

Queue interface

```
public interface QueueInterface<T> {
      /**
       * Adds a new entry to the back of this queue.
       * Object may be null. Note this is different from java library spec.
       *
       * @param newEntry An object (possibly null) to be added.
       */
      public void enqueue(T newEntry);

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

      /**
       * Gets the size of the queue
       * @return queue size
       */
      public int size();
      /**
       * Gets an array consisting of a copy of
       * all elements in the queue
       * @return array of queue elements
       */
      public T[] toArray();
}
```

The effect of operations on a queue of strings



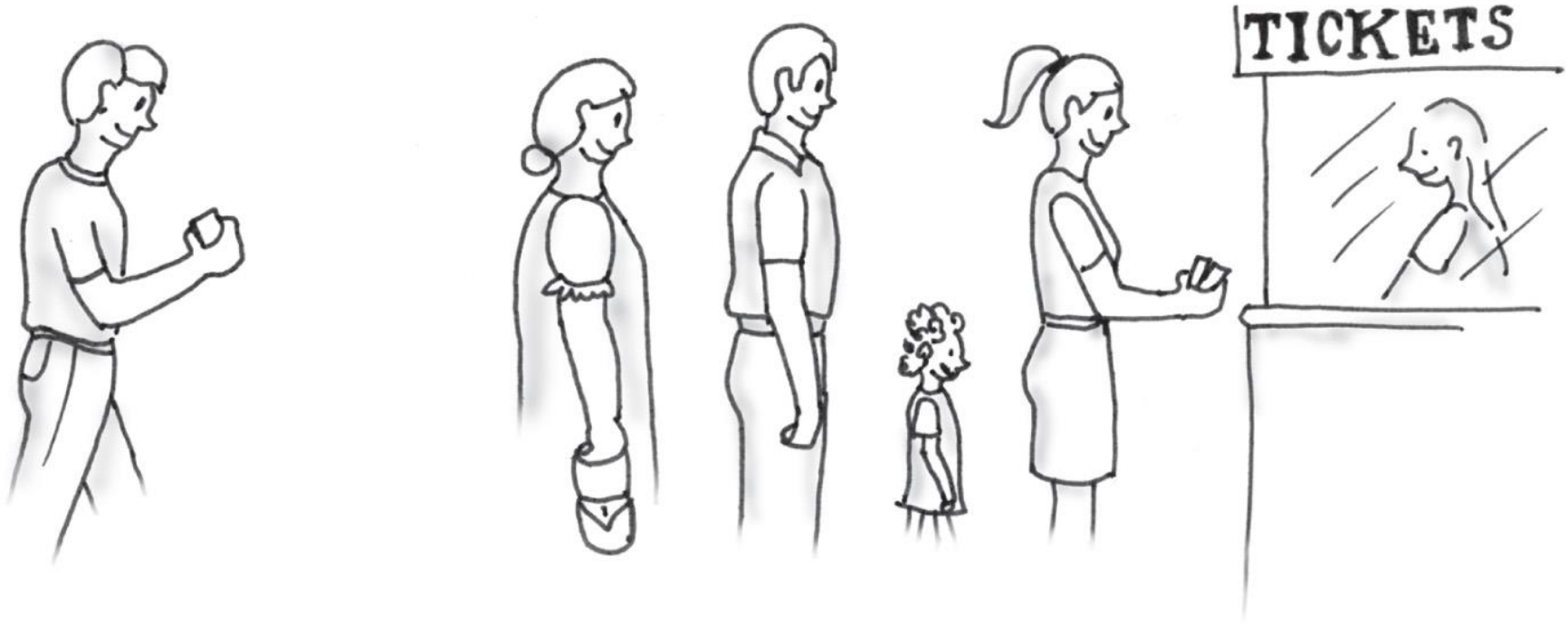
Java Class Library – Queue Interface

- Methods
  - add – throws exception
  - offer – like add(), returns null no exception thrown
  - remove – throws exception
  - poll – like remove(), no exception
  - element – returns element at index
  - peek – front of queue, no exception
  - isEmpty
  - size
- Nulls not allowed in data
- QueueUsingLibraryQueue class
  - Implements our QueueInterface using the Java library Queue Interface

QueueUsingLibraryQueue class

```
public class QueueUsingLibraryQueue<T> implements
QueueInterface<T> {

    // internal space for queue entries

    private Queue<T> queue;

    public QueueUsingLibraryQueue() {
        // need to use one of the classes that
implements Queue interface
        queue = new ArrayDeque<>();
//      queue = new LinkedList<>();
    }

    @Override
    public void enqueue(T newEntry) {
        // does not throw an exception if no
capacity
        queue.offer(newEntry);
    }

    public T dequeue() {
        if(isEmpty())
            throw new EmptyQueueException();
        return queue.poll();
    }

    public T getFront() {
        if(isEmpty())
            throw new EmptyQueueException();
        return queue.peek();
    }

    public boolean isEmpty() {
        return queue.isEmpty();
    }

    public void clear() {
        queue.clear();

    }

    public int size() {
        return queue.size();
    }

    @SuppressWarnings("unchecked")
    @Override
    public T[] toArray() {
        return (T[]) queue.toArray();
    }
}
```

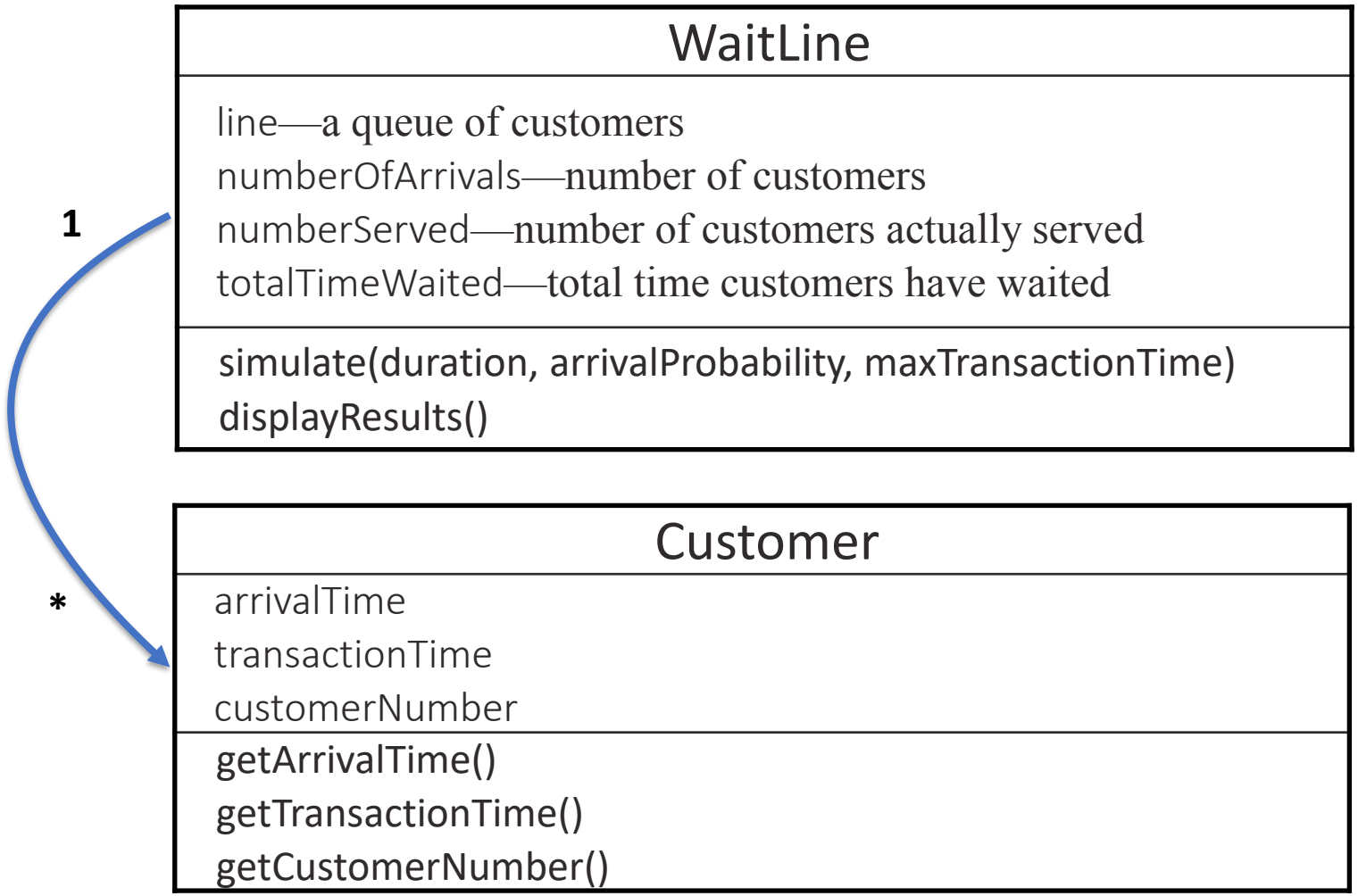
Simulating a Waiting Line



A CRC card for the class WaitLine

WaitLine
Responsibilities Simulate customers entering and leaving a waiting line Display number served, total wait time, average wait time, and number left in line
Collaborations Customer

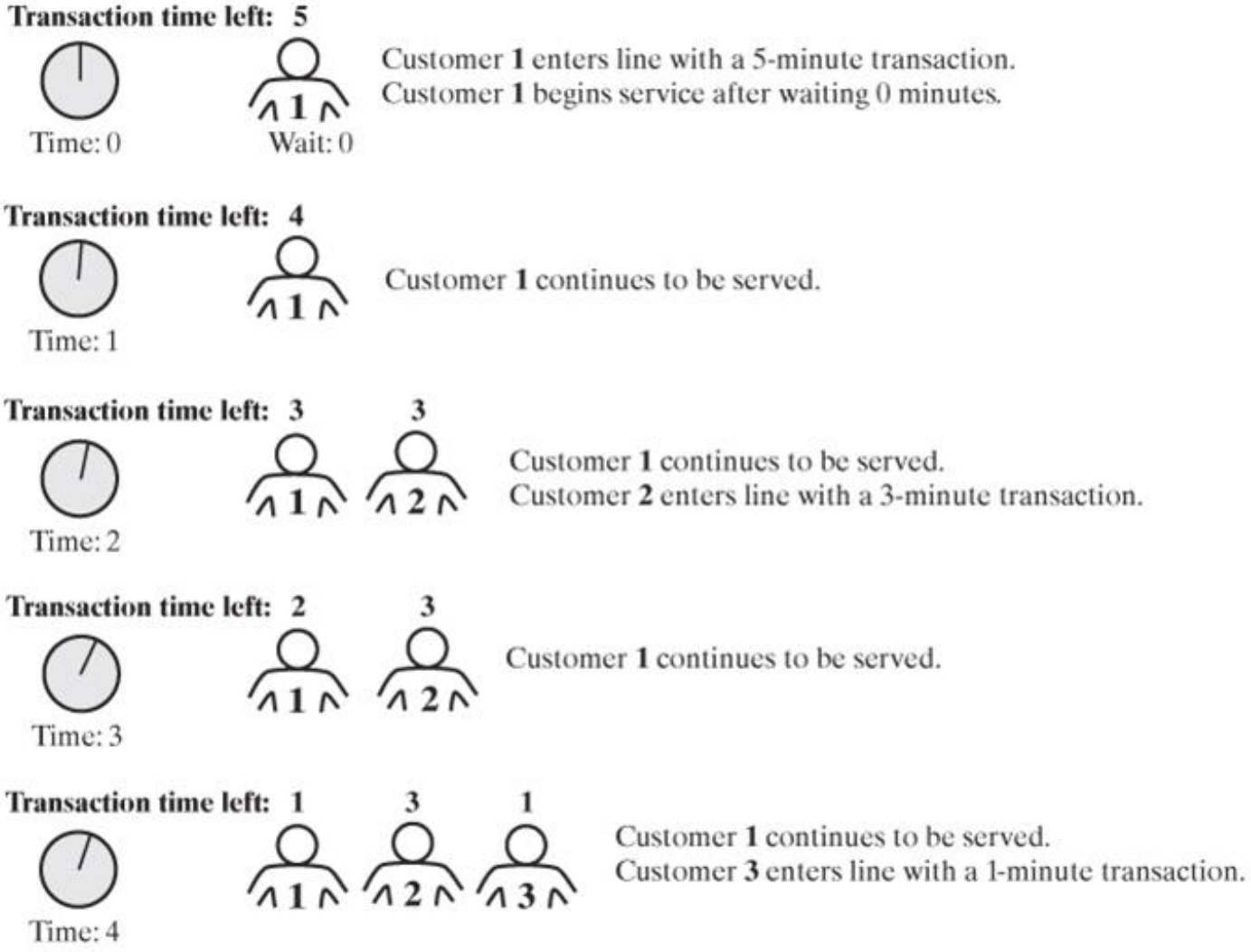
A diagram of the classes WaitLine and Customer



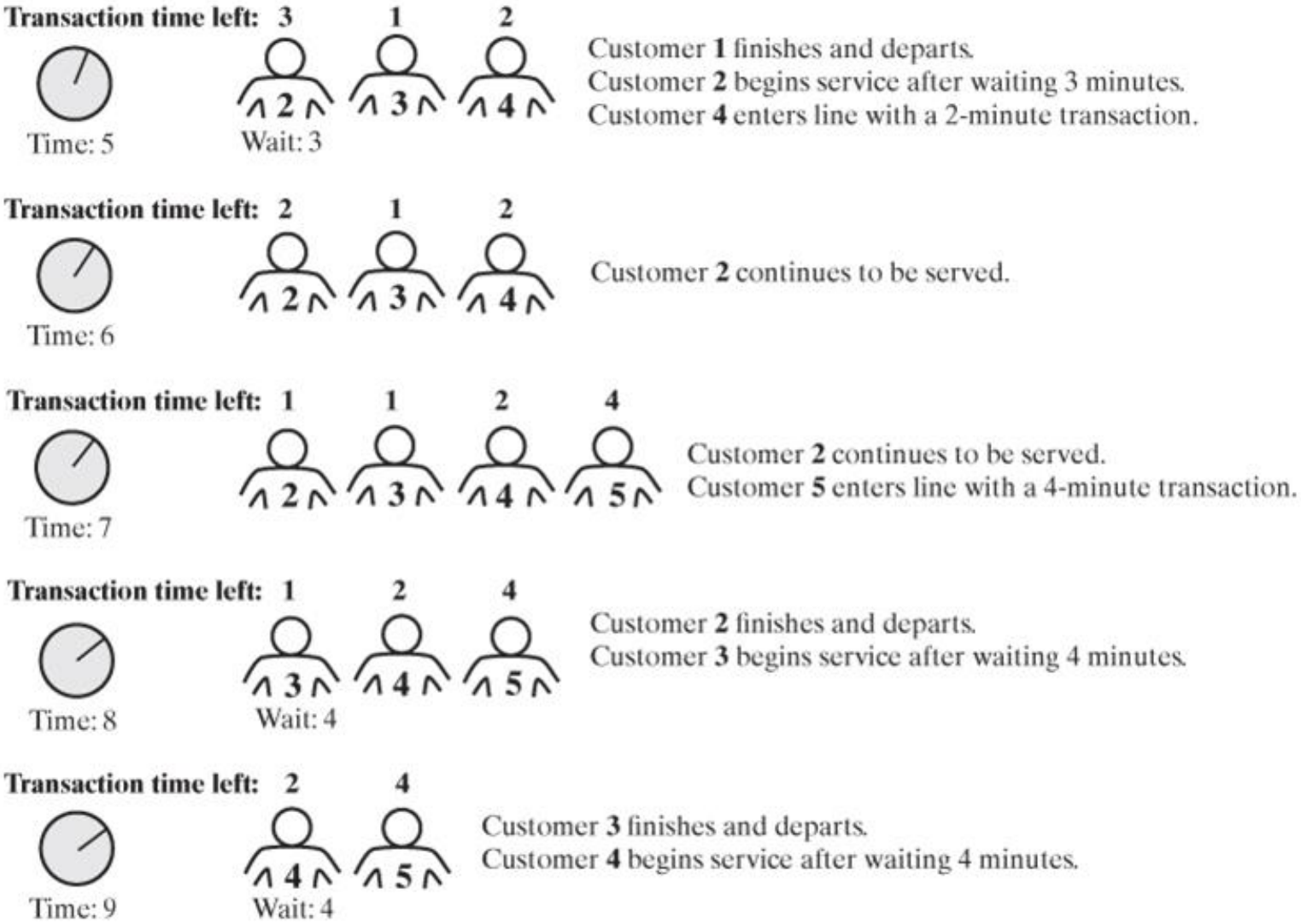
Algorithm for Simulating a Waiting Line

```
Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
transactionTimeLeft = 0
for (clock = 0; clock < duration; clock++)
{
        if (a new customer arrives)
        {
                numberOfArrivals++
                transactionTime = a random time that does not exceed
            maxTransactionTime
                nextArrival = a new customer containing clock, transactionTime, and
                                a customer number that is
                        numberOfArrivals
                line.enqueue(nextArrival)
        }
        if (transactionTimeLeft > 0) // If present customer is still being served
                transactionTimeLeft−−
        else if (!line.isEmpty())
        {
                nextCustomer = line.dequeue()
                transactionTimeLeft = nextCustomer.getTransactionTime() − 1
                timeWaited = clock − nextCustomer.getArrivalTime()
                 totalTimeWaited = totalTimeWaited  +  timeWaited
            numberServed++
        }
}
```

Simulated Waiting Line



Simulated Waiting Line



Simulating a waiting line

- Use a queue of Customers
- The line counts the number of arrivals, number served, and total time waited.

```
public class WaitLine {
	private QueueInterface<Customer> line;
	private int numberOfArrivals;
	private int numberServed;
	private int totalTimeWaited;

	public WaitLine() {

		// modified to use QueueUsingLibraryQueue, which is based on the java library Queue interface
		// other ADTs can be substituted here, as long as they implement the QueueInterface

//		line = new CompletedLinkedQueue<>();
//		line = new CompletedArrayQueue<>();
//		line = new CompletedTwoPartCircularLinkedQueue<Customer>();

		line = new QueueUsingLibraryQueue<Customer>();
//		line = new LinkedQueue<>();
//		line = new TwoPartCircularLinkedQueue<Customer>();
//		line = new ArrayQueue<>();

		reset();
	}
```

Simulating a waiting line

- Simulation runs for a set of (fake) minutes, really integer ticks
- Customers are generated at a certain time with a probability.
- Maximum transaction time is set as an upper bound

```
      /**
       * Simulates a waiting line with one serving agent.
       *
       * @param duration           The number of simulated minutes.
       * @param arrivalProbability A real number between 0 and 1, and the probability
       *                           that a customer arrives at a given time
       * @param maxTransactionTime The longest transaction time for a customer
       */
      public void simulate(int duration, double arrivalProbability, int maxTransactionTime)
{
```

Enqueueing the customer for service

- If a random number is less than the probability, create a Customer with a random transaction time.
- Arrival number is actually the customer number
- Enqueue the customer

```
        int transactionTimeLeft = 0;

        // the clock ticks away

        for (int clock = 0; clock < duration; clock++) {

            // pick a random number. If it is less than the probability, let them in
            if (Math.random() < arrivalProbability) {
                numberOfArrivals++;

                // set the transaction time (time the customer waits in the queue)

                int transactionTime = (int) (Math.random() * maxTransactionTime + 1);
                Customer nextArrival = new Customer(clock, transactionTime, numberOfArrivals);

                line.enqueue(nextArrival);
                System.out.println("Customer " + numberOfArrivals + " enters line at time " + clock
                        + ". Transaction time is " + transactionTime);


            }
```

Serve the Customer

- Count down the transaction timer
- If it is zero, then dequeue a Customer.
  - Set the Customer's transaction time to the transaction timer
- Update statistics

```
  // if nobody is waiting just loop back around

  if (transactionTimeLeft > 0)
    transactionTimeLeft--;
  else if (!line.isEmpty()) {

    // timer for the customer waiting expired, so dequeue them, and
    //
    Customer nextCustomer = line.dequeue();
    transactionTimeLeft = nextCustomer.getTransactionTime() - 1;
    int timeWaited = clock - nextCustomer.getArrivalTime();
    totalTimeWaited = totalTimeWaited + timeWaited;
    numberServed++;
    System.out.println("Customer " + nextCustomer.getCustomerNumber() + " begins service at time " +
clock
        + ". Transaction time left is " + transactionTimeLeft + ". Time waited is " + timeWaited);
  }
 }
```

Other methods

- display() shows statistics
- reset() initializes everything

```
/**
 * Displays summary results of the simulation.
 */
public void displayResults() {
      System.out.println();
      System.out.println("Number served = " + numberServed);
      System.out.println("Total time waited = " + totalTimeWaited);
      double averageTimeWaited = ((double) totalTimeWaited) / numberServed;
      System.out.println("Average time waited = " + averageTimeWaited);
      int leftInLine = numberOfArrivals - numberServed;
      System.out.println("Number left in line = " + leftInLine);
} // end displayResults

/** Initializes the simulation. */
public final void reset() {
      line.clear();
      numberOfArrivals = 0;
      numberServed = 0;
      totalTimeWaited = 0;
}
```

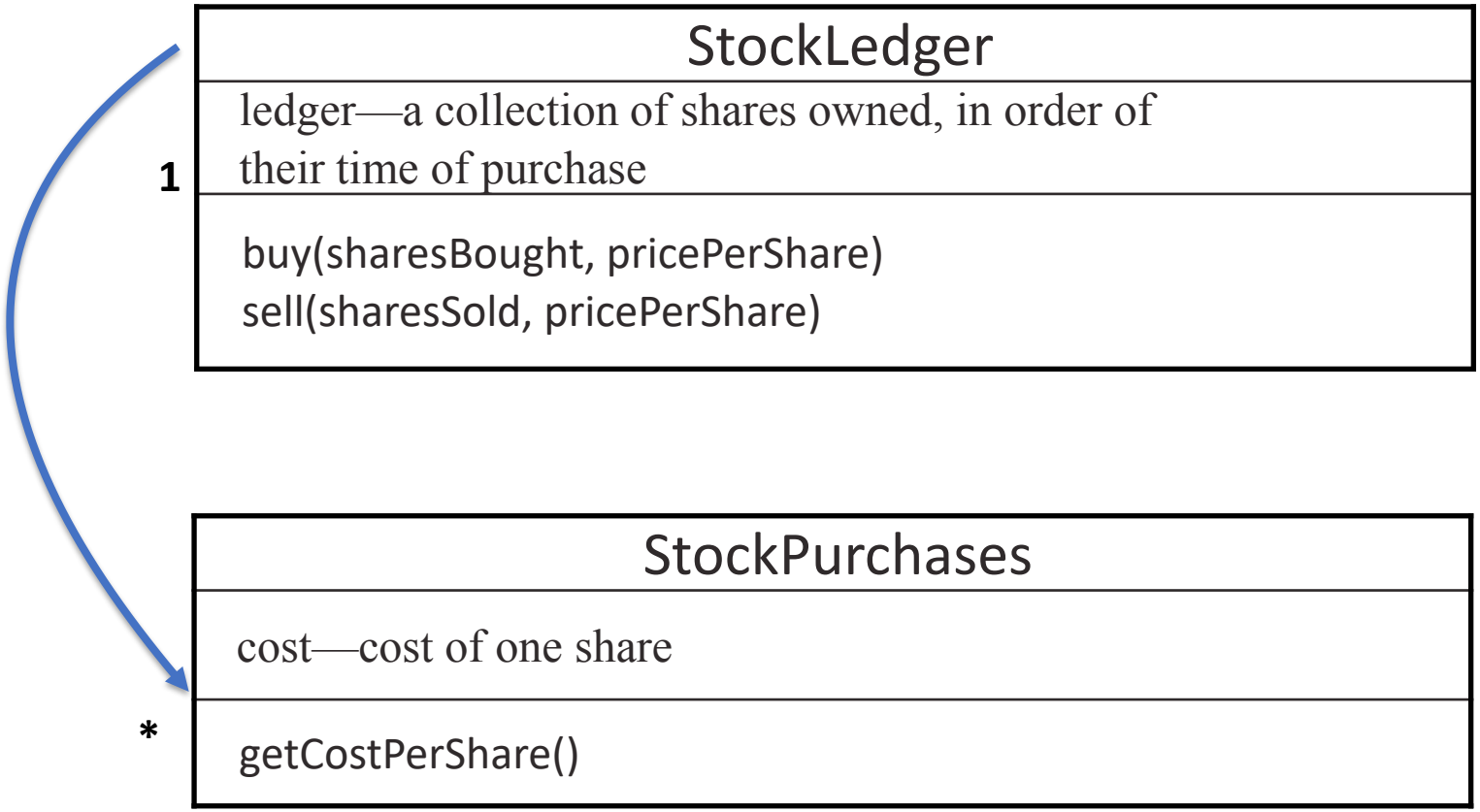
Computing the Capital Gain in a Sale of Stock

- A CRC card for the class `StockLedger`

StockLedger
Responsibilities
Record the shares of a stock purchased,
in chronological order
Remove the shares of a stock sold,
beginning with the ones held the longest
Compute the capital gain (loss) on shares of a
stock sold
Collaborations
Share of stock

Computing the Capital Gain in a Sale of Stock

- A diagram of the classes StockLedger and StockPurchase



StockPurchase

- Simply the cost of the stock per share

```
public class StockPurchase {
            private double cost;

            public StockPurchase(double cost) {
                        this.cost = cost;
            }

            public double getCostPerShare() {
                        return cost;
            }
}
```

StockLedger – buy()

- Each share of stock bought is put on the queue

```
public class StockLedger  {
     private QueueInterface<StockPurchase> ledger;

     public StockLedger() {
          // can use any other class that implements QueueInterface here

//        ledger = new CompletedLinkedQueue<StockPurchase>();
//        ledger = new CompletedArrayQueue<StockPurchase>();
//        ledger = new CompletedTwoPartCircularLinkedQueue<StockPurchase>();
          ledger = new QueueUsingLibraryQueue<StockPurchase>();
//        ledger = new LinkedQueue<StockPurchase>();
//        ledger = new ArrayQueue<StockPurchase>();
//        ledger = new TwoPartCircularLinkedQueue<StockPurchase>();
     }

     /**
      * Records a stock purchase in this ledger.
      *
      * @param sharesBought  The number of shares purchased.
      * @param pricePerShare The price per share.
      */
     public void buy(int sharesBought, double pricePerShare) {
          while (sharesBought > 0) {
               StockPurchase purchase = new StockPurchase(pricePerShare);
               ledger.enqueue(purchase);
               sharesBought--;
          }
     }
```

StockLedger – sell(), display()

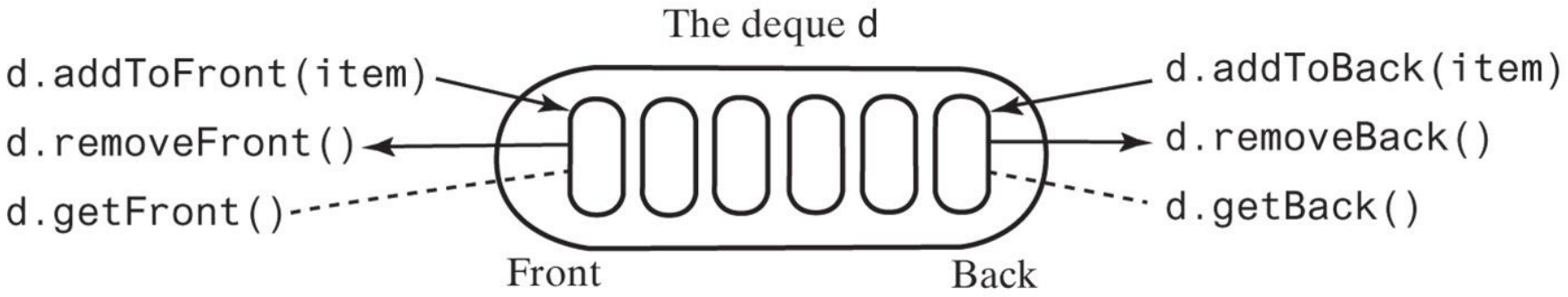
- Each share sold at a price is removed from the queue.
- Gain or loss is calculated

```
/**
 * Removes from this ledger any shares that were sold and computes the capital
 * gain or loss.
 *
 * @param sharesSold    The number of shares sold.
 * @param pricePerShare The price per share.
 * @return The capital gain (loss).
 */
public double sell(int sharesSold, double pricePerShare) {
    double saleAmount = sharesSold * pricePerShare;
    double totalCost = 0;

    while (sharesSold > 0) {
        StockPurchase share = ledger.dequeue();
        double shareCost = share.getCostPerShare();
        totalCost = totalCost + shareCost;
        sharesSold--;
    }

    return saleAmount - totalCost; // Gain or loss
}

/**
 * Show all stocks in the ledger
 */
public void display() {
    Object[] stocks = ledger.toArray();

    System.out.println("Stocks held");
    for(Object stock : stocks)
        System.out.println("Stock Price " + ((StockPurchase) stock).getCostPerShare());
}
```

The ADT `Deque`

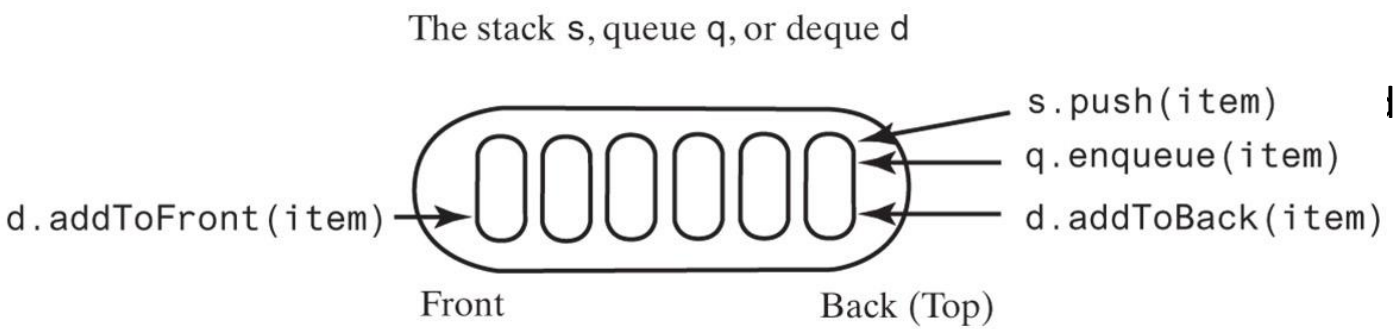
- A **double ended** queue
- `Deque` pronounced “deck”
- Has both queue-like operations and stack-like operations

An instance d of a deque

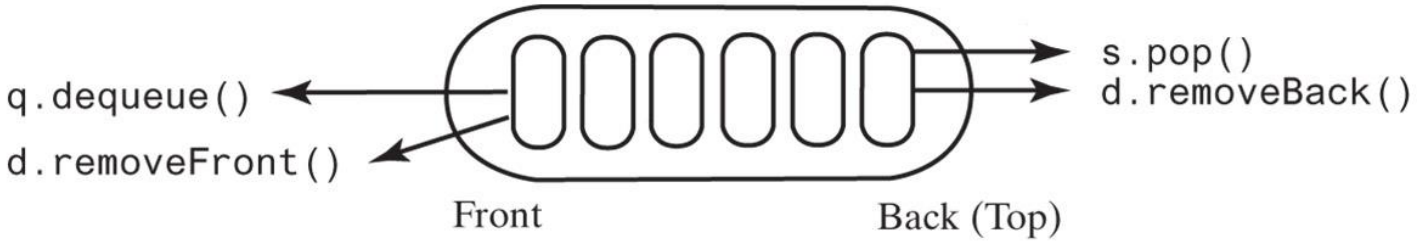


A comparison of operations for a stack s, a queue q, and a deque d

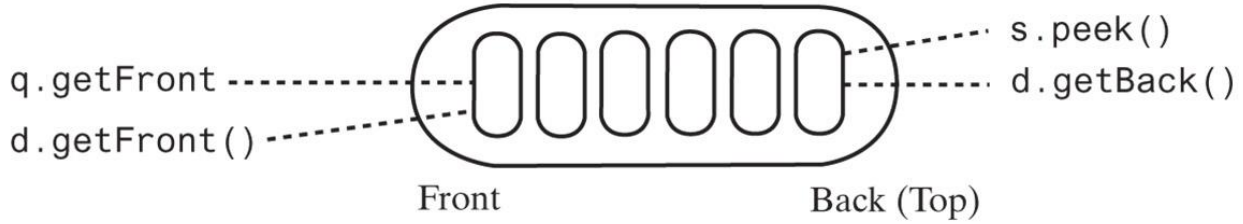
(a) Add



(b) Remove



(c) Retrieve



Deque (deck) interface

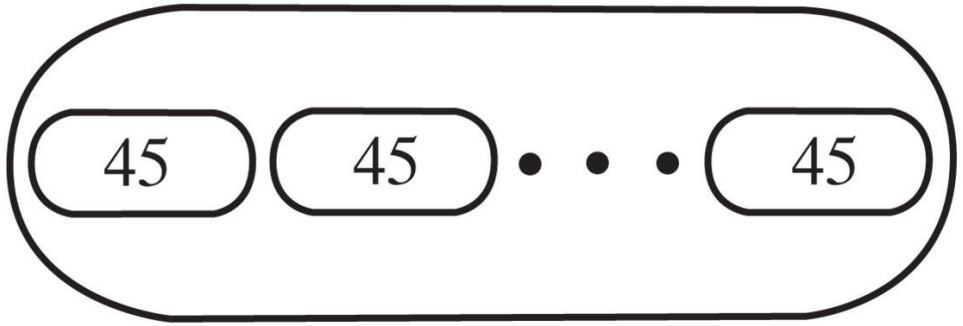
```
public interface DequeInterface<T> {
      /**
       * Adds a new entry to the front of this deque.
       *
       * @param newEntry An object to be added.
       */
      public void addToFront(T newEntry);

      /**
       * Adds a new entry to the back of this deque.
       *
       * @param newEntry An object to be added.
       */
      public void addToBack(T newEntry);

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

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

      /**
       * Retrieves the front entry of this deque.
       *
       * @return front of the deque
       * @throws  EmptyQueueException if the deque is empty
before the operation.
       */
      public T getFront();

      /**
       * Retrieves the front entry of this deque.
       *
       * @return back of the deque
       * @throws  EmptyQueueException if the deque is empty
before the operation.
       */
      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();

      /**
       * Gets the size of the deque
       * @return queue size
       */
      public int size();

      /**
       * Gets an array consisting of a copy of
       * all elements in the deque
       * @return array of queue elements
       */
      public T[] toArray();
}
```

Using deque to read/display input

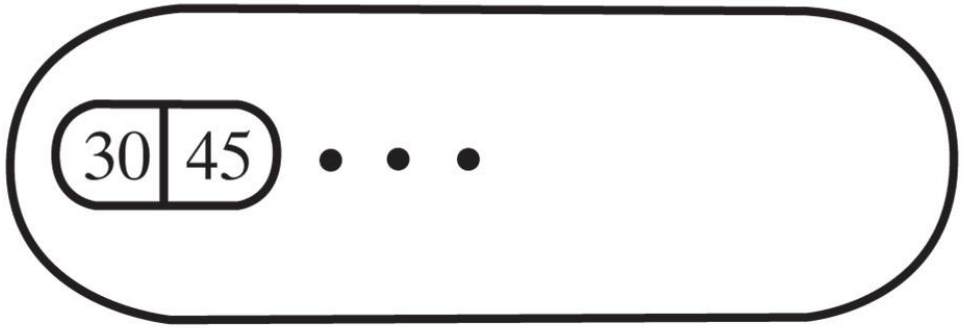
```
// Read a line
d = a new empty deque
while (not end of line)
{
        character = next character read
        if (character == ←)
                d.removeBack()
        else
                d.addToBack(character)
}
// Display the corrected line
while (!d.isEmpty())
        System.out.print(d.removeFront())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

Two representations of stock shares in a queue

(a) Individual shares of stock in a queue



(b) Grouped shares of stock as objects in a queue



Modify StockLedger to use Deque

- Change StockPurchase to include number of shares and cost
- To buy a stock, add it to the back of the queue.
- To sell a stock, remove from the back.
  - If it is more shares than desired
    - Create a new purchase of leftover shares
    - Put on the **front,** replacing the old one.
- Complete solution is not provided.

StockLedgerUsingDeque – sell()

```
/**
 * Removes from this ledger any shares that were sold and computes the capital
 * gain or loss.
 *
 * @param sharesSold    The number of shares sold.
 * @param pricePerShare The price per share.
 * @return The capital gain (loss).
 */
public double sell(int sharesSold, double pricePerShare) {
   double saleAmount = sharesSold * pricePerShare;
   double totalCost = 0;

   while (sharesSold > 0) {
      StockPurchaseUsingDeque transaction = ledger.removeFront();
      double shareCost = transaction.getCostPerShare();
      int numberOfShares = transaction.getNumberOfShares();

      if (numberOfShares > sharesSold) {
         totalCost = totalCost + sharesSold * shareCost;
         int numberToPutBack = numberOfShares - sharesSold;
         StockPurchaseUsingDeque leftOver = new StockPurchaseUsingDeque(numberToPutBack, shareCost);
         ledger.addToFront(leftOver); // Return leftover shares
         // Note: Loop will exit since sharesSold will be <= 0 later
      } else
         totalCost = totalCost + numberOfShares * shareCost;

      sharesSold = sharesSold - numberOfShares;
   }

   return saleAmount - totalCost; // Gain or loss
}
```

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`
- `add/remove throw exceptions for empty queue`
- `offer/poll/peek do not`

Java Class Library: The Class `ArrayDeque`

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

Use of ArrayDeque

- Implement QueueInterface
- Can be used in QueueDemo to test

```
public class QueueUsingLibraryQueue<T> implements
QueueInterface<T> {

    // internal space for queue entries

    private Queue<T> queue;

    public QueueUsingLibraryQueue() {
        // need to use one of the classes that
implements Queue interface
        queue = new ArrayDeque<>();
//      queue = new LinkedList<>();
    }

    @Override
    public void enqueue(T newEntry) {
        // does not throw an exception if no capacity
        queue.offer(newEntry);
    }

    public T dequeue() {
        if(isEmpty())
            throw new EmptyQueueException();
        return queue.poll();
    }

    public T getFront() {
        if(isEmpty())
            throw new EmptyQueueException();
        return queue.peek();
    }

    public boolean isEmpty() {
        return queue.isEmpty();
    }

    public void clear() {
        queue.clear();

    }

    public int size() {
        return queue.size();
    }

    @SuppressWarnings("unchecked")
    @Override
    public T[] toArray() {
        return (T[]) queue.toArray();
    }
}
```

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

Java library PriorityQueue

- Adds items and orders them by priority
- What is priority?
  - One item comes before another
- This is determined by requiring an object implements Comparable
  - Incoming item (via add()) is compared to each item in the queue and is placed according to the result of compareTo()

Tracking Your Assignments

- UML diagrams of the class `Assignment` and `AssignmentLog`

Assignment
course—the course code task—a description of the assignment date—the due date
getCourseCode() getTask() getDueDate() compareTo()

AssignmentLog
log—a priority queue of assignments
addProject(newAssignment) addProject(courseCode, task, dueDate) getNextProject() removeNextProject()

Java Class Library: The Class `PriorityQueue`

- Basic methods
  - add
  - offer
  - remove
  - poll
  - element
  - peek
  - isEmpty, clear, size

Assignment

- Implements Comparable
  - Note compareTo() required for PriorityQueue

```java
import java.sql.Date;

/**
 * An assignment has a course, a task, and a due date.
 * Implements Comparable so that java library PQ can use it to insert
 * @author mhrybyk
 *
 */
public class Assignment implements Comparable<Assignment> {

	private String course; // course code
	private String task;   // task or assignment description
	private Date date;     // due date

	/**
	 * Create a new assignment from arguments
	 * @param newCourse
	 * @param newTask
	 * @param newDueDate
	 */
	Assignment(String newCourse, String newTask, Date newDueDate) {
		course = newCourse;
		task = newTask;
		date = newDueDate;
	}

	/**
	 * Get the course code
	 * @return
	 */
	public String getCourseCode() {
		return course;
	}

	/**
	 * Compare due dates for assignment. Sooner is better.
	 */
	public int compareTo(Assignment other) {
		return date.compareTo(other.date);
	}

	/**
	 * Get the task
	 * @return
	 */
	public String getTask() {
		return task;
	}

	/**
	 * Get the due date
	 * @return
	 */
	public Date getDueDate() {
		return date;
	}

	@Override
	public String toString() {
		return "Assignment: " + course + " : " + task + " : " +
date;
	}
}
```

AssignmentLog using PQ

- Keeps an internal PQ
- Adds projects using offer()
- offer() uses compareTo()

```
import java.util.PriorityQueue;
import java.sql.Date;

/**
 * A class that represents a log of assignments ordered by priority.
 *
 * @author Frank M. Carrano
 * @author Timothy M. Henry
 * @version 5.0
 *
 * @author mhrybyk
 * Added display method
 */
public class AssignmentLogUsingLibraryPQ {

	// Use java library PQ for the log
	private PriorityQueue<Assignment> log;

	/**
	 * Create a new Assignment log
	 */
	public AssignmentLogUsingLibraryPQ() {
		log = new PriorityQueue<>();
	}

	/**
	 * Add an assignment to the log
	 * @param newAssignment
	 */
	public void addProject(Assignment newAssignment) {
		log.offer(newAssignment);
	}

	/**
	 * Add a new assignment from course code, task, and due date
	 * @param courseCode
	 * @param task
	 * @param dueDate
	 */
	public void addProject(String courseCode, String task, Date dueDate) {
		Assignment newAssignment = new Assignment(courseCode, task, dueDate);
		addProject(newAssignment);
	}
```

AssignmentLog using PQ

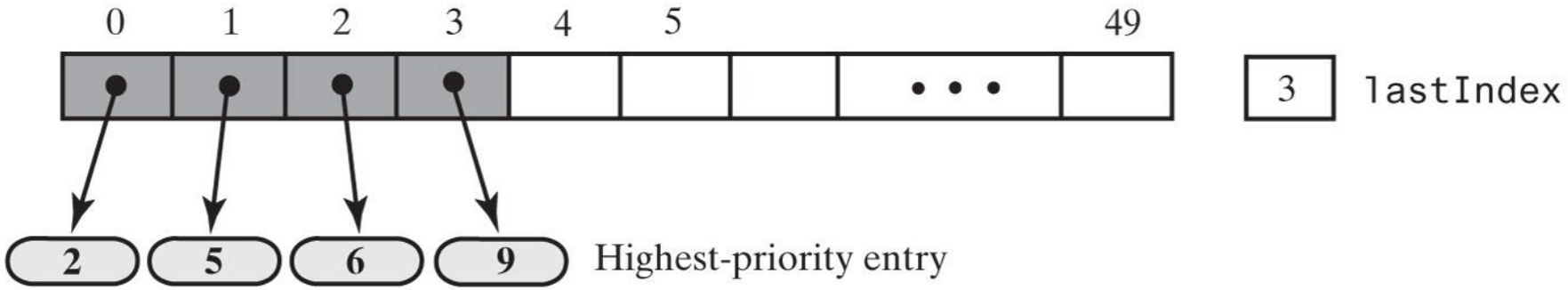
```
	/**
	 * Get the next assignment to be done according to priority
	 * @return
	 */
	public Assignment getNextProject() {
		return log.peek();
	} // end getNextProject

	/**
	 * Remove the top priority project from the queue
	 * @return assignment
	 */
	public Assignment removeNextProject() {
		return log.poll();
	} // end removeNextProject

	/**
	 * Get the number of assignments in the queue
	 * @return
	 */
	public int getNumberOfAssignments() {
		return log.size();
	}

	/**
	 * Display all assignments in the queue
	 */
	public void displayAssignments() {

		// this allows toArray() to determine type without giving size

		Assignment[] assignments = log.toArray(new Assignment[0]);

		System.out.println("Assignment log");
		for(Assignment assignment : assignments)
			System.out.println(assignment);
		System.out.println();

	}
```

AssignmentLogDemo

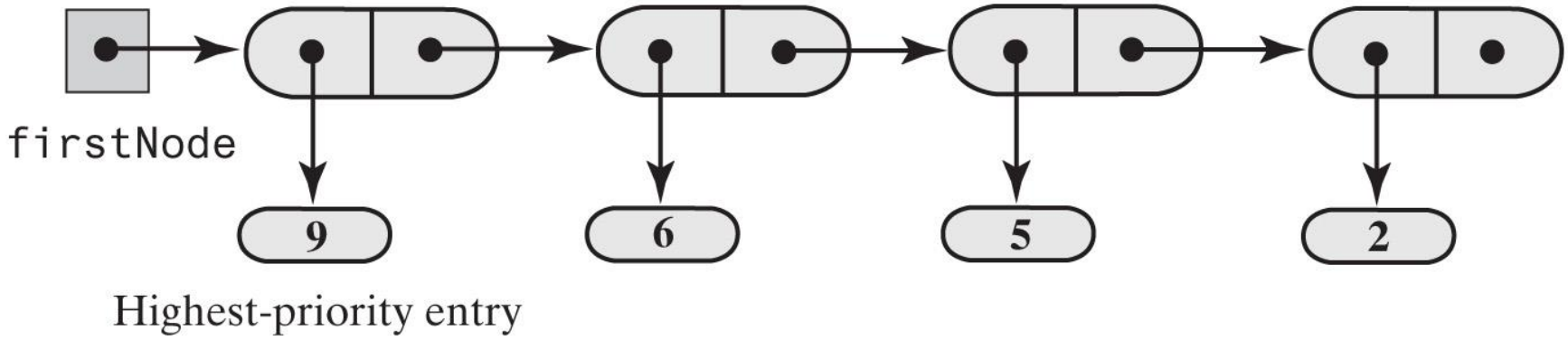
```
public class AssignmentLogDemo {

	public static void main(String[] args) {
		AssignmentLogUsingLibraryPQ myHomework = new AssignmentLogUsingLibraryPQ();

		// add a few assignments
		// demos use of different constructors

		myHomework.addProject("CSC211", "Pg 50, Ex 2", Date.valueOf("2019-4-20"));
		myHomework.addProject("CSC210", "Pg 55, Ex 7", Date.valueOf("2019-5-20"));

		Assignment pg75Ex8 = new Assignment("CSC215", "Pg 75, Ex 8", Date.valueOf("2019-3-14"));
		myHomework.addProject(pg75Ex8);

		myHomework.displayAssignments();
		// note that the next assignment will be the earliest one due

		// show the next assignment, and then remove it. Repeat for all.

		showNextAssignment(myHomework);
		myHomework.removeNextProject();
		System.out.println("Assignment done\n");

		showNextAssignment(myHomework);
		myHomework.removeNextProject();
		System.out.println("Assignment done\n");

		myHomework.displayAssignments();

		showNextAssignment(myHomework);
		myHomework.removeNextProject();
		System.out.println("Assignment done\n");

		System.out.println("Assignments finished");

		showNextAssignment(myHomework);

	}

	public static void showNextAssignment(AssignmentLogUsingLibraryPQ log) {
		System.out.println("The following assignment is due next:");
		System.out.println(log.getNextProject());
		System.out.println("Number left to be done: " + log.getNumberOfAssignments());
	}
}
```

Possible Implementations of a Priority Queue

(a) Array based

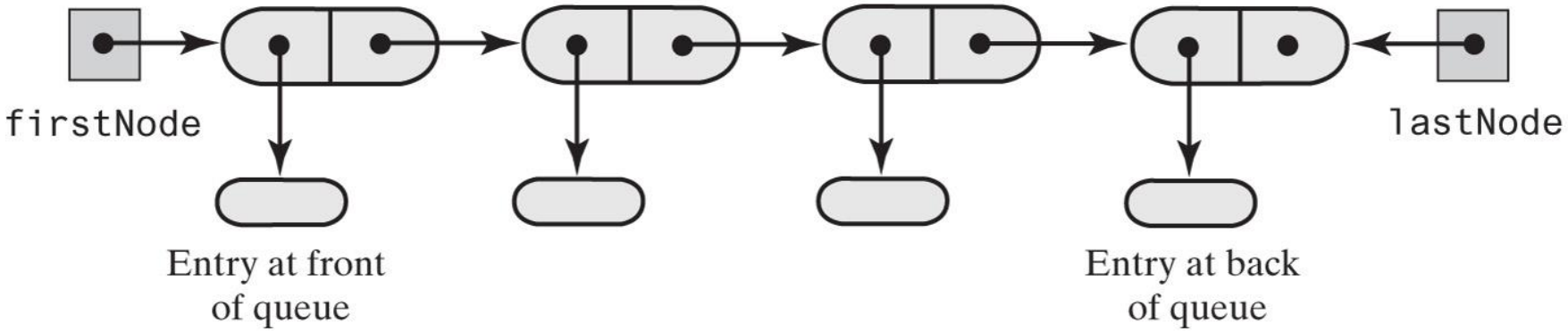


(b) Link based



Linked Implementation of a Queue

- A chain of linked nodes that implements a queue

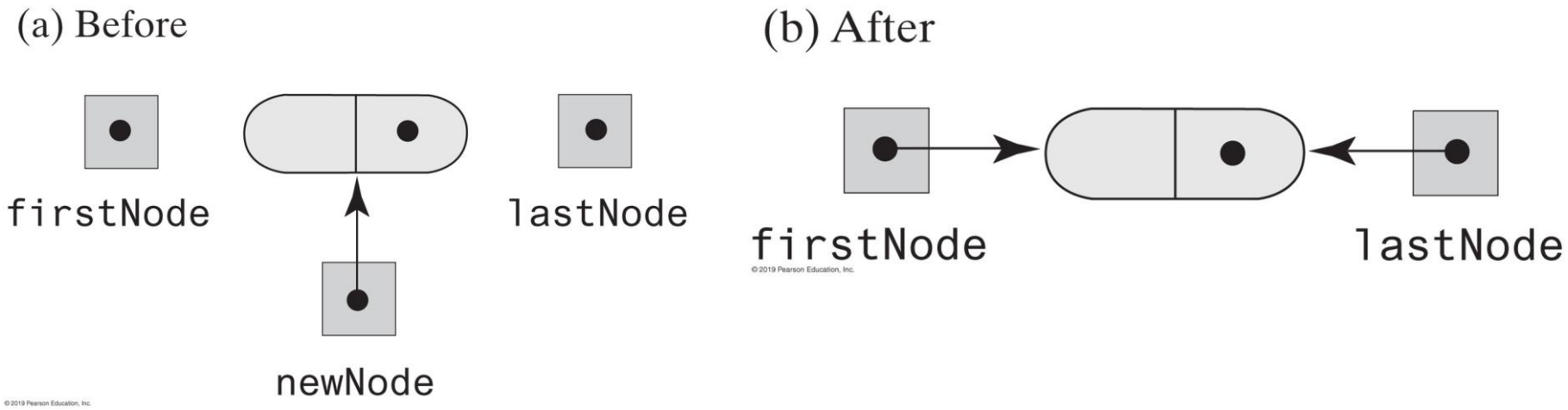


Node class – used for linked queue

```
public class Node<T> {
    private T data; // Entry in bag
    private Node<T> next; // Link to next node

    /**
     * Create a new node containing data
     * @param dataPortion
     */
    public Node(T dataPortion) {
        this(dataPortion, null);
    }

    /**
     * Create a new node containing data
     * and set the next node.
     * @param dataPortion
     * @param nextNode
     */
    public Node(T dataPortion, Node<T> nextNode) {
        data = dataPortion;
        next = nextNode;
    }

    /**
     * Get the data from the node
     * @return
     */
    public T getData() {
        return data;
    }

    /**
     * Set the data in the node
     * @param newData
     */
    public void setData(T newData) {
        data = newData;
    }

    /**
     * Get the next node
     * @return
     */
    public Node<T> getNextNode() {
        return next;
    }

    /**
     * Set the next node
     * @param nextNode
     */
    public void setNextNode(Node<T> nextNode) {
        next = nextNode;
    }
}
```

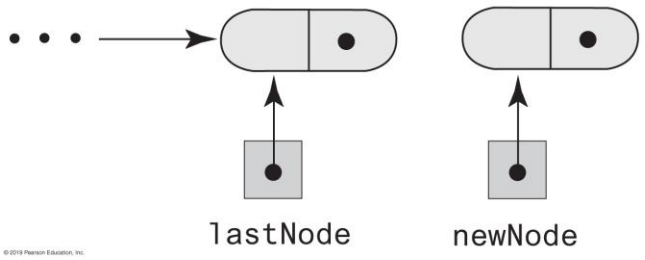
Linked Implementation of a Queue

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

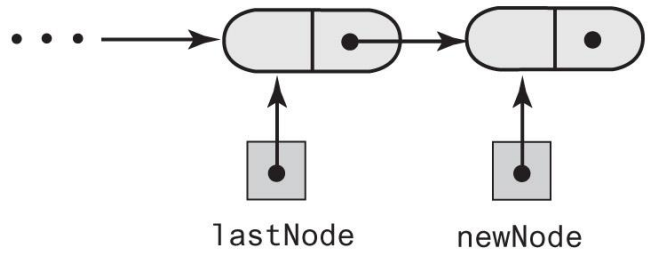


Adding a new node to the end of a nonempty chain that has a tail reference

(a) Before the addition



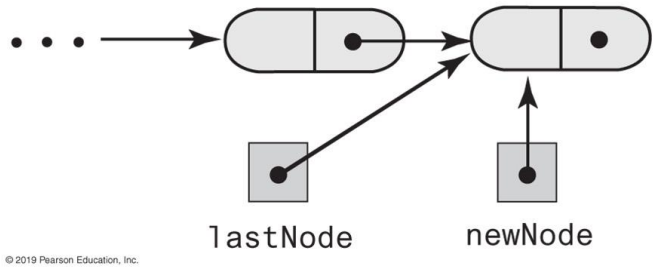
(b) During the addition



After executing

```
lastNode.setNextNode(newNode);
```

(c) After the addition



After executing

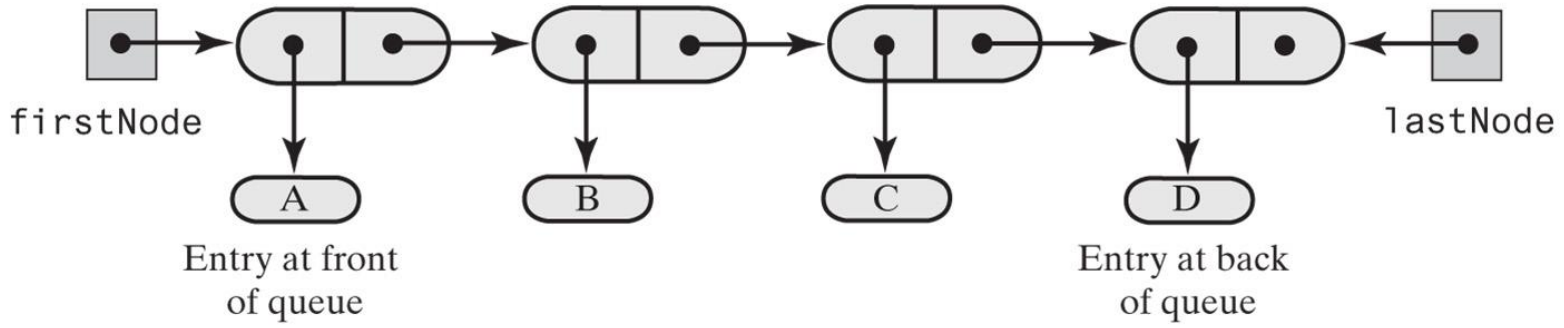
```
lastNode = newNode;
```

LinkedQueue – enqueue(), O(1)

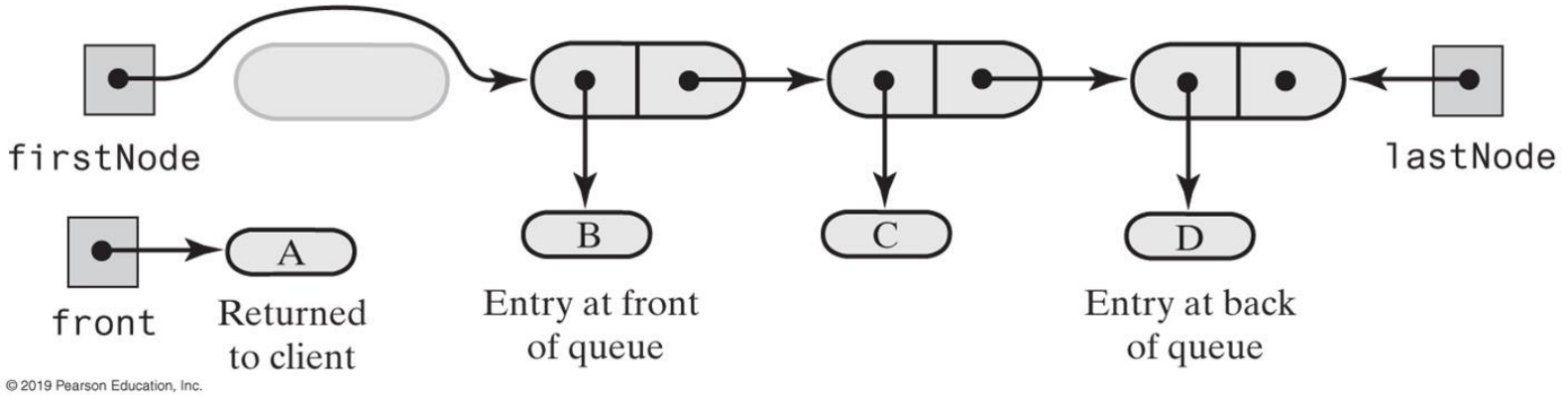
```
public final class CompletedLinkedQueue<T> implements QueueInterface<T>
{
   private Node<T> firstNode; // References node at front of queue
   private Node<T> lastNode;  // References node at back of queue
   int numberOfEntries;

   public CompletedLinkedQueue()
   {
      firstNode = null;
      lastNode = null;
      numberOfEntries = 0;
   }

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

              // make the new node the next on the chain from
              // the last node

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

              // then set the last node to the new node

              lastNode = newNode;

              numberOfEntries++;
       }
```

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

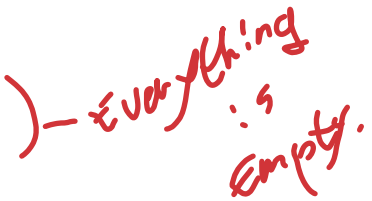


(b) After removing the entry at the queue's front



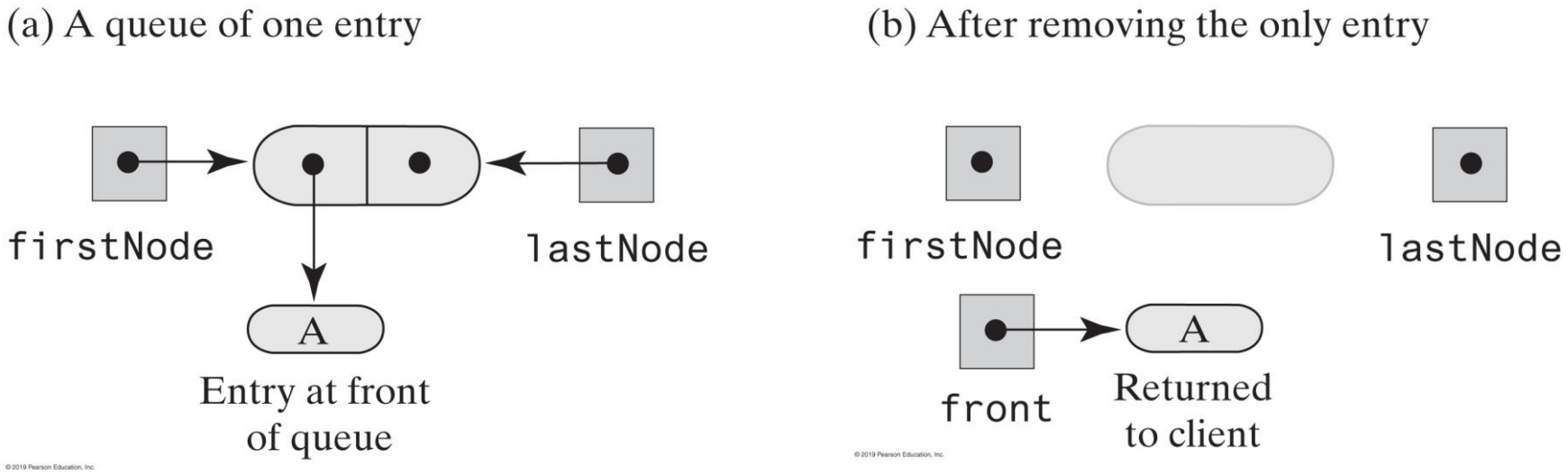
getFront(), dequeue()

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

public T dequeue() {

      T front = getFront();

      // clear the data in the first node
      //  then skip around it, setting the first node
      //  to its next in the chain
      firstNode.setData(null);
      firstNode = firstNode.getNextNode();

      // chain is empty
      if (firstNode == null)
            lastNode = null;

      numberOfEntries--;
      return front;
}
```



Linked Implementation of a Queue

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



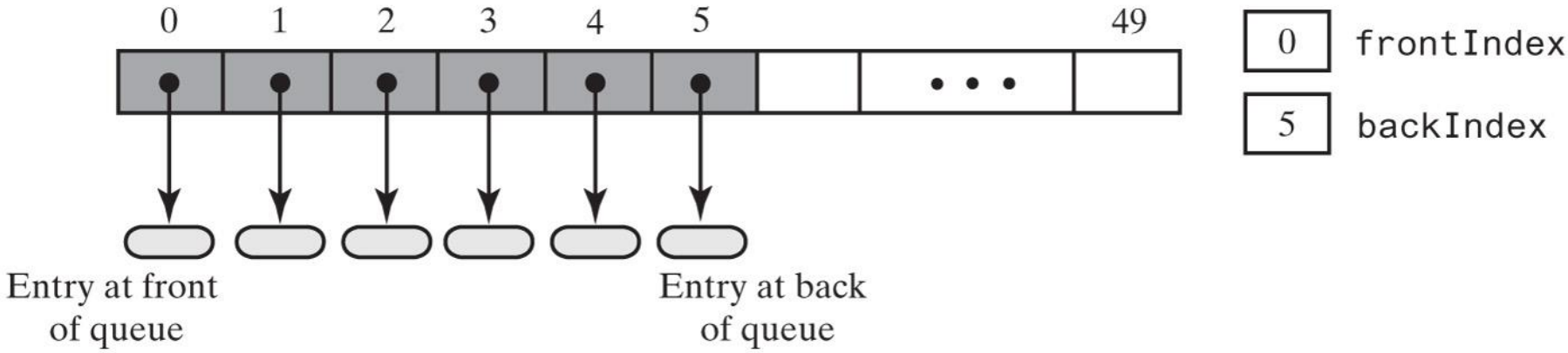
LinkedDequeue – other methods

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

public void clear() {
     firstNode = null;
     lastNode = null;
     numberOfEntries = 0;
}

public int size() {
     return numberOfEntries;
}

public T[] toArray() {
     // create a new array
     @SuppressWarnings("unchecked")
     T[] items = (T[]) new Object[size()];

     // walk along the chain, copying the data to the array each iteration
     int index = 0;
     for (Node<T> currentNode = firstNode; currentNode != null; currentNode = currentNode.getNextNode()) {
          items[index] = currentNode.getData();
          index++;
     }

     // ok to return an empty array
     return items;
}
```

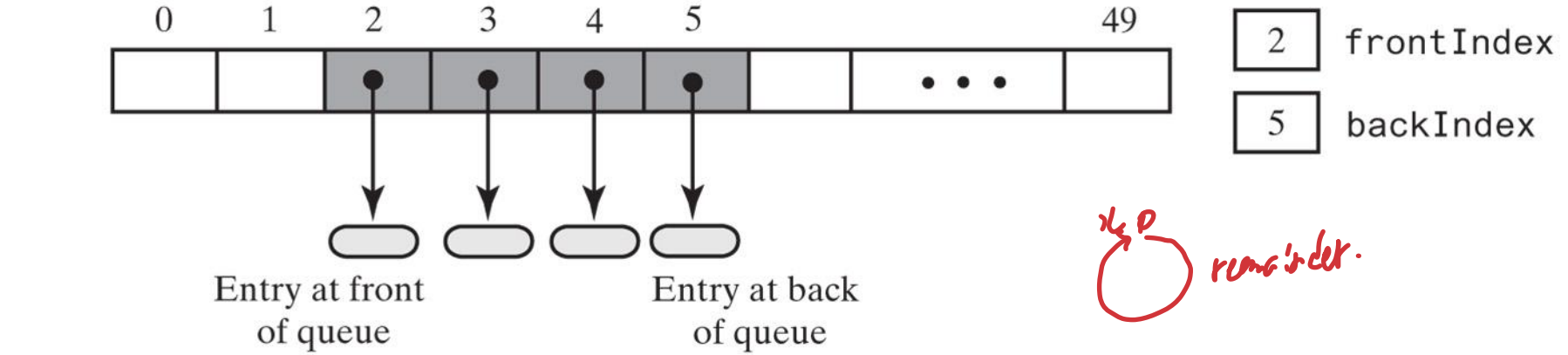
Array-Based Implementation of a Queue: Circular Array

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

(a) The queue initially



(b) After two removals of the front entry



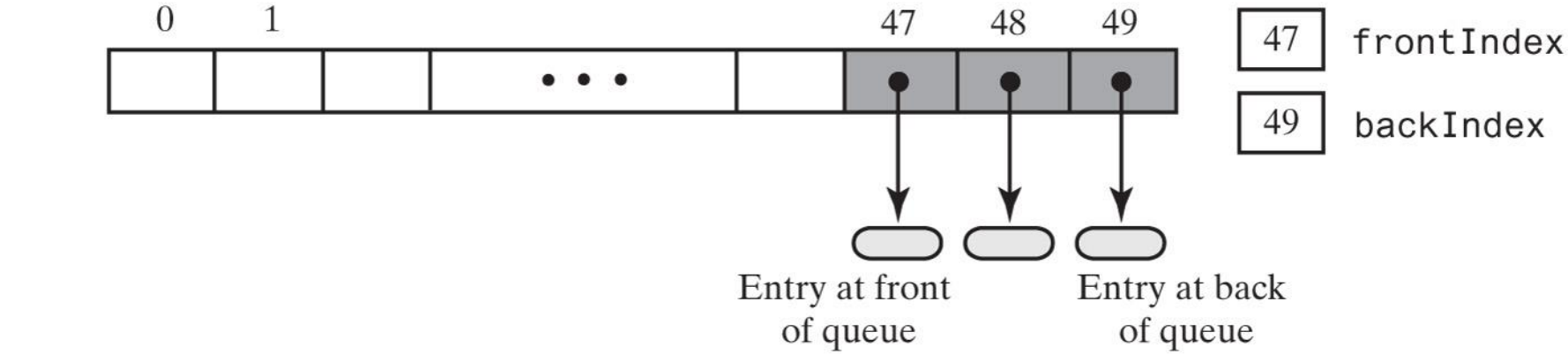
ArrayQueue – Circular Array implementation

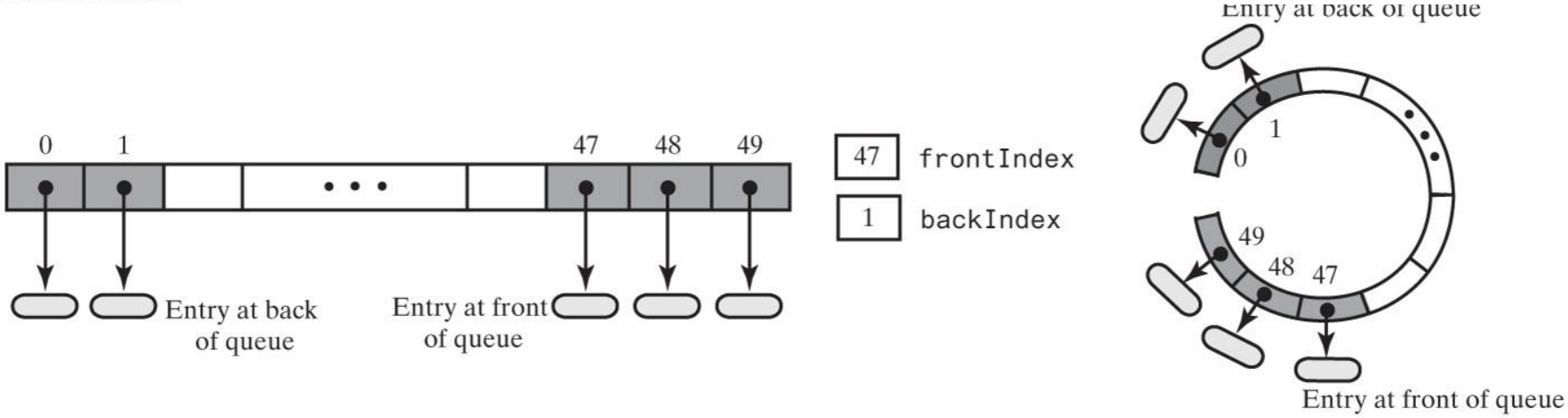
- Uses an internal array with a front and back index
- Array always has an unused slot for next possible addition

```
public final class CompletedArrayQueue<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 static final int DEFAULT_CAPACITY = 3;
      private static final int MAX_CAPACITY = 10000;

      public CompletedArrayQueue() {
            this(DEFAULT_CAPACITY);
      }

      public CompletedArrayQueue(int initialCapacity) {

            checkCapacity(initialCapacity);

            // The cast is safe because the new array contains null entries
            @SuppressWarnings("unchecked")

            // always leave space for the next entry

            T[] tempQueue = (T[]) new Object[initialCapacity + 1];
            queue = tempQueue;
            frontIndex = 0;
            backIndex = initialCapacity;

      }
```

Circular Array: adding 2 entries

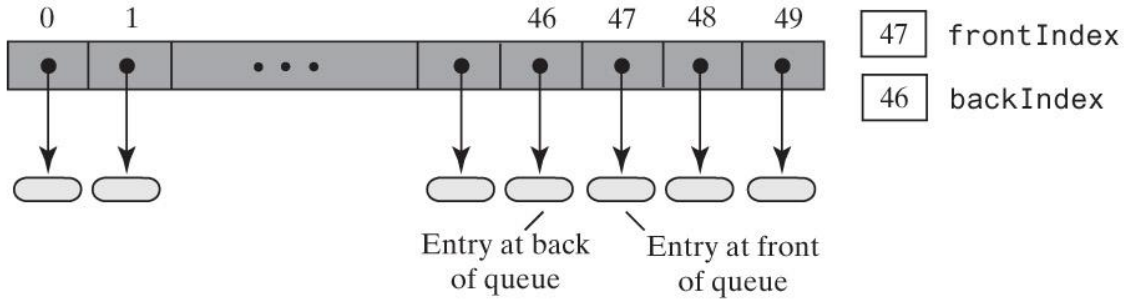
(c) After several more additions and removals

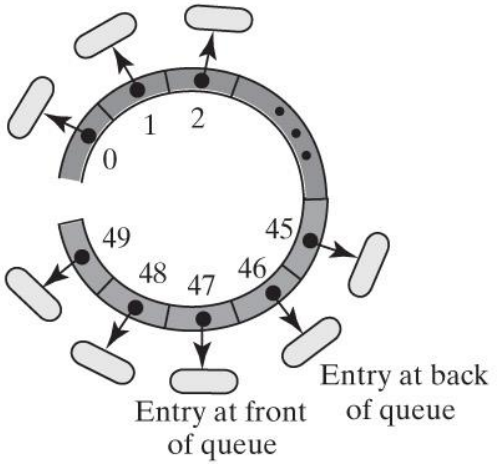




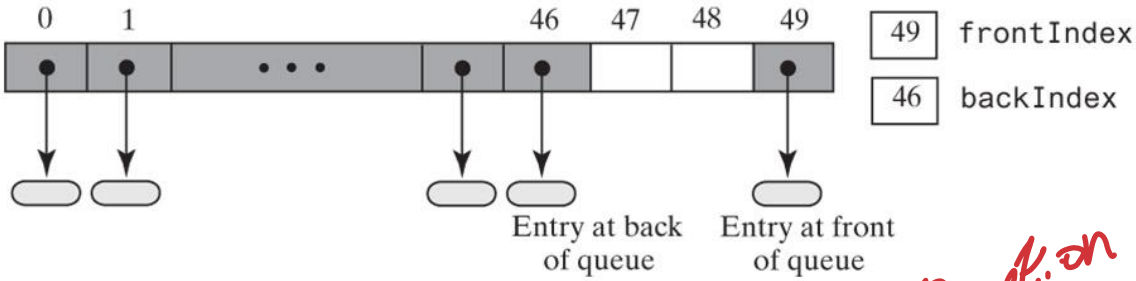
Circular Array: removing entries

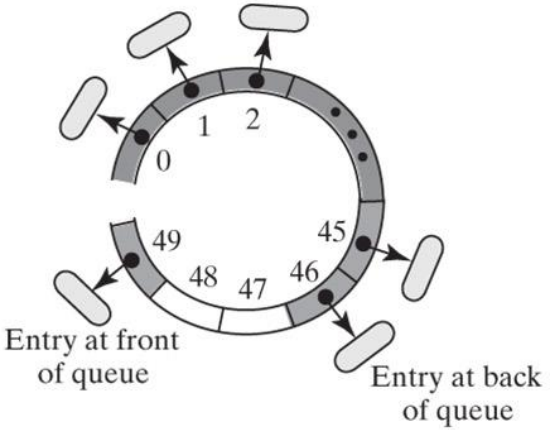
(a) After adding more entries to the queue in Figure 8-7 until it is full





(b) After removing two entries

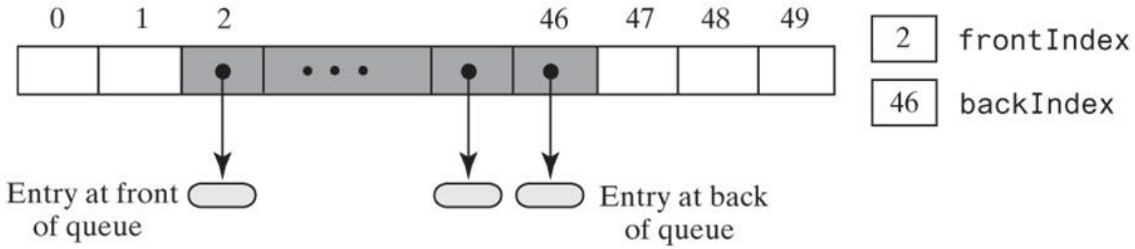


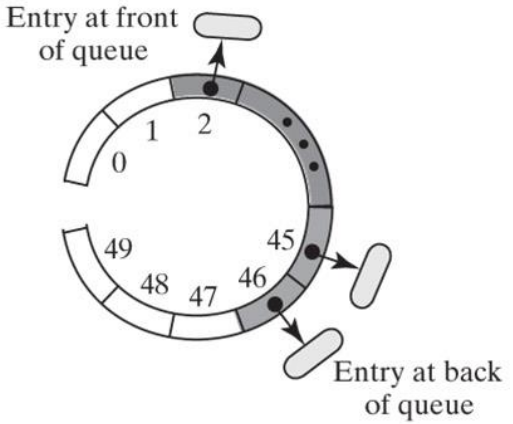


Quest. on here!

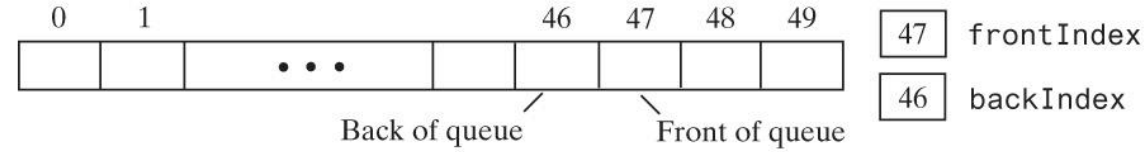
Circular Array: removing entries

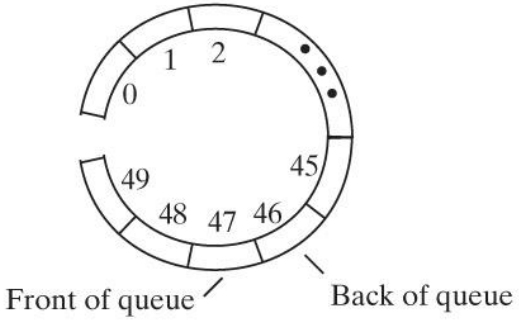
(c) After removing three more entries





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





ArrayQueue – getFront()

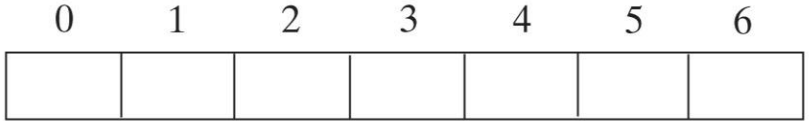
- Uses frontIndex

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

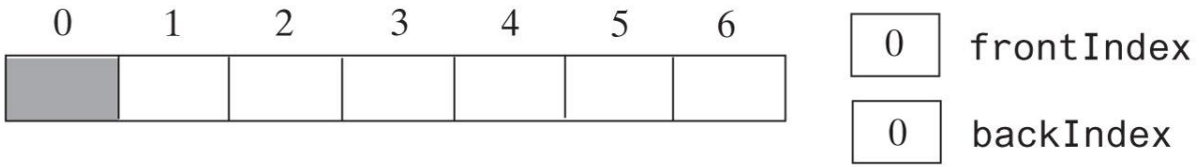
Circular Array (Part 1)

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

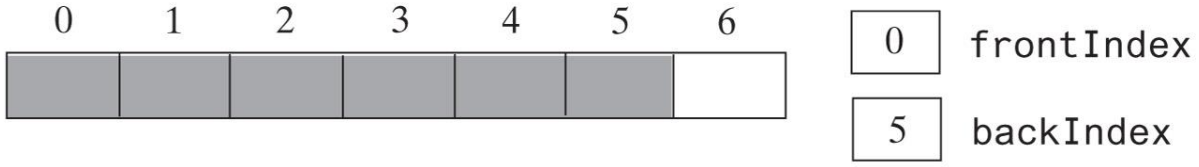
(a) Initially, the queue is empty



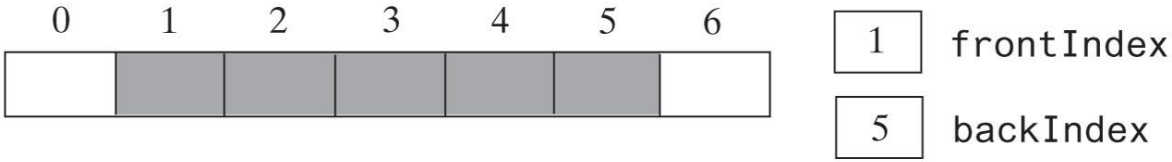
(b) After enquеuing one entry



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

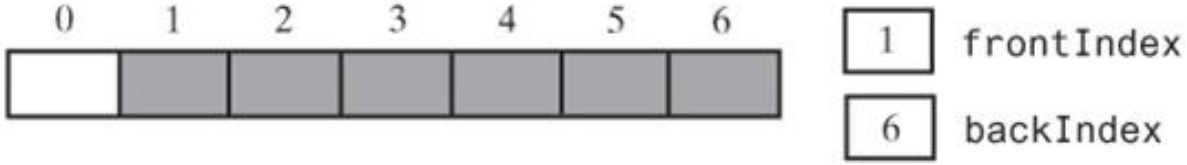


(d) After dequeuing an entry

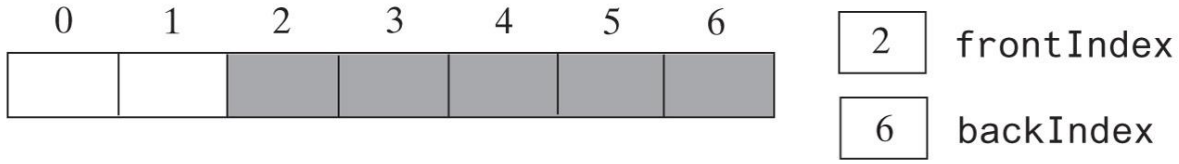


Circular Array (Part 2)

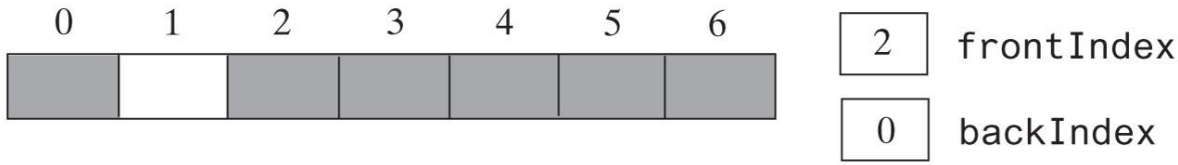
(e) After enqueuing an entry, the queue becomes full again



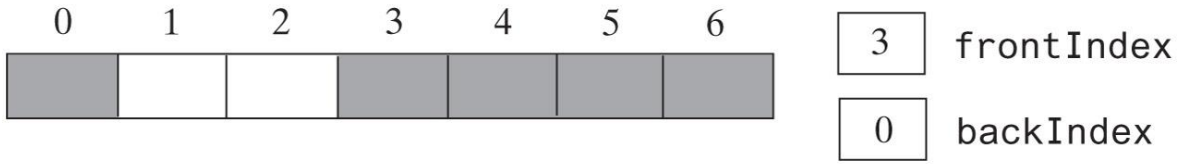
(f) After dequeuing an entry



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



(h) After dequeuing an entry



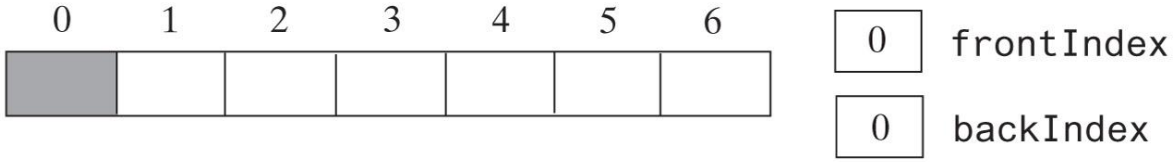
nextIndex() and enqueue()

- To move to the next index, what happens if we are at the end of the array?
  - Use modulo arithmetic to wrap around to the beginning
  - nextIndex() uses this so we can safely proceed to the next index
- enqueue() uses nextIndex() to find the next slot after the current back index

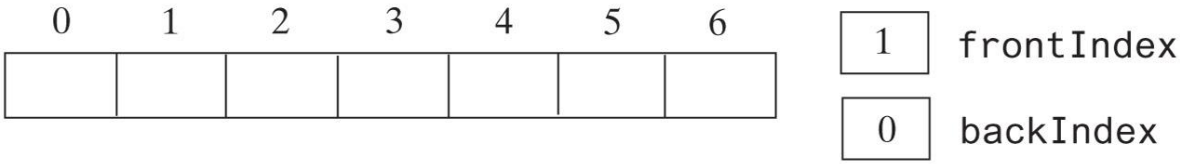
```
/**
 * Since this is a circular queue, incrementing the index
 * will always be modulo queue length to ensure wrap around.
 *
 * Let's encode this here to make things more readable
 * @param index current index in the circular queue
 * @return the next index in the circular queue
 */
private int nextIndex(int index) {
        return (index + 1) % queue.length;
}

public void enqueue(T newEntry) {
        ensureCapacity();

        // increment back index, then set new entry to its slot
        backIndex = nextIndex(backIndex);
        queue[backIndex] = newEntry;
}
```

Circular Array (Part 3)

(i) After dequeuing all but one entry



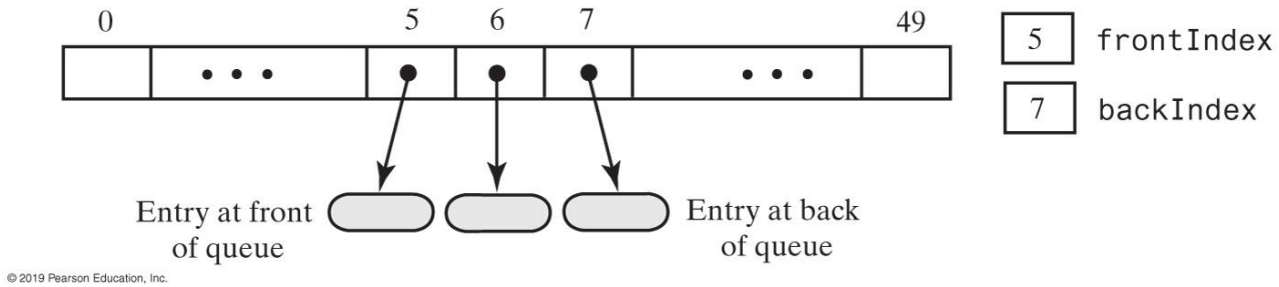
(j) After dequeuing the remaining entry, the queue is now empty



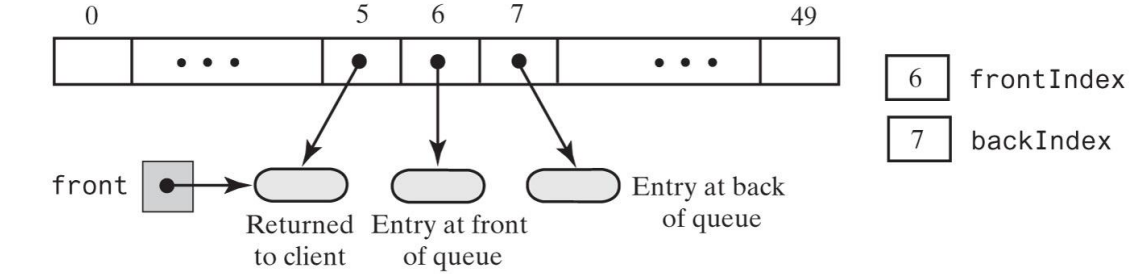
Circular Array with One Unused Location

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

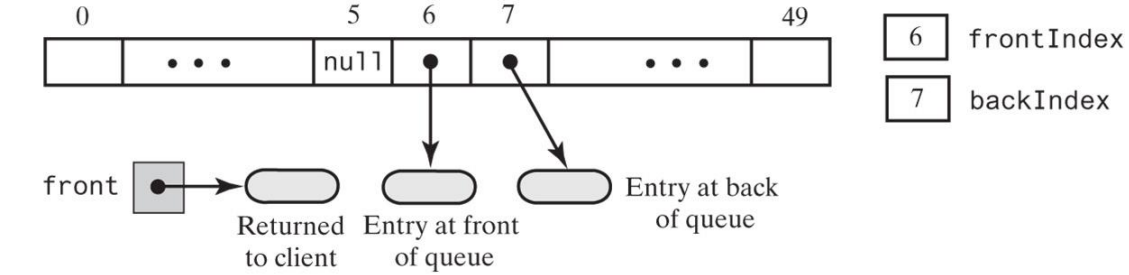
(a) Initially



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



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



dequeue()

- Get the front item
- Set front index to the next one.
  - It may wrap around to the beginning

```
public T dequeue() {

    if (isEmpty())
        throw new EmptyQueueException();

    // get the front of the queue
    T front = queue[frontIndex];

    // set the current front to null, clearing it
    queue[frontIndex] = null;

    // increment the front
    frontIndex = nextIndex(frontIndex);
    return front;

}
```

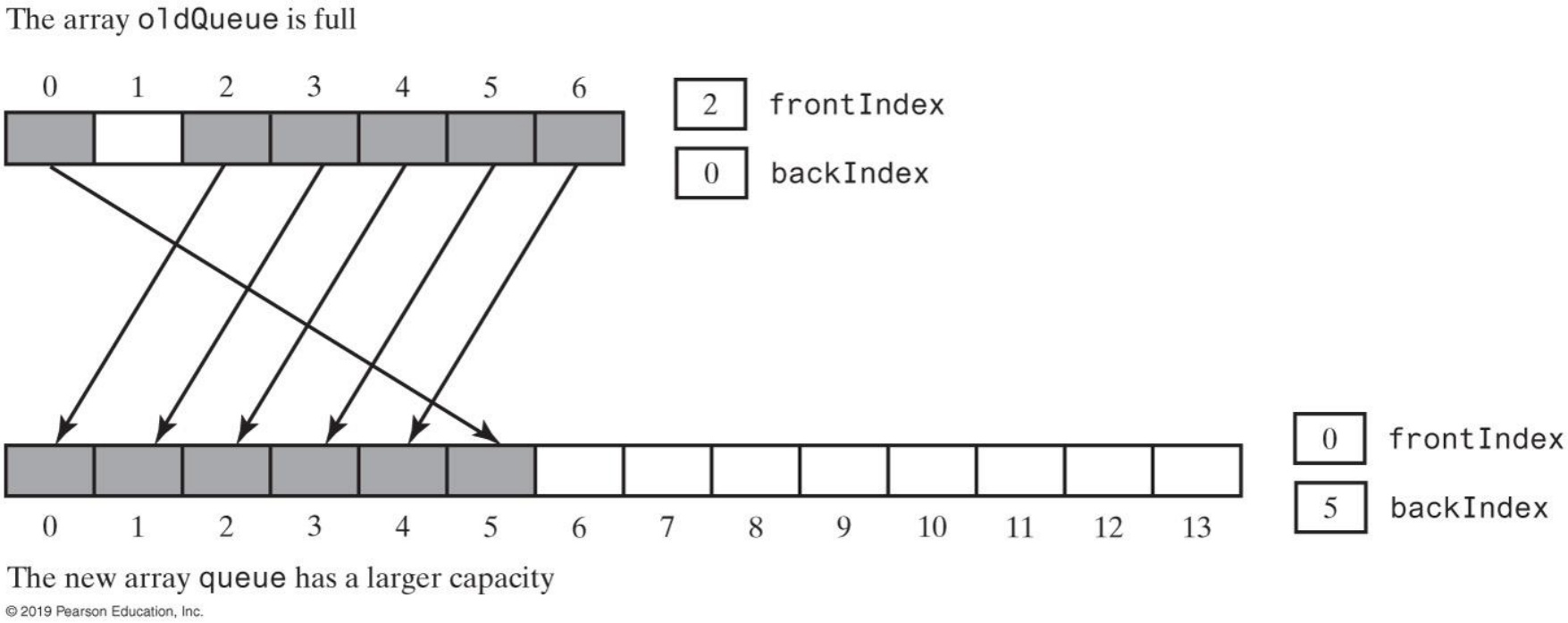
Other methods

- size() must take into consideration wraparound.
  - Better to compute directly rather than keep a count
- clear() needs to iterate through the array with items
  - although it could be done via simply allocating a new array

```
    public boolean isEmpty() {
        // if we increment backIndex and get frontIndex, we
        // have wrapped around, and so the queue is empty
        return frontIndex == nextIndex(backIndex);
    }

    public void clear() {
        // sets all allocated entries to null
        // this could also be done using dequeue()

        if (!isEmpty()) {
            // iterate through the queue
            for (int index = frontIndex; index != nextIndex(backIndex); index = nextIndex(index)) {
                queue[index] = null;
            }
        }

        // reset front and back indices
        frontIndex = 0;
        backIndex = queue.length - 1;
    } // end clear

    @Override
    public int size() {

        // increment the back and subtract the front
        // If they are the same, size is zero
        // if difference is positive, this is the exact size
        //    as back is behind front
        // if it is negative, back is before front, so adjust
        //    by array size
        int size = nextIndex(backIndex) - frontIndex;
        if(size < 0)
            size = size + queue.length;

        return size;
    }
```

Circular Array with One Unused Location

- Doubling the size of an array-based queue



copyToArray()

- used by toArray() and when resizing array using ensureCapacity()
- Iterate through the items in the array, copying to a new array
  - Note use of nextIndex()

```
public T[] toArray() {
    @SuppressWarnings("unchecked")
    T[] items = (T[]) new Object[size()];

    // copy the queue contents to the output array
    copyToArray(items);
    return items;
}

/**
 * Copy the items in the queue over to an array
 * Array must be equal or larger in size than queue
 * @param array destination array for copy.
 */
private void copyToArray(T[] array) {
    // should be assertions
    if(array == null)
        return;

    if(array.length < size())
        return;

    // iterate through the queue and copy items over to the array
    int tempIndex = 0;
    for(int currentIndex = frontIndex; currentIndex != nextIndex(backIndex); currentIndex = nextIndex(currentIndex)) {
        array[tempIndex] = queue[currentIndex];
        tempIndex++;
    }
}
```

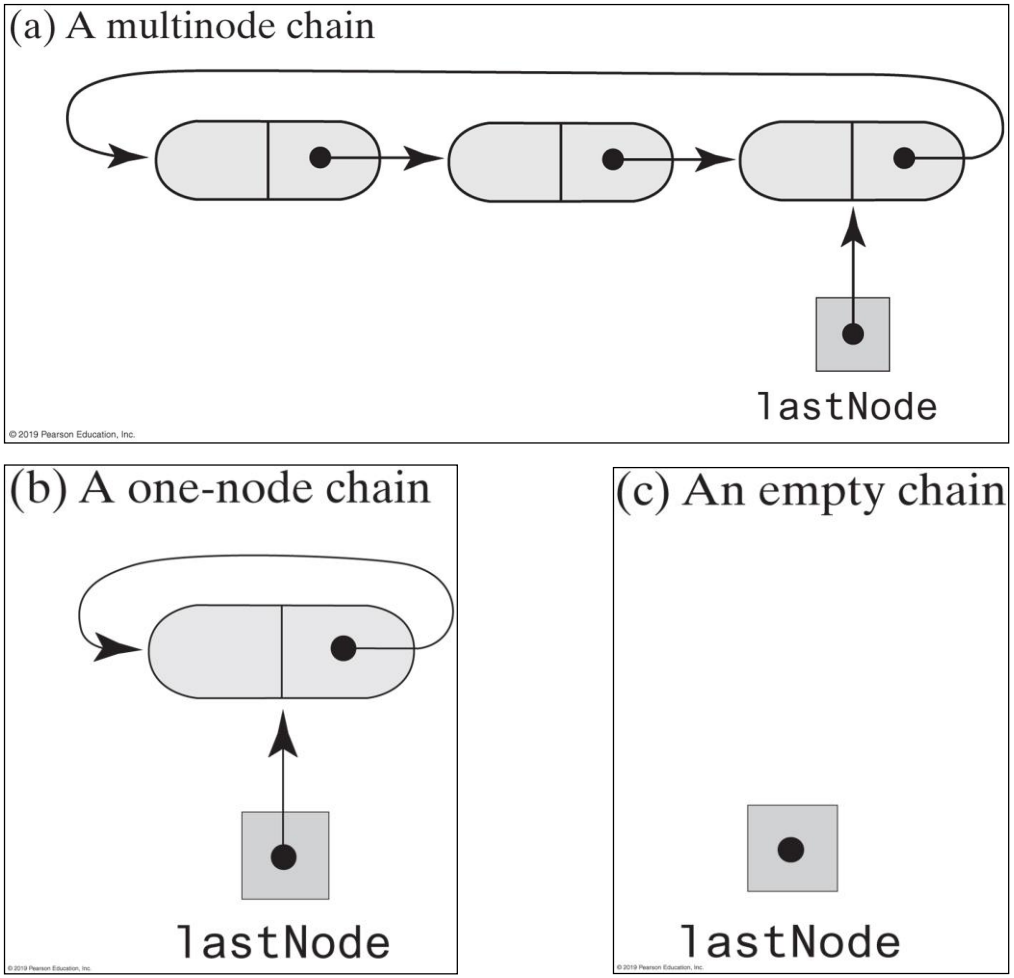
ensureCapacity()

- Create a larger array, and copy existing
- Reset front and back index

```
/**
 * Throws an exception if the client requests a capacity that is too large.
 * @param capacity
 */
private void checkCapacity(int capacity) {
     if (capacity > MAX_CAPACITY)
          throw new IllegalStateException(
                    "Attempt to create a queue " + "whose capacity exceeds " + "allowed maximum.");
}

/**
 * Doubles the size of the array queue if it is full.
 */
private void ensureCapacity() {

     // first check to see if array is full which
     //   means we need an additional slot besides the one
     //   we need to fill (hence backIndex + 1)

     if (frontIndex == nextIndex(backIndex + 1))
     {
          // double size of array
          int oldSize = queue.length;
          int newSize = 2 * oldSize;
          checkCapacity(newSize);

          // create the larger queue
          // The cast is safe because the new array contains null entries
          @SuppressWarnings("unchecked")
          T[] tempQueue = (T[]) new Object[newSize];

          // copy the queue contents to the the new array.

          copyToArray(tempQueue);
          queue = tempQueue; // set the queue to the new array

          // reset front and back indices. front starts at zero again
          frontIndex = 0;
          // we wanted two slots back from end, so reset this
          backIndex = oldSize - 2;

     }
}
```

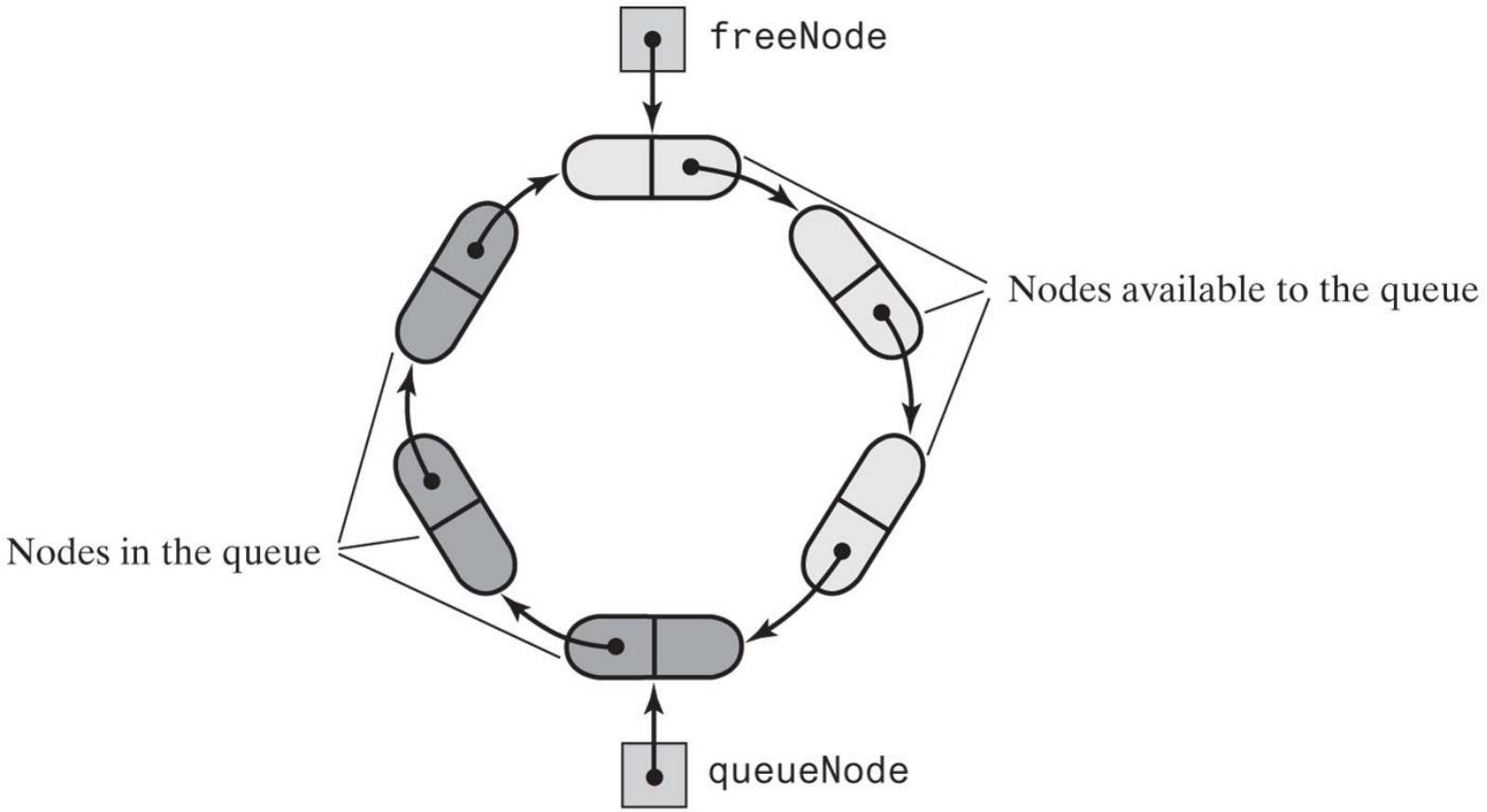
Circular Linked Implementations of a Queue

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



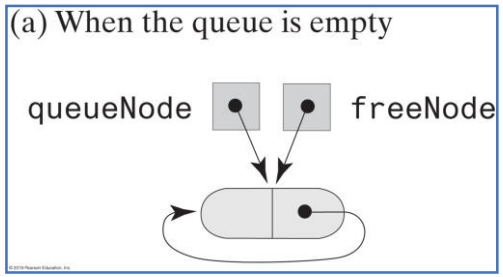
Two-Part Circular Linked Chain

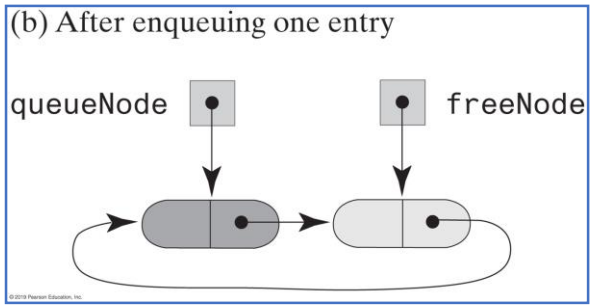
- A two-part circular linked chain that represents both a queue and the nodes available to the queue

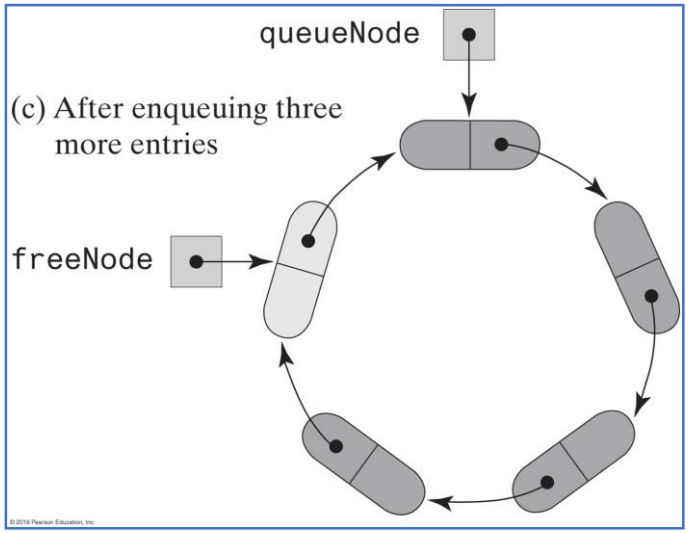


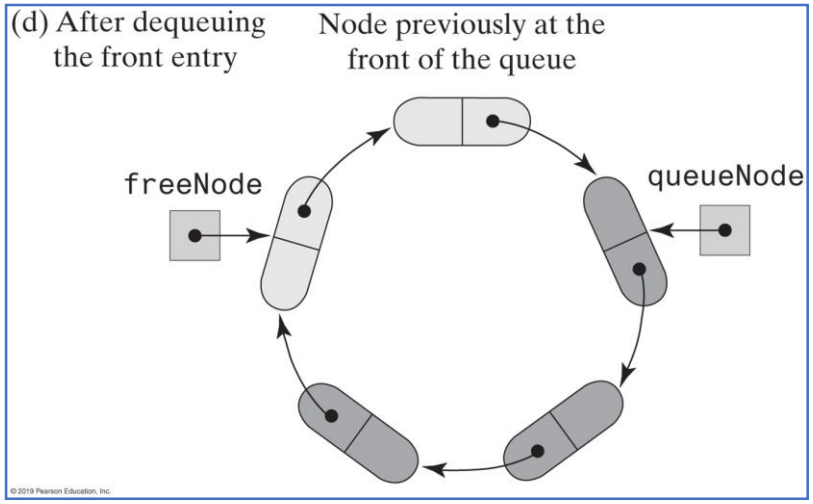
Two-Part Circular Linked Chain

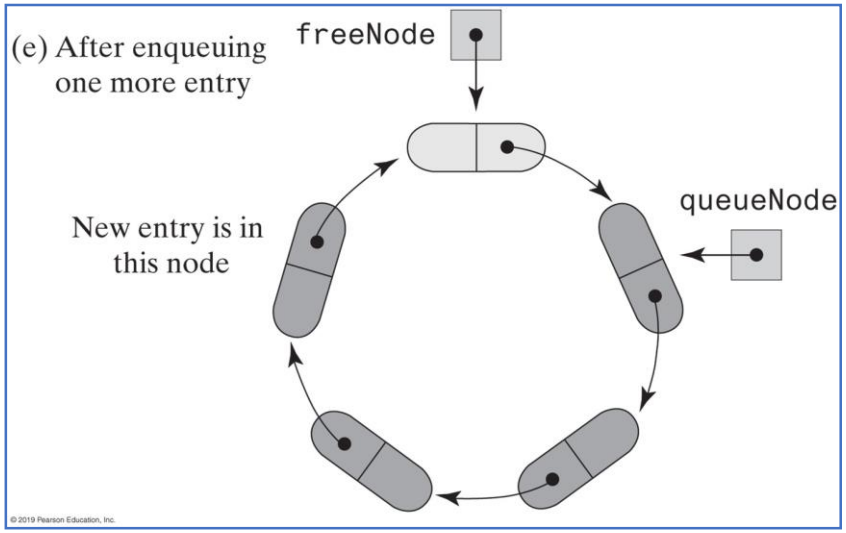
Various states of a two-part circular linked chain that represents a queue









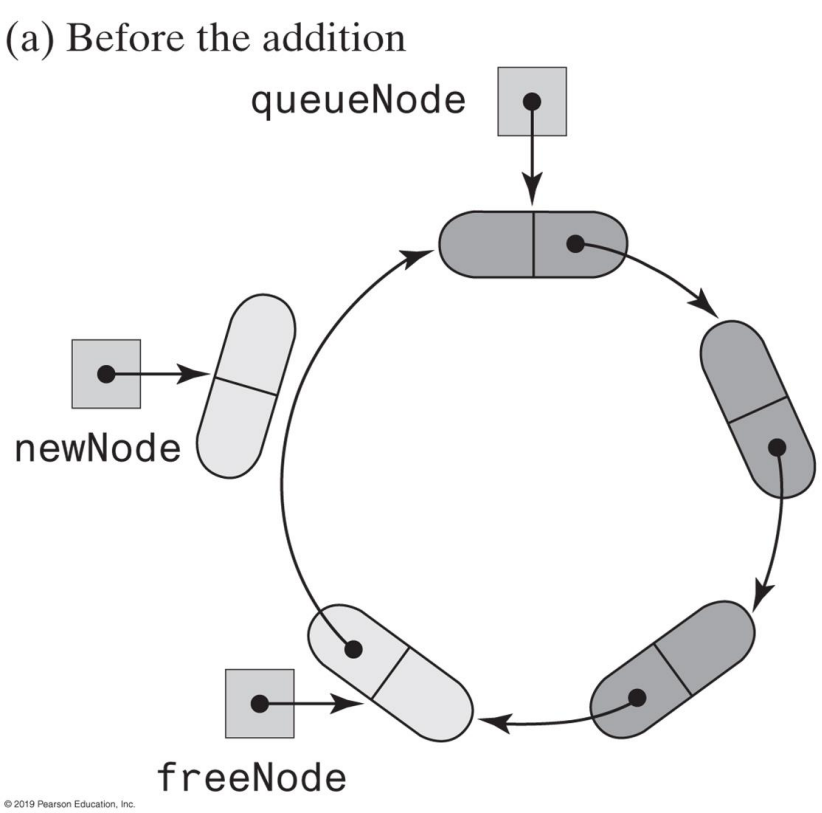


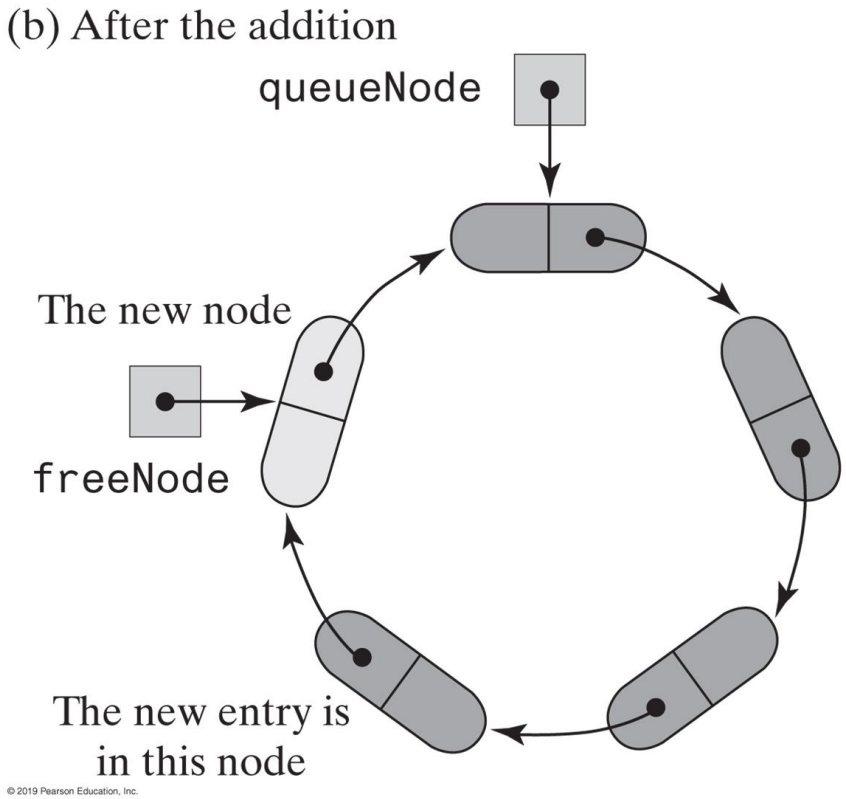
Two Part Circular Linked Queue

- Use of public class Node
- Need to keep track of the front node and the start of the free node chain.
- First node is set to null and links to itself.

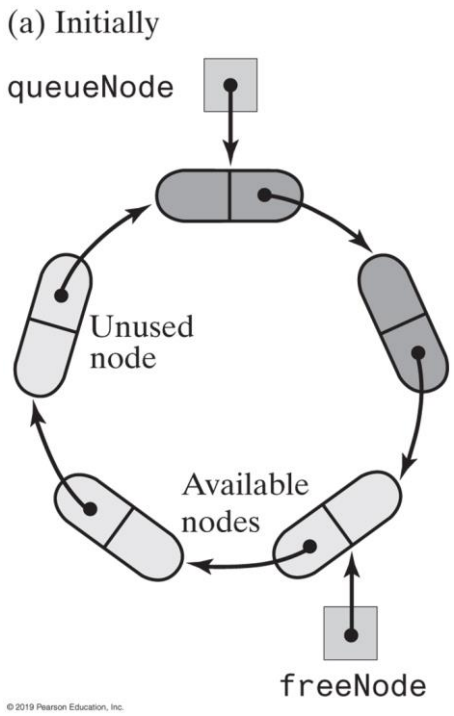
```
public final class CompletedTwoPartCircularLinkedQueue<T> implements QueueInterface<T> {
	private Node<T> queueNode; // References first node in queue
	private Node<T> freeNode; // References node after back of queue, chain of
unallocated nodes
	int numberOfEntries;

	public CompletedTwoPartCircularLinkedQueue() {
		freeNode = new Node<>(null, null);
		freeNode.setNextNode(freeNode);   // links to itself
		queueNode = freeNode;  // queueNode will be at the top of the chain
		numberOfEntries = 0;
	}
```

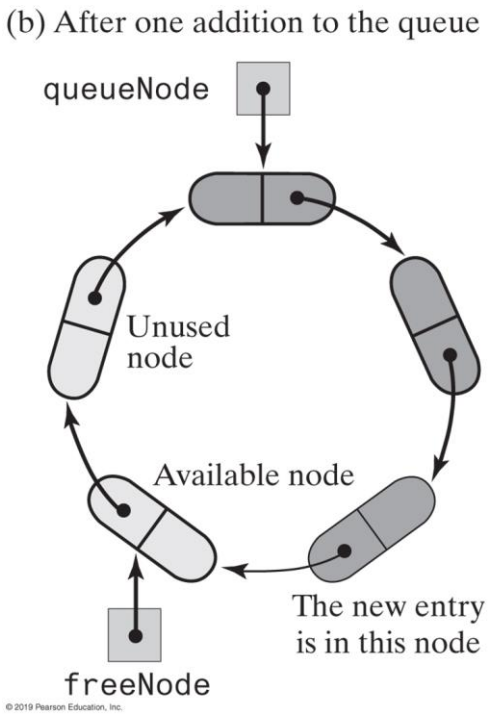
Two-part circular chain requires a new node for an addition to a queue

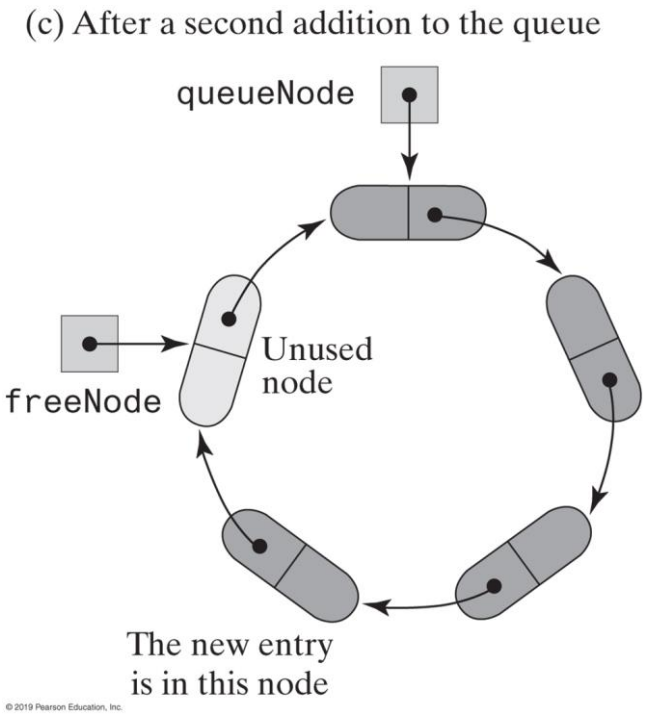




Two-part circular linked chain with nodes available for addition to a queue







enqueue()

- Basically allocated data in freeNode, then create a new freeNode.

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

     if (isNewNodeNeeded()) {
          // Allocate a new node (null) and insert it after the node that
          // freeNode references

          Node<T> newNode = new Node<>(null, freeNode.getNextNode());
          freeNode.setNextNode(newNode);

     }

     // we have a null as next node, so set free node to it.
     freeNode = freeNode.getNextNode();
     numberOfEntries++;
}

/**
 * In a two part circular chain, check if the node at the top of the queue is the
 * same as the free node's next in the chain.
 * @return
 */
private boolean isNewNodeNeeded() {
     return queueNode == freeNode.getNextNode();
}
```

getFront(), dequeue()

- To remove a node, set data to null.
- Note that freeNode chain will still point to head of queue as its next node, as it is circular.

```
public T getFront() {

      if (isEmpty())
            throw new EmptyQueueException();
      else
            return queueNode.getData();
}

public T dequeue() {

      // get the front of the queue
      T front = getFront();

      // clear the data, and set the front to the next node
      queueNode.setData(null);
      queueNode = queueNode.getNextNode();

      numberOfEntries--;
      return front;
}
```

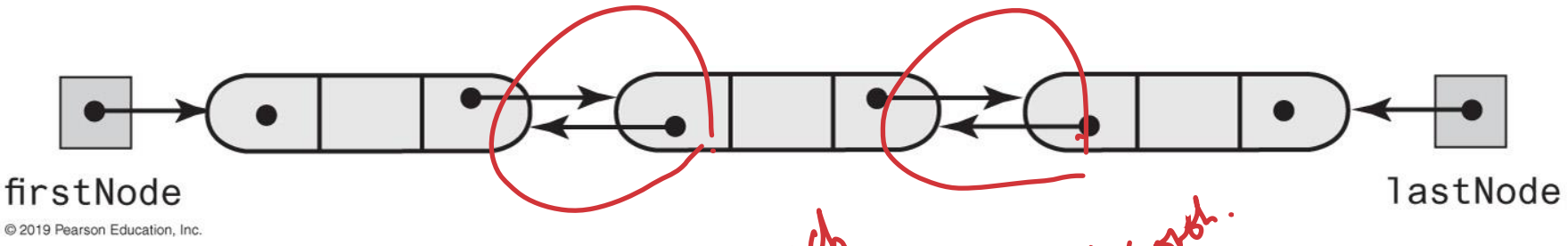
Other methods

- Notice how the chain is traversed in toArray().
  - Stop when freeNode is reached.

```
public boolean isEmpty() {
      return queueNode == freeNode;
}
public void clear() {
      while (!isEmpty())
            dequeue();
}

@Override
public int size() {
      return numberOfEntries;
}

@Override
public T[] toArray() {
      // create a new array
      @SuppressWarnings("unchecked")
      T[] items = (T[]) new Object[size()];

      // walk along the chain, copying the data to the array each iteration
      // note that the end of the queue is at freeNode
      int index = 0;
      for (Node<T> node = queueNode; node != freeNode; node = node.getNextNode()) {
            items[index] = node.getData();
            index++;
      }

      // ok to return an array with no items
      return items;
}
```

Doubly Linked Implementation of a Deque

- A doubly linked chain with head and tail references



Point each other, easy to go back and forth.

DoublyLinkedNode class

- subclass of Node
- adds previous link

```
public class DoublyLinkedNode<T> extends Node<T> {

	private DoublyLinkedNode<T> previous; // Link to previous node

	/**
	 * Create a node with next and previous set to null
	 * @param dataPortion
	 */
	public DoublyLinkedNode(T dataPortion) {
		super(dataPortion); // sets next to null as well
		previous = null;
	}

	/**
	 * Create a node with previous set to null.
	 * @param dataPortion
	 * @param nextNode
	 */
	public DoublyLinkedNode(T dataPortion, Node<T> nextNode) {
		super(dataPortion, nextNode);  // set next
		previous = null;
	}

	/**
	 * Create a node with previous and next set by args
	 * @param previousNode
	 * @param dataPortion
	 * @param nextNode
	 */
	public DoublyLinkedNode(DoublyLinkedNode<T> previousNode, T dataPortion, DoublyLinkedNode<T> nextNode) {
		super(dataPortion, nextNode); // set next
		previous = previousNode;
	}
```

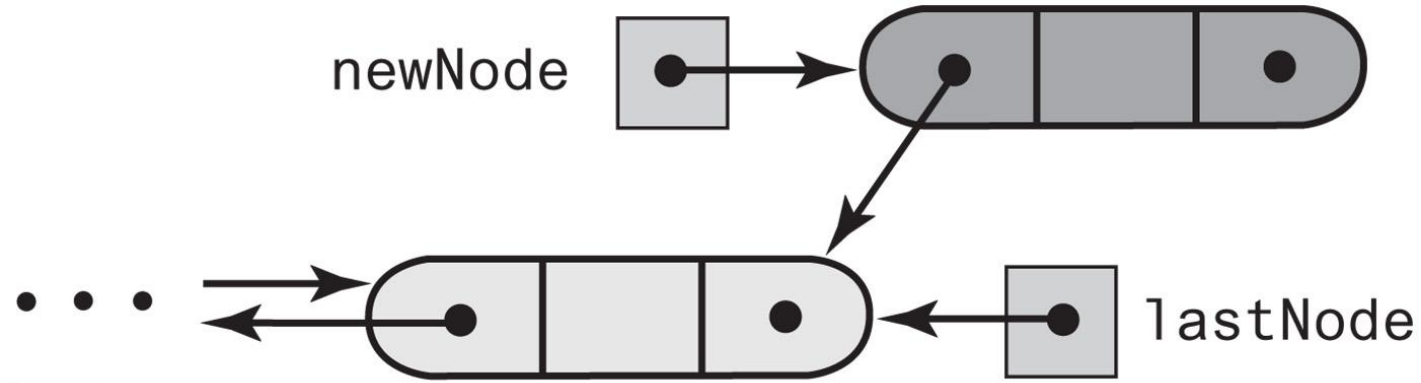
DoublyLinkedNode – set, get

- Note overridden getNextNode() so that it returns a DoublyLinkedNode, and caller does not need to cast

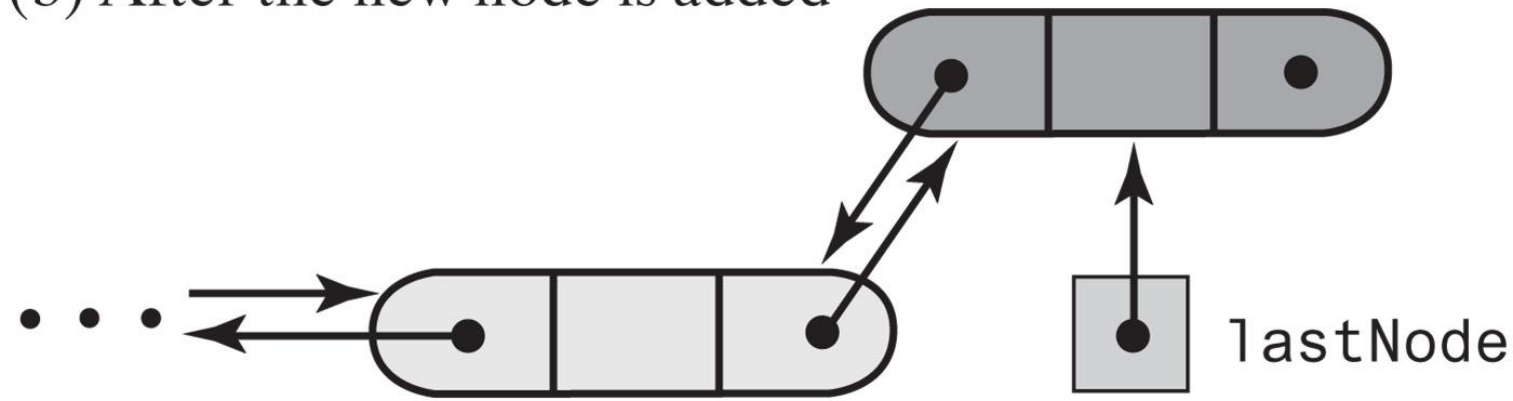
```
/**
 * Gets the previous node
 * @return
 */
public DoublyLinkedNode<T> getPreviousNode() {
     return previous;
}

/**
 * Sets the previous node
 * @param previousNode
 */
public void setPreviousNode(DoublyLinkedNode<T> previousNode) {
     previous = previousNode;
}

/**
 * Gets the next node
 *
 * @return nextNode
 */
public DoublyLinkedNode<T> getNextNode() {
     return (DoublyLinkedNode<T>) super.getNextNode();
}
```

Adding to the back of a nonempty deque

(a) After the new node is allocated

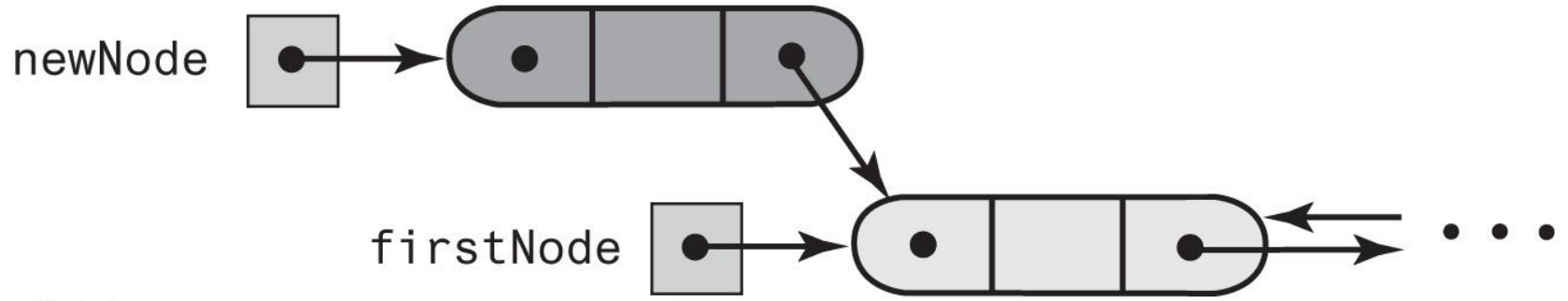


(b) After the new node is added

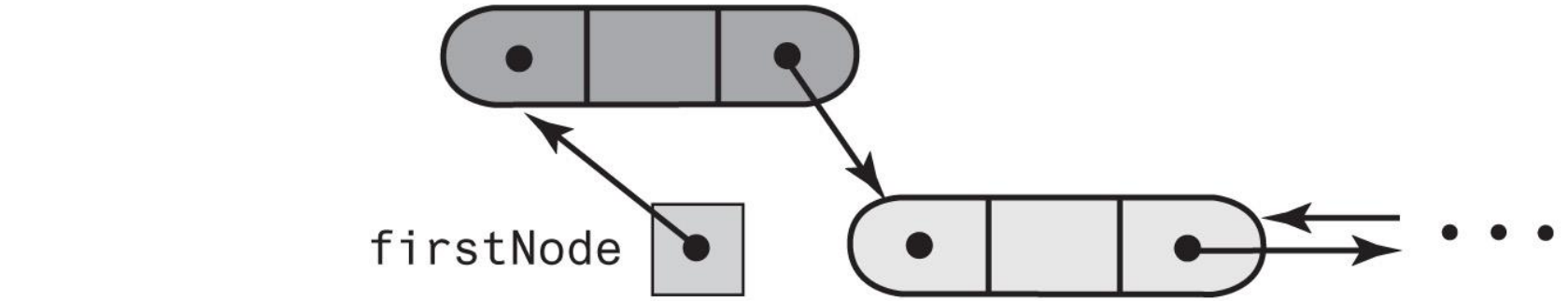


Adding to the front of a nonempty deque

(a) After the new node is allocated



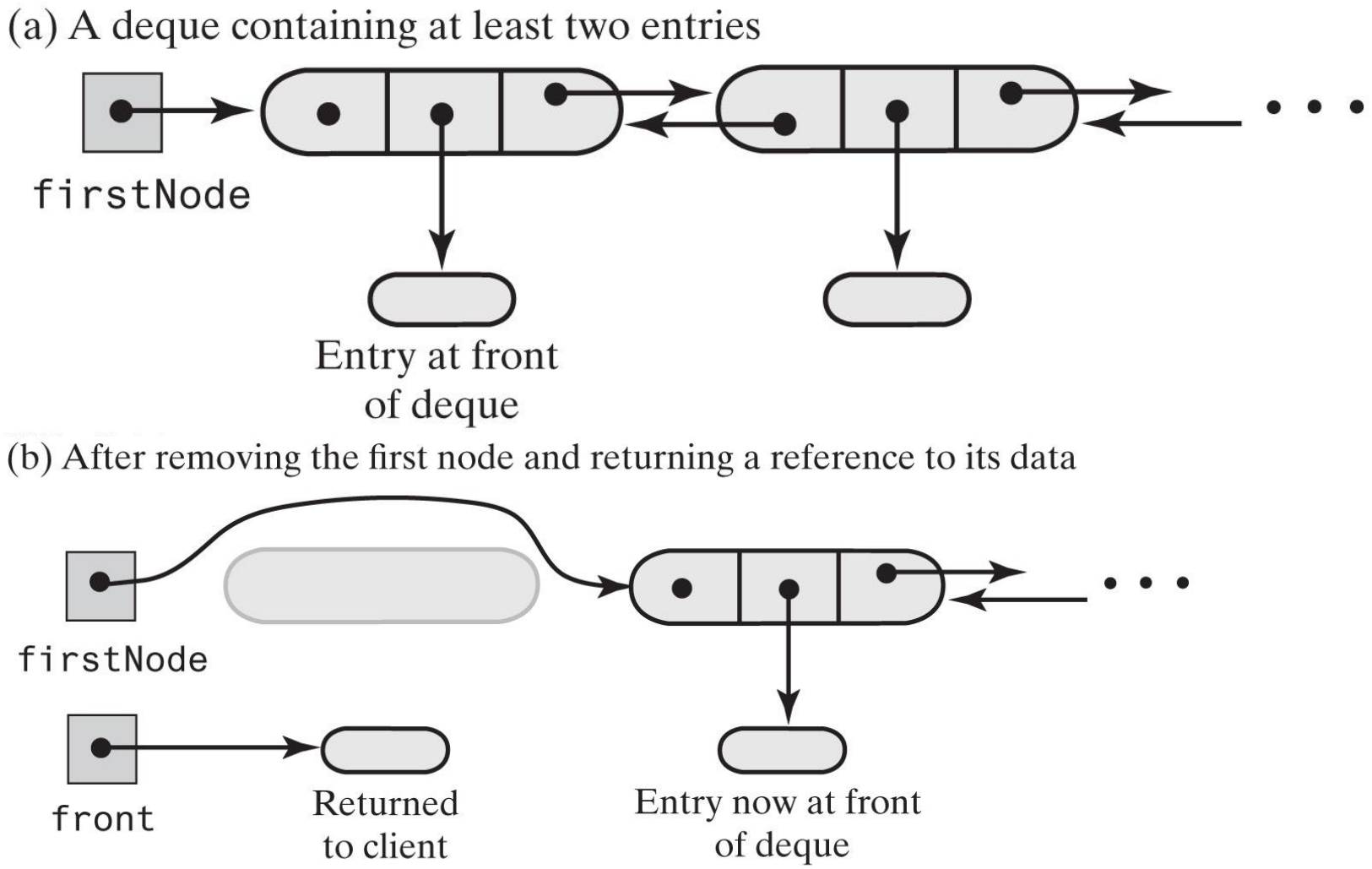
(b) After the new node is added to the front



LinkedDeque – add front/back

```
public void addToBack(T newEntry) {

      // set the previous node for the entry, and next is null

      DoublyLinkedNode<T> newNode = new DoublyLinkedNode<>(lastNode, newEntry, null);

      // if both are null, just need to set the first one
      // otherwise lastNode's next is set to null, so set it to this one.
      if (isEmpty())
            firstNode = newNode;
      else
            lastNode.setNextNode(newNode);

      // finally make the new node the last node, now that all the links are fixed up

      lastNode = newNode;
      numberOfEntries++;
}

public void addToFront(T newEntry) {

      // instead, previous node is null, and next is the start of the list

      DoublyLinkedNode<T> newNode = new DoublyLinkedNode<>(null, newEntry, firstNode);

      // if first and last are null, set last to the new node
      // otherwise set the previous node for for the start of the list to this one

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

      // links are all fixed up, make the first node this one

      firstNode = newNode;

      numberOfEntries++;
}
```

Removing the front of a deque containing at least two entries



LinkedDeque – get front/back

```
public T getBack() {
     if (isEmpty())
          throw new EmptyQueueException();
     else
          return lastNode.getData();
}

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

LinkedDeque – remove front/back

```
	public T removeFront() {

		T front = getFront();

		if(front == null)
			return null;

		// skip around node, resetting first node to
		//   the next on the chain
		// make sure the previous node is also set to null

		firstNode = firstNode.getNextNode();

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

		numberOfEntries--;

		return front;
	}

	public T removeBack() {
		T back = getBack();

		if(back == null)
			return null;

		// move the last node to the one before
		//  then set its next to null
		lastNode = lastNode.getPreviousNode();

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

		numberOfEntries--;

		return back;
	}
```

In class exercises

- Complete Queue Implementations
  - Implementations
    - ArrayQueue.java
    - LinkedQueue.java
    - TwoPartCircularLinkedQueue.java (optional, but need to understand how it works)
  - Test using
    - QueueTestDriver.java
    - StockLedgerDemo.java
    - WaitLineDemo.java
- Complete and test Deque Implementation
  - LinkedDeque.java
  - Test using LinkedDequeDemo.java
- These classes will be used again in the future, so save your work.