SYBEX Sample Chapter

VBA Developer's Handbook

Ken Getz
Mike Gilbert

Chapter 8: Creating Dynamic Data Structures Using Class Modules

ISBN: 0-7821-2978-1

chapter 8

Creating Dynamic Data Structures Using Class Modules

- Using class modules to implement abstract data structures
- Emulating a stack
- Emulating a queue
- Creating and using ordered linked lists
- Creating and using binary trees

Almost any application requires that you maintain some data storage in memory. As your application runs, you read and write data in some sort of data structure, and when your application shuts down, it either discards the data structure (and its data) or writes the data to some persistent storage.

VBA provides two built-in data structures: arrays and collections. Each has its good and bad points, and there are compelling reasons to use each of these structures. (For more information on using arrays and collections, see Chapter 7.) On the other hand, if you've previously programmed in other languages or have studied data structures in a college course, you may find the need to use abstract data structures, such as linked lists, binary trees, stacks, and queues, as part of your applications. Although all these structures can be implemented using arrays or collections, neither of those constructs is well suited for linked data structures.

This chapter introduces techniques for using class modules to construct abstract data structures. Amazingly, VBA requires very little code to create these somewhat complex structures. Once you've worked through the examples in this chapter, you'll be able to exploit the power of linked lists, stacks, queues, and binary trees in your own VBA applications. Table 8.1 lists the sample files you'll find on the accompanying CD-ROM.

TABLE 8.1: Sample Files

Filename	Description
DYNAMIC.XLS	Excel file with sample modules and classes
DYNAMIC.MDB	Access 2000 file with sample modules and classes
DYNAMIC.VBP	VB6 project with sample modules and classes
LISTTEST.BAS	Test routines for List class
QUEUETEST.BAS	Test routines for Queue class
STACKTEST.BAS	Test routines for Stack class
TREETEST.BAS	Test routines for Tree class
LIST.CLS	Linked List class
LISTITEM.CLS	ListItem class
QUEUE.CLS	Queue class
QUEUEITEM.CLS	QueueItem class

TABLE 8.1: Sample Files *(continued)*

Filename	Description
STACK.CLS	Stack class
STACKITEM.CLS	StackItem class
TREE.CLS	Tree class
TREEITEM.CLS	TreeItem class
MAIN.FRM	Start-up form for VB project

Dynamic versus Static Data Structures

VBA provides a simple data structure: the array. If you know how many elements you're going to need to store, arrays may suit you fine. On the other hand, arrays present some difficulties:

They are linear only. You cannot overlay any kinds of relationships between the elements of an array without going through a lot of work.

They're essentially fixed size. Yes, you can ReDim (Preserve) to resize the array, but all VBA does in that case is create a new data structure large enough for the new array and copy all the elements over, one by one. This isn't a reasonable thing to do often, or for large arrays.

They often use too much space. No matter how many elements you're going to put into the array, you must pre-declare the size. It's just like the pre-payment rip-off the car rental companies provide—you pay for a full tank, regardless of whether you actually use it. The same goes for arrays: If you dimension the array to hold 50 elements and you store only 5, you're wasting space for the other 45.

Because of these limitations, arrays are normally referred to as *static* data structures.

On the other hand, a *dynamic* data structure is one that can grow or shrink as needed to contain the data you want stored. That is, you can allocate new storage when it's needed and discard that storage when you're done with it.

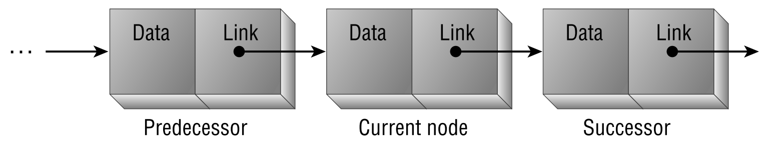
Dynamic data structures generally consist of at least some simple data storage (in our case, it will be a class module), along with a link to the next element in the structure. These links are often called *pointers* or *references.* You'll see both terms used here.

The study of dynamic data structures could be a full-semester college course on its own, so we can't delve too deeply into it in this limited space. However, we do introduce the basic concepts and show how you can use class modules to create your own dynamic data structures. In addition, we suggest some ways in which you might use these data structures in your own applications.

Simple Dynamic Structures

Linear structures are the simplest class of dynamic data structures. Each element of structures of this type contains some information and a pointer to the next element. The diagram in Figure 8.1 shows a simple data structure in which each element of the structure contains a piece of data and a reference to the next item in the structure. (This structure is normally called a *linked list* because it contains a list of items that are linked together.)

FIGURE 8.1
The simplest type of dynamic data structure



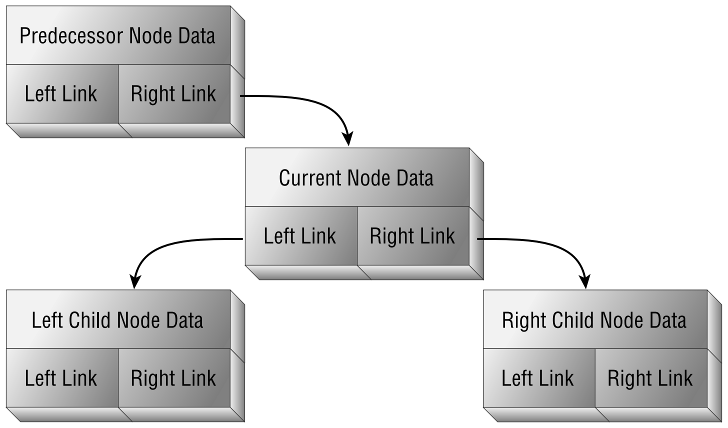
What differentiates one instance of this kind of data structure from another? It's just the arbitrary rules about how you can add or delete nodes. For example, stacks and queues are both types of linear linked data structures, but a stack can accept new items only at its "top," and a queue can accept new items only at its "bottom." With a stack, you can retrieve items only from the same place you added them. But with a queue, you retrieve them from the other end of the structure. This chapter discusses creating both of these simple data structures with VBA class modules.

If you need to be able to traverse your structure in both directions, you can, of course, include links in both directions. Although we won't handle this additional step in this chapter, it takes very little extra work to provide links in both directions. You'll find this extra pointer useful when you must traverse a list in either direction.

Recursive Dynamic Structures

You'll normally use iterative code to loop through the elements of a simple, linear dynamic data structure. On the other hand, many popular dynamic data structures lend themselves to being traversed recursively. For example, programmers often use the *ordered binary tree* structure for data storage and quick retrieval. In this kind of structure, each node has one predecessor and two successors. (Normally, you think of one successor as being the "left child" and the other as the "right child.") Figure 8.2 shows the simplest recursive data structure: a binary tree. The tree data structure is well suited to recursive algorithms for adding items and traversing the nodes.

FIGURE 8.2
Ordered binary trees are an example of a recursive data structure.



NOTE

The term *dynamic data structures* always refers to *in-memory* data structures. All the techniques covered in this chapter deal only with data that you work with in the current instance of your application and have nothing to do with storing or retrieving that data from permanent storage. VBA provides its own techniques for reading and writing disk files. You'll use the data structures presented in this chapter once you've retrieved the data you need to work with.

How Does This Apply to VBA?

Because VBA supports class modules and because you can create a new instance of a class (that is, *instantiate* a new member of the class) at any time, you can create class modules that emulate these abstract data structures. Each element of the structure, because it's just like every other element, is just another instance of the class. (For information on getting started with class modules, see Chapter 5.)

You can most easily represent abstract structures in VBA using two class modules: one to represent a data type that does nothing more than point to the real data structure, and another to represent each element of the structure. For example, if you want to create a stack data structure (and you will later in this section), you'll need one class module to act as a pointer to the "top" of the stack. This is where you can add new items to the stack. You'll also need a different class module for the elements in the stack. This class module will contain two pieces of data: the information to be stored in the stack and a reference to the next item on the stack.

Retrieving a Reference to a New Item

At some point, you'll need to retrieve a reference to a new instance of your class. If you want to add a new item to your data structure, you'll need a pointer to that new item so you can get back to it later. Of course, Basic (after all, as many folks will argue, this *is* still just Basic) has never supported real pointers, and dynamic data structures require pointers, right? Luckily, not quite!

VBA allows you to instantiate a new element of a class and retrieve a reference to it:

Dim *objVar* As New *className*

or

Dim *objVar* as *className*

' Possibly some other code in here.

Set *objVar* = New *className*

You choose one of the two methods for instantiating a new item based on your needs. In either case, you end up with a variable that refers to a new instance of the class.

WARNING Be wary of using the New keyword in the Dim statement. Although this makes your code shorter, it can also cause trouble. This usage allows VBA to instantiate the new object whenever it needs to (normally, the first time you attempt to set or retrieve a property of the object) and, therefore, runs the new object's Initialize event at that time. If you want control over exactly when the new object comes into being (and when its Initialize event procedure runs), use the New keyword with the Set statement. This will instantiate the object when you're ready to, not at some time when you might not be expecting it. In addition, using the New keyword as part of the Dim statement causes VBA to add extra code to your application because it must check at runtime whether it needs to instantiate the object between each line of code where the object is in scope. You don't need this extra overhead.

After either of these statements, *objVar* contains a pointer to the new member of the *className* class. Even though you can't manipulate, view, or otherwise work with pointer values as you can in C/C++, the Set/New combination at least gives VBA programmers almost the same functionality that Pascal programmers have always had, although the mechanism is a bit clumsier: You can create pointers only to classes in VBA, while Pascal allows pointers to almost any data type.

Making an Object Variable Refer to an Existing Item

Just as you can use the Set keyword to retrieve a reference to a new object, you can use it to retrieve a reference to an existing object. If objItem is an object variable that refers to an existing member of a class, you can use code like this to make objNewItem refer to the existing item:

```
Set objNewItem = objItem
```

After this statement, the pointers named `objNewItem` and `objItem` refer to the same object.

What If a Variable Doesn't Refer to Anything?

How can you tell if an object variable doesn't refer to anything? When working with dynamic data structures, you'll find it useful to be able to discern whether a reference has been instantiated. Pascal uses *Nil,* C uses *Null,* and VBA uses *Nothing* to represent the condition in which an object variable doesn't currently refer to a real object.

If you have an object variable and you've not yet assigned it to point to an object, its value is Nothing. You can test for this state using code like this:

```
If objItem Is Nothing Then
    ' You know that objItem isn't currently referring to anything
End If
```

If you want to release the memory used by an object in memory, you must sever all connections to that object. As long as some variable refers to an object, VBA won't be able to release the memory used by that object. (Think of it as a hot-air balloon tied down with a number of ropes; until someone releases the last rope, that balloon isn't going anywhere.) To release the connection, set the object variable to Nothing:

```
Set objItem = Nothing
```

Once you've released all references to an object, VBA can dispose of the object and free up the memory it was using.

Emulating Data Structures with Class Modules

Before you can do any work with dynamic data structures, you need to understand how to use class modules to emulate the elements of these structures. For example, In Figure 8.1, each element of the structure contains a piece of data and a reference to the next element. How can you create a class module that does that?

It's easy: Create a class module named ListItem with two module-level variables:

```
Public Value As Variant
Public NextItem As ListItem
```

The first variable, Value, will contain the data for each element. The second variable, NextItem, will contain a reference to the next item in the data structure. The surprising, and somewhat confusing, issue is that you can create a variable of the same type as the class in the definition of the class itself. It's just this sort of self-referential declaration that makes dynamic data structures possible in VBA.

To add an item to the list, you might write code like this in your class module:

```
Public Function AddItem(varValue As Variant) As ListItem
    Set NextItem = New ListItem
    NextItem.Value = varValue
    ' Set the return value for the function.
    Set AddItem = NextItem
End Sub
```

The first line of the procedure creates a new item in the data structure and makes the NextItem variable in the current element refer to that new element. The second line uses NextItem to refer to the next element and sets its Value variable to the value passed to the current procedure, varValue. The final line sets up the function call to return a reference to the new item that was just added to the list.

In reality, you probably wouldn't write a data structure this way because it provides no way to find a particular item or the beginning or end of the list. In other words, there's something missing that makes these structures possible: a reference to the entire structure. The next section tells you how you should actually create such a data structure.

How about the complicated binary tree structure shown in Figure 8.2? The only difference between this structure and a linear list is that each element in this structure maintains a pointer to two other structures rather than just one. The class module for an element (class name TreeItem) of a binary tree structure might contain these elements:

```
Public Value As Variant
Public LeftChild As TreeItem
Public RightChild As TreeItem
```

Creating a Header Class

Although you can use a class module to emulate the elements of a dynamic data structure, as shown in the previous section, you'll need a different class module to "anchor" the data structure. This class module will generally have only a single instance per data structure and will contain pointers to the beginning, and perhaps the end, of the data structure. In addition, this class often contains the code necessary to add and delete items in the list.

Generally, the header class contains one or more references to objects of the type used in building the data structure, and perhaps other information about the structure itself. For example, a hypothetical class named ListHeader, with the following information, has a reference to the first item in a list and the last item in the list:

```
Private liFirst As ListItem
Private liLast As ListItem
```

Note that the class doesn't contain a self-referential data element. There's generally no reason for a list header to refer to another list header, so this example doesn't contain a reference to anything but the list items. In addition, the header class only needs to contain a reference to the first item in the data structure.

How you work with the items in the data structure—adding, deleting, and manipulating them—depends on the logical properties of the data structure you're creating. Now that you've seen the basics, it's time to dig into some data structures that emulate stacks and queues, each of which has its own ideas about adding and deleting items.

Creating a Stack

A stack is a simple logical data structure, normally implemented using a linked list to contain its data. Of course, you could use an array to implement a stack, and many programmers have done this. However, using an array forces you to worry about the size of the stack, which a linked list structure would not. A stack allows you to control data input and output in a very orderly fashion: New items can be added only to the top of the stack. And, as you remove items, they too are removed from the top. In essence, a stack data structure works like the stack of cafeteria trays at your local eatery or like the pile of problems to solve on your desk (unless you're as compulsive as one of us is—we're not telling which one—and solve your problems in a queue-like fashion). This sort of data storage is often referred to as LIFO (Last In, First Out)—the most recent item added to the stack is the first to be removed.

Why Use a Stack?

Why use a stack in an application? You might want to track forms as a user opens them and then be able to back out of the open forms in the opposite order: That is, you may want to store form references in the stack and then, as the user clicks the OK button on each form, bring the correct form to the top, popping the most recent form from the stack. Or you may want to track the procedure call tree within your application as your user runs it. That way, you could push the name of the procedure as you enter the procedure. On the way out, you could pop the stack. Using this technique, the top of the stack always contains the name of the current procedure. Otherwise, this value is impossible to retrieve. (Perhaps some day VBA will allow you to gather information about the internal call stack programmatically. At this point, you're left handling it yourself.) You could also build your own application profiler. By storing the current time in the stack for each procedure as you push it on the stack and then subtracting that from the current time as you pop the stack, you can find out how long the code was working in each procedure.

Implementing a Stack

Figures 8.3 and 8.4 show a sample stack in memory, before and after a fifth item is added to the stack. At each point, the top of the stack points to the top-most element. After the new element is added, the top of the stack points at the newest element, and that element's link points to the item that used to be at the top of the stack.

It takes very little code to create and maintain a stack. The structure requires two class modules: the Stack and StackItem classes.

FIGURE 8.3
A sample stack just before adding a fifth item

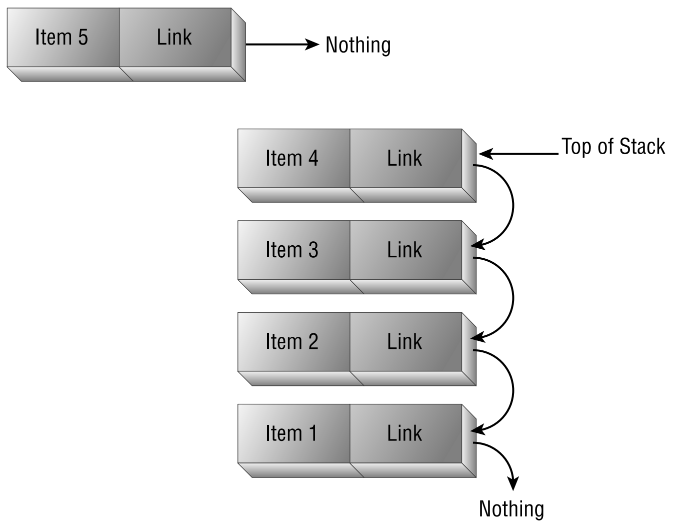
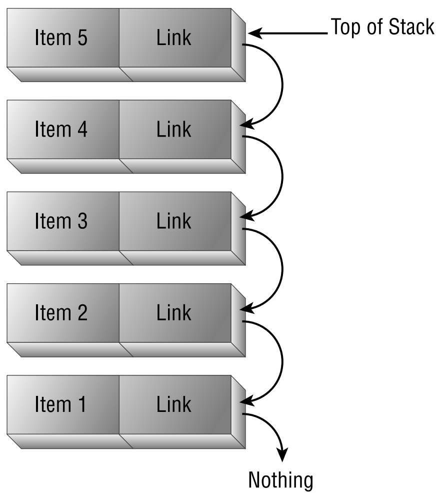


FIGURE 8.4
The same stack after adding the new item



The StackItem Class

It doesn't get much simpler than this. The StackItem class maintains a data item, as well as a pointer to the next item in the structure, as shown in Listing 8.1.

Listing 8.1: Code for the StackItem Class

```
' Keep track of the next stack item,
' and the value of this item.

Public Value As Variant
Public NextItem As StackItem
```

The Stack Class

The Stack class contains a single item: a pointer to the first item in the stack (the stack top). That pointer always points to the top of the stack, and it's at this location that you'll add (push) and delete (pop) items from the stack. The Stack class module implements the two methods (Push and Pop), as well as two read-only properties, StackTop (which returns the value of the element at the top of the stack without popping the item) and StackEmpty (which returns a Boolean value indicating the status of the stack—True if there are no items in the stack and False if there are items).

Pushing Items onto the Stack

To add an item to the stack, you "push" it to the top of the stack. This is similar to pushing a new cafeteria tray to the top of the tray stack. When you push the new tray, each of the other trays moves down one position in the stack. Using linked lists, the code must follow these steps:

1. Create the new node.
2. Place the value to be stored in the new node.
3. Make the new node point to whatever the current stack top pointer refers to.
4. Make the stack top point to this new node.

The code in Listing 8.2 shows the Push method of the Stack class. The four lines of code correspond to the four steps listed previously.

Listing 8.2: Use the Push Method to Add a New Item to the Stack

```
Public Sub Push(ByVal varText As Variant)
    ' Add a new item to the top of the stack.
    Dim siNewTop As StackItem
    Set siNewTop = New StackItem
    siNewTop.Value = varText
    Set siNewTop.NextItem = siTop
    Set siTop = siNewTop
End Sub
```

Figures 8.5 and 8.6 demonstrate the steps involved in pushing an item onto a stack. In the example case, you're attempting to push the value 27 onto a stack that already contains three elements.

NOTE In the figures, to save space, we've collapsed the Dim and As New statements into one line. The examples use separate lines of code for each step, as we've recommended earlier.

FIGURE 8.5 The first three steps in pushing an item onto a stack

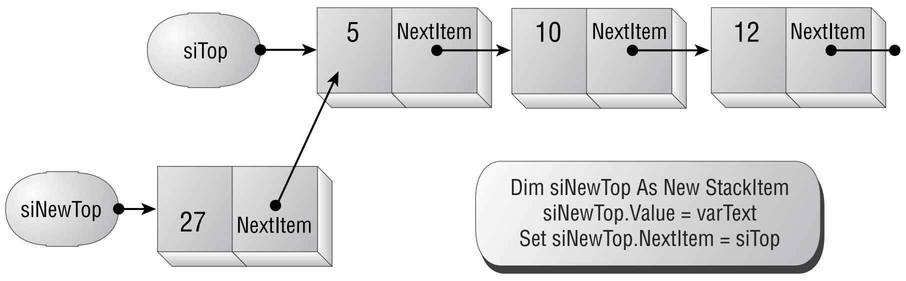
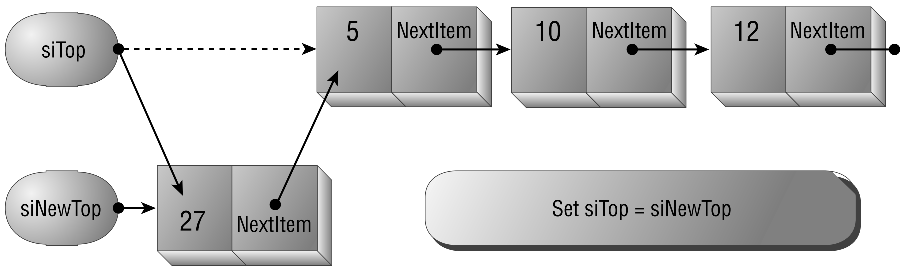


FIGURE 8.6 The final step in pushing an item onto a stack



What if the stack is empty when you try to push an item? In that case, siTop will be Nothing when you execute the following code:

```
Set siNewTop.NextItem = siTop
```

The new node's NextItem property will point to Nothing, as it should. Executing the final line of code:

```
Set siTop = siNewTop
```

causes the top of the stack to point to this new node, which then points to Nothing. It works just as it should!

TIP

If you find this final line of code confusing, look at it this way: When you assign siTop to be siNewTop, you're telling VBA to make siTop contain the same address that siNewTop currently contains. In other words, you're telling siTop to point to whatever siNewTop currently points to. Read that a few times while looking at Figure 8.6, and, hopefully, it will all come into focus.

Popping Items from the Stack

Popping an item from the stack removes it from the stack and makes the top pointer refer to the new item on the top of the stack. In addition, in this implementation, the Pop method returns the value that was just popped.

The code for the Pop method, as shown in Listing 8.3, follows these steps:

1. Makes sure there's something in the stack. (If not, Pop doesn't do anything and returns a null value.)
2. Sets the return value of the function to the value of the top item.
3. Makes the stack top point at whatever the first item is currently pointing to. This effectively removes the first item in the stack.

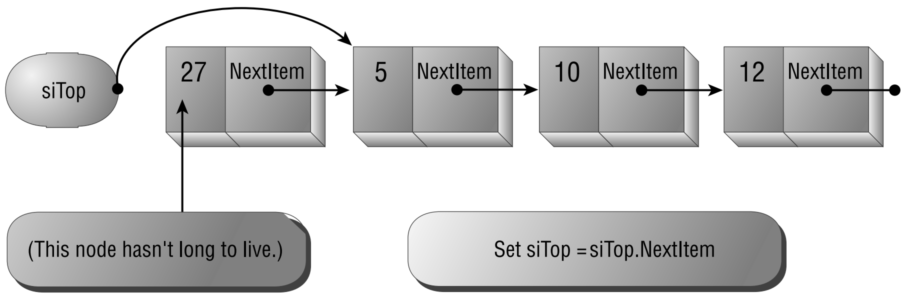
Listing 8.3: Use the Pop Method to Remove an Item from the Stack

```
Public Function Pop() As Variant
    If Not StackEmpty Then
        ' Get the value from the current top stack element.
        ' Then, get a reference to the new stack top.
        Pop = siTop.Value
        Set siTop = siTop.NextItem
    End If
End Function
```

NOTE What happens to the node that used to be at the top of the stack? Once there are no more references to an instance of a class module, VBA can remove that instance from memory, effectively "killing" it. If you're not convinced, add a Debug.Print statement to the Terminate event procedure for the StackItem class. You'll see that VBA kills off unneeded objects as soon as there are no more references to the object.

The diagram in Figure 8.7 demonstrates the tricky step: popping an item from the stack. The code causes the stack pointer, siTop, to refer to the item to which siTop previously referred. That is, it links around the current top item in the stack. Once that occurs, there's no reference to the current top item, and VBA can "kill" the item.

FIGURE 8.7
Link around the top node to pop an item from the stack.



Is the Stack Empty?

You may need to be able to detect whether the stack is currently empty. To make that possible, the example implementation of the Stack data structure provides a read-only StackEmpty property. Providing the information is simple: If siTop is currently Nothing, the stack must be empty.

```
Property Get StackEmpty() As Boolean
    ' Is the stack empty?  It can
    ' only be empty if siTop is Nothing.
    StackEmpty = (siTop Is Nothing)
End Property
```

Given this property, you can write code that pops items until the stack is empty, like this:

```
Do While Not stk.StackEmpty
    Debug.Print stk.Pop()
Loop
```

What's on Top?

You may need to know what's on the top of the stack without removing the item. To make that possible, the example implementation of the Stack data structure includes a read-only StackTop property that returns the value of the item to which siTop points (or Null if siTop is Nothing):

```
Property Get StackTop() As Variant
    If StackEmpty Then
        StackTop = Null
    Else
        StackTop = siTop.Value
    End If
End Property
```

A Simple Example

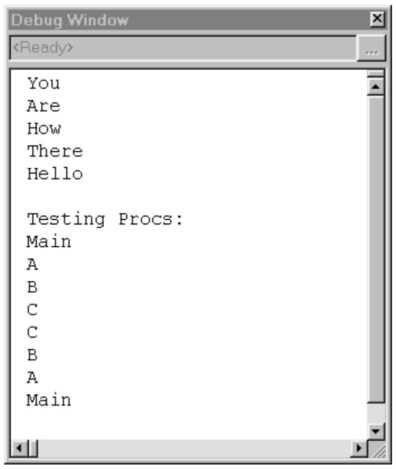
Listing 8.4 shows a few examples using a stack data structure. The first example pushes a number of text strings onto a stack and then pops the stack until it's empty, printing the text to the Immediate window. The second example calls a series of procedures, each of which pushes its name onto the stack on the way in and pops it off on the way out. The screen in Figure 8.8 shows the Immediate window after running the sample.

Listing 8.4: Using the Stack Data Structure

```
Private stkTest As Stack

Sub TestStacks()

    Set stkTest = New Stack

    ' Push some items, and then pop them.
    stkTest.Push ""Hello"
    stkTest.Push "There"
    stkTest.Push "How"
    stkTest.Push "Are"
    stkTest.Push "You"
    Do Until stkTest.StackEmpty
        Debug.Print stkTest.Pop()
    Loop
    ' Now, call a bunch of procedures.
    ' For each procedure, push the proc name
    ' at the beginning, and pop it on the way out.
    Debug.Print
    Debug.Print "Testing Procs:"
    stkTest.Push "Main"
    Debug.Print stkTest.StackTop
    Call A
    Debug.Print stkTest.Pop
End Sub

Sub A()
    stkTest.Push "A"
    Debug.Print stkTest.StackTop
    Call B
    Debug.Print stkTest.Pop
End Sub

Sub B()
    stkTest.Push "B"
    Debug.Print stkTest.StackTop
    Call C
    Debug.Print stkTest.Pop
End Sub

Sub C()
    stkTest.Push "C"
    Debug.Print stkTest.StackTop
    ' You'd probably do something in here...
    Debug.Print stkTest.Pop
End Sub
```

TIP

As you can see from the previous example, it's not hard to create a procedure stack, keeping track of the current procedure from within your code. Unfortunately, you must take care of the details yourself. If you do implement something like this, make sure there's no way to exit a procedure without popping the stack, or your stack will get awfully confused about the identity of the current procedure as you work your way back out, popping things from the stack.



```
You
Are
How
There
Hello

Testing Procs:
Main
A
B
C
C
B
A
Main
```

FIGURE 8.8 The Immediate window after the stack example has run

Creating a Queue

A queue, like a stack, is a data structure based on the linked list concept. Instead of allowing you to add and remove items at a single point, a queue allows you to add items at one end and remove them at the other. In essence, this forms a First In First Out (FIFO) data flow: The first item into the queue is also the first item out. Of course, this is the way your to-do list ought to work—the oldest item ought to get handled first. Unfortunately, most people handle their workflow based on the stack data model, not based on a queue.

Why Use a Queue?

You'll use a queue data structure in an application when you need to maintain a list of items ordered not by their actual value but by their temporal value. For example, you might want to allow users to select a list of reports throughout the day and, at idle times throughout the day, print those reports. Although there are many ways to store this information internally, a queue makes an ideal mechanism. When you need to find the name of the next report to print, just pull it from the top of the queue. When you add a new report to be printed, it goes to the end of the queue.

You can also think of a queue as a pipeline—a means of transport for information from one place to another. You could create a global variable in your application to refer to the queue and have various parts of the application send messages to each other using the queue mechanism, much as Windows itself does with the various running applications.

TIP

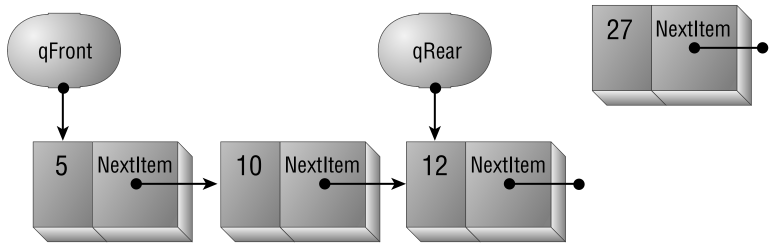
If you're planning on creating an industrial-strength queue in an application to pass information from one user to another, you'll want to study the concepts presented here, but also look into using MSMQ, a server-based product from Microsoft that manages enterprise-wide queuing for you. In one sense, MSMQ works the same way as the queues shown here do. However, in a real sense, comparing MSMQ to the queues shown here is just as accurate as comparing a desktop computer to an abacus. They both perform calculations, but one is far more powerful than the other. If you need disconnected queuing and guaranteed delivery of information in an enterprise-wide environment, you'll want to look into MSMQ.

Implementing a Queue

The diagrams in Figures 8.9, 8.10, and 8.11 show a simple queue before and after adding a new item and before and after removing an item. At each point, you can add a new item only at the rear of the queue and can remove an item only from the front of the queue. (Note that the front of the queue, where you delete items, is at the left of the diagrams. The rear of the queue, where you add items, appears to the right.)

Maintaining a queue takes a bit more code than maintaining a stack, but not much. Although the queue is handled internally as a linked list, it has some limitations as to where you can add and delete items. The underlying code handles these restrictions. The queue structure requires two class modules, one each for the Queue and QueueItem classes.

FIGURE 8.9
A simple queue just before a fourth item is added



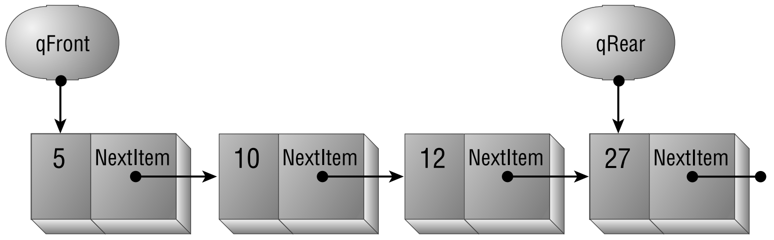


FIGURE 8.10
The simple queue after the fourth item is added and before an item is removed

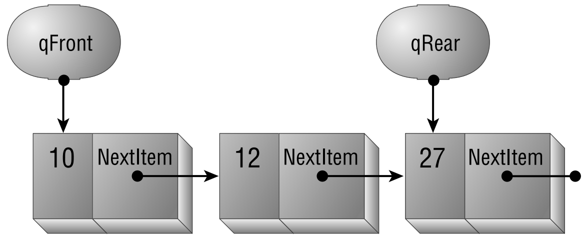


FIGURE 8.11
The simple queue after an item has been removed

The QueueItem Class

Just like the StackItem class, the QueueItem class stores just a data value and a pointer to the next data element, as shown in Listing 8.5.

Listing 8.5: Code for the QueueItem Class

```
' Keep track of the next queue item,
' and the text of this item.
Public NextItem As QueueItem
Public Value As Variant
```

The Queue Class

As with the Stack class, all the interesting code required in working with the data structure is part of the parent class—in this case, the Queue class. It's here you'll find the methods for adding and removing items in the queue, as well as a read-only property that indicates whether the queue is currently empty. Because a queue needs to be able to work with both the front and the rear of the queue, the Queue class includes two pointers rather than just one, making it possible to add items at one end and to remove them from the other. These pointers are defined as qFront and qRear, as shown here, and are module-level variables:

```
Private qFront As QueueItem
Private qRear As QueueItem
```

Adding an Item to the Queue

To add an item to a queue, you "enqueue" it. That is, you add it to the rear of the queue. To do this, the Add method follows these steps:

1. Creates the new node.
2. Places the value to be stored in the new node.
3. If the queue is currently empty, makes the front and rear pointers refer to the new node.
4. Otherwise, links the new node into the list of nodes in the queue. To do that, it makes the final node (the node the "rear pointer" currently points to) point to the new item. Then it makes the rear pointer in the queue header object refer to the new node.

The code in Listing 8.6 shows the Add method of the Queue class.

Listing 8.6: Use the Add Method to Add a New Item to a Queue

```
Public Sub Add(varNewItem As Variant)
    Dim qNew As QueueItem
    Set qNew = New QueueItem

    qNew.Value = varNewItem
    ' What if the queue is empty? Better point
    ' both the front and rear pointers at the
    ' new item.
    If IsEmpty Then
        Set qFront = qNew
        Set qRear = qNew
    Else
        Set qRear.NextItem = qNew
        Set qRear = qNew
    End If
End Sub
```

The diagrams in Figures 8.12 and 8.13 demonstrate the steps for adding a new node to an existing queue.

NOTE As we did earlier, we've collapsed the Dim and New statements in the figures into a single line of code in order to save space. We don't recommend doing this in your own code.

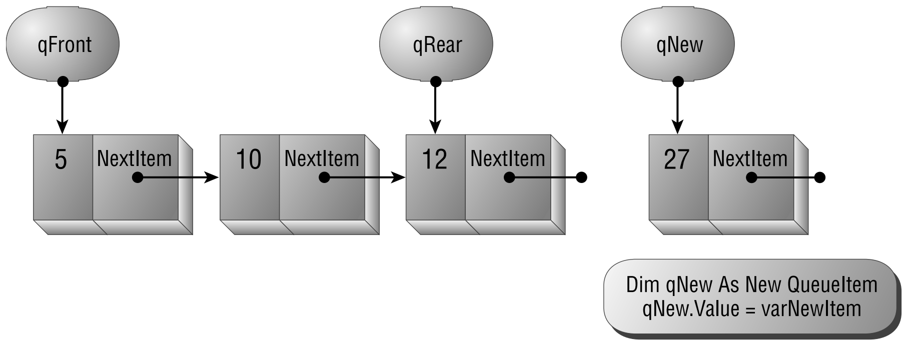


FIGURE 8.12 After you create the new node, the Add method is ready to attach it to the queue.

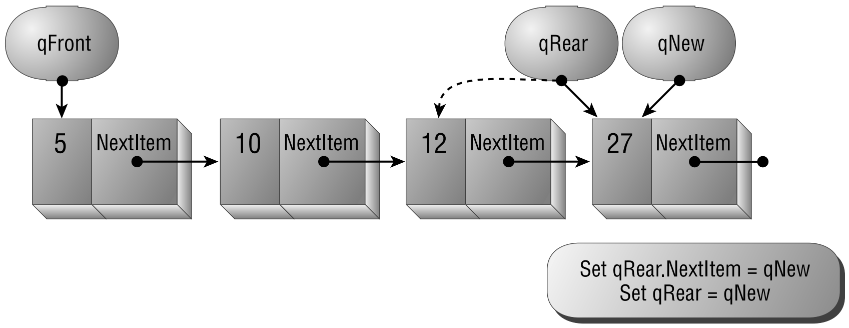
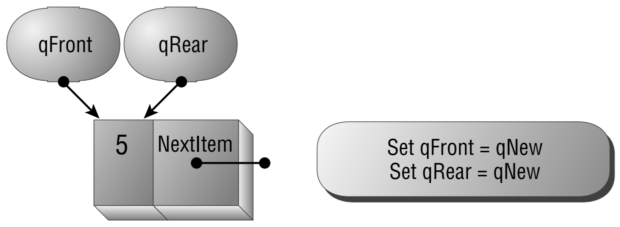


FIGURE 8.13 To finish adding the node, set qRear to point to the new node.

What if the queue was empty when you tried to add an item? In that case, all you need to do is make the head and rear of the queue point to the new node. Afterward, the queue will look like the one in Figure 8.14.

FIGURE 8.14
After a new node is added to an empty queue, both the head and rear pointers refer to the same node.



Removing Items from the Queue

Removing an item from the queue both removes the front node from the data structure and makes the next front-most item the new front of the queue. In addition, this implementation of the queue data structure returns the value of the removed item as the return value from the Remove method.

The code for the Remove method, as shown Listing 8.7, follows these steps:

1. Makes sure there's something in the queue. If not, the Remove method doesn't do anything and returns a null value.
2. Sets the return value of the function to the value of the front queue item.
3. If there's only one item in the queue, sets both the head and rear pointers to Nothing. There's nothing left in the queue.
4. If there was more than one item in the queue, sets the front pointer to refer to the second item in the queue. This effectively kills the old first item.

Listing 8.7: Use the Remove Method to Drop Items from a Queue

```
Public Function Remove() As Variant
    ' Remove an item from the head of the
    ' list, and return its value.
    If IsEmpty Then
        Remove = Null
    Else
        Remove = qFront.Value
        ' If there's only one item
        ' in the queue, qFront and qRear
        ' will be pointing to the same node.
        ' Use the Is operator to test for that.
```

```
            If qFront Is qRear Then
                Set qFront = Nothing
                Set qRear = Nothing
            Else
                Set qFront = qFront.NextItem
            End If
        End If
    End Function
```

TIP

How can you tell when there's only one item in the queue? The Is operator comes in handy here. By checking whether "qFront Is qRear", you can find out whether the two variables refer to the same object. If the condition is True, they do refer to the same object, and, therefore, there's only one item in the queue.

The diagram in Figure 8.15 demonstrates the one difficult step in removing an item. The diagram corresponds to this line of code:

```
Set qFront = qFront.NextItem
```

By moving the front pointer to the item that the first item previously pointed to, you eliminate the reference to the old first item, and VBA removes it from memory. After this step, the queue will contain one less item.

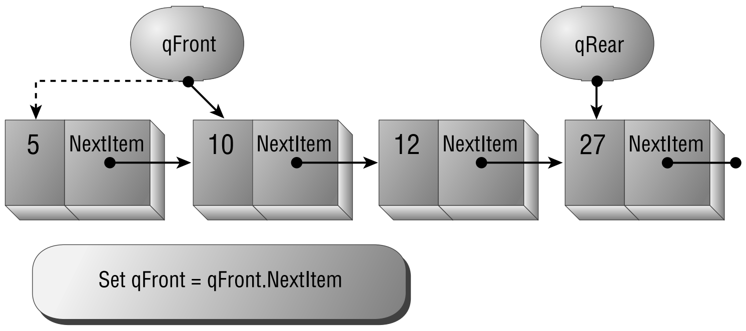


FIGURE 8.15 To remove an item, move the front pointer to the second node in the queue.

Is the Queue Empty?

You'll often need to be able to detect whether the queue is empty, and the example implementation includes the read-only IsEmpty property for this reason. The queue can be empty only if both the front and rear pointers are Nothing. The code shown here checks for this condition:

```
Public Property Get IsEmpty() As Boolean
    ' Return True if the queue contains
    ' no items.
    IsEmpty = ((qFront Is Nothing) And (qRear Is Nothing))
End Property
```

The IsEmpty property allows you to write code like this:

```
Do Until q.IsEmpty
    Debug.Print q.Remove()
Loop
```

A Simple Queue Example

The code in Listing 8.8 demonstrates the use of the queue data structure. It creates a new queue, adds five words to the queue, and then removes the words, one at a time. The words should come out in the same order in which they were entered. Note that if you'd used a stack for the same exercise, the words would have come out in the opposite order from the order in which they were entered.

Listing 8.8: Using the Queue Data Structure

```
Sub TestQueues()
    Dim qTest As Queue

    Set qTest = New Queue
    With qTest
        .Add "Hello"
        .Add "There"
        .Add "How"
        .Add "Are"
        .Add "You"
        Do Until .IsEmpty
            Debug.Print .Remove()
        Loop
    End With
End Sub
```

Creating Ordered Linked Lists

A linked list is a simple data structure, as shown earlier in Figure 8.1, that allows you to maintain an ordered list of items without having to know ahead of time how many items you'll be adding. To build this data structure, you need two class modules: one for the list head and another for the items in the list. The example presented here is a sorted linked list. As you enter items into the list, the code finds the correct place to insert them and adjusts the links around the new nodes accordingly.

The ListItem Class

The code for the ListItem class, shown here, is simple, as you can see in Listing 8.9. The code should look familiar—it's parallel to the code in Listing 8.1. (Remember, the Stack data structure is just a logical extension of the simple linked list.)

```
Public Value As Variant
Public NextItem As ListItem
```

The class module contains storage for the value to be stored in the node, plus a pointer to the next node. As you instantiate members of this class, you'll set the NextItem property to refer to the next item in the list, which depends on where in the list you insert the new node.

The List Class

The List class includes but a single data element:

```
Dim liHead As ListItem
```

The liHead item provides a reference to the first item in the linked list. (If there's nothing yet in the list, liHead is Nothing.) The List class also includes three Public methods: Add, Delete, and DebugList. The Add method adds a new node to the list, in sorted order. The Delete method deletes a given value from the list if it's currently in the list. The DebugList method walks the list from one end to the other, printing the items in the list to the Immediate window.

Finding an Item in the List

Both the Add and Delete methods count on a Private method, Search, which takes three parameters:

- The value to find (passed by value)
- The current list item (passed by reference)
- The previous list item (passed by reference)

The Search procedure fills in the current and previous list items (so the calling procedure can work with both items). Both parameters are passed using ByRef, so the procedure can modify their values. The function returns a Boolean value indicating whether it actually found the requested value. The function, shown in Listing 8.9, follows these steps:

1. Assumes the return value is False, sets liPrevious to point to Nothing, and sets liCurrent to point to the head of the list:

   ```
   blnFound = False

   Set liPrevious = Nothing
   Set liCurrent = liHead
   ```

2. While not at the end of the list (while the current pointer isn't Nothing), does one of the following:

   - If the search item is greater than the stored value, it sets the previous pointer to refer to the current node and sets the current node to point to the next node.
   - If the search item is less than or equal to the stored value, then you're done, and it exits the loop.

   ```
   Do Until liCurrent Is Nothing
       With liCurrent
           If varItem > .Value Then
               Set liPrevious = liCurrent
               Set liCurrent = .NextItem
           Else
               Exit Do
           End If
       End With
   Loop
   ```

3. Establishes whether the sought value was actually found.

```
If Not liCurrent Is Nothing Then
    blnFound = (liCurrent.Value = varItem)
End If
```

4. Returns the previous and current pointers in ByRef parameters and the found status as the return value.

Listing 8.9: Use the Search Function to Find a Specific Element in the List

```
Function Search(ByVal varItem As Variant, _
 ByRef liCurrent As ListItem, ByRef liPrevious As ListItem) _
 As Boolean
    Dim blnFound As Boolean

    blnFound = False

    Set liPrevious = Nothing
    Set liCurrent = liHead
    Do Until liCurrent Is Nothing
        With liCurrent
            If varItem > .Value Then
                Set liPrevious = liCurrent
                Set liCurrent = .NextItem
            Else
                Exit Do
            End If
        End With
    Loop

    ' You can't compare the value in liCurrent to the sought
    ' value unless liCurrent points to something.
    If Not liCurrent Is Nothing Then
        blnFound = (liCurrent.Value = varItem)
    End If
    Search = blnFound
End Function
```

Taking the most common case (searching for an item in the middle of an existing list), the diagrams in Figures 8.16, 8.17, 8.18, and 8.19 demonstrate the steps in the logic of the Search method. In this example, the imaginary code running is searching for the value 7 in a list that contains the values 3, 5, and 10.

FIGURE 8.16
Check to see if it's time to stop looping, based on the current value and the value to find.

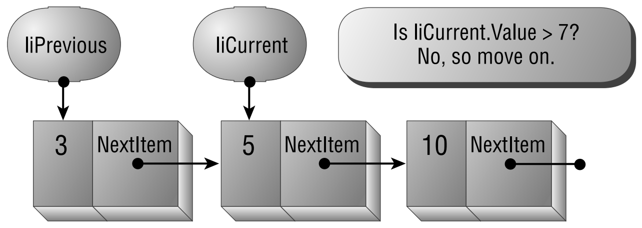


FIGURE 8.17
Set the previous pointer to point to the current node.

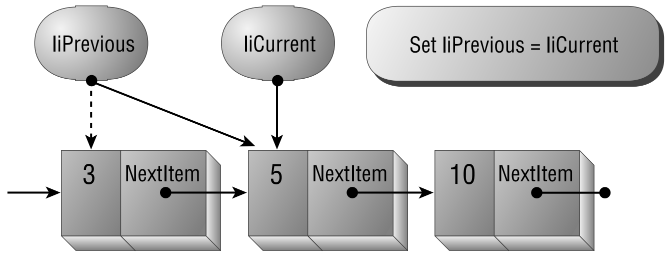


FIGURE 8.18
Set the current pointer to point to the next node.

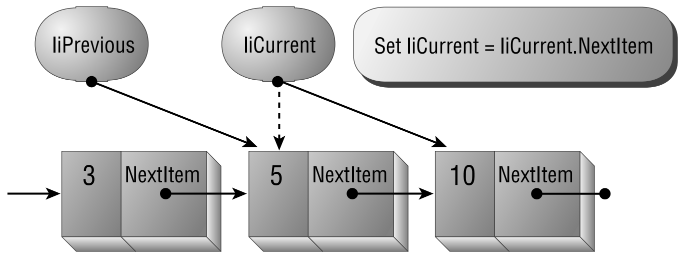
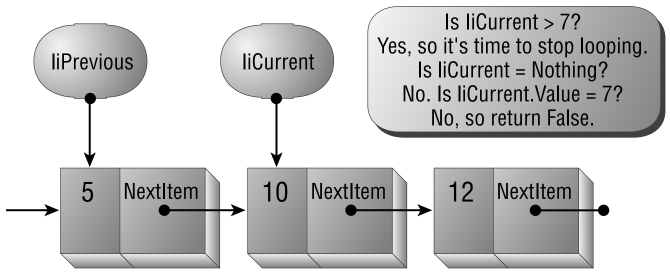


FIGURE 8.19
It's time to stop looping. The item wasn't found, so return False.



What happens in the borderline cases?

What if the list is currently empty? In that case, liCurrent will be Nothing at the beginning of the procedure (because you've made it point to the same thing that liHead points to, which is Nothing). The function will do nothing and will return False. After you call the function, liCurrent and liPrevious will both be Nothing.

What if the item to be found is less than anything currently in the list? In that case, the item should be placed before the item liHead currently points to. As soon as the code enters the loop, it will find that liCurrent.Value is greater than varItem and will jump out of the loop. The function will return False because the value pointed to by liCurrent isn't the same as the value being sought. After the function call, liCurrent will refer to the first item in the list, and liPrevious will be Nothing.

What if the item is greater than anything in the list? In that case, the code will loop until liCurrent points to what the final node in the list points to (Nothing), and liPrevious will point to the final node in the list. The function will return False because liCurrent is Nothing.

Adding an Item to the List

Once you've found the right position using the Search method of the List class, inserting an item is relatively simple. The Add method, shown in Listing 8.10, takes the new value as a parameter, calls the Search method to find the right position in which to insert the new value, and then inserts it. The procedure follows these steps:

1. Creates a new node for the new item and sets its value to the value passed as a parameter to the procedure:

   ```
   Set liNew = New ListItem
   liNew.Value = varValue
   ```

2. Calls the Search method, which fills in the values of liCurrent and liPrevious. Disregard the return value when adding an item, as you don't care whether the value was already in the list:

   ```
   Call Search(varValue, liCurrent, liPrevious)
   ```

3. If inserting an item anywhere but at the head of the list, adjusts pointers to link in the new item:

```
Set liNew.NextItem = liPrevious.NextItem
Set liPrevious.NextItem = liNew
```

4. If inserting an item at the beginning of the list, sets the head pointer to refer to the new node.

```
Set liNew.NextItem = liHead
Set liHead = liNew
```

Listing 8.10: Use the Add Method to Add a New Item to a List

```
Public Sub Add(varValue As Variant)
    Dim liNew As New ListItem
    Dim liCurrent As ListItem
    Dim liPrevious As ListItem

    Set liNew = New ListItem
    liNew.Value = varValue

    ' Find where to put the new item. This function call
    ' fills in liCurrent and liPrevious.
    Call Search(varValue, liCurrent, liPrevious)

    If Not liPrevious Is Nothing Then
        Set liNew.NextItem = liPrevious.NextItem
        Set liPrevious.NextItem = liNew
    Else
        ' Inserting at the head of the list:
        ' Set the new item to point to what liHead currently
        ' points to (which might just be Nothing). Then
        ' make liHead point to the new item.
        Set liNew.NextItem = liHead
        Set liHead = liNew
    End If
End Sub
```

Inserting an item at the head of the list is easy. All you need to do is make the new node's NextItem pointer refer to the current head of the list and then make the list head pointer refer to the new node. The diagrams in Figures 8.20, 8.21, and 8.22 show how you can insert an item at the head of the list. In this example, you're attempting to insert a node with the value 3 into a list containing 5, 10, and 12. Because 3 is less than any item in the list, the code will insert it at the head of the list.

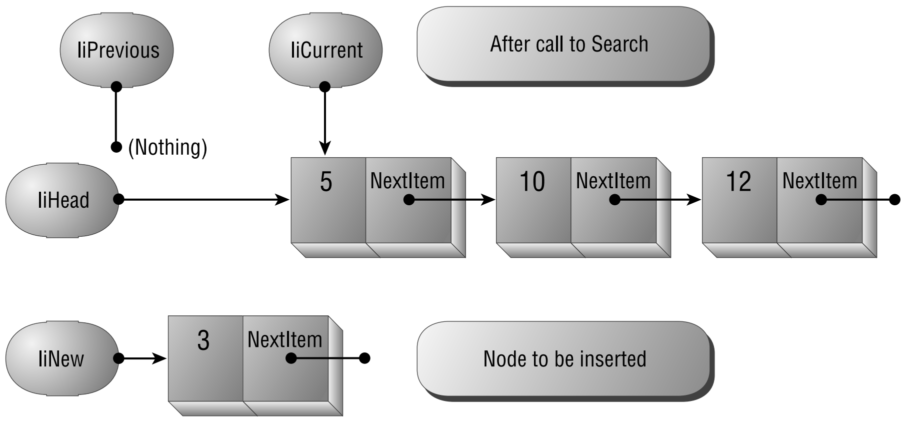


FIGURE 8.20 After Search is called, liPrevious is Nothing, indicating an insertion at the head of the list.

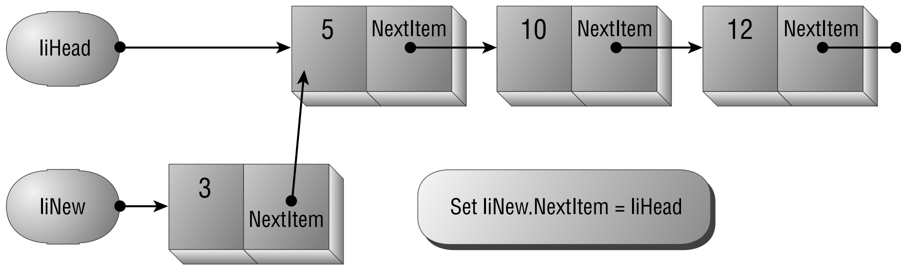


FIGURE 8.21 Make the new node's NextItem pointer refer to the item currently referred to by liHead.

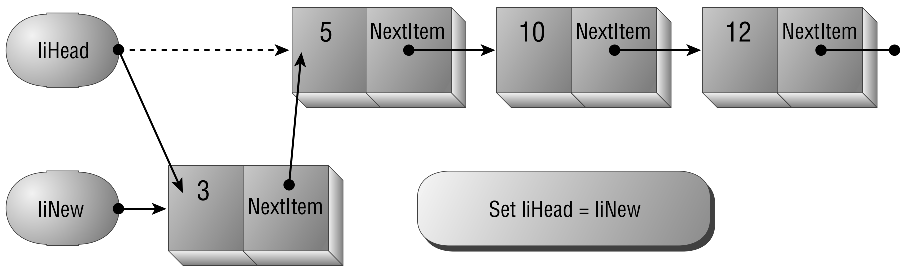


FIGURE 8.22 Make the list header point to the new node.

Inserting an item anywhere in the list besides the head works similarly, but the steps are a bit different. If liPrevious isn't Nothing after the Add method calls Search, you must make the new node's NextItem point to what liPrevious currently points at and then make whatever liPrevious is pointing at point at liNew instead. The diagrams in Figures 8.23, 8.24, and 8.25 illustrate an insertion in the middle (or at the end) of the list. In this series of figures, you're attempting to add an item with value 7 to a list containing 5, 10, and 12.

FIGURE 8.23
After the Add method calls Search, liPrevious isn't Nothing, indicating an insertion after the head of the list.

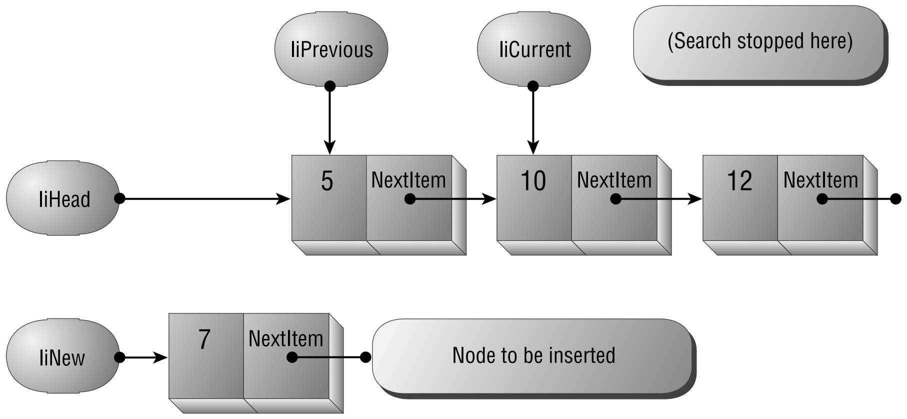
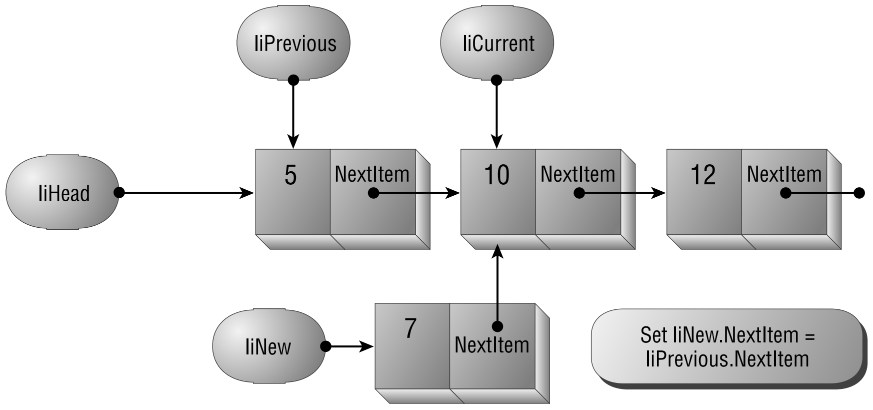


FIGURE 8.24
Make the new item point to the item after the one liPrevious points to.



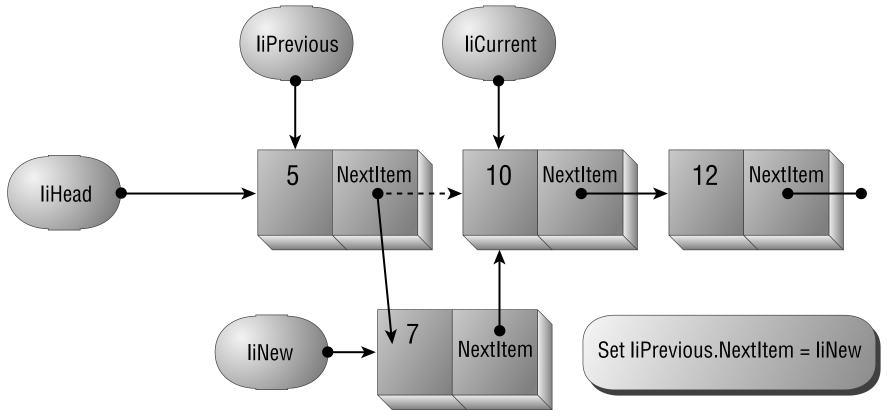


FIGURE 8.25 Make the item that liPrevious points to point to the new item, linking it into the list.

Deleting an Item from the List

Again, just as with adding an item, once you've found the right position using the Search method of the List class, deleting an item doesn't take much code. The Delete method, shown in Listing 8.11, takes the new value as a parameter; calls the Search method to find the item to be deleted; and, if it's there, deletes it. The procedure follows these steps:

1. Calls the Search method, which fills in the values of liCurrent and liPrevious. If the function returns False, there's nothing else to do.

   ```
   blnFound = Search(varItem, liCurrent, liPrevious)
   ```

2. If deleting at the beginning of the list, sets the head pointer to refer to the node pointed to by the selected node. (It links the head pointer to the current second node in the list.)

   ```
   Set liHead = liHead.NextItem
   ```

3. If deleting anywhere but at the head of the list, sets the previous item's pointer to refer to the node pointed to by the item to be deleted. (That is, it links around the deleted node.)

   ```
   Set liPrevious.NextItem = liCurrent.NextItem
   ```

4. When liCurrent goes out of scope, VBA destroys the node to be deleted because no other pointer refers to that instance of the class.

Listing 8.11: Use the Delete Method to Delete an Item from a List

```
Public Function Delete(varItem As Variant) As Boolean
    Dim liCurrent As ListItem
    Dim liPrevious As ListItem
    Dim blnFound As Boolean

    ' Find the item. This function call
    ' fills in liCurrent and liPrevious.
    blnFound = Search(varItem, liCurrent, liPrevious)
    If blnFound Then
        If liPrevious Is Nothing Then
            ' Deleting from the head of the list.
            Set liHead = liHead.NextItem
        Else
            ' Deleting from the middle or end of the list.
            Set liPrevious.NextItem = liCurrent.NextItem
        End If
    End If
    Delete = blnFound
End Function
```

To delete an item from the head of the list, all you need to do is make the header's pointer refer to the second item in the list. The diagrams in Figures 8.26, 8.27, and 8.28 show how you can delete an item at the head of the list.

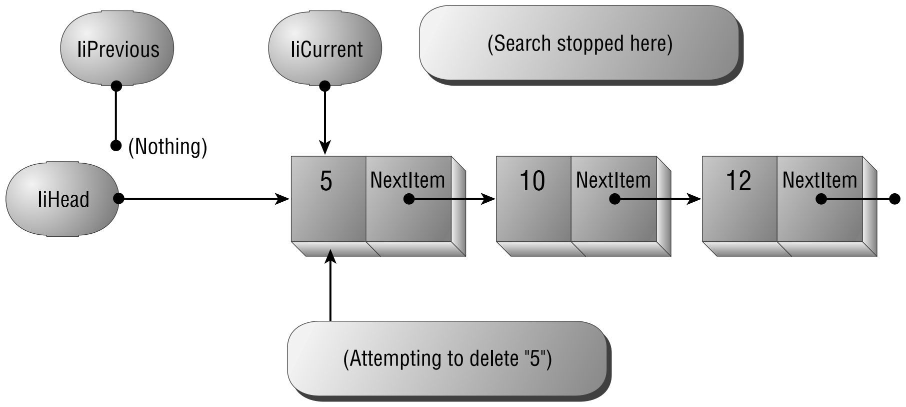


FIGURE 8.26 If the search ends at the head of the list, liPrevious will be Nothing.

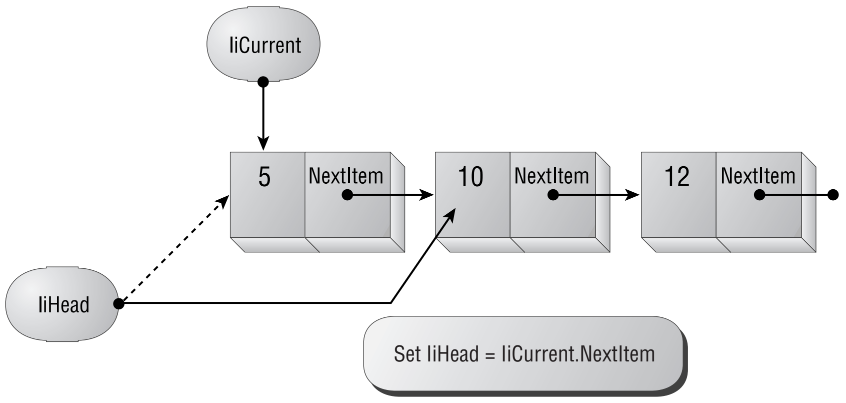


FIGURE 8.27 To delete the first item, make liHead point to the second item in the list.

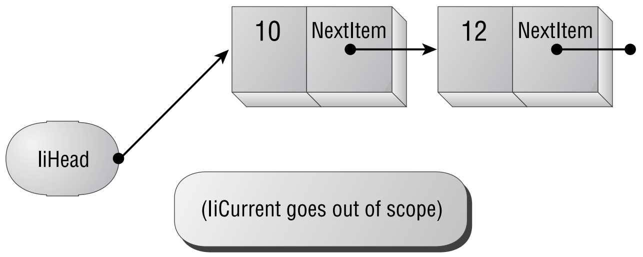


FIGURE 8.28 When liCurrent goes out of scope, VBA destroys the deleted item.

What about deleting an item other than the first? That's easy too: Just link around the item to be deleted. The diagrams in Figures 8.29, 8.30, and 8.31 show how you can delete an item that's not the first item in the list. In this case, you're attempting to delete the node with value 10 from a list that contains 5, 10, and 12.

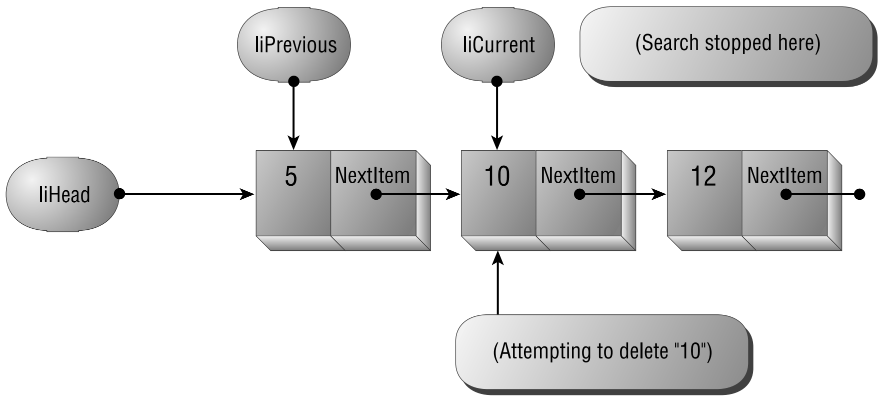


FIGURE 8.29 The search found the node to be deleted. (liCurrent points to it.)

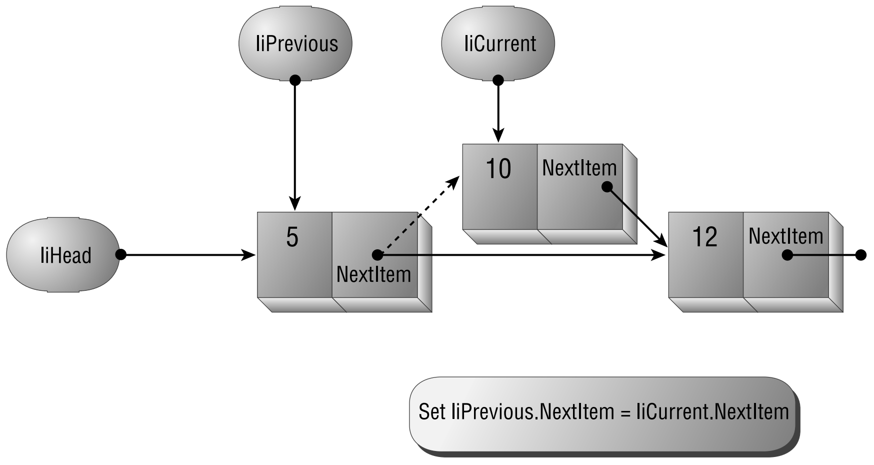


FIGURE 8.30 Link around the node to be deleted.

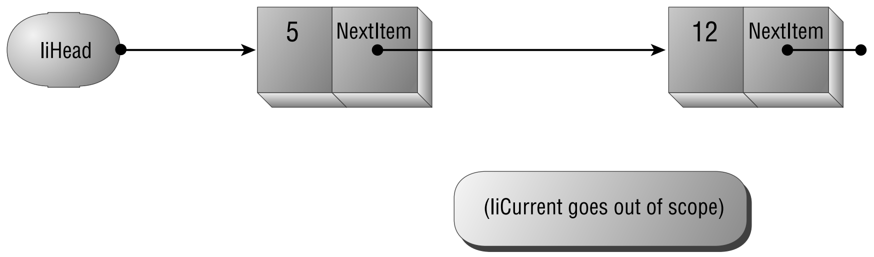


FIGURE 8.31 When liCurrent goes out of scope, VBA destroys the deleted item.

Traversing the List

A list wouldn't do you much good if you couldn't traverse it, visiting each element in turn. The example project includes a DebugList method of the List class. Calling this method walks the list one item at a time, printing each value in turn to the Immediate window:

```
Public Sub DebugList()
    ' Print the list to the Immediate window.
    Dim liCurrent As ListItem
    Set liCurrent = liHead
    Do Until liCurrent Is Nothing
        Debug.Print liCurrent.Value
        Set liCurrent = liCurrent.NextItem
    Loop
End Sub
```

To do its work, the code in DebugList first sets a pointer to the head of the list. Then, as long as that pointer isn't Nothing, the code prints out the current value and sets the current node pointer to refer to the next item in the list.

Testing It Out

The ListTest module includes a simple test procedure that exercises the methods in the List class. When you run this procedure, shown in Listing 8.12, the code will add the 10 items to the list, display the list, delete a few items (including the first and last item), and then print the list again.

Listing 8.12: Sample Code Demonstrating the Ordered Linked List

```
Sub TestLists()
    Dim liTest As List
    Set liTest = New List
    With liTest
        .Add 5
        .Add 1
        .Add 6
        .Add 4
        .Add 9
        .Add 8
        .Add 7
        .Add 10
        .Add 2
        .Add 3
        Call .DebugList
        Debug.Print "====="
        .Delete 1
        .Delete 10
        .Delete 3
        .Delete 4
        Call .DebugList
    End With
End Sub
```

Why Use a Linked List?

That's a good question, because the native VBA Collection object provides much of the same functionality as a linked list, without the effort. Internally, collections are stored as a complex linked list, with links in both directions (instead of only one). The data structure also includes pointers that make it possible to traverse the collection as though it were a binary tree. This way, VBA can traverse the collection forward and backward, and it can find items quickly. (Binary trees provide very quick random access to elements in the data structure.)

It's just this flexibility that makes the overhead involved in using VBA's collections onerous. You may find that you need to create a sorted list, but working with collections is just too slow, and maintaining collections in a sorted order is quite difficult. In these cases, you may find it more worthwhile to use a linked list, as demonstrated in the preceding example, instead.

Creating Binary Trees

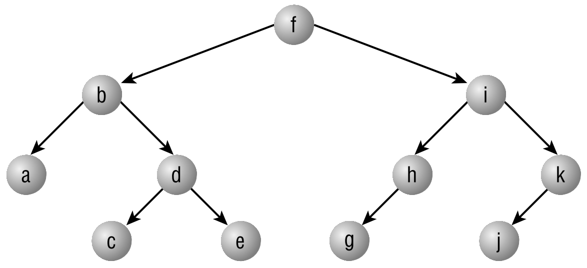
A simple binary tree, as shown earlier in Figure 8.2, is the most complex data structure discussed in this chapter. This type of binary tree is made up of nodes that contain a piece of information and pointers to left and right child nodes. In many cases, you'll use binary trees to store data in a sorted manner: As you add a value, you'll look at each existing node. If the new value is smaller than the existing value, look in the left child tree; if it's greater, look in the right child tree. Because the process at this point is the same no matter which node you're currently at, many programmers use recursive algorithms to work with binary trees.

Why use a binary tree? Besides the fact that finding items in a binary tree is faster than performing a linear search through a list or an array, if you insert the items in an ordered fashion, you not only get efficient storage, but you also get sorting for free—it's like finding a prize in the bottom of your cereal box! Who could ask for more?

Traversing Binary Trees

Once you've created a binary tree, you can use one of three standard methods for traversing the tree. All three of the following examples use the tree illustrated in Figure 8.32. In that figure, the nodes contain letters, but their ordering here doesn't mean anything. They're just labeled to make it easy to refer to them.

FIGURE 8.32
Use this binary tree to demonstrate tree traversal.



Inorder Traversal

To traverse a tree using inorder traversal, you visit each node; but, as you visit each node, you must first visit the left subtree, then the root node, and then the right subtree, in that order. When visiting the subtrees, you take the same steps. If you listed the value each time you visited a root node in the tree shown in Figure 8.32, you'd list the nodes in the following order:

```
a b c d e f g h i j k
```

Preorder Traversal

Using preorder traversal, you first visit the root node, then the left subtree, and then the right subtree. Using this method, you'll always print out the root value and then the values of the left and right children. Using the example shown in Figure 8.32, you'd print the nodes in this order:

```
f b a d c e i h g k j
```

Postorder Traversal

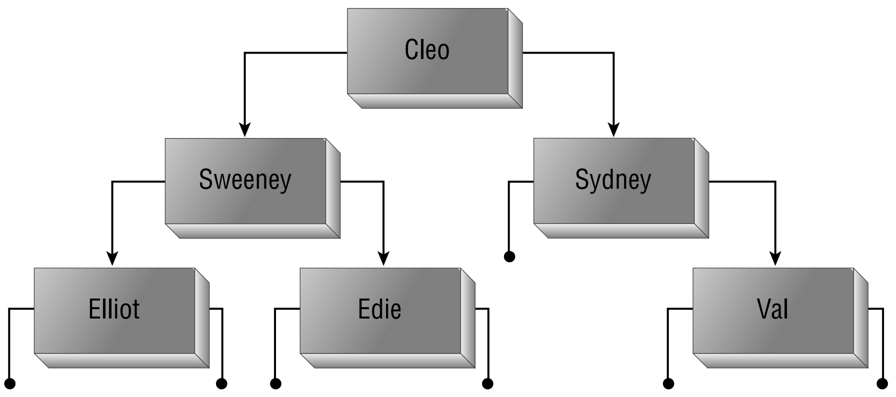
Using postorder traversal, you visit the left subtree; then the right subtree; and, finally, the root node. Using the example shown in Figure 8.32, you'd visit the nodes in this order:

```
a c e d b g h j k i f
```

What's This Good For?

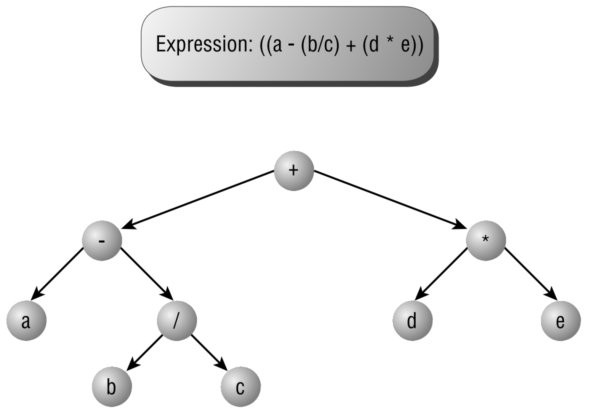
Binary trees have many analogs in the real world. For example, a binary tree can represent a pedigree tree for a purebred cat. Each node represents a cat, with the left and right links to the cat's two parents. If a parent is unknown, the link will point to Nothing. The diagram in Figure 8.33 shows a parentage tree for a hypothetical purebred cat.

FIGURE 8.33
A binary tree can represent parentage (two parents per node)



A binary tree can also represent an algebraic expression. If you place algebraic identifiers (constants and variables) in terminal nodes and operators in the interior nodes, you can represent any algebraic expression in a tree. This makes it possible to write expression evaluators: By parsing the expression, placing the various expressions correctly in the tree, and then traversing the tree in the correct order, you can write a simple expression evaluator. The diagram in Figure 8.34 shows how you might represent a simple algebraic expression in a binary tree.

FIGURE 8.34
A binary tree can represent an algebraic expression.



Depending on how you traverse the tree, you could visit the nodes in any of the following manners:

- Inorder traversal:

  ```
  (a - (b/c) + (d * e))
  ```

- Preorder traversal (the order that might be used by a functional calculator):

  ```
  Add(Subtract(a, Divide(b, c)), Multiply(d, e))
  ```

- Postorder traversal (the order used by “reverse Polish” notation calculators that use a stack for their calculations):

  ```
  Push a
  Push b
  Push c
  Divide
  Subtract
  Push d
  Push e
  Multiply
  Add
  ```

Implementing a Binary Tree

The following sections discuss in some detail how the code that implements the binary Tree class operates. You’ll find the code for this section in Tree.cls, TreeItem.cls, and TreeTest.bas.

The TreeItem Class

As with the structure items in the previous sections, the TreeItem class is simple. It includes just the three necessary data items: the value to be stored at the current node, the pointer to the left child node, and the pointer to the right child node, as shown here:

```
Public Value As Variant
Public LeftChild As TreeItem
Public RightChild As TreeItem
```

Of course, there’s nothing stopping you from storing more information in the TreeItem class. For example, you may need to write a program that can parse a text file, create a binary tree containing all the distinct words in the file, and store each word in its own node, along with a list of all the page numbers on which that word occurred. In this case, you might want to store a pointer to a linked list in the TreeItem class, along with the text item. That linked list could store the list of all the page numbers on which the word was found. (See what fun you can have with complex data structures. Just have a few cups of strong coffee first!)

The Tree Class

As with the previous data structures, the base Tree class stores the bulk of the code required to make the data structure work. The class contains but a single data item:

```
Private tiHead As TreeItem
```

As with the other data structures, tiHead is an anchor for the entire data structure. It points to the first item in the binary tree. From there, the items point to other items.

In addition, the Tree class module contains two module-level variables:

```
' These private variables are used when
' adding new nodes.
Private mblnAddDupes As Boolean
Private mvarItemToAdd As Variant
```

The method that adds items to the binary tree uses these module-level variables. If they weren't module-level, the code would have to pass them as parameters to the appropriate methods. What's wrong with that? Because the Add method is recursive, the procedure might call itself many times. Each call takes up memory that isn't released until the entire procedure has completed. If your tree is very deep, you could eat up a large chunk of stack space adding a new item. To avoid that issue, the Tree class doesn't pass these values as parameters; it just makes them available to all the procedures in the Tree class, no matter where they're called.

Adding a New Item

When adding items to a binary tree, you may or may not want to add an item if its value already appears in the data structure. To make it easy to distinguish between those two cases, the Tree class contains two separate methods: Add and AddUnique, shown in Listing 8.13. Each of the methods ends up calling the AddNode procedure, shown in Listing 8.14.

Listing 8.13: The Tree Class Provides Two Ways to Add New Items

```
Public Sub Add(varNewItem As Variant)
    ' Add a new node, allowing duplicates.
    ' Use module variables to place as little as
    ' possible on the stack in recursive procedure calls.
    mblnAddDupes = True
    mvarItemToAdd = varNewItem
    Call AddNode(tiHead)
End Sub

Public Sub AddUnique(varNewItem As Variant)
    ' Add a new node, skipping duplicate values.
    ' Use module variables to place as little as
    ' possible on the stack in recursive procedure calls.
    mblnAddDupes = False
    mvarItemToAdd = varNewItem
    Call AddNode(tiHead)
End Sub
```

The recursive AddNode procedure adds a new node to the binary tree pointed to by the TreeItem pointer it receives as a parameter. Once you get past the recursive nature of the procedure, the code is reasonably easy to understand:

- If the TreeItem pointer, ti, is Nothing, it sets the pointer to a new TreeItem and places the value into that new node:

  ```
  If ti Is Nothing Then
      Set ti = New TreeItem
      ti.Value = mvarItemToAdd
  ```

- If the pointer isn't Nothing, then:
  - If the new value is less than the value in ti, the code calls AddNode with the left child pointer of the current node:

  ```
  If mvarItemToAdd < ti.Value Then
      Set ti.LeftChild = AddNode(ti.LeftChild)
  ```

  - If the new value is greater than the value in ti, the code calls AddNode with the right child pointer of the current node:

  ```
  ElseIf mvarItemToAdd > ti.Value Then
      Set ti.RightChild = AddNode(ti.RightChild)
  ```

- If the new value is equal to the current value, then, if you've instructed the code to add duplicates, the code arbitrarily calls AddNode with the right child pointer. (You could use the left instead, if you wanted.) If you don't want to add duplicates, the procedure just returns.

```
Else
    ' You're adding a node that already exists.
    ' You could add it to the left or to the right,
    ' but this code arbitrarily adds it to the right.
    If mblnAddDupes Then
        Set ti.RightChild = AddNode(ti.RightChild)
    End If
End If
```

- Sooner or later, after calling AddNode for each successive child node, the code will find a pointer that is Nothing, at which point it takes the action in the first step. Because nothing follows the recursive call to AddNode in the procedure, after each successive layer has finished processing, the code just works its way back up the list of calls.

Listing 8.14: The Recursive AddNode Procedure Adds a New Node to the Tree

```
Private Function AddNode(ti As TreeItem) As TreeItem
    ' Add a node to the tree pointed to by ti.
    ' Module variables used:
    '    mvarItemToAdd: the value to add to the tree.
    '    mblnAddDupes: Boolean indicating whether to add items
    '      that already exist or to skip them.
    If ti Is Nothing Then
        Set ti = New TreeItem
        ti.Value = mvarItemToAdd
    Else
        If mvarItemToAdd < ti.Value Then
            Set ti.LeftChild = AddNode(ti.LeftChild)
        ElseIf mvarItemToAdd > ti.Value Then
            Set ti.RightChild = AddNode(ti.RightChild)
        Else
            ' You're adding a node that already exists.
            ' You could add it to the left or to the right,
            ' but this code arbitrarily adds it to the right.
```

```
                If mblnAddDupes Then
                    Set ti.RightChild = AddNode(ti.RightChild)
                End If
            End If
        End If
        Set AddNode = ti
    End Function
```

Adding a New Node: Walking the Code

Suppose you were to try adding a new node to the tree shown in Figure 8.35 with the value “m”. Table 8.2 outlines the process involved in getting the node added. (This discussion assumes that the class module’s tiHead member points to the tree shown in Figure 8.35.) For each step, the table includes, in column 1, the recursion level—that is, the number of times the procedure has called itself.

FIGURE 8.35 Revisiting the alphabetic tree, attempting to add a new node

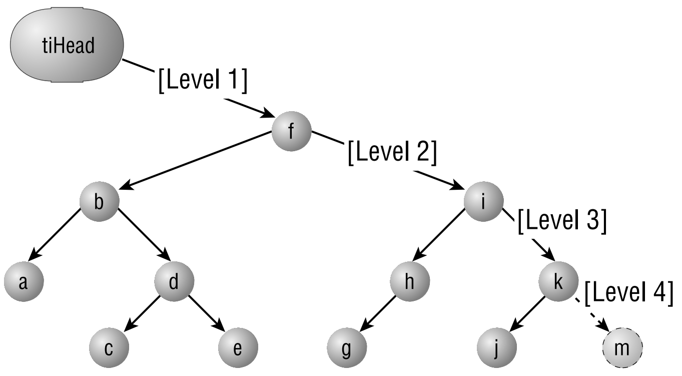


TABLE 8.2: Recursive Steps to Add “m” to the Sample Tree

Level	Action
0	You call the Add method, passing the value “m”.
0	The Add method sets mblnAddDupes to True and sets varNewItem to the value “m”. It then calls the AddNode method, passing the pointer to the first item in the tree (a node with the value “f”, in this case). [Call to Level 1]

TABLE 8.2: Recursive Steps to Add "m" to the Sample Tree *(continued)*

Level	Action
1	AddNode checks to see whether ti is Nothing. It's not. (It points to the node containing "f".)
1	Because "m" is greater then "f", AddNode calls itself, passing the right child pointer of the node ti currently points to. (That is, it passes a pointer to the node containing "i".) [Call to Level 2]
2	AddNode checks to see whether ti is Nothing. It's not. (It points to the node containing "i".)
2	Because "m" is greater then "i", AddNode calls itself, passing the right child pointer of the node ti currently points to. (That is, it passes a pointer to the node containing "k".) [Call to Level 3]
3	AddNode checks to see whether ti is Nothing. It's not. (It points to the node containing "k".)
3	Because "m" is greater then "k", AddNode calls itself, passing the right child pointer of the node ti currently points to (that is, the right child pointer of the node containing "k", which is Nothing). [Call to Level 4]
4	AddNode checks to see whether ti is Nothing. It is, so it creates a new node, sets the pointer passed to it (the right child of the node containing "k") to point to the new node, and returns.
4	There's nothing else to do, so the code returns. [Return to Level 3]
3	There's nothing else to do, so the code returns. [Return to Level 2]
2	There's nothing else to do, so the code returns. [Return to Level 1]
1	The code returns back to the original caller.

Traversing the Tree

As mentioned earlier in this discussion, there are three standard methods for traversing a tree: inorder, preorder, and postorder. Because of the recursive nature of these actions, the code for each is simple; it is shown in Listing 8.15. The class provides three Public methods (WalkInOrder, WalkPreOrder, WalkPostOrder). Each of these calls a Private procedure, passing a pointer to the head of the tree as the only argument. From then on, each of the Private procedures follows the prescribed order in visiting nodes in the tree.

Of course, in your own applications, you'll want to do something with each node besides print its value to the Immediate window. In that case, modify the three Private procedures to do what you need done with each node of your tree.

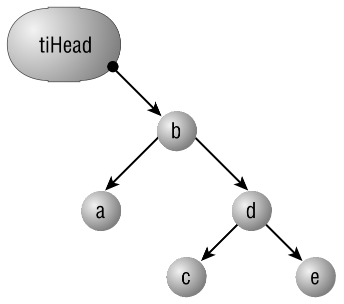
Listing 8.15: Because of Recursion, the Code to Traverse the Tree Is Simple

```
Public Sub WalkInOrder()
    Call InOrder(tiHead)
End Sub

Public Sub WalkPreOrder()
    Call PreOrder(tiHead)
End Sub

Public Sub WalkPostOrder()
    Call PostOrder(tiHead)
End Sub

Private Sub InOrder(ti As TreeItem)
    If Not ti Is Nothing Then
        Call InOrder(ti.LeftChild)
        Debug.Print ti.Value; " ";
        Call InOrder(ti.RightChild)
    End If
End Sub

Private Sub PreOrder(ti As TreeItem)
    If Not ti Is Nothing Then
        Debug.Print ti.Value; " ";
        Call PreOrder(ti.LeftChild)
        Call PreOrder(ti.RightChild)
    End If
End Sub

Private Sub PostOrder(ti As TreeItem)
    If Not ti Is Nothing Then
        Call PostOrder(ti.LeftChild)
        Call PostOrder(ti.RightChild)
        Debug.Print ti.Value; " ";
    End If
End Sub
```

Traversing a Tree: Walking the Code

In order to understand tree traversal, assume you'd like to perform a postorder traversal of the tree shown in Figure 8.36. Although this example doesn't include many nodes, the steps are the same no matter the size of the tree.

FIGURE 8.36
Use this small example for the tree traversal example



To visit each node in the tree using the postorder traversal, follow the steps listed in Table 8.3. (You'll want to keep a firm finger on the diagram as you work your way through these steps.)

TABLE 8.3: Recursive Steps to Perform a Postorder Traversal

Level	Action
0	Call the WalkPostOrder method of the Tree class.
1	The code in WalkPostOrder calls the PostOrder procedure, passing tiHead as a parameter. [Call to Level 2]
2	PostOrder checks to see whether ti (its parameter) is Nothing. It's not (it's a reference to the node that contains "b"), so it can continue.
2	PostOrder calls itself, passing the left child pointer of the node ti points to. (That is, it passes a pointer to the node containing "a".) [Call to Level 3]
3	PostOrder checks to see whether ti (its parameter) is Nothing. It's not (it's a reference to the node that contains "a"), so it can continue.
3	PostOrder calls itself, passing the left child pointer of the node ti points to. (That is, it passes a pointer that is Nothing.) [Call to Level 4]
4	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 3]
3	PostOrder calls itself, passing the right child pointer of the node ti points to. (That is, it passes a pointer that is Nothing.) [Call to Level 4]

TABLE 8.3: Recursive Steps to Perform a Postorder Traversal *(continued)*

Level	Action
4	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 3]
3	PostOrder prints its value ("a") and then returns. [Return to Level 2]
2	PostOrder calls itself, passing the right child pointer of the node ti points to. (That is, it passes a pointer to the node containing "d".) [Call to Level 3]
3	PostOrder checks to see whether ti (its parameter) is Nothing. It's not (it's a reference to the node that contains "d"), so it can continue.
3	PostOrder calls itself, passing the left child pointer of the node ti points to. (That is, it passes a pointer to the node containing "c".) [Call to Level 4]
4	PostOrder checks to see whether ti (its parameter) is Nothing. It's not (it's a reference to the node that contains "c"), so it can continue.
4	PostOrder calls itself, passing the left child pointer of the node ti points to. (That is, it passes a pointer that's Nothing.) [Call to Level 5]
5	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 4]
4	PostOrder calls itself, passing the right child pointer of the node ti points to. (That is, it passes a pointer that's Nothing.) [Call to Level 5]
5	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 4]
4	PostOrder prints its value ("c") and then returns. [Return to Level 3]
3	PostOrder calls itself, passing the right child pointer of the node ti points to. (That is, it passes a pointer to the node containing "e".) [Call to Level 4]
4	PostOrder checks to see whether ti (its parameter) is Nothing. It's not (it's a reference to the node that contains "e"), so it can continue.
4	PostOrder calls itself, passing the left child pointer of the node ti points to. (That is, it passes a pointer that's Nothing.) [Call to Level 5]
5	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 4]
4	PostOrder calls itself, passing the right child pointer of the node ti points to. (That is, it passes a pointer that's Nothing.) [Call to Level 5]
5	PostOrder checks to see whether ti (its parameter) is Nothing. It is, so it can't do anything and just returns. [Return to Level 4]

TABLE 8.3: Recursive Steps to Perform a Postorder Traversal *(continued)*

Level	Action
4	PostOrder prints its value ("e") and then returns. [Return to Level 3]
3	PostOrder prints its value ("d") and then returns. [Return to Level 2]
2	PostOrder prints its value ("b") and then returns to WalkPostOrder. [Return to Level 1, and exit]

Optimizing the Traversals

If you worked your way through the many steps it took to traverse the simple tree, you can imagine how much work it takes to perform the operation on a large tree. You could optimize the code a bit by checking to see whether the child node is Nothing before you recursively call the procedure. That is, you could modify InOrder like this:

```
Private Sub InOrder(ti As TreeItem)
    If Not ti Is Nothing Then
        If Not ti.LeftChild Is Nothing Then
            Call InOrder(ti.LeftChild)
        End If
        Debug.Print ti.Value; " ";
        If Not ti.RightChild Is Nothing Then
            Call InOrder(ti.RightChild)
        End If
    End If
End Sub
```

This code would execute a tiny bit faster than the original InOrder tree-traversal procedure (one less procedure call for both children of all the bottom-level nodes), but it's a little harder to read.

The Sample Project

The code in the sample module performs some simple tree manipulations: It adds nodes, walks the tree in all the traversal orders, and deletes some nodes using the TreeDelete method (not covered in this book, but the code is there in the Tree class for you to use). Try the TestTrees procedure in the TreeTest module to see how you might use a binary tree in your applications. The first few tests correspond to the tree shown in Figure 8.32 earlier in this chapter, and you can use the code in the project to test your understanding of the different traversal orders.

What Didn't We Cover?

We actually omitted more about binary trees than we covered here. Binary trees usually fill multiple chapters in textbooks for courses in standard data structures. Consider the following:

- Deleting nodes from binary trees is a science unto itself. The sample project includes code to delete nodes from a tree, but it's just one of many solutions, and possibly not the most efficient one.
- Balancing trees is crucial if you want optimized performance. For example, if you add previously sorted data to a tree, you end up with a degenerate tree—all the nodes are linked as the right child of the parent. In other words, you end up with a linked list. Searching through linked lists isn't particularly efficient, and you lose the benefit of using a binary tree. Courses in data structures normally cover various methods you can use to keep your trees balanced (that is, with the left and right subtrees having approximately the same depth).
- In a course on data structures, you'll normally find a number of variants on binary trees (B-trees, for example) that also take into account data stored on disk.

If you're interested in finding out more about these variants on binary trees, find a good textbook that focuses on data structures. Of course, most such textbooks are written for Pascal programmers (most universities use Pascal as a teaching language), so you'll need to do some conversion. However, it's not hard once you've got the hang of it.

Summary

In this chapter, we've taken a stab at revisiting Computer Science 201: Data Structures or a similar university course you might have taken once. Of course, in this limited space, we can do little more than provide a "proof of concept"—the technique of using self-referential, abstract data structures in VBA works, and it works well. Because of the availability of class modules, you can use the techniques provided here to create hybrid data structures that you just couldn't manage with VBA's arrays and collections. Linked lists of binary trees, collections of linked lists, linked lists of linked lists—all these, and more, are possible, but we suggest drawing pictures on paper first!

Note that all the ideas presented in this chapter rely on data in memory. That is, there's no concept of persistent storage when working with these data structures. If you want to store information contained in one of these abstract structures from one session to the next, you'll need to design some storage mechanism, whether it be in the Registry, an INI file, or a database table. In addition, if you run out of memory, you'll receive a run-time error when you attempt to use the New keyword. Obviously, this shouldn't happen. In production code, you'd want to add error handling to make sure your application didn't die under low-memory conditions.

This chapter presented a number of topics to keep in mind when working with data in memory, including:

- Using class modules to represent elements of linked data structures
- Building stacks, queues, ordered linked lists, and binary trees using class modules
- Using recursion to work with and traverse binary trees