Chapter 15 An Introduction to Data Structures

CHAPTER GOALS

- To learn how to use the linked lists provided in the standard library
- To be able to use iterators to traverse linked lists
- To understand the implementation of linked lists
- To distinguish between abstract and concrete data types
- To know the efficiency of fundamental operations of lists and arrays
- To become familiar with the stack and queue types

Up to this point, we used arrays as a one-size-fits-all mechanism for collecting objects. However, computer scientists have developed many different data structures that have varying performance tradeoffs. In this chapter, you will learn about the *linked list*, a data structure that allows you to add and remove elements efficiently, without moving any existing elements. You will also learn about the distinction between concrete and abstract data types. An abstract type spells out what fundamental operations should be supported efficiently, but it leaves the implementation unspecified. The stack and queue types, introduced at the end of this chapter, are examples of abstract types.

15.1 Using Linked Lists

A *linked list* is a data structure used for collecting a sequence of objects, which allows efficient addition and removal of elements in the middle of the sequence.

To understand the need for such a data structure, imagine a program that maintains a sequence of employee objects, sorted by the last names of the employees. When a new employee is hired, an object needs to be inserted into the sequence. Unless the company happened to hire employees in dictionary order, the new object probably needs to be inserted somewhere near the middle of the sequence. If we use an array to store the objects, then all objects following the new hire must be moved toward the end.

Conversely, if an employee leaves the company, the object must be removed, and the hole in the sequence needs to be closed up by moving all objects that come after it. Moving a large number of values can involve a substantial amount of processing time. We would like to structure the data in a way that minimizes this cost.

A linked list consists of a number of nodes, each of which has a reference to the next node.

Rather than storing the values in an array, a linked list uses a sequence of *nodes*. Each node stores a value and a reference to the next node in the sequence (see Figure 1). When you insert a new node into a linked list, only the neighboring node references need to be updated. The same is true when you remove a node. What's the catch? Linked lists allow speedy insertion and removal, but element access can be slow.

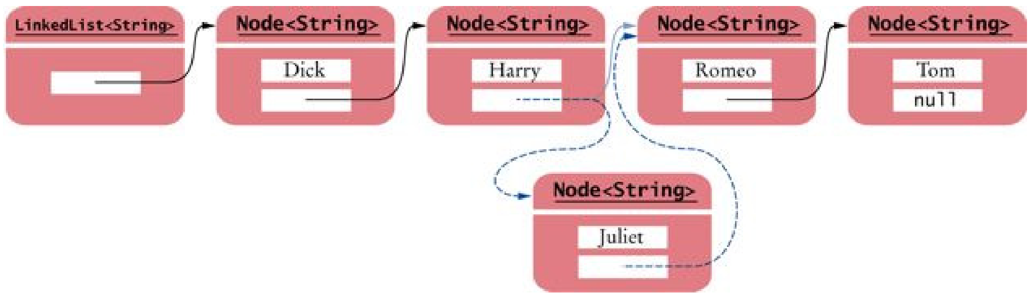
Adding and removing elements in the middle of a linked list is efficient.

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. For example, a binary search requires random access, whereas a linear search requires sequential access.

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

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. 666

Figure 1



Inserting an Element into a Linked List

The Java library provides a linked list class. In this section you will learn how to use the library class. In the next section you will peek under the hood and see how some of its key methods are implemented.

The `LinkedList` class in the `java.util` package 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<Product>`.

The following methods give you direct access to the first and the last element in the list. Here, `E` is the element type of `LinkedList<E>`.

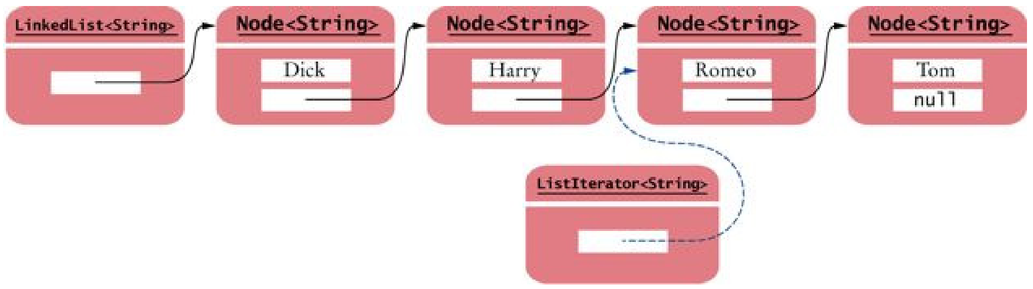
```
void addFirst(E element)
void addLast(E element)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

How do you add and remove elements in the middle of the list? The list will not give you references to the nodes. If you had direct access to them and somehow messed them up, you would break the linked list. As you will see in the next section, where you implement some of the linked list operations yourself, keeping all links between nodes intact is not trivial.

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

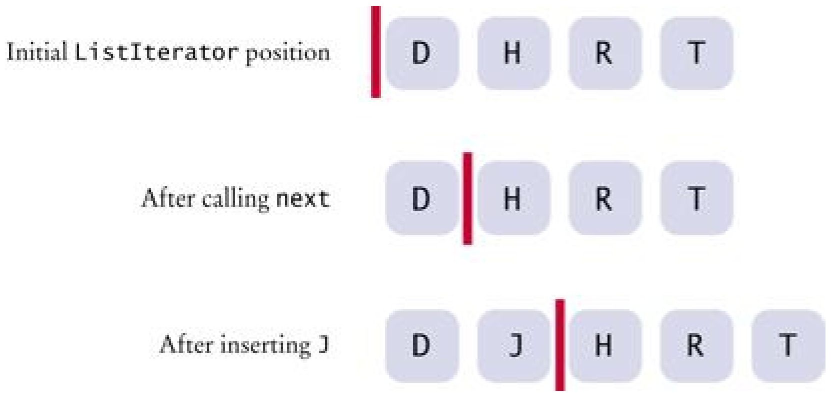
Instead, the Java library supplies a `ListIterator` type. A list iterator encapsulates a position anywhere inside the linked list (see Figure 2).

Figure 2



A List Iterator

Figure 3



A Conceptual View of the List Iterator

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 3). 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<Product>` visits the elements in a `LinkedList<Product>`.

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 method `hasNext` 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 type parameter.

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 (see Advanced Topic 15.1).

The nodes of the `LinkedList` class store two links: one to the next element and one to the `previous` one. Such a list is called a *doubly linked list*. You can use the `previous` and `hasPrevious` methods of the `ListIterator` interface to move the iterator position backwards.

The `add` method adds an object after the iterator, then moves the iterator position past the new element.

```
iterator.add("Juliet");
```

You can visualize insertion to be like typing text in a word processor. Each character is inserted after the cursor, and then the cursor moves past the inserted character (see Figure 3). Most people never pay much attention to this—you may want to try it out and watch carefully how your word processor inserts characters.

The `remove` method removes the object that was returned by the last call to `next` or `previous`. For example, the following loop removes all names that fulfill a certain condition:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name fulfills condition)
      iterator.remove();
}
```

You have to be careful when calling `remove`. It can be called only once after calling `next` or `previous`, and you cannot call it immediately after a call to `add`. If you call the method improperly, it throws an `IllegalStateException`.

Here is a sample program that inserts strings into a list and then iterates through the list, adding and removing elements. Finally, the entire list is printed. The comments indicate the iterator position.

ch15/uselist/ListTester.java

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

/**
    A program that tests the LinkedList class.
```

```java
*/
public class ListTester
{
   public static void main(String[] args)
    {
       LinkedList<String> staff = new LinkedList<String>();
       staff.addLast("Dick");
       staff.addLast("Harry");
```

```java
       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

       for (String name : staff)
          System.out.print(iterator.next() + " ");
       System.out.println();
       System. out. println("Expected: Dick Harry Juliet Nina Tom");
    }
}
```

Output

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Self Check

1. Do linked lists take more storage space than arrays of the same size?

2. Why don't we need iterators with arrays?

Advanced Topic 15.1: The Iterable Interface and the "For Each" Loop

You can use the "for each" loop

```
for (Type variable : collection)
```

with any of the collection classes in the standard Java library. This includes the `ArrayList` and `LinkedList` classes as well as the library classes which will be discussed in Chapter 16. In fact, the "for each" loop can be used with any class that implements the `Iterable` interface:

```
public interface Iterable<E>
{
      Iterator<E> iterator();
}
```

The interface has a type parameter `E`, denoting the element type of the collection. The single method, `iterator`, yields an object that implements the `Iterator` interface.

```
public interface Iterator<E>
{
      boolean hasNext();
      E next();
      void remove();
}
```

The ListIterator interface that you saw in the preceding section is a subinterface of Iterator with additional methods (such as add and previous).

The compiler translates a "for each" loop into an equivalent loop that uses an iterator. The loop

```
for (Type variable : collection)
   body
```

is equivalent to

```
Iterator<Type> iter = collection.iterator();
while (iter.hasNext())
{
     Type variable = iter.next();
     body
}
```

The ArrayList and LinkedList classes implement the Iterable interface. If your own classes implement the Iterable interface, you can use them with the "for each" loop as well—see Exercise P15.15.

15.2 Implementing Linked Lists

In the last section you saw how to use the linked list class supplied by the Java library. In this section, we will look at the implementation of a simplified version of this class. This shows 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. Our list will not use a type parameter. We will simply store raw Object values and insert casts when retrieving 671
them. The result will be a fully functional list class that shows how the links are 672
updated in the add and remove operations and how the iterator traverses the list.

A Node object stores an object 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 private. Instead, we make

Node a private inner class of the LinkedList class. Because none of the list methods returns a Node object, it is safe to leave the instance variables public.

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

The LinkedList class holds a reference first to the first node (or null, if the list is completely empty).

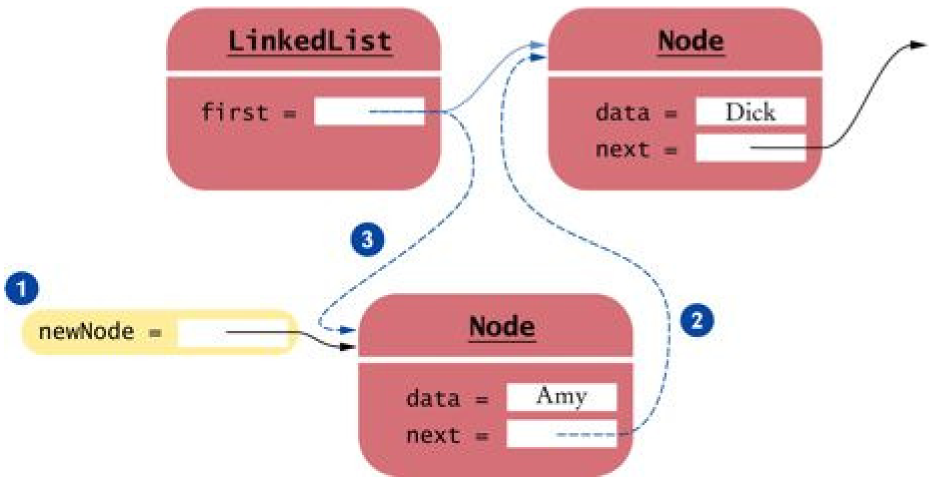
```
public class LinkedList
{
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
   . . .
   private Node first;
}
```

Now let us turn to the addFirst method (see Figure 4). When a new node is added to the list, 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();
      newNode.data = element;
```

```
            newNode.next = first;
            first = newNode;
        }
        . . .
    }
```

Figure 4



Adding a Node to the Head of a Linked List

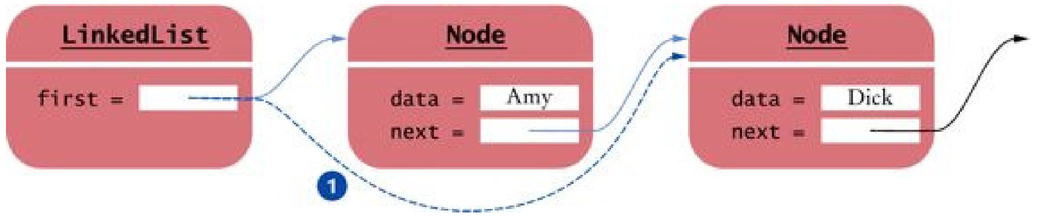
Removing the first element of the list works as follows. The data of the first node are saved and later returned as the method result. The successor of the first node becomes the first node of the shorter list (see Figure 5). Then there are no further references to the old node, and the garbage collector will eventually recycle it.

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

```
    }
```

Next, let us turn to the iterator class. The `ListIterator` interface in the standard library defines nine methods. We omit four of them (the methods that move the iterator backwards and the methods that report an integer index of the iterator).

Figure 5



Removing the First Node from a Linked List

673

674

Our `LinkedList` class defines a private inner class `LinkedListIterator`, which implements the simplified `ListIterator` interface. Because `LinkedListIterator` is an inner class, it has access to the private features of the `LinkedList` class—in particular, the `first` field 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.

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }
   private class LinkedListIterator
      implements ListIterator
   {
      public LinkedListIterator()
      {
         position = null;
         previous = null;
      }
      . . .
```

```
         private Node position;
         private Node previous;
      }
      . . .
   }
```

Each iterator object has a reference `position` to the last visited node. We also store a reference to the last node before that. We will need that reference to adjust the links properly in the `remove` method.

The `next` method is simple. The `position` reference is advanced to `position.next`, and the old position is remembered in `previous`. 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`.

```
   private class LinkedListIterator
      implements ListIterator
   {
      . . .
      public Object next()
      {
         if (!hasNext())
            throw new NoSuchElementException();
         previous = position; // Remember for remove
         if (position == null)
            position = first;
         else
            position = position.next;
         return position.data;
```

```
      }
      . . .
   }
```

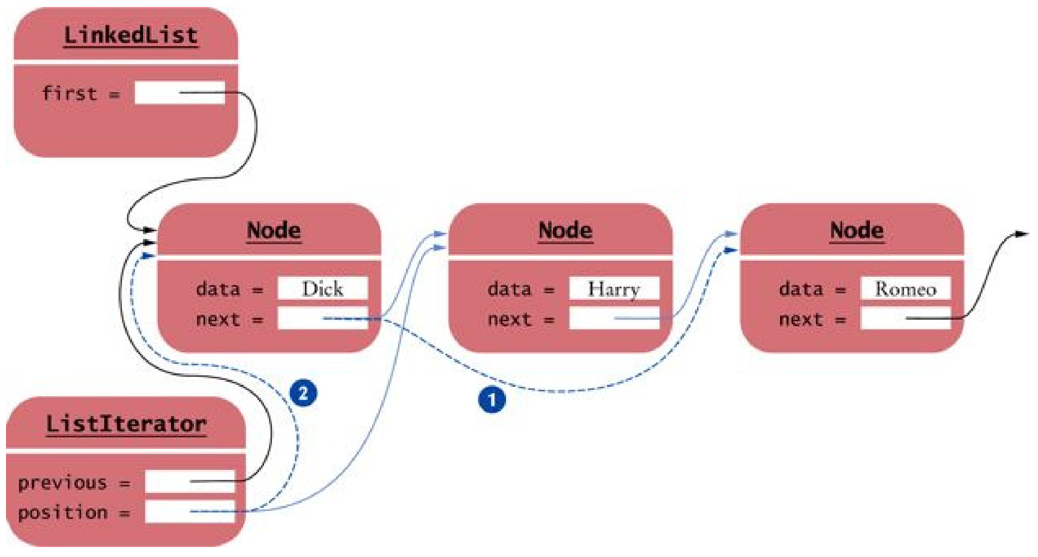
The `next` method is supposed to be called only when the iterator is not yet at the end of the list. The iterator is at the end if the list is empty (that is, `first == null`) or if there is no element after the current position (`position.next == null`).

```
   private class LinkedListIterator
      implements ListIterator
   {
      . . .
      public boolean hasNext()
      {
         if (position == null)
            return first ! = null;
         else
            return position.next ! = null;
      }
      . . .
   }
```

Removing the last visited node is more involved. If the element to be removed is the first element, we just call `removeFirst`. Otherwise, an element in the middle of the list must be removed, and the node preceding it needs to have its `next` reference updated to skip the removed element (see Figure 6). If the `previous` reference equals `position`, then this call to `remove` does not immediately follow a call to `next`, and we throw an `IllegalStateException`.

Implementing operations that modify a linked list is challenging—you need to make sure that you update all node references correctly.

Figure 6

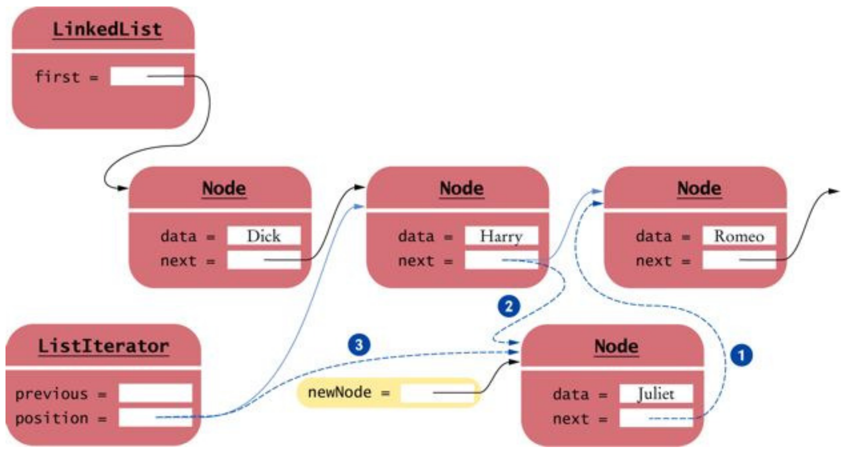


Removing a Node from the Middle of a Linked List

According to the definition of the `remove` method, it is illegal to call `remove` twice in a row. Therefore, the `remove` method sets the `previous` reference to `position`.

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
   public void remove()
   {
      if (previous == position)
         throw new IllegalStateException();
      if (position == first)
      {
         removeFirst();
      }
      else
      {
         previous.next = position.next;
      }
      position = previous;
   }
   . . .
}
```

Figure 7



Adding a Node to the Middle of a Linked List

The set method changes the data stored in the previously visited element. Its implementation is straightforward because our linked lists can be traversed in only one direction. The linked-list implementation of the standard library must keep track 676 677 of whether the last iterator movement was forward or backward. For that reason, the standard library forbids a call to the set method following an add or remove method. We do not enforce that restriction.

```
public void set(Object element)
{
   if (position == null)
      throw new NoSuchElementException();
   position.data = element;
}
```

Finally, the most complex operation is the addition of a node. You insert the new node after the current position, and set the successor of the new node to the successor of the current position (see Figure 7).

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
   public void add(Object element)
   {
      if (position == null)
      {
         addFirst(element);
         position = first;
      }
      else
      {
         Node newNode = new Node();
         newNode.data = element;
         newNode.next = position.next;
         position.next = newNode;
         position = newNode;
      }
      previous = position;
   }
   . . .
}
```

At the end of this section is the complete implementation of our `LinkedList` class.

You now know how to use the `LinkedList` class in the Java library, and you have had a peek “under the hood” to see how linked lists are implemented.

ch15/impllist/LinkedList.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
{
/**
   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;
   first = first. next;
   return element;
}

/**
   Adds an element to the front of the linked list.
   @param elementthe element to add
*/
public void addfirst(Object element)
{
   Node newNode = new Node();
   newNode.data = element;
   newNode.next = first;
   first = newNode;
}

/**
   Returns an iterator for iterating through this list.
   @return an iterator for iterating through this list
*/
public ListIterator listIterator()
{
   return new LinkedListIterator();
}

```

```
   private Node first;

   private class Node
   {
      public Object data;
      public Node next;
   }

   private class LinkedListIterator implements ListIterator
   {
```

```
   /**
      Constructs an iterator that points to the front
      of the linked list.
   */
   public LinkedListIterator()
   {
      position = null;
      previous = null;
   }

   /**
      Moves the iterator past the next element.
      @return the traversed element
   */
   public Object next()
   {
      if (!hasNext())
         throw new NoSuchElementException();
      previous = position; // Remember for remove

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

      return position.data;
   }

   /**
      Tests if there is an element after the iterator
      position.
      @return true if there is an element after the iterator
      position
   */
   public boolean hasNext()
   {
      if (position == null)
         return first != null;
      else
         return position.next != null;
   }

   /**
      Adds an element before the iterator position
      and moves the iterator past the inserted element.
      @param element the element to add
   */
   public void add(Object element)
   {
      if (position == null)
      {
         addFirst(element);
         position = first;
      }
      else
      {
         Node newNode = new Node();
         newNode.data = element;
         newNode.next = position.next;
         position.next = newNode;
         position = newNode;
      }
      previous = position;
   }

   /**
      Removes the last traversed element. This method may
      only be called after a call to the next() method.
   */
   public void remove()
   {
      if (previous == position)
         throw new IllegalStateException();

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

```
    }

    /**
        Sets the last traversed element to a different value.
        @param element the element to set
    */
    public void set(Object element)
    {
        if (position == null)
            throw new NoSuchElementException();
        position.data = element;
    }
```

680

681

```

        private Node position;
        private Node previous;
    }
}
```

ch15/impllist/ListIterator.java

```
/**
    A list iterator allows access to a position in a linked list.
    This interface contains a subset of the methods of the
    standard java.util.ListIterator interface. The methods for
    backward traversal are not included.
*/
public interface ListIterator
{
    /**
        Moves the iterator past the next element.
        @return the traversed element
    */
    Object next();

    /**
        Tests if there is an element after the iterator position.
        @return true if there is an element after the iterator
```

```
         position
      */
      boolean hasNext();

      /**
         Adds an element before the iterator position
         and moves the iterator past the inserted element.
         @param element the element to add
      */
      void add (Object element);

      /**
         Removes the last traversed element. This method may
         only be called after a call to the next() method.
      */
      void remove();

      /**
         Sets the last traversed element to a different
         value.
         @param element the element to set
      */
      void set(Object element);
   }
```

SELF CHECK

3. Trace through the addFirst method when adding an element to an empty list.

4. Conceptually, an iterator points between elements (see Figure 3). Does the position reference point to the element to the left or to the element to the right?

5. Why does the add method have two separate cases?

Advanced Topic 15.2: Static Inner Classes

You first saw the use of inner classes for event handlers. Inner classes are useful in that context, because their methods have the privilege of accessing private data members 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, the `Node` inner class has no need to access the outer class. In fact, it has no methods. Thus, there is no need to store a reference to the outer list class with each `Node` object. To suppress the outer-class reference, you can declare the inner class as `static`:

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

The purpose of the keyword `static` in this context is to indicate that the inner-class objects do not depend on the outer-class objects that generate them. In particular, the methods of a static inner class cannot access the outer-class instance variables. Declaring the inner class `static` is efficient, because its objects do not store an outer-class reference.

However, the `LinkedListIterator` class cannot be a static inner class. It frequently references the `first` element of the enclosing `LinkedList`.

15.3 Abstract and Concrete Data Types

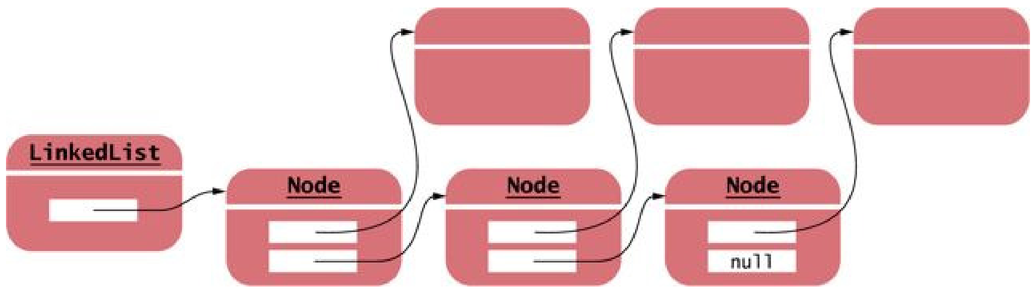
There are two ways of looking at a linked list. One way is to think of the concrete implementation of such a list as a sequence of node objects with links between them (see Figure 8).

On the other hand, you can think of the *abstract* concept of the linked list. In the abstract, a linked list is an ordered sequence of data items that can be traversed with an iterator (see Figure 9).

An abstract data type defines the fundamental operations on the data but does not specify an implementation.

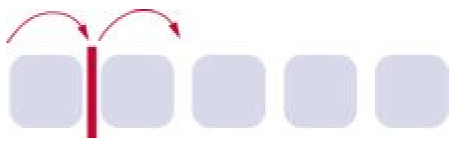
Similarly, there are two ways of looking at an array list. Of course, an array list has a concrete implementation: a partially filled array of object references (see Figure 10). 682
But you don't usually think about the concrete implementation when using an array 683
list. You take the abstract point of view. An array list is an ordered sequence of data items, each of which can be accessed by an integer index (see Figure 11).

Figure 8



A Concrete View of a Linked List

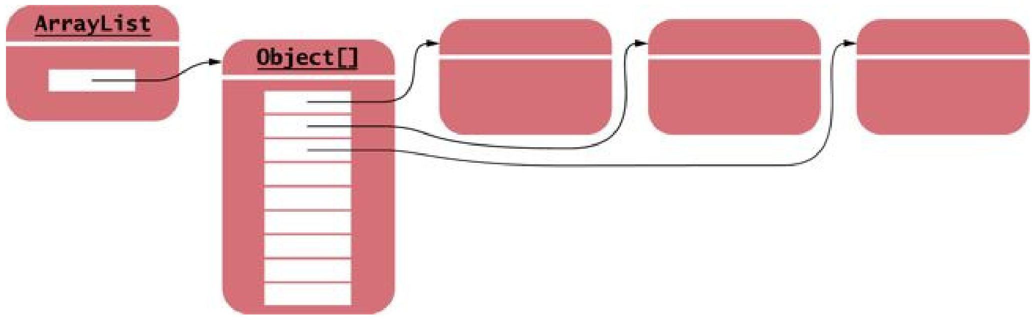
Figure 9



An Abstract View of a Linked List

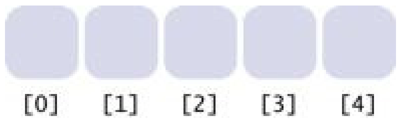
The concrete implementations of a linked list and an array list are quite different. The abstractions, on the other hand, seem to be similar at first glance. To see the difference, consider the public interfaces stripped down to their minimal essentials.

Figure 10



A Concrete View of an Array List

Figure 11



An Abstract View of an Array List

An array list allows *random access* to all elements. You specify an integer index, and you can get or set the corresponding element.

```
public class ArrayList
{
   public Object get(int index) { . . . }
   public void set(int index, Object element) { . .
. }
   . . .
}
```

With a linked list, on the other hand, element access is a bit more complex. A linked list allows sequential access. You need to ask the linked list for an iterator. Using that iterator, you can easily traverse the list elements one at a time. But if you want to go to a particular element, say the 100th one, you first have to skip all elements before it.

```
public class LinkedList
{
   public ListIterator listIterator() { . . . }
```

```
      . . .
   }
   public interface ListIterator
   {
      Object next();
      boolean hasNext();
      void add(Object element);
      void remove();
      void set(Object element);
      . . .
   }
```

Here we show only the *fundamental* operations on array lists and linked lists. Other operations can be composed from these fundamental operations. For example, you can add or remove an element in an array list by moving all elements beyond the insertion or removal index, calling `get` and `set` multiple times.

Of course, the `ArrayList` class has methods to add and remove elements in the middle, even if they are slow. Conversely, the `LinkedList` class has `get` and `set` methods that let you access any element in the linked list, albeit very inefficiently, by performing repeated sequential accesses.

In fact, the term `ArrayList` signifies that its implementors wanted to combine the interfaces of an array and a list. Somewhat confusingly, both the `ArrayList` and the `LinkedList` class implement an interface called `List` that defines operations both for random access and for sequential access.

That terminology is not in common use outside the Java library. Instead, let us adopt a more traditional terminology. We will call the abstract types *array* and *list.* The Java library provides concrete implementations `ArrayList` and `LinkedList` for these abstract types. Other concrete implementations are possible in other libraries. In fact, Java arrays are another implementation of the abstract array type.

To understand an abstract data type completely, you need to know not just its fundamental operations but also their relative efficiency.

Table 1 Efficiency of Operations for Arrays and Lists

Operation	Array	List
Random access	$O(1)$	$O(n)$
Linear traversal step	$O(1)$	$O(1)$
Add/remove an element	$O(n)$	$O(1)$

In a linked list, an element can be added or removed in constant time (assuming that the iterator is already in the right position). A fixed number of node references need to be modified to add or remove a node, regardless of the size of the 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 $O(1)$. Random access in an array list also takes $O(1)$ time.

An abstract list is an ordered sequence of items that can be traversed sequentially and that allows for insertion and removal of elements at any position.

Adding or removing an arbitrary element in an array takes $O(n)$ time, where n is the size of the array list, because on average $n/2$ elements need to be moved. Random access in a linked list takes $O(n)$ time because on average $n/2$ elements need to be skipped.

An abstract array is an ordered sequence of items with random access via an integer index.

Table 1 shows this information for arrays and lists.

Why consider abstract types at all? If you implement a particular algorithm, you can tell what operations you need to carry out on the data structures that your algorithm manipulates. You can then determine the abstract type that supports those operations efficiently, without being distracted by implementation details.

For example, suppose you have a sorted collection of items and you want to locate items using the binary search algorithm (see Section 14.7). That algorithm makes a random access to the middle of the collection, followed by other random accesses. Thus, fast random access is essential for the algorithm to work correctly. Once you know that an array supports fast random access and a linked list does not, you then look for concrete implementations of the abstract array type. You won't be fooled into using a `LinkedList`, even though the `LinkedList` class actually provides `get` and `set` methods.

In the next section, you will see additional examples of abstract data types.

SELF CHECK

6. What is the advantage of viewing a type abstractly?

7. How would you sketch an abstract view of a doubly linked list? A concrete view?

8. How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

15.4 Stacks and Queues

In this section we will consider two common abstract data types that allow insertion and removal of items at the ends only, not in the middle. A *stack* lets you insert and remove elements at only one end, traditionally called the *top* of the stack. To visualize a stack, think of a stack of books (see Figure 12).

A stack is a collection of items with “last in first out” retrieval.

New items can be added to the top of the stack. Items are removed at the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last in, first out* or *LIFO* order. For example, if you add items `A`, `B`, and `C` and then remove them, you obtain `C`, `B`, and `A`. Traditionally, the addition and removal operations are called `push` and `pop`.

A queue is a collection of items with “first in first out” retrieval.

A *queue* is similar to a stack, except that you add items to one end of the queue (the *tail)* and remove them from the other end of the queue (the *head)*. To visualize a queue, simply think of people lining up (see Figure 13). People join the tail of the queue and wait until they have reached the head of the queue. Queues store items in a *first in, first out* or *FIFO* fashion. Items are removed in the same order in which they have been added.

There are many uses of queues and stacks in computer science. The Java graphical user interface system keeps an event queue of all events, such as mouse and keyboard events. The events are inserted into the queue whenever the operating system notifies the application of the event. Another thread of control removes them from the queue and passes them to the appropriate event listeners. Another example 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 all bytes that need to be sent to the printer into a file and inserts that file into 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.

Figure 12

A Stack of Books

Figure 13

A Queue

Stacks are used when a “last in, first out” rule is required. For example, consider an algorithm that attempts to find a path through a maze. When the algorithm encounters an intersection, it pushes the location on the stack, and then it explores the first branch. If that branch is a dead end, it returns to the location at the top of the stack. If all branches are dead ends, it pops the location off the stack, revealing a previously encountered intersection. Another important example is the *run-time stack* that a processor or virtual machine keeps to organize the variables of nested methods. Whenever a new method is called, its parameters and local variables are pushed onto a stack. When the method exits, they are popped off again. This stack makes recursive method calls possible.

There is a `Stack` class in the Java library that implements the abstract stack type and the `push` and `pop` operations. The following sample code shows how to use that class.

```
Stack<String> s = new Stack<String>();
```

```
s.push("A");
s.push("B");
s.push("C");
// The following loop prints C, B, and A
while (s.size() > 0)
   System.out.println(s.pop());
```

The `Stack` class in the Java library uses an array to implement a stack. Exercise P15.11 shows how to use a linked list instead.

The implementations of a queue in the standard library are designed for use with multithreaded programs. However, it is simple to implement a basic queue yourself:

```
public class LinkedListQueue
{
   /**
      Constructs an empty queue that uses a linked list.
   */
   public LinkedListQueue()
   {
      list = new LinkedList();
   }
   /**
      Adds an element to the tail of the queue.
      @param element the element to add
   */
   public void add(Object element)
   {
      list.addLast(element);
   }
   /**
      Removes an element from the head of the queue.
      @return the removed element
   */
   public Object remove()
   {
      return list.removeFirst();
   }
   /**
      Gets the number of elements in the queue.
      @return the size
   */
   int size()
```

```
        {
           return list.size();
        }
        private LinkedList list;
     }
```

You would definitely not want to use an `ArrayList` to implement a queue. Removing the first element of an array list is inefficient—all other elements must be moved towards the beginning. However, Exercise P15.12 shows you how to implement a queue efficiently as a “circular” array, in which all elements stay at the position at which they were inserted, but the index values that denote the head and tail of the queue change when elements are added and removed.

In this chapter, you have seen the two most fundamental abstract data types, arrays and lists, and their concrete implementations. You also learned about the stack and queue types. In the next chapter, you will see additional data types that require more sophisticated implementation techniques. 688

689

Self Check

9. Draw a sketch of the abstract queue type, similar to Figures 9 and 11.

10. Why wouldn't you want to use a stack to manage print jobs?

Random Fact 15.1: Standardization

You encounter the benefits of standardization every day. When you buy a light bulb, you can be assured that it fits the socket without having to measure the socket at home and the light bulb in the store. In fact, you may have experienced how painful the lack of standards can be if you have ever purchased a flashlight with nonstandard bulbs. Replacement bulbs for such a flashlight can be difficult and expensive to obtain.

Programmers have a similar desire for standardization. Consider the important goal of platform independence for Java programs. After you compile a Java program into class files, you can execute the class files on any computer that has a Java virtual machine. For this to work, the behavior of the virtual machine has to be strictly defined. If virtual machines don't all behave exactly the same way, then the slogan of “write once, run anywhere” turns into “write once, debug everywhere". In order for multiple implementors to create compatible virtual machines, the virtual machine needed to be *standardized*. That is, someone needed to create a definition of the virtual machine and its expected behavior.

Who creates standards? Some of the most successful standards have been created by volunteer groups such as the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). You can find the Requests for Comment (RFC) that standardize many of the Internet protocols at the IETF site, http://www.ietf.org/rfc.html. For example, RFC 822 standardizes the format of e-mail, and RFC 2616 defines the Hypertext Transmission Protocol (HTTP) that is used to serve web pages to browsers. The W3C standardizes the Hypertext Markup Language (HTML), the format for web pages—see http://www.w3c.org. These standards have been instrumental in the creation of the World Wide Web as an open platform that is not controlled by any one company.

Many programming languages, such as C++ and Scheme, have been standardized by independent standards organizations, such as the American National Standards Institute (ANSI) and the International Organization for Standardization—called ISO for short (not an acronym; see http://www.iso.ch/iso/en/aboutiso/introduction/whatisISO.html). ANSI and ISO are associations of industry professionals who develop standards for everything from car tires and credit card shapes to programming languages.

The process of standardizing the C++ language turned out to be very painstaking and time-consuming, and the standards organization followed a rigorous process to ensure fairness and to avoid being influenced by companies with vested interests.

When a company invents a new technology, it has an interest in its invention becoming a standard, so that other vendors produce tools that work with the invention and thus increase its likelihood of success. On the other hand, by handing over the invention to a standards committee, especially one that insists on a fair process, the company may lose control over the standard. For that reason, Sun Microsystems, the inventor of Java, never agreed to have a third-party organization standardize the Java language. They run their own standardization process, involving other companies but refusing to relinquish control. Another unfortunate but common tactic is to create a weak standard. For example, Netscape 689
and Microsoft chose the European Computer Manufacturers Association (ECMA) 690

to standardize the JavaScript language (see Random Fact 10.1). ECMA was willing to settle for something less than truly useful, standardizing the behavior of the core language and just a few of its libraries. Because most useful JavaScript programs need to use more libraries than those defined in the standard, programmers still go through a lot of tedious trial and error to write JavaScript code that runs identically on different browsers.

Often, competing standards are developed by different coalitions of vendors. For example, at the time of this writing, hardware vendors are in disagreement whether to use the HD DVD or Blu-Ray standard for high-density video disks. As Grace Hopper, the famous computer science pioneer, observed: “The great thing about standards is that there are so many to choose from”.

Of course, many important pieces of technology aren't standardized at all. Consider the Windows operating system. Although Windows is often called a de-facto standard, it really is no standard at all. Nobody has ever attempted to define formally what the Windows operating system should do. The behavior changes at the whim of its vendor. That suits Microsoft just fine, because it makes it impossible for a third party to create its own version of Windows.

As a computer professional, there will be many times in your career when you need to make a decision whether to support a particular standard. Consider a simple example. In this chapter, we use the `LinkedList` class from the standard Java library. However, many computer scientists dislike this class because the interface muddies the distinction between abstract lists and arrays, and the iterators are clumsy to use. Should you use the `LinkedList` class in your own code, or should you implement a better list? If you do the former, you have to deal with a design that is less than optimal. If you do the latter, other programmers may have a hard time understanding your code because they aren't familiar with your list class.

CHAPTER SUMMARY

1. A linked list consists of a number of nodes, each of which has a reference to the next node.

2. Adding and removing elements in the middle of a linked list is efficient.

3. Visiting the elements of a linked list in sequential order is efficient, but random access is not.
4. You use a list iterator to access elements inside a linked list.
5. Implementing operations that modify a linked list is challenging—you need to make sure that you update all node references correctly.
6. An abstract data type defines the fundamental operations on the data but does not specify an implementation.
7. An abstract list is an ordered sequence of items that can be traversed sequentially and that allows for insertion and removal of elements at any position.

8. An abstract array is an ordered sequence of items with random access via an integer index.
9. A stack is a collection of items with "last in first out" retrieval.
10. A queue is a collection of items with "first in first out" retrieval.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.util.Collection<E>
   add
   contains
   iterator
   remove
   size
java. util.Iterator<E>
   hasNext
   next
   remove
java.util.LinkedList<E>
   addfirst
   addLast
   getfirst
   getLast
   removefirst
   removeLast
java. util.List<E>
   listIterator
java. util.ListIterator<E>
   add
   hasPrevious
   previous
   set
```

REVIEW EXERCISES

★ **Exercise R15.1.** Explain what the following code prints. Draw pictures of the linked list after each step. Just draw the forward links, as in Figure 1.

```
LinkedList<String> staff = new
LinkedList<String>();
staff.addfirst("Harry");
staff .addfirst("Dick");
staff.addfirst("Tom");
System.out.println(staff. removefirst());
System.out.println(staff.removefirst());
System.out.println(staff.removefirst());
```

★ **Exercise R15.2.** Explain what the following code prints. Draw pictures of the linked list after each step. Just draw the forward links, as in Figure 1.

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

691

692

★ **Exercise R15.3.** Explain what the following code prints. Draw pictures of the linked list after each step. Just draw the forward links, as in Figure 1.

```
LinkedList<String> staff = new
LinkedList<String>();
staff.addfirst("Harry");
staff.addLast("Dick");
staff.addfirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removefirst());
System.out.println(staff.removeLast());
```

★ **Exercise R15.4.** Explain what the following code prints. Draw pictures 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("Dick");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next() .equals("Tom"))
   iterator.remove();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

★ **Exercise R15.5.** Explain what the following code prints. Draw pictures 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("Dick");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

★★ **Exercise R15.6.** The linked list class in the Java library supports operations `addLast` and `removeLast`. To carry out these operations efficiently, the `LinkedList` class has an added reference `last` to the last node in the linked list. Draw a “before/after” diagram of the changes of the links in a linked list under the `addLast` and `removeLast` methods.

★★ **Exercise R15.7.** The linked list class in the Java library supports bidirectional iterators. To go backward efficiently, each `Node` has an added reference, `previous`, to the predecessor node in the linked list. Draw a “before/after” diagram of the changes of the links in a linked list under the `addFirst` and `removeFirst` methods that shows how the `previous` links need to be updated.

★★ **Exercise R15.8.** What advantages do lists have over arrays? What disadvantages do they have?

★★ **Exercise R15.9.** Suppose you needed to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred lookups against the collection every day. Would you use an array or a list to store the information?

★★ **Exercise R15.10.** Suppose you needed to keep a collection of appointments. Would you use a list or an array of `Appointment` objects?

★ **Exercise R15.11.** Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?

★ **Exercise R15.12.** Suppose the strings `"A"` … `"Z"` are pushed onto a stack. Then they are popped off the stack and pushed onto a second stack. Finally, they are all popped off the second stack and printed. In which order are the strings printed?

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★★ **Exercise P15.1.** Using only the public interface of the linked list class, write a method

```
public static void downsize(LinkedList<String>
staff)
```

that removes every other employee from a linked list.

★★ **Exercise P15.2.** Using only the public interface of the linked list class, write a method

```
public static void reverse(LinkedList<String>
staff)
```

that reverses the entries in a linked list.

★★★ **Exercise P15.3.** Add a method `reverse` to our implementation of the `LinkedList` class that reverses the links in a list. Implement this method by directly rerouting the links, not by using an iterator.

★ **Exercise P15.4.** 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.

★ **Exercise P15.5.** Add a `currentSize` field to our implementation of the `LinkedList class`. Modify the `add` and `remove` methods of both 693 the linked list and the list iterator to update the `currentSize` field so 694 that it always contains the correct size. Change the `size` method of the preceding exercise so that it simply returns the value of this instance variable.

★★ **Exercise P15.6.** The linked list class of the standard library has an add method that allows efficient insertion at the end of the list. Implement this method for the `LinkedList` class in Section 15.2. Add an instance field to the linked list class that points to the last node in the list. Make sure the other mutator methods update that field.

★★★ **Exercise P15.7.** Repeat Exercise P15.6, but use a different implementation strategy. Remove the reference to the first node in the `LinkedList` class, and make the `next` reference of the last node point to the first node, so that all nodes form a cycle. Such an implementation is called a *circular linked list.*

★★★ **Exercise P15.8.** Reimplement the `LinkedList` class of Section 15.2 so that the `Node` and `LinkedListIterator` classes are not inner classes.

★★★ **Exercise P15.9.** Add a `previous` field to the `Node` class in Section 15.2, and supply `previous` and `hasPrevious` methods in the iterator.

★★ **Exercise P15.10.** The standard Java library implements a `Stack` class, but in this exercise you are asked to provide your own implementation. Do not implement type parameters. Use an `Object[]` array to hold the stack elements. When the array fills up, allocate an array of twice the size and copy the values to the larger array.

★ **Exercise P15.11.** Implement a `Stack` class by using a linked list to store the elements. Do not implement type parameters.

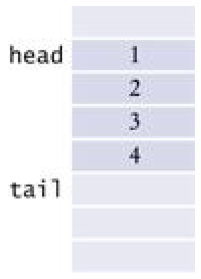
★★ **Exercise P15.12.** Implement a queue as a *circular array* as follows: Use two index variables `head` and `tail` that contain the index of the next element to be removed and the next element to be added. After an element is removed or added, the index is incremented (see Figure 14).

After a while, the `tail` element will reach the top of the array. Then it "wraps around" and starts again at 0—see Figure 15. For that reason, the array is called "circular".

```
public class CircularArrayQueue
{
   public CircularArrayQueue(int capacity) {.
. .}
   public void add(Object x) {. . .}
   public Object remove() {. . .}
   public int size() {. . .}
   private int head;
   private int tail;
   private int theSize;
   private Object[] elements;
}
```

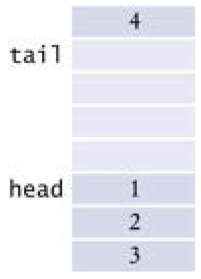
This implementation supplies a *bounded* queue—it can eventually fill up. See the next exercise on how to remove that limitation.

Figure 14



Adding and Removing Queue Elements

Figure 15



A Queue That Wraps Around the End of the Array

★★★ **Exercise P15.13.** The queue in Exercise P15.12 can fill up if more elements are added than the array can hold. Improve the implementation as follows. When the array fills up, allocate a larger array, copy the values to the larger array, and assign it to the `elements` instance variable. *Hint*: You can't just copy the elements into the same position of the new array. Move the head element to position `0` instead.

★★ **Exercise P15.14.** Modify the insertion sort algorithm of Advanced Topic 14.1 to sort a linked list.

★★ **Exercise P15.15.** Modify the `Invoice` class of Chapter 12 so that it implements the `Iterable<LineItem>` interface. Then demonstrate how an `Invoice` object can be used in a "for each" loop.

★★G **Exercise P15.16.** Write a program to display a linked list graphically. Draw each element of the list as a box, and indicate the links with line segments. Draw an iterator as in Figure 3. Supply buttons to move the iterator and to add and remove elements.

Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 15.1.** Implement a class `Polynomial` that describes a polynomial such as

$$p(x) = 5x^{10} + 9x^{7} - x - 10$$

Store a polynomial as a linked list of terms. A term contains the coefficient and the power of *x*. For example, you would store *p*(*x*) as

$$(5,\ 10),\ (9,\ 7),\ (-1,\ 1),\ (10,\ 0)$$

Supply methods to add, multiply, and print polynomials, and to compute the derivative of a polynomial.

695

696

★★★ **Project 15.2.** Make the list implementation of this chapter as powerful as the implementation of the Java library. (Do not implement type parameters, though.)

- Provide bidirectional iteration.
- Make `Node` a static inner class.
- Implement the standard `List` and `ListIterator` interfaces and provide the missing methods. *(Tip*: You may find it easier to extend `AbstractList` instead of implementing all `List` methods from scratch.)

★★★ **Project 15.3.** Implement the following algorithm for the evaluation of arithmetic expressions.

Each operator has a *precedence*. The `+` and `−` operators have the lowest precedence, `*` and `/` have a higher (and equal) precedence, and ∧ (which denotes "raising to a power" in this exercise) has the highest. For example,

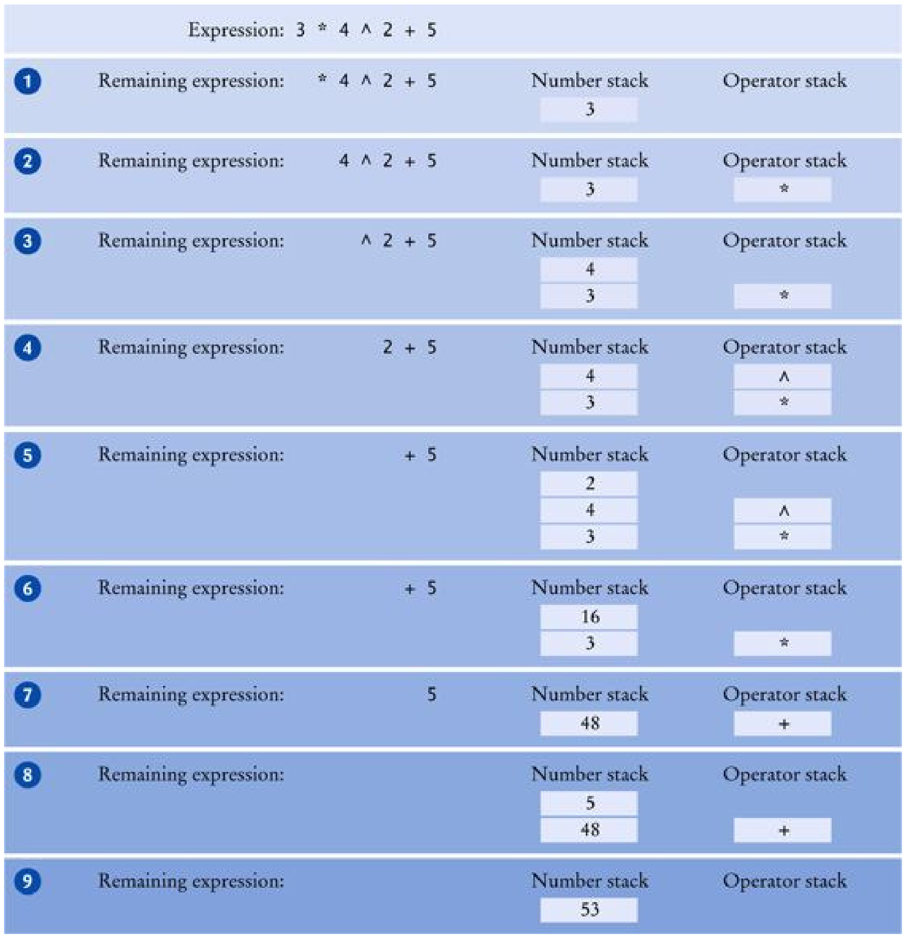
```
3 * 4 ^ 2 + 5
```

should mean the same as

```
(3 * (4 ^ 2)) + 5
```

with a value of 53.

In your algorithm, use two stacks. One stack holds numbers, the other holds operators. When you encounter a number, push it on the number stack. When you encounter an operator, push it on the operator stack if it has higher precedence than the operator on the top of the stack. Otherwise, pop an operator off the operator stack, pop two numbers off the number stack, and push the result of the computation on the number stack. Repeat until the top of the operator stack has lower precedence. At the end of the expression, clear the stack in the same way. For example, here is how the expression 3 * 4 ∧ 2 + 5 is evaluated:



Expression: 3 * 4 ^ 2 + 5
1 Remaining expression: * 4 ^ 2 + 5 Number stack 3 Operator stack
2 Remaining expression: 4 ^ 2 + 5 Number stack 3 Operator stack *
3 Remaining expression: ^ 2 + 5 Number stack 4 3 Operator stack *
4 Remaining expression: 2 + 5 Number stack 4 3 Operator stack ^ *
5 Remaining expression: + 5 Number stack 2 4 3 Operator stack ^ *
6 Remaining expression: + 5 Number stack 16 3 Operator stack *
7 Remaining expression: 5 Number stack 48 Operator stack +
8 Remaining expression: Number stack 5 48 Operator stack +
9 Remaining expression: Number stack 53 Operator stack

You should enhance this algorithm to deal with parentheses. Also, make sure that subtractions and divisions are carried out in the correct order. For example, `12 - 5 - 3` should yield 4.

ANSWERS TO SELF-CHECK QUESTIONS

1. Yes, for two reasons. You need to store the node references, and each node is a separate object. (There is a fixed overhead to store each object in the virtual machine.)

2. An integer index can be used to access any array location.

3. When the list is empty, `first` is `null`. A new `Node` is allocated. It's `data` field is set to the newly inserted object. It's `next` field is set to

null because first is null. The first field is set to the new node. The result is a linked list of length 1.

4. It points to the element to the left. You can see that by tracing out the first call to next. It leaves position to point to the first node.

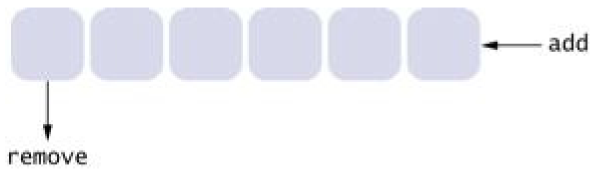
5. If position is null, we must be at the head of the list, and inserting an element requires updating the first reference. If we are in the middle of the list, the first reference should not be changed.

6. You can focus on the essential characteristics of the data type without being distracted by implementation details.

7. The abstract view would be like Figure 9, but with arrows in both directions. The concrete view would be like Figure 8, but with references to the previous node added to each node.

8. To locate the midde element takes *n* / 2 steps. To locate the middle of the sub-interval to the left or right takes another *n* / 4 steps. The next lookup takes *n* / 8 steps. Thus, we expect almost *n* steps to locate an element. At this point, you are better off just making a linear search that, on average, takes *n* / 2 steps.

9.



10. Stacks use a "last in, first out" discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.