2.1.1

Describe the effect of each of the following operations on object myList as shown at the bottom of Figure 2.2. What is the value of `myList.size()` after each operation?

```
myList.add("Pokey");
myList.add("Campy");
int i = myList.indexOf("Happy");
myList.set(i, "Bouncy");
myList.remove(myList.size() - 2);
String temp = myList.get(1);
myList.set(1, temp.toUpperCase());
```

start size 6

```
myList.add("Pokey");
```

adds element to end of list, size now at 7

```
myList.add("Campy");
```

adds element to end of list, size now 8

```
int i = myList.indexOf("Happy");
```

returns integer 4 to i

```
myList.set(i, "Bouncy");
```

sets element i (4) to "Bouncy". Size = 8

```
myList.remove(myList.size() - 2);
```

Removes item 6 ("Pokey"). Size = 7

```
String temp = (String) myList.get(1);
```

Stores "Awful" in temp. Size = 7

```
myList.set(1, temp.toUpperCase());
```

Sets item 1 to "AWFUL", replacing "Awful".
Final size = 7

2.2.1

What does the following code fragment do?

```
ArrayList<Double> myList = new ArrayList<Double>();
myList.add(3.456);
myList.add(5.0);
double result = myList.get(0) + myList.get(1);
System.out.println("Result is " + result);
```

```
ArrayList<Double> myList = new ArrayList<Double>();
```

Initialized myList to an empty ArrayList<Double>

```
myList.add(3.456);
```

Appends the value 3.456 to the end of the list

```
myList.add(5.0);
```

Appends the value 5.0 to the end of the list

```
double result = myList.get(0) + myList.get(1);
```

Sets result to 3.456 + 5.0

```
System.out.println("Result is " + result);
```

Outputs `Result is 8.456`

2.3.1

Trace the execution of the following:

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = 3; i < anArray.length - 1; i++)
    anArray[i + 1] = anArray[i];
```

and the following:

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = anArray.length - 1; i > 3; i--)
    anArray[i] = anArray[i - 1];
```

What are the contents of `anArray` after the execution of each loop?

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	5	6	7

```
for (int i = 3; i < anArray.length - 1; i++)
```

i = 3

```
    anArray[i + 1] = anArray[i];
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	3	5	6	7

```
for (int i = 3; i < anArray.length - 1; i++)
```

i = 4

```
    anArray[i + 1] = anArray[i];
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	3	3	6	7

```
for (int i = 3; i < anArray.length - 1; i++)
```

i = 5

```
    anArray[i + 1] = anArray[i];
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	3	3	3	7

```
for (int i = 3; i < anArray.length - 1; i++)
```

i = 6

```
    anArray[i + 1] = anArray[i];
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	3	3	3	3

```
for (int i = 3; i < anArray.length - 1; i++)
```

i = 6

Loop exits

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	5	6	7

```
for (int i = anArray.length-1; i > 3; i--)
```

i = 7

```
    anArray[i] = anArray[i-1];
```

anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	5	6	6

```
for (int i = anArray.length-1; i > 3; i--)
```
i = 6
```
    anArray[i] = anArray[i-1];
```
anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	5	5	6

```
for (int i = anArray.length-1; i > 3; i--)
```
i = 5
```
    anArray[i] = anArray[i-1];
```
anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	4	5	6

```
for (int i = anArray.length-1; i > 3; i--)
```
i = 4
```
    anArray[i] = anArray[i-1];
```
anArray

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	3	4	5	6

```
for (int i = anArray.length-1; i > 3; i--)
```
i = 3

Loop exits

2.4.1

Determine how many times the output statement is displayed in each of the following fragments. Indicate whether the algorithm is **O**(n) or **O**(n^2).

a.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        System.out.println(i + " " + j);
```

b.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < 2; j++)
        System.out.println(i + " " + j);
```

c.
```
for (int i = 0; i < n; i++)
    for (int j = n - 1; j >= i; j--)
        System.out.println(i + " " + j);
```

d.
```
for (int i = 1; i < n; i++)
    for (int j = 0; j < i; j++)
        if (j % i == 0)
            System.out.println(i + " " + j);
```

a. n^2 $O(n^2)$

b. $2n$ $O(n)$

c. $n(n-1)/2$ $O(n^2)$

d. $n-1$ $O(n)$

2.4.3

How does the performance grow as n goes from 2000 to 4000 for the following? Answer the same question as n goes from 4000 to 8000. Provide tables similar to Table 2.4.

a. **O**(log n)
b. **O**(n)
c. **O**(n log n)
d. **O**(n^2)

e. O(*n*3)

O(f(n))	f(2000)	f(4000)	f(8000)/f(4000)
O(log n)	10.97	11.97	1.09
O(n)	2000	4000	2
O(n log n)	21932	47863	2.18
O(n^2)	4000000	16000000	4
O(n3)	8×10^9	6.4×10^{10}	8

O(f(n))	f(4000)	f(8000)	f(100)/f(50)
O(log n)	11.97	12.97	1.08
O(n)	4000	8000	2
O(n log n)	47863	103726	2.17
O(n^2)	16000000	64000000	4
O(n3)	6.4×10^{10}	5.12×10^{11}	8

2.5.1

What is the big-O for the single-linked list `get` operation?

The list must be searched for the index, thus O(n)

2.5.3

What is the big-O for each `add` method?

Since a reference to the tail of the list is available, the add method is constant or O(1).

2.5.5

For the single-linked list in Figure 2.16, data field `head` (type `Node`) references the first node. Explain the effect of each statement in the following fragments.

a.
```
head = new Node<String>("Shakira", head.next);
```

b.
```
Node<String> nodeRef = head.next;
nodeRef.next = nodeRef.next.next;
```

c.
```
Node<String> nodeRef = head;
while (nodeRef.next != null)
    nodeRef = nodeRef.next;
nodeRef.next = new Node<String>("Tamika");
```

d.
```
Node<String> nodeRef = head;
while (nodeRef != null && !nodeRef.data.equals("Harry"))
    nodeRef = nodeRef.next;
if (nodeRef != null) {
    nodeRef.data = "Sally";
    nodeRef.next = new Node<String>("Harry", nodeRef.next.next);
}
```

a.
```
head = new Node<String>("Shakira", head.next);
```
Inserts a node containing "Shakira" as the first item in the list.

b.
```
Node<String> nodeRef = head.next;
nodeRef.next = nodeRef.next.next;
```
Removes the node "Harry" from the list.

c.
```
Node<String> nodeRef = head;
while (nodeRef.next != null)
    nodeRef = nodeRef.next;
nodeRef.next = new Node<String>("Tamika");
```
Appends a new node "Tamika" to the end of the list

d.
```
Node<String> nodeRef = head;
while (nodeRef != null && !nodeRef.data.equals("Harry"))
    nodeRef = nodeRef.next;
if (nodeRef != null) {
    nodeRef.data = "Sally";
    nodeRef.next = new Node<String>("Harry", nodeRef.next.next);
}
```
Changes the node "Harry" to "Sally" and then inserts a new node "Harry" following the node "Sally"

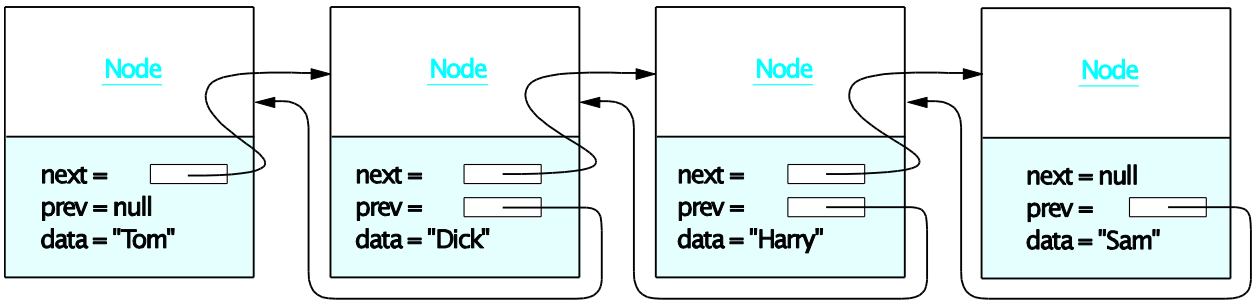
2.6.1

Answer the following questions about lists.
a. Each node in a single-linked list, has a reference to _____ and _____.
b. In a double-linked list, each node has a reference to _____, _____, and _____.
c. To remove an item from a single-linked list, you need a reference to _____.
d. To remove an item from a double-linked list, you need a reference to _____.

a. Each node in a single-linked list, has a reference to the data and the next node.
b. In a double-linked list, each node has a reference to the data, the next node, and the previous node.
c. To remove an item from a single-linked list, you need a reference to the previous node.
d. To remove an item from a double-linked list, you need a reference to the node.

2.7.1

The method `indexOf`, part of the `List` interface, returns the index of the first occurrence of an object in a `List`. What does the following code fragment do?
```
int indexOfSam = myList.indexOf("Sam");
ListIterator<String> iteratorToSam = listIterator(indexOfSam);
iteratorToSam.previous();
iteratorToSam.remove();
```
where the internal nodes of `myList` (type `LinkedList<String>`) are shown in the figure below:



It removes the node "Harry" from the list.

2.7.3

In Question 1, what if we omit the statement
`iteratorToSam.previous();`

An IllegalStateException is thrown by the statement
`iteratorToSam.remove();`

2.8.1

Why didn't we write the `hasPrevious` method as follows?

```
public boolean hasPrevious() {
    return nextItem.prev != null || (nextItem == null && size != 0);
}
```

If `nextItem` *was null, then the expression* `nextItem.prev` *would result in a* `NullPointerException`.

2.8.3

What happens if we call `remove` after we call `add`? What does the Java API documentation say? What does our implementation do?

The Java API documentation says that an `IllegalStateException` *is thrown if* `remove` *is called after a call to* `add`. *Our implementation will throw a* `NullPointerException` *because* `lastItemReturned` *is set to null by* `add`.

2.9.1

Look at the `AbstractCollection` definition in the Java API documentation. What methods are abstract? Could we use the `KWArrayList` and extend the `AbstractCollection`, but not the `AbstractList`, to develop an implementation of the `Collection` interface? How about using the `KWLinkedList` and the `AbstractCollection`, but not the `AbstractSequentialList`?

Only two methods, `iterator` *and* `size`*, are abstract in the* `AbstractCollection` *class. Our* `KWArrayList` *class does not implement the* `iterator` *method, so it does not meet the* `Collection` *interface. Our* `KWLinkedList` *class implements the* `iterator` *and* `size` *methods, thus, by extending* `AbstractCollection` *it fully implements the* `Collection` *interface. Our* `KWLinkedList` *class also can extend* `AbstractSequentialList` *to implement the* `Collection` *interface. By doing so, it also fully implements the* `List` *interface.*

2.10.1

Why don't we implement the `OrderedList` by extending `LinkedList`? What would happen if someone called the `add` method? How about the `set` method?

By extending the `LinkedList` *class we expose all of the public methods of the* `LinkedList` *class. If either the* `add` *or* `set` *method were called the invariant that the items were ordered could be violated.*

2.10.3

Why don't we provide a `listIterator` method for the `OrderedList` class?

If we implement the `listIterator` *method using delegation, we would obtain an instance of a* `ListIterator` *that supported the* `add` *and* `set` *methods (see question 10.1). However, we could have defined our own* `ListIterator` *class that delegated to the* `ListIterator` *returned from the underlying* `List` *class, but did not pass the* `add` *and* `set` *methods through.*

2.11.1

Explain why a method that does not match its declaration in the interface would not be discovered during white-box testing.

Methods which do not match their declaration in the interface are detected by the Java compiler, and thus would never get to white-box testing.

2.11.3

List two boundary conditions that should be checked when testing method readInt below.

```
/** Method to return an integer data value.
    @param prompt Message
    @return The data value read as an int
*/
public static int readInt(String prompt) {
    while (true) { // Repeat until valid number is read.
        try {
            String numStr = JOptionPane.showInputDialog(prompt);
            return Integer.parseInt(numStr);
        }
        catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog (null,
                "Bad numeric string - Try again",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

```
  /** Method to return an integer data value between two
      specified end points.
      pre: minN <= maxN.
      @param prompt Message
      @param minN Smallest value in range
      @param maxN Largest value in range
      @throws IllegalArgumentException
      @return The first data value that is in range
   */
  public static int readInt(String prompt, int minN, int maxN) {
    if (minN > maxN) {
      throw new IllegalArgumentException(
          "In readInt, minN " + minN
          + " not <= maxN " + maxN);
    }
    boolean inRange = false; // Assume no valid number read.
    int n = 0;
    while (!inRange) { // Repeat until valid number read.
      try {
        String line = JOptionPane.showInputDialog(
            prompt + "\nEnter an integer between "
            + minN + " and " + maxN);
        n = Integer.parseInt(line);
```

```
        inRange = (minN <= n && n <= maxN);
      }
      catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(
            null,
            "Bad numeric string - Try again",
            "Error", JOptionPane.ERROR_MESSAGE);
      }
    } // End while
    return n; // n is in range
  }
```

`minN == maxN` (e.g. minN = maxN = 1), and test for input 1 and 2
`minN > maxN` (e.g. minN = 5 and maxN = 2)