chapter 12

Array-Based Lists

Goals

- To be able to distinguish between an array and a list.
- To be able to insert an item into a list.
- To be able to delete an item from a list.
- To be able to search for an item in a list.
- To be able to sort the items in a list into ascending or descending order.
- To be able to build a list in sorted order.
- To be able to search for an item using a binary search.
- To be able to use the Visual Basic `Comparable` interface.

Chapter 11 introduced the array, a structured data type that holds a collection of components of the same type or class given a single name. In general, a one-dimensional array is a structure used to hold a list of items. We all know intuitively what a "list" is; in our everyday lives we use lists all the time—grocery lists, lists of things to do, lists of addresses, lists of party guests. In computer programs, lists are very useful and common ways to organize the data. In this chapter, we examine algorithms that build and manipulate lists implemented using a one-dimensional array to hold the items.

12.1 Lists

From a logical point of view, a list is a homogenous collection of elements, with a **linear relationship** between elements. *Linear* means that, at the logical level, each element in the list except the first one has a unique predecessor, and each element except the last one has a unique successor.[1] The number of items in the list, which we call the length of the list, is a property of a list. That is, every list has a length.

Linear relationship Each element except the first has a unique predecessor, and each element except the last has a unique successor

Length The number of items in a list; the length can vary over time.

Unsorted list A list in which data items are placed in no particular order with respect to their content; the only relationship between data elements is the list predecessor and successor relationships.

Sorted list A list with predecessor and successor relationships determined by the contents of the keys of the items in the list; there is a semantic relationship among the keys of the items in the list

Key A member of a class whose value is used to determine the logical and/or physical order of the items in a list

Lists can be **unsorted**—their elements may be placed into the list in no particular order—or they can be **sorted** in a variety of ways. For instance, a list of numbers can be sorted by value, a list of strings can be sorted alphabetically, and a list of addresses can be sorted by zip code. When the elements in a sorted list are composite types, their logical (and often physical) order is determined by one of the members of the structure, the **key** member. For example, a list of students on a class roll can be sorted alphabetically by name or numerically by student identification number. In the first case, the name is the key; in the second case, the identification number is the key (see Figure 12.1).

If a list cannot contain items with duplicate keys, it is said to have *unique* keys (see Figure 12.2). This chapter deals with both unsorted lists and lists of elements with unique keys, sorted from smallest to largest key value. The items on the list can be of any type, atomic or composite. In the following discussion, "item," "element," and "component" are synonyms; they are what is stored in the list.

[1]At the implementation level, there is also a relationship between the elements but the physical relationship may not be same as the logical one.

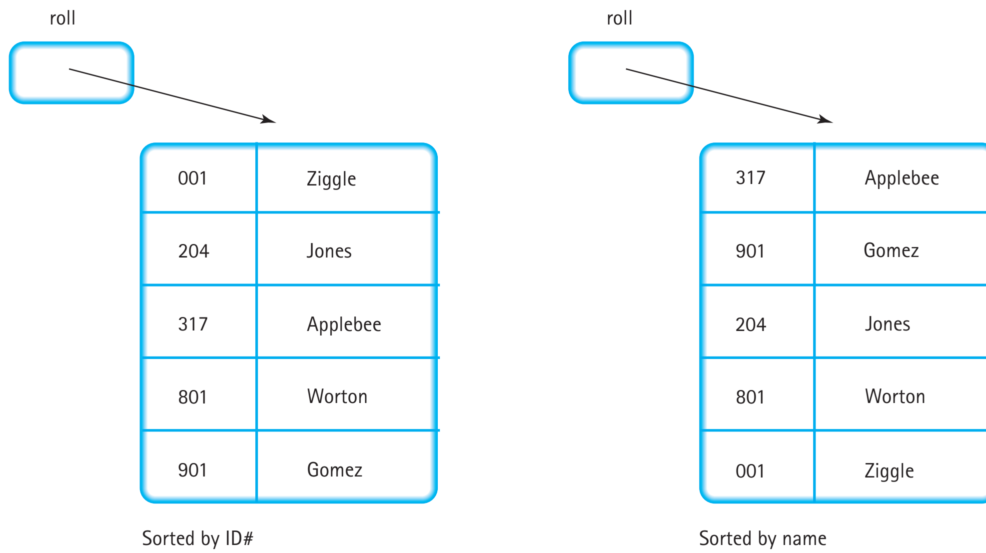


Figure 12.1 *List sorted by two different keys*

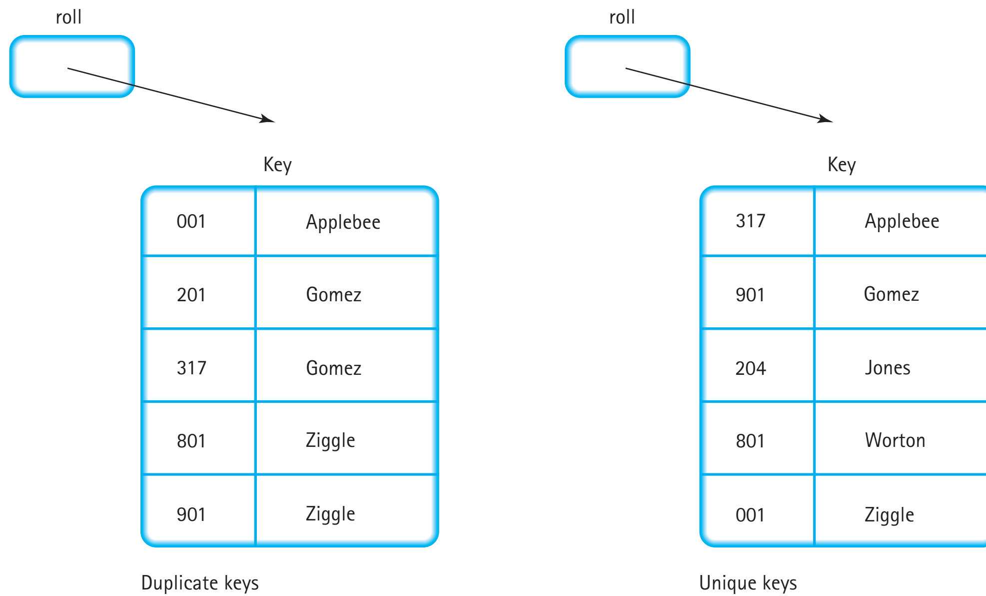


Figure 12.2 *(a) List with duplicate keys and (b) List with unique keys*

12.2 List Class

In this section, we design and implement a general-purpose class that represents a list of items. Let's think in terms of a to-do list. But before we begin to brainstorm, we must ask an important question: For whom are we designing the class? We may be designing it for ourselves to keep in our library of classes. We may be designing it for others to use in a team project. When we design a class, the software that uses it is called the **client** of the class. In our discussion, we use the terms *client* and *user* interchangeably, thinking of the client as referring to the people writing the software that uses the class, rather than the software itself.

Client Software that declares and manipulates objects of a particular class

Brainstorming the `List` Class

Because we are designing a general-purpose class, our brainstorming must be more speculative. We don't have a specific problem to solve; we have to think in terms of what we do with our to-do lists and what other things we might like to do if we could. Hopefully, we start with an empty list each morning and add things to it. As we accomplish a task on the list, we cross it off. We check to see if an item is on the list. We check to see if the list is empty (we wish!). We go through the list one item at a time. Let's translate these observations into responsibilities for the list with their responsibility type.

Class Name: List	Superclass: Object	Subclasses:

Responsibilities	Collaborations
Create itself (maxItems)	None
Is list full? Observer	None
return boolean	
Is list empty? Observer	None
return boolean	
Know length, Observer	None
return int	
Is an item in the list? Observer	None
return boolean	
Insert into list (item), Transformer	None
Delete from list (item), Transformer	None
Know each item, Iterator	None

Although we have designed our CRC card for a to-do list, the responsibilities are valid for any kind of list. For example, if we are creating a list of people to invite to a wedding, all of these operations are valid. We add names to the list, check whether a name is on the list, count names on the list, check to see if the list if full (i.e., the length is equal to the number of invitations bought), delete names, and review the names one at a time.

To make the rest of the discussion more concrete, let's assume that the items on the list are strings. We then show how the items can be made even more general.

Refining the Responsibilities

Let's go back through the responsibilities, refining them and converting them into method headings. Because we are designing a general-purpose class, we do not have any specific scenarios that we can use. Instead, we consider a variety of simplified scenarios that are examples of how we believe the class might be employed. Because the class is intended for widespread use, we should pay special attention to the documentation right from the design stage.

The observers, testing for full and empty, returning the number of items, and checking to see if an item is in the list, need no further discussion. Here are their method headings.

```
Public Function isFull() As Boolean
' Returns True if no room to add a component, False otherwise

Public Function isEmpty() As Boolean
' Returns True if no components in the list, False otherwise

Public Function Length() As Integer
' Returns the number of components in the list

Public isThere(item As String) As Boolean
' Returns True if item is in the list, False otherwise
```

In designing the transformers, we have to make some decisions. For example, do we allow duplicates in our list? This has implications for deleting items as well as inserting items. If we allow duplicates, what do we mean by removing an item? Do we delete just one or all of them? Because the focus of this chapter is on algorithms, we just make a decision and design our algorithms to fit. We examine the effect of other choices in the exercises. Let's allow only one copy of an item in the list. This decision means that delete just removes one copy. However, do we assume that the item to be removed is in the list? Is it an error if it is not? Or does the delete operation mean "Delete, if there"? Let's use the last meaning.

We now incorporate these decisions into the documentation for the method headings.

```
Public Sub insert(item As String)
' Adds item to the list
' Assumption: item is not already in the list

Public Sub delete(item As String)
' Item is removed from the list if it is in the list
```

The iterator allows the user to see each item in the list one at a time. Let's call the method that implements the Know Each Item responsibility `getNextItem`. The list must keep track of the next item to return when the iterator is called so it uses a state variable to record the position. The constructor initializes this position to 0, and it is incremented in `getNextItem`. A client can use the length of the list to control a loop that asks to see each item in turn. As a precaution, the current position should be reset to 0 after the last item has been accessed. In an application, we might need a transformer iterator that goes through the list applying an operation to each item; however, for our general discussion we provide only an observer iterator.

What happens if a user inserts or deletes an item in the middle of an iteration? Nothing good you can be sure! Adding and deleting items changes the length of the list, making the termination condition of our iteration-counting loop invalid. Depending on whether an addition or deletion occurs before or after the iteration point, our iteration loop could end up skipping or repeating items.

We have several choices of how to handle this possibly dangerous situation. The list can throw an exception, the list can reset the current position when inserting or deleting, or the list can disallow transformer operations while an iteration is taking place. We choose the latter here by way of an assumption in the documentation. In case the user wants to restart an iteration, let's provide a `resetMethod` that reinitializes the current position.

```
Public Sub resetList()
' The current position is reset

Public Sub getNextItem() As String
' Assumption: No transformers have been called since the iteration began
```

Before we go to the implementation phase, let's look at how we might use `getNextItem`. Suppose the client program wants to print out the items in the list. The client program cannot directly access the list items, but it can use `getNextItem` to iterate through the list. The following code fragment prints the string values in `list`.

```
Dim next As String
Dim index As Integer
For index = 1 to list.Length()
   next = list.getNextItem()
   Console.WriteLine(next & " is still in the list.")
Next index
```

Now is also the time to look back over the list and see if we need to add any responsibilities. For example, do we need to provide an equals test? If we want to perform a deep comparison of two lists, we must provide equals; however, comparing lists is not a particularly useful operation, and we provide the client the tools to write a comparison operation if necessary. If fact, here is the client code to compare two lists. The algorithm determines if the lengths are the same and, if so, iterates through the lists checking to see if corresponding items are the same.

isDuplicate

```
If lengths are not the same
  return False
Else
  Set counter to 1
  Set same to True
  Set limit to length of first list
  While they are still the same AND counter is less than or equal to limit
    Set next1 to next item in the first list
    Set next2 to next item in the second list
    Set same to result of seeing if next1.compareTo(next2) is 0
    Increment counter
Return same
```

We can implement this algorithm without having to know anything about the list. We just use the instance methods supplied in the interface.

```
Public Function isDuplicate(list1 As List, list2 As List) As Boolean
   If (list1.Length() <> list2.Length()) Then
      return False
   Else
      Dim next1 As String
      Dim next2 As String
      Dim counter As Integer = 1
      Dim same As Boolean = True
      Dim limit As Integer = list1.Length()
      list1.resetList()
      list2.resetList()
      While (same And (counter <= limit))
         next1 = list1.getNextItem()
         next2 = list2.getNextItem()
         same = (next1.compareTo(next2) = 0)
```

```
            counter += 1
        End While
    End If
    return same
End Function
```

Internal Data Representation

How are we going to represent the items in the list? An array of strings is the obvious answer. What other data fields do we need? We have to keep track of the number of items in our list, and we need a state variable that tells us where we are in the list during an iteration.

```
Public Class List
    Protected listItems() As String
    Protected numItems As Integer
    Protected currentPos As Integer
    ...
End Class
```

We introduced the concept of subarray processing in the last chapter and pointed out that every Visual Basic array object has a field called `length` that contains the number of components defined for the array object. The literature for lists uses the identifier "length" to refer to the number of items that have been put into the list. Faced with this ambiguity in terminology, we still talk about the length of the list, but we refer to the field that contains the number of items in the list as `numItems`.

It is very important to understand the distinction between the array object that contains the list items and the list itself. The array object is `listItems(0)..listItems(listItems.Length-1)`; the items in the list are `listItems(0)..listItems(numItems-1)`. This distinction is illustrated in Figure 12.3. Six items have been stored into the list, which is instantiated with the following statement:

```
Dim myList = new List(10)
```

Responsibility Algorithms for Class List

As Figure 12.3 shows, the list exists in the array elements `listItems(0)` through `listItems(numItems-1)`. To create an empty list, it is sufficient to set the `numItems` field to 0. We do not need to store any special values into the data array to make the list empty, because only those values in `listItems(0)` through `listItems(numItems-1)` are processed by the list algorithms. We explain why `currentPos` is set to 0 when we look more closely at the interator.

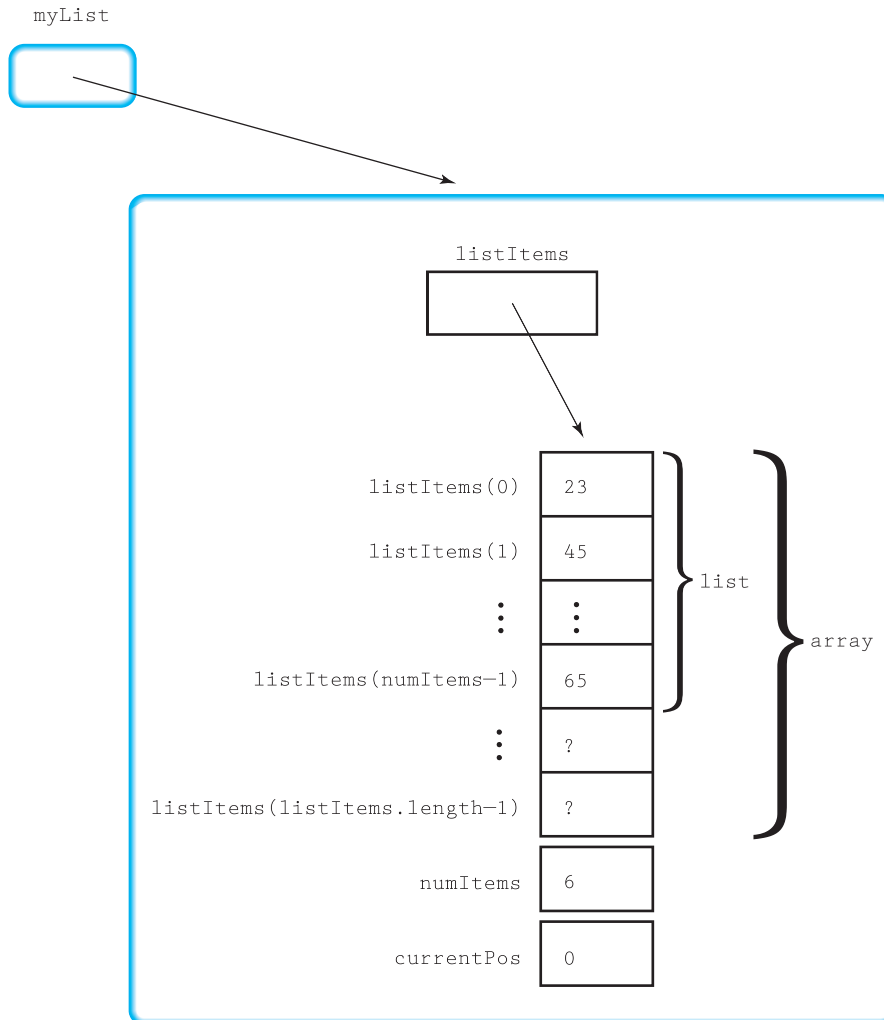


Figure 12.3 *An instance of class* `List`

```
Public Sub New(maxItems As Integer)
' Instantiates an empty list object with room for maxItems items
   numItems = 0
   Dim listItems(maxItems) As String
   currentPos = 0
End Sub
```

Should the class provide a default constructor? Let's do so as a precaution.

```
Public Sub New()
' Instantiates an empty list object with room for 100 items
   numItems = 0
   Dim listItems(100) As String
   currentPos = 0
End Sub
```

The observers `isFull`, `isEmpty`, and `Length` are very straightforward. Each is only one line long, as is so often the case in methods within the object-oriented paradigm.

```
Public Function isFull() As Boolean
' Returns True if no room to add a component, False otherwise
   Return (listItems.Length() = numItems)
End Sub

Public Function isEmpty() As Boolean
' Returns True if no components are in the list, False otherwise
   Return (numItems = 0)
End Sub

Public Function Length() As Integer
' Returns the number of components in the list
   Return numItems
End Sub
```

We have one more observer to implement: `isThere`. Because `isThere` is an instance method, it has direct access to the items in the list. We just loop through the items in the list looking for the one specified on the parameter list. The loop ends when we find the matching item or have looked at all the items in the list. Our loop expression has two conditions: (1) the index is within the list and (2) the corresponding list item is not equal to the one for which we are searching. After the loop is exited, we return the assertion that the index is still within the list. If this assertion is true, then the search item was found.

isThere

Set index to 0
While more to examine and item not found
  Increment index
Return (index is within the list)

This algorithm can be coded directly into Visual Basic, using the `compareTo` method of `String`.

```
Public Function isThere(item As String) As Boolean
' Returns True if item is in the list
   Dim index As Integer = 0
   While((index < numItems) And (listItems(index).compareTo(item) <> 0))
      index += 1
   End While
   Return (index < numItems)
End Sub
```

This algorithm is called a *sequential* or *linear search* because we start at the beginning of the list and look at each item in sequence. We stop the search as soon as we find the item we are looking for (or when we reach the end of the list, concluding that the desired item is not present in the list).

We can use this algorithm in any program requiring a list search. In the form shown, it searches a list of `String` components, but the algorithm works for any class that has a `compareTo` method.

Let's look again at the heading for the operation that puts an item into the list.

```
Public Sub insert (item As String)
' Adds item to the list
' Assumption: item is not already in the list
```

Is there anything in the documentation that says where each new item should go? No, this is an unsorted list. Where we put each new item is up to the implementer. In this case, let's put each new item in the easiest place to reach: the next free slot in the array. Therefore, we can store a new item into `listItems(numItems)`—and then increment `numItems`.

This algorithm brings up a question: Do we need to check that there is no room in the list for the new item? We have two choices. The `insert` method can test `numItems` against `listItems.Length` and throw an exception if there isn't room, or we can let the client code make the test before calling `insert`. Our documentation is incomplete because it does not specify what occurs in this situation. Let's make the client code responsible for checking the `isFull` operation before an insertion. If the client attempts to insert an item into a full list, the operation fails and the list is unchanged.

This algorithm is so simple, we just go directly to code.

```
Public Sub insert(item As String)
' If the list is not full, puts item in the last position in the list:
' otherwise list is unchanged.
   If (Not listItems.IsFull())
      listItems(numItems) = item
      numItems += 1
   End If
End Sub
```

Deleting a component from a list consists of two parts: finding the item and removing it from the list. We can use the same algorithm we used for `isThere` to look for the item. We know from the documentation that the item may or may not be in the list. If we find it, how do we remove it? We shift every item after it up one array slot.

Delete

Set index to location of item to be deleted if found
If found
  Shift remainder of list up
  Decrement numItems

shiftUp

index is the location of the item to be deleted
Set listItems(index) to listItems(index+1)
Set listItems(index+1) to listItems(index+2)
.
.
.
Set listItems(numItems–2) to listItems(numItems–1)

We can implement this algorithm using a `For` loop.

```
Public Sub Delete(item As String)
' Removes item from the list if it is there
   Dim index As Integer = 0
   Dim found As Boolean = False
   While (index < numItems And Not(found))
      If (listItems(index).compareTo(item) = 0) Then
         found = True
      Else
         index += 1
```

```
      End If
   End While
   If (found) Then
   ' Shift remainder of list up to delete item
      Dim count As Integer
      For count = index To numItems-1
         listItems(count) = listItems(count+1)
      Next count
      numItems -= 1
   End If
End Sub
```

The `resetList` method is analogous to the open operation for a file in which the file pointer is positioned at the beginning of the file, so that the first input operation accesses the first component of the file. Each successive call to an input operation gets the next item in the file. Therefore, `resetList` must initialize `currentPos` to the first item in the list. Where is the first item in an array-based list? In the 0th position. The `getNextItem` operation is analogous to an input operation; it accesses the current item and then increments `currentPos`. When `currentPos` reaches `numItems`, we reset it to 0.

```
Public Sub resetList() As String
' The iteration is initialized by setting currentPos to 0
   currentPos = 0
End Sub

Public Sub getNextItem() As String
' Returns the item at the currentPos position; increments currentPos;
' resets current position to first item after the last item is returned
' Assumption: No transformers have been invoked since last call
   Dim next As String = listItems(currentPos)
   If (currentPos = numItems-1) Then
      currentPos = 0
   Else
      currentPos += 1
   End If
   Return next
End Sub
```

Both of the methods change `currentPos`. Shouldn't we consider them transformers? We could certainly argue that they are, but their intention is to set up an iteration through the items in the list, returning one item at a time to the client.

Test Plan

The documentation for the methods in class `List` helps us identify the tests necessary for a black-box testing strategy. The code of the methods determines a clear-box testing

strategy. To test the List class implementation, we use a combination of black-box and clear-box strategies. We first test the constructor by testing to see if the list is empty (a call to Length returns 0).

The methods Length, Insert, and Delete must be tested together. That is, we insert several items and delete several items. How do we know that the Insert and Delete work correctly? We must write an auxiliary method printList that iterates through the list using Length and getNextItem to print out the values. We call printList to check the status of the list after a series of insertions and deletions. To test the isFull operation, we must test it when the list is full and when it is not. We must also call Insert when the list is full to see that the list is returned unchanged.

Are there any special cases that we need to test for Delete and isThere? We look at the end cases. What are the end cases in a list? The item is in the first position in the list, the item is in the last position in the list, and the item is the only one in the list. So we must be sure that our Delete can correctly delete items in these positions. We must check that isThere can find items in these same positions and correctly determine that values are not in the list.

These operations are summarized in the following test plan. The tests are shown in the order in which they should be performed.

Operation to be Tested Description of Action	Input Values	Expected Output	Observed Output
Constructor (4)		0	
print Length			
Insert			
Insert four items and print	mary, john, betty, ann	mary, john, betty, ann	
Insert item and print		mary, john, betty, ann	
isThere			
isThere susy and print whether found		item is not found	
isThere mary and print whether found		item is found	
isThere ann and print whether found		item is found	
isThere betty and print whether found		item is found	
isFull			
invoke (list is full)		list is full	
delete ann and invoke		list is not full	
Delete			
Print		mary, john, betty	
Delete betty and print		mary, john,	
Delete mary and print		john	
Delete john and print		(empty)	
isEmpty		yes	

But what about testing `resetList` and `getNextItem`? They do not appear explicitly in the test plan, but they are tested each time the auxiliary method `printList` is called to print the contents of the list. We do however have to add one additional test involving the iterator. We must print out length plus 1 items to test whether the current position is reset after the last item is returned.

To implement this test plan, we must construct a test driver that carries out the tasks outlined in the first column of the plan. We can make the test plan a document separate from the driver, with the last column filled in and initialed by a person running the program and observing the screen output. Or we can incorporate the test plan into the driver as comments and have the output go to a file. The key to properly testing any software is in the plan: It must be carefully thought out and it must be written.

12.3 Sorting the List Items

`getNextItem` presents the items to the user in the order in which they were inserted. Depending on what we are using the list for, there might be times when we want to rearrange the list components into a certain order before an iteration. For example, if we are using the list for wedding invitations, we might want to see the names in alphabetical order. Arranging list items in order is a very common operation and is known in software terminology as **sorting**.

Sorting Arranging the components of a list in order (for instance, words into alphabetical order or numbers into ascending or descending order)

If you were given a sheet of paper with a column of 20 names on it and asked to write the numbers in ascending order, you would probably do the following:

1. Make a pass through the list, looking for the lowest name (the one that comes first alphabetically).
2. Write it on the paper in a second column.
3. Cross the name off the original list.
4. Repeat the process, always looking for the lowest name remaining in the original list.
5. Stop when all the names have been crossed off.

We could implement this algorithm as a client code, using `getNextItem` to go through the list searching for the lowest. When we find it, we could insert it into another list and delete it from the original. However, we would need two lists—one for the original list and a second for the sorted list. In addition, the client would have destroyed the original list. If the list is large, we might not have enough memory for two copies even if one is empty. A better solution is to derive a class from `List` that adds a method that sorts the values in the list. Because the data fields in `List` are protected, we can inherit them and access them directly. By accessing the values directly, we can keep from having to have two lists.

Class Name: ListWithSort	Superclass: List	Subclasses:

Responsibilities	Collaborations
Create itself (maxItems), Constructor	super
Sort the items in the list, Transformer	string

Responsibility Algorithms for Class `ListWithSort`

The constructor takes the maximum number of items and calls the list's constructor. None of the other methods need to be overridden.

Going back to our by-hand algorithm, we can search `listItems` for the smallest value, but how do we "cross off" a list component? We could simulate crossing off a value by replacing it with an empty string. We thus set the value of the crossed-off item to something that doesn't interfere with the processing of the rest of the components. However, a slight variation of our hand-done algorithm allows us to sort the components *in place*. We do not have to use a second list; we can put a value into its proper place in the list by having it swap places with the component currently in that list position.

We can state the algorithm as follows. We search for the smallest value in the array and exchange it with the component in the first position in the array. We search for the next-smallest value in the array and exchange it with the component in the second position in the array. This process continues until all the components are in their proper places. Figure 12.4 illustrates how this algorithm works.

selectSort

For count going from 0 through numItems − 2
  Find the minimum value in listItems(count)...listItems(numItems−1)
  Swap minimum value with listItems(count)

Observe that we perform `numItems-1` passes through the list because `count` runs from 0 through `numItems-2`. The loop does not need to be executed when `count` equals `numItems-1` because the last value, `listItems(numItems-1)`, is in its proper place after the preceding components have been sorted.

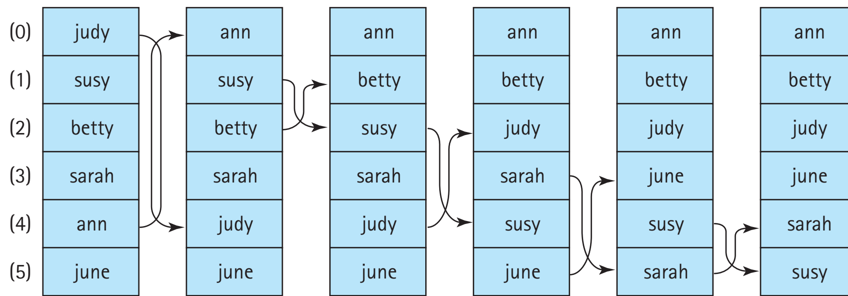


Figure 12.4 *Straight selection sort*

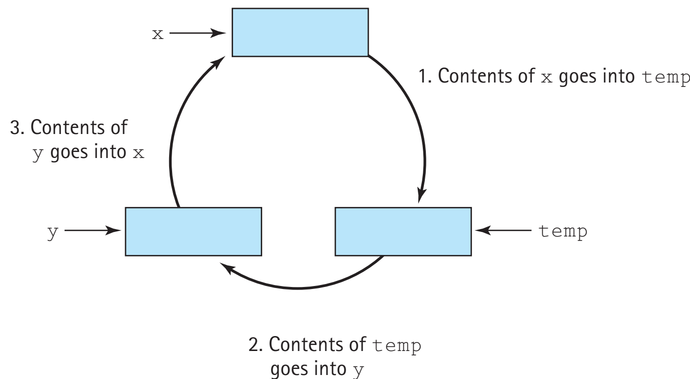


Figure 12.5 *Swapping the contents of two variables, x and y*

This sort, known as the *straight selection sort*, belongs to a class of sorts called *selection sorts*. There are many types of sorting algorithms. Selection sorts are characterized by finding the smallest (or largest) value left in the unsorted portion at each iteration and swapping it with the value indexed by the iteration counter. Swapping the contents of two variables requires a temporary variable so that no values are lost (see Figure 12.5).

Class `ListWithSort`

We are now ready to code our derived class. We need to include in the documentation that the alphabetic order may be lost with future insertions.

```
Public Class ListWithSort
Inherits List
   ' The items in the list are rearranged into ascending order
   ' This order is not preserved in future insertions

  Public Sub New(maxItems As Integer)
  ' An empty list object has been instantiated with room for maxItems items
```

```
    numItems = 0
    Dim listItems(maxItems) As String
    currentPos = 0
  End Sub

  Public Sub New()
  ' An empty list object has been instantiated with room for 100 items
    numItems = 0
    Dim listItems(100) As String
    currentPos = 0
  End Sub

  Public Sub selectSort()
  ' Arranges list items in ascending order
  ' Selection sort algorithm is used.
    Dim temp As Integer
    Dim passCount As Integer
    Dim searchIndex As Integer
    Dim minIndex As Integer
    For passCount = 0 To numItems - 1
       minIndex = passCount
       For searchIndex = passCount + 1 To numItems - 1
         If (listItems(searchIndex).compareTo(listItems(minIndex) < 0) Then
            minIndex = searchIndex
         End If
       Next searchIndex
       ' Swap listItems(minIndex) and listItems(passCount)
       temp = listItems(minIndex)
       listItems(minIndex) = listItems(passCount)
       listItems(passCount) = temp
    Next passCount
  End Sub

End Class
```

Note that with each pass through the outer loop in `selectSort`, we are looking for the minimum value in the rest of the array (`listItems(passCount)` through `listItems(numItems-1)`). Therefore, `minIndex` is initialized to `passCount` and the inner loop runs from `searchIndex` equal to `passCount+1` through `numItems-1`. Upon exit from the inner loop, `minIndex` contains the position of the smallest value. (Note that the *If* statement is the only statement in the loop.)

This method may swap a component with itself, which occurs if no value in the remaining list is smaller than `listItems(passCount)`. We could avoid this unnecessary swap by checking to see if `minIndex` is equal to `passCount`. Because this comparison is made in each iteration of the outer loop, it is more efficient not to check for this possibility and just to swap something with itself occasionally.

This algorithm sorts the components into ascending order. To sort them into descending order, we must scan for the maximum value instead of the minimum value. Simply changing the test in the inner loop from less than to greater than can accomplished this. Of course, `minIndex` would no longer be an appropriate identifier and should be changed to `maxIndex`.

12.4 Sorted List

It is important to note that `ListWithSort` does not provide the user with a sorted list class. The `Insert` and `Delete` algorithms do not preserve ordering by value. The `Insert` operation places a new item at the end of the list, regardless of its value. After `selectSort` has been executed, the list items remain in sorted order only until the next insertion or deletion takes place. Of course, the client could sort the list after every insertion, but this is inefficient. Let's now look at a sorted list design in which all the list operations cooperate to preserve the sorted order of the list components.

Brainstorming Sorted List

There is nothing about order in the design for class `List`. If we want the list items kept in sorted order, we need to specify this. Let's go back to the CRC card design for class `List` and indicate that we want the list to be sorted.

Class Name: List	Superclass: Object	Subclasses:

Responsibilities	Collaborations
Create itself (maxItems)	None
Is list full? Observer	None
return boolean	
Is list empty? Observer	None
return boolean	
Know length, Observer	None
return int	
Is an item in the list? Observer	None
return boolean	
Insert into list (item) keeping the list sorted, Transformer	String
Delete from list (item) keeping the list sorted, Transformer	String
Look at each item in sorted order, Iterator	String

The first thing to notice is that the observers do not change. They are the same whether the list sorted by value or not. The transformers `Insert` and `Delete` and the iterator now have additional constraints. Rather than designing an entirely new class, we can derive `SortedList` from class `List`, overriding those methods whose implementations need changing.

Class Name: SortedList	Superclass: List	Subclasses:

Responsibilities	Collaborations
Create itself (maxItems)	None
Insert into list (item) keeping the list sorted, Transformer	String
Delete from list (item) keeping the list sorted, Transformer	String
Look at each item in sorted order, Iterator	String

Responsibility Algorithms for Class `SortedList`

Let's look first at `Insert`. Figure 12.6 illustrates how it should work.

The first item inserted into the list can go into the first position. If there is only one item, the list is sorted. If a second item being inserted is less than the first item, the first item must be moved into the second position and the new item put into the first position. If the second item is larger, it goes into the second position. If we add a third item that is smaller than the first item, the other two are shifted down one and the third item goes into the first position. If the third item is greater than the first item but less than the second, the second is shifted down and the third item goes into the second position. If the third item is greater than the second item, it goes into the third position.

To generalize, we start at the beginning of the list and scan until we find an item greater than the one we are inserting. We shift that item and the rest of the items in the list down one position to make room for the new item. The new item goes in the list at that point.

Insert

```
If (list is not full)
   While place not found AND more places to look
      If item > current item in the list
         Increment current position
```

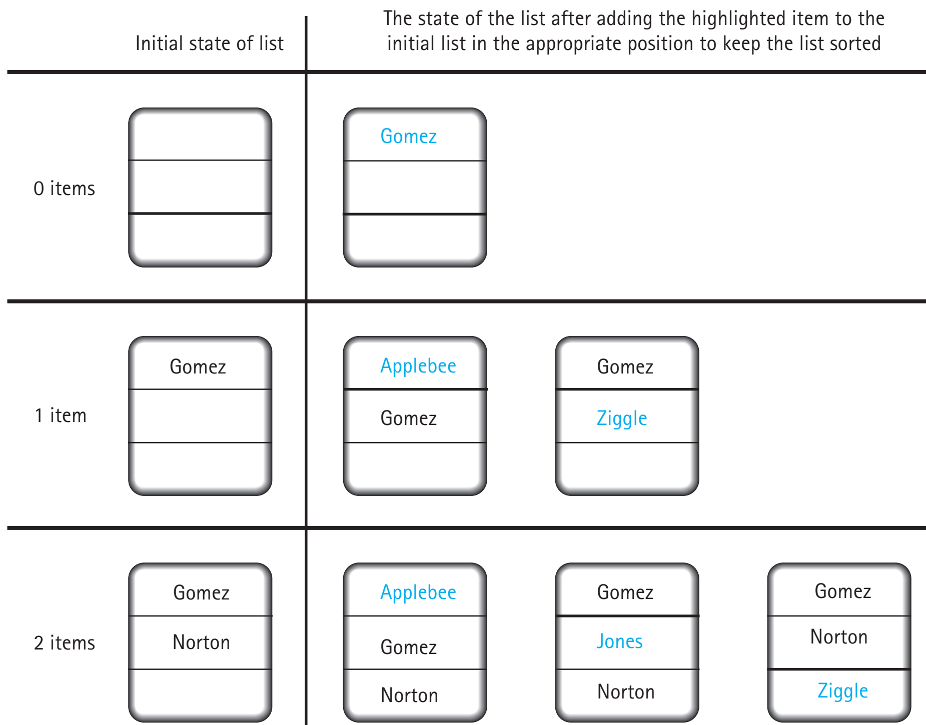


Figure 12.6 *All of the different places where an item can be inserted into a sorted list, starting with 0, 1, or 2 items already in the list.*

```
    Else
       Place found
Shift remainder of the list down
Insert item
Increment numItems
```

If we assume that `index` is the place where item is to be inserted, the algorithm for shifting the remainder down is as follows.

shiftDown

Set listItems(numItems) to listItems(numItems − 1)
Set listItems(numItems – 1) to listItems(numItems − 2)
.
.
.
Set listItems(index+1) to listItems(index)

This algorithm is illustrated in Figure 12.7. Like the `shiftUp` algorithm, `shiftDown` can be implemented using a *`For`* loop.

This algorithm is based on how we would accomplish this task by hand. Often, such an adaptation is the best way to solve a problem. However, in this case, further thought reveals a slightly better way. Notice that we search from the front of the list (people always do), and we shift down from the end of the list upward. We can combine the searching and shifting by beginning at the *end* of the list.

If `item` is the new item to be inserted, compare `item` to the value in `listItems(numItems-1)`. If `item` is *less*, put `listItems(numItems-1)` into `list-`

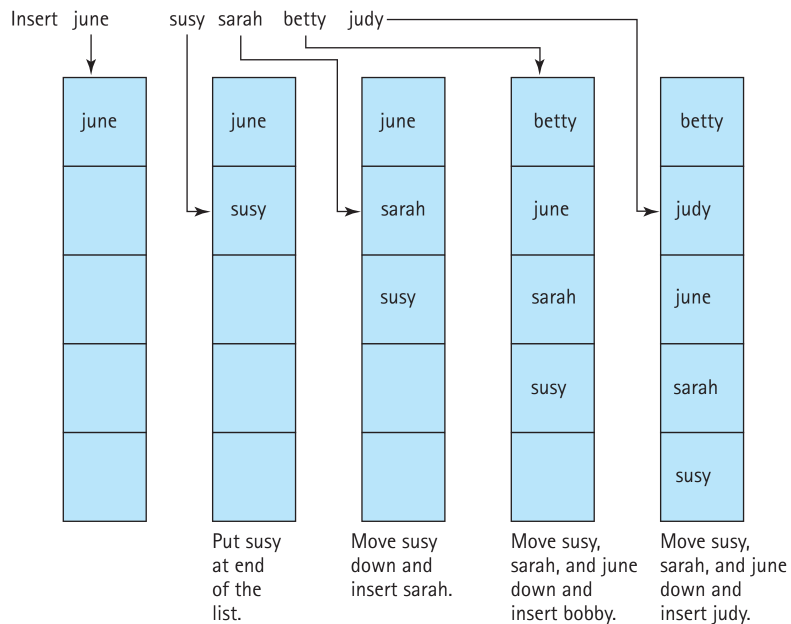


Figure 12.7 *Inserting into a sorted list*

`Items(numItems)` and compare item to the value in `listItems(numItems-2)`. This process continues until you find the place where `item` is greater than (or equal to, if duplicates are allowed) the list item. Store `item` directly after it. Here is the algorithm:

Insert (revised)

```
If (list is not full)
  Set index to numItems − 1
  While index >= 0 And (item.compareTo(listItems(index)) < 0)
    Set listItems(index + 1) to listItems(index)
    Decrement index
  Set listItems(index + 1) to item
  Increment numItems
```

Notice that this algorithm works even if the list is empty. When the list is empty, `numItems` is 0 and the body of the *While* loop is not entered. So item is stored into `listItems(0)`, and `numItems` is incremented to 1. Does the algorithm work if `item` is the smallest? The largest? Let's see. If `item` is the smallest, the loop body is executed `numItems` times, and `index` is −1. Thus, `item` is stored into position 0, where it belongs. If `item` is the largest, the loop body is not entered. The value of `index` is still `numItems` − 1, so item is stored into `listItems(numItems)`, where it belongs.

Are you surprised that the general case also takes care of the special cases? This situation does not happen all the time, but it occurs sufficiently often that we find it is good programming practice to start with the general case. If we begin with the special cases, we may still generate a correct solution, but we may not realize that we don't need to handle the special cases separately. So begin with the general case, then treat as special cases only those situations that the general case does not handle correctly.

The methods `Delete` and `getNextItem` must maintain the sorted order—but they already do! An item is deleted by removing it and shifting all of the items larger than the one deleted up one position, and `getNextItem` only returns a copy of an item, it does not change an item. Only `Insert` needs to be overridden in derived class `SortedList`.

```
Public Class SortedList
Inherits List
Public Sub New(maxItems As Integer)
' An empty list object has been instantiated with room for maxItems item
   Dim numItems = 0
   Dim listItems(maxItems) As String
```

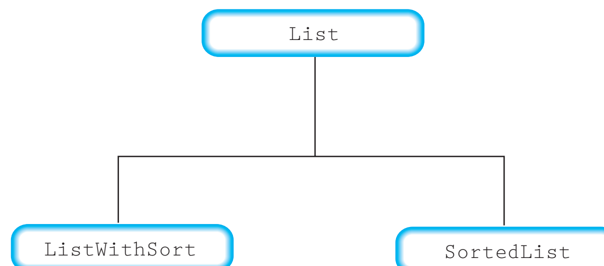
```
    currentPos = 0
End Sub
    Public Sub New()
  ' An empty list object has been instantiated with room for 100 items
  numItems = 0
  Dim listItems(100) As String
  currentPos = 0
  End Sub

    Public Sub Insert(item As String)
  ' If the list is not full, puts item in its proper place in the
  ' list, otherwise list is unchanged
     If (Not listItems.isFull()) Then
        Dim index As Integer = numItems - 1
        while (index >= 0 And (item.compareTo.(listItems(index)) < 0))
           listItems(index+1) = listItems(index)
           index -= 1
        end while
        listItems(index+1) = item
        numItems += 1
     End If
  End Sub
End Class
```

Test Plan

The same test plan can be used for the sorted list that we used for the unsorted version. The only difference is that in the expected output column, the list items should appear in sorted order.

12.5 The `List` Class Hierarchy and Abstract Classes

We have create a hierarchy with class `List` at the top and two derived classes. We can visualize the hierarchy as follows:



ListWithSort is a List. SortedList is a List. ListWithSort is not a SortedList, and SortedList is not a ListWithSort.

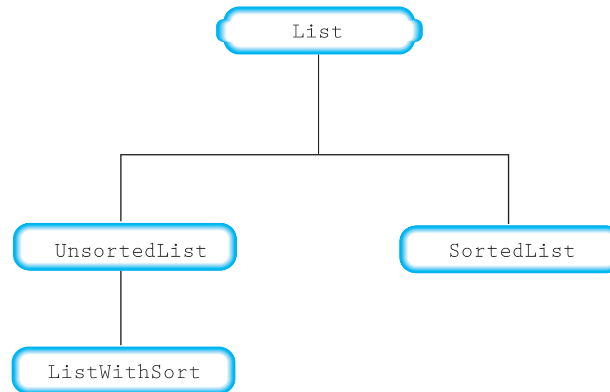
We could have organized the hierarchy using an abstract class. Recall from Chapter 8 that an abstract class is a class that is headed by the word MustInherit and leaves one or more methods incomplete. An abstract class cannot be instantiated. Another class must extend the abstract class and implement all of the abstract methods. We could have implemented the observers and iterator in the abstract class and left the implementation of the transformers to the derived class. Then the unsorted list and the sorted version could both inherit from the abstract class. Methods that are to be implemented by the derived class begin with the MustOverride keyword. An abstract method that is derived from the abstract class must include the Overrides modifier. The class and method headings would be as follows:

```
MustInherit Class list
   Public Sub New()
   End Sub

   Public Sub New(ByVal maxItems As Integer)
   End Sub

   Public Function isFull() As Boolean
   End Function

   Public Function isEmpty() As Boolean
   End Function

   Public Function Length() As Integer
   End Function

   Public Function isThere(ByVal item As String) As Boolean
   End Function

   Public Sub resetList()
   End Sub

   Public Function getNextItem() As String
   End Function

   MustOverride Sub Delete(ByVal item As String)
   MustOverride Sub Insert(ByVal item As String)

End Class

Class UnsortedList
   Inherits List
```

```
    Public Sub New()
    End Sub

    Public Sub New(ByVal maxItems As String)
    End Sub

    Public Overrides Sub Delete(ByVal item As String)
    End Sub

    Public Overrides Sub Insert(ByVal item As String)
    End Sub

End Class

Class SortedList
    Inherits List
    Public Sub New()
    End Sub

    Public Sub New(ByVal maxItems As Integer)
    End Sub

    Public Overrides Sub Delete(ByVal item As String)
    End Sub

    Public Overrides Sub Insert(ByVal item As String)
    End Sub

End Class

Class ListWithSort
    Inherits UnsortedList

    Public Sub New()
    End Sub

    Public Sub New(ByVal maxItems As Integer)
    End Sub

    Public Sub selectSort()
    End Sub

End Class
```

Notice that the classes that inherit from an abstract class cannot be declared as `Public`.

Under these new conditions, the class hierarchy looks like this:



12.6 Searching

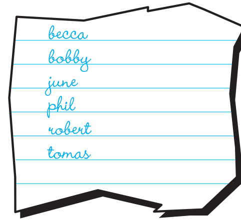
In our `SortedList` class we overrode the `Insert` method, the only method that had to be rewritten to keep the list in sorted form. However, if the list is in sorted form, we can perform a more efficient search. In this section, we look at two searching algorithms that depend on the list items being in sorted order.

Sequential Search

The `isThere` algorithm assumes that the list to be searched is unsorted. A drawback to searching an unsorted list is that we must scan the entire list to discover that the search item is not there. Think what it would be like if your city telephone book contained people's names in random rather than alphabetical order. To look up Marcus Anthony's phone number, you would have to start with the first name in the phone book and scan sequentially, page after page, until you found it. In the worst case, you might have to examine tens of thousands of names only to find out that Marcus's name is not in the book.

Of course, telephone books are alphabetized, and the alphabetical ordering makes searching easier. If Marcus Anthony's name is not in the book, you discover this fact quickly by starting with the A's and stopping the search as soon as you have passed the place where his name should be. Although the sequential search algorithm in `isThere` works in a sorted list, we can make the algorithm more efficient by taking advantage of the fact that the items are sorted.

How does searching in a sorted list differ from searching in an unsorted list? When we search for an item in an unsorted list, we won't discover that the item is missing until we reach the end of the list. If the list is already sorted, we know that an item is missing when we pass the place where it should be in the list. For example, if a list contains the values



and we are looking for judy, we need only compare judy with becca, bobby, and june to know that judy is not in the list.

If the search item is greater than the current list component, we move on to the next component. If the item is equal to the current component, we have found what we are looking for. If the item is less than the current component, then we know that it is not in the list. In either of the last two cases, we stop looking. In our original algorithm, the loop conditions were that the index was within the list and the corresponding list item was not the one searched for. In this algorithm, the second condition must be that the item being searched for is less than the corresponding list item. However, determining whether the item is found is a little more complex. We must first assert that the index is within the list and, if that is true, assert that the search item is equal to the corresponding list item.

isThere (in a sorted list)

Set index to 0
While index is within the list AND item is greater than listItems(index)
  Increment index
Return (index is within the list AND item is equal to listItems(index))

Why can't we just test to see if `item` is equal to `listItems(index)` at the end of the loop? This works on all cases but one. What happens if `item` is larger than the last element in the list? We would exit the loop with `index` equal to `numItems`. Trying to access `listItems(index)` would then cause the program to crash with the index out of range error. Therefore, we must check on the value of index first.

```
Public Sub isThere(item As String) As Boolean
' Returns True if item is in the list, False otherwise
' Assumption: List items are in ascending order
   Dim index As Integer = 0
   while (index < numItems And item.compareTo(listItems(index)) > 0) Then
      index += 1
   end while
   return (index < numItems And item.compareTo(listItems(index)) = 0)
End Sub
```

On average, searching a sorted list in this way takes the same number of iterations to find an item as searching an unsorted list. The advantage of this new algorithm is that we find out sooner if an item is missing. Thus, it is slightly more efficient. There is another search algorithm that works only on a sorted list, but it is more complex: a binary search. However, the complexity is worth it.

Binary Search

The *binary search* algorithm on a sorted list is considerably faster both for finding an item and for discovering that an item is missing. A binary search is based on the principle of successive approximation. The algorithm divides the list in half (divides by 2—that's why it's called a *binary* search) and decides which half to look in next. Division of the selected portion of the list is repeated until the item is found or it is determined that the item is not in the list.

This method is analogous to the way in which we look up a name in a phone book (or word in a dictionary). We open the phone book in the middle and compare the name with one on the page that we turned to. If the name we're looking for comes before this name, we continue our search in the left-hand section of the phone book. Otherwise, we continue in the right-hand section of the phone book. We repeat this process until we find the name. If it is not there, we realize that either we have misspelled the name or our phone book isn't complete. See Figure 12.8.

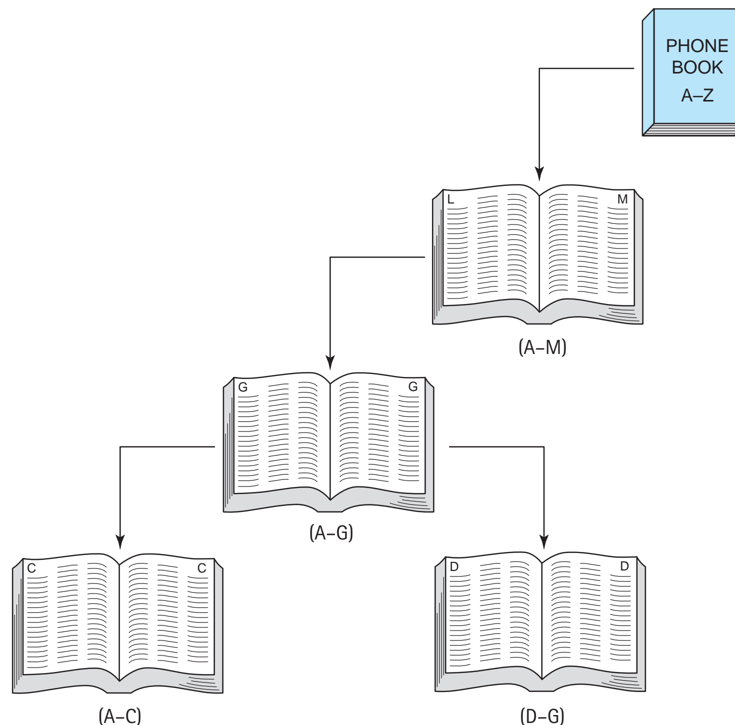


Figure 12.8 *A binary search of the phone book*

We start with the whole list (indexes 0 through `numItems-1`) and compare our search value to the middle list item. If the search item is less than the middle list item, we continue the search in the first half of the list. If the search item is greater than the middle list item, we continue the search in the second half of the list. Otherwise, we have found a match. We keep comparing and redefining the part of the list in which to look (the search area) until we find the item or the search area is empty. Let's write the algorithm bounding the search area by the indexes `first` and `last`. See Figure 12.9.

Binary Search

```
Set first to 0
Set last to numItems − 1
Set found to false
While search area is not empty and Not found
   Set middle to (first + last) divided by 2
   If (item is equal to listItems(middle))
      Set found to True
   Else If (item is less than listItems(middle))
      Set last to middle − 1     ' Look in first half
   Else
      Set first to middle + 1    ' Look in second half
```

This algorithm should make sense. With each comparison, at best, we find the item for which we are searching; at worst, we eliminate half of the remaining list from consideration. Before we code this algorithm, we need to determine when the search area is

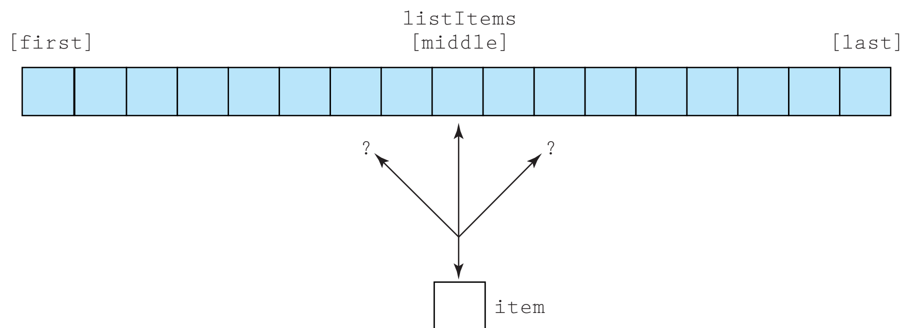


Figure 12.9 *Binary search*

empty. If the search area is between `listItems(first)` and `listItems(last)`, then this area is empty if `last` is less than `first`.

Let's do a walk-through of the binary search algorithm. The item being searched for is "`bat`". Figure 12.10a shows the values of `first`, `last`, and `middle` during the first iteration. In this iteration, "`bat`" is compared with `"dog"`, the value in `listItems(middle)`.

Because `"bat"` is less than (comes before) `"dog"`, `last` becomes `middle-1` and `first` stays the same. Figure 12.10b shows the situation during the second iteration. This time, `"bat"` is compared with `"chicken"`, the value in `listItems(middle)`. Because `"bat"` is less than (comes before) `"chicken"`, `last` becomes `middle-1` and `first` again stays the same.

In the third iteration (Figure 12.10c), `middle` and `first` are both 0. The item `"bat"` is compared with `"ant"`, the item in `listItems(middle)`. Because `"bat"` is greater than (comes after) `"ant"`, `first` becomes `middle+1`. In the fourth iteration (Figure 12.10d), `first`, `last`, and `middle` are all the same. Again, `"bat"` is compared with the item in `listItems(middle)`. Because `"bat"` is less than `"cat"`, `last` becomes `middle-1`. Now that `last` is less than `first`, the process stops; `found` is `False`.

The binary search is the most complex algorithm that we have examined so far. The following table shows `first`, `last`, `middle`, and `listItems(middle)` for searches of the items `"fish"`, `"snake"`, and `"zebra"`, using the same data as in the previous example. Examine the results in this table carefully.

Iteration	first	last	middle	listItems[middle]	Terminating Condition
item: fish					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	6	7	6	fish	found is `True`
item: snake					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	rat	
Fourth	10	10	10	snake	found is `True`
item: zebra					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	rat	
Fourth	10	10	10	snake	
Fifth	11	10			last < first

Notice in the table that whether we search for `"fish"`, `"snake"`, or `"zebra"`, the loop never executed more than four times. It never executes more than four times in a list of 11 components because the list is being cut in half each time through the loop.

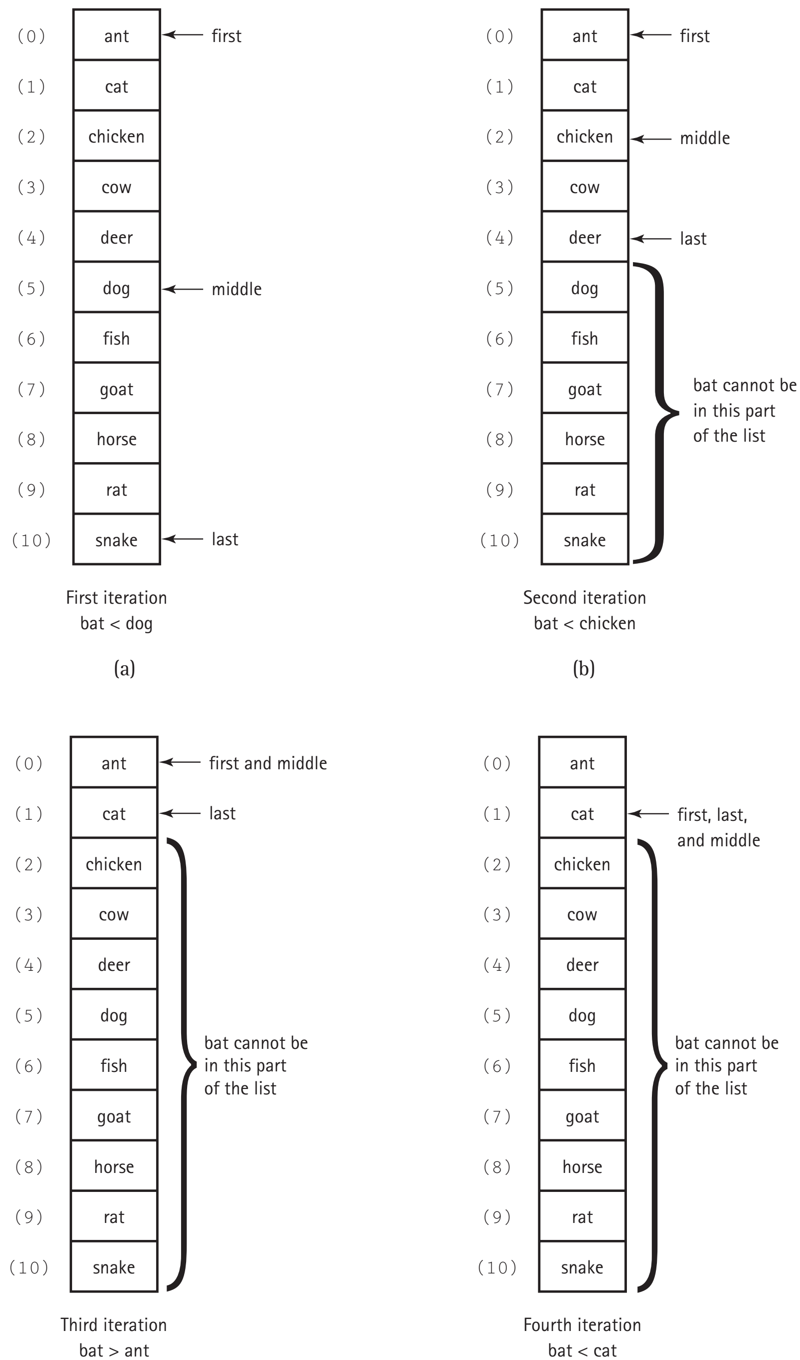


Figure 12.10 *Walk-through of binary search algorithm*

The table below compares a sequential search and a binary search in terms of the average number of iterations needed to find an item.

	Average Number of Iterations	
Length	Sequential Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1,000	500.5	9.0
10,000	5000.5	12.0

If the binary search is so much faster, why not use it all the time? It certainly is faster in terms of the number of times through the loop, but more computations are performed within the binary search loop than in the other search algorithms. So if the number of components in the list is small (say, less than 20), the sequential search algorithms are faster because they perform less work at each iteration. As the number of components in the list increases, the binary search algorithm becomes relatively more efficient.

Here is the code for `isThere` that uses the binary search algorithm.

```
Public Sub isThere(item As String)
' Returns True if item is in the list, False otherwise
' Binary search algorithm is used
' Assumption: List items are in ascending order
   Dim first As Integer = 0
   Dim last As Integer = numItems - 1
   Dim middle As Integer
   Dim found As Boolean = False
   While (last >= first And (Not Found))
      middle = (first + last) / 2
      If (item.compareTo(listItems(middle)) = 0) then
         found = True
      ElseIf (item.compareTo(listItems(middle)) < 0) then
         ' Item not in listItems(middle)...listItems(last)
         last = middle - 1
      Else
         ' Item not in listItems(first)...listItems(middle)
         first = middle + 1
   End While
   Return found
End Sub
```

12.7 Generic Lists

Generic lists are lists where the operations are defined, but the objects in the list are not. Although we called the components of our lists "items," they are of type `String`. Is it possible to construct a truly general-purpose list where the items can be anything? For example, could we have a list of `Name` objects as defined in Chapter 7 or `Address` objects as defined in Chapter 8? Yes, we can. All we have to do is declare the objects on the list to be `Object`. What is `Object`? It is the base data type for all the other data types we have used in this book and can be used to define a generic object. An object declared as type `Object` can be any legal Visual Basic.NET data type. Now, let's see how we can use it to make our lists generic.

The `Object` Type

All data types in Visual Basic.NET inherit from the `Object` type. Normally, when we declare an object, we pick one of the inherited data types for the object. We can, however, declare an object to be of type `Object`. An object of type `Object` can then be assigned any data type. Look at the following example.

```
Dim item As Object
item = 12000
item = "Mike"
```

The two assignment statements above are perfectly legal when the variable is of type `Object`. You should use the `Object` data type only when you don't know in advance what data type a variable will receive.

To make our `List` class as generic as possible, we replace `String` with `Object` throughout the class. This means that any object can be passed as a parameter to `Insert`, `Delete`, or `isThere`. This change also means that the array must hold `Object` objects, and `getNextItem` must return an `Object` object. Here is the complete abstract class `List`.

```
MustInherit Class List
   Protected listItems() As Object
   Protected numItems As Integer
   Protected currentPos As Integer

   Public Sub New(ByVal maxItems As Integer)
      ' Instantiates an empty List object
      ' with room for maxItems items
      numItems = 0
      Dim listItems(maxItems) As Object
      currentPos = 0
   End Sub
```

```
Public Sub New()
   numItems = 0
   Dim listItems(100) As Object
   currentPos = 0
End Sub

Public Function isFull() As Boolean
   ' Returns True if there is not room for
   ' another component; False otherwise

   Return (listItems.Length = numItems)
End Function

Public Function isEmpty() As Boolean
   ' Returns True if there are no components in
   ' the list; False otherwise

   Return (numItems = 0)
End Function

Public Function Length() As Integer
   ' Returns the number of components in the list

   Return numItems
End Function

MustOverride Function isThere(ByVal item As Object) As Boolean
' Returns True if item is in the list; False otherwise

' Transformers
MustOverride Sub Insert(ByVal item As Object)
' If list is not full, insert item into list;
' otherwise, list is unchanged.
' Assumption: item is not already in the list

MustOverride Sub Delete(ByVal item As Object)
' Removes item from the list if it is there

' Iterator Pair
Public Sub resetList()  ' Prepare for iteration
   currentPos = 0
End Sub
```

```
    Public Function getNextItem() As Object
        ' Returns the item at the currentPos position;
        ' resets current position to first item after
        ' the last item is returned
        ' Assumption: No transformers have been invoked since
        ' last call
        Dim nextItem As Object = listItems(currentPos)
        If (currentPos = numItems - 1) Then
            currentPos = 0
        Else
            currentPos += 1
        End If
        Return nextItem
    End Function

End Class
```

Notice that we have made the `isThere` method abstract. This way the derived class can determine which searching algorithm to use.

Polymorphism

We have discussed polymorphism several times, as it is one of the major features of object-oriented programming. In a hierarchy of classes, polymorphism enables us to override a method name with a different implementation in a derived class. Thus, there can be multiple forms of a given method within the hierarchy (literally, polymorphism means *having multiple forms*.)

The Visual Basic.NET compiler decides which form of a polymorphic instance method to use by looking at the class of its associated instance. For example, if `compareTo` is associated with a `String` variable, then the version of `compareTo` defined in class `String` is called.

Thus far, this is all straightforward. But consider the case in which `compareTo` is applied to an object that has been passed as a parameter declared to be `Object`. The abstract `Insert` method that we defined in the last section is precisely the example we have in mind.

```
MustOverride Sub Insert(byval Item As Object)
```

An instance of any class can be passed as an argument to this parameter. The class of the argument object determines which form of `compareTo` is actually called within `Insert`. At compile time, however, there is no way for the Visual Basic.NET compiler to determine the class of the argument object. Instead, it must insert Bytecode that identifies the argument's class at run time and then calls the appropriate method. Programming language designers call this **dynamic binding**. When the compiler can identify the appropriate form of the method to use, it is called a **static binding**.

Dynamic binding Determining at run time which form of a polymorphic method to call

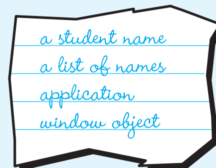
Static binding Determining at compile time which form of a polymorphic method to call

Problem-Solving Case Study

Exam Attendance

Problem You are the grader for a U.S. government class of 200 students. The instructor has asked you to prepare two lists: an alphabetical list of the students taking an exam and an alphabetical list of the students who have missed it. The catch is that she wants the lists before the exam is over. You decide to write an application for your notebook computer that takes each student's name as the student enters the exam room and prints the lists of absentees and attendees for your instructor.

Brainstorming The words *student*, *name*, and *list* are sprinkled throughout the problem statement. The fundamental objects, then, are lists of students, each represented by his or her name. We need a form object to hold the student names as they enter the exam. We also need an object that represents the person checking in the students. This object is the driver class for the application.

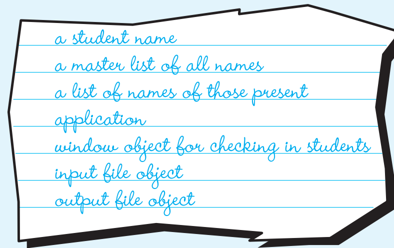


Filtering None of these classes overlap, so there is no consolidation that we can do. We may be missing a class, but we can't tell yet.

Scenarios Let's look at a by-hand algorithm to get a better picture of the processing. You could stand by the door, ask the students their names as they enter, and check off the names from a master list. After all of the students have entered, you make a list of those that were not checked off and a list of those that were. Of course! You must have a master list of all the students in the class.

How do you simulate "Ask the students their names"? You type each name on the keyboard of your laptop. How do you simulate "Check off"? Your program reads the name and deletes it from the master list. If you then insert the name into a list of those present, you have both lists ready for printing as soon as the last student signs in.

You check with your instructor, and she says that the master class list is available in file "`students.dat`". You must add an input file object for reading the master list of students and a list to store them in. What about an output file? Yes, you need that too. You can write the two lists to a file and take it down the hall to be printed while the students are taking the exam. What about the student names? You have class `Name` that you can use from Chapter 7.



CRC Cards This application makes use of two lists: the master list and the list of the attendees. We do not make a CRC card for this class because we can use one of the list classes developed in this chapter. Here is the CRC card for the application class. Let's call it class `ExamAttendance`.

Class Name: ExamAttendance	Superclass: Object	Subclasses:

Responsibilities	Collaborations
Prepare window object	Frame, Button, Label, Textfield
Create master list of students	Name, BufferedReader, List
Get a student name from the window	Name, Button
Delete name from master list	List, Name
Insert name into present list	List, Name
Print list of those present	List, Name, PrintWriter
Print list of those absent	List, Name, PrintWriter

Internal Data Representation Your application must prepare the initial list of students from the class-roster file, which is ordered by social security number. If you use an unsorted list for the list of all students, the list must be sorted before printing (class `ListWithSort`). You decide to use class `SortedList` instead. So the internal representation is two sorted lists of `Name` objects. The interface for class `Name` is repeated here.

```
Interface Name
    ' This interface defines a name consisting of three parts
    ' Constructors are not implemented in an interface but are
    ' included in comments here
    ' Public Sub New(firstName As String, middleName As String,
    '                lastName As String)
    ' Initializes a Name object with first, middle, and last
    ' name
```

```
    Function knowFirstName() As String
    Function knowMiddleName() As String
    Function knowLastName() As String

    ' Additional observers
    Function firstLastMid() As String
    Function lastFirstMid() As String
    Function Equal(ByVal otherName As Name) As Boolean
    Function compareTo(ByVal otherName As Name) As Integer

End Interface
```

Responsibility Algorithms Preparing the window object involves creating a form for the students to enter their names. The names are to be stored in a `Name` object, which require a first name, a middle name, and a last name. This means that the form must have three labels and three textboxes. There must be a button to signal the name is ready for input and a button to signal that the last student has entered the exam.

Prepare Window Objects

```
Add three labels to form
Add three textboxes to form
Add "Enter" button to form
Add "Quit" button to form
```

Creating the master list of students requires preparing the file for input, reading the names, storing them into a `Name` object, and inserting the object into the list. Reading the names requires that you know how the data is written in file "`students.dat`". The file format is first name, middle name or initial, last name, and social security number. There is exactly one space between each part of the name. If there is an initial rather than a middle name, it is not followed by a period. We should extend `StreamReader` to input this information and return a `Name` object. We design this class later.

Create MasterList

```
Prepare file
While more data
  Set name to datafile.getName()
  Insert name into masterList
```

The next three responsibilities, getting a student name, deleting the name from the master list, and inserting the name in the present list, take place in the event handler for the "Enter" button.

Get a Student Name

```
Set firstName to firstField.getText()
Set middleName to middleField.getText()
Set lastName to lastField.getText()
Set name to new Name(firstName, middleName, lastName)
```

Delete Name from Master List

```
presentList.Insert(name)
```

The last two responsibilities, printing the list of those present and printing the list of those absent, take place within the event handler for the "Quit" button. Because the algorithm for printing a list is identical in both cases, you write it as a helper module that takes the file name and the list. The event handler can also close the window.

Print List(outFile, list)

```
List.resetList()
For index going from 1 to List.Length()
   Set name to List.getNextItem()
   outFile.WriteLine(Name.firstLastMid())
```

Now you need to design the class that does the file input. It is derived from `StreamReader`. It must create itself, input a string, and break the string into the first name, middle name, and last name. It should also have a Boolean method that tells when the file is empty. Here is its CRC card for class `ExamDataReader`.

Class Name: ExamDataReader	Superclass: BufferedReader	Subclasses:

Responsibilities	Collaborations
Create itself, Constructor	None
Get name, Observer return name	BufferedReader
More data? Observer return boolean	BufferedReader

Because the fields of the name are delimited by blanks, you can use the `ReadLine` method in `StreamReader` to input the entire line. In order to separate out the first, middle, and last names of a student, you can use the methods provided in the `String` class. The `indexOf` method finds the spaces. The first name begins at the 0th position and ends at the position before the space. The original line is replaced by the line with the first name removed. This process is repeated to get the middle name and the last name. The social security number can just be ignored. This problem doesn't need it.

We must be sure to read the first line of data in the constructor and read at the end of `getName`. The More Data responsibility can then check to see if the last read was an empty string.

Get Name

Set index to dataLine.indexOf(" ")
Set firstName to dataLine.Substring(0, index)
Set dataLine to dataLine.Substring(index+1, dataLine.Length())
Set index to dataLine.indexOf(" ")
Set middleName to dataLine.Substring(0, index)
Set dataLine to dataLine.Substring(index+1, dataLine.Length())
Set index to dataLine.indexOf(" ")
Set lastName to dataLine.Substring(0, index)
Set name to new Name(firstName, middleName, lastName)
Set dataLine to Me.ReadLine()

Return name

```
Imports CName
Imports System.IO
Imports ListClass
Imports WindowsApplication6.ExamDataReader     ' Imports program class

Public Class ExamDataReader
   Dim dataLine As String
   Dim inFile As StreamReader
   Public Sub New(ByVal dataFile As String)
      Dim theFile As File
      inFile = theFile.OpenText(dataFile)
      dataLine = inFile.ReadLine()
   End Sub
   Public Function getName() As Name
      ' Data fields
      Dim firstName As String
      Dim middleName As String
      Dim lastName As String
```

```
      Dim name As Name
      Dim index As Integer
      ' Extract first name
      index = dataLine.IndexOf(" ")
      firstName = dataLine.Substring(0, index)
      dataLine = dataLine.Substring(index + 1, dataLine.Length())
      ' Extract middle name
      index = dataLine.IndexOf(" ")
      middleName = dataLine.Substring(0, index)
      dataLine = dataLine.Substring(index + 1, dataLine.Length())
      ' Extract last name
      index = dataLine.IndexOf(" ")
      lastName = dataLine.Substring(0, index)
      name = New Name(firstName, middleName, lastName)
      dataLine = Me.inFile.ReadLine()
      Return name
   End Function

   Public Function moreData() As Boolean
      Return (inFile.Peek <> -1)
   End Function

End Class


Public Class Form1
   Inherits System.Windows.Forms.Form
   Dim firstName As String           ' First name field
   Dim middleName As String          ' Middle name field
   Dim lastName As String            ' Last name field
   Dim aName As Name                 ' A name
   Dim masterList As SortedList      ' List of students
   Dim presentList As SortedList     ' List of those present
   Dim theFile As File               ' File object
   Dim dataFile As StreamReader      ' Master file of students
   Dim outFile As StreamWriter       ' File for printing
   'Dim dataFile As New ExamDataReader("students.dat")

#Region " Windows Form Designer generated code "

   Public Sub New()
      MyBase.New()

      ' This call is required by the Windows Form Designer.
      InitializeComponent()
```

```
      masterList = New SortedList(200)
      presentList = New SortedList(200)

' Get the master list of students
      While (dataFile.moreData())
         aName = dataFile.getName()
         masterList.insert(Name)
      End While
      ' Add any initialization after the InitializeComponent() call

   End Sub

   ' Form overrides dispose to clean up the component list.
   Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
      If disposing Then
         If Not (components Is Nothing) Then
            components.Dispose()
         End If
      End If
      MyBase.Dispose(disposing)
   End Sub
      ' Required by the Windows Form Designer
   Private components As System.ComponentModel.Container

   ' NOTE: The following procedure is required by the Windows Form Designer
   ' It can be modified using the Windows Form Designer.
   ' Do not modify it using the code editor.
   Private Sub InitializeComponent()

   Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
      firstName = firstText.Text
      middleName = middleText.Text
      lastName = lastText.Text
      aName = New Name(firstName, middleName, lastName)

   End Sub

   Private Function printList(outFile As StreamWriter, list As SortedList)
   ' Helper method to print a list
       Dim name As Name
       Dim limit As Integer = list.length()
       Dim index As Integer
       list.resetList()
       For index = 1 To limit
           Name = list.getNextItem()
```

```
            outFile.WriteLine(name.firstLastMid())
        Next index
    End Function
End Class
```

Here is a screen shot of the user interface.

Summary

This chapter has provided practice in working with lists where the items are stored in a one-dimensional array. We have examined algorithms that insert, delete, and search data stored in an array-based unsorted list, and we have written methods to implement these algorithms. We have also examined an algorithm that takes the array in which the list items are stored and sorts them into ascending order.

We have examined several search algorithms: sequential search in an unsorted list, sequential search in a sorted list, and binary search. The sequential search in an unsorted list compares each item in the list to the one being searched for. All items must be examined before it can be determined that the search item is not in the list. The sequential search in a sorted list can determine that the search item is not in the list when the place where the item belongs has been passed. The binary search looks for the search item in the middle of the list. If it is not there, then the search continues in the half where the item should be. This process continues to cut the search area in half until either the item is found or the search area is empty.

We have examined the insertion algorithm that keeps the items in the list sorted by value. We generalized the list in an abstract class `List`, leaving the `Insert`, `Delete`, and `isThere` methods abstract. We demonstrated using the `Object` data type as a way to make the list items generic.

Quick Check

1. What is the difference between a list and an array? (p. 492)
2. If the list is unsorted, does it matter where a new item is inserted? (p. 492)
3. The following code fragment implements the “Delete, if it’s there” meaning for the `Delete` operation in an unsorted list. Change it so that the other meaning is

"Implement"; that is, there is an assumption that the item is in the list. (pp. 502–503)

```
While (index < numItems And Not(found))
   Dim count As Integer
   If (listItems(index) = item) Then
      found = True
   Else
      index += 1
   End If
End While
If (found) Then
   For count = index To numItems-2
      listItems(count) = listItems(count+1)
   Next count
   numItems -= 1
End If
```

4. In a sequential search of an unsorted array of 1,000 values, what is the average number of loop iterations required to find a value? What is the maximum number of iterations? (pp. 505–506)
5. The following program fragment sorts list items into ascending order. Change it to sort into descending order. (pp. 507–508)

```
For passCount = 0 To length – 1
   minIndex = passCount
   For searchIndex = passCount + 1 To numItems – 1
      If (listItems(searchIndex.compareTo(listItems(minIndex))_
                  < 0) Then
         minIndex = searchIndex
      End If
   Next searchIndex
   temp = listItems(minIndex)
   listItems(minIndex) = listItems(passCount)
   listItems(passCount) = temp
Next passCount
```

6. Describe how the `Insert` operation can be used to build a sorted list from unsorted input data. (pp. 512–514)
7. Describe the basic principle behind the binary search algorithm. (pp. 519–523)

Answers

1. A list is a variable-sized structured data type; an array is a built-in type often used to implement a list.
2. No.

3.
```
index = 0
While (item <> listItems(index)
   index += 1
End While
Dim count As Integer
For count = index To numItems - 2
   listItems(count) = listItems(count+1)
Next count
numItems -= 1
```

4. The average number is 500 iterations. The maximum is 1,000 iterations. 5. The only required change is to replace the less than with greater than in the inner loop. As a matter of style, the name `minIndex` should be changed to `maxIndex`. 6. The list initially has a length of 0. Each time a value is read, `Insert` adds the value to the list in its correct position. When all the data values have been read, they are in the array in sorted order. 7. The binary search takes advantage of sorted list values, looking at a component in the middle of the list and deciding whether the search value precedes or follows the midpoint. The search is then repeated on the appropriate half, quarter, eighth, and so on, of the list until the value is located.

Exam Preparation Exercises

1. The following values are stored in an array in ascending order.

   28 45 97 103 107 162 196 202 257

   Applying the unsorted linear search algorithm to this array, search for the following values and indicate how many comparisons are required to either find the number or find that it is not in the list.
   a. 28
   b. 32
   c. 196
   d. 194
2. Repeat Exercise 1, applying the algorithm for a sequential search in a sorted list.
3. The following values are stored in an array in ascending order.

   29 57 63 72 79 83 96 104 114 136

   Apply the binary search algorithm looking for 114 in this list and trace the values of `first`, `last`, and `middle`. Indicate any undefined values with a *U*.
4. (True or False?) A binary search is always better to use than a sequential search.
5. If `resetList` initializes `currentPos` to −1 rather than 0, what corresponding change would have to be made in `getNextItem`?
6. We have said that arrays are homogeneous structures, yet Visual Basic.NET implements them with an associated integer. Explain.
7. Why does the outer loop of the sorting method run from 0 through `numItems-2` rather than `numItems-1`?
8. A method that returns the number of days in a month is an example of
   a. a constructor

   b. an observer
   c. an iterator
   d. a transformer
9. A method that adds a constant to the salary of everyone in a list is an example of
   a. a constructor
   b. an observer
   c. an iterator
   d. a transformer
10. A method that stores values into a list is an example of
   a. a constructor
   b. an observer
   c. an iterator
   d. a transformer
11. What kind of class cannot be instantiated?
12. Class `List` assumes that there are no duplicate items in the list.
   a. Which method algorithms would have to be changed to allow duplicates?
   b. Would there still be options for the `Delete` operation? Explain.

Programming Warm-Up Exercises

1. Complete the implementation of `UnsortedList` as a derived class from abstract class `List`.
2. Complete the implementation of `ListWithSort` as a class derived from `UnsortedList`.
3. Derive a subclass of `UnsortedList` that has the following additional methods:
   a. A value-returning instance method named `Occurrences` that receives a single parameter, `item`, and returns the number of times `item` occurs in the list
   b. A Boolean instance method named `greaterFound` that receives a single parameter, `item`, and searches the list for a value greater than `item`. If such a value is found, the method returns `True`; otherwise, it returns `False`.
   c. An instance method named `Component` that returns a component of the list given a position number (`pos`). The position number must be within the range 0 through `numItems-1`.
   d. A copy constructor for the `List` class that takes a parameter that specifies how much to expand the array holding the items. Implement the copy constructor by creating a larger array and copying all of the items in the list into the new array.
4. Complete the implementation of `SortedList` as a derived class from abstract class `List`.
5. Derive a subclass of `SortedList` that has the additional methods outlined in Exercise 3.
6. Write a Visual Basic.NET method named `Exclusive` that has three parameters: `item`, `list1`, and `list2` (both of class `List` as defined in this chapter). The method returns `True` if item is present in either `list1` or `list2` but not both.

7. The `Insert` method in class `SortedList` returns items into the list in ascending order. Derive a new class from `List` that sorts the items in descending order.
8. Exam Preparation Exercise 12 asked you to examine the implication of a list with duplicates.
   a. Design an abstract class `ListWithDuplicates` that allows duplicate keys.
   b. How does your design differ from `List`?
   c. Implement your design where the items are unsorted and `Delete` deletes all of the duplicate items.
   d. Implement your design where the items are sorted and `Delete` deletes all of the duplicate items.
   e. Did you use a binary search in part (d)? If not, why not?
9. Rewrite method `Insert` in class `SortedList` so that it implements the first insertion algorithm discussed for sorted lists. That is, the place where the item should be inserted is found by searching from the beginning of the list. When the place is found, all the items from the insertion point to the end of the list are shifted down one position.

Programming Problems

1. A company wants to know the percentages of total sales and total expenses attributable to each salesperson. Each person has a pair of data lines in the input file. The first line contains his or her name, last name first. The second line contains his or her sales (`Integer`) and expenses (`Single`). Write an application that produces a report with a header line containing the total sales and total expenses. Following this header should be a table with each salesperson's name, percentage of total sales, and percentage of total expenses, sorted by salesperson's name. Use one of the list classes developed in this chapter.
2. Only authorized shareholders are allowed to attend a stockholder's meeting. Write an application to read a person's name from the keyboard, check it against a list of shareholders, and print a message on the screen saying whether or not the person may attend the meeting. The list of shareholders is in a file `owners.dat` in the following format: first name, blank, last name. Use the end-of-file condition to stop reading the file. The maximum number of shareholders is 1,000.
3. Enhance the program in Problem 2 as follows:
   a. Print a report file showing the number of stockholders at the time of the meeting, how many were present at the meeting, and how many people who tried to enter were denied permission to attend.
   b. Follow this summary report with a list of the names of the stockholders, with either *Present* or *Absent* after each name.
4. An advertising company wants to send a letter to all its clients announcing a new fee schedule. The clients' names are on several different lists in the company. The various lists are merged to form one file, "`clients`," but obviously, the company does not want to send a letter twice to anyone.

Write an application that removes any names appearing on the list more than once. On each data line there is a four-digit code number, followed by a blank and then the client's name. For example, Amalgamated Steel is listed as

```
0231 Amalgamated Steel
```

Your program is to output each client's code and name, but no duplicates should be printed. Use one of the list classes developed in this chapter.

Case Study Follow-Up Exercises

1. Write a test plan for application `ExamAttendance`.
2. If the event handler does not delete the name from the class list when a student arrives, what other algorithm could be used to determine the names of those students that did not attend the exam?
3. Redesign the solution to this problem to use `ListWithSort`.
4. Implement the design in Question 3.