VISUAL BASIC COLLECTIONS AND HASH TABLES

Thomas Niemann

ePaper Press

Preface

Hash tables offer a method for quickly storing and accessing data based on a key value. When you access a Visual Basic *collection* using a key, a hashing algorithm is used to determine the location of the associated record. Collections, as implemented, do not support duplicate keys. For this reason, and other performance considerations, you may wish to code your own hashing algorithm.

I'll discuss *open hashing*, where data is stored in a *node*, and nodes are chained from entries in a hash table. I'll also examine the effect of hash table size on execution time. This is followed by a section on hashing algorithms. Then we'll look at different techniques for representing nodes. Finally, I'll compare the various strategies, examining execution time and storage requirements.

Source code for all examples may be downloaded from the site listed below. Cormen [2001] and Knuth [1998] both contain excellent discussions on hashing. Stephens [1998] is a good reference for hashing and node representation in Visual Basic. This article also appears in the spring 1999 issue of the Technical Guide to Visual Programming, published by Fawcette Technical Publications.

THOMAS NIEMANN
Portland, Oregon

web site: epaperpress.com

Open Hashing

A hash table is simply an array that is addressed via a hash function. For example, in Figure 1, **`HashTable`** is an array with 8 elements. Each element is a pointer to a linked list of numeric data. The hash function for this example simply divides the data key by 8, and uses the remainder as an index into the table. This yields a number from 0 to 7. Since the range of indices for **`HashTable`** is 0 to 7, we are guaranteed that the index is valid.

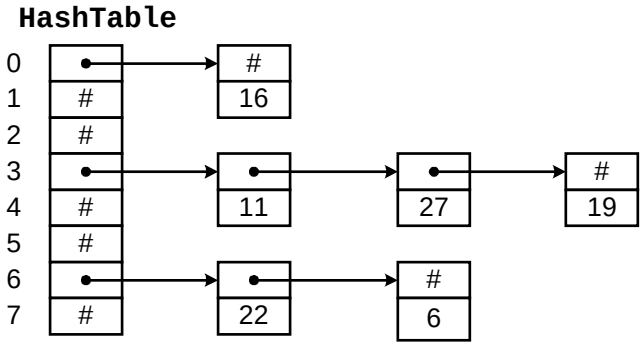

Figure 1: A Hash Table

To insert a new item in the table, we hash the key to determine which list the item goes on, and then insert the item at the beginning of the list. For example, to insert 11, we divide 11 by 8 giving a remainder of 3. Thus, 11 goes on the list starting at **`HashTable(3)`**. To find a number, we hash the number and chain down the correct list to see if it is in the table. To delete a number, we find the number and remove the node from the linked list.

Entries in the hash table are dynamically allocated and entered on a linked list associated with each hash table entry. This technique is known as chaining. If the hash function is uniform, or equally distributes the data keys among the hash table indices, then hashing effectively subdivides the list to be searched. Worst-case behavior occurs when all keys hash to the same index. Then we simply have a single linked list that must be sequentially searched. Consequently, it is important to choose a good hash function. The following sections describe several hashing algorithms.

Table Size

Assuming *n* data items, the hash table size should be large enough to accommodate a reasonable number of entries. Table 1 shows the maximum time required to search for all entries in a table containing 10,000 items.

size	*time (ms)*
1	23,544
10	2,473
100	331
1,000	100
10,000	70

Table 1: Table Size vs Search Time

A small table size substantially increases the time required to find a key. A hash table may be viewed as a collection of linked lists. As the table becomes larger, the number of lists increases, and the average number of nodes on each list decreases. If the table size is 1, then the table is really a single linked list of length *n*. Assuming a perfect hash function, a table size of 2 has two lists of length *n*/2. If the table size is 100, then we have 100 lists of length *n*/100. This greatly reduces the length of the list to be searched. There is considerable leeway in the choice of table size.

Hash Functions

In the previous example, we determined a hash value by examining the remainder after division. In this section we'll examine several algorithms that compute a hash value.

Division Method (TableSize = Prime)

A hash value, from 0 to (*HashTableSize* - 1), is computed by dividing the key value by the size of the hash table and taking the remainder:

```
Public Function Hash(ByVal Key As Long) As Long
   Hash = Key Mod HashTableSize
End Function
```

Selecting an appropriate *HashTableSize* is important to the success of this method. For example, a *HashTableSize* divisible by two would yield even hash values for even keys, and odd hash values for odd keys. This is an undesirable property, as all keys would hash to an even value if they happened to be even. If *HashTableSize* is a power of two, then the hash function simply selects a subset of the key bits as the table index. To obtain a more random scattering, *HashTableSize* should be a prime number not too close to a power of two.

Multiplication Method (TableSize = 2^N)

The multiplication method may be used for a *HashTableSize* that is a power of 2. The key is multiplied by a constant, and then the necessary bits are extracted to index into the table. One method uses the fractional part of the product of the key and the golden ratio, or $\left(\sqrt{5} - 1\right)/2$. For example, assuming a word size of 8 bits, the golden ratio is multiplied by 28 to obtain 158. The product of the 8-bit key and 158 results in a 16-bit integer. For a table size of 25 the 5 most significant bits of the least significant word are extracted for the hash value. The following definitions may be used for the multiplication method:

```
' 8-bit index
Private Const K As Long = 158

' 16-bit index
Private Const K As Long = 40503

' 32-bit index
Private Const K As Long = 2654435769

' bitwidth(index)=w, size of table=2^m
Private Const S As Long = 2^(w - m)
Private Const N As Long = 2^m - 1
Hash = ((K * Key) And N) \ S
```

For example, if *HashTableSize* is 1024 (2^{10}), then a 16-bit index is sufficient and **S** would be assigned a value of $2^{(16 - 10)} = 64$. Constant **N** would be 2^{10} - 1, or 1023. Thus, we have:

```
Private Const K As Long = 40503
Private Const S As Long = 64
Private Const N As Long = 1023

Public Function Hash(ByVal Key As Long) As Long
    Hash = ((K * Key) And N) \ S
End Function
```

Variable String Addition Method (TableSize = 256)

To hash a variable-length string, each character is added, modulo 256, to a total. A hash value, range 0-255, is computed.

```
Public Function Hash(ByVal S As String) As Long
   Dim h As Byte
   Dim i As Long

   h = 0
   For i = 1 to Len(S)
      h = h + Asc(Mid(S, i, 1))
   Next i
   Hash = h
End Function
```

Variable String Exclusive-Or Method (Tablesize = 256)

This method is similar to the addition method, but successfully distinguishes similar words and anagrams. To obtain a hash value in the range 0-255, all bytes in the string are exclusive-or'd together. However, in the process of doing each exclusive-or, a random component is introduced.

```
Private Rand8(0 To 255) As Byte

Public Function Hash(ByVal S As String) As Long
   Dim h As Byte
   Dim i As Long

   h = 0
   For i = 1 To Len(S)
      h = Rand8(h Xor Asc(Mid(S, i, 1)))
   Next i
   Hash = h
End Function
```

Rand8 is a table of 256 8-bit unique random numbers. The exact ordering is not critical. The exclusive-or method has its basis in cryptography, and is quite effective (Pearson [1990]).

Variable String Exclusive-Or Method (Tablesize <= 65536)

If we hash the string twice, we may derive a hash value for an arbitrary table size up to 65536. The second time the string is hashed, one is added to the first character. Then the two 8-bit hash values are concatenated together to form a 16-bit hash value.

```
Private Rand8(0 To 255) As Byte

Public Function Hash(ByVal S As String,
      ByVal HashTableSize As Long) As Long
   Dim h1 As Byte
   Dim h2 As Byte
   Dim c As Byte
   Dim i As Long

   if Len(S) = 0 Then
      Hash = 0
      Exit Function
   End If

   h1 = Asc(Mid(S, 1, 1))
   h2 = h1 + 1
   For i = 2 To Len(S)
      c = Asc(Mid(S, i, 1))
      h1 = Rand8(h1 Xor c)
      h2 = Rand8(h2 Xor c)
   Next i

   ' Hash is in range 0 .. 65535
   Hash = (h1 * 256) + h2
   ' scale Hash to table size
   Hash = Hash Mod HashTableSize
End Function
```

Hashing strings is computationally expensive, as we manipulate each byte in the string. A more efficient technique utilizes a DLL, written in C, to perform the hash function. Included in the download is a test program that hashes strings using both C and Visual Basic. The C version is typically 20 times faster.

Node Representation

If you plan to code your own hashing algorithm, you'll need a way to store data in nodes, and a method for referencing the nodes. This may be done by storing nodes in objects and arrays. I'll use a linked-list to illustrate each method.

Objects

References to objects are implemented as pointers in Visual Basic. One implementation simply defines the data fields of the node in a class, and accesses the fields from a module:

```
' in class CObj
Public NextNode As CObj
Public Value As Variant

' in module Main
Private hdrObj As CObj
Private pObj As CObj

' add new node to list
Set pObj = New CObj
Set pObj.NextNode = hdrObj
Set hdrObj = pObj
pObj.Value = value

' find value in list
Set pObj = hdrObj
Do While Not pObj Is Nothing
   If pObj.Value = value Then Exit Do
   Set pObj = pObj.NextNode
Loop

' delete first node
Set pObj = hdrObj.NextNode
Set hdrObj = pObj
Set pObj = Nothing
```

In the above code, **`pObj`** is internally represented as a pointer to the class. When we add a new node to the list, an instance of the node is allocated, and a pointer to the node is placed in **`pObj`**. The expression **`pObj.Value`** actually de-references the pointer, and accesses the **`Value`** field. To delete the first node, we remove all references to the underlying class.

Arrays

An alternative implementation allocates an array of nodes, and the address of each node is represented as an index into the array.

```
' list header
Private hdrArr As Long

' next free node
Private nxtArr As Long

' fields of node
Private NextNode(1 To 100) As Long
Private Value(1 To 100) As Variant

' initialization
hdrArr = 0
nxtArr = 1

' add new node to list
pArr = nxtArr
nxtArr = nxtArr + 1
NextNode(pArr) = hdrArr
hdrArr = pArr
Value(pArr) = value

' find value in list
pArr = hdrArr
Do While pArr <> 0
   If Value(pArr) = value Then Exit Do
   pArr = NextNode(pArr)
Loop
```

Each field of a node is represented as a separate array, and referenced by subscripts instead of pointers. For a more robust solution, there are several problems to solve. In this example, we've allowed for 100 nodes, with no error checking. Enhancements could include dynamically adjusting the arrays size when nxtArr exceeds array bounds. Also, no provisions have been made to free a node for possible re-use. This may be accomplished by maintaining a list of subscripts referencing free array elements, and providing functions to allocate and free subscripts. Included in the download is a class designed to manage node allocation, allowing for dynamic array resizing and node re-use.

Comparison

Table 2 illustrates resource requirements for a hash table implemented using three strategies. The *array* method represents nodes as elements of an array, the *object* method represents nodes as objects, while the *collection* method utilizes the built-in hashing feature of Visual Basic collections.

		time(ms)				
n	*method*	*insert*	*find*	*delete*	*kBytes*	*faults*
	array	10	10	10	72	17
1,000	object	50	20	40	104	26
	collection	60	40	40	100	25
	array	40	50	40	228	132
5,000	object	301	90	261	744	186
	collection	490	220	220	516	129
	array	90	101	90	412	297
10,000	object	711	200	822	1,604	401
	collection	1,111	471	491	1,044	261
	array	450	541	540	2,252	1,524
50,000	object	7,481	1,062	13,279	8,480	2,118
	collection	9,394	2,623	2,794	5,420	1,355
	array	912	1,141	1,122	4,504	3,047
100,000	object	22,182	2,103	48,570	17,072	4,266
	collection	27,830	5,658	5,918	10,896	2,724

Table 2: Resource Requirements

Memory requirements and page faults are shown for insertion only. Hash table size for arrays and objects was $1/10^{th}$ the total count. Tests were run using a 200Mhz Pentium with 64Meg of memory on a Windows NT 4.0 operating system. Statistics for memory use and page faults were obtained from the NT Task Manager. Code may be downloaded that implements the above tests, so you may repeat the experiment.

It is immediately apparent that the array method is fastest, and consumes the least memory. Objects consume four times as much memory as arrays. In fact, overhead for a single object is about 140 bytes. Collections take about twice as much room as arrays.

An interesting anomaly is the high deletion time associated with objects. When we increase the number of nodes from 50,000 to 100,000 (a factor of 2), the time for deletion increases from 13 to 48 seconds (a factor of 4). During deletion, no page faults were noted. Consequently, the extra overhead was compute time, not I/O time. One implementation used at run-time for freeing memory involves maintaining a list, ordered by memory location, of free nodes. When memory is freed, the list is traversed so that memory to be released can be returned to the appropriate place in the list. This is done so that memory *chunks* may be recombined when adjacent chunks are freed. Unfortunately, this algorithm runs in $\boldsymbol{O}(n^2)$ time, where execution time is roughly proportional to the square of the number of items being released.

I encountered a similar problem while working on compilers for Apollo. In this case, however, the problem was exacerbated by page faults that occurred while traversing links. The solution involved an in-core index that reduced the number of links traversed.

Conclusion

Hashing is an effective method to quickly access data using a key value. Fortunately, Visual Basic includes collections; an effective solution that is easy to code. In this article, we compared collections with hand-coded solutions. Along the way we discovered that storing data in objects for large datasets can incur substantial penalties in both execution time and storage requirements. In this case, you can make significant gains by coding your own algorithm, utilizing arrays for node storage. For smaller datasets, however, collections remain a good choice.

Bibliography

Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest [2001]. Introduction to Algorithms. McGraw-Hill, New York.

Knuth, Donald. E. [1998]. The Art of Computer Programming, Volume 3, Sorting and Searching. Addison-Wesley, Reading, Massachusetts.

Pearson, Peter K [1990]. Fast Hashing of Variable-Length Text Strings. *Communications of the ACM*, 33(6):677-680, June 1990.

Stephens, Rod [1998]. Ready-to-Run Visual Basic Algorithms. John Wiley & Sons, Inc., New York.