[CHAPTER] 6

Linear Data Structures

6.1 A Taxonomy of Data Structures

This chapter begins our study of data structures. We will show, as mentioned in Chapter 5, how the proper choice of data structure for a given problem can have a profound effect on the efficiency of the final solution.

A **data type** is a collection of values along with a set of operations defined on these values. For example, the Integer data type is composed of the signed and unsigned whole numbers, up to some maximum value, along with the operations defined on these numbers, such as `+`, `-`, `*`, `/`, and `abs()`. In a **simple data type**, also called a **primitive data type**, the elements that belong to the data type cannot be decomposed into simpler and more basic structures. An integer is a simple data type because its elements, such as the values 5 and -99, cannot be further decomposed.[1]

In a **composite data type**, also called a **data structure**, the elements *can* be decomposed into either primitive types or other composite data types. Essentially, a data structure is a *collection* of elements rather than a single element. An example of a composite data type is the array. The following one-dimensional array *X*:

X:	21	-4	302

is composed of three primitive integer values. The 3×3, two-dimensional array *Y*:

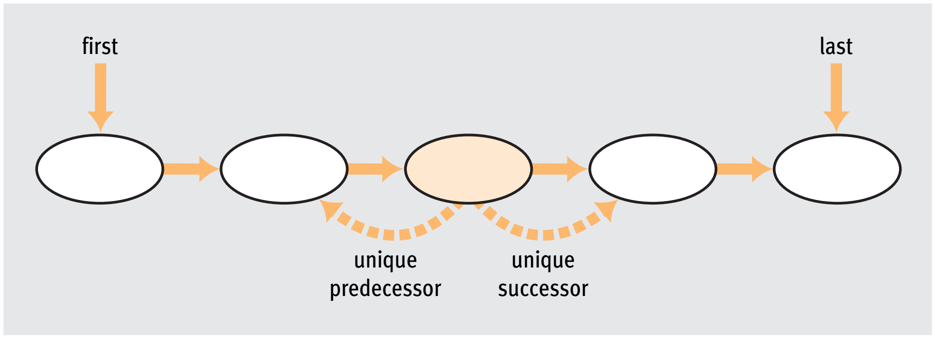
Y:	21	-4	302
	8	90	-1
	0	12	13

can be initially subdivided into three one-dimensional arrays (the rows), and each of these one-dimensional arrays can be further subdivided into three integers.

The following three chapters will introduce many different data structures—for example, lists, queues, binary trees, hash tables, weighted graphs—all with different characteristics and behaviors. However, these structures should not be viewed as distinct and unrelated topics. On the contrary, there is a *taxonomy*, or classification scheme, that allows us to impose a logical structure on this huge and complex topic. It is based on categorizing the relationships between individual elements of a data structure into one of four groupings.

The first grouping, the **linear data structures**, have a (1:1) relationship between elements in the collection. That is, if the data structure is not empty, there is a **first** element and a **last** element. Every element except the first has a unique **predecessor**—the element that comes immediately before—and every element except the last has a unique **successor**, the element that comes immediately after. A model for all linear structures is diagrammed in Figure 6-1.

[1] At a lower level of abstraction, simple types *can* be further decomposed. For example, the integer 5 might be stored in memory as the 16-bit sequence 0000000000000101. Thus, 5 can be decomposed into 16 separate binary digits. However, at this level of abstraction, we view simple types as nondecomposable.

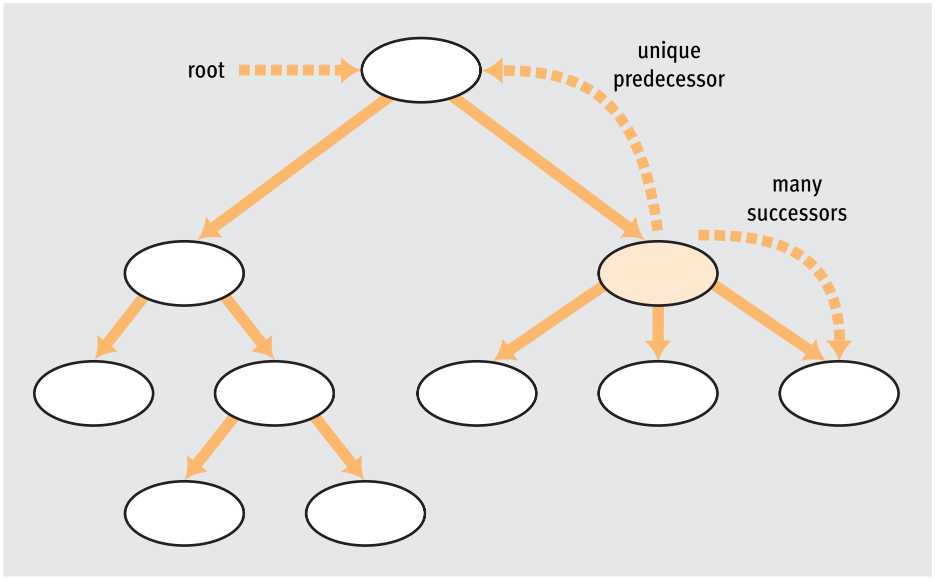


[FIGURE 6-1] General model of a linear data structure

A linear structure has a unique ordering of its elements, so there is only a single way to begin at the first element and follow each element to its successor until you arrive at the last element. You can clearly see this unique ordering in Figure 6-1. The process of sequencing through the nodes of a linear structure from beginning to end is called **iteration**.

There are many linear data structures, including lists, stacks, queues, and priority queues. They do not differ in their fundamental organization, which is shown in Figure 6-1. Instead, they differ in terms of restrictions they impose on exactly where we can perform such operations as insertions, deletions, and retrievals. We discuss linear data structures in detail in this chapter.

The second grouping is the **hierarchical data structures**, which have a (1:many) relationship between elements in the collection. That is, if the data structure is not empty, there is a unique element called the **root**, and zero, one, or more elements called **leaves**. Every element except the root has a unique predecessor, and every element except the leaves has one or more successors. Leaves have no successors. If an entry in a hierarchical data structure is neither the root nor a leaf, then it is an **internal node**. The model for all hierarchical structures is shown in Figure 6-2.



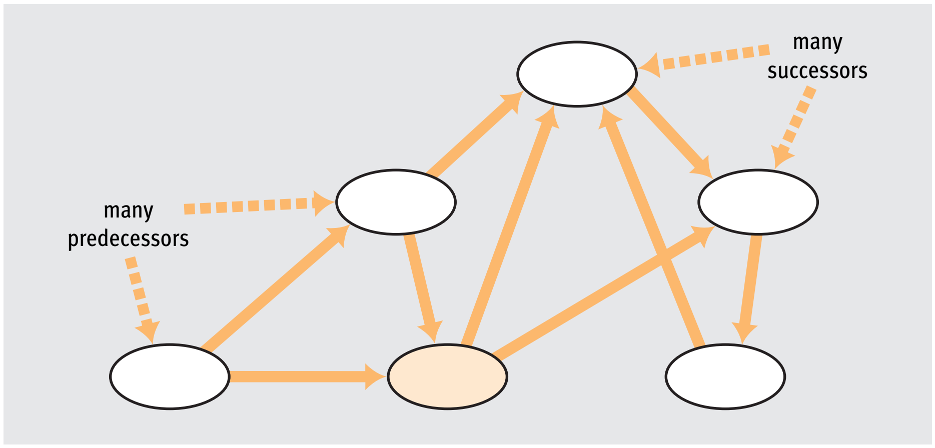
[FIGURE 6-2] General model of a hierarchical data structure

As you move down through the structure in Figure 6-2, you see that a node may point to many others; that is, it may have multiple successors. Thus, there is not a unique way to iterate through the elements of a hierarchical structure beginning at the root. As you move up through the structure, every node except the root is connected to a single element. That is, it has a unique predecessor.

The linear structure of Figure 6-1 is a special case of the hierarchical structure of Figure 6-2 in which the number of successors of a node is limited to one. In other words, the hierarchical classification includes linear structures; mathematicians do not view the two structures as distinct categories. However, in computer science it is convenient to treat these groups as distinct because they display quite different performance characteristics for certain types of problems.

Hierarchical structures are referred to as **trees**, and are an extremely important form of data representation. There are many types of tree structures, such as generalized trees, binary trees, and binary search trees, to name a few. However, they all have the same basic structure shown in Figure 6-2. The only differences are details, such as where in the tree we can make insertions, deletions, and retrievals, and whether there are limits on the maximum number of a node's successors. Hierarchical data structures are introduced in Chapter 7.

The third class of composite data type is the **graph**, and it represents the richest and most complex form of data representation. A graph has a (many:many) relationship between elements in a collection. That is, there are no restrictions on the number of predecessors or successors of any element. Informally, we say that in a graph structure, an element E can be related to an arbitrary number of other elements, including itself, and an arbitrary number of other elements can be related to E. The general model for a graph is diagrammed in Figure 6-3.



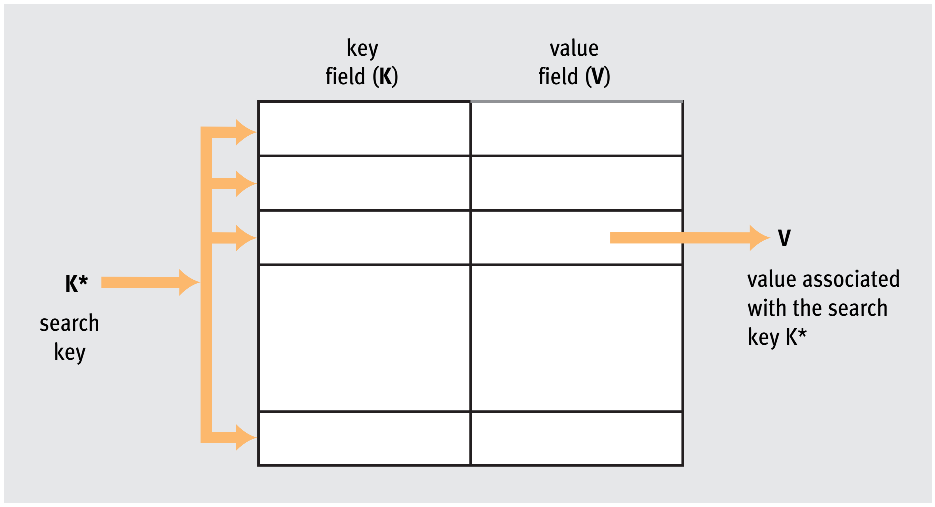
[FIGURE 6-3] General model of a graph data structure

It is easy to see from Figure 6-3 that graphs subsume both the linear and hierarchical groupings; from a mathematical point of view, it is only necessary to identify and study the graph classification. (The field of mathematics that studies the properties of these data structures is called graph theory.) However, as mentioned earlier, it is often useful to distinguish between these three composite structures because of their different uses and performance characteristics. We investigate the topic of graph structures in Chapter 8.

The fourth and final class of data structures is the **set**. In a set, there is no positional relationship between individual elements in a collection. There is no first element, no last element, no predecessor or successor, no root, and no leaves. Think of the elements of a set as having a (none:none) relationship with each other! Furthermore, duplicates are not allowed, so the set {1, 2, 2, 3, 3} should properly be written as {1, 2, 3}, which is identical to the sets {3, 2, 1} and {2, 1, 3}[2]. If we repositioned the elements in Figure 6-1, 6-2, or 6-3, we would end up with different linear, hierarchical, or graph structures. However, if we changed the location of an element in a set, we would end up with the identical set. The only relationship shared by elements of a set structure is *membership*—they either do or do not belong to the same collection. The position of a given element is irrelevant. (Sets are an extremely important structure in mathematics, and the field that investigates their formal properties is called set theory.)

[2] In computer science, a set structure that allows for duplicate entries is often called a **bag**. This chapter does not discuss bags.

Although sets are in widespread use, computer scientists more often use the closely related concept of a **table**, also called a **map**, to describe this type of position-independent data structure. In a table, the elements of the collection are usually expressed as pairs of values (K, V)—where K is a unique **key field** and V is a **value field** associated with this key. The general model of a table structure is diagrammed in Figure 6-4.



[FIGURE 6-4] General model of a table data structure

To access elements in the table, you provide a special key K* and determine if there is a (K, V) pair anywhere in the table in which the key field K matches the special key. If such a pair is found, then we return the value V associated with that key. Otherwise, the key K* is not a member of this table.

An example of this type of access would be a table containing (ID, exam score) pairs. We do not care exactly where any specific (ID, exam score) pair is located in the table; we only want to input a specific student ID number and retrieve the exam score associated with this ID, or find out that this ID number is not in the table. We will study tables and other forms of set structures in Chapter 8.

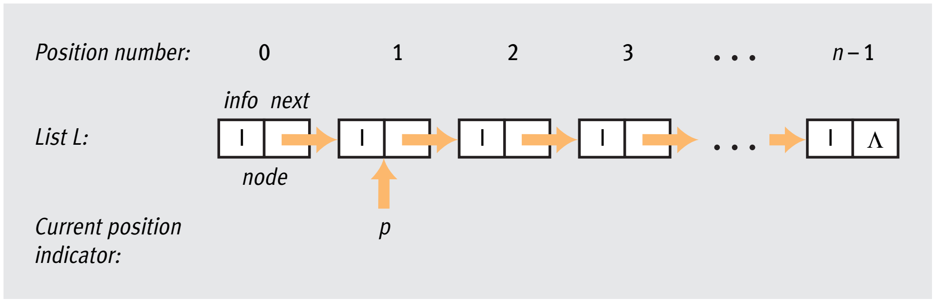
These are the four classes of composite data types, or data structures, that we will discuss. There are an enormous variety of data structures with very different characteristics, but they all fall into one of the four groupings presented in this section.

6.2 Lists

6.2.1 Introduction

We begin our discussion by examining the linear data structures; this section takes a look at the **list**, the most general and flexible of all linear structures. All other linear structures place restrictions on their usage, such as where you can make insertions, deletions, or retrievals. With a list there are no such restrictions, and retrievals, insertions, and deletions are allowed anywhere within the structure—beginning, middle, or end.

Formally, a list is an ordered collection of zero, one, or more information units called **nodes**. Each node contains two fields: an **information field**, also called the **`data` field**, which we abbreviate as I, and a **`next` field**, which represents an explicit way of identifying the unique successor of this node. (The type of the information field is not important and is left unspecified for now.) A model of an n-element list L is shown in Figure 6-5. The symbol Λ (the Greek lambda) is traditionally used to signify that there is no successor—that is, Λ is the last node in the list. It is the equivalent of the Java constant **null**.



[FIGURE 6-5] Conceptual model of a list structure

However, a word of caution is in order when you examine Figure 6-5. This is a *logical*, or *conceptual*, view of a list, not an implementation model. The fact that this figure shows connected nodes using "arrows" that point explicitly to their successor does not imply that the only way to implement a list is via pointers or reference variables. Although references can certainly be used to implement lists (and we provide many examples in later sections), it is not the only way. For example, Section 6.2.3.1 shows an array-based implementation of a list. The key idea is that each node must explicitly identify its successor, but we are free to select how to implement that identification.

6.2.2 Operations on Lists

Because lists are such general structures, we can implement an enormous number of operations on them. Therefore, when selecting methods to include in a list class, you often need to have additional information about how client classes will use your list. This information allows you to select the most useful and helpful operations. This section describes a fairly typical set of methods that operate on list objects; however, keep in mind that many others are possible.

When working with individual items in a list, we need to identify the specific node on which we are operating. For example, we must identify a particular node we want to remove; to insert a new node, we must specify between which two nodes it will go.

There are two ways to do this. The first is to use a **current position indicator**, sometimes termed a **cursor**. This variable references, or points to, one of the nodes in the list and identifies the exact position in the list where an operation should be performed. This cursor is part of the state information maintained for every list object. In Figure 6-5, the cursor p references the second node in the list, and any method that operates on an individual node will work on this element. For example, the call `L.remove()` could mean to delete the node in list L currently referenced by p. The call `L.getInfo()` would mean to return the information field of the node currently referenced by p. When initiating a list operation that uses the cursor, we must be sure that it points to the node we want to use. If not, we must reposition it before invoking the desired method.

The second way to implement list operations is to use the concept of **position numbers**, also called an **index**. Because every nonempty list has a first node and every node has a unique predecessor, there is a well-defined numbering scheme for nodes in a nonempty list L. Assign the first node in L the position number 0. Then, for every remaining node n in L, let position(n) $=$ 1 $+$ position(predecessor of n). This assigns the unique integer values 0, 1, 2, 3, … to the nodes, as shown in Figure 6-5. In a sense, this approach treats a list like a one-dimensional array, with the position numbers serving the role of subscript. (However, unlike an array, we cannot be guaranteed the ability to access individual list elements in O(1) time.)

If we use position numbers, our list methods must include a parameter that specifies which position number to use. For example, `L.remove(3)` might mean to delete the fourth node in list L. (This assumes that L contains at least four nodes. If not, the operation is undefined.) `L.add(I, 4)` might mean to insert a new node containing object I in the information field L, so that the position number of the new node is 4—that is, the fifth node. All nodes that come after this new node will have their position numbers increased by one.

Both techniques—cursors and position numbers—can be used to implement list methods, and we will show examples that use both approaches. The following examples all assume that L is a list, I is an object of the type stored in the information field of the node, p is the current position indicator of L, and *pos* is an index value in the range $0 \le pos <$ size of the list.

1 *List methods based on the concept of a current position indicator p:*

To illustrate the behavior of these cursor-based methods, assume that list L and position indicator p currently have the following values:

$$\begin{tikzcd} L: & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\ & & p \arrow[u] & & \end{tikzcd}$$

For each method we introduce, we describe the effect of the operation on the original list L shown above.

a *first()*—Reset p so it refers to the first node in L. If L is empty, then reset p so that it no longer points to any node in the list. When this happens, we say that p has been "moved off the list." The structure and content of L are unchanged. Executing the operation `L.first()` on the original list L shown previously produces the following:

$$\begin{tikzcd} L: & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\ & p \arrow[u] & & & \end{tikzcd}$$

B *last()*—Reset p so it refers to the last node in L. If L is empty, p is moved off the list. The structure and content of L are unchanged. Executing the operation `L.last()` on the original list L produces:

$$\begin{tikzcd} L: & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\ & & & & p \arrow[u] \end{tikzcd}$$

c *next()*—Reset *p* so it refers to the successor of the node currently pointed at by *p*. If *p* currently points to the last node in the list, then this operation moves *p* off the list. If *p* is currently off the list, then this operation has no effect. The structure and content of the list are unchanged. After executing `L.next()`, the original list will be in the following state:

$$\begin{tikzcd}
L: & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\
 & & & p \arrow[u] &
\end{tikzcd}$$

d *previous()*—Reset *p* so it refers to the predecessor of the node currently pointed at by *p*. If *p* points to the first node, then this operation moves *p* off the list. If *p* is currently off the list, then this operation has no effect. The structure and content of the list are unchanged. After executing `L.previous()`, the original list will be in the following state:

$$\begin{tikzcd}
L: & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\
 & p \arrow[u] & & &
\end{tikzcd}$$

e *remove()*—Delete the node referenced by the current position pointer *p*, and reset *p* so that it points to the successor of the node just deleted. If we delete the last node in the list, then *p* is moved off the list. If *p* is currently off the list, then this operation has no effect. The size of the list is reduced by 1. After performing the operation `L.remove()`, we will have:

$$\begin{tikzcd}
L: & 3 \arrow[r] & 24 \arrow[r] & 19 \\
 & & p \arrow[u] &
\end{tikzcd}$$

f *add(I)*—Create a new node containing the object *I* in its information field. Add this new node as the successor of the node currently referenced by *p*, and reset *p* to refer to the newly added node. The size of the list increases by 1. If *p* is currently off the list, then this operation has no effect. The call `L.add(5)` produces the following list:

$$\begin{tikzcd}
L: & 3 \arrow[r] & 7 \arrow[r] & 5 \arrow[r] & 24 \arrow[r] & 19 \\
& & & p \arrow[u] & &
\end{tikzcd}$$

g *addFirst(I)*—Create a new node containing object *I* in the information field. Add this new node as the first node in the list, regardless of the current value of *p*. The value of *p* is then reset to point to this new node, and the size of the list is increased by 1. The operation `L.addFirst(99)` produces the following:

$$\begin{tikzcd}
L: & 99 \arrow[r] & 3 \arrow[r] & 7 \arrow[r] & 24 \arrow[r] & 19 \\
& p \arrow[u] & & & &
\end{tikzcd}$$

h *get()*—This method returns the information field of the node referenced by *p*. This method does not change the structure of the list or the position of *p*. In the original diagram, if we perform the operation `L.get()`, the value returned by the function is 7, the contents of the information currently referenced by *p*. If *p* is currently off the list, then this operation has no effect.

i *set(I)*—This method changes the information field of the node referenced by *p* from its current value to the new value *I*. This method does not change the position of *p*. If *p* is currently off the list, then this operation has no effect. The call `L.set(35)` produces the following new list:

$$\begin{tikzcd}
L: & 3 \arrow[r] & 35 \arrow[r] & 24 \arrow[r] & 19 \\
& & p \arrow[u] & &
\end{tikzcd}$$

j *isOnList()*—This Boolean function returns true if the current position indicator *p* is referencing one of the nodes in the list. If the indicator has been moved off the list, then the method returns false. For example, the call `L.isOnList()` returns true because *p* is referencing the second element on the list.

2 *List methods based on the concept of a position number:*

To illustrate the behavior of our position number-based list methods, assume that this time we are operating on the following list *L*:

$$\begin{tikzcd}
\textit{position number:} & 0 & 1 & 2 & 3 & 4 \\
\textit{List } L: & 13 \arrow[r] & 1 \arrow[r] & 20 \arrow[r] & 7 \arrow[r] & 18 \\
 & & p \arrow[u] & & &
\end{tikzcd}$$

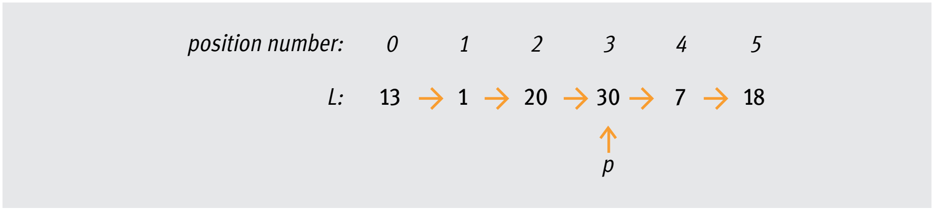
For each method introduced next, we describe its effect on the original list *L* shown above; the list contains both position numbers (0 .. 4) and a current position indicator *p*. In all cases, if the position number is outside the legal range, the operation has no effect.

a *size()*—This integer returns the total number of nodes contained in list *L*. The structure and content of the list are unchanged. In the above example, `L.size()` returns a 5. Thus, we can see that the position numbers in a non-empty list will range from 0 to `L.size()-1`.

b *remove(pos)*—This method deletes the node whose position number is *pos*. The current position indicator *p* is reset to refer to the successor of the node just removed. If we remove the last node, then *p* is moved off the list. The size of the list decreases by 1. Executing the method `L.remove(3)` produces:

$$\begin{tikzcd}
\textit{position number:} & 0 & 1 & 2 & 3 \\
L: & 13 \arrow[r] & 1 \arrow[r] & 20 \arrow[r] & 18 \\
 & & & & p \arrow[u]
\end{tikzcd}$$

c *add(I, pos)*—This method creates a new node containing the object *I* in the information field. The new node is then inserted into the list so that its position number is *pos*. The current position indicator *p* is reset to point to this new node, and the size of the list increases by 1. Executing `L.add(30, 3)` produces the following:



d *get(pos)*—This method returns the information field of the node whose position number is *pos*. The structure and content of the list as well as the current position pointer are unchanged. `L.get(4)`, when applied to the original list *L*, returns the value 18.

e *set(I, pos)*—This method changes the information field of the node whose position number is *pos* from its current value to the object *I*. The current position pointer *p* is unaffected. The call `L.set(-44, 0)` produces the following structure:

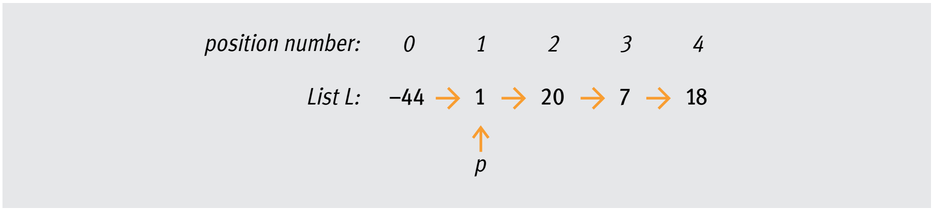


Figure 6-6 presents the specifications of a Java **interface** for a `List` data structure that includes the 15 operations just described. In the following section, we will develop two separate classes that implement the interface in Figure 6-6 using quite different approaches.

```
/**
 * This interface defines the operations that a list is expected to
 * provide. The operations it can support can be defined without
 * regard to the way in which the list is implemented.
 *
 * A list maintains an internal cursor, which refers to the current
 * element in the list.
 */
```

continued

```
public interface List<T> {

    /**
     * Move the cursor to the first element in the list.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *   If the list is empty the cursor is moved off the list.
     *     Otherwise, the cursor is set to the first element.
     *   The list's structure and content are unchanged.
     */
    public void first();

    /**
     * Move the cursor to the last element in the list.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *   If the list is empty the cursor is moved off the list.
     *     Otherwise, the cursor is set to the last element.
     *   The list's structure and content are unchanged.
     */
    public void last();

    /**
     * Move the cursor to the next element in the list.
     *
     * Preconditions:
     *   The cursor is on the list.
     *
     * Postconditions:
     *   If the cursor is at the end of the list, the cursor is moved
     *     off the list; otherwise, it is set to the next element.
     *   The list's structure and content are unchanged.
     */
    public void next();

    /**
     * Move the cursor to the previous element in the list.
     *
     * Preconditions:
     *   The cursor is on the list.
     * Postconditions:
     *   If the cursor is at the front of the list, the cursor is moved
     *     off the list; otherwise, it is moved to the previous element.
```

continued

```
     *   The list's structure and content are unchanged.
     */
    public void previous();

    /**
     * Remove the element the cursor is referring to.
     *
     * Preconditions:
     *   The cursor is on the list.
     *
     * Postconditions:
     *   The element the cursor is referring to is removed.
     *   The size of the list has decreased by 1.
     *   If the element that was removed was the last element in the
     *     list, the cursor is moved off the list.  Otherwise, the
     *     cursor is moved to the removed element's successor.
     */
    public void remove();

    /**
     * Remove the element at the specified position in the list.
     *
     * Preconditions:
     *   position >= 0 and position < size().
     *
     * Postconditions:
     *   The element at the specified position is removed.
     *   Elements at positions greater than the specified position
     *     (if any) are shifted to the left by one position.
     *   The size of the list is decreased by 1.
     *   If the element that was removed was the last element in the
     *     list, the cursor is moved off the list.  Otherwise the
     *     cursor is moved to the removed element's successor.
     *
     * @param position the position where the element will be placed.
     */
     public void remove( int position );

    /**
     * Add after the position indicated by the cursor.
     *
     * Preconditions:
     *   The list is empty, or the cursor is on the list.
     *
     * Postconditions:
     *   If the list is empty, the element becomes the only element
```

continued

```
     *     in the list.  Otherwise the element is added to the list
     *     as the successor of the node specified by the cursor.
     *   Elements at positions greater than the current position
     *     of the cursor (if any) are shifted to the right.
     *   The size of the list has increased by 1.
     *   The cursor refers to the element added to the list.
     *
     * @param element the element to be added to the list.
     */
    public void add( T element );

    /**
     * Add a new element at the specified position in the list.
     *
     * Preconditions:
     *   position >= 0 and position <= size().
     *
     * Postconditions:
     *   The element is added at the specified position in the list.
     *   Elements at positions greater than or equal to the specified
     *     position (if any) are shifted to the right by one position.
     *   The size of the list is increased by 1.
     *   The cursor refers to the element added to the list.
     *
     * @param position the position where the element will be placed.
     * @param element the element to be added to the list.
     */
     public void add( T element, int position );

    /**
     * Add a new element at the beginning of the list.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *     If the list is empty, the element becomes the only element
     *       in the list.  Otherwise the element is added to the list
     *       at position 0.
     *   All of the current elements in the list (if any) are
     *     shifted to the right.
     *   The size of the list has increased by 1.
     *   The cursor refers to the element added to the list.
     *
     * @param element the element to be added to the list.
     */
```

continued

```
    public void addFirst( T element );

   /**
    * Returns the element that the cursor is referring to.
    *
    * Preconditions:
    *   The cursor is on the list.
    * Postconditions:
    *   The list's structure, content, and cursor are unchanged.
    *
    * @return the element the cursor refers to.
    */
   public T get();

   /**
    * Returns the element stored in the specified position in the
    * list.
    *
    * Preconditions:
    *   position >= 0 and position < size().
    * Postconditions:
    *   The list's structure, content, and cursor are unchanged.
    *
    * @param position the location of the element to be retrieved.
    * @return the element at the given position
    */
   public T get( int position );

   /**
    * Sets the element stored at the location identified by the
    * cursor.
    *
    * Preconditions:
    *   The cursor is on the list.
    * Postconditions:
    *   The element identified by the cursor is changed to the
    *     specified value.
    *   The list's structure and cursor are unchanged.
    *
    * @param element the element to place in the list.
    */
   public void set( T element );

   /**
    * Sets the element stored in the specified position in the list.
    *
    * Preconditions:
```

continued

```
     *   position >= 0 and position < size().
     * Postconditions:
     *   The element at the specified position in the list is changed
     *     to the specified value.
     *   The list's structure and cursor are unchanged.
     *
     * @param position the location of the element to be changed.
     * @param element the element to place in the list.
     */
    public void set( T element, int  position );

    /**
     * Return the number of elements in the list.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The list's structure, content, and cursor are unchanged.
     *
     * @return the number of elements in the list
     */
    public int size();

    /**
     * Determine if the cursor is on the list.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The list's structure, content, and cursor are unchanged.
     *
     * @return true if the cursor is on the list, false otherwise
     */
    public boolean isOnList();

}
```

[FIGURE 6-6] List interface

The methods included in the `List` interface of Figure 6-6 are certainly not the only ones we could have selected. It is easy to think of many other possibilities: for example, *L*`.swap(pos1, pos2)`, which interchanges the two nodes in list *L* located at positions *pos1* and *pos2*; or *L*`.printList()`, which prints the contents of the information field of every node in L in order from first to last. Whether we should include these or any other methods depends on exactly how our lists will be used.

However, the interface in Figure 6-6 does include virtually all of the important list operations we need in the upcoming pages, and it allows us to investigate some interesting implementations of the list data structure.

6.2.3 Implementation of Lists

There are two quite different ways to implement the interface in Figure 6-6—**array-based** and **reference-based** methods. This section describes both techniques.

6.2.3.1 An Array-Based Implementation

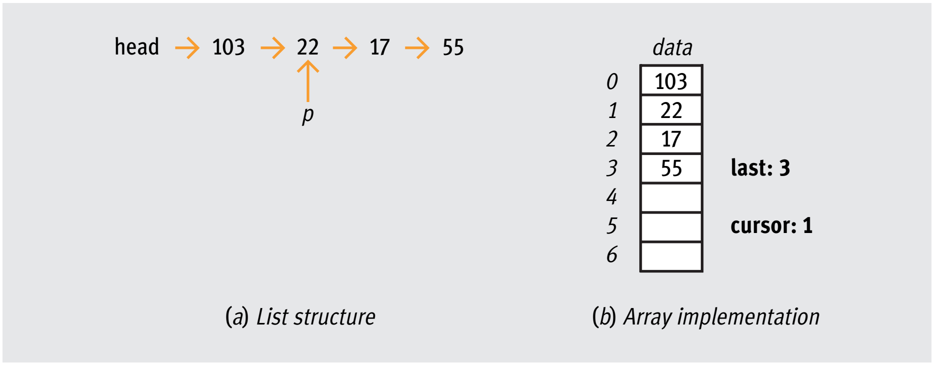
A simple way to implement a list structure is to use a one-dimensional array. The array is used to store the contents of the nodes' information fields in order of their position number in the list—in other words, the node at position number 0 has its information field stored in array position 0, the node at position number 1 has its information field stored in array position 1, and so forth, up to the last node in the list. There is no need to explicitly store the `next` field because the logical successor of a node can be determined using array subscripts—the successor node of the one in array position i is the one stored in array position $i+1$. Similarly, we do not need a head pointer in an array-based implementation, because the head of a list is always presumed to be located in position 0. In essence, an array-based implementation replaces the idea of a "logical sequence" of values, which exists in a list, with the concept of a "physical sequence" of values that exists in an array. The state information needed for an array representation of a list is shown in Figure 6-7.

```
private static final int OFF_LIST = -1;  // Cursor off list
private static final int SIZE = 20;      // Default array size

private T data[];                        // Elements in the list

private int last;                        // The end of the list
private int cursor;                      // Current node
```

[FIGURE 6-7] Declarations for the array representation of a list

The `data` array declared in Figure 6-7 is used to store the information field of each node in the list. The instance variable *cursor* represents the current position indicator `p` in Figure 6-5. The constant OFF_LIST, defined in Figure 6-7 as −1, is used to represent the fact that the cursor is off the list. We chose the value −1 because the indices of Java arrays begin at 0. A value of "off the list" will never be mistaken for an array subscript. Finally, the instance variable `last` is the index of the last item currently stored in the array.

Thus, the four-node list diagrammed in Figure 6-8(a) could be implemented in array form as shown in Figure 6-8(b), assuming that the value of SIZE is 7 and the information field is storing integer values.



[FIGURE 6-8] Array implementation of a list

In Figure 6-8(b), the four list values 103, 22, 17, and 55 are stored in the first four elements of the `data` array in exactly that order. The value 103 is the first element of the list because it is in location `data[0]`. The integer value 22 is its successor because it is in location `data[1]`. We know that 55 is the last element in the list because the instance variable `last` has the value 3, the location of 55 in the array. Finally, `cursor` is pointing to the value 22, the item in position 1.

Figure 6-9 shows the code for the class `ArrayList`, an array-based implementation of the `List` interface in Figure 6-6 that uses the declarations in Figure 6-7.

This code must address and solve one important issue related to our choice of an array-based implementation. A list is a data structure that can grow without bound, or at least until we run out of memory on our computer. However, an array is a fixed-size structure that cannot be enlarged once you create it. In the constructor for the `ArrayList` in Figure 6-7, we would see the following statement:

```
data = (T[])new Object[SIZE];
```

This statement creates a fixed-length `data` array with `SIZE` entries, where `SIZE` is a constant. If `SIZE` were 20, for example, we would be fine as long as the list had 20 or fewer nodes. However, if we tried to add a 21st node, we would get an "array index out of bounds" error.

We could choose to treat this as a fatal error and terminate execution, but this runs counter to the logical view of a list as something that can grow as large as we want. Another undesirable solution is to make the original array so large that it could never become full. However, we may have no idea of the maximum number of nodes, so we would need to make the array as big as it could ever possibly be—perhaps thousands of times larger than need be. Creating monstrously large arrays can waste a great deal of memory.

The approach used in `ArrayList` is a technique called **array resizing**. If the existing array ever becomes full, we create a new array that is twice the size of the current one and copy all existing entries from the full array into the new, larger structure. Because we must move n elements, this is an $O(n)$ linear-time operation. For large n, this could be quite time consuming. However, it does allow our array-based implementation to mimic the unbounded growth capabilities of a list without wasting an excessive amount of memory.

```
/**
 * The following is an array-based implementation of the List
 * interface. Javadoc comments for methods specified in the List
 * interface have been omitted.
 *
 * This code assumes that the preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class ArrayList<T> implements List<T> {
    private static final int OFF_LIST = -1;  // Cursor off list
    private static final int SIZE = 20;      // Default array size

    private T data[];                        // Elements in the list

    private int last;                        // The end of the list
    private int cursor;                      // Current node

    /**
     * Constructor for the list
     */
    public ArrayList() {
        // The cast is necessary because you cannot
        // create generic arrays in Java. This statement will
        // generate a compiler warning.
        data = (T[])new Object[ SIZE ];

        last = OFF_LIST;
        cursor = OFF_LIST;
    }
```

continued

```
public void first() {
    // If there are no elements, move the cursor off the list.
    if( last == OFF_LIST ){
        cursor = OFF_LIST;
    }
    else{
        //Since Java arrays start at index 0
        cursor = 0;
    }
}

public void last()  {
    // If there are no elements, move the cursor off the list.
    if( last == OFF_LIST ){
        cursor = OFF_LIST;
    }
    else{
        cursor = last;
    }
}

public void next() {
    // If the cursor is at the end of the list, move it off.
    if( cursor >= last ){
        cursor = OFF_LIST;
    }
    else{
        cursor = cursor + 1; // Move to the next element
    }
}

public void previous() {
    // If the cursor is at the beginning of the list, move it off.
    if( cursor == 0 ){
        cursor = OFF_LIST;
    }
    else{
        cursor = cursor - 1; // Move to the previous element
    }
}

 public void remove() {
    remove( cursor );
}

public void remove( int position ) {
    // Shift all elements greater than the specified element
    // (position) to the left by one, overwriting (thus removing)
```

continued

```
        // the element at the index given by position.
        if( position != OFF_LIST){
            for( int i = position; i < last; i++ ){
                data[ i ] = data[ i + 1 ];
            }

            // If this is the last element, move the cursor off list
            if( last == position ){
                cursor = OFF_LIST;
            }

            // Decrement the size of the list by decreasing last by 1
            last = last - 1;

            // NOTE: The cursor actually doesn't need to be changed;
            // since the elements have been shifted, it is not pointing
            // at the removed element's successor.
        }
    }

        public void add( T element ) {
        // Add the node to the list (as the successor to the cursor)
        cursor = cursor + 1;
        add( element, cursor );
        }

        public void add( T element, int position ) {
        // If the list is empty, make this the first element
        if( last == OFF_LIST ){
            data[ 0 ] = element;
            // Increment the size of the array to one.
            last = 0;
        }
        else{
            // Check whether adding the element will overflow the array
            if( last == SIZE - 1 ){
                // Array is full—double the size
                doubleArraySize();
            }

            // Shift all elements greater than this one down one
            for( int i = last; i >= position; i-- ){
                data[ i + 1 ] = data[ i ];
            }

            // Assign the element to the specified position
            data[ position ] = element;
            // Increase the size by incrementing last by one
```

continued

```
            last = last + 1;
            // Assign the cursor to the new element
            cursor = position;
        }
    }

    public void addFirst( T element ){
        // Call add with the position set to the beginning (0)
        add( element, 0 );
    }

    public T get() {
        return data[ cursor ];
    }

    public T get( int position ) {
        return data[ position ];
    }

    public void set( T element ) {
        data[ cursor ] = element;
    }

    public void set( T element, int position ) {
        data[ position ] = element;
    }

    public int size() {
        // Return last + 1 since last is zero-indexed
        return last + 1;
    }

    public boolean isOnList() {
        return cursor != OFF_LIST;
    }

    /**
     * Return a string representation of this list.
     *
     * @return a string representation of this list.
     */
    public String toString() {
        // Use a StringBuffer so we don't create a new
        // string each time we append
        StringBuffer retVal = new StringBuffer();

        retVal = retVal.append( "[ " );
```

continued

```
        // Step through the list and use toString on the elements to
        // determine their string representation
        for ( int cur = 0; cur <= last; cur++ ) {
            retVal = retVal.append( data[ cur ] + " " );
        }

        retVal = retVal.append( "]" );

        // Convert the StringBuffer to a string
        return retVal.toString();
    }

    /**
     * Takes the data array and doubles the size of it. A temporary
     * array is used while the elements are being copied to the new
     * array.
     */
    private void doubleArraySize(){
        // Double the size
        SIZE = SIZE * 2;

        T temp[] = (T[])new Object[ SIZE ];

        // Copy all of the elements to the new array
        for( int i = 0; i <= last; i++ ){
            temp[ i ] = data [ i ];
        }

        // Set the data variable to the temp array, which is our new
        // array
        data = temp;
    }
}
```

[FIGURE 6-9] Array-based implementation of the `List` interface of Figure 6-6

Because we implemented the list concept of "logical adjacency" using "physical adjacency," some array-based operations will run extremely well while others are quite inefficient. It is important to analyze and understand the behavior of all operations in Figure 6-9 so we can make intelligent decisions about whether the array-based approach is the best choice for a specific problem.

An array is a random-access data structure, which means that we can directly access any element in O(1) time, once we know its location within the array. We make use of this feature every time we write something like:

```
val = data[i];   // access the ith element of the array
data[j] = 5.6;   // modify the jth element of the array
```

Therefore, the following list operations, which access or modify a node in the list using the current position pointer or a position number, all run in O(1) time:

```
first()
last()
next()
previous()
size()
get()
get(position)
set(element)
set(element, position)
```

For example, to implement the operation `get(13)`, which retrieves the information field of the node at position 13 in the list, we can execute:

```
info = data[12];    // directly access data[12],
                    // the 13th element in the list
```

Similarly, to set the value of the node pointed at by the current position indicator to the value 12.3 (in other words, `set(12.3)`), we only need to write:

```
data[cursor] = 12.3; // directly access the element
                     // referred to by cursor, the
                     // current position pointer
```

These behaviors lead us to the conclusion that if the most common operations carried out on a list are iterating through the list (for example, `first()`, `last()`, `next()`, and `previous()`) and accessing and/or modifying the contents of existing nodes (for example, `get()`, `get(position)`, `set(element)`, and `set(element, position)`), then the array implementation just described would be an excellent and highly efficient choice, because we can complete these operations in constant time.

However, the following operations, which add new elements to the list and delete existing elements from the list, all run in linear, O(*n*) time:

```
remove()
remove(position)
add(element)
add(element, position)
addFirst(element)
```

The reason for this slower behavior is that to add a new element to a list (except at the very end), we must make room for it by moving all other elements down one position to open up a "hole." This is due to the array's use of physically adjacent memory cells in the computer to store elements; there is no room available between array elements `data[i]` and `data[i+1]` to place a new node. We can see this situation in Figure 6-10, in which we are executing the operation `add(99, 3)`, which adds a new node containing the value 99 into position 3. The original seven-element list is shown in Figure 6-10(a). Before we can complete this insertion, we must create a free slot at position 3 by moving all elements from position 3 (the insertion point) to position 6 (the last element) down one slot. The situation after performing this operation is shown in Figure 6-10(b). Now we can insert the new value 99, which is diagrammed in Figure 6-10(c).

On average, we will insert the new node somewhere in the middle of the array, and the "move-down" operation in Figure 6-10(b) must be repeated $n/2$ times, where n is the size of the array. This is an O(n) operation.

INDEX	VALUE		INDEX	VALUE		INDEX	VALUE	
0	32		0	32		0	32	
1	−6		1	−6		1	−6	
2	25		2	25		2	25	
3	88	← insert	3		← hole	3	99	← new value
4	−91		4	88		4	88	
5	67		5	−91		5	−91	
6	11	← last	6	67		6	67	
			7	11	← last	7	11	← last
(a) Before insertion			*(b) Opening a hole*			*(c) After insertion*		

[FIGURE 6-10] Insertion into an array-based list

Similarly, removing an item from somewhere in the middle of the list requires moving an average of $n/2$ items up one position to "fill in" the hole created by the deletion. As with the insertion operation in Figure 6-10, this "move-up" process is a linear-time O(n) operation.

Finally, as mentioned earlier in this section, we have one additional problem. An array is a static, fixed-size data structure that cannot be enlarged once it has been created. Therefore, if a list grows beyond its original capacity, it must be dynamically resized with the elements of the original array copied into this new, larger structure. This is also an O(n) operation.

The analysis of the last few paragraphs leads us to conclude that if we are working with a list that is highly dynamic, with many insertions and deletions, then an array-based implementation could be a poor choice because both insertions and deletions run in O(n) time.

Furthermore, if we have no clue about the maximum size to which the list may grow, it is hard to estimate how large to make the array, leading to either wasted memory or a good deal of resizing. In both cases, we might want to consider an alternative implementation. The next section will present just such an alternative—a reference-based implementation of lists.

The Java Collection Framework includes a class `ArrayList`, an array-based implementation of the list data structure that is similar, but not identical to, the structure presented in this section. We will describe these differences in the coming pages and discuss the detailed characteristics of `ArrayList` in Chapter 9.

Figure 6-11 summarizes the complexity of the list methods in the `ArrayList` class shown in Figure 6-9.

METHOD	WORST-CASE TIME COMPLEXITY
`first()`	O(1)
`last()`	O(1)
`next()`	O(1)
`previous()`	O(1)
`size()`	O(1)
`remove()`	O(n)
`remove(position)`	O(n)
`add(element)`	O(n)
`add(position, element)`	O(n)
`addFirst(element)`	O(n)
`get()`	O(1)
`get(position)`	O(1)
`set(element)`	O(1)
`set(position, element)`	O(1)

[FIGURE 6-11] Time complexity of the list methods of Figure 6-6 using the array-based implementation of Figure 6-9

6.2.3.2 A Reference-Based Implementation

In this section we show how to use reference variables, rather than arrays, to implement the `List` interface of Figure 6-6 and determine when this technique would be superior to the array-based implementation presented in the previous section.

For our reference-based implementation, we will use a class called `LinkedNode` to specify the structure of a single node in the list. Each node in the list includes the two fields we have been using throughout the chapter—`data` and `next`. These two fields are implemented as **private** instance variables, so our `LinkedNode` class must include two accessor methods, `getData` and `getNext`, and two mutator methods, `setData` and `setNext`, that allow us to access or change the value of these two fields. The `LinkedNode` class is shown in Figure 6-12.

```
/**
 * This class represents the nodes in a singly linked list.
 */
public class LinkedNode<T> {

    private T data;               // The data stored in this node
    private LinkedNode<T> next;  // The next node in the data structure

    /**
     * Default constructor for the LinkedNode
     */
    public LinkedNode() {
        this( null, null );
    }

    /**
     * Construct a node given the info and next references.
     *
     * @param newData the data to be associated with this node
     * @param newNext a reference to the next node in the list
     */
    public LinkedNode( T newData, LinkedNode<T> newNext ) {
        data = newData;
        next = newNext;
    }

    /**
     * Return the data stored in this node.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The node is unchanged.
     *
     * @return a reference to the data stored in this node
     */
    public T getData() {
        return data;
    }
```

continued

```
    /**
     * Return a reference to the next node in the data structure.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *   The node is unchanged.
     *
     * @return a reference to the next node in the list, or null if
     *         this node has no successor.
     */
    public LinkedNode<T> getNext() {
        return next;
    }

    /**
     * Set the data associated with this node.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *   The data associated with this node has been changed.
     *
     * @param newData the data to be associated with this node.
     */
    public void setData( T newData ) {
        data = newData;
    }

    /**
     * Set the reference to the next node.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The reference to the next node has been changed.
     *
     * @param newNext the reference to the next node in the list.
     */
    public void setNext( LinkedNode<T> newNext ) {
        next = newNext;
    }

}
```

[FIGURE 6-12] The `LinkedNode` class

The instance variables needed for the class `LinkedList`, a reference-based implementation of the `List` interface of Figure 6-6, are (1) a variable `head` that points to the first node in the list and (2) a current position indicator `cursor`. The default constructor initializes both of these variables to **null** to create an empty list. In addition to these two variables, we are free to add more state information to increase the efficiency of our new implementation. For example, we might want to create a `last` pointer to identify the node at the end of the list, as we showed in Figure 6-1. In addition, we might want to include the variable `count` to hold the total number of nodes in the list. We have included both of these state variables in our implementation.

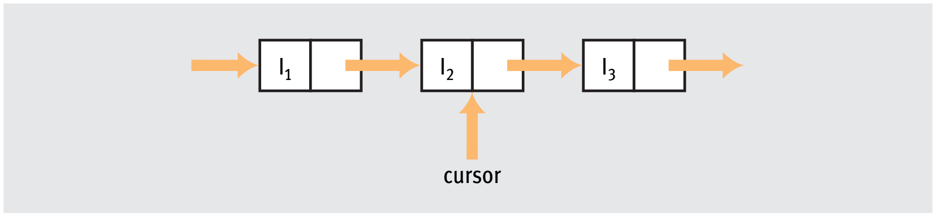
The declarations for the reference-based implementation of the `List` interface are shown in Figure 6-13. They use the node structure defined by the `LinkedNode` class of Figure 6-12.

```
public class LinkedList<T> implements List<T> {

    protected int count;               // The number of nodes in the list
    protected LinkedNode<T> head;      // The first node in the list
    protected LinkedNode<T> last;      // The last node in the list
    protected LinkedNode<T> cursor;    // The current node in the list
```

[FIGURE 6-13] Declarations for the reference-based implementation of a list

Let's work through the development of some of the reference-based methods included in the `List` interface of Figure 6-6.

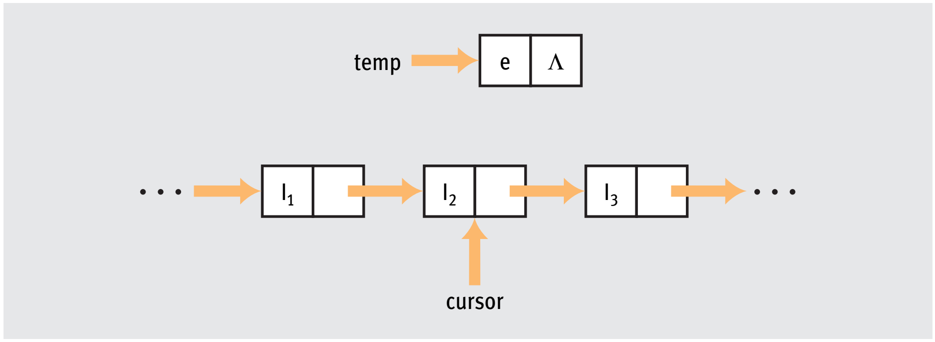
The first method we will implement is `add(element)`. This method creates a new node that contains `element` in the information field and inserts this new node into the list as the successor of the node pointed at by `cursor`. An obvious precondition of this method is that `cursor` is not currently off the list. Here are the starting conditions:



To insert the new node in its proper place, we must carry out the following algorithm. At each step we provide the Java code needed to accomplish the operation and include a diagram that shows the state of the list at this point in the addition process.

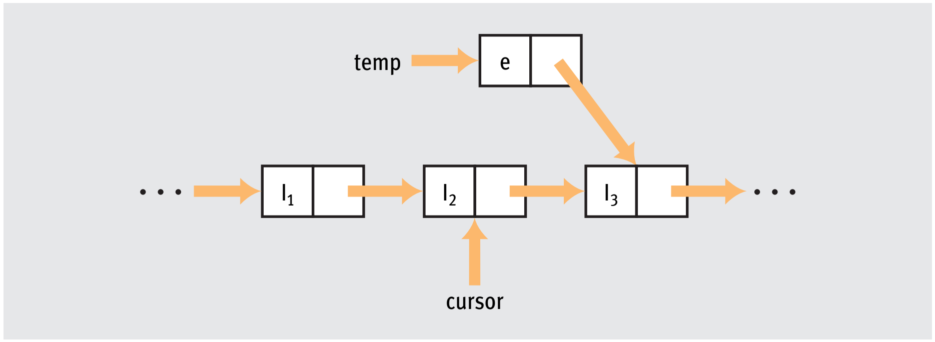
1 Allocate space for a new node, called `temp`, containing the value `element` in the information field (in the following diagrams, `element` is labeled as "e"):

```
LinkedNode<T> temp = new LinkedNode<T>();
temp.data = element;
temp.next = null;
```



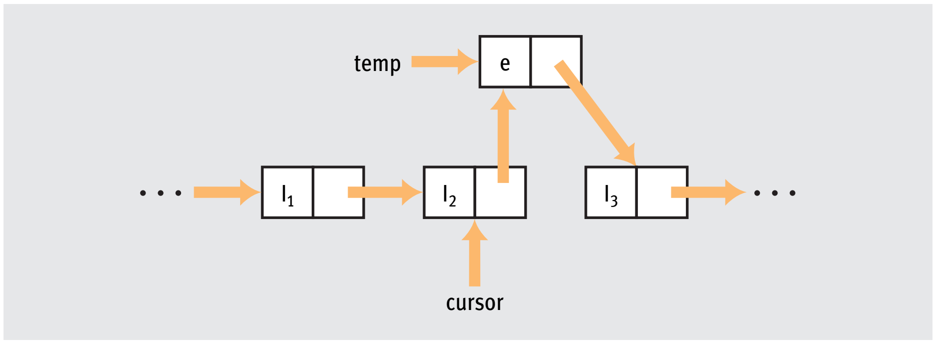
2 Set the `next` field of `temp` to refer to the successor of `cursor`. This value is **null** if we are adding the new node to the end of the list. In that case we must also update the variable `last`.

```
temp.next = cursor.next;
if (temp.next == null) last = temp;
```



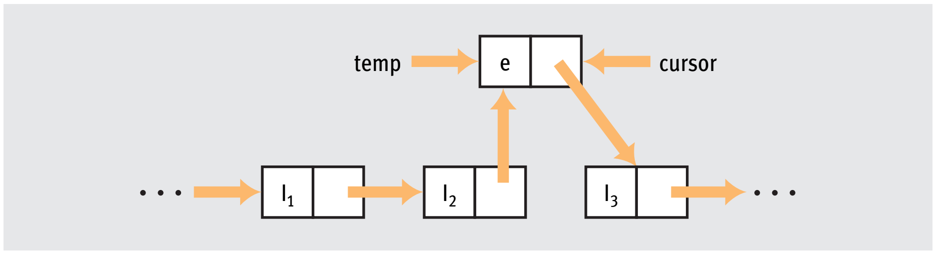
3 Reset the `next` field of the node referenced by `cursor` to point to `temp`.

```
cursor.next = temp;
```



4 Finally, reset `cursor` to point to `temp`, the new node just added, and update the total node count by 1.

```
cursor = temp;
count++;
```



The `add(element)` method is now complete. We have added the new node in its proper location, updated `cursor` (and, if necessary, `last`), and tallied that one more node has been added to the list. The time complexity of this operation is O(1) because a constant number of steps are required, regardless of the length of the list.

Many methods in this reference-based implementation run in O(1) time for two reasons: (1) the existence of `cursor`, which directly points to the location where this operation is to be performed, and (2) because we are not using physical adjacency to implement the concept of a successor node. Instead, each node explicitly points to its successor in the list using the `next` field. Therefore, space is available for a new value between two logically adjacent nodes.

For example, `set(element)`, which resets the `data` field of the node pointed at by `cursor`, can be implemented in a single step:

```
cursor.setData(element);
```

Similarly, `next()`, which advances `cursor` to the successor of the current node, is also O(1):

```
cursor = cursor.getNext();
```

Another reason for the O(1) behavior of many methods is the additional information we added to our class. For example, in Figure 6-13 we included a state variable `last` that references the last node in the list. Because of the existence of this instance variable, the method `last()`, which resets `cursor` to the last element in the list, also requires only one line of code:

```
cursor = last;
```

Without the instance variable `last`, we would need to iterate through the list to locate the end—an O(n) operation. This demonstrates how the intelligent inclusion of additional state information can speed up the implementation of methods that operate on an object.

As a slightly more complex example, let's develop the code for the `remove()` method. This method deletes the node pointed at by `cursor`. This is not as easy as it sounds, because we must first locate the predecessor of the node referenced by `cursor` to reset its `next` field. Unfortunately, our implementation of `LinkedNode` included only a single pointer to the *successor* of each node. Therefore, to locate the *predecessor* of `cursor`, we must traverse the entire list until we come to the node whose `next` field references the node pointed at by `cursor`. Only then can we implement `remove()`. Unfortunately, this operation takes O(n) time. (In the following section, we show how to reduce this to O(1) by maintaining additional state information in the node, this time a predecessor pointer.)

The operation to locate the predecessor is identical to the method `previous()` in the `List` interface of Figure 6-6. The code for `previous()` is shown in Figure 6-14.

```
public void previous() {
    cursor = findPrevious();
}

private LinkedNode<T> findPrevious() {
    LinkedNode<T> prev = null;  // Where we are in the list
```

continued

```
        // If the cursor is off the list or the cursor is at the
        // head, there is no previous node.
        if ( cursor != null && cursor != head ) {
            prev = head;

            // Keep looking until we find the node whose next reference
            // is equal to the current node.
            while ( prev.getNext() != cursor ) {
                prev = prev.getNext();
            }
        }

        // The variable prev now refers to the previous node
        return prev;
    }
```

[FIGURE 6-14] Code for the previous() method

Before finishing the code for the method `remove()`, we will point out some aspects of the code in Figure 6-14 that are common programming pitfalls when working with reference-based list structures.

The first problem is failing to check for the possibility of working on an **empty list**. If a list can be empty and you do not determine this fact, then you could attempt to dereference the **null** pointer—a fatal error. That is why we included a check for the condition `(cursor != null)` in Figure 6-14.

Another common problem is an **off-by-one error**, in which you iterate through a list either one node too few or one node too far. In Figure 6-14, if the criterion to terminate the loop was improperly written as `while (prev != cursor)` rather than `while (prev.getNext() != cursor)`, the program would incorrectly move one step beyond the predecessor node we were trying to locate. Whenever you iterate through a list, be sure to check that the loop terminates with the iterator in the desired position.

One final pitfall is a failure to check for **special cases** that require unique handling. We already mentioned one common case—an empty list. Two other special cases we may need to check include:

- Operating on the **first node** in a list—This is a special case because the first node does not have a predecessor and is referenced by the pointer `head` rather than the `next` field of the preceding node. In Figure 6-14, we included a check for the case `(cursor != head)`.
- Operating on the **last node** in a list—This is a special case because the last node does not have a successor and is referenced by the pointer `last`. In Step 2 of the `add(element)` method shown earlier, we checked to see if we were adding the new element to the end of the list, because that meant we had to update the `last` pointer.

Thus, as you can see in the previous examples, there are many sources of errors when working with reference-based lists, and you must carefully check to ensure that your code works properly in all of the following circumstances:

- Empty lists
- Terminating a list traversal at the correct location
- Working on the first element of a list
- Working on the last element of a list

Now that we have written and analyzed `previous()`, we can finish the implementation of `remove()`. Figure 6-15 shows the code for the `remove()` method. Its complexity is O(n) because it uses `previous()` to locate the predecessor of the node to be deleted.

```
public void remove() {
    removeNode( cursor, findPrevious() );
}

private void removeNode(LinkedNode<T> target, LinkedNode<T>
prev) {
    if(target != null){
        // Cursor is always set to the target's successor.
        cursor = target.getNext();

        if ( prev == null ) {
            // We are deleting the head.
            head = target.getNext();
        }
        else {
            // We are somewhere in the middle of the list.
            prev.setNext( target.getNext() );
        }

        // Did we just delete the tail?
        if ( target == last ) {
            last = prev;
        }

        // One less item in the list
        count = count - 1;
    }
}
```

[FIGURE 6-15] Code for the remove() method

Figure 6-16 shows a complete, reference-based `LinkedList` class that implements the `List` interface of Figure 6-6. This implementation uses the declarations in Figures 6-12 and 6-13.

```
/**
 * An implementation of the List interface using references.
 * Javadoc comments for methods specified in the List interface have
 * been omitted.
 *
 * This code assumes preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class LinkedList<T> implements List<T> {

    protected int count;              // The number of nodes in the list
    protected LinkedNode<T> head;     // The first node in the list
    protected LinkedNode<T> last;     // The last node in the list
    protected LinkedNode<T> cursor;   // The current node in the list

    /**
     * Create a new list.
     */
    public LinkedList() {
        count = 0;
        head = null;
        last = null;
        cursor = null;
    }

    public void first() {
        cursor = head;
    }

    public void last() {
        cursor = last;
    }

    public void next() {
        cursor = cursor.getNext();
    }

    public void previous() {
        cursor = findPrevious();
    }

    public void remove() {
        removeNode( cursor, findPrevious() );
    }
```

continued

```
    public void remove( int position ) {
        LinkedNode<T> target = head;
        LinkedNode<T> prev = null;

        // Find the node that contains the element we want to
        // delete and the node immediately before it.
        while ( position > 0 ) {
            prev = target;
            target = target.getNext();
            position = position - 1;
        }

         // Target now refers to the node we want to remove
        removeNode( target, prev );
    }

    public void add( T element ) {
        LinkedNode<T> after = null;
        if( cursor != null ){
             after = cursor.getNext();
        }

        // Add the node to the list
        addNode( element, cursor , after );
    }

    public void add( T element, int position ) {
        LinkedNode<T> after = null;
        LinkedNode<T> before = null;

        // Set before and after to the appropriate values
        if ( position == 0 ) {
            before = null;
            after = head;
        }
        else {
            before = positionToReference( position - 1 );
            after = before.getNext();
        }

        // Add the node to the list
        addNode( element, before, after );
    }

     public void addFirst(T element) {
        addNode( element, null, head );
    }
```

continued

```
public T get() {
    return cursor.getData();
}

public T get( int position ) {
    return positionToReference( position ).getData();
}

public void set( T element ) {
   cursor.setData( element );
}

public void set( T element, int position ) {
   LinkedNode<T> target = positionToReference( position );
   target.setData( element );
}

public int size() {
  return count;
}

public boolean isOnList() {
      return cursor != null;
}

/**
 * Return a string representation of this list.
 *
 * @return a string representation of this list.
 */
public String toString() {
    // Use a StringBuffer so we don't create a new
    // string each time we append
    StringBuffer retVal = new StringBuffer();

    retVal = retVal.append( "[ " );

    // Step through the list and use toString on the elements to
    // determine their string representation
    for ( LinkedNode<T> cur = head; cur != null;
          cur = cur.getNext() ) {
        retVal = retVal.append( cur.getData() + " " );
    }

    retVal = retVal.append( "]" );

    // Convert the StringBuffer to a string
    return retVal.toString();
```

continued

```
    }

    /**
     * Return a reference to the previous node in the list.
     *
     * @return a reference to the node before the cursor.
     */
    private LinkedNode<T> findPrevious() {
        LinkedNode<T> prev = null;  // Where we are in the list

        // If the cursor is off the list or the cursor is at the
        // head, there is no previous node.
        if ( cursor != null && cursor != head ) {
            prev = head;

            // Keep looking until we find the node whose next reference
            // is equal to the current node
            while ( prev.getNext() != cursor ) {
                prev = prev.getNext();
            }
        }

        // The variable prev is now referring to the previous node.
        return prev;
    }

    /**
     * Remove the specified node from the list.
     *
     * @param target the node to remove.
     * @param prev the node before the node to be deleted.
     */
    private void removeNode(LinkedNode<T> target, LinkedNode<T> prev) {
        if(target != null){
            // Cursor is always set to the target's successor
            cursor = target.getNext();

            if ( prev == null ) {
                // We are deleting the head.
                head = target.getNext();
            }
            else {
                // We are somewhere in the middle of the list.
                prev.setNext( target.getNext() );
            }

            // Did we just delete the tail?
```

continued

```
        if ( target == last ) {
            last = prev;
        }

        // One less item in the list
        count = count - 1;
    }
}

/**
 * Add the specified element to the list between the nodes
 * identified by before and after.
 *
 * @param element the element to add.
 * @param before the node before the node to be added.
 * @param after the node after the node to be added.
 */
private void addNode( T element, LinkedNode<T> before,
                      LinkedNode<T> after ) {

    // Create the node
    cursor = new LinkedNode<T>( element, after );

    // Is this a new head?
    if ( before == null ) {
          head = cursor;
    }
    else {
        before.setNext( cursor );
    }

    // Is it a new tail?
    if ( after == null ) {
        last = cursor;
    }

    // One more element in the list
    count = count + 1;
}

/**
 * Return a reference to the node at the specified position.
 *
 * @param position the node to obtain the reference.
 * @return a reference to the node at the specified position.
 */
private LinkedNode<T> positionToReference( int position ) {
    LinkedNode<T> retVal = head;
```

continued

```
        while ( position > 0 ) {
            retVal = retVal.getNext();
            position = position - 1;
        }

        return retVal;
    }
}
```

[FIGURE 6-16] Reference-based implementation of the List interface of Figure 6-6

The time complexity of the methods in Figure 6-16 is summarized in Figure 6-17.

METHOD	WORST-CASE TIME COMPLEXITY
`first()`	O(1)
`last()`	O(1)
`next()`	O(1)
`previous()`	O(n)
`size()`	O(1)
`remove()`	O(n)
`remove(position)`	O(n)
`add(element)`	O(1)
`add(element, position)`	O(n)
`addFirst(element)`	O(1)
`get()`	O(1)
`get(position)`	O(n)
`set(element)`	O(1)
`set(element, position)`	O(n)

[FIGURE 6-17] Time complexity of the list methods of Figure 6-6 using the reference-based implementation of Figure 6-16

There are some significant differences in the run-time behaviors of the two implementations of `List` we have developed—that is, in the complexity values shown in Figures 6-11 and 6-17.

In the reference-based approach of Figure 6-16, any operation that must initially move to a given position number in the list, such as:

```
add(element, position)
remove(position)
get(position)
set(element, position)
```

requires a minimum of O(n) time. In a reference-based implementation we cannot move directly to an arbitrary position in the list in a single step. Instead, we must start at the head of the list and traverse nodes, one at a time, until we come to the desired position. This contrasts with an array in which the `get(position)` and `set(position, element)` operations are both O(1).

Similarly, any operation that requires us to locate the predecessor of a node:

```
remove()
previous()
```

requires O(n) time, as we showed in Figures 6-14 and 6-15. In an array-based implementation, `previous()` runs in O(1) time, while `remove()` still requires O(n) time because of the need to move array elements up one slot to fill in the hole created by the deletion.

All of the other operations listed in Figure 6-16:

```
first()
last()
next()
size()
add(element)
addFirst(element)
get()
set(element)
```

run in O(1) time. The most noticeable improvement is the O(1) running time of both the `add(element)` and `addFirst(element)` operations. The addition of a new node at the beginning of the list or after the node currently pointed at by `cursor` is a common operation, and our reference-based approach has reduced the time required for both of these operations from O(n) using an array to O(1).

The preceding paragraphs lead to the conclusion that if the most common operations on a list are inserting a new value at the `cursor` location (`add(element)`) or at the beginning (`addFirst(element)`), or if we frequently access or modify the node at the `cursor` location (`get()/set(element)`), then a reference-based implementation is a good choice. However, if we frequently insert, delete, and access a node at an arbitrary position number, then the random-access capabilities of the array likely make it the superior choice.

Although we have certainly gained some benefits from our new reference-based implementation, we still need to address a few problems. Specifically, list operations that require you to locate the predecessor of a node still require O(*n*) time, including such popular methods as `remove()` and `previous()`. The next section presents an improved reference-based implementation of lists that is much more efficient.

6.2.3.3 Doubly Linked Lists and Circular Lists

The operations in Figure 6-17 that still require O(*n*) time are those that must access the predecessor of `cursor`, such as `previous()` and `remove()`, or those that must traverse the list from the beginning to locate a specific location, such as `get(position)` or `set(element, position)`.

The `LinkedNode` class of Figure 6-12 contains a single reference field in each node—a pointer to the successor. Therefore, our physical implementation of a list is virtually identical to the conceptual model diagrammed in Figure 6-5. This type of linear structure is termed a **singly linked list**, and it allows us to iterate through the collection in the "forward" direction—in other words, from a node to its successor. However, if we ever need to access the node that precedes the current one, the only way is to traverse the list, one node at a time, from the very beginning.

We can solve this problem by realizing that, just because the logical structure of a list (Figure 6-5) has pointers in only one direction, we do not have to conform to that exact structure. The state information we maintain about a list can include anything that helps us build a correct and efficient representation. (That is why, for example, we included `count` and `last` as state variables in our singly linked implementation.) Therefore, we can create a node class that includes *both* forward and backward links, called `next` and `previous`, as shown in Figure 6-18. The `next` field points to the successor node and is identical to the `next` field in Figure 6-12. The new `previous` field points to the predecessor of a node. The existence of these two pointers allows us to move in both directions through a list—from first to last as well as from last to first.

The `DoublyLinkedNode` class shown in Figure 6-18 produces a structure that is called a **doubly linked list**. Logically, this is still a linear structure but with new state information (the `previous` field) to reduce the running time of some key list methods.

```
/**
 * This is a doubly linked list node class suitable for building linked
```

continued

```
 * data structures such as lists, stacks, and queues.  It includes
 * a second, "rear-facing" link that permits O(1) movement toward the
 * front of the list.
 */
public class DoublyLinkedNode<T> {
    private T data;                       // The data in this node
    private DoublyLinkedNode<T> next;     // The forward link
    private DoublyLinkedNode<T> previous; // The reverse link

    /**
     * Create a new node.
     */
    public DoublyLinkedNode() {
        this( null, null, null );
    }

    /**
     * Construct a node given the info and next references.
     *
     * @param newData the data to be associated with this node.
     * @param newPrevious a reference to the previous node in the list.
     * @param newNext a reference to the next node in the list.
     */
    public DoublyLinkedNode( T newData,
                             DoublyLinkedNode<T> newPrevious,
                             DoublyLinkedNode<T> newNext ) {

        data = newData;
        next = newNext;
        previous = newPrevious;
    }

    /**
     * Return the data stored in this node.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The node is unchanged.
     *
     * @return a reference to the data stored in this node.
     */
    public T getData() {
        return data;
    }
```

continued

```
/**
 * Return a reference to the next node in the data structure.
 *
 * Preconditions:
 *   None
 * Postconditions:
 *   The node is unchanged.
 *
 * @return a reference to the next node in the list, or null if
 *         this node has no successor.
 */
public DoublyLinkedNode<T> getNext() {
      return next;
}

/**
 * Return a reference to the previous node in the data structure.
 *
 * Preconditions:
 *   None.
 * Postconditions:
 *   The node is unchanged.
 *
 * @return a reference to the previous node in the data structure.
 */
public DoublyLinkedNode<T> getPrevious() {
      return previous;
}

/**
 * Set the data associated with this node.
 *
 * Preconditions:
 *   None
 * Postconditions:
 *   The data associated with this node has been changed.
 *
 * @param newData the data to be associated with this node.
 */
public void setData( T newData ) {
      data = newData;
}

/**
 * Set the reference to the next node.
 *
 * Preconditions:
 *   None.
```

continued

```
     * Postconditions:
     *   The reference to the next node has been changed.
     *
     * @param newNext the next node in the list.
     */
    public void setNext( DoublyLinkedNode<T> newNext ) {
          next = newNext;
    }

    /**
     * Set the reference to the previous node in the data structure.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The reference to the previous node has been changed.
     */
    public void setPrevious( DoublyLinkedNode<T> newPrev ) {
          previous = newPrev;
    }
}
```

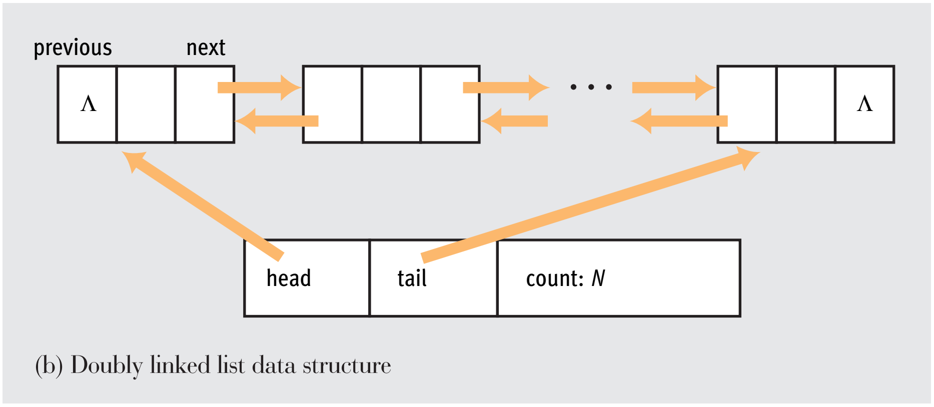
[FIGURE 6-18] Node class for a doubly linked implementation of a list

We are only changing the structure of individual nodes, not the structure of the list built from these nodes. The state information required by a `DoublyLinkedList` object is still the same—references to the first node (`head`) and last node (`last`) in the list, a current position (`cursor`), and a count of the total number of nodes in the list (`count`).

Figure 6-19(a) shows the declarations required to implement the `List` interface of Figure 6-6 using the doubly linked node structure of Figure 6-18. Figure 6-19(b) shows what this doubly linked implementation looks like for a typical list.

```
public class DoublyLinkedList<T> implements List<T> {
    private int count;                     // Number of nodes in the list
    private DoublyLinkedNode<T> head;      // The first node
    private DoublyLinkedNode<T> last;      // The last node
    private DoublyLinkedNode<T> cursor;    // The current node
```

(a) Declarations for a doubly linked implementation of a list



[FIGURE 6-19] Doubly linked list data structure

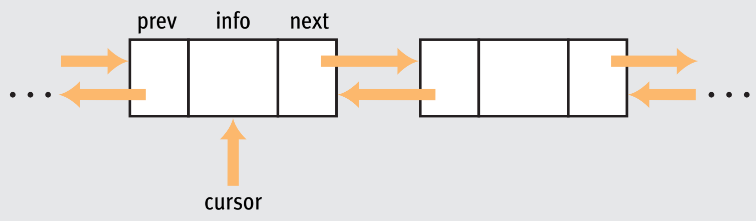
Using a singly linked list, the time complexity of `previous()` was O(*n*). Using our new doubly linked structure, this operation can be completed in a single line:

```
cursor = cursor.getPrevious();  // assuming cursor
                                // is not off the list
```

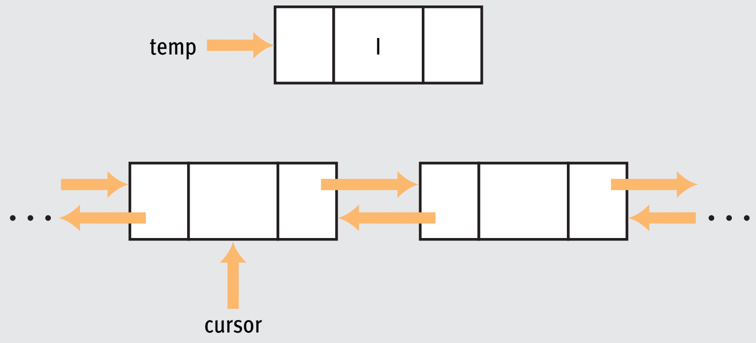
The existence of the `previous` field makes the implementation of this operation trivial and reduces its running time from O(*n*) to O(1).

However, as economists are fond of saying, there is no such thing as a "free lunch," and the price we must pay for this improved running time is twofold. First, the amount of memory required to store the complete list has increased because of the second reference field inside each node. If the list is large, this increased memory may be significant. Second, programming complexity increases because we must maintain and update two reference fields, not just one.

In the previous section, we discussed the pitfalls that await a programmer writing a reference-based implementation of a list—pitfalls such as empty lists, special cases, and off-by-one errors. With a doubly linked list, the possibility of encountering one of these pitfalls is even greater. For example, Figure 6-20 shows the steps required to add a new node to a doubly linked list.

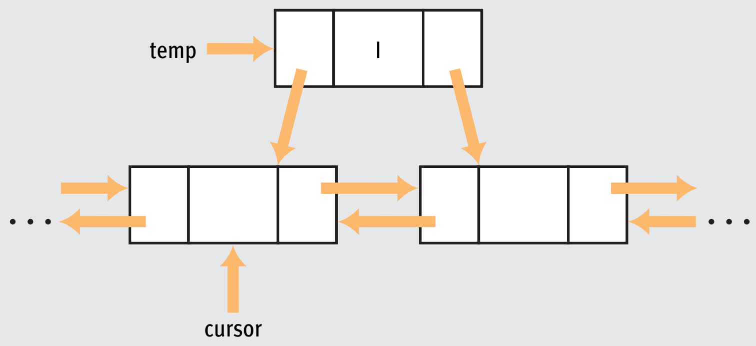


(a) Starting conditions



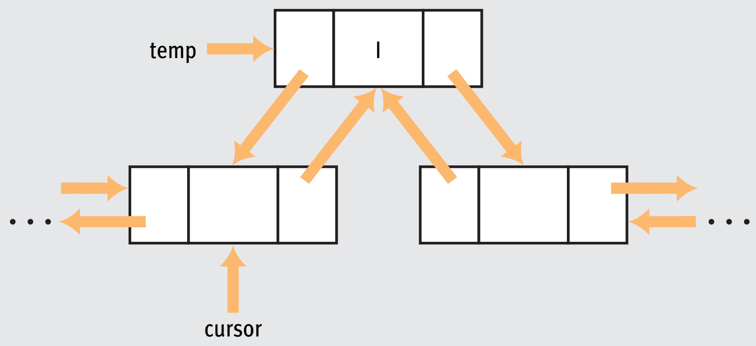
```
DoublyLinkedNode temp = new DoublyLinkedNode();
temp.info = I;
```

(b) Creating the new node



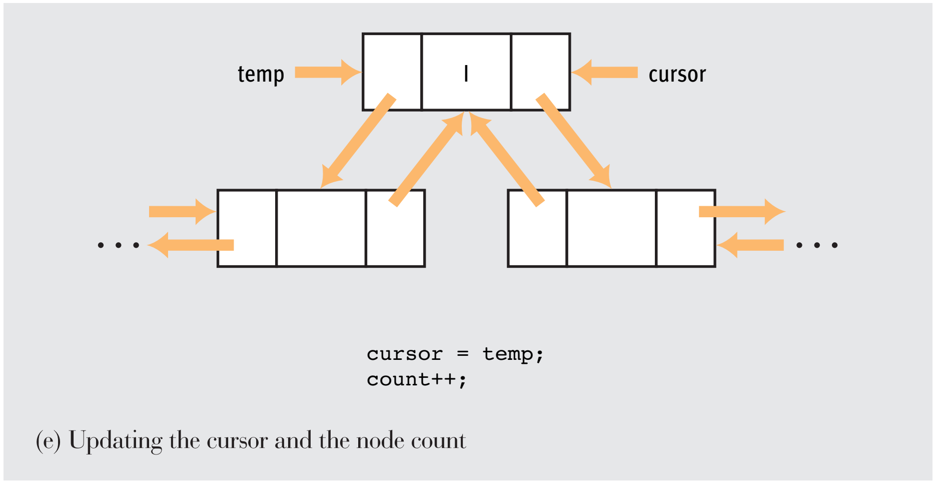
```
temp.previous = cursor;
temp.next     = cursor.next;
```

(c) Setting the pointer fields of the new node



```
cursor.next = temp;
(temp.next).previous = temp;
```

(d) Linking the new node into the list



```
cursor = temp;
count++;
```

(e) Updating the cursor and the node count

[FIGURE 6-20] Steps involved in adding a new node to a doubly linked list

Even though the operations diagrammed in Figure 6-20 execute in constant time, they are still complex. Four pointer fields must be correctly set to insert the new node—the `next` and `previous` links of the new node, the `next` link of the node pointed at by `cursor`, and the `previous` link of the successor of `cursor`. In addition, we must reset `cursor` to point to the newly added node. Finally, we must check three special cases (not shown in Figure 6-20) to ensure that the operation works correctly in all situations:

- The cursor is off the list, but the list is not empty. We will not know where to add the new node.
- The successor of `cursor` is **null**. We are adding this new node to the end of the list, so we must update `last` and modify step (d) in Figure 6-20 to avoid dereferencing a null pointer.
- We are adding the new node to an empty list. In that case, we must add the node and update both `head` and `last`.

When writing methods that manipulate doubly linked list structures, you must take great care to properly set all reference fields, and ensure that the code works correctly on both the expected cases and the special cases mentioned in the previous section.

The complete doubly linked implementation of the `List` interface from Figure 6-6 is shown in Figure 6-21. It uses the declarations in Figures 6-18 and 6-19(a). The Java Collection Framework includes the class `LinkedList` that implements the doubly linked list data structure described in this section. We will describe its behavior in detail in Chapter 9.

This is our third complete implementation of the `List` interface from Figure 6-6, and these redesigns clearly demonstrate the power of the **interface** concept in Java.

Each redesigned implementation—`ArrayList`, `SinglyLinkedList`, and `DoublyLinkedList`—is intended to provide either greater flexibility or better run-time efficiency. However, users would be unaffected by a change from any one of these classes to another (except perhaps for a slight difference in performance) because all three classes implement the `List` interface. This means users can be sure that the same methods with identical signatures are provided by all three classes.

```
/**
 * An implementation of a list using forward and backward references.
 * This changes the order of moving backward in the list from O(n)
 * to O(1), at the cost of adding one reference to each node. Javadoc
 * comments for methods specified in the List interface have been
 * omitted.
 *
 * This code assumes that the preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class DoublyLinkedList<T> implements List<T> {
    private int count;                      // Number of nodes in the list
    private DoublyLinkedNode<T> head;       // The first node
    private DoublyLinkedNode<T> last;       // The last node
    private DoublyLinkedNode<T> cursor;     // The current node

    /**
     * Create a new list.
     */
    public DoublyLinkedList() {
        count = 0;
        head = null;
        last = null;
        cursor = null;
    }

    public void first() {
        cursor = head;
    }

    public void last() {
        cursor = last;
    }

    public void next() {
        cursor = cursor.getNext();
    }
```

continued

```
public void previous() {
    cursor = cursor.getPrevious();
}

public void remove() {
    removeNode( cursor );
}

public void remove( int position ) {
    removeNode( positionToReference( position ) );
}

public void add( T element ) {
    if(cursor != null){
        addNode( element, cursor.getNext() );
    }
    else{
        addNode( element, null );
    }
}

public void add( T element, int position ) {
    addNode( element, positionToReference( position ) );
}

public void addFirst(T element) {
    addNode( element, head );
}

public T get() {
    return cursor.getData();
}

public T get( int position ) {
    return positionToReference( position ).getData();
}

public void set( T element ) {
    cursor.setData( element );
}

public void set( T element, int position ) {
    DoublyLinkedNode<T> target = positionToReference( position );
    target.setData( element );
}
```

continued

```
    public int size() {
        return count;
    }

    public boolean isOnList() {
        return cursor != null;
    }

    /**
     * Return a string representation of this list.
     *
     * @return a string representation of this list.
     */
    public String toString() {
        // Use a StringBuffer so we don't create a new
        // string each time we append
        StringBuffer retVal = new StringBuffer();

        retVal = retVal.append( "[ " );

        // Step through the list and use toString on the elements to
        // determine their string representation
        for ( DoublyLinkedNode<T> cur = head; cur != null;
              cur = cur.getNext() ) {
            retVal = retVal.append( cur.getData() + " " );
        }

        retVal = retVal.append( "]" );

        // Convert the StringBuffer to a string
        return retVal.toString();
    }

    /**
     * Remove the specified node from the list.
     *
     * @param target the node to remove.
     * @param prev the node before the node to be deleted.
     */
    private void removeNode( DoublyLinkedNode<T> target ) {
        if( target != null ){
            DoublyLinkedNode<T> before = target.getPrevious();
            DoublyLinkedNode<T> after = target.getNext();

            // Cursor is always set to the target's successor
            cursor = after;
```

continued

```
            // Set the next reference of the node before the target
            if ( before == null ) {
                // We are deleting the head
                head = after;
            }
            else {
                // We are somewhere in the middle of the list
                before.setNext( after );
            }

            // Set the previous reference of the node after the target
            if ( after == null ) {
                // We are deleting the tail
                last = before;
            }
            else {
                // We are somewhere in the middle of the list
                after.setPrevious( before );
            }

            // One less item in the list
            count = count - 1;
        }
    }

    /**
     * Add the specified element to the list between the nodes
     * identified by before and after.
     *
     * @param element the element to add.
     * @param after the node after the node to be added.
     */
    private void addNode( T element, DoublyLinkedNode<T> after ) {
        DoublyLinkedNode<T> before = null;

        // Determine what is before the new node
        if ( after == null ) {
            // Adding a new tail to the list
            before = last;
        }
        else {
            // Adding somewhere in the middle of the list
            before = after.getPrevious();
        }

        // Create the node
        cursor = new DoublyLinkedNode<T>( element, before, after );
```

continued

```
        // Take care of the next reference of the node before the new
        // node
        if ( before == null ) {
            // Adding a new head
            head = cursor;
        }
        else {
            // Somewhere in the middle of the list
            before.setNext( cursor );
        }

        // Take care of the previous reference of the node after the
        // new node
        if ( after == null ) {
            // Adding a new tail
            last = cursor;
        }
        else {
            after.setPrevious( cursor );
        }

        // One more node in the list
        count = count + 1;
    }

    /**
     * Return the node at the specified position in the list.
     *
     * @param position the node to obtain the reference for.
     * @return a reference to the node at the specified position.
     */
    private DoublyLinkedNode<T> positionToReference( int position ) {
        DoublyLinkedNode<T> retVal = head;

        while ( position > 0 ) {
            retVal = retVal.getNext();
            position = position - 1;
        }

        return retVal;
    }
}
```

[FIGURE 6-21] A doubly linked implementation of the List interface

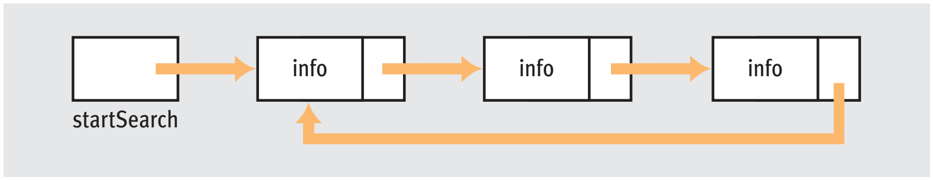
Our final implementation of the `List` interface is a **circular list**. Assume that we have a "traditional" singly linked list structure and want to locate a specific data value, I_k, stored somewhere in the list. Assuming that we have no knowledge of where it is located, the only solution is to begin at the head and iterate until we either find the desired value or come to the end. In some situations this approach is perfectly acceptable. However, there are applications in which a beginning-to-end search can cause problems.

For example, imagine that our list contains the name and location of resources on a network that can be dynamically allocated to clients, such as a list of networked printers. Each printer is either "free" or "inUse," and when a client makes a request, our software will search the list until it finds the first available printer. It then marks that printer as "inUse" and returns the name and location to the requesting user, who is now the printer's owner and may use it as long as needed.

If we always begin our search for a free resource at the head of the list, it is obvious that the resources whose names appear near the front will be allocated much more often than those near the back. If these resources are printers, then the ones near the head of the list will have much greater wear and tear on their print head mechanisms. It would be fairer to allocate resources so that all devices are used an equal amount of time.

A simple way to make this allocation is to maintain a state variable called `startSearch` that tells us where the last search ended. When a request comes in, we do not begin our search from the node pointed at by `head`, but from the node pointed at by `startSearch`. This way, the starting point moves circularly through the list and all nodes are allocated on a roughly equal basis in the long run. The instance variable `startSearch` serves a role similar to the variable `cursor` in Figure 6-13, in that it tells us where to initiate the operation.

To implement this circular search procedure, we must be able to easily iterate through the list starting from any initial point, including going from the last node in the list back to the first. We could check for the occurrence of the special value **null** that marks the end of the list, and when we find it, reset `startSearch` to the value of `head`. However, there is an easier way. Instead of marking the end of the list with a `next` value of null, we can simply have it point back to the first node. This produces a structure called a **singly linked circular list**, also called a **ring**, diagrammed in Figure 6-22.



[FIGURE 6-22] Logical model of a circular list

The nice thing about this circular list is that we can traverse it beginning at any node without having to worry about the special case of reaching the end of the list. We simply keep iterating until we return to the node where we started. In a sense, the circular list of Figure 6-22 eliminates the logical concept of a first and last node.

The singly linked circular list is not much different from the singly linked list structure diagrammed in Figure 6-5. The only difference is that if we are at the last node and do a `next()` operation, we go to the first node rather than the last, and if we are at the first node and do a `previous()` operation, we go to the last node. Otherwise, everything else behaves the same.

The fact that the circular list has only two specialized behaviors (`next()` and `previous()`) suggests that the implementation of the singly linked circular list in Figure 6-22 should use the specialization form of inheritance and extend the `LinkedList` class of Figure 6-17. This allows us to inherit all the methods that work the same way on both lists and only write new behaviors for `next()` and `previous()` to handle the automatic wrap-around feature. (We might also want to overwrite `first()` and `last()`, because there is no longer any sense of first and last nodes.) The implementation of `CircularLinkedList` is shown in Figure 6-23. It includes the two methods—`next()` and `previous()`—to implement the specialized traversal behavior of a circular list. Also note that if we are not at the front or back of the list, then we simply invoke the same method in the superclass via the commands `super.next()` or `super.previous()`.

This is a good example of the gains in software productivity that you can achieve through subclassing and inheritance. A complex data structure, such as a singly linked circular list, can be implemented in only eight lines of code because all remaining behaviors are inherited directly from the base class `LinkedList`.

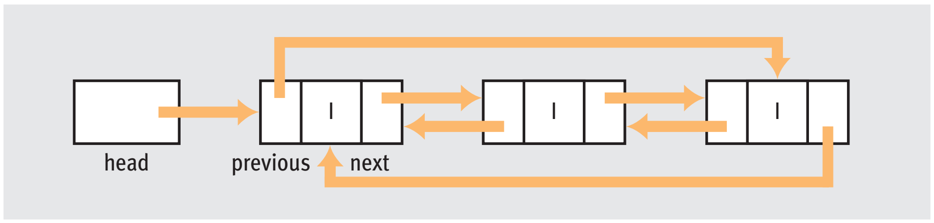
```java
/**
 * This is a simple circular linked list class. Some methods have
 * different semantics to restrict this class's behaviors to more
 * closely match those of the ADT.
 *
 * Javadoc comments for methods specified in the List interface have
 * been omitted.
 */
public class CircularLinkedList<T> extends LinkedList<T> {
    public void next() {
        // If the cursor is at the end of the list, wrap around to the
        // head of the list; otherwise, advance the cursor as usual
        if ( cursor == last ) {
            cursor = head;
        }
        else {
            super.next();
        }
    }
```

continued

```
        public void previous() {
            // If the cursor is at the front of the list, wrap around to
            // the end of the list; otherwise, go backward as usual
            if (cursor == head) {
                cursor = last;
            }
            else {
                super.previous();
            }
        }
    }
```

[FIGURE 6-23] Declarations to create a singly linked circular list

Now let's combine the last two ideas just presented—doubly linked lists and circular lists—to create a **doubly linked circular list** with the structure depicted in Figure 6-24.



[FIGURE 6-24] A doubly linked circular list

The structure diagrammed in Figure 6-24 uses the same doubly linked list declarations shown in Figures 6-18 and 6-19(a). We replaced the two **null** values at the two ends of the list with references to the first and last nodes; that is, the successor of the last node now points to the first node, and the predecessor of the first now points to the last. This doubly linked circular list structure allows us to efficiently iterate forward or backward through the list, beginning at any arbitrary location. For some applications these added capabilities could be very important.

We leave it as an exercise for you to design a class that implements the `List` interface of Figure 6-6 using the logical structure shown in Figure 6-24. Your class should use inheritance and extend the `DoublyLinkedList` class of Figure 6-21.

6.2.4 Summary

This section introduced an important taxonomic classification scheme for data structures—namely, the linear, hierarchical, graph, and set structures. It also began our discussion of data structures by introducing the most flexible of all linear structures—the list. Even though a list is an elementary data structure with a simple (1:1) relationship between elements, there are a number of techniques for implementing it. This section presented both array-based and reference-based approaches; in the latter case, the section described the singly linked structure and the more complex doubly linked, circular, and circular doubly linked variations.

Lists are not just a textbook curiosity but a fundamentally important data representation in computer science. For example, operating systems maintain lists of processes waiting to execute on a processor. They also keep lists of available memory blocks that can be allocated to processes. Compilers use linear structures to keep track of return addresses so that a method can return to the correct location in the program when the method finishes execution. Simulation models use lists to model the waiting lines found in many systems, such as network messages waiting to use a communications link. In functional programming languages, such as LISP and Scheme, the list is the *only* data structure available. These languages do not support such familiar and well-known structures as arrays. Instead, programmers must express their solutions only in terms of lists and list operations like those in Figure 6-6.

Even though the list is the most general and flexible of all linear structures, it is certainly not the only one. The next sections introduce a number of linear structures that place restrictions on where you can carry out certain operations, such as additions, removals, and retrievals. These restrictions produce linear structures with some very interesting and useful properties.

LISP: A LANGUAGE AHEAD OF ITS TIME

When asked to name the oldest high-level programming languages, most computer scientists identify FORTRAN (1956), COBOL (1959), ALGOL (1960), BASIC (1963), RPG (1965), C (1972), and other classical languages from the early days of computing. However, many computer scientists are unaware that the second-oldest high-level programming language after FORTRAN is not any of those listed above, but LISP, developed by John McCarthy of MIT in 1958.

LISP is a **functional programming language** that is quite distinct from other languages of its time. Languages such as FORTRAN and COBOL were designed for the standard Von Neumann model of computing. Therefore, they use the idea of assignment of value (storing a value into a memory cell) and the array as the primary data structure (a high-level view of consecutive memory cells). LISP, however, is based on a totally different model, the **lambda calculus**, developed by Alonzo Church in the 1930s. The name *LISP* is an acronym for *LIS*t *P*rocessing, because linked lists are the primary data structure of the language. All collections of data as well as the source code itself are represented as lists

continued

of values. As a result, LISP programs can manipulate data and programs with similar ease, giving rise to the concept of a self-modifying program.

Given its unusual model of computation and its (to say the least) unique syntax, LISP was not widely used outside academia during its early years. Furthermore, its use of lists as the primary data structure made it difficult to run on the memory-limited machines of its day, keeping it somewhat of an academic curiosity. However, newer variations of the language (Common-LISP, Scheme) and more powerful computing systems have renewed interest in the functional model of computing. Today, the descendants of LISP are found in the classroom and the research lab, especially in the area of artificial intelligence.

6.3 Stacks

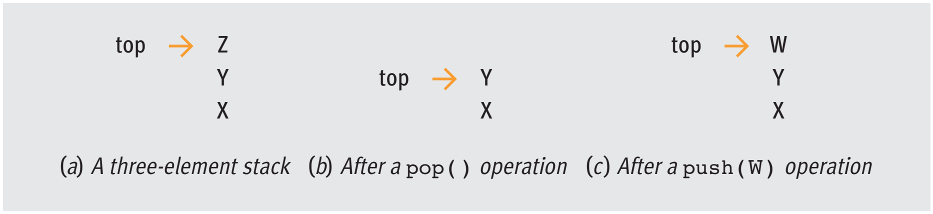
The list structure described in the previous section is the most general of all linear structures because it allows insertions, deletions, and retrievals to be done anywhere. The following sections examine two linear structures that place restrictions on where we can perform insertion, deletion, and retrieval operations. These two structures, the stack and the queue, are widely used in computing applications.

6.3.1 Operations on Stacks

A **stack** is a Last In First Out (LIFO) linear data structure in which the only object that can be accessed is the last one placed on the stack. This element is called the **top** of the stack. When a new value is inserted into the stack, it becomes the new top element. Thus, unlike the list, in which we could retrieve, change, and delete any arbitrary node, in a stack we are limited to retrieving, changing, and deleting only the top node.

Figure 6-25(a) is a diagram of a stack that contains three objects—X, Y, and Z—which were placed on the stack in just that order. Only the last item inserted, in this case Z, can currently be retrieved or deleted. Deleting the top item of a stack is called **popping** the stack, and executing the operation `pop()` produces the stack shown in Figure 6-25(b). The element Y has become the top item because it is the most recent item inserted.

The process of adding a new element to the stack is called **pushing** a value onto the stack. Executing the operation `push('W')` on the stack of Figure 6-25(b) makes W the new top element and results in the situation shown in Figure 6-25(c). No element, other than the one most recently pushed, can be accessed via a push or pop operation.


top → Z
Y
X
top → Y
X
top → W
Y
X
(a) A three-element stack
(b) After a pop( ) operation
(c) After a push(W) operation

[FIGURE 6-25] Last In First Out behavior of a stack

Because of these restrictions, the set of operations permitted on stacks is limited, and the typical `Stack` interface is smaller and much more standardized than the `List` interface of Figure 6-6.

In addition to the two mutator operations `push(data)`, which adds a new value `data` to the top of the stack, and `pop()`, which discards the top value of the stack, another essential operation is determining whether a stack is empty. In an object-oriented environment, stack users are not allowed to know how a stack has been built. Therefore, they cannot check for an empty condition by seeing if `top = null`, `top = 0`, or running any other test that depends on knowledge of the underlying implementation. If they did, then a change to the implementation would "break" the user's code. Instead, we provide an `empty()` method to test for this condition. The `empty()` operation returns true if the stack is empty and false otherwise. It is typically used in the following manner:

```
if  (S.empty())
     // do something when the stack S is empty
else
     // do something when the stack S is not empty
```

A similar method is `full()`. This operation is used to determine whether attempting to push another value onto the stack would cause a fatal error. The `full()` operation returns true if the stack is full and false otherwise.

A method that is similar to the `pop()` operation is `top()`, which returns the value of the stack's top element without deleting it. Recall from Figure 6-25(b) that `pop()` is a mutator that deletes the top element. However, `top()` is an accessor that returns the element currently on top of the stack but does not alter the stack in any way. In particular, then, if you execute two successive `top()` operations on the same stack, you get the same element. (However, Exercise 16 at the end of the chapter shows a slightly different way to define the `top()` and `pop()` operations on stacks.)

An interface for the stack data structure just developed is shown in Figure 6-26. It includes the five operations described above—`push`, `pop`, `top`, `empty`, and `full`. Although other stack operations are certainly possible, these are the most common.

```
/**
 * An interface for a simple stack ADT.
 */
public interface Stack<T> {
    /**
     * Places its argument on the top of the stack.
     *
     * Preconditions:
     *      Stack is not full.
     * Postconditions:
     *      Stack size has increased by 1.
     *      Data is on top of the stack.
     *      The rest of the stack is unchanged.
     *
     * @param data Object to place on the stack.
     */
    public void push( T data );

    /**
     * Removes the top of the stack.
     *
     * Preconditions:
     *      Stack is not empty.
     * Postconditions:
     *      Stack size has decreased by 1.
     *      The top item of the stack has been removed.
     *      The rest of the stack is unchanged.
     */
    public void pop();

    /**
     * Returns the top element of the stack.
     *
     * Preconditions:
     *      The stack is not empty.
     * Postconditions:
     *      The stack is unchanged.
     *
     * @return the top element of the stack.
     */
    public T top();

    /**
     * Determines if the stack is empty.
     *
     * Preconditions:
     *      None.
     * Postconditions:
```

continued

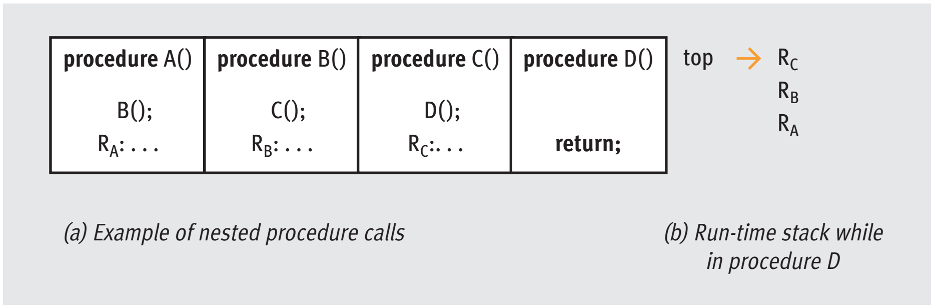
```
     *      The stack is unchanged.
     *
     * @return true if the stack is empty, false otherwise.
     */
    public boolean empty();

    /**
     * Determines if the stack is full.
     *
     * Preconditions:
     *      None.
     * Postconditions:
     *      The stack is unchanged.
     *
     * @return true if the stack is full and false otherwise.
     */
    public boolean full();
}
```

[FIGURE 6-26] Stack interface

Last In First Out stack data structures are widely used in computer science. Take a look at one interesting example.

Whenever a method is called within a program, the computer must save the memory address of the instruction immediately following the call in order to return to the correct location when the called method has completed execution. Furthermore, all addresses must be saved regardless of how many nested calls are made, and we must return to each procedure in exactly the right order.

In Figure 6-27(a), procedure A has called procedure B, which has called procedure C, which has called procedure D. The R_i symbols indicate where a method must return when it has finished—that is, when method B has terminated execution, R_A is the location in procedure A where B should return.



[FIGURE 6-27] Using stacks to implement nested procedure calls

When you have nested method calls, you have many saved return addresses. We must be sure to return to each of them in exactly the right order. For example, when D finishes execution, the program must return to the address immediately following the call to D in procedure C, the location labeled R_C in Figure 6-27(a), not locations R_B or R_A. A stack makes this task quite easy for one important reason—we return to the methods in exactly the reverse order in which the methods were invoked. This fits the LIFO model of a stack perfectly.

When a method is called, the return address is pushed onto a **run-time stack** S, as shown in Figure 6-27(b). That is, a compiler would translate the Java method call `P();` into the following sequence:

```
if (! S.full())
  S.push(address of the instruction after the method call);
  begin execution of method P()
else
  "stack overflow error"
```

This situation is diagrammed in Figure 6-27(b), which shows a simplified model of the run-time stack during execution of procedure D in Figure 6-27(a). (It is simplified because the run-time stack holds more than just a return address. It also holds information on parameters and local variables.)

When a procedure finishes execution, the return address is popped from the run-time stack, and program execution is transferred to that location. That is, the statement **return** would be translated by a Java compiler into:

```
return address = S.top();
S.pop();
continue execution with the instruction at
  the return address;
```

Our choice of a stack to hold return addresses allows us to return to each procedure in exactly the correct order. This technique also supports *recursive* procedures, because the run-time stack holds many invocations of the same procedure rather than different ones.

Stacks are an extremely useful linear data structure whenever an application can make use of its Last In First Out behavior.

6.3.2 Implementation of a Stack

6.3.2.1 An Array-Based Implementation

A simple way to implement a stack is to use a one-dimensional array. The implementation must include an instance variable `top` that points to the top element of the stack. A `top` value in the range [0 .. STACK_SIZE − 1] refers to a specific element in a nonempty stack, while a value of −1 represents the empty stack. The declarations for this array-based implementation are shown in Figure 6-28.

```
public class ArrayStack<T> implements Stack<T> {
    private T theStack[];                        // The stack itself
    private final static int STACK_SIZE = 100;   // Stack capacity
    private int top;                             // Position of top
```

[FIGURE 6-28] Declarations for an array-based implementation of a stack

Implementing the operations in the `Stack` interface using the declarations of Figure 6-28 is quite simple. For example, to push a new value onto the stack, we simply increment `top` and store the new value in this slot of the array, assuming that the stack is not full. To pop a value, we decrement `top`, assuming that the stack is not empty. To retrieve the top value, we return `theStack[top]`. All five stack operations are executed in O(1) time. The restrictions placed on the location of insertions, retrievals, and deletions within a stack make the array an efficient implementation model. An array-based class that implements the `Stack` interface of Figure 6-26 is shown in Figure 6-29.

```
/**
 * An array-based implementation of a stack.  Javadoc comments
 * for the methods specified in the Stack interface have been omitted.
 */
public class ArrayStack<T> implements Stack<T> {
    private T theStack[];                        // The stack itself
    private final static int STACK_SIZE = 100;   // Stack capacity
    private int top;                             // Position of top

    /**
     * Construct a new stack
     */
    public ArrayStack() {
        // Create storage for the stack.  Note the cast is
        // necessary because we cannot create generic arrays.
```

continued

```
        // This statement generates a compiler warning.
        theStack = (T[])new Object[ STACK_SIZE ];

        // Top will be -1 for an empty stack because it is incremented
        // before it is used to add a new element to the stack.
        top = -1;
    }

    public void pop() {
        theStack[ top ] = null;  // Erase reference so the object being
                                 // removed can be collected as garbage

        top = top - 1;           // Lower the top of the stack
    }

    public void push( T data ) {
        top = top + 1;           // Increment the stack pointer
        theStack[ top ] = data;  // Put the data in the stack
    }

    public T top() {
        return theStack[ top ];  // Return the top element
    }

    public boolean empty() {
        return top == -1;        // Empty if top is -1
    }

    public boolean full() {
        // Stack is full if top is at the end of the array
        return top == theStack.length - 1;
    }
}
```

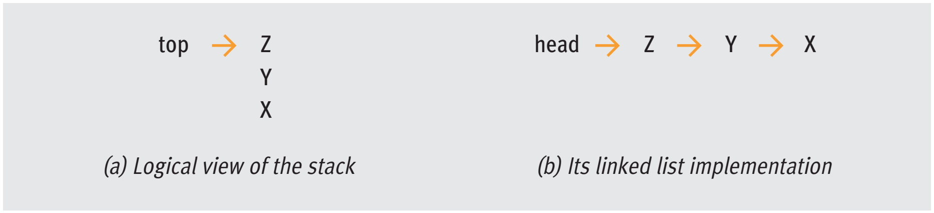
[FIGURE 6-29] Implementation of a stack using arrays

The code in Figure 6-29 is simple, straightforward, and easy to understand, which is why arrays are often used to implement stacks. However, there is still the same problem we discussed previously—the fixed-size restriction on arrays. Once our array is filled with `STACK_SIZE` elements, we cannot do any more `push(v)` operations. Instead, we must either remove existing elements or dynamically resize the array and copy its current values into the new structure. A better implementation would allow our stack to grow to any size we want without the possibility of stack overflow, or at least not until all available memory has been exhausted. We can achieve this using a linked list-based implementation, as introduced in the next section.

6.3.2.2 A Linked List-Based Implementation

The previous section showed how to use an array to implement a stack. However, it is sometimes better to implement a stack using a singly linked list. As mentioned earlier, the advantage of a list is that we do not need to declare a maximum stack size. Instead, the stack can grow as large as needed.

In a linked list implementation, `head` points to the top element of the stack. Because we only access the top element, we will always access our stack via the head pointer. The `next` field of each node points to the element in the stack "underneath" this one. This approach is diagrammed in Figure 6-30, which shows the three-element stack of Figure 6-25(a); the objects X, Y, and Z are implemented as a linked list.


top
Z
Y
X
(a) Logical view of the stack
head
Z
Y
X
(b) Its linked list implementation

[FIGURE 6-30] List implementation of a stack

The declarations needed to create this linked list implementation are similar to the declarations that produced the singly linked list structure of Figures 6-12 and 6-13. The only differences are: (1) there is no longer a need for a `last` pointer, because we can no longer access the node at the end of the list, and (2) there is no need for a current position indicator, because we only work with the top node.

The linked list implementation of the five methods in the `Stack` interface is again quite simple. For example, to push an object `e` onto the stack, we add it as the first element in the stack so that it is referenced by the head pointer:

```
// get space for the new node
LinkedNode<T> temp = new LinkedNode<T>();

temp.data = e;       // Fill in the data field
temp.next = head;    // Have next field point
                     // to the previous top element
head = temp;         // Have head point to this one
```

Similarly, `pop()` removes the node currently pointed at by `head`, an operation that is identical to the `remove(0)` method in Figure 6-6.

```
    head = head.getNext(); // assuming stack not empty
```

The `top()` operation is easily implemented by the following single line:

```
    // assumes the stack is not empty
    return(head.getData());  // return data contained
                             // in the first node
```

Figure 6-31 presents the code for a `LinkedStack` class that implements the `Stack` interface of Figure 6-26. The node structure is specified by the class `LinkedNode`.

```
/**
 * A simple linked implementation of the Stack ADT.  Javadoc
 * comments have been omitted for methods specified in the
 * Stack interface.
 */
public class LinkedStack<T> implements Stack<T> {
    private LinkedNode<T> top;  // Reference to the top of the stack

    /**
     * Create a stack.
     */
    public LinkedStack() {
        top = null;  // Create an empty stack
    }

    public void push( T data ) {
        // Create a node and make it point to the top of the stack.
        LinkedNode<T> newTop = new LinkedNode<T>( data, top );

        // The top of the stack is the new node
        top = newTop;
    }

    public void pop() {
        // The new top is the node after the current top
        top = top.getNext();
    }
```

continued

```
    public T top()  {
        return top.getData();
    }

    public boolean empty() {
        // The stack is empty if top is null
        return top == null;
    }
    public boolean full() {
        // LinkedStacks are never full
         return false;
    }
}
```

[FIGURE 6-31] Linked list-based stack class

All five operations in Figure 6-31 are completed in O(1) time, and this linked list implementation is equally as efficient as the array-based one shown earlier. However, we no longer need to worry about memory limitations imposed by the underlying implementation—in Figure 6-31 the `full()` method always returns false. This implementation of a stack never becomes full as long as the memory manager can satisfy our requests. The small price we pay is the increased space required for the `next` field of each node.

It is interesting to review what we have done to implement our `Stack` interface. We have taken the interface of Figure 6-26 and built two totally different implementations: a one-dimensional array-based implementation (Figure 6-29) and a singly linked list implementation (Figure 6-31).

Each of these approaches has certain advantages and disadvantages. However, the most important point is that, regardless of which technique is ultimately chosen, users are unaware of the myriad technical details regarding pointers, declarations, efficiency, or memory space. All they care about are the resources provided by these classes and how to access them. This is the beauty of object-oriented programming.

The Java Collection Framework includes a class, `Stack`, which implements a stack data structure using the idea of a resizable array. This class includes the operations described in this section—`empty()`, `pop()`, `push(v)`, and `peek()`, which is their name for `top()`. Furthermore, the stack operations of Figure 6-26 could be easily recast in terms of the standard list operations found in the Java class `LinkedList`. Thus, the Java Collection Framework includes both implementations described in this section. We discuss these classes in greater detail in Chapter 9.

EDSGER DIJKSTRA, 1930–2002

Edsger Dijkstra was a Dutch computer scientist and one of the most influential teachers and researchers in the fields of algorithms, software development, programming language design, operating systems, and formal verification.

Dijkstra was Professor of Computer Science at the Eindhoven University of Technology in the Netherlands, a research fellow for the Burroughs Corporation, and, from 1984 to 2000, Distinguished Professor and Chair of the Computer Science Department at the University of Texas in Austin. He died after a long struggle with cancer in August 2002.

Dijkstra was one of the developers of the first ALGOL 60 compiler and a major proponent of well-structured code. His paper "Go to Statement Considered Harmful," which appeared in the March 1968 *Communications of the ACM*, was considered a milestone in the field of language design and led to the development of the software design model called **structured programming**. He was actively involved in algorithm design and created a widely used shortest-path algorithm named after him (which we will study in Chapter 8). He created the semaphore concept, one of the most important synchronization tools in the field of operating systems and distributed processing. (Semaphores are an important tool for coordinating the activities of concurrently executing programs.)

Dijkstra was active in formal verification and advocated strongly for the simultaneous development of a program and the proof of its correctness, a technique he called "correctness by construction." In fact, it is hard to find an area of computer science in which he did not make a fundamental contribution.

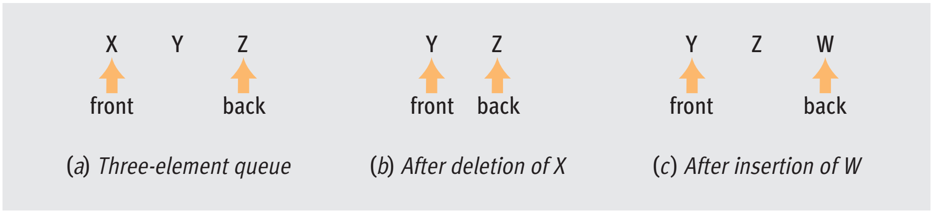
Dijkstra was also a prolific writer. He penned a huge number of technical reports called "EWDs" (his initials) that circulated widely in the computer science community and were discussed and debated at great length. These documents could cover almost any topic, from the detailed analysis of complex algorithms to informal diaries of trips he had taken. As of last count, more than 1300 EWDs were available on the Web at *www.cs.utexas.edu/users/EWD/*.

In 1972, Dijkstra won the prestigious A. M. Turing Award from the Association for Computing Machinery for his many important contributions to the field of computer science.

6.4 Queues

A **queue** is a First In First Out (FIFO) linear data structure in which all retrievals and deletions are made at one end, called the **front** or **head** of the queue, and all insertions are made at the other end, called the **back** or **tail** of the queue. Thus, while a stack uses only one end, the top, for all operations, the queue uses both ends—the front for taking things out and the back for putting new things in.

This situation is diagrammed in Figure 6-32(a), which shows a queue containing the elements X, Y, and Z added in just that order; X is the element at the front of the queue, while Z is the element at the back. The only value we can access—in other words, retrieve, modify, or delete—is X, the object at the front. If we delete it, we have the situation in Figure 6-32(b) in which Y has moved to the front. If we add a new element, W, to the queue of Figure 6-32(b), it is placed at the back of the queue, producing the situation shown in Figure 6-32(c).

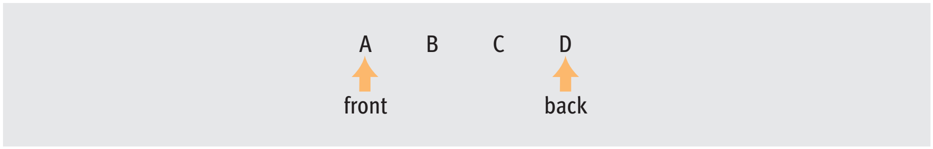


[FIGURE 6-32] The queue data structure

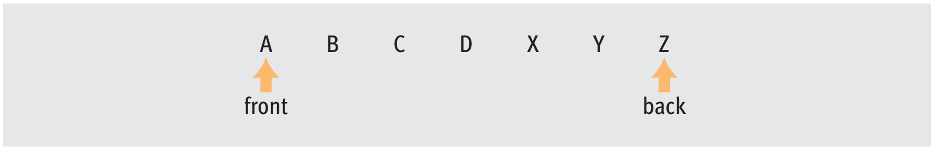
6.4.1 Operations on Queues

The operation of inserting a new item at the back of a queue is called **enqueue**, and its behavior is shown in Figure 6-32(c). The `enqueue(data)` method is a mutator that produces a queue in which the element `data` has been added at the back, and the queue length is one greater than its previous value. If the queue is full when this method is invoked, then its behavior is undefined.

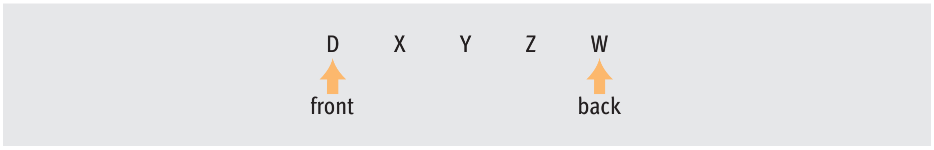
For example, start with the following four-element queue:



and execute the three operations `enqueue('X')`, `enqueue('Y')`, and `enqueue('Z')` in that order. We end up with the following:



The process of removing the value at the front of the queue is termed **dequeue** (pronounced *dee-cue*), and its behavior is diagrammed in Figure 6-32(b). `Dequeue()` is a mutator that produces a queue in which the item at the front has been removed, and the queue length is one less than its previous value. If the queue is empty when this method is invoked, its behavior is undefined. If we start with the previous seven-element queue and perform the operations `dequeue()`, `dequeue()`, `enqueue('W')`, and `dequeue()`, we end up with the following queue structure:



The `enqueue()` and `dequeue()`operations are the most important mutator operations on queues.

There are two important accessor methods for queues. The method `front()` returns the element that is currently at the front of the queue, but like the stack method `top()` discussed in the previous section, `front()` does not alter the queue. This method allows us to inspect the object at the front without removing it. Invoking `front()` on the queue of Figure 6-32(c) returns the value Y. Similarly, `back()` allows us to inspect the last object in the queue without deleting it. Invoking `back()` on the queue of Figure 6-32(c) returns W.

The two predicates `full()` and `empty()` operate exactly as their counterparts in the stack classes of Figures 6-29 and 6-31. The method `full()` returns true if the queue is full—in other words, an attempt to enqueue a new item results in a fatal error. If the queue is not full, the method returns false. The `empty()` method returns true if the queue is currently empty, and false otherwise.

Like the stack structure of the previous section, the queue is a very important data structure in computer science. The FIFO characteristic of a queue accurately models the behavior of a waiting line in which newly arriving objects are placed at the back, and the object at the front of the line is the next to be served. Thus, a queue is useful whenever a time-ordered set of objects is waiting for service and the service policy is first come, first served. This includes such common computing applications as the following:

- Processes waiting to obtain a processor on which to execute
- Memory requests waiting for resources from the memory manager
- Network messages waiting in line to use the capabilities of a transmission link
- Disk requests waiting for service from a disk drive

When a request is made in any of these cases, the request object *r* is placed at the end of the waiting line *q*, assuming that all requests are treated equally. (In the next section we learn how to handle prioritized requests.)

```
if (! q.full())
      q.enqueue(r); // enqueue request object r
else
      "the request must be discarded because the
         waiting line is full"
```

When a server has finished serving a request, it checks whether there are any more requests in line. If there are, the server removes the first request and begins serving it.

```
if ( ! q.empty())
     Request r = (Request) q.first();
     q.dequeue();
     "begin serving request r"
else
     "go idle until a new request arrives"
```

Figure 6-33 shows a `Queue` interface that contains the six methods described earlier in this section. Although these six are standard and almost universally included in every queue class, other operations are certainly possible. For example, we might want to include a method `size()` that returns the number of elements in the queue. We could also choose to combine the functions of `dequeue()` and `front()` to produce a single method that both removes and returns the first item in line. Whether this or other operations are worthwhile to include depends on how the queue is used.

```
/**
 * An interface for a queue
 */
public interface Queue<T> {

    /**
     * Places the given object at the back of the queue
     *
     * Preconditions:
     *      The queue is not full.
     * Postconditions:
     *      Data has been added to the back of the queue.
     *      The size of the queue has increased by one.
     *      No other structure of the queue has changed.
     *
     * @param data the object to place in the queue.
     *
     */
    public void enqueue( T data );

    /**
     *
     * Removes the element at the front of the queue.
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The size of the queue has decreased by one.
     *      The first item of the queue has been removed.
     *      No other structure of the queue has changed.
     */
    public void dequeue();

    /**
     * Return the element at the front of the queue
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return the element currently at the front of the queue.
     */
```

continued

```
    public T front();

    /**
     * Return the element at the back of the queue
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return the element at the back of the queue.
     */
    public T back();

    /**
     * Determine if the queue is empty.
     *
     * Preconditions:
     *      None
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return true if the queue is empty and false otherwise.
     */
    public boolean empty();

    /**
     * Determine if the queue is full.
     *
     * Preconditions:
     *      None
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return true if the queue is full and false otherwise.
     */
    public boolean full();
}
```

[FIGURE 6-33] Queue interface

6.4.2 Implementation of a Queue

6.4.2.1 An Array-Based Implementation

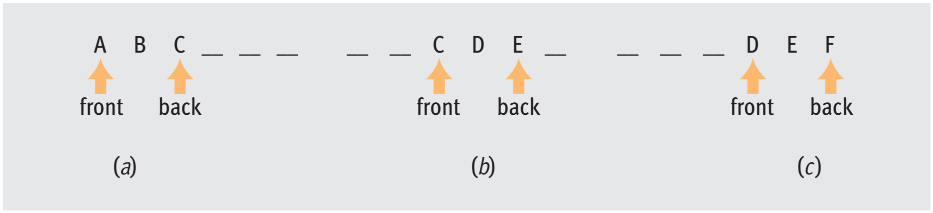
We can use a one-dimensional array to implement a queue, but there is one small problem. If we start with the six-element array and the queue *q* shown in Figure 6-34(a), and carry out the following four operations:

```
    q.dequeue();
    q.dequeue();
    q.enqueue('D');
    q.enqueue('E');
```

we end up with the condition shown in Figure 6-34(b). The elements in the queue have "slid" to the right. This is called the "inchworm effect" because we are removing elements from one end but adding them to the other. If we now perform the following two operations:

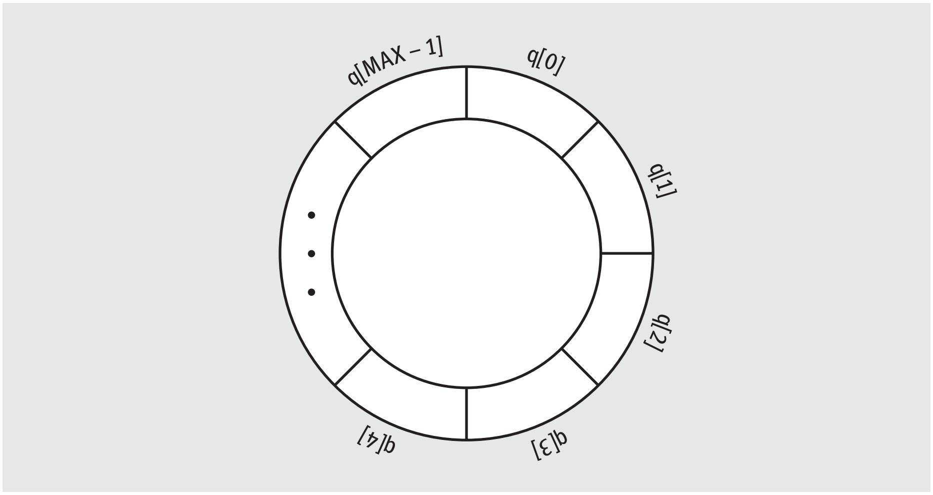
```
    q.dequeue();
    q.enqueue('F');
```

we end up with the queue in Figure 6-34(c), where we have reached the end of the array even though there are still three empty slots at the front.



[FIGURE 6-34] Example of queue operations

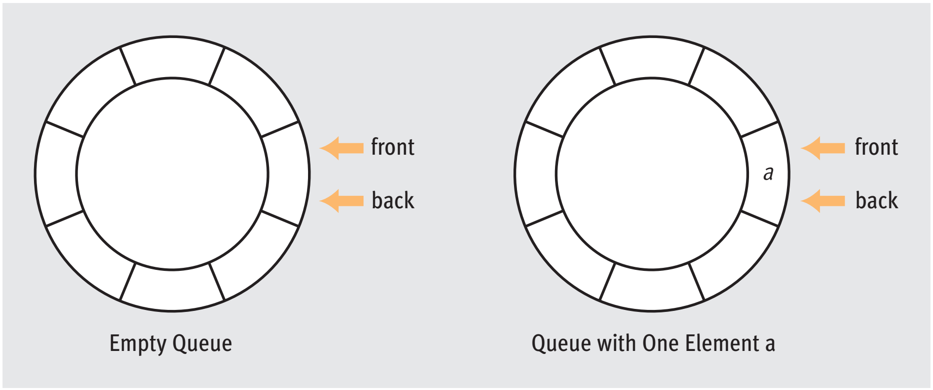
How can we solve this problem? A rather simple solution is not to view the array as a structure indexed linearly from 0 to MAX − 1, but as a **circular array**, or **ring**, in which the first element of the array, q[0], immediately follows the last element of the array, q[MAX − 1], as shown in Figure 6-35.



[FIGURE 6-35] A circular array

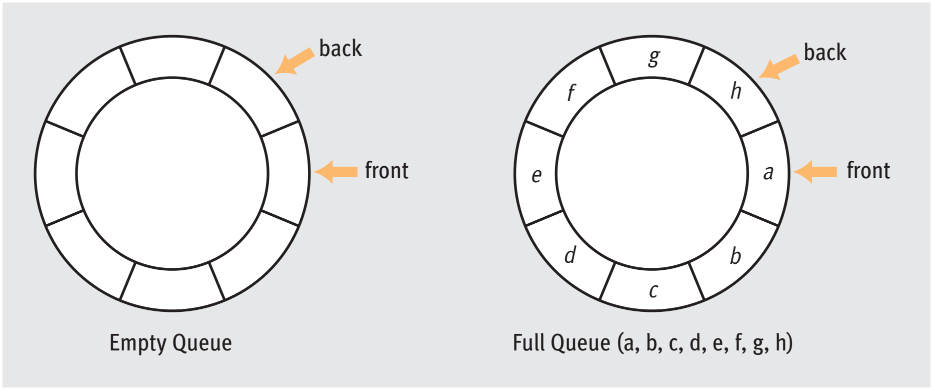
We do this by incrementing our array index modulo the array size MAX. Then the array element accessed after q[MAX − 1] is q[0], because [(MAX − 1) + 1] % MAX = 0.

However, this solution leads to another problem—detecting the difference between an empty queue and at least one other queue that is not empty. For example, if we initialize the pointers of an empty queue so that both front and back are 0 (or any integer value between 0 and MAX − 1), we cannot distinguish between a queue with no elements and a queue with exactly one element. When a single element is stored in location 0 of the array, both the front and back pointers are also set to 0, as shown in Figure 6-36a.



[FIGURE 6-36a] Trying to distinguish an empty queue from a queue with one element

We might instead choose to initialize the queue so `back = front - 1`. If we do, however, we will be unable to tell the difference between an empty queue and a full queue. When the queue is full, the front pointer has some value n, $0 \le n \le$ MAX $-$ 1, and the back pointer will have the value $(n - 1)$ % MAX—exactly the same condition as the empty queue. This situation is diagrammed in Figure 6-36b.



[FIGURE 6-36b] Trying to distinguish an empty queue from a full queue

In fact, no matter how you initialize the front and back pointers, their initial empty state is always indistinguishable from at least one nonempty state. This is because there are $(n + 1)$ distinct numbers of elements that can be stored in a queue of size n, namely the $(n + 1)$ values 0, 1, 2, 3, … , n. However, once you have fixed the location of one of the pointers, such as `front`, there are only n possible positions to assign to the other pointer. Thus, it is impossible to represent $(n + 1)$ distinct queue states with only n possible locations for front and back.

There are a number of solutions to this problem. The easiest solution is to use an auxiliary state variable called `size` that counts the number of elements in the queue, as shown in Figure 6-37. By checking this variable, we can determine whether a queue is empty or full. Of course, all methods that alter the number of elements in the queue must correctly reset this state variable.

An `ArrayQueue` class that implements the `Queue` interface of Figure 6-33 using a circular array is shown in Figure 6-37.

```
/**
 * An implementation of a queue using circular arrays.  Javadoc
 * comments for methods specified in the Queue interface have been
 * omitted.
 */
```

continued

```
public class ArrayQueue<T> implements Queue<T> {

    // The size of the array
    public static final int MAX_SIZE = 100;

    private T theQueue[];       // The queue
    private int front, back;    // Front and back positions
    private int size;           // Number of elements in the queue

    /**
     * Create a new queue.
     */
    public ArrayQueue() {
        // Create the array that will be the queue
        theQueue = (T[])new Object[ MAX_SIZE ];

        // Initialize state
        size = 0;
        front = 0;
        back = -1;
    }

    public void enqueue( T data ) {
        // Determine where the new element will be placed
        back = ( back + 1 ) % MAX_SIZE;

        // Add the element
        theQueue[ back ] = data;

        // One more thing in the queue
        size = size + 1;
    }

    public void dequeue() {
        // Eliminate reference for garbage collection
        theQueue[ front ] = null;

        // Remove the first element by incrementing the location
        // of the first element
        front = ( front + 1 ) % MAX_SIZE;

        // There is now one less element in the queue
        size = size - 1;
    }

    public T front() {
        return theQueue[ front ];
    }
```

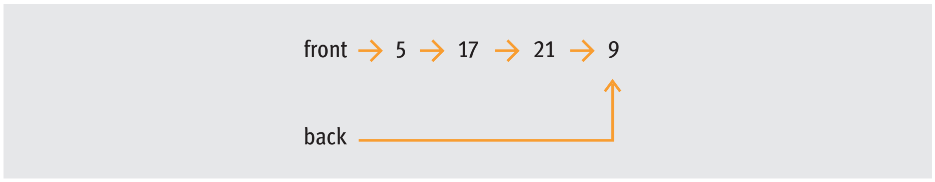
continued

```
        public T back() {
            return theQueue[ back ];
        }

        public boolean empty() {
            return size == 0;
        }

        public boolean full() {
            return size == MAX_SIZE;
        }
    }
```

[FIGURE 6-37] Queue class using a circular array implementation

6.4.2.2 A Linked List-Based Implementation

Rather than using an array, we can implement our queue as a linked list, as shown in Figure 6-38.



[FIGURE 6-38] Linked list view of a queue

It may actually be easier to implement a queue using the representation in Figure 6-38 instead of the array representation in the previous section. First, we do not have to worry about the inchworm effect diagrammed in Figure 6-34(c). In addition, there is no problem distinguishing between empty and full queues because a queue can never be full with a linked list. Therefore, applications that use queues may prefer the linked list shown in Figure 6-38.

The linked list implementation of a queue is similar to the linked list implementation of a stack, and the declarations to create them are virtually identical. The only major difference is the inclusion of a `back` pointer that references the last element in the queue. This state variable was not needed in the stack.

The default constructor creates an empty queue by setting the `front` and `back` pointers to **null** and the queue size to 0.

The method `enqueue(data)`, which adds a new object `data` at the back of the queue, is similar to the `add(element)` operation on lists discussed in Section 6.2.3.2. The existence of the `back` pointer allows the `enqueue()` method to be completed in O(1) time. Without it, we have to iterate through the queue to locate the end, where the enqueue takes place, an O(n) operation.

The `dequeue()` method, which deletes the element at the front of the queue, is virtually identical to the `pop()` operation on stacks, which also removes the first element in the collection. The only difference is ensuring that the `back` pointer is correctly reset if we delete the last item. An attempt to dequeue from an empty queue is a fatal error.

A complete implementation of the `LinkedQueue` class that implements the `Queue` interface of Figure 6-33 is shown in Figure 6-39.

```java
/**
 * A linked list implementation of a queue.  The Javadoc comments
 * for the methods specified in the Queue interface have been omitted.
 */
public class LinkedQueue<T> implements Queue<T> {

    protected LinkedNode<T> front;  // The first node
    protected LinkedNode<T> back;   // The last node
    protected int size;             // The number of elements

    /**
     * Constructs a LinkedQueue
     */
    public LinkedQueue() {
        // Initialize state
        size = 0;
        front = null;
        back = null;
    }

    public void enqueue( T data ) {
        // Initialize a new node
        LinkedNode<T> newNode = new LinkedNode<T>( data, null );

        // Is the queue empty?
        if ( empty() ) {
            // The new node is also the front of the queue
            front = newNode;
        }
        else {
            // Last node should refer to the new element
            back.setNext( newNode );
        }
```

continued

```
            // Back should refer to the new node
            back = newNode;

            // One more element in the queue
            size = size + 1;
        }

        public void dequeue() {
            // Remove the front element
            front = front.getNext();

            // Queue is empty; back should not refer to anything
            if (front == null) {
                back = null;
            }

            // One less item in the queue
            size = size - 1;
        }

        public T front()  {
            return front.getData();
        }

        public T back() {
            return back.getData();
        }

        public boolean empty() {
            return size == 0;
        }

        public boolean full() {
            return false;  // Linked structures are never full
        }
    }
```

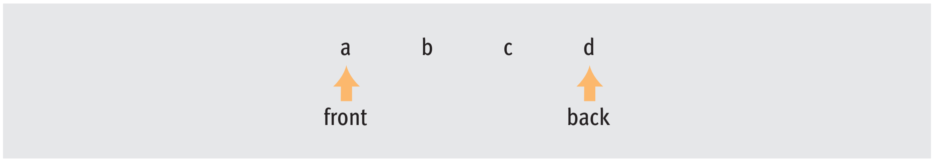
[FIGURE 6-39] Linked list implementation of the Queue interface

The Java Collection Framework contains a `Queue` interface that is similar to the interface in Figure 6-33. It includes methods to enqueue a new element at the front of the queue (called `offer(e)` in the interface), dequeue an element from the back of the queue (`remove()`), and examine the front of the queue (`element()`). This interface is implemented by a number of Java classes, including `LinkedList` and `ArrayBlockingQueue`. Thus, the Java Collection Framework offers a choice of implementations of the `Queue` data structure.

6.4.3 Queue Variations—the Deque and Priority Queue

We conclude this chapter by explaining two interesting and useful variations on the queue structure we just described. They are the deque and the priority queue.

The word **deque** (pronounced *deck*) stands for *d*ouble-*e*nded *que*ue. With a deque, you can make insertions and deletions at *either* end of the queue; that is, either at the front or the back. So, while we are still restricted to performing operations at only the two ends of the queue, we allow both of our mutator methods—`enqueue` and `dequeue`—to be done at either end. For example, given the following four-element deque:



we can add a new item *e* either at the back, as in a regular queue, producing *a b c d e*, or at the front, producing *e a b c d*. Similarly, we are permitted to remove either the front item, producing *b c d*, or the back item, producing *a b c*. Thus, the front and the back of a deque are functionally equivalent.

Another interesting variation is the **priority queue**. If a queue can be thought of as a data structure that models a waiting line, then a priority queue can be viewed as a data structure that models a waiting line with "cuts." That is, we allow objects to get into line anywhere based on a numerical value called a **priority**, which is a measure of the object's importance. However, we still are restricted to removing only the first object in front of the line, because it has the highest priority.

The `dequeue()`, `first()`, `last()`, `full()`, and `empty()` operations on priority queues all behave identically to their counterparts in the regular queue described in Section 6.4.1. Only the `enqueue` operation needs to be modified. It now must include a parameter that specifies the priority value, or importance, of the following item:

```
public void enqueue(T item, int priority);
```

Instead of adding this new item to the back of the queue, as we did previously, we add it to the queue in priority order—that is, behind all other items with equal or higher priority but ahead of all items with lower priority.

In a sense, a regular queue, which is ordered by time, can be considered a priority queue in which the item's priority is the time of its insertion. With a priority queue, we are allowed to use values other than time to order elements in the collection. For example, assume you have the following priority queue in which the pairs represent an element and its associated priority (with higher numbers representing higher priority):

$$\begin{tikzcd}
(a, 9) \arrow[r] & (b, 7) \arrow[r] & (c, 7) \arrow[r] & (d, 5) \arrow[r] & (e, 2) \arrow[r] & (f, 1) \\
\text{front} \arrow[u] & & & & & \text{back} \arrow[u]
\end{tikzcd}$$

Invoking the method `enqueue('g', 6)` inserts *g* between the existing elements *c* and *d*, producing the following:

$$\begin{tikzcd}
(a, 9) \arrow[r] & (b, 7) \arrow[r] & (c, 7) \arrow[r] & (g, 6) \arrow[r] & (d, 5) \arrow[r] & (e, 2) \arrow[r] & (f, 1) \\
\text{front} \arrow[u] & & & & & & \text{back} \arrow[u]
\end{tikzcd}$$

Because a new item may be inserted anywhere, depending on its priority level, the `enqueue` operation for priority queues behaves much like the insert operation on lists. However, the operation `dequeue()` still removes only the front element, the value *a* in the priority queue shown previously, because the object at the front has the highest priority. Similarly, `first()` and `last()` still return *a* and *f*, and `empty()` and `full()` still both return false. With the exception of `enqueue`, all operations behave identically when applied to a priority queue.

Priority queues are widely used. For example, when we insert processes into a waiting line for access to a processor, we may not want to treat them all equally. Instead, we may want to give operating system processes, like the garbage collector, memory manager, or disk scheduler, a higher priority than user processes. This would guarantee that important system routines receive a higher level of service. Similarly, we might want to give messages that control network operation higher priority to the transmission line than user e-mails or Web pages. This would ensure that the network is being managed in a timely manner.

A complete linked list implementation of a priority queue is shown in Figure 6-40. The `enqueue` operation now requires O(n) time because we must, in the worst case, search the entire queue to locate the proper insertion spot. This makes the complexity of the `enqueue` operation on priority queues significantly slower than the same operation on queues, which is O(1). In the next chapter, we introduce a new data structure that allows us to insert an object into a priority queue in O(log n), logarithmic time, rather than linear time, which is a significant improvement.

```
/**
 * A linkable-node class suitable for building prioritized linked data
 * structures such as lists, stacks, and queues.
 */
public class PrioritizedLinkedNode<T> {
    private int priority;                      // This node's priority
    private T data;                            // The data in this node
    private PrioritizedLinkedNode<T> next;  // The next node

    /**
     * Construct a node given the info and next references.
     *
     * @param newData the data to be associated with this node.
     * @param newNext a reference to the next node in the list.
     */
    public PrioritizedLinkedNode( T newData, int p ) {
        data = newData;
        priority = p;
        next = null;
    }

    /**
     * Return the data stored in this node.
     *
     * Preconditions:
     *   None.
     * Postconditions:
     *   The node is unchanged.
     *
     * @return a reference to the data stored in this node.
     */
    public T getData() {
        return data;
    }
    /**
     * Return a reference to the next node in the data structure.
     *
     * Preconditions:
     *   None
     * Postconditions:
     *   The node is unchanged.
     *
     * @return a reference to the next node in the list, or null if
     *         this node has no successor.
     */
    public PrioritizedLinkedNode<T> getNext() {
        return next;
```

continued

```
}

/**
 * Set the data associated with this node.
 *
 * Preconditions:
 *   None
 * Postconditions:
 *   The data associated with this node has been changed.
 *
 * @param newData the data to be associated with this node.
 */
public void setData( T newData ) {
  data = newData;
}

/**
 * Set the reference to the next node.
 *
 * Preconditions:
 *   None.
 * Postconditions:
 *   The reference to the next node has been changed.
 *
 * @param newNext the reference to the next node.
 */
public void setNext( PrioritizedLinkedNode<T> newNext ) {
    next = newNext;
}

/**
 * setPriority() changes the priority of this node.
 *
 * Preconditions:
 *      None
 * Postconditions:
 *      Node's priority has been updated.
 *
 * @param p this datum's priority
 */
public void setPriority( int p ) {
    priority = p;
}

/**
 * getPriority() returns the priority of this node
 *
```

continued

```
     * Preconditions:
     *      None
     * Postconditions:
     *      Node is unchanged.
     *
     * @return priority of this node
     */
    public int getPriority() {
        return priority;
    }
}

/**
 * A simple prioritized linked queue.  This class assumes that
 * priorities are nonnegative integers, with 0 being the highest
 * priority.
 */
public class LinkedPriorityQueue<T> implements Queue<T> {
    /**
     * Minimum priority.
     */
    public static final int MIN_PRIORITY = 0;

    private PrioritizedLinkedNode<T> front;  // Front of the queue
    private PrioritizedLinkedNode<T> back;   // Back of the queue
    private int size;                        // Number of elements

    /**
     *  Create a new priority queue.
     */
    public LinkedPriorityQueue() {
        // Initialize state
        front = null;
        back = null;
        size = 0;
    }

    /**
     * Remove the item at the front of the queue.
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The size of the queue has decreased by one.
     *      The first item of the queue has been removed.
     *      No other structure of the queue has changed.
     */
```

continued

```
    public void dequeue() {
        // Remove the front element
        front = front.getNext();

        // Queue is empty; back should not refer to anything.
        if (front == null) {
            back = null;
        }

        // One less item in the queue
        size = size - 1;
    }

    /**
     * Return the item at the front of the queue.
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return the item at the front of the queue.
     */
    public T front()  {
        return front.getData();
    }

    /**
     * Return the item at the back of the queue.
     *
     * Preconditions:
     *      The queue is not empty.
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return the item at the back of the queue.
     */
    public T back() {
        return back.getData();
    }

    /**
     * Determine if the queue is empty
     *
     * Preconditions:
     *      None
```

continued

```
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return true if the queue is empty and false otherwise.
     */
    public boolean empty() {
        return size == 0;
    }

    /**
     * Determine if the queue is full.
     *
     * Preconditions:
     *      None
     * Postconditions:
     *      The queue is unchanged.
     *
     * @return true if the queue is full and false otherwise.
     */
    public boolean full() {
        // Linked structures are never full
        return false;
    }

    /**
     * Add an element to the queue.  The item is added with
     * minimum priority.  This method must be provided to
     * satisfy the Queue interface.
     *
     * Preconditions:
     *      The queue is not full.
     * Postconditions:
     *      The value V has been added to the back of the queue.
     *      The size of the queue has increased by one.
     *      No other structure of the queue has changed.
     *
     * @param data  Object to put into the queue
     */
    public void enqueue( T data ) {
        // Let the other enqueue method do the work.
        enqueue( data, 0 );
    }

    /*
     * Add an element to the queue with the given priority.
     *
```

continued

```
     * Preconditions:
     *      The queue is not full.
     * Postconditions:
     *      The value V has been added to the back of the queue.
     *      The size of the queue has increased by one.
     *      No other structure of the queue has changed.
     *
     * @param data  Object to put into the queue
     * @param priority the priority of this item.
     */
    public void enqueue( T data, int priority ) {
        PrioritizedLinkedNode<T> cur = front;  // The current node
        PrioritizedLinkedNode<T> prev = null;  // The last node visited

        // The node this item will be placed in
        PrioritizedLinkedNode<T> newNode =
            new PrioritizedLinkedNode<T>( data, priority );

        // Adding to an empty queue is easy
        if ( back == null ) {
            front = newNode;
            back = newNode;
        }
        else {
            // Step through the queue looking for the first node with
            // a priority less than the priority of the new node.
            // When the loop has terminated, cur will refer to the
            // node after the new node in the queue and prev will
            // refer to the node before the new node.
            while ( cur != null && cur.getPriority() >= priority ) {
                prev = cur;
                cur = cur.getNext();
            }

            if ( cur == null ) {
                // Item must be added at the end of the queue
                back.setNext( newNode );
                back = newNode;
            }
            else if ( prev == null ) {
                // Item must be added to the front of the queue
                newNode.setNext( front );
                front = newNode;
            }
            else {
                // Insert between prev and cur
                prev.setNext( newNode );
```

continued

```
                    newNode.setNext( cur );
                }
            }

            // One more item in the queue
            size = size + 1;
        }
    }
```

[FIGURE 6-40] Implementation of a LinkedPriorityQueue

6.5 Summary

The last two sections of this chapter continued our investigation of linear data structures, this time examining structures that restrict where we may make insertions, deletions, and retrievals. Thus, they are less general and flexible than the list structures discussed at the beginning of the chapter.

The stack allows operations at only one end, the location called the `top` of the stack. The queue allows insertions at one end, called the `back` of the queue, while deletions and retrievals are permitted at the other end, called the `front` of the queue. These are the two most important restricted linear structures, and they are widely used throughout computer science. This chapter also introduced two queue variations called the deque and the priority queue.

Figure 6-41 summarizes the behavior of the five linear data structures presented in this chapter.

STRUCTURE	INSERTIONS	DELETIONS, RETRIEVALS
List	Anywhere	Anywhere
Priority queue	Anywhere (by priority)	Front
Deque	Front or back	Front or back
Queue	Back	Front
Stack	Front (called the top)	Front

[FIGURE 6-41] Summary of the behavior of linear data structures

The next chapter begins our investigation of an important new class of data structures that have more complex (1:many) relationships between their elements. This classification is called the hierarchical data structures.

EXERCISES

1 Using the taxonomy introduced in Section 6.1, explain which of the four data structure classifications best fits each of the following collections:

   a A character string such as *ABCDEF*

   b The organization chart of a corporation

   c A line of people waiting to get into a movie

   d A road map

   e The names and identification numbers of students in a computer science class

2 Review the `List` interface in Figure 6-6, then propose additional positioning, insertion, deletion, or retrieval operations that you think might be useful in this interface. For each operation you propose, give its pre- and postconditions and its calling sequence.

3 Assume that you are using the array-based implementation of a list in Figure 6-9. Write the Boolean method `find` that attempts to locate a specific object in the list. The calling sequence for `find` is as follows:

```
/*  Preconditions:  None
    Postconditions:  Find searches through a list
       looking for the first occurrence
       of an item. If the item is found, the
       method returns the position number in
       the list where the item occurred, and
       resets cursor to point to this item. If
       the item does not occur anywhere in List,
       the method returns a -1 and cursor is
       unaffected   */
public int find(Object item)
```

What is the complexity of your method?

4 Assume that you are using the array-based implementation of a list in Figure 6-9. Write the class method `concatenate` that merges two separate lists into a single list. The calling sequence for `concatenate` is:

```
/*  Precondition:   L1 is a list of length m, m >= 0; L2
        is a list of length n, n >= 0
    Postcondition:  L1 is a list of length (m + n), with
        all the elements of L2 added after the elements
        of list L1.  L2 is the empty list.  */
public static void concatenate(List L1, List L2)
```

What is the complexity of your method?

5 Here is the array implementation of a list. Diagram the logical structure of the list that is represented by this implementation:

Index	Info	
0	0	
1	−1	
2	9	
3	15	last = 4
4	12	cursor = 1

6 Why did we include both `add(element)` and `addFirst(element)` methods in the `List` interface of Figure 6-6? Explain the problems we would encounter if we omitted the `addFirst` method from the interface.

7 a Assume that you are using the reference-based implementation of a list shown in Figure 6-16. Would the following two operations be constant time O(1) or linear time O(n) operations?

- `swapNext()`—Swap the contents of the `data` field of the node referenced by the cursor with the `data` field of the successor node.
- `swapPrev()`—Swap the contents of the `data` field of the node referenced by the cursor with the `data` field of the predecessor node.

b Implement these two instance methods using the declarations in Figures 6-12 and 6-13. For each method, include the pre- and postconditions in the comments.

8 You are given a singly linked list L of n integer values I_k, $k = 0, \ldots, n - 1$.

$L \rightarrow 12 \rightarrow 15 \rightarrow 35 \rightarrow 42 \rightarrow 51$

The list is implemented using the declarations in Figures 6-12 and 6-13. Write the Boolean instance method `inOrder()` that compares adjacent values and ensures that the second value is always greater than or equal to the first. That is, $I_k \leq I_{k+1}$, $k = 0, 1, 2, \ldots, n - 2$. If all pairs of nodes in the list meet this condition, return true; otherwise, return false. Be careful that you correctly handle all special cases.

9 Using the declarations in Figures 6-12 and 6-13, write an instance method of the class `LinkedList` to reverse a list. That is, if L has the initial value:

$L \rightarrow 12 \rightarrow 15 \rightarrow 35 \rightarrow 42 \rightarrow 51$

then a call to `reverse()` should produce the new list:

$L \rightarrow 51 \rightarrow 42 \rightarrow 35 \rightarrow 15 \rightarrow 12$

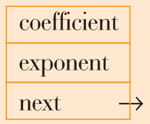
If the list is originally empty, then the method should do nothing and return. If the complexity of your algorithm is $O(n^2)$, think about how you can get it to run in $O(n)$ time instead.

10 A classic use of linked lists is to represent and manipulate polynomials. A polynomial is a mathematical formula of the following form:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

This formula can be represented by a list using the following node structure. Each node in the linked list would store the information about one nonzero term of the polynomial.

Node:



This would allow us to efficiently represent polynomials of an arbitrarily large degree. For example, the following polynomial:

$$5x^{20} + 2x - 8$$

is a 20th degree polynomial, but it has only three nonzero terms. Therefore, its linked list representation has only three nodes and is represented as follows:

$$L \rightarrow (5, 20) \rightarrow (2, 1) \rightarrow (-8, 0)$$

Design and build a `Polynomial` class that implements polynomials using the `LinkedList` representation of Figure 6-16. Your polynomial class should be able to read and write polynomials from the standard input file and add and subtract polynomials. Try out your `Polynomial` class on the following problems:

- **a** Input the following polynomial: $A = x^5 + x^3 + x^2 - 5$.
- **b** Input the following polynomial: $B = 3x^4 + 2x^3 - x^2 + 6$.
- **c** Write out both A and B.
- **d** Compute and print the value of $A + B$.
- **e** Compute and print the value of $A - B$.

11 Review the doubly linked list class of Figure 6-21, then add an instance method called `back(n)`, which moves you backward n nodes in the list beginning from the current position indicator. That is, move the cursor to the nth predecessor of the current node. If there are not n predecessors of the current node (in other words, you encounter a **null** pointer), then reset the cursor to **null**.

12 Repeat Exercise 11, but this time assume that the list is a circular doubly linked list of the type shown in Figure 6-24. Discuss how the use of a circular doubly linked list did or did not simplify the implementation of this instance method.

13 Given the following stack S:

stack S

```
C  ⇐ top
B
A
```

Show exactly what the stack will look like after each of the following sequences of operations. (Assume each sequence begins from the above state.)

a
```
S.push('D')
```

b
```
S.push('D')
S.push('E')
```

c
```
S.pop()
S.pop()

S.push('D')
S.pop()
S.push('E')
```

d
```
S.pop()
S.pop()

S.pop()
S.pop()
```

e
```
S.push('D')
S.pop()

S.push('E')
```

14 Add the following two instance methods to the `ArrayStack` class in Figure 6-29:

a	`remove(n)`	Remove the top *n* elements from the stack. If the stack does not contain at least *n* elements, leave it empty.
b	`size()`	Return the total number of elements contained in the stack. The contents of the stack are unchanged.

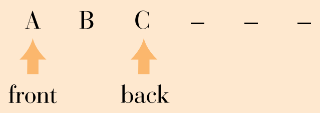
For each operation, provide the pre- and postconditions. Propose additional useful operations on the stack data structure.

15 Repeat Exercise 14, but this time use the `LinkedStack` class of Figure 6-31.

16 Change the behavior of the `pop()` method in Figure 6-29 so that it now combines the behaviors of both `pop()` and `top()`. That is, `pop` removes the top element of the stack *and* returns it. Which of these two approaches do you think represents a better design?

17 Propose other solutions than the one used in Section 6.4.2.1 to address the problem of distinguishing between a full and empty queue. Discuss whether you think your solution is better than the one proposed in the text—that is, using a size field that specifies how many nodes are contained in the queue.

18 Given the following six-element circular queue containing the three elements A, B, and C:



Show what state the queue would be in after each of the following series of operations. (Assume each one starts from the preceding conditions.)

a
```
dequeue()
dequeue()
enqueue(D)
enqueue(E)
```

b
```
enqueue(D)
enqueue(E)
dequeue()
```

c
```
enqueue(D)
enqueue(E)
enqueue(F)
```

19 Add the following three instance methods to the circular array-based `ArrayQueue` class shown in Figure 6-37:

a	`size()`	Returns the total number of elements in the queue
b	`remove(n)`	Removes the first *n* items from the front of the queue. If the queue does not contain at least *n* items, then leave it empty.
c	`cutsInLine(I, n)`	Inserts item *I* into position *n* of the queue, rather than at the end of the queue

20 Repeat Exercise 19, but this time use the linked list representation of a queue in Figure 6-39.

21 Design a `Deque` interface that includes all of the important methods carried out on the deque data structure described in Section 6.4. Then select an appropriate representation for a deque and build a class that implements this interface.

CHALLENGE WORK EXERCISE

Stacks are widely used in the design and implementation of compilers. For example, they are used to convert arithmetic expressions from **infix notation** to **postfix notation**. An infix expression is one in which operators are located between their operands. This is how we are accustomed to writing expressions in standard mathematical notation. In postfix notation, the operator immediately follows its operands.

Examples of infix expressions are:

*a*b*

*f*g−b*

*d/e*c+2*

d/e(c+2)*

The corresponding postfix expressions (assuming Java precedence rules) are:

*ab**

*fg*b−*

*de/c*2+*

*de/c2+**

In a postfix expression, a binary operator is applied to the two immediately preceding operands. A unary operator is applied to the one immediately preceding operand. Notice that the operands in a postfix expression occur in the same order as in the corresponding infix expression. Only the position of the operators is changed. A useful feature of postfix is that the order in which operators are applied is always unambiguous. That is, there is only a single interpretation of any postfix expression. This is not true of infix. For example, the meaning of the following infix expression:

a + b * c

is unclear. Does it mean $(a + b) * c$ or $a + (b * c)$? Without parentheses or additional rules, we cannot say. In Java, precedence rules dictate that multiplication is performed before addition; we say that multiplication "takes precedence" over addition. The corresponding postfix expression is $a\ b\ c\ *\ +$, and there is no ambiguity. It is explicit that the multiplication operator applies to the immediately two preceding operands b and c. The addition operation is then applied to the operand a and the result $(b * c)$. If we had meant the expression $(a + b) * c$, we would have written it in postfix notation as $a\ b + c\ *$. Now the $+$ operation is applied to the two operands a and b, and $*$ is performed on the two operands c and the result $(a + b)$.

Because of the existence of this unique interpretation, some compilers first convert arithmetic expressions from the standard infix notation to postfix to simplify code generation. To convert an infix expression to postfix, we must use a stack to hold operators that cannot be processed until their operands are available.

Assume that we are given as input the infix string $a + b * c - d$. This is equivalent to the parenthesized infix arithmetic expression $(a + (b * c)) - d$. To convert this to postfix format, we scan the input from left to right. If the next symbol in the input stream is an operand (for example, a), it is placed directly into the output stream without further processing.

```
Input: a + b * c – d     Output: a     opStack: empty
          ↑
```

If the next symbol in the input string is an operator (for example, $+$), then we compare the precedence of this operator to the one on top of a stack called opStack, short for *op*erator *stack,* which is initialized to empty at the start of the scan. If opStack is empty or if the precedence of the next operator in the input stream is greater than the precedence of the one on top of opStack, then we push the operator from the input string onto opStack and continue.

```
Input: a + b * c – d     Output: a     opStack: +
            ↑
```

Let's explain what is happening a little more intuitively. We are trying to locate the highest-precedence operation to determine which operation to perform next.

As long as the operators being scanned are increasing in precedence, we cannot do anything with our infix expression. Instead, we stack them and continue scanning the input string.

Input: $a + \underset{\uparrow}{b} * c - d$ Output: $a\ b$ opStack: $+$

Input: $a + b \underset{\uparrow}{*} c - d$ Output: $a\ b$ opStack: $+\ *$

Input: $a + b * \underset{\uparrow}{c} - d$ Output: $a\ b\ c$ opStack: $+\ *$

Input: $a + b * c \underset{\uparrow}{-} d$ Output: $a\ b\ c$ opStack: $+\ *$

At this point, we encounter the − operator, whose precedence is lower than *, the operator on top of the stack. We now pop operators from opStack until either (1) opStack is empty or (2) the precedence of the operator on top of opStack is strictly less than that of the current operator in the input string. In this example, we pop both the * and + operators and place them into the output stream:

Input: $a + b * c \underset{\uparrow}{-} d$ Output: $a\ b\ c\ *$ opStack: $+$

Input: $a + b * c \underset{\uparrow}{-} d$ Output: $a\ b\ c\ *\ +$ opStack: empty

This scanning process continues until we come to the end of the input. Then, any remaining operators on opStack are popped into the output and the algorithm terminates.

Input: $a + b * c \underset{\uparrow}{-} d$ Output: $a\ b\ c\ *\ +$ opStack: $-$

Input: $a + b * c - \underset{\uparrow}{d}$ Output: $a\ b\ c\ *\ +\ d$ opStack: $-$

Input: $a + b * c - d\ \underset{\uparrow}{}$ Output: $a\ b\ c\ *\ +\ d\ -$ opStack: empty

The algorithm has produced the expression $a\ b\ c\ *\ +\ d\ -$, the correct postfix representation of our original input.

In this example, a stack was the right structure to hold operators from the input stream because of the nature of the algorithm. Operators must be saved until we determine their proper location, and then they are placed into the output stream in the reverse order of their occurrence in the input. Again, this is a perfect match with the LIFO model of a stack.

Write a program for converting an infix expression (without parentheses) into its equivalent postfix expression. It should use the resources of the classes in this chapter that implement the `Stack` interface of Figure 6-26.