■ **R15.4** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

■ **R15.5** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.6** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.7** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next().equals("Tom")) { iterator.remove(); }
while (iterator.hasNext())  { System.out.println(iterator.next()); }
```

■ **R15.8** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
```

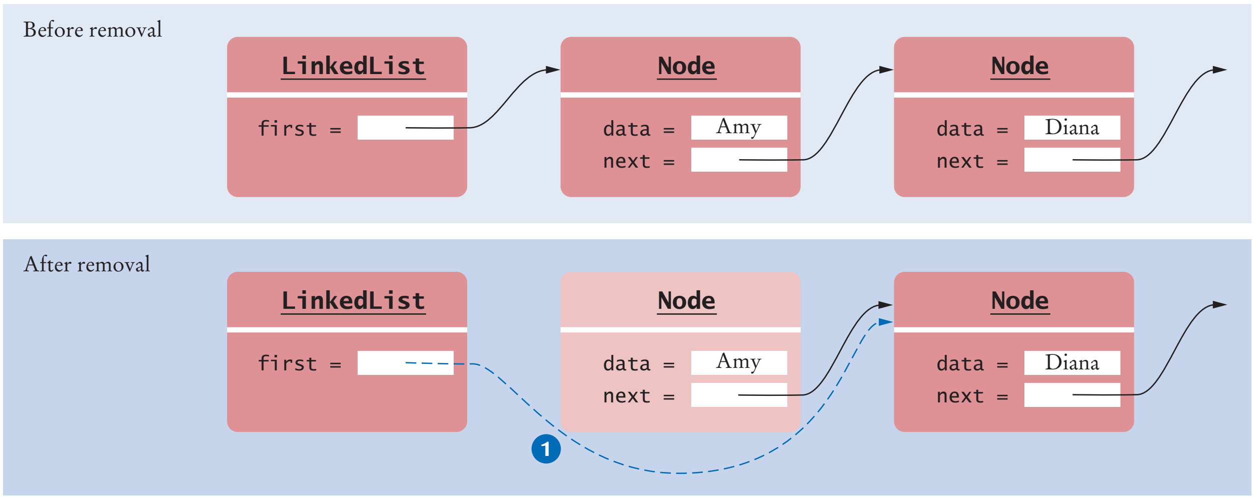

Figure 2 Removing the First Node from a Linked List

Removing the first element of the list works as follows. The data of the first node are saved and later returned as the method result. The successor of the first node becomes the first node of the shorter list (see Figure 2). Then there are no further references to the old node, and the garbage collector will eventually recycle it.

```
public class LinkedList
{
   . . .
   public Object removeFirst()
   {
      if (first == null) { throw new NoSuchElementException(); }
      Object element = first.data;
      first = first.next; 1
      return element;
   }
   . . .
}
```

16.1.3 The Iterator Class

The `ListIterator` interface in the standard library declares nine methods. Our simplified `ListIterator` interface omits four of them (the methods that move the iterator backward and the methods that report an integer index of the iterator). Our interface requires us to implement list iterator methods `next`, `hasNext`, `remove`, `add`, and `set`.

Our `LinkedList` class declares a private inner class `LinkedListIterator`, which implements our simplified `ListIterator` interface. Because `LinkedListIterator` is an inner class, it has access to the private features of the `LinkedList` class—in particular, the instance variable `first` and the private `Node` class.

Note that clients of the `LinkedList` class don't actually know the name of the iterator class. They only know it is a class that implements the `ListIterator` interface.

A list iterator object has a reference to the last visited node.

Each iterator object has a reference, `position`, to the currently visited node. We also store a reference to the last node before that, `previous`. We will need that reference to adjust the links properly in the `remove` method. Finally, because calls to `remove` and `set`

```
      public LinkedList() { first = null; }

      public Object getFirst()
      {
         if (first == null) { throw new NoSuchElementException(); }
         return first.data;
      }
   }
```

16.1.2 Adding and Removing the First Element

When adding or removing the first element, the reference to the first node must be updated.

Figure 1 shows the addFirst method in action. When a new node is added, it becomes the head of the list, and the node that was the old list head becomes its next node:

```
public class LinkedList
{
   . . .
   public void addFirst(Object element)
   {
      Node newNode = new Node(); ❶
      newNode.data = element;
      newNode.next = first; ❷
      first = newNode; ❸
   }
   . . .
}
```

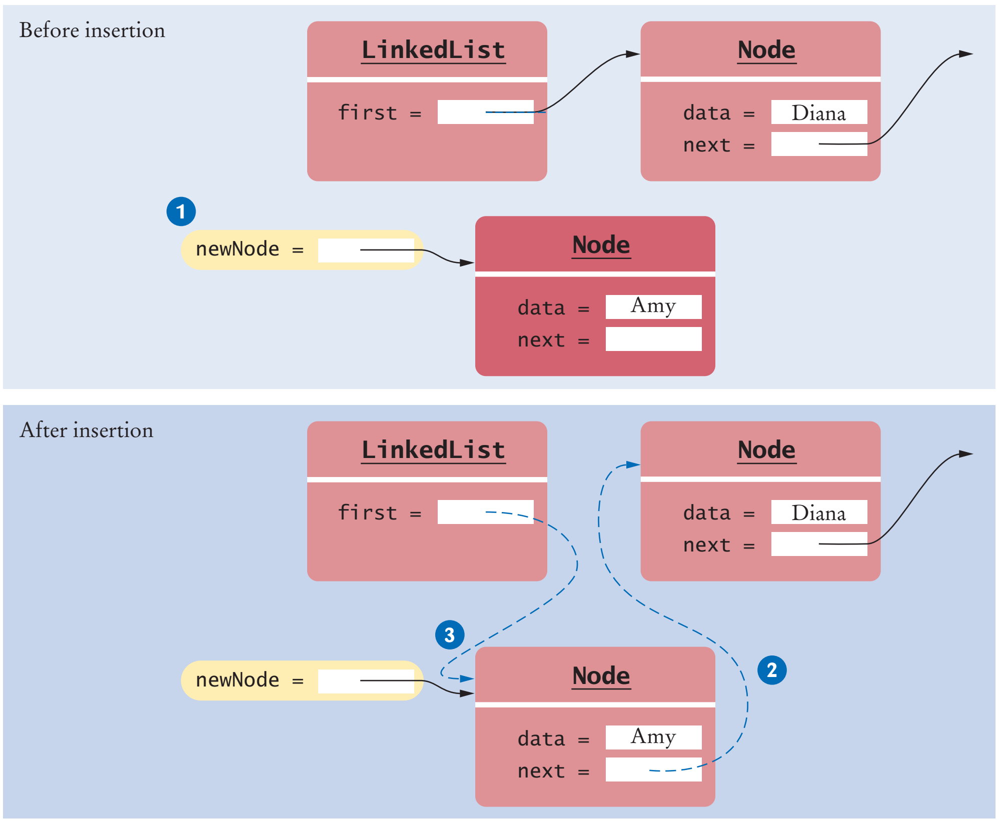

Figure 1 Adding a Node to the Head of a Linked List

```
   */
   void set(Object element);
}
```

SELF CHECK

1. Trace through the `addFirst` method when adding an element to an empty list.
2. Conceptually, an iterator is located between two elements (see Figure 9 in Chapter 15). Does the `position` instance variable refer to the element to the left or the element to the right?
3. Why does the `add` method have two separate cases?
4. Assume that a `last` reference is added to the `LinkedList` class, as described in Section 16.1.8. How does the `add` method of the `ListIterator` need to change?
5. Provide an implementation of an `addLast` method for the `LinkedList` class, assuming that there is no `last` reference.
6. Expressed in big-Oh notation, what is the efficiency of the `addFirst` method of the `LinkedList` class? What is the efficiency of the `addLast` method of Self Check 5?
7. How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

Practice It Now you can try these exercises at the end of the chapter: R16.1, E16.2, E16.4, E16.6.

Special Topic 16.1

Static Classes

You first saw the use of inner classes for event handlers in Chapter 10. Inner classes are useful in that context, because their methods have the privilege of accessing private instance variables of outer-class objects. The same is true for the `LinkedListIterator` inner class in the sample code for this section. The iterator needs to access the `first` instance variable of its linked list.

However, there is a cost for this feature. Every object of the inner class has a reference to the object of the enclosing class that constructed it. If an inner class has no need to access the enclosing class, you can declare the class as static and eliminate the reference to the enclosing class. This is the case with the `Node` class.

You can declare it as follows:

```
public class LinkedList
{
   . . .
   static class Node
   {
      . . .
   }
}
```

However, the `LinkedListIterator` class cannot be a static class. Its methods must access the `first` element of the enclosing `LinkedList`.

WORKED EXAMPLE 16.1 **Implementing a Doubly-Linked List**

Learn how to modify a singly-linked list to implement a doubly-linked list. Go to `wiley.com/go/javaexamples` and download Worked Example 16.1.

To get to the kth node of a linked list, one must skip over the preceding nodes.

method in Section 16.1.6. To add an element, one updates a couple of references in the neighboring nodes and the iterator. This operation requires a constant number of steps, independent of the size of the linked list.

Using the big-Oh notation, an operation that requires a bounded amount of time, regardless of the total number of elements in the structure, is denoted as $O(1)$. Adding an element to a linked list takes $O(1)$ time.

Similar reasoning shows that removing an element at a given position is an $O(1)$ operation.

Now consider the task of adding an element at the end of the list. We first need to get to the end, at a cost of $O(n)$. Then it takes $O(1)$ time to add the element. However, we can improve on this performance if we add a reference to the last node to the `LinkedList` class:

```
public class LinkedList
{
   private Node first;
   private Node last;
   . . .
}
```

Of course, this reference must be updated when the last node changes, as elements are added or removed. In order to keep the code as simple as possible, our implementation does not have a reference to the last node. However, we will always assume that a linked list implementation can access the last element in constant time. This is the case for the `LinkedList` class in the standard Java library, and it is an easy enhancement to our implementation. Worked Example 16.1 shows how to add the `last` reference, update it as necessary, and provide an `addLast` method for adding an element at the end.

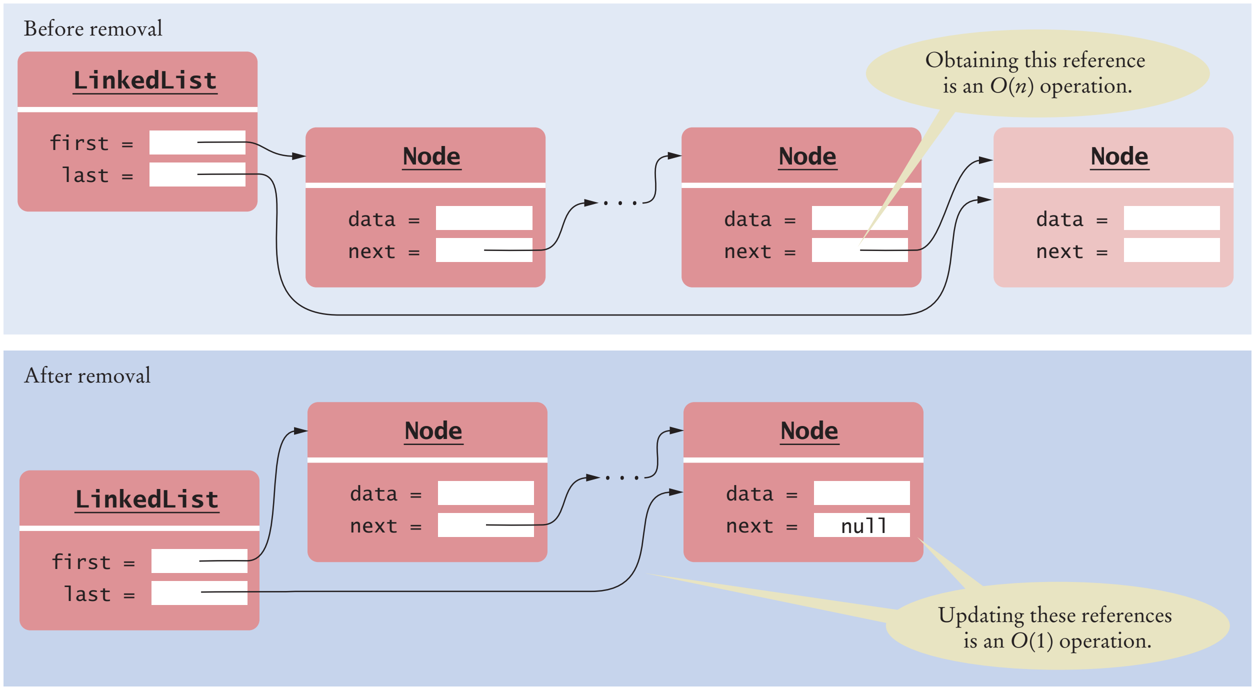

Figure 5 Removing the Last Element of a Singly-Linked List

The same is true when you remove a node (see Figure 8). What's the catch? Linked lists allow efficient insertion and removal, but element access can be inefficient.

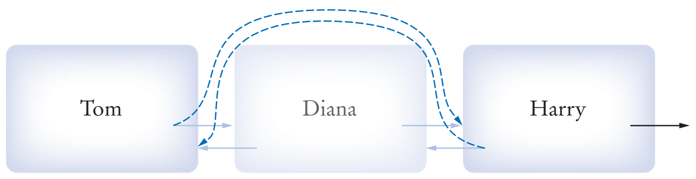

Figure 8 Removing a Node from a Linked List

Adding and removing elements at a given location in a linked list is efficient.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term "random access" is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

15.2.2 The LinkedList Class of the Java Collections Framework

The Java library provides a `LinkedList` class in the `java.util` package. It is a **generic class**, just like the `ArrayList` class. That is, you specify the type of the list elements in angle brackets, such as `LinkedList<String>` or `LinkedList<Employee>`.

Table 2 shows important methods of the `LinkedList` class. (Remember that the `LinkedList` class also inherits the methods of the `Collection` interface shown in Table 1.)

As you can see from Table 2, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**. We discuss iterators next.

Table 2 Working with Linked Lists

`LinkedList<String> list =`
`   new LinkedList<String>();`
	An empty list.

`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 678).

are only valid after a call to next, we use the isAfterNext flag to track when the next method has been called.

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }

   class LinkedListIterator implements ListIterator
   {
      private Node position;
      private Node previous;
      private boolean isAfterNext;

      public LinkedListIterator()
      {
         position = null;
         previous = null;
         isAfterNext = false;
      }
      . . .
   }
}
```

16.1.4 Advancing an Iterator

To advance an iterator, update the position and remember the old position for the remove method.

When advancing an iterator with the next method, the position reference is updated to position.next, and the old position is remembered in previous. The previous position is used for just one purpose: to remove the element if the remove method is called after the next method.

There is a special case, however—if the iterator points before the first element of the list, then the old position is null, and position must be set to first:

```
class LinkedListIterator implements ListIterator
{
   . . .
   public Object next()
   {
      if (!hasNext()) { throw new NoSuchElementException(); }
      previous = position; // Remember for remove
      isAfterNext = true;

      if (position == null)
      {
         position = first;
      }
      else
      {
         position = position.next;
      }

      return position.data;
   }
   . . .
}
```

In the preceding chapter, you learned how to use the collection classes in the Java library. In this and the next chapter, we will study how these classes are implemented. This chapter deals with simple data structures in which elements are arranged in a linear sequence. By investigating how these data structures add, remove, and locate elements, you will gain valuable experience in designing algorithms and estimating their efficiency.

16.1 Implementing Linked Lists

In the last chapter you saw how to use the linked list class supplied by the Java library. Now we will look at the implementation of a simplified version of this class. This will show you how the list operations manipulate the links as the list is modified.

To keep this sample code simple, we will not implement all methods of the linked list class. We will implement only a singly-linked list, and the list class will supply direct access only to the first list element, not the last one. (A worked example and several exercises explore additional implementation options.) Our list will not use a type parameter. We will simply store raw `Object` values and insert casts when retrieving them. (You will see how to use type parameters in Chapter 18.) The result will be a fully functional list class that shows how the links are updated when elements are added or removed, and how the iterator traverses the list.

16.1.1 The Node Class

A linked list stores elements in a sequence of nodes. We need a class to represent the nodes. In a singly-linked list, a `Node` object stores an element and a reference to the next node.

Because the methods of both the linked list class and the iterator class have frequent access to the `Node` instance variables, we do not make the instance variables of the `Node` class private. Instead, we make `Node` a private **inner class** of the `LinkedList` class. An inner class is a class that is defined inside another class. The methods of the outer class can access the public features of the inner class. However, because the inner class is private, it cannot be accessed anywhere other than from the outer class.

```
public class LinkedList
{
   . . .
   class Node
   {
      public Object data;
      public Node next;
   }
}
```

A linked list object holds a reference to the first node, and each node holds a reference to the next node.

Our `LinkedList` class holds a reference `first` to the first node (or `null`, if the list is completely empty):

```
public class LinkedList
{
   private Node first;
```

FULL CODE EXAMPLE
Go to wiley.com/go/javacode to download programs that demonstrate safety problems when using collections without type parameters.

The `add` method of the generic `ArrayList` class is safer. It is impossible to add a `String` object into an `ArrayList<BankAccount>`, but you can accidentally add a `String` into a `LinkedList` that is intended to hold bank accounts:

```
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList(); // Should hold BankAccount objects
accounts1.add("my savings"); // Compile-time error
accounts2.addFirst("my savings"); // Not detected at compile time
```

The latter will result in a class cast exception when some other part of the code retrieves the string, believing it to be a bank account:

```
BankAccount account = (BankAccount) accounts2.getFirst(); // Run-time error
```

Type parameters make generic code safer and easier to read.

Code that uses the generic `ArrayList` class is also easier to read. When you spot an `ArrayList<BankAccount>`, you know right away that it must contain bank accounts. When you see a `LinkedList`, you have to study the code to find out what it contains.

In Chapters 16 and 17, we used inheritance to implement generic linked lists, hash tables, and binary trees, because you were already familiar with the concept of inheritance. Using type parameters requires new syntax and additional techniques—those are the topic of this chapter.

SELF CHECK

1. The standard library provides a class `HashMap<K, V>` with key type `K` and value type `V`. Declare a hash map that maps strings to integers.
2. The binary search tree class in Chapter 17 is an example of generic programming because you can use it with any classes that implement the `Comparable` interface. Does it achieve genericity through inheritance or type parameters?
3. Does the following code contain an error? If so, is it a compile-time or run-time error?
```
ArrayList<Integer> a = new ArrayList<Integer>();
String s = a.get(0);
```
4. Does the following code contain an error? If so, is it a compile-time or run-time error?
```
ArrayList<Double> a = new ArrayList<Double>();
a.add(3);
```
5. Does the following code contain an error? If so, is it a compile-time or run-time error?
```
LinkedList a = new LinkedList();
a.addFirst("3.14");
double x = (Double) a.removeFirst();
```

Practice It Now you can try these exercises at the end of the chapter: R18.4, R18.5, R18.6.

18.2 Implementing Generic Types

In this section, you will learn how to implement your own generic classes. We will write a very simple generic class that stores *pairs* of objects, each of which can have an arbitrary type. For example,

```
Pair<String, Integer> result = new Pair<String, Integer>("Harry Morgan", 1729);
```

In the supermarket, a generic product can be sourced from multiple suppliers. In computer science, generic programming involves the design and implementation of data structures and algorithms that work for multiple types. You have already seen the generic `ArrayList` class that can be used to collect elements of arbitrary types. In this chapter, you will learn how to implement your own generic classes and methods.

18.1 Generic Classes and Type Parameters

Generic programming is the creation of programming constructs that can be used with many different types. For example, the Java library programmers who implemented the `ArrayList` class used the technique of generic programming. As a result, you can form array lists that collect elements of different types, such as `ArrayList<String>`, `ArrayList<BankAccount>`, and so on.

In Java, generic programming can be achieved with inheritance or with type parameters.

The `LinkedList` class that we implemented in Section 16.1 is also an example of generic programming—you can store objects of any class inside a `LinkedList`. That `LinkedList` class achieves genericity by using *inheritance*. It uses references of type `Object` and is therefore capable of storing objects of any class. For example, you can add elements of type `String` because the `String` class extends `Object`. In contrast, the `ArrayList` and `LinkedList` classes from the standard Java library are **generic classes**. Each of these classes has a **type parameter** for specifying the type of its elements. For example, an `ArrayList<String>` stores `String` elements.

A generic class has one or more type parameters.

When declaring a generic class, you supply a variable for each type parameter. For example, the standard library declares the class `ArrayList<E>`, where `E` is the **type variable** that denotes the element type. You use the same variable in the declaration of the methods, whenever you need to refer to that type. For example, the `ArrayList<E>` class declares methods

```
public void add(E element)
public E get(int index)
```

You could use another name, such as `ElementType`, instead of `E`. However, it is customary to use short, uppercase names for type variables.

Type parameters can be instantiated with class or interface types.

In order to use a generic class, you need to *instantiate* the type parameter, that is, supply an actual type. You can supply any class or interface type, for example

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

However, you cannot substitute any of the eight primitive types for a type parameter. It would be an error to declare an `ArrayList<double>`. Use the corresponding wrapper class instead, such as `ArrayList<Double>`.

When you instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class. For example, the `add` method for `ArrayList<BankAccount>` has the type variable `E` replaced with the type `BankAccount`:

```
public void add(BankAccount element)
```

Contrast that with the `add` method of the `LinkedList` class in Chapter 16:

```
public void add(Object element)
```

Therefore, removing an element from the end of a doubly-linked list is also an $O(1)$ operation. Worked Example 16.1 contains a full implementation.

Table 1 summarizes the efficiency of linked list operations.

Table 1 Efficiency of Linked List Operations

Operation	Singly-Linked List	Doubly-Linked List
Access an element.	$O(n)$	$O(n)$
Add/remove at an iterator position.	$O(1)$	$O(1)$
Add/remove first element.	$O(1)$	$O(1)$
Add last element.	$O(1)$	$O(1)$
Remove last element.	$O(n)$	$O(1)$

section_1/LinkedList.java

```java
import java.util.NoSuchElementException;

/**
   A linked list is a sequence of nodes with efficient
   element insertion and removal. This class
   contains a subset of the methods of the standard
   java.util.LinkedList class.
*/
public class LinkedList
{
   private Node first;

   /**
      Constructs an empty linked list.
   */
   public LinkedList()
   {
      first = null;
   }

   /**
      Returns the first element in the linked list.
      @return the first element in the linked list
   */
   public Object getFirst()
   {
      if (first == null) { throw new NoSuchElementException(); }
      return first.data;
   }

   /**
      Removes the first element in the linked list.
      @return the removed element
   */
   public Object removeFirst()
   {
      if (first == null) { throw new NoSuchElementException(); }
      Object element = first.data;
```

16.2 Implementing Array Lists

Array lists were introduced in Chapter 7. They are conceptually similar to linked lists, allowing you to add and remove elements at any position. In the following sections, we will develop an implementation of an array list, study the efficiency of operations on array lists, and compare them with the equivalent operations on linked lists.

16.2.1 Getting and Setting Elements

An array list maintains a reference to an array of elements. The array is large enough to hold all elements in the collection—in fact, it is usually larger to allow for adding additional elements. When the array gets full, it is replaced by a larger one. We discuss that process in Section 16.2.3.

In addition to the internal array of elements, an array list has an instance field that stores the current number of elements (see Figure 7).

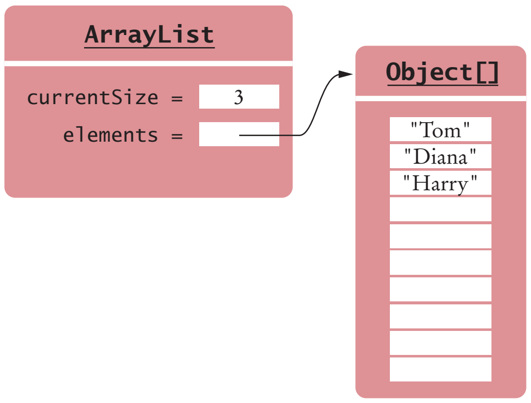

Figure 7
An Array List Stores its Elements in an Array

For simplicity, our `ArrayList` implementation does not work with arbitrary element types, but it simply manages elements of type `Object`. (Chapter 18 shows how to implement classes with type parameters.)

```
public class ArrayList
{
   private Object[] elements;
   private int currentSize;

   public ArrayList()
   {
      final int INITIAL_SIZE = 10;
      elements = new Object[INITIAL_SIZE];
      currentSize = 0;
   }

   public int size() { return currentSize; }
   . . .
}
```

To access array list elements, we provide `get` and `set` methods. These methods simply check for valid positions and access the internal array at the given position:

15.5.2 Queues

A queue is a collection of elements with "first-in, first-out" retrieval.

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*). Queues yield items in a *first-in, first-out* or *FIFO* fashion. Items are removed in the same order in which they were added.

To visualize a queue, think of people lining up.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the "first-in, first-out" rule, which is a fair arrangement for users of the shared printer.

The `Queue` interface in the standard Java library has methods `add` to add an element to the tail of the queue, `remove` to remove the head of the queue, and `peek` to get the head element of the queue without removing it (see Table 8).

The `LinkedList` class implements the `Queue` interface. Whenever you need a queue, simply initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<String>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); } // Prints A B C
```

The standard library provides several queue classes that we do not discuss in this book. Those classes are intended for work sharing when multiple activities (called threads) run in parallel.

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

15.5.3 Priority Queues

When removing an element from a priority queue, the element with the most urgent priority is retrieved.

A **priority queue** collects elements, each of which has a *priority*. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others. Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, it is the item with the most urgent priority.

The code for the addLast method is very similar to the addFirst method in Section 16.1.2. It too requires constant time, independent of the length of the list. We conclude that, with an appropriate implementation, adding an element at the end of a linked list is an $O(1)$ operation.

How about removing the last element? We need a reference to the next-to-last element, so that we can set its next reference to null. (See Figure 5.)

We also need to update the last reference and set it to the next-to-last reference. But how can we get that next-to-last reference? It takes $n - 1$ iterations to obtain it, starting at the beginning of the list. Thus, removing an element from the back of a singly-linked list is an $O(n)$ operation.

We can do better in a doubly-linked list, such as the one in the standard Java library. In a doubly-linked list, each node has a reference to the previous node in addition to the next one (see Figure 6).

```
public class LinkedList
{
   . . .
   class Node
   {
      public Object data;
      public Node next;
      public Node previous;
   }
}
```

In that case, removal of the last element takes a constant number of steps:

```
last = last.previous; 1
last.next = null; 2
```

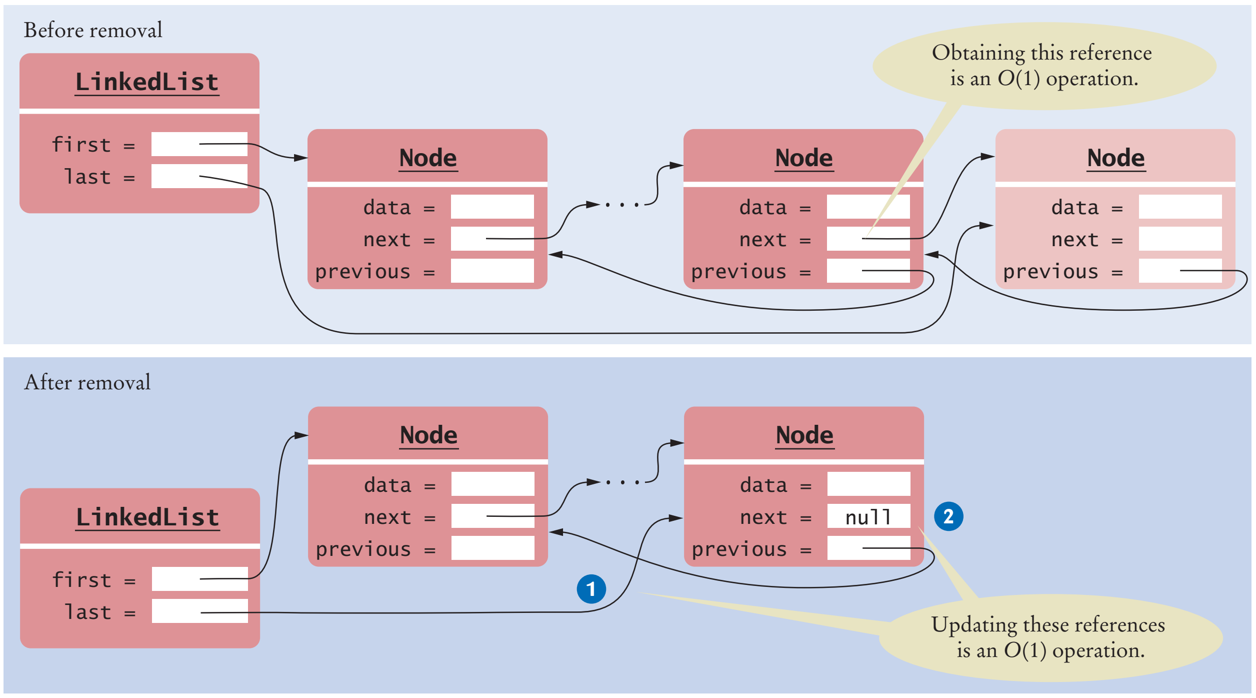

Figure 6 Removing the Last Element of a Doubly-Linked List

```
public void addFirst(Object element)
{
   Node newNode = new Node();
   first = newNode;
   newNode.data = element;
   newNode.next = first;
}
```

Develop a program `ListTest` with a test case that shows the error. That is, the program should print a failure message with this implementation but not with the correct implementation.

■■ **E16.3** Consider a version of the `LinkedList` class of Section 16.1 in which the iterator's `hasNext` method has been replaced with the following faulty version:

```
public boolean hasNext()
{
   return position != null;
}
```

Develop a program `ListTest` with a test case that shows the error. That is, the program should print a failure message with this implementation but not with the correct implementation.

■ **E16.4** Add a method `size` to our implementation of the `LinkedList` class that computes the number of elements in the list by following links and counting the elements until the end of the list is reached.

■■ **E16.5** Solve Exercise E16.4 recursively by calling a recursive helper method

```
private static int size(Node start)
```

Hint: If `start` is `null`, then the size is `0`. Otherwise, it is one larger than the size of `start.next`.

■ **E16.6** Add an instance variable `currentSize` to our implementation of the `LinkedList` class. Modify the `add`, `addLast`, and `remove` methods of both the linked list and the list iterator to update the `currentSize` variable so that it always contains the correct size. Change the `size` method of Exercise E16.4 so that it simply returns the value of this instance variable.

■■■ **E16.7** Reimplement the `LinkedList` class of Section 16.1 so that the `Node` and `LinkedListIterator` classes are not inner classes.

■■■ **E16.8** Reimplement the `LinkedList` class of Section 16.1 so that it implements the `java.util.LinkedList` interface. *Hint:* Extend the `java.util.AbstractList` class.

■■■ **E16.9** Provide a `listIterator` method for the `ArrayList` implementation in Section 16.2. Your method should return an object of a class implementing `java.util.ListIterator`. Also have the `ArrayList` class implement the `Iterable` interface type and provide a test program that demonstrates that your array list can be used in an enhanced `for` loop.

■ **E16.10** Provide a `removeLast` method for the `ArrayList` implementation in Section 16.2 that shrinks the internal array by 50 percent when it is less than 25 percent full.

■ **E16.11** Complete the implementation of a stack in Section 16.3.2, using an array for storing the elements.

■ **E16.12** Complete the implementation of a queue in Section 16.3.3, using a sequence of nodes for storing the elements.

section_2/ListDemo.java

```
import java.util.LinkedList;
import java.util.ListIterator;

/**
   This program demonstrates the LinkedList class.
*/
public class ListDemo
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Diana");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT

      // Add more elements after second element

      iterator.add("Juliet"); // DHJ|RT
      iterator.add("Nina"); // DHJN|RT

      iterator.next(); // DHJNR|T

      // Remove last traversed element

      iterator.remove(); // DHJN|T

      // Print all elements

      System.out.println(staff);
      System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
   }
}
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

5. Do linked lists take more storage space than arrays of the same size?
6. Why don't we need iterators with arrays?
7. Suppose the list `letters` contains elements "A", "B", "C", and "D". Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
```

15.2.3 List Iterators

You use a list iterator to access elements inside a linked list.

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor in a word processor points between two characters (see Figure 9). In the conceptual view, think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

You obtain a list iterator with the `listIterator` method of the `LinkedList` class:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Note that the iterator class is also a generic type. A `ListIterator<String>` iterates through a list of strings; a `ListIterator<Book>` visits the elements in a `LinkedList<Book>`.

Initially, the iterator points before the first element. You can move the iterator position with the `next` method:

```
iterator.next();
```

The `next` method throws a `NoSuchElementException` if you are already past the end of the list. You should always call the iterator's `hasNext` method before calling `next`—it returns `true` if there is a next element.

```
if (iterator.hasNext())
{
   iterator.next();
}
```

The `next` method returns the element that the iterator is passing. When you use a `ListIterator<String>`, the return type of the `next` method is `String`. In general, the return type of the `next` method matches the list iterator's type parameter (which reflects the type of the elements in the list).

You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   Do something with name
}
```

As a shorthand, if your loop simply visits all elements of the linked list, you can use the "for each" loop:

```
for (String name : employeeNames)
{
   Do something with name
}
```

Then you don't have to worry about iterators at all. Behind the scenes, the `for` loop uses an iterator to visit all list elements.

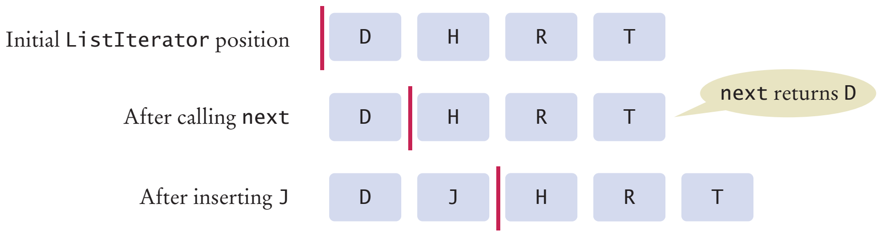

Figure 9
A Conceptual View of the List Iterator

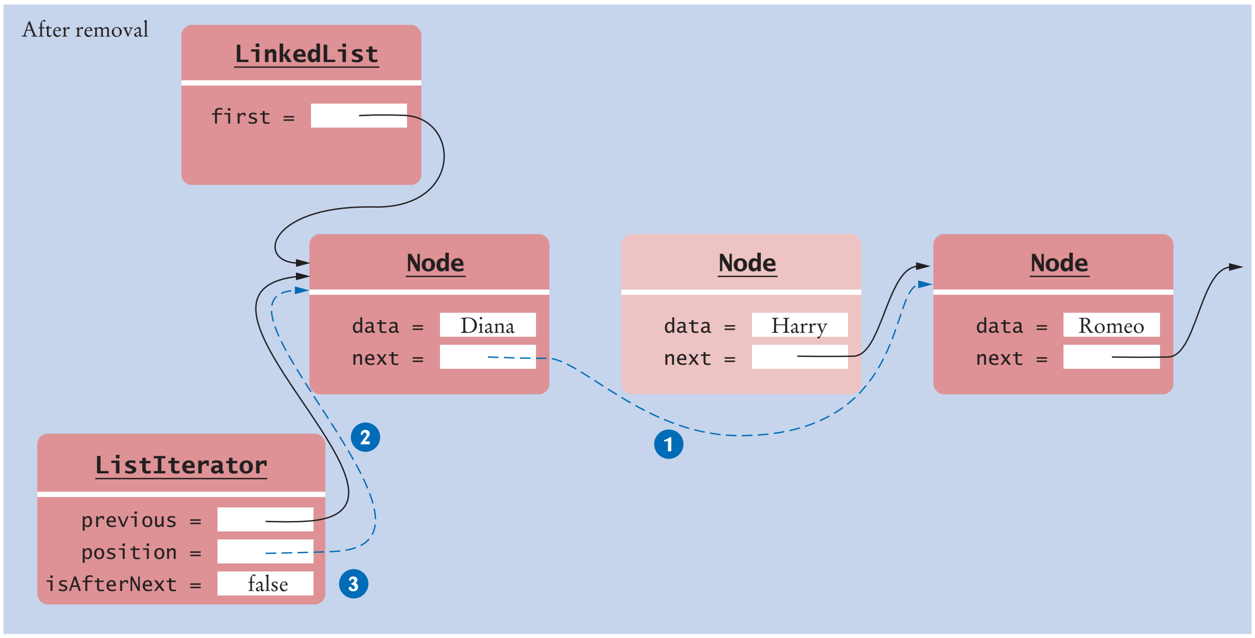

Figure 3 (continued) Removing a Node from the Middle of a Linked List

According to the specification of the `remove` method, it is illegal to call `remove` twice in a row. Our implementation handles this situation correctly. After completion of the `remove` method, the `isAfterNext` flag is set to `false`. An exception occurs if `remove` is called again without another call to `next`.

```
class LinkedListIterator implements ListIterator
{
   . . .
   public void remove()
   {
      if (!isAfterNext) { throw new IllegalStateException(); }

      if (position == first)
      {
         removeFirst();
      }
      else
      {
         previous.next = position.next;  1
      }
      position = previous;  2

      isAfterNext = false;      3
   }
   . . .
}
```

There is a good reason for disallowing `remove` twice in a row. After the first call to `remove`, the current position reverts to the predecessor of the removed element. Its predecessor is no longer known, which makes it impossible to efficiently remove the current element.