Ballet Mechanique

by Julian Bucknall

Recently our esteemed editor, Chris Frizelle, and I had an email conversation discussing what kind of articles I could be writing for *The Delphi Magazine*. He mentioned that, although articles about databases and visual widgets and Internet topics are all very interesting and are covered in this magazine (extremely well) and in the Good, Bad and Ugly books on your local bookstore shelves (maybe not quite as well), every magazine under the sun is also publishing them. What he wanted and proposed to me was a more solid, hard core series of articles for medium to advanced level programmers. The general theme? On what we may call "computer science" topics. In other words, to assume that you, the programmer-reader, know about linked lists, stacks and insertion sorts and would like to advance and extend your knowledge on some other algorithms and structures. Over the next few articles we shall be extending your toolset and mindset with some of the classical algorithms cast in the Delphi mold.

And, dare I say it, I, your guide on this quest, shall be doing some learning of my own at the same time. Some of the topics we shall be exploring will be new to me; although I may have heard of them, I certainly have never used them or developed with them before. On the other hand, other topics we shall cover will be very familiar to me and come out of my work at TurboPower Software and for my EZDSL data structures library *[check out the bonus\tpower\pub\misc\funcs folder on your Collection '97 CD-ROM, which you have of course purchased, for a copy. Ed]*. To be unfair, I won't tell you which is which, so you'll just have to trust me!

Let's get to work then. This article is going to be the first in a small series on hash tables.

On Track

Before you raise your hands in horror ("What the heck is a hash table?"), let's lay some foundations and get us thinking along the same lines. I assume you know from programming in Delphi what a `TStringList` is. The concept is pretty simple: a class that represents an array of strings and you can have an object (or a pointer to a record, or a generic 32-bit value) associated with each string. If the `TStringList` you are using is not in sorted order, then you will agree that the only way to find a particular string (and hence its associated object) is to start at the beginning (or end), wander through the list and compare each string in the list with the one you want (Listing 1). If you get through the whole list without finding the string then, by gum, it ain't there. If you do find it, then you have the index of the string and hence can get at the object.

If there are N strings in the list, you may find the element you want straight away ("Hey, it's at element 0"), or you may find it right at the end ("Rats, it's at element N-1"). A minor bit of probability theory later and you can calculate that the expected number of comparisons you will have to make to find an existing string is ½N and this is the same as the number of element accesses as well. So, for a list of 1024 elements, you will retrieve and examine 512 elements on average to find the one you want.

To exaggerate the effect, if you pretend that each access to an element requires reading from a file, I'm sure that you will agree that this is not very good.

Ahhh, you say, but a `TStringList` can be sorted alphabetically. In this particular situation, to find a given string we apply an algorithm called binary searching. Look at the element at ½N. Compare its string to the one we want. If it is equal, we've discovered the element we want. If it is greater than our string then we can immediately state that the element we want is in the first half of the list. If it is less than our string, then we know that the element we want is in the latter half of the list. In either case, we can ignore the other half of the list. Repeat the same process with the half list we have and we'll end up with a quarter list. Repeat again and we'll end up with an eighth list, and so on. Every time we halve the number of elements to look through and so eventually we're left with one element that is either the one we want, or it isn't. The standard algorithm is shown in Listing 2.

This is a slight variation on the standard binary search algorithm

➤ *Listing 1*

```
var
  i, Index : integer;
  FoundIt : boolean;
...
FoundIt := False;
for i := 0 to (MyList.Count - 1) do
  if (MyList[i] = StringToFind)
    then begin
    Index := i;
    FoundIt := True;
    Break;
  end;
```

➤ *Listing 2*

```
var
  LeftIndex, RightIndex, MidIndex : integer;
  FoundIt : boolean
LeftIndex := 0;
RightIndex := pred(MyList.Count);
FoundIt := false;
while (LeftIndex <= RightIndex) do begin
  MidIndex := (LeftIndex + RightIndex) div 2;
  if (MyList[MidIndex] = StringToFind) then begin
    FoundIt := true;
    Break;
  end else if (MyList[MidIndex] < StringToFind) then
    LeftIndex := MidIndex + 1
  else {it's greater than}
    RightIndex := MidIndex - 1;
end;
```

which makes use of the Delphi `Break` statement and an extra variable to help out a bit. Also, the algorithm is generally cast as a `repeat..until` loop in the literature, but having used this algorithm in various guises over the years, I think it makes more sense as a `while` loop, especially if the number of elements is ever likely to be zero. Anyway, if you trace through this code on paper, you'll come out at the statement after the loop with either `FoundIt` set to `True` and `MidIndex` pointing to the element that matches the string we want, or `FoundIt` will be `False`. The algorithm works by bracketing the element we want with a range defined by `LeftIndex` and `RightIndex` and then gently reducing the range by half, each time through the loop. Looking at the code you'll notice that there is one access to an element per loop (at least you could force this by copying the element into a local variable), with generally two comparisons per loop.

Since the algorithm reduces the range by half each time though the loop (actually a little more than a half), playing around with some example numbers you can see that the number of element accesses you will have to make is $\log_2 N$. So for example when we have 1024 elements, at worst we'll need to make 10 accesses and 20 comparisons to find the correct element that matches the string we need to find. This is more like it! Since the number of accesses is proportional to $\log_2 N$, we can see that to find an element in a sorted set of 1,000,000 strings we'll need about 20 accesses using a binary search algorithm, compared with half a million using a standard sequential search. Amazing.

Now, of course, we have to do some work when we add elements to the list in order to maintain the sorted order and this would be much more work than just maintaining a list in an unsorted or arrival sequence. In fact we would have to try and find the element we want to add in the list and then add it at the point we arrive at (in Listing 1, if `FoundIt` was `False` at the end of the loop then `LeftIndex` will hold the index to insert the new element at). So we are, in effect, trading off extra work during the insertion phase to make the search phase that much faster.

```
function CalcELFHash(const S : string) : integer;
var
  G : longint;
  i : integer;
begin
  Result := 0;
  for i := 1 to length(S) do begin
    Result := (Result shl 4) + ord(S[i]);
    G := Result and $F0000000;
    if (G <> 0) then
      Result := Result xor (G shr 24);
    Result := Result and (not G);
  end;
  Result := Result mod ElementCount;
end;
```

➤ *Listing 3*

Blender? Oh Yeah

So is there anything better than this? Is there some other data structure and/or algorithm that will enable us to find a specific element in a list of N strings with less than $\log_2 N$ accesses? Well, yes, there is, otherwise I wouldn't be writing this. The structure we need is a hash table and the number of accesses required will generally be one.

Let us imagine that we have a magic function that takes in a string and spits out the index in the list we require. All we need to do then is to get that particular element in our list and, bingo, we're done. When we add a string to the list we call the same magic function which will tell us where to put the string in the list. Is this magic function too good to be true, or what? We'll see.

This magic function is called a hash. Like many computer terms hash can either be a noun or a verb, as we'll see. As I described above, it takes in a string, does some kind of manipulation on that string and returns a value that can be used as an index into an array or table (in our case, between 0 and one less than the number of elements in the array). Different strings would hash to different values, obviously (it would a pretty silly hash if every string hashed to the same value for what would we do then?). We'll see that hash functions tend to be great randomizers: for the better ones there seems to be no correlation between the string and the eventual hash value.

Let's investigate some examples of hash functions so that we can get a 'feel' for what's good and what's bad. Mmmm, that of course presupposes we know the meaning of good and bad when applied to hash functions! A definition of a good hash function is one that will produce a good spread of values for our particular set of strings. An amazingly good hash function will produce a different value for each string we have; in other words, there are no two strings in our set that hash to the same value (ie, that do not collide, using the hash table vernacular). A bad hash function is one that produces a large number of equal values for our set of strings (ie, a large number of collisions).

OK, having said that, here's my first hash function (Hash 1): return the length of the string. Suppose that we are storing the surnames of our N friends in our hash table: is this a good hash for this application? Well, I don't know about you, but thinking about my friends' last names, I have none with a zero (!), one, two or three character surname, I have several with four or five or six letters in their last name and I have absolutely none at all with more than 12 letters. So I'd say that this particular hash function is absolutely appalling. If I had 50 friends the great majority of the list would be empty and I would have to try and squeeze 50 names into less than 10 entries in the list.

So, here's another: take the first letter of the name and use its ASCII value (Hash 2). The merest moment's thought would soon

	None	1	2	3	4	5	6	7	8	9
Hash 1	46	0	3	0	0	0	3	0	0	1
Hash 2	37	5	6	4	1	0	0	0	0	0
Hash 3	32	12	7	1	1	0	0	0	0	0
Hash 4	27	21	3	2	0	0	0	0	0	0
Hash 5	27	20	5	1	0	0	0	0	0	0
Expected	28	18	6	1	0	0	0	0	0	0

➤ *Table 1:*
Distributions of hash values for various hashes in a 53 element table

dispatch that hash function as being too bad for words, so we won't say any more on the subject.

All right, how about going a little further with that idea? Let's treat the letters in the names as ASCII values, and adding them all up to some big value and then taking the modulus of this result with the number of elements in the list (Hash 3)? Recall that `X mod N` returns a value between 0 and N-1. This seems much better, at least on paper. As an experiment, I wrote an application that took the surnames of all the 33 employees at TurboPower in an old phone list I had on disk and generated their hashes for a 50 element table: in other words I made sure the hashes generated had values between 0 and 49.

There were 23 hash values not generated at all, 21 hash values generated from one name and 6 from two names. In comparison, the first hash had 43 values not generated at all, 3 values from 2 names, 3 from 6 and one from 9; the second hash had 34 not generated at all, 5 from 1, 6 from 2, 4 from 3, and 1 value that was generated from 4 names. So, this small experiment shows we're on the right track. Could we do better?

The fourth hash we shall look at is the ELF hash (Listing 3), devised a while ago for UNIX object files (Hash 4). Here the results didn't seem to be much better, despite the complexity of the algorithm: it does quite a bit of internal randomization. I found 23 values not generated at all, 22 values from one name, 4 from 2 and 1 generated from 3 names.

Maybe we're missing something? One small point to make (which isn't obvious from our experiments) is that a prime number is usually used for a divisor because primes tend to distribute remainders (which is what we get with the `mod` operator) more randomly than non-primes. So, let's ensure that our list has a prime number of elements, say 53. Rerunning the same test on the TurboPower phone list gives a better distribution for the last two hashes described above, with the ELF hash being slightly better. But there are still hash values which are generated from two or more names. This is pretty awful for us since we want a magic hash function that will generate a different index for each of our strings so that when we add string to the list or table it won't clash (or collide) with a string that's already there.

Just for fun I ran the same experiment with a hash function that just returned a random value (Hash 5). Of course this would be completely useless in practice, but it serves as a comparison. The results of the five hash functions are plotted in Table 1. The way to read this table is this: let's take the column headed '2'. This column shows the number of hash values that were generated by two strings in our set. So, for example if we look down that column at Hash 3, we see that there were 7 hash values that were generated from two different strings in our set.

Hawaiian Chance

Let's take a little side step into probability theory. Imagine a dartboard where the bull's eye and double and triple rings don't exist. In other words, a circle divided into 20 equal pie slices. I throw 15 darts at the board. With my dart throwing 'skills', it'll be pretty random where these darts stick in. Now there will be slices on the board where there won't be a dart, there will be slices where there is one dart and there may be slices with more than one dart. There may even be a slice with three darts.

It can be shown that the distribution of darts in the board (the number of slices with no darts, the number with one, etc) follows a Poisson distribution. The Poisson distribution curve is really a set of discrete points, each point representing the probabilities of no events, one event, two events, and so on. In general, the curve is humped towards the y-axis, with the probabilities of 0, 1 and maybe 2 events being significant, and the probability of x events where x increases from 3 getting asymptotically close to the x-axis. Figure 1 shows a Poisson distribution with mean 1.

Fascinating, I'm sure, but what does all this have to do with hash tables? Well, as I've already said the better hash functions have a randomizing aspect to them. The strings we use to generate hash values are like the darts, and the hash values are like the slices in the dartboard. The distribution of hash values from our set of strings follows a Poisson distribution. In Table 1 the row labeled *Expected* is the set of expected values according to probability theory.

In fact, the expected number of hash values that will be generated by x different strings is given by the formula

$$\text{Expected Number} = N\frac{(r/N)^x e^{-(r/N)}}{x!}$$

where r is the number of strings we are trying to insert in a list of N elements. For example, plugging r=33 and N=53 in to this formula, we expect that there is going to be one hash value (rounded from 1.4) that will be generated by 3 different strings.

Crash Dance

The outcome of this little detour is that no matter how clever we are in devising a stupendously complex hash function, there are likely to be some hash values that are generated by two or more strings. And, in turn, that means we will have to come up with a scheme to cope with this situation.

When the same hash value is generated by two or more strings, it's known as a collision. For example, suppose we add Smith to our hash table. We calculate that the hash value for Smith is 42 using our whizzomatic hash function and so we put 'Smith' and his associated value into the 42nd element of our list. If we want to find Smith again, we calculate the hash (42) and go and get the 42nd element from our list. Piece of cake.

Now we want to add Jones to our hash table. We calculate the hash for Jones and it also comes out as 42. We go to element 42 of our list and note that it is already filled with information for Smith. What do we do now? Where can we put Jones and his associated information? Obviously we can't replace element 42, otherwise we would never find Smith again.

How How

There are several options open to us. The simplest is called linear probing, sometimes known as progressive overflow). The way this works is as follows. Continuing our recent example, we note that element 42 is taken. We go and have a look at element 43 instead. If that is not taken we put Jones there. If it is taken, we continue this process with element 44, 45 and so on (wrapping round at the end of the table if necessary) until we do find an empty slot. Now, let's consider a search for Jones and his information. We hash Jones to get 42, and go get element 42 from our list. It's for Smith, so we reject it and go and get element 43. It's for Jones and we've completed our search. Here's the table around that part at this point:

```
Element 41: <empty>
Element 42: Smith
Element 43: Jones
Element 44: <empty>
```

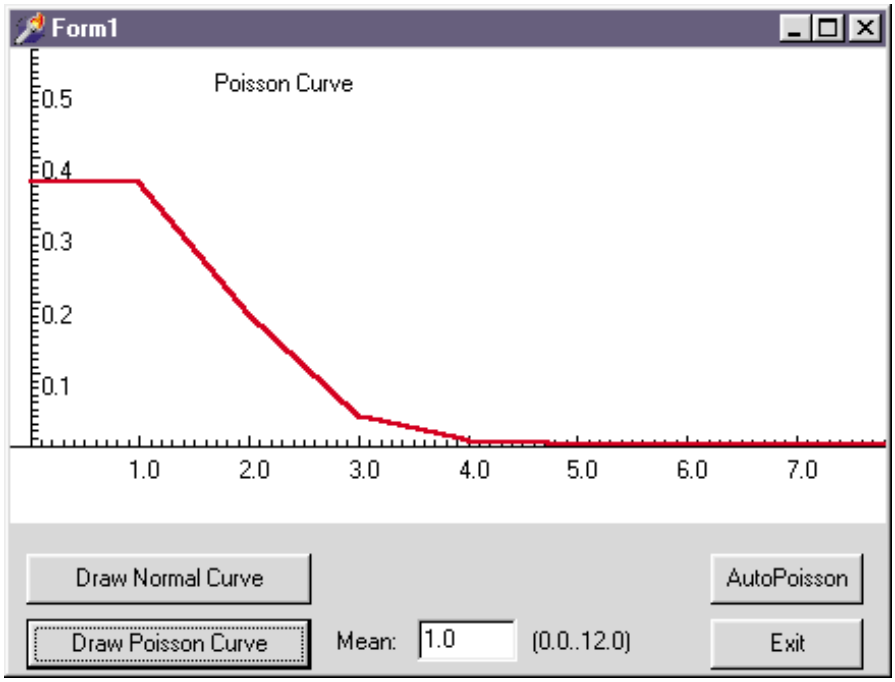
What happens if we have to find the element for Rhys, who isn't in our table? We hash Rhys to get 42 (oh no, not again!). We go get element 42, it's Smith. Ignore it and go get element 43. It's for Jones. Ignore it and go get element 44, which is empty, and from this we can determine that Rhys is not in our hash table. Pretty easy, huh?

A couple of notes about linear probing might be worth mentioning here before we continue. The maximum number of strings we can insert into a hash table using this method to resolve collisions is obviously equal to the number of items in the hash table. If we have a table of 53 elements, we can't add 54 strings to it. Period. Unless we code a table that dynamically expands, that is, and we'll get to this in a later installment.

The next thing to realize is that linear probing starts to degenerate the greater the loading of the table. Suppose that we've added 52 strings to our 53 element hash table and we're just about to add the last one. We could be extremely unlucky and get a collision and have to retrieve all 52 elements and reject them until we reached the empty slot. In fact research has shown that if the hash table is 66% or less full the average number of accesses for finding a particular string is 2 or less. In other words, if our hash table of 53 elements is filled with our 33 strings (about 62% full), and we try and find every single string we've added, we'll make in the region of 66 accesses to find all 33 strings. If the hash table were 80% full, we'd make on average 3 accesses for each string. At 87% the average rises to 4, at 90% it's now 5. So if we're going to use linear probing as our collision solution, it makes sense to keep our hash tables two thirds or less full (to put it another way our hash tables should always be one third or more empty if they use linear probing as a collision resolution mechanism).

Also it doesn't matter how bad our hash function is, we'll still be able to insert 53 strings into the table. But at what cost? Imagine Hash 1 above being used to insert the TurboPower phone list into a 53-element hash table. You can see that there will be many, many collisions in trying to populate the hash table (for example, there will be nine strings that generate the same hash value, ie, there will be at least 8 collisions at one point in the table). So we do need to use a good hash function and by doing so we reduce the number of collisions.

➤ *Figure 1*



By reducing the number of collisions we improve the efficiency of the hash table; we know that there will probably be some collisions, we just need to reduce the total number.

Another point which hasn't really been mentioned before is that for linear probing to work well, the hash function must give a good spread of values. If there is a 'clumping' of values around some particular hash value then any collisions in that clump will cause long chains of 'get and reject' operations in implementing the linear probe. For example, suppose we've added seven strings and in doing so the hash values 13-19 have all been taken. If we now add another string and it collides at 13, then the linear probe will have to get and reject 7 elements until it reaches an empty element. This aspect of hash functions is extremely hard to predict.

Bananas To The Beat

Having warned you sufficiently about some of the obvious and not so obvious problems of linear probing, there is one more. What happens if we want to delete a string from our hash table? Seems pretty easy at first glance: hash the string, find the actual element in the list (maybe doing a couple of probes in the process) and then mark that element empty.

Bzzzzt! Let's illustrate the problem. Say we have an empty hash table and we insert Smith, Jones and Rhys into the table in that order. They all hash to 42 and hence the table looks like this in that region:

```
Element 41: <empty>
Element 42: Smith
Element 43: Jones
Element 44: Rhys
Element 45: <empty>
```

Now we delete Jones and mark his element as empty:

```
Element 41: <empty>
Element 42: Smith
Element 43: <empty>
Element 44: Rhys
Element 45: <empty>
```

Now try and find Rhys. Rhys hashes to 42. Get element 42: it's Smith, so reject it and move to element 43. It's empty and so we conclude that Rhys is not in the table. What happened here is that we broke the linear probe chain that led to Rhys. If we delete a string from the list we cannot mark its element as empty, we must instead mark it as 'deleted': it may be that that element is forming part of a linear probe chain that we don't know anything about. Let's do the same experiment of deleting Jones again and this time we mark the element as deleted.

```
Element 41: <empty>
Element 42: Smith
Element 43: <deleted>
Element 44: Rhys
Element 45: <empty>
```

Find Rhys again. Rhys hashes to 42. Get element 42: it's Smith, so reject it and move to element 43. It's marked as deleted, so reject it and move onto element 44. It's Rhys and we're done.

Using this scheme, insertion becomes a little more complicated. Let's reinsert Jones into the table. It hashes to 42, get element 42. This is for Smith so reject it. Get element 43. It's deleted and so we could reuse it. But, we have not yet shown that Jones is definitely not in the table: Jones could be somewhere else along the chain we're tracing. So, we need to follow the linear probe chain until we get to an empty element without finding Jones. Once we do that we are allowed to insert Jones into the deleted element. So get element 44 (Rhys) reject, get element 45 (empty): Jones is not present, hence we can reuse the deleted element. It must be noted that we could use the empty element at number 45 instead of the deleted element at 43, but remember we always need to keep our linear probe chains as small as we can.

On the disk, you'll find a hash table class (`ThtHashTableLinear` in HASHTBL1.PAS) that uses linear probing to resolve collisions. There are methods to insert, delete and find a string, and to empty the table. Included is a debug print routine that enables you to investigate the different problems you'll encounter since it shows a list of the items in the table and also prints out the efficiency of the table as an average seek path value. There's a small example program that puts the hash table through its paces.

You Gotta Say Yes To Another Excess

Next time, we'll move onto other collision resolution methods for hash tables and work our way towards developing a hash table on disk as a very fast index to a set of records. We'll also be looking at how to enlarge a hash table, the so-called dynamic hash tables. If you do have any queries about this article or subsequent ones (or, horror of horrors, I'm wrong somewhere), please email me.

Until then, Pinball Cha Cha!

Julian Bucknall works for TurboPower Software. In between that and home, he spends too much time listening to CDs in his car (which isn't yello). He can be reached by e-mail at julianb@turbopower.com or on CompuServe at 100116,1572. The code that accompanies this article is freeware and can be used as is in your own applications.

Copyright © 1998 Julian M Bucknall

Looking for the latest software development news?

Visit the News section of the Developers Review website at www.itecuk.com