Hang Onto Yourself

Dynamic hash tables and how to resolve collisions...

by Julian Bucknall

In the last instalment (February 1998) we finished off with a class that encapsulated a hash table. This hash table used linear probing to resolve collisions. One of the problems with this implementation was that the hash table was a static size: it could only accept so many strings. Another problem was that the efficiency of linear probing started to degenerate badly after the table was about two-thirds full. This time around we'll be investigating hash tables that can dynamically extend or shrink according to the number of strings stored in them and we'll be looking at other collision resolution techniques.

I would advise you to review the material from the previous article before we move on.

Panic In Detroit

Actually, there was one thing I forgot to make really clear last time. Hash tables are great for finding a particular string and its associated data quickly. Providing you have a linear probe hash table that is less than two-thirds full, it takes maybe one or two accesses to get the string you want. It is *hopeless* at retrieving strings in alphabetic order: it just cannot be done. If you need to be able to access the strings in any other order than the order in which they're stored then use another data structure (a `TStringList` being the obvious choice). But if you want to either find a particular string in the table or show that it's not there, a hash table is the best.

Jump They Say

Well, let's make the hash table we've already written dynamic in size, or extensible. It's pretty easy really. When we insert a new string, we add it as before and then check to see if the table is now more than two-thirds full. If it is, we calculate a new size of the table that is roughly double the existing size (we have to make sure that the size is a prime, remember). We allocate a new table of this size. We then transfer every single string and pointer combination from the old table to the new one and then free the old table. The hash for each string has to be calculated according to the new number of strings, obviously. We are left with a table that is double the size of the original and roughly one-third full.

What about deleting strings? Well we could shrink the table once the table is less than one-third full, for example, but personally I don't think it's particularly worth it. Consider what happens if we insert and delete elements right around the boundary: we'd be continually flipping from a large table to a small one and back again. Also hash tables work best if there is only one access to get the correct element, and with small numbers of elements there is a greater probability of that.

I won't show you the code that extends the hash table here, it's really pretty obvious and if you want to you can look at the `ThtlHashTableLinear.htlGrowTable` method in the HASHLINP.PAS file on this month's diskette. This file contains the final, extensible, version of the linear probe hash table class which we introduced last time. It also supports table shrinking if the number of elements falls below one-sixth of the total number.

Before considering other collision resolution methods, let's review a frequently used variant of linear probing. This variant is called double hashing. With double hashing, instead of moving on by one element when we get a collision, we hash the string with another completely different hash function and then advance a number of elements given by that hash value. Generally, strings that hash equal with the first function won't with the second. Hence the elements with the same hash value will no longer be clumped together as with linear probing, they will be well separated. The separation is determined by another hash function, independent from the first. I leave it as an easy exercise for the reader to modify the linear probe hash table class to use double hashing instead.

Space Oddity

We have been blithely assuming so far that each element is only one string and its associated data. What happens if each element is a fixed array of strings instead, say 10 items long? What happens to the efficiency of the hash table then? Why would we do such a thing anyway?

Elements that can store several strings are called buckets. Let's answer the last question first and provide a signpost to where these articles are leading. The reasoning behind buckets is that you can retrieve a bucket of several strings in one access. For in-memory hash tables that's not particularly important. For hash tables that are on disk, it's of vital importance. This is where we are leading: we're going to be building a hash indexed record manager. Eat your heart out, Steve Troxell!

OK, back to our hypothetical array buckets. Suppose we have a linear probe hash table that has 30 elements filled of a possible 50, ie 60% full. The average search path is going to be a little less than 2 accesses as we discussed last time. Let us now assume that we have a 25-element hash table where each element is a 2-item array bucket and fill it with the same 30 strings. What's happened? Well, it's still 60% full, that's for sure. What's the average

search path now? Without a lot more mathematics than we really need to go into, we cannot calculate it from first principles (indeed, I deliberately skirted the mathematics last time). So we shall calculate it by simulation. I wrote a variant of the linear probe hash table that had buckets holding two items and the answer was 1.18 seeks per record, compared with 1.82 seeks for the simple case. In fact it was even better than this: my bucketed hash table had 23 elements whereas the simple case had 53, less than half the elements. Pretty good, and that was just with 2 item buckets.

Another way of bucketing is to use a linked list. Every string that hashes to a particular value gets inserted into a linked list at that value. The hash table then becomes an array of linked lists, each linked list containing nodes that have a string and its data, and each of those linked lists' strings hashing to the same value. This is of benefit for memory hash tables, since it is very easy to extend this data structure to contain more and more strings without growing the hash table itself. As you can imagine there comes a point when there are so many strings in the structure that too much time is being spent moving up and down the linked lists. At that point it's probably better to grow the hash table and shorten the linked lists. The difficulty is knowing when to do this and grow the table. I know of no research into this area (I haven't really looked though) and have not had time to experiment myself. By the way, forgive the gratuitous plug, but this is the method used by the string dictionary class in TurboPower's SysTools product. This particular hash table implementation has a most simple efficiency improvement: when a string is accessed it is moved to the front of its linked list. This means that the most 'popular' strings are close together and you don't generally have to go searching though long linked lists.

In fact, that last thought brings up another point. If you know, by experiment, by collecting statistics, or whatever, that certain strings are searched for in your hash table more often than others, then it is always better to insert them first. That way, those popular strings will be found with one seek and the less popular strings will have to be found with more than one seek. Overall, however, the effect is that the efficiency of the table is much greater. For example, if I was writing a program that maintained statistics on how many messages each TurboPower employee posted on our newsgroups, and I was using a hash table to index the names of the employees, I would make sure that our Tech Support Supervisor, Brian Warner, was the first name to hash and insert into the table. He posts far more messages than anyone else on our newsgroups (I was third if I correctly remember some recent statistics) and so my theoretical program would be accessing his record more often than anyone else's.

Of course, generally you don't know the relative importance of each string that you are hashing. I'm thinking here especially of compilers that insert identifiers from program source code into a hash table. I suppose that Borland could do some statistics on what identifier names Delphi programmers tend to use (I always seem to use a loop counter called `i` for example, and people use `Insert`, `Clear`, `Delete`, `Remove` and `Count` a lot) and they could pre-seed a hash table with these strings in the setup phase of compiling a program. This might slightly improve Delphi's compile speed.

Bang Bang

Well, I think we've taken in memory hash tables as far as we need to at present. Let's now

➤ *Listing 1*

```
type
  TMyRecord = packed record
    Name      : string[99]
    Data      : TMyData;
    IsDeleted : boolean;
  end;
  TMyFile = file of TMyRecord;
var
  MyRecord : TMyRecord;
  MyFile   : TMyFile;
```

➤ *Listing 2*

```
procedure GetMyRecord(const aMyFile : TMyFile;
  aRecNo : integer; var aMyRec : TMyRecord);
begin
  System.Seek(aMyFile, aRecNo);
  System.Read(aMyFile, aMyRec);
end;

...
GetMyRecord(MyFile, 0, MyRecord);  {get the first record}
```

consider hash tables on disk. For our initial discussion, let us assume that each string and its data are captured in a fixed length record in a data file and hence that file can be considered as a file of that record type (Listing 1). Pretty simple, so far, but note that each record has a flag that tells you whether the record is deleted or not. Each of the records in the file will have a record number, with the first record being record 0 and the second record 1 and so on. To access a given record we could write the code shown in Listing 2.

Again, nothing too contentious, apart from the missing error handling, I suppose. Similarly we could write routines that wrote a record to a particular record number, added a new record onto the end of the file, opened and closed the file and so on. We'll touch on deleting a record in a minute.

So, fairly simply and quickly, we have come up with a set of routines for managing a data file of records. Now we really need to index them, but instead of writing and using a B-tree or something similar, we recognize that we will just be accessing the records in the manner of a hash table. In other words we'll be reading a record based on a key (the `Name` field) or we'll be testing to see if the record with that key is there. We won't be attempting to read through the records in alphabetic order by `Name`, for example. And it must be fast, one seek or two to get a record, so that lets out the "let's read all the records in sequence looking for the one we want" answer.

So, fairly obviously (especially if you have been following this as we go along!), we need a hash table of some sort. For simplicity's sake, let's assume to begin with what we shall be using: a static linear probe hash table, thus the hash table is a certain number of elements large (prime, remember). In our in memory hash table, the elements of the hash table themselves contained the strings and their data, but we have no need of this in our file-based case since the strings and data are in our records in our data file. So what should the hash table elements contain? Why the record number of course! The hash table can be written quickly to another file, called the index file, and read from it into memory equally as easily.

Let's see how far we get with this definition just by talking our way through it. We want to insert Smith and his data into our data/index files. We create another record in the data file and retain its record number. We then hash `Smith` to get 42. We look at element 42 in the hash table and see that it's empty. But, hang on a minute, *how* do we know it's empty? We don't: each element is defined to be a record number and we haven't got any flags to say empty or in use or deleted. OK, here's Plan A: let's arbitrarily say that record number -1 (which cannot exist in the data file) means 'empty' and record number -2 means 'deleted'. This does mean that when the hash table is created (and initially written to disk), all elements must be set to -1. Moving on, we can see that element 42 is empty and so we write Smith's record number (ie 0 as he's the first) in there. In short order, we add Jones (who also hashes to 42) and Rhys (who also hashes to 42). The hash table then looks like this around this area:

```
Element 41: -1 <empty>
Element 42: record number 0 (Smith)
Element 43: record number 1 (Jones)
Element 44: record number 2 (Rhys)
Element 45: -1 <empty>
```

Let's now find the record for Rhys. We hash it to 42. This element has 0 as the record number. We retrieve that record and look at its `Name` field; it's not the right one. Back to the hash table and we look at element 43. This points to record number 1 which, after we've read it, is also wrong. Back again to the hash table, element 44 says record 2 and once we've read it we see that it's the correct one. As you can see this all translates pretty easily from what we had before in the in-memory case. Let's say we delete Jones. We find out from element 43 that it's record number 1. We mark the record as deleted (there's a flag for that, remember) and we mark element 43 in the hash table as record number -2. The hash table then looks like this:

```
Element 41: -1 <empty>
Element 42: record number 0 (Smith)
Element 43: -2 <deleted>
Element 44: record number 2 (Rhys)
Element 45: -1 <empty>
```

All pretty familiar. If we now need to add Jones again, we create another record (record 3 this

time) and then the new record number gets set in element 43 in the manner to which we've all become accustomed (and which we talked about at length in the last article).

So, it's all wrapped up then? Nope, 'fraid not, as I'm sure this pretty simple walkthrough has shown, by revealing the flaw you have already spotted. Record number 1 has not been reused, it's just lying there taking up disk space. If we do a lot of record deletions then we are going to get a lot of deleted records in our data file that would never get reused. Unless, of course, we took the time and trouble to read through the entire data file looking for deleted records to reuse when we inserted a new one. And that is exactly why we don't do it, it would take forever and a day and defeats the purpose of the hash table in the first place.

So although Plan A has the benefit of simplicity, for the type of record manager we want to write, it's not very good. Plan B then!

Ricochet, Plan B

To be honest, Plan B isn't all that different. Here it is: each element in the hash table has a record number and a flag that details whether the element is empty, in use, or deleted. A moment's thought would show that this neatly solves the deleted record problem: when we delete a record we mark its element in the hash table as deleted, but we leave the record number in place. Our hash table then tracks our deleted records for us! Wonderful stuff. Back to the reinsertion of Jones, the operation we were trying to do before that revealed the problem. The hash table looks like this:

```
Element 41: <empty>
Element 42: <in use>  record 0 (Smith)
Element 43: <deleted> record 1
Element 44: <in use>  record 2 (Rhys)
Element 45: <empty>
```

As you can see, walking through the insertion of Jones (which hashes to 42, remember) using the methodology we discussed in the last article, we not only reuse element 43 for Jones, but we also reuse record number 1. Marvelous.

The only problem with this scenario is that it uses a total of 5 bytes per element. Which doesn't seem like much to worry about, but when you factor in the fact that Pentiums like variables aligned on 4-byte boundaries (it's much more speed efficient) then you see that each element would be better as 8 bytes (with 3 wasted for our goal of speed). Is there a Plan C, where we don't use 8 bytes per element? Well, how does 2 bits per element grab you? (Yes, that's bits, not bytes.) A 32-fold space improvement is not to be sneezed at.

Shining Star, Plan C

Imagine that you have been using Plan B for a while with a particular file, adding and deleting many different records (not that I recommend doing this to a hash table with no buckets, by the way: why?). What size is the data file going to be at this stage? Pretty obviously, it will have the same number of records as there are elements in the hash table. Some of these records will be deleted, yes, but there probably won't be any empty slots in the hash table any more, they'll either be in use or deleted, and in either case there will be a record number associated with each hash table element.

So what? Well, if there are as many records as hash table elements, then why bother with record numbers in the hash table at all? Force element 0 of the hash table to be record number 0 in the data file (or vice versa). Then each element of the hash table just has to contain a flag for whether the record is empty/in use/deleted. What that also means in practice is that you have to write blank records to the data file when you create the hash table. So if you create the data/index files to contain a maximum of 509 records, not only do you have to create a 509 element hash table, you also have to write out 509 blank records to the data file.

To insert a record in this situation, you move operations around a little bit. Let's insert Smith, Jones and Rhys again, in that order. Hash Smith to 42, get element 42 which indicates that the record is empty. Write Smith to record number 42, mark the table element as 'in use'. Now Jones: hash to 42, element 42 says 'in use', but retrieving the record shows that it's the wrong one; element 43 is empty so you write Jones to record number 43 and mark the table element as 'in use'. Similarly for Rhys.

And remember throughout that all the elements in the hash table just contain a flag detailing whether the associated record is empty, in use or deleted. That's three possible values, and two bits is ample for that (you can store four possible values in two bits). So the hash table can be mightily compressed to four elements per byte. A 509 element hash table could be compressed to 128 bytes using this scheme.

Of course you don't have to write out all those blank records either if you don't want to. Win32 operating systems allow you to seek beyond the end of a file and write data there: it'll do it quite happily and fill the intervening region with whatever garbage bytes happened to be on disk. If you don't want this to happen, then you will have to write out the intervening blank records. Just don't try doing this with Windows 3.x!

Oh, and the answer to my quickie question above is that, in the scenario described, there would be no empty elements in the hash table. Remember that to insert a new record you would hash the string key, and then read through the elements in the hash table (and reading records off disk to compare strings) looking for either the key you want or the first empty slot. Since there are no empty slots, it would mean that every single 'in use' record would have to be examined every single time you wanted to insert a record. In fact, this is a real problem with linear probing in general: if the data within the hash table is highly dynamic, it makes sense to periodically create a new hash table and copy the data over.

Miracle Goodnight

Are we done? Gimme a break. Another speed improvement we discussed earlier was buckets. In a disk environment, buckets are a good thing. They are even better if the bucket size is equal to the cluster size of your disk, for example 4Kb, 8Kb, 16Kb or 32Kb.

Recall that when you read from a file your disk driver will in fact read an entire cluster at once and just parcel out the bit you want. Since the driver is going to read a cluster anyway, let's pack as many records as we can into a cluster and make a cluster a bucket. This gives us more room in the hash table and improves on the number of seeks to find a record. We can easily read a cluster, sorry, bucket, from disk and search through the bucket to see whether the record we want is there. It does make our data file a file of buckets rather than a file of records, but that's a small price to pay for improved access.

The hash table index we have just been theorizing about is static: its size is determined by whatever number of elements it was created with. We need to make that step further and create a dynamic hash table index on disk.

A moment's thought would show that Plan C above would be impracticable. Every time we grew the hash table index we would have to create a new data file, since the number of the hash table element would also be the record number and elements would be rearranged. We'd rather not have to rebuild the data file, just in order to expand the index.

What to do then? We could use Plan B and every time we grew the hash index we have to completely rewrite it. Possible, but in a shared environment it's less than optimal; however, it is a minor flaw. If we use buckets with Plan B then we're stuck with having to rebuild the data file every time the hash table index needs extending (the reason is that most if not all of the records in the bucket will have the same hash value, and when we change the hash function by expanding the hash table the records in the buckets no longer have the same hash value as each other).

The answer is to use extensible hashing, or linear hashing, or any of the variants thereof. But any discussion on extensible disk hashing would require a starter article at least as long as this one. I'm afraid I can't just polish it off in a couple of paragraphs, and so it must wait until another time. If you want to know about extensible hashing, by all means let either myself or our Editor, Chris Frizelle, know and we'll cover it in a future issue.

Outside

However, all is not quite over yet. The code on the diskette is an implementation of a hash indexed record manager that uses a variant of Plan B with 4Kb buckets to store records. I've even made it an extensible one as well, although to extend such a table essentially requires the data file to be copied over into a new one. So, try and choose your hash table size properly so that no extension is required.

[Editor's note: As usual, Julian is being very modest about his accomplishments! I strongly recommend you take a look at the hash indexed record manager on the disk. It's ideal for many of those occasions where you thought "Rats, I suppose I'll have to use some database engine or other now for this little bit of my application" and began to despair of all the extra hassle and all that extra code being linked into your app, or those massive DLLs...]

As usual with freeware code from my articles, you are free to use it in your own programs, but I retain all copyright in it.

Julian Bucknall works for TurboPower Software. He acts in his spare time (a Cracked Actor? – a lad insane) and listens to CDs whilst programming. 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.