Chapter 18

Searching

Contents

Prerequisites

Chapter 4 The Efficiency of Algorithms
Chapter 7 Recursion
Chapter 12 Lists
Chapter 13 List Implementations That Use Arrays
Chapter 14 A List Implementation That Links Data
Chapter 16 Sorted Lists

Objectives

After studying this chapter, you should be able to

- Search an array by using a sequential search
- Search an array by using a binary search
- Search a chain of linked nodes sequentially
- Describe the time efficiency of a search

People are always looking for something, be it a date, a mate, or a lost sock. In fact, searching is one of the most common tasks done for us by computers. Just think of how many times you search the Internet. This chapter looks at two simple search strategies, the sequential search and the binary search. You can use these strategies when implementing the method `contains` for either the ADT list or the ADT sorted list. A binary search is usually much faster than a sequential search when the data is in an array rather than a chain of linked nodes and when the data is sorted. Sorting data, however, usually takes much more time than searching it. This fact should influence your choice of search method in a given situation.

The Problem

18.1 Like the people in Figure 18-1, you can search your desk for a pen, your closet for your favorite sweater, or a list of names to see whether you are on it. Searching for a particular item—called the **target**—among a collection of many items is a common task.

FIGURE 18-1 Searching is an everyday occurrence

Let's find your name on that list. If `nameList` is an instance of an ADT list whose entries are names, we can search it by using the list operation `contains`. Recall that this method is boolean-valued and returns true if a given item is in the list.

The implementation of `contains` depends upon how we store the list entries. Among the implementations of the ADT list given in Chapters 13 and 14 is one that stores the list's entries in an array and another that uses a chain of linked nodes. Let's look at the array first.

Searching an Unsorted Array

18.2 As Segment 13.13 of Chapter 13 mentioned, a sequential search of a list compares the desired item—the target—with the first entry in the list, the second entry in the list, and so on until it either locates the desired entry or looks at all the entries without success. In an array-based implementation of the list, we search the array that contains the list's entries. We can implement this search either iteratively or recursively. This section looks at both approaches and examines their efficiencies.

VideoNote
Searching an array

Recall that two data fields of our list implementation in Chapter 13 are the array `list`, which contains the list's entries, and the integer `numberOfEntries`, which is the number of entries.

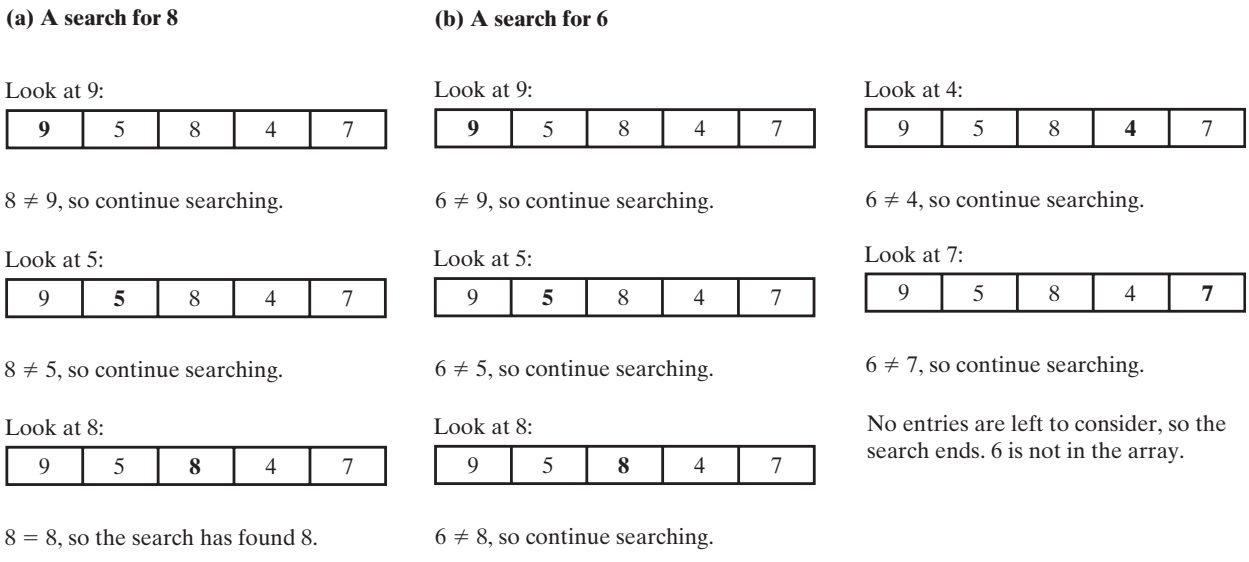
An Iterative Sequential Search of an Unsorted Array

18.3 The following implementation of `contains` was given in Segment 13.13. It uses a loop to search the array `list` containing `numberOfEntries` objects having the generic type `T` for a particular object `anEntry`:

```
public boolean contains(T anEntry)
{
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(list[index]))
         found = true;
   } // end for

   return found;
} // end contains
```

The loop exits as soon as it locates the first entry in the array that matches the desired item. In this case, `found` is true. On the other hand, if the loop examines all the entries in the list without finding one that matches `anEntry`, `found` remains false. Figure 18-2 provides an example of these two outcomes. For simplicity, our illustrations use integers.

FIGURE 18-2 An iterative sequential search of an array that (a) finds its target; (b) does not find its target



Question 1 Write a method `contains` that returns the index of the first array entry that equals `anEntry`. If the array does not contain such an entry, return `-1`.

Question 2 Write a method `contains` that performs an iterative sequential search of a list by using only operations of the ADT list. The method should return true if a given item is in a given list.

A Recursive Sequential Search of an Unsorted Array

18.4 We begin a sequential search of an array by looking at the first entry in the array. If that entry is the desired one, we end the search. Otherwise we search the rest of the array. Since this new search is also sequential and since the rest of the array is smaller than the original array, we have a recursive description of a solution to our problem. Well, almost. We need a base case. An empty array could be the base case because it never contains the desired item.

For the array `a`, we search the `n` elements `a[0]` through `a[n - 1]` by beginning with the first element, `a[0]`. If it is not the one we seek, we need to search the rest of the array—that is, we search array elements `a[1]` through `a[n - 1]`. In general, we search the array elements `a[first]` through `a[n - 1]`. To be even more general, we can search array elements `a[first]` through `a[last]`, where `first` ≤ `last`.

18.5 The following pseudocode describes the logic of our recursive algorithm:

```
Algorithm to search a[first] through a[last] for desiredItem
if (there are no elements to search)
   return false
else if (desiredItem equals a[first])
   return true
else
   return the result of searching a[first + 1] through a[last]
```

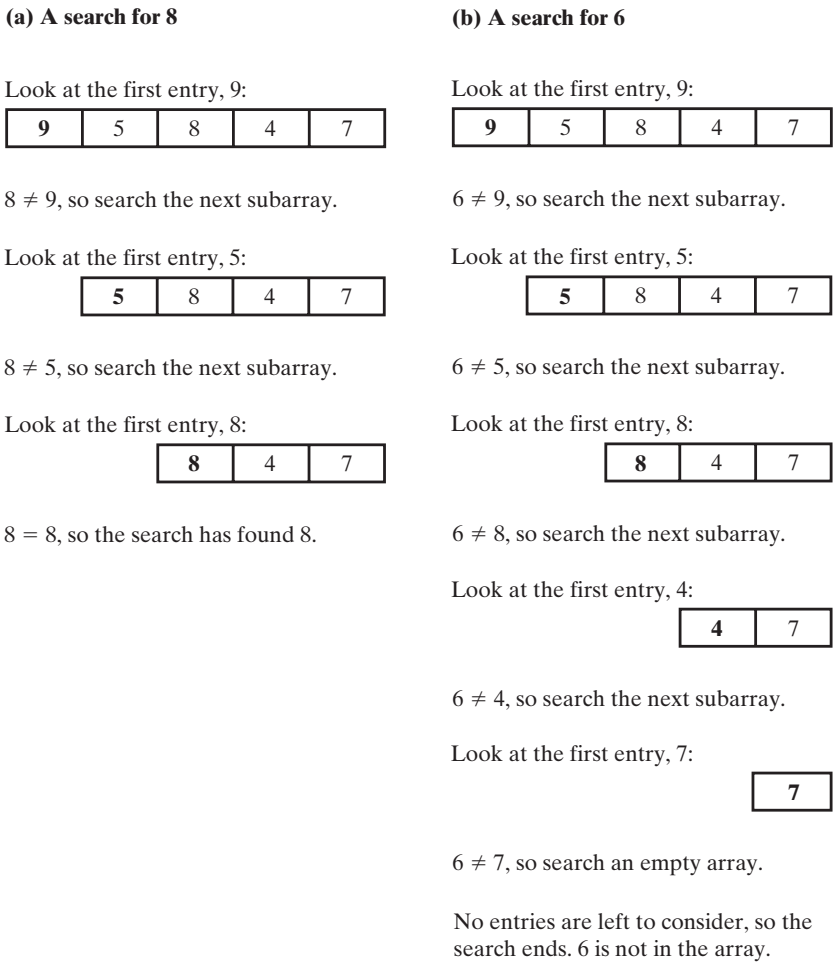
Figure 18-3 illustrates a recursive search of an array.

18.6 The method that implements this algorithm will need parameters `first` and `last`. To spare the client the detail of providing values for these parameters, and to allow the method `contains` to have the same header as it did in Segment 18.3, we implement the algorithm as a private method `search` that `contains` invokes. Since we again assume the array-based list implementation from Chapter 13, the array `list` takes the place of the array `a` in the previous algorithm, and `numberOfEntries` is the number of elements to search. Because `list` and `numberOfEntries` are data fields of the class that implements the list, they are not parameters of the methods that follow.

```
/** Searches the list for anEntry. */
public boolean contains(T anEntry)
{
   return search(0, numberOfEntries - 1, anEntry);
} // end contains

/** Searches list[first] through list[last] for desiredItem.
    @param first        an integer index >= 0 and < numberOfEntries
    @param last         an integer index >= 0 and < numberOfEntries
    @param desiredItem  the object to be found
    @return true if desiredItem is found */
```

```
private boolean search(int first, int last, T desiredItem)
{
   boolean found;

   if (first > last)
      found = false; // no elements to search
   else if (desiredItem.equals(list[first]))
      found = true;
   else
      found = search(first + 1, last, desiredItem);

   return found;
} // end search
```

FIGURE 18-3 A recursive sequential search of an array that (a) finds its target; (b) does not find its target



Question 3 List the comparisons that the previous method `search` makes while searching for the object o in the array of objects

o1 o2 o3 o4 o5

Question 4 Implement at the client level a recursive method `search` by using only operations of the ADT list. The method should return true if a given item is in a given list.

The Efficiency of a Sequential Search of an Array

18.7 Whether you implement a sequential search iteratively or recursively, the number of comparisons will be the same. In the best case, you will locate the desired item first in the array. You will have made only one comparison, and so the search will be O(1). In the worst case, you will search the entire array. Either you will find the desired item at the end of the array or you will not find it at all. In either event, you will have made n comparisons for an array of n entries. The sequential search in the worst case is therefore O(n). Typically, you will look at about one-half of the entries in the array. Thus, the average case is O(n/2), which is just O(n).

Note: **The time efficiency of a sequential search of an array**

Best case	O(1)
Worst case:	O(n)
Average case:	O(n)

Searching a Sorted Array

A sequential search of an unsorted array is rather easy to understand and to implement. When the array contains relatively few entries, the search is efficient enough to be practical. However, when the array contains many entries, a sequential search can be time-consuming. For example, imagine that you are looking through a jar of coins for one minted during the year of your birth. A sequential search of 10 coins is not a problem. With 1000 coins, the search could be lengthy; with 1 million coins, it is overwhelming. A faster search method would be welcome. Fortunately, faster searches are possible.

A Sequential Search of a Sorted Array

18.8 Suppose that before you begin searching your coins, someone arranges them in sorted order by their dates. If you search the sorted coins in Figure 18-4 sequentially for the date 1998, you would look at the coins dated 1992, 1995, and 1997 before arriving at 1998. If, instead, you look for the date 2000, you would look at the first five coins without finding it. Should you keep looking? If the coins are sorted into ascending order and you have reached the one dated 2005, you will not find 2000 beyond it. If the coins were not sorted, you would have to examine all of them to see that 2000 was not present.

Note: A sequential search can be more efficient if the data is sorted.

FIGURE 18-4 Coins sorted by their mint dates

If our array is sorted into either ascending or descending order, we can use the previous ideas to revise the sequential search. This modified search can tell whether an item does not occur in an array faster than a sequential search of an unsorted array. The latter search always examines the entire array in this case. With a sorted array, however, the modified sequential search often makes far fewer comparisons to make the same determination. Exercise 2 at the end of this chapter asks you to implement a sequential search of a sorted array.

After expending the effort to sort an array, you often can search it even faster by using the method that we discuss next.

A Binary Search of a Sorted Array

18.9 Think of a number between 1 and 1 million. When I guess at your number, tell me whether my guess is correct, too high, or too low. At most, how many attempts will I need before I guess correctly? You should be able to answer this question by the time you reach the end of this section!

If you had to find a new friend's telephone number in a printed directory, what would you do? Typically you would open the book to a page near its middle, glance at the entries, and quickly see whether you were on the correct page. If you were not, you would decide whether you had to look at earlier pages—those in the left "half" of the book—or later pages—those in the right "half." What aspect of a telephone directory enables you to make this decision? The alphabetical order of the names does.

If you decided to look in the left half, you could ignore the entire right half. In fact, you could tear off the right half and discard it, as Figure 18-5 illustrates. You have reduced the size of the search problem dramatically, as you have only half of the book left to search. You then would repeat the process on this half. Eventually you would either find the telephone number or discover that it is not there. This approach—called a **binary search**—sounds suspiciously recursive.

18.10 Let's adapt these ideas to searching an array `a` of `n` integers that are sorted into ascending order. (Descending order would also work with a simple change in our algorithm.) We know that

$$a[0] \le a[1] \le a[2] \le \ldots \le a[n-1]$$

Because the array is sorted, we can rule out whole sections of the array that could not possibly contain the number we are looking for—just as you ruled out an entire half of the telephone directory.

For example, if we are looking for the number 7 and we know that `a[5]` is equal to 9, then, of course, we know that 7 is less than `a[5]`. But we also know that 7 cannot appear after `a[5]` in the array, because the array is sorted. That is,

$$7 < a[5] \le a[6] \le \ldots \le a[n-1]$$

FIGURE 18-5 Ignoring one half of the data when the data is sorted

We know this without looking at the elements beyond `a[5]`. We therefore can ignore these elements as well as `a[5]`. Similarly, if the sought-after number were greater than `a[5]` (for example, if we were looking for 10), we could ignore `a[5]` and all the elements before it.

Replacing the index `5` in the preceding example with whatever index is in the middle of the array leads to a first draft of an algorithm for a binary search of an array:

```
Algorithm to search a[0] through a[n - 1] for desiredItem
mid = approximate midpoint between 0 and n - 1
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return the result of searching a[0] through a[mid - 1]
else if (desiredItem > a[mid])
   return the result of searching a[mid + 1] through a[n - 1]
```

Notice that to

Search `a[0]` *through* `a[n - 1]`

you have to either

Search `a[0]` *through* `a[mid - 1]`

or

Search `a[mid + 1]` *through* `a[n - 1]`

These two searches of a portion of the array are smaller versions of the very task we are solving, and so can be accomplished by calling the algorithm itself recursively.

18.11 One complication arises, however, when we write the recursive calls in the previous pseudocode. Each call searches a subrange of the array. In the first case, it is the elements indexed by `0` through `mid - 1`. In the second case, it is the elements indexed by `mid + 1` through `n - 1`. Thus, we need two extra parameters—`first` and `last`—to specify the first and last indices of the subrange of the array that is to be searched. That is, we search `a[first]` through `a[last]` for `desiredItem`.

Using these parameters and making the recursive calls look more like Java, we can express the pseudocode as follows:

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = approximate midpoint between first and last
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else if (desiredItem > a[mid])
   return binarySearch(a, mid + 1, last, desiredItem)
```

To search the entire array, we initially set `first` to `0` and `last` to `n - 1`. Each recursive call will then use some other values for `first` and `last`. For example, the recursive call that appears first would set `first` to `0` and `last` to `mid - 1`.

When you write any recursive algorithm, you should always check that the recursion is not infinite. Let's check whether every possible invocation of the algorithm will lead to a base case. Consider the three cases in the nested `if` statement in the previous pseudocode. In the first case, the sought-after item is found in the array, so there is no recursive call, and the process terminates. In each of the other two cases, a smaller portion of the array is searched by a recursive call. If the sought-after item is in the array, the algorithm uses smaller and smaller portions of the array until it finds the item. But what if the item is not anywhere in the array? Will the resulting series of recursive calls eventually lead to a base case? Unfortunately not, but that is not hard to fix.

18.12 Note that in each recursive call, either the value of `first` is increased or the value of `last` is decreased. If they ever pass each other and `first` actually becomes larger than `last`, we will have run out of array elements to check. In that case, `desiredItem` is not in the array. If we add this test to our pseudocode and refine the logic a bit, we get the following more complete algorithm:

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

Figure 18-6 provides an example of a binary search.

Question 5 When the previous binary search algorithm searches the array in Figure 18-6 for 8 and for 16, how many comparisons to an array entry are necessary in each case?

FIGURE 18-6 A recursive binary search of a sorted array that (a) finds its target; (b) does not find its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$16 > 10$, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

$16 < 18$, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

$16 > 12$, so search the right half of the array.

Look at the middle entry, 15:

15
7

$16 > 15$, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

18.13 Imagine an array-based implementation of the ADT sorted list. An array `list`—which is a private data field—holds the list's entries in sorted order. Another field `numberOfEntries` records the number of entries. When implementing the ADT's method `contains`, the algorithm `binarySearch` becomes a private method that `contains` invokes. The array `list` takes the place of the array `a` in the algorithm, and `numberOfEntries` takes the place of `n`. As before in Segment 18.6, since `list` and `numberOfEntries` are data fields, they are not parameters of `contains` and `binarySearch`.

Although the implementations of the sequential search that were given in Segments 18.3 and 18.6 use the method `equals` to make the necessary comparisons, the binary search requires more than a test for equality. To make the necessary comparisons, we need the method `compareTo`. Since all classes inherit `equals` from the class `Object` and can override it, all objects can invoke `equals`. But for an object to invoke `compareTo`, it must belong to a class that implements the interface `Comparable`. Such is the case for objects in a sorted list, as Segment 16.1 indicated.

Like the class `SortedLinkedList` in Segment 16.7, our implementation of a sorted list could begin as follows:

```
public class SortedArrayList<T extends Comparable<? super T>>
             implements SortedListInterface<T>
```

Thus, the method `binarySearch` can have the following implementation:

```
private boolean binarySearch(int first, int last, T desiredItem)
{
   boolean found;
   int mid = first + (last - first) / 2;

   if (first > last)
      found = false;
   else if (desiredItem.equals(list[mid]))
      found = true;
   else if (desiredItem.compareTo(list[mid]) < 0)
      found = binarySearch(first, mid - 1, desiredItem);
   else
      found = binarySearch(mid + 1, last, desiredItem);

   return found;
} // end binarySearch
```

Now `contains` appears as follows:

```
public boolean contains(T anEntry)
{
   return binarySearch(0, numberOfEntries - 1, anEntry);
} // end contains
```

Note: Notice that the Java computation of the midpoint `mid` is

```
int mid = first + (last - first) / 2;
```

instead of

```
int mid = (first + last) / 2;
```

as the pseudocode would suggest. If you were to search an array of at least 2^{30}, or about one billion, elements, the sum of `first` and `last` could exceed the largest possible `int` value of $2^{30} - 1$. Thus, the computation `first + last` would overflow to a negative integer and result in a negative value for `mid`. If this negative value of `mid` was used as an array index, an `ArrayIndexOutOfBoundsException` would occur. The computation `first + (last - first) / 2`, which is algebraically equivalent to `(first + last) / 2`, avoids this error.

Programming Tip: Classes that implement the `Comparable` interface must define a `compareTo` method. Such classes should also define an `equals` method that overrides the `equals` method inherited from `Object`. Both `compareTo` and `equals` should use the same test for equality. The previous method `binarySearch` calls both the method `equals` and the method `compareTo`. If the objects in the array did not have an appropriate `equals` method, `binarySearch` would not execute correctly. Note, however, that you could use `compareTo` instead of `equals` to test for equality.

Question 6 During a binary search, which elements in the array

4 8 12 14 20 24

are compared to the target when the target is **a.** 2; **b.** 8; **c.** 15.

Question 7 Modify the previous method `contains` so that it returns the index of the first array entry that equals `anEntry`. If the array does not contain such an entry, return –1. You will have to modify `binarySearch` also.

Question 8 What changes to the binary search algorithm are necessary when the array is sorted in descending order (from largest down to smallest) instead of ascending order, as we have assumed during our discussion?

Java Class Library: The Method `binarySearch`

18.14 The class `Arrays` in the package `java.util` defines several versions of a static method `binarySearch` with the following specification:

```
/** Searches an entire array for a given item.
    @param array        an array sorted in ascending order
    @param desiredItem  the item to be found in the array
    @return index of the array entry that equals desiredItem;
            otherwise returns -belongsAt - 1, where belongsAt is
            the index of the array element that should contain
            desiredItem */
public static int binarySearch(type[] array, type desiredItem);
```

Here, both occurrences of *type* must be the same; *type* can be `Object` or any of the primitive types `byte`, `char`, `double`, `float`, `int`, `long`, or `short`.

The Efficiency of a Binary Search of an Array

18.15 The binary search algorithm eliminates about half of the array from consideration after examining only one element. It then eliminates another quarter of the array, and then another eighth, and so on. Thus, most of the array is not searched at all, saving much time. Intuitively, the binary search algorithm is very fast.

But just how fast is it? Counting the comparisons that occur will provide a measure of the algorithm's efficiency. To see the algorithm's worst-case behavior, you count the maximum number of comparisons that can occur when searching an array of n items. Comparisons are made each time the algorithm divides the array in half. After each division, half of the items are left to search.

That is, beginning with n items, we would be left with $n/2$ items, then $n/4$ items, and so on. In the worst case, the search would continue until only one item was left. That is, $n/2^k$ would equal 1 for some integer value of k. This value of k gives us the number of times the array is divided in half, or the number of recursive calls to `binarySearch`.

If n is a power of 2, n is 2^k for some positive k. By the definition of a logarithm, k is $\log_2 n$. If n is not a power of 2, you can find a positive integer k so that n lies between 2^{k-1} and 2^k. For example, if n is 14, $2^3 < 14 < 2^4$. Thus, we have for some $k \geq 1$,

$$
\begin{aligned}
&2^{k-1} < n < 2^k \\
&k - 1 < \log_2 n < k \\
&k = 1 + \log_2 n \text{ rounded down} \\
&\quad = \log_2 n \text{ rounded up}
\end{aligned}
$$

To summarize,

$k = \log_2 n$ when n is a power of 2
$k = \lceil \log_2 n \rceil$ when n is not a power of 2

In general, k—the number of recursive calls to `binarySearch`—is $\lceil \log_2 n \rceil$.

Note: Ceiling and floors

The *ceiling* of a number x, denoted as $\lceil x \rceil$, is the smallest integer greater than or equal to x. For example, $\lceil 4.1 \rceil$ is 5. The *floor* of a number x, denoted as $\lfloor x \rfloor$, is the largest integer less than or equal to x. For example, $\lfloor 4.9 \rfloor$ is 4. When you *truncate* a positive real number to an integer, you actually are computing the number's floor by discarding any fractional portion.

Each call to `binarySearch`, with the possible exception of the last one, makes two comparisons between the target and the middle element in the array: One tests for equality and one for less than or greater than. Thus, the binary search performs at most $2\lceil \log_2 n \rceil$ comparisons, and so in the worst case is O(log n).

To search an array of 1000 elements, the binary search will compare the target to about 10 array entries in the worst case. In contrast, a simple sequential search could compare the target to as many as all 1000 array entries, and on average will compare it to about 500 array elements.

Note: The time efficiency of a binary search of an array

Best case:	O(1)
Worst case:	O(log n)
Average case:	O(log n)

Question 9 Think of a number between 1 and 1 million. When I guess at your number, tell me whether my guess is correct, too high, or too low. At most, how many attempts will I need before I guess correctly? *Hint*: You are counting guesses, not comparisons.

18.16 **Another approach.** The binary search makes comparisons each time it locates the midpoint of the array. Thus, to search n items, the binary search looks at the middle item and then searches $n/2$ items. If we let $t(n)$ represent the time requirement for searching n items, we find that at worst

$$t(n) = 1 + t(n/2) \text{ for } n > 1$$
$$t(1) = 1$$

We encountered this recurrence relation in Segment 7.25 of Chapter 7. There, we showed that

$$t(n) = 1 + \log_2 n$$

Thus, the binary search is O(log n) in the worst case.

Searching an Unsorted Chain

18.17 Within a linked implementation of either the ADT list or the ADT sorted list, the method `contains` would search a chain of linked nodes for the target. As you will see, a sequential search is really the only practical choice. We begin with a chain whose data is unsorted, as typically would be the case for the ADT list.


VideoNote
Searching a linked chain

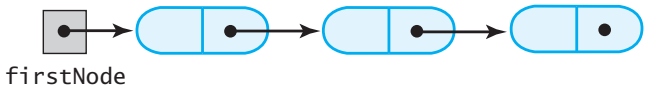
Regardless of a list's implementation, a sequential search of the list looks at consecutive entries in the list, beginning with the first one, until either it finds the desired entry or it looks at all entries without success. When the implementation is linked, however, moving from node to node is not as simple as moving from one array location to another. Despite this fact, you can implement a sequential search of a chain of linked nodes either iteratively or recursively and with the same efficiency as that of a sequential search of an array.

An Iterative Sequential Search of an Unsorted Chain

18.18 Figure 18-7 illustrates a chain of linked nodes that contain the list's entries. Recall from Segment 14.8 of Chapter 14 that `firstNode` is a data field of the class that implements the list. While it is clear that a method can access the first node in this chain by using the reference `firstNode`, how can it access the subsequent nodes? Since `firstNode` is a data field that always references the first node in the chain, we would not want our search to alter it or any other aspect of the list. Thus, an iterative method `contains` should use a local reference variable `currentNode` that initially contains the same reference as `firstNode`. To make `currentNode` reference the next node, we would execute the statement

```
currentNode = currentNode.getNextNode();
```

FIGURE 18-7 A chain of linked nodes that contain the entries in a list



The iterative sequential search has the following straightforward implementation:

```
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
```

```
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```

This implementation is like the one given in Segment 14.19 of Chapter 14.

A Recursive Sequential Search of an Unsorted Chain

18.19 When done recursively, a sequential search looks at the first entry in the list and, if it is not the desired entry, searches the rest of the list. This recursive approach is the same regardless of whether you implement the search at the client level by using only the list's ADT operations—as you did in Question 4—or as a public method of an array-based implementation of the list—as we did in Segment 18.6. We use the same approach for a linked implementation of the list, as follows.

How would you implement the step *search the rest of the list* when the list's entries are in a chain of linked nodes? The iterative method `contains` that you saw in the previous segment uses a local variable `currentNode` to move from node to node. A recursive method could not have `currentNode` as a local variable, since `currentNode` would get reset to an initial value at each recursive call. Instead, such a method needs `currentNode` as a formal parameter. But then we would have a method whose parameter depends on the list's implementation, making it unsuitable as a public method. Just as we did earlier in Segments 18.6 and 18.13, we would make this search method private and call it from the public method `contains`.

18.20 The private recursive method `search` examines the list entry in the node that its parameter `currentNode` references. If the entry is not the desired one, the method recursively calls itself with an argument that references the next node in the chain. Thus, the method `search` has the following implementation:

```
// Recursively searches a chain of nodes for desiredItem,
// beginning with the node that currentNode references.
private boolean search(Node currentNode, T desiredItem)
{
   boolean found;

   if (currentNode == null)
      found = false;
   else if (desiredItem.equals(currentNode.getData()))
      found = true;
   else
      found = search(currentNode.getNextNode(), desiredItem);

   return found;
} // end search
```

Now we write the public method `contains` as follows:

```
public boolean contains(T anEntry)
{
   return search(firstNode, anEntry);
} // end contains
```

Notice that the call to the method `search` initializes the parameter `currentNode` to `firstNode`, much as an iterative method initializes its local variable `currentNode` to `firstNode`.

The Efficiency of a Sequential Search of a Chain

18.21 The efficiency of a sequential search of a chain is really the same as that of a sequential search of an array. In the best case, the desired item will be first in the chain. Thus, at best the search will be O(1), since you will have made only one comparison. In the worst case, you will search the entire chain, making n comparisons for a chain of n nodes. Therefore, the sequential search in the worst case is O(n). Typically, you will look at about half of the nodes in the chain. Thus, the average-case search is O(n/2), which is just O(n).

Note: The time efficiency of a sequential search of a chain of linked nodes

Best case: O(1)
Worst case: O(n)
Average case: O(n)

Searching a Sorted Chain

We now search a chain whose data is sorted. Such a chain would occur in a linked implementation of the ADT sorted list.

A Sequential Search of a Sorted Chain

18.22 Searching a chain of linked nodes whose data is sorted is similar to sequentially searching a sorted array, as described in Segment 18.8. Here, we incorporate that logic into the following implementation of `contains`:

```
public boolean contains(T anEntry)
{
   Node currentNode = firstNode;

   while ( (currentNode != null) &&
           (anEntry.compareTo(currentNode.getData()) > 0) )
   {
      currentNode = currentNode.getNextNode();
   } // end while

   return (currentNode != null) &&
          anEntry.equals(currentNode.getData());
} // end contains
```

The method traverses the chain until it either reaches a node that could contain the desired object or examines all nodes without success. Following the traversal, a final test is necessary to draw a conclusion.

A Binary Search of a Sorted Chain

18.23 A binary search of an array looks first at the element that is at or near the middle of the array. It is easy to determine the index `mid` of this element by computing `first + (last - first) / 2`, where `first` and `last` are the indices of the first and last elements, respectively, in the array. Accessing this middle element is also easy: For an array `a`, it is simply `a[mid]`.

Now consider searching a chain of linked nodes, such as the one you saw earlier in Figure 18-7, whose nodes are sorted. How would you access the entry in the middle node? Since this chain has only three nodes, you can get to the middle node quickly, but what if the

chain contained 1000 nodes? In general, you need to traverse the chain, beginning at the first node, until you reach the middle node. How will you know when you get there? If you know the length of the chain, you can divide the length in half and count nodes as you traverse. The details are not as important as a realization that it takes a bit of work to access the middle node.

After looking at the entry in the middle node, you probably need to ignore half of the chain and search the other half. Do not change the chain when ignoring part of it. Remember that you want to search the chain, not destroy it. Once you know which half to search, you must find its middle node, again by traversing the chain. It should be clear to you that a binary search of a linked chain of nodes would be challenging to implement and less efficient than a sequential search.

Note: A binary search of a chain of linked nodes is impractical.

Choosing a Search Method

18.24 **Choosing between a sequential search and a binary search.** You just saw that you should use a sequential search to search a chain of linked nodes. But if you want to search an array of objects, you need to know which algorithms are applicable. To use a sequential search, the objects must have a method `equals` that ascertains whether two distinct objects are equal in some sense. Since all objects inherit `equals` from the class `Object`, you must ensure that the objects you search have overridden `equals` with an appropriate version. To perform a binary search on an array of objects, on the other hand, the objects must have a `compareTo` method and the array must be sorted. If these conditions are not met, you must use a sequential search.

If both search algorithms are applicable to your array, what search should you use? If the array is small, you can simply use a sequential search. If the array is large and already sorted, a binary search is typically much faster than a sequential search. But if the array is not sorted, should you sort it and then use a binary search? The answer depends on how often you plan to search the array. Sorting takes time, typically more time than a sequential search would. If you plan to search an unsorted array only a few times, sorting the array so that you can use a binary search likely will not save you time; use a sequential search instead.

Figure 18-8 summarizes the time efficiencies of the sequential search and the binary search. Only the sequential search is applicable to unsorted data. The efficiencies given for the binary search are for an array-based sorted list. For a large, sorted list, the binary search is typically much faster than a sequential search.

FIGURE 18-8 The time efficiency of searching, expressed in Big Oh notation

	Best Case	Average Case	Worst Case
Sequential search (unsorted data)	O(1)	O(n)	O(n)
Sequential search (sorted data)	O(1)	O(n)	O(n)
Binary search (sorted array)	O(1)	O(log n)	O(log n)

18.25 **Choosing between an iterative search and a recursive search.** Since the recursive sequential search is tail recursive, you can save some time and space by using the iterative version of the search. The binary search is fast, so using recursion will not require much additional space for the recursive calls. Also, coding the binary search recursively is somewhat easier than coding it iteratively. To convince yourself of this, try to code an iterative version of the binary search. (See Exercise 6 at the end of this chapter.)

Chapter Summary

- A sequential search of either a list, an array, or a chain looks at the first item, the second item, and so on until it either finds a particular item or discovers that the item does not occur in the group.
- The average-case performance of a sequential search is O(n).
- Typically, you perform a sequential search iteratively, although a simple recursive approach is also possible.
- A binary search of an array requires that the array be sorted. It looks first to see whether the desired item is at the middle of the array. If it is not, the search decides in which half of the array the item can occur and repeats this strategy on only this half.
- A binary search is O(log n) in the worst case.
- Typically, you perform a binary search recursively, although an iterative approach is also possible.
- A binary search of a linked chain of nodes is impractical.

Programming Tip

- Classes that implement the `Comparable` interface must define a `compareTo` method. Such classes should also define an `equals` method that overrides the `equals` method inherited from `Object`. Both `compareTo` and `equals` should use the same test for equality. The method `binarySearch` in Segment 18.13 calls both the method `equals` and the method `compareTo`. If the objects in the array did not have an appropriate `equals` method, `binarySearch` would not execute correctly. Note, however, that you could use `compareTo` instead of `equals` to test for equality.

Exercises

1. Revise the recursive method `search`, as given in Segment 18.6, so that it looks at the last entry in the array instead of the first one.
2. When searching a sorted array sequentially, you can ascertain that a given item does not appear in the array without searching the entire array. For example, if you search the array

   2 5 7 9

   for 6, you can use the approach described in Segment 18.8. That is, you compare 6 to 2, then to 5, and finally to 7. Since you did not find 6 after comparing it to 7, you do not have to look further, because the other entries in the array are greater than 7 and therefore cannot equal 6. Thus, you do not simply ask whether 6 equals an array entry, you also ask whether it is greater than the entry. Since 6 is greater than 2, you continue the search. Likewise for 5. Since 6 is less than 7, you have passed the point in the array where 6 would have had to occur, so 6 is not in the array.

   a. Write an iterative method `contains` to take advantage of these observations when searching a sorted array sequentially.
   b. Write a recursive method `search` that a method `contains` can call to take advantage of these observations when searching a sorted array sequentially.

3. How many comparisons are made by the recursive method `search` described in Part *b* of the previous exercise when searching the array in Figure 18-6 for 8 and for 16?

4. Trace the method `binarySearch`, as given in Segment 18.13, when searching for 4 in the following array of values:

5 8 10 13 15 20 22 26 30 31 34 40

Repeat the trace when searching for 34.

5. Modify the method `binarySearch` in Segment 18.13 so that it returns the index of the first array entry that equals `desiredItem`. If the array does not contain such an entry, return `-(belongsAt + 1)`, where `belongsAt` is the index of the array location that should contain `desiredItem`. At the end of Segment 18.13, Question 7 asked you to return -1 in this case. Notice that both versions of the method return a negative integer if and only if `desiredItem` is not found.

6. Implement a binary search of an array iteratively. Model your methods after the ones given in Segment 18.13.

7. Write a recursive method to find the largest object in an array-based list of `Comparable` objects. Like the binary search, your method should divide the array into halves. Unlike the binary search, your method should search both halves for the largest object. The largest object in the array will then be the larger of these two largest objects.

8. Suppose that you are searching an unsorted array of objects that might contain duplicates. Devise an algorithm that returns a list of the indices of all objects in the array that match a given object. If the desired object is not in the list, return an empty list.

9. Repeat the previous exercise for a sorted array. Your algorithm should be recursive and efficient.

10. In Segment 18.13, the method `contains` calls a private method that performs a binary search of an array. Assuming a linked implementation of the ADT sorted list, revise this private method to perform a binary search of a chain of nodes. Do not alter the chain.

11. Consider the number $f(n)$ of comparisons that a sequential search makes in the worst case.

a. Write a recurrence relation for $f(n)$.
b. Prove by induction on n that $f(n) = n$.

12. At the end of Segment 18.3, Question 2 asked you to write a method that performs an iterative sequential search of a list by using only operations of the ADT list. Compare the time efficiency of this method with the ADT operation `contains`.

13. In Segment 18.7, we said that a sequential search of an array will examine on average about half of the n entries. Let's look a little more carefully at this computation. A sequential search is either successful or not. Let α be the probability that we will find the desired value in the array and $1 - \alpha$ be the probability that we will not. We further assume that the value, if found, is equally likely to be in each of the locations of the array. We need to consider each possibility.

For each case, we count the comparisons and determine its probability of occurrence. To find the average number of comparisons made by the search, we first multiply each probability by the number of comparisons in each case. The following table summarizes these results:

	Probability	Number of Comparisons	Product
Found at index 0	α / n	1	α / n
Found at index 1	α / n	2	$2\alpha / n$
Found at index 2	α / n	3	$3\alpha / n$
...	...	...	...
Found at index n - 2	α / n	$n - 1$	$(n - 1)\alpha / n$
Found at index n - 1	α / n	n	α
Not found	$1 - \alpha$	n	$(1 - \alpha)\, n$

a. Compute the average number of comparisons by adding all the products in the last column of the table.
b. What is the average number of comparisons if the search is guaranteed to be successful ($\alpha = 1$)?
c. What is the average number of comparisons if the search is guaranteed to be unsuccessful ($\alpha = 0$)?
d. What is the average number of comparisons if the search is successful half of the time ($\alpha = 0.5$)?

14. Repeat Part *a* of the previous exercise, but now assume that we are not equally likely to search for each value in the array. We could arrange the *n* items in the array such that the ones we are more likely to search for occur first. Suppose that we search for the first item one half of the time, the second item one quarter of the time, the third item one eighth of the time, and so on. We will search for the last two items $1/2^{n-1}$ of the time. Revise the table in the previous exercise accordingly.

PROJECTS

1. When an object does not occur in an array, a sequential search for it must examine the entire array. If the array is sorted, you can improve the search by using the approach described in Exercise 2. A **jump search** is an attempt to reduce the number of comparisons even further.

Instead of examining the *n* objects in the array *a* sequentially, you look at the elements *a*[*j*], *a*[2*j*], *a*[3*j*], and so on, for some positive $j < n$. If the target *t* is less than one of these objects, you need to search only the portion of the array between the current object and the previous object. For example, if *t* is less than *a*[3*j*] but is greater than *a*[2*j*], you search the elements *a*[2*j* + 1], *a*[2*j* + 2], . . ., *a*[3*j* - 1] by using the method in Exercise 2. What should you do when $t > a[k * j]$, but $(k + 1) * j > n$?

Devise an algorithm for performing a jump search. Then, using $\lceil \sqrt{n} \rceil$ as the value of *j*, implement the jump search.

2. An **interpolation search** assumes that the data in an array is sorted and uniformly distributed. Whereas a binary search always looks at the middle item in an array, an interpolation search looks where the sought-for item is more likely to occur. For example, if you searched your telephone book for Victoria Appleseed, you probably would look near its beginning rather than its middle. And if you discovered many Appleseeds, you would look near the last Appleseed.

Instead of looking at the element `a[mid]` of an array `a`, as the binary search would, an interpolation search examines `a[index]`, where

```
p = (desiredElement - a[first])/(a[last] - a[first])
index = first + ⌈(last − first) × p⌉
```

Implement an interpolation search of an array. For particular arrays, compare the outcomes of an interpolation search and of a binary search. Consider arrays that have uniformly distributed entries and arrays that do not.

3. Suppose that you have numerical data stored in a two-dimensional array, such as the one in Figure 18-9. The data in each row and in each column is sorted in increasing order.

a. Devise an efficient search algorithm for an array of this type.
b. If the array has *m* rows and *n* columns, what is the Big Oh performance of your algorithm?
c. Implement and test your algorithm.

FIGURE 18-9 A two-dimensional array for Project 3

1	4	55	88
7	15	61	91
14	89	90	99

4. Consider an array `data` of *n* numerical values in sorted order and a list of numerical target values. Your goal is to compute the smallest range of array indices that contains all of the target values. If a target value is smaller than `data[0]`, the range should start with -1. If a target value is larger than `data[n - 1]`, the range should end with *n*.

For example, given the array in Figure 18-10 and the target values (8, 2, 9, 17), the range is -1 to 5.

a. Devise an efficient algorithm that solves this problem.
b. If you have *n* data values in the array and *m* target values in the list, what is the Big Oh performance of your algorithm?
c. Implement and test your algorithm.

FIGURE 18-10 An array for Project 4

5	8	10	13	15	20	22	26
0	1	2	3	4	5	6	7

5. One way to organize a collection of words is to use an array of sorted lists. The array contains one sorted list for each letter of the alphabet. To add a word to this data structure, you add it to the sorted list that corresponds to the word's first letter. Design an ADT for such a collection, including the operations `add` and `contains`. Define a Java interface for your ADT. Then implement your interface as a class and test it. Use a text file of words to populate your data structure.

Answers to Self-Test Questions

1.
```
public int contains(T anEntry)
{
   boolean found = false;
   int result = -1;
   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(list[index]))
      {
         found = true;
         result = index;
      } // end if
   } // end for

   return result;
} // end contains
```

2.
```
public static <T> boolean contains(AList<T> theList, T anEntry)
{
   boolean found = false;
   int length = theList.getLength();
   for (int position = 1; !found && (position <= length); position++)
   {
      if (anEntry.equals(theList.getEntry(position)))
         found = true;
   } // end for
   return found;
} // end contains
```

3. The object o is compared with o1, then o2, o3, o4, and o5.

4.

```
public static <T> boolean contains(AList<T> theList, T anEntry)
{
   return search(theList, 1, theList.getLength(), anEntry);
} // end contains

private static <T> boolean search(AList<T> theList, int first, int last,
                                  T desiredItem)
{
   boolean found;

   if (first > last)
      found = false;
   else if (desiredItem.equals(theList.getEntry(first)))
      found = true;
   else
      found = search(theList, first + 1, last, desiredItem);
   return found;
} // end search
```

5. Searching for 8 requires seven comparisons, as follows:
8 == 10?
8 < 10?
8 == 5?
8 < 5?
8 == 7?
8 < 7?
8 == 8?

Searching for 16 requires eight comparisons, as follows:
16 == 10?
16 < 10?
16 == 18?
16 < 18?
16 == 12?
16 < 12?
16 == 15?
16 < 15?

6.
a. 12 and 4.
b. 12, 4, and 8.
c. 12, 20, and 14.

7.

```
public int contains(T anEntry)
{
   return binarySearch(0, numberOfEntries - 1, anEntry);
} // end contains

private int binarySearch(int first, int last, T desiredItem)
{
   int result;
   int mid = first + (last - first) / 2;

   if (first > last)
      result = -1;
   else if (desiredItem.equals(list[mid]))
      result = mid;
   else if (desiredItem.compareTo(list[mid]) < 0)
```

```
      result = binarySearch(first, mid - 1, desiredItem);
   else
      result = binarySearch(mid + 1, last, desiredItem);
   return result;
} // end binarySearch
```

8. In the second `else if`, change < to >.

9. 20 (log 1,000,000 rounded up).

This page intentionally left blank