9 Lists

This chapter covers lists, the last of the three major linear ADTs discussed in the book, the other two being stacks and queues. Lists support a much wider range of operations than stacks and queues and, consequently, are both more widely used and more difficult to implement. To make sense of a list's profusion of fundamental operations, we classify them into three groups: index-, content-, and position-based operations. We present prototypes that illustrate the two most common list implementations: arrays and linked structures. The package `java.util` includes professional versions based on both approaches, and we examine these. The chapter's Case Study shows how to implement a common desktop tool, a personal to-do list.

9.1 Overview of Lists

A list supports manipulation of items at any point within a linear collection and is unlike stacks and queues, which restrict access only to the ends. Some common examples of lists include:

- A *recipe,* which is a list of instructions.
- A *string,* which is a list of characters.
- A *document,* which is a list of words.
- A *file,* which is a list of data blocks on disk.

In all these examples, order is critically important, and shuffling the items renders the collections meaningless; however, the items in a list are not necessarily sorted. Words in a dictionary and names in a phone book are examples of sorted lists, but the words in this paragraph equally form a list and are unsorted. In Chapter 12, we discuss sorted lists in detail. While the items in a list are always logically contiguous, they need not be physically contiguous. Array implementations of lists use physical position to represent logical order, but linked implementations do not.

Borrowing from the terminology of queues, we say that the first item in a list is at the head and the last is at the tail. Using this terminology, we can characterize a stack as a list in which manipulations are restricted to the head and a queue as a list in which insertions are made at the tail and removals from the head. If we relax these restrictions slightly and allow insertions and deletions to occur at both the head and the tail, we have what is called a deque or double-ended queue.

In a list, items retain position relative to each other over time, and additions and deletions affect predecessor/successor relationships only at the point of modification. Figure 9.1 shows how a list changes in response to a succession of operations. The operations are:

`add(o)`	which adds object o to a list's tail
`add(i, o)`	which inserts object o at the *i*th index in a list, where the first item is at index 0
`remove(o)`	which removes the first instance of object o from a list
`remove(i)`	which removes the object at the *i*th index in a list
`set(i, o)`	which replaces the object at the *i*th index with object o

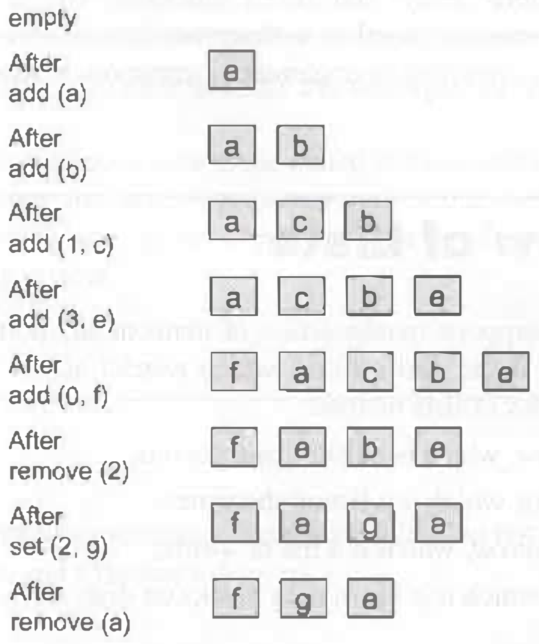

Figure 9.1 The states in the lifetime of a list

9.2 A List Prototype

The fundamental operations for stacks and queues are universally agreed on—push and pop for stacks and enqueue and dequeue for queues—but for lists, there are no such standards. For instance, the operation of putting a new item in a list is sometimes called "add" and sometimes "insert". However, if we look at most textbooks on data structures and at the list ADT provided in Java, we can discern several broad categories of operations, which we call index-based operations, content-based operations, and position-based operations. Before trying to write list prototypes, we present these categories, and to avoid confusion later, we proceed in a manner that is consistent with the approach taken in the interfaces `java.util.List` and `java.util.ListIterator`.

Index-based operations manipulate items at designated indices within a list and provide the convenience of random access. Suppose a list contains n items. Because a list is linearly ordered, we can unambiguously refer to an item in a list via its relative position from the head of the list using an index that runs from 0 to $n - 1$. Thus, the head is at index 0 and the tail is at index $n - 1$. Here are some fundamental index-based operations:

`add(i, o)`	which opens up a slot in the list at index `i` and inserts object `o` in this slot
`get(i)`	which returns the object at index `i`
`remove(i)`	which removes and returns the object at index `i`
`set(i, o)`	which replaces the object at index `i` with the object `o` and returns the original object

When viewed from this perspective, lists are sometimes called ***vectors*** or ***sequences***, and in their use of indices, they are reminiscent of arrays. However, an array is a concrete data type with a specific and unvarying implementation based on a single block of physical memory, whereas a list is an abstract data type that can be represented in a variety of different ways, among which are array implementations. In addition, a list has a much larger repertoire of basic operations than an array, even though all list operations can be mimicked by suitable sequences of array operations.

Content-based operations are based not on an index but on the content of a list. Most of these operations search for an object equal to a given object before taking further action. Here are some basic content-based operations:

`add(o)`	which adds object `o` at a list's tail
`contains(o)`	which returns `true` if a list contains an object equal to object `o`
`indexOf(o)`	which returns the index of the first instance of object `o` in a list
`remove(o)`	which removes the first instance of object `o` from a list and returns `true` if `o` is removed, else returns `false`

Position-based operations are performed relative to a currently established position within a list, and in `java.util`, they are provided via an extended iterator called a list iterator. A list iterator uses the underlying list as a backing collection, and the iterator's current position is always in one of three places:

1. just before the first item
2. between two adjacent items
3. just after the last item

Initially, when a list iterator is first instantiated, the position is immediately before the first item. From this position the user can either navigate to another position or modify the list in some way. Here are the navigational operations:

`hasNext()`	which returns `true` if there are any items following the current position
`next()`	which returns the next item and advances the position
`hasPrevious()`	which returns `true` if there are any items preceding the current position
`previous()`	which returns the previous item and moves the position backward
`nextIndex()`	which returns the index of the next item or –1 if none
`previousIndex()`	which returns the index of the previous item or –1 if none

Conspicuously absent from this roster of operations are ones for moving directly to either the beginning or end of a list. The authors of `java.util` omitted these operations for reasons best known to themselves. The lack of an operation for returning to the head of a list is not critical because instantiating a new list iterator has the same effect; however, a succession of next operations is the only way to move to a list's tail. We will endure this defect, but in the exercises at the end of Sections 9.4 and 9.6, we ask you to remedy the problem.

The remaining position-based operations are used to modify the backing list. These operations work at the currently established position in the list:

`add(o)`	which inserts object `o` at the current position
`remove()`	which removes the last item returned by next or previous
`set(o)`	which replaces the last item returned by next or previous

Although there are many list operations, our classification scheme reduces the potential confusion. Table 9.1 gives a recap.

Table 9.1

Summary of Basic List Operations

Index-Based	Content-Based	Position-Based
`add(i, o)`	`add(o)`	`hasNext()`
`get(i)`	`contains(o)`	`next()`
`remove(i)`	`indexOf(o)`	`hasPrevious()`
`set(i, o)`	`remove(o)`	`previous()`
		`nextIndex()`
		`previousIndex()`
		`add(o)`
		`remove()`
		`set(o)`

Based on the foregoing discussion of list operations, we now propose two list prototypes. Bear in mind that these are just prototypes and include only those operations needed to illustrate the basic issues involved in implementing lists. The first prototype, `ListPT`, includes several index-based and content-based operations. The second prototype, `ListIteratorPT`, contains position-based operations.

This split of the list ADT into two components is consistent with the design decisions made in `java.util`. The split has the major advantage that one can have several list iterators going at once over the same underlying list. However, there is an accompanying disadvantage. A list iterator can be invalidated if changes are made to the underlying list by any means other than through the list iterator in question. To deal with the problem, list iterators follow the same fail-fast policy as regular iterators. As soon as a list iterator detects an unexpected change in the backing list, it throws an exception. List iterators are created by sending the `listIterator` message to a list. This is analogous to the manner in which we create regular iterators for stacks and queues. Tables 9.2 and 9.3 formalize our definitions of `ListPT` and `ListIteratorPT`.

Table 9.2

The Interface for the List Prototype (ListPT)

Fundamental Methods

`void`	**`add(int i, Object o)`** Adds the object `o` to the list at index `i`. Throws an exception if the object `o` is `null` or the list is full or if `i` is out of range (`i < 0 || i > size()`).
`boolean`	**`contains(Object o)`** Returns `true` if the object `o` is in the list, else returns `false`.
`Object`	**`get(int i)`** Returns the object at index `i`. Throws an exception if `i` is out of range (`i < 0 || i >= size()`).
`int`	**`indexOf(Object o)`** Returns the index of the first object equal to object `o` or `-1` if there is none.

Continues

Table 9.2 *(Continued)*

The Interface for the List Prototype (ListPT)

`Object`	**`remove(int i)`** Removes and returns the object at index `i`. Throws an exception if `i` is out of range (`i < 0 or i >= size()`).
`Object`	**`set(int i, Object o)`** Returns the object at index `i` after replacing it with the object `o`. Throws an exception if the object `o` is `null` or if `i` is out of range (`i < 0 or i >= size()`).
Supporting Methods	
`boolean`	**`isEmpty()`** Returns `true` if this list contains no items.
`boolean`	**`isFull()`** Returns `true` if this list is full and can accept no more items.
`int`	**`size()`** Returns the number of items in this list.
General Methods	
`ListIteratorPT`	**`listIterator()`** Returns a list iterator over this list.

Table 9.3

The Interface for the List Iterator Prototype (ListIteratorPT)

Navigation Methods	
`boolean`	**`hasNext()`** Returns `true` if there are any items after the current position, else returns `false`.
`boolean`	**`hasPrevious()`** Returns `true` if there are any items preceding the current position, else returns `false`.
`Object`	**`next()`** Returns the item following the current position and advances the current position. Throws an exception if `hasNext` would return `false`.
`Object`	**`previous()`** Returns the item preceding the current position and moves the current position back. Throws an exception if `hasPrevious` would return `false`.
Modification Methods	
`void`	**`add(Object o)`** Inserts the object `o` at the current position. After insertion, the current position is located immediately after the newly inserted item. Throws an exception if the object `o` is `null` or the list is full.

Continues

Table 9.3 *(Continued)*

The Interface for the List Iterator Prototype (ListIteratorPT)	
`void`	`remove()` Removes the last object returned by `next` or `previous`. Throws an exception if `add` or `remove` has occurred since the last `next` or `previous`.
`void`	`set(Object o)` Replaces the last object returned by `next` or `previous` with object `o`. Throws an exception if `add` or `remove` has occurred since the last `next` or `previous`.

9.5 Three Applications of Lists

Lists have many applications. In this section, we look briefly at three: Java's object heap, organization of files on a disk, and implementation of other ADTs.

9.5.1 Heap Storage Management

In Section 7.5, we discussed one aspect of Java memory management, the call stack. Now we complete that discussion by showing how free space in the ***object heap*** can be managed using a linked list. Heap management schemes can have a significant impact on an application's overall performance, especially if the application creates and abandons many objects during the course of its execution. Implementers of Java virtual machines are therefore willing to spend a great deal of effort to organize the heap in the most efficient manner possible. Their elaborate solutions are beyond this book's scope, so we present a simplified scheme here.

In our scheme, contiguous blocks of free space on the heap are linked together in a ***free list.*** When an application instantiates a new object, the JVM searches the free list for the first block large enough to hold the object and returns any excess space to the free list. When the object is no longer needed, the garbage collector returns the object's space to the free list. The scheme as stated has two defects. Over time, large blocks on the free list become fragmented into many smaller blocks, and searching the free list for blocks of sufficient size can take O(n) running time, where n is the number of blocks in the list. To counteract fragmentation, the garbage collector periodically reorganizes the free list by recombining physically adjacent blocks. To reduce search time, multiple free lists can be used. For instance, if an object reference requires 4 bytes, then list 1 could consist of blocks of size 4; list 2, blocks of size 8; list 3, blocks of size 16; list 4, blocks of size 32; etc. The last list would contain all blocks over some designated size. In this scheme, space is always allocated in units of 4 bytes, and space for a new object is taken from the head of the first non-empty list containing blocks of sufficient size. Allocating space for a new object now takes O(1) time unless the object requires more space than is available in the first block of the last list. At that point, the last list must be searched, giving the operation a maximum running time of O(n), where n is the size of the last list.

In this discussion, we have completely ignored two difficult problems. The first problem has to do with deciding when to run the garbage collector. Running the garbage collector takes time away from the application, but not running it means the free lists are never replenished. The second problem concerns how the garbage collector identifies objects that are no longer referenced and, consequently, no longer needed.

9.5.2 Organization of Files on a Disk

A computer's file system has three major components—a directory of files, the files themselves, and free space. To understand how these work together to create a file system, we first consider a disk's physical format. Figure 9.2 shows the standard arrangement. The disk's surface is divided into concentric tracks, and each track is further subdivided into sectors. The numbers of these vary depending on the disk's capacity and physical size; however, all tracks contain the same number of sectors and all sectors contain the same number of bytes. For the sake of this discussion, let us suppose that a sector contains 8K bytes of data plus a few additional bytes reserved for a pointer. A sector is the smallest unit of information transferred to and from the disk, regardless of its actual size, and a pair of numbers (*t*, *s*) specifies a sector's location on the disk, where *t* is the track number and *s* the sector number. Figure 9.2 shows a disk with *n* tracks. The *k* sectors in track 0 are labeled from 0 to *k* – 1.

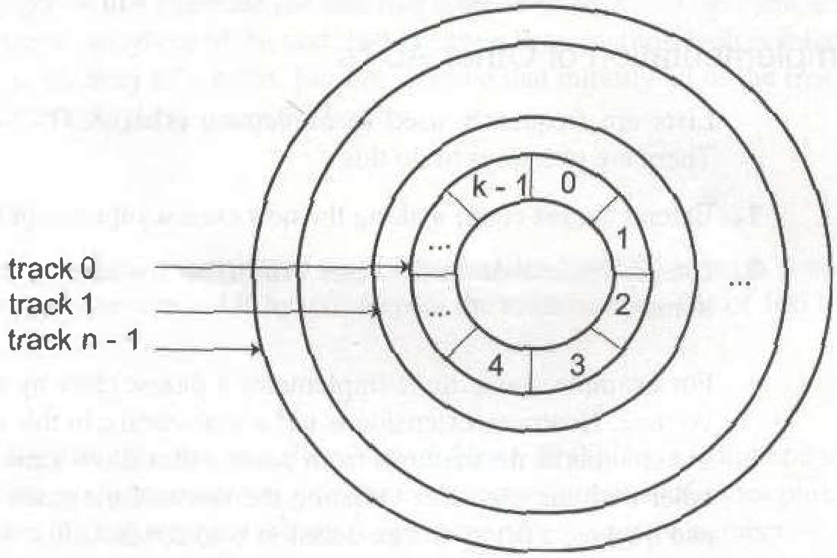

Figure 9.2 Tracks and sectors on the surface of a disk

A file system's directory is organized as a hierarchical collection. The directory's internal structure is not a suitable topic for this chapter; however, let us suppose that it occupies the first few tracks on the disk and contains an entry for each file. This entry bolds the file's name, creation date, size, etc. In addition, it holds the address of the sector containing the first bytes in the file. Depending on its size, a file might be completely contained within a single sector or it might span several. Usually, the last sector is only partially full, and no attempt is made to recover the unused space. The sectors that make up a file do not need to be physically adjacent because each sector, except the last, ends with a pointer to the sector containing the next portion of the file. Finally, sectors that are not in use are linked together in a free list. When new files are created, they are allocated space from this list, and when old files are deleted, their space is returned to the list.

Because all sectors are the same size and because space is allocated in sectors, a file system does not experience the same fragmentation problem encountered in Java's object heap. Nonetheless, there is still a difficulty. To transfer data to or from the disk, read/write heads must first be positioned to the correct track, the disk must rotate until the desired sector is under the heads, and then the transfer of data takes place. Of these three steps, the transfer of data takes the least time. Fortunately, data can be transferred to or from several adjacent sectors during a single rotation without the need to reposition the heads. Thus, a disk system's performance is optimized when multisector files are not scattered across the disk. However, over time, as files of varying sizes are created and destroyed, this sort of scattering becomes frequent, and the file system's performance degrades. As a countermeasure, file systems include a utility, run either automatically or at the explicit request of the user, that reorganizes the file system so that the sectors in each file are contiguous and have the same physical and logical order.

9.5.3 Implementation of Other ADTs

Lists are frequently used to implement other ADTs, such as stacks and queues. There are two ways to do this:

1. Extend the list class, making the new class a subclass of the list class.
2. Use an instance of the list class within the new class and let the list contain the data items.

For example, `java.util` implements a `Stack` class by extending a list class called `Vector`. However, extension is not a wise choice in this case because this version of `Stack` inherits the methods from `Vector` that allow users to access items at positions other than the top, thus violating the spirit of the stack ADT. In the case of stacks and queues, a better design decision is to contain a list within the stack or queue. In that case, all of the list operations are available to the implementer, but only the essential stack or queue operations are available to the user.

ADTs that use lists inherit their performance characteristics. For example, a stack that uses an array-based list has the performance characteristics of an array-based stack, whereas a stack that uses a link-based list has characteristics of a link-based stack.

The primary advantage of using a list ADT to implement another ADT is that coding becomes easier. Instead of operating on a concrete array or linked structure, the implementer of a stack need only call the appropriate list methods. The main disadvantage is that some additional space overhead can occur, for instance, when a stack is implemented using a list that in turn is based on doubly linked nodes. We saw in Chapter 7 that a singly linked list works well for the link-based implementation of a stack.

In Chapter 11 (trees), Chapter 13 (unordered collections), and Chapter 14 (graphs), we will see other situations in which lists can be used in the implementation of ADTs, although in the interest of optimizing performance we do not always do so.