Dictionaries and hash tables

Speed up keyed access to data using hash functions to search an array

In this issue...

▸ WHAT'S COVERED

We're all familiar with dictionaries. Given a word, we can look it up and read its definition. With a little judicious rewriting of 'key' for word and 'record' for definition, we get one of the most fundamental and useful data structures in computer science: the dictionary, or hash table.

The array is a common structure used in many programming languages. This structure has been around since the very early days of computing for the simple reason that – at the machine level – it's both quick and easy to access individual elements of the structure.

Consider this: all of the elements are held in contiguous memory, so there's no hunting and pecking through vast empty spaces. If the start of the array is at memory address X, for example, then the first element is at X. Assuming that the size of individual elements is Y bytes, the second element is at X+Y, the third at X+2Y, the fourth at X+3Y and so on.

Since, in general, elements in an array are counted from 0 (so called zero-based arrays) to read element n, all the run-time library has to do is make a quick calculation to find the address – X+nY – and go there. The process is as instantaneous as possible.

In this sense, the array works like a very simple dictionary: given a number (usually known as the index), you can immediately go to the right place in the structure and read the data associated with that number.

But what we'd like more is a method that given a string (or to be most general, a key) would be able to go to the right place in the structure and read the data associated with the string or key. (The key is generally called a value, hence the common phrase 'key-value pair'.) So, if we were going to emulate an everyday dictionary, for example, we'd like a structure that given the word 'PCPlus' would return the definition: an eminent magazine that covers computer hardware and software.

At first blush, such a structure could just be an array of records with two fields – the word and the definition – and the structure would look up a given word by searching through the entire array, comparing keys one after the other. However, this would be an extremely slow process, with the time taken proportional to the number of elements in the array.

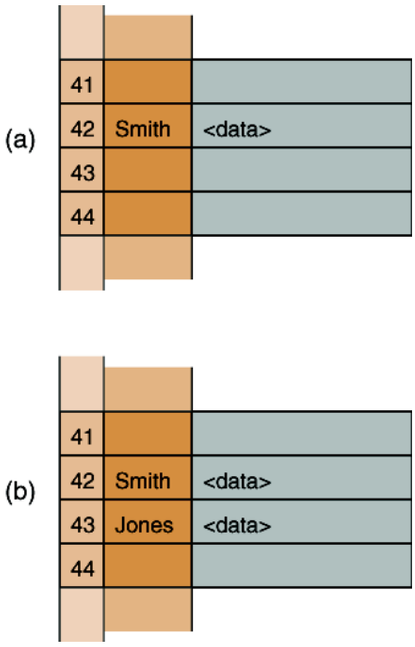
Another possibility would be to pre-sort the array of records in key order (just as a real-life dictionary does). Using this structure in conjunction with binary search, we could find the record for a key in log(n) time (where n is the number of records in the array). This is a much better proposition: for an array of a million records, the binary search would only have to compare about 20 keys to find a record.

However, what we'd like most of all is some way of mapping a key to an index in the array. That way, we'd be able to use the address calculation described above to get the record.

This mapping is known as hashing the key. It involves chopping the key up in some manner and using what's known as a hash function to obtain an index value. We can then use this value to directly index the key into an array. This will result in a type of dictionary, which in computing terms is known as a hash table.

Unfortunately, the hash function is not the only algorithm you need in order to create a dictionary. The problem is that hash functions are never perfect, and often we'll find that two different keys will map to the same index value. ▸

> "The problem is that hash functions are never perfect, and often two different keys will map to the same index value"



▲ **Figure 1: When keys hash to the same value, one will be placed in the next available empty slot.**

This means that the second algorithm we would need to consider is what we do when this happens. For obvious reasons, two or more keys mapping to the same index is called a collision. The second algorithm is therefore known as collision resolution.

So to recap, a dictionary is a pre-set array of n possible key-value pairs (initially empty) associated with a hash function (that given a key will return a value between 0 and n-1) and an algorithm to resolve possible collisions. The default operations on the dictionary are 'Insert' and 'Find'. 'Delete' may be used later, although usage of most dictionaries tends to be centred more around inserting key-value pairs and then using the dictionary to find values given a key than deleting those key-value pairs.

Hash functions

The first thing to do is write a hash function, and preferably one that – given the breadth of keys we may encounter – will dole out values that won't cause collisions.

Let's assume that our keys will be strings. The hash function will somehow convert the string to an integer value, and once we have that then we can use the modulus operator to force the value between 0 and n-1.

So how would we convert a string into an integer? One way might be to use the length of the string key. This has the advantage of being simple and fast but the huge disadvantage of generating numerous collisions. In our real-world dictionary example, words such as 'string' and 'simple' would convert to the same value as 'PCPlus': 6. In fact, the vast majority of words in a 50,000-word dictionary would be less than 20 characters long, giving a pretty bad spread of values.

No, we must use the data in the string key in some fashion. One idea is to take the numerical values of each letter in the string ('A' would be 65, 'a' would be 97 and so on) and sum them. In this scheme 'simple' would hash to 650 while 'PCPlus' would hash to 567. This isn't too bad, but words that are palindromes (toot, dad) or anagrams of each other (read and dare) would suffer from collisions.

Nevertheless, using the numerical values of each character has merit; we just have to weight these values with their position in the string itself. A very simple hash function that produces a good distribution of values from string keys is this: for each character, multiply the current sum by 17, add in the numerical value of the character and then set the resulting sum to the modulus of it and the dictionary size. 17 is a bit of a magic number (it's a prime number close to 16, the base of hex digits); combined with the fact that dictionaries tend to have a prime number of elements, it means that the final modulus division won't divide evenly.

Of course, all that multiplying and dividing could take a while. To combat this, various alterations have been made so that the algorithm uses logical operators instead.

If the key is not a string, it will still be accessible as a sequence of bytes and this simple hashing algorithm will still work well.

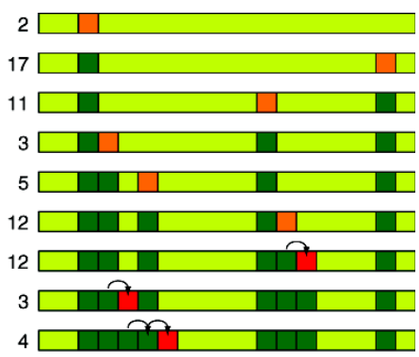
Collision resolution

Given that we're going to get collisions no matter what, we have to decide what to do. There are several algorithms available that enable us to store the key-value pairs in an array by using the empty slots in the table to store items that collide with those that are already present. This class of algorithms are known collectively as open-addressing schemes, and the simplest example is one called linear probing.

Let's explain by using a simple example. Suppose we are inserting surnames into a dictionary, and assume that we have a hash function of some pedigree or other. To start off, we insert the name Smith into the empty dictionary. We hash the key 'Smith' with our hash function and get the index value 42. We set element 42 of our pre-allocated array to Smith. The dictionary now looks like Figure 1(a) around this element.

That was pretty easy. Let's now insert the name Jones. We shall proceed as before: hash the key 'Jones' and then insert Jones at the resulting index. Unfortunately, our hash function is of dubious provenance and hashes Jones to the value 42 again. We go to the dictionary and notice that we have a collision: slot 42 is already taken up with Smith. So what should we do now?

With linear probing, we try the next slot to see if it's empty. It is, so we set element 43 of our dictionary to Jones. If 43 were taken, we would have a look at the next slot and so on, going back to the beginning if we reach the end of the array. Eventually we'd find an empty slot, or we'd get back to where we started and conclude that the array was full.



▲ **Figure 2: Clusters grow as you insert more items, which results in more frequent collisions.**

Spotlight on… Alternative probes

In the main article, we discussed linear probing as a solution to resolve collisions. Are there any other solutions – and if there are, are they better? The main issue with linear probing is its tendency to produce clusters. If on inserting a new key the hash value hit a cluster, the key-value pair becomes part of that cluster.

The first alternative is quadratic probing. Instead of looking at the next element on a collision, and then the one after that, we can follow a quadratic progression: one after, four after, nine after and so on. This would avoid most of the issues with clustering, but it also has an insoluble problem: we can't guarantee that all empty slots would be visited.

Better is a technique called double hashing, or rehashing. Here we have two hash functions. If there is a collision with the first hash function, we use the second to create a step value other than 1. Since the different keys are unlikely to produce the same hash with two functions, we neatly sidestep the linear clustering problem. Figure 3 shows the effects of applying a second hash function to the collisions from Figure 2.

Closed addressing

Open addressing attempts to place all key-value pairs in the array. Another way of resolving collisions is to create a list of key-value pairs for all of the elements in the array that hash to the same value. If another key comes along that hashes to the same value as one or more other keys, just add it to the end of the list (or at the front, or in sorted order) at that index. This type of closed addressing is also known as chaining. If the list is a linked list, you can obtain such optimisations as sorting the list by the key-value pairs that are most frequently entered or searched for. This would mean that more popular keys are found at the front of the list and require less search time.

> With linear probing, items tend to form clusters of occupied slots – and adding more items causes the clusters to grow in size

The act of checking a slot in the dictionary is called a probe, and we're probing for empty slots one after the other, hence the name of the algorithm: linear probing.

The dictionary now looks like Figure 1(b) around the area of interest. Having inserted two items in our hypothetical dictionary, let's see if we can find them again.

Let's try Smith first. We hash Smith to give an index of 42. We look at element 42 and find the Smith item right there. (Note that this requires a comparison of the names.) Now Jones. We hash Jones to give an index of 42. We look at element 42. It's the Smith item (a comparison), which isn't the one we want. What we then do is the same thing that we did when we were inserting: we visit the next element in the dictionary to see if it is ours. And as it happens, it is.

How about searching for an item that's not in the table? Let's search for 'Brown'. We hash with our hash function and get the index value 43. We visit element 43 and see that it's the item for Jones. We advance one step to element 44 and notice that it's empty. We conclude that Brown is not in the dictionary.

▲ **An array performs a similar function to a real dictionary, looking up a key to read its data.**

▲ **Figure 3: Use another hash function in conjunction with linear probing to avoid clustering problems.**

Problems with open addressing

In general, if there are few occupied slots in the array, we'd expect most searches – whether successful or unsuccessful – to take just one or two probes. Once the array gets fairly full, the number of empty slots would be very few, and so we'd start to expect unsuccessful searches to take many probes – even as many as n-1 probes if there was only one empty slot. In fact, if we're using an open-addressing scheme like linear probing, it makes sense to ensure that the array doesn't get overloaded. Our probing sequences would take a long time otherwise.

There are a couple of points about linear probing that are worth mentioning. The first thing is somewhat obvious: if there are n elements in an array, you can only insert n items (this is true for any open-addressing scheme). However, this theory is a little simplistic. To find out whether a key is present in the array or not, we need at least one empty slot so that the search will stop once the entire array has been searched. This means that the array can never be completely full.

The second point is the problem of clustering. If you use linear probing, you'll find that items tend to form clumps or clusters of occupied slots. Adding another item causes the clusters to grow in size. This is because as more items are added, it gets more and more likely that the inserted item will collide with an item that's in a cluster. And as collisions get more likely, the clusters will grow in size.

We can illustrate this with a simple figure. Figure 2 shows clusters growing in a small array, using random numbers for the hashes and linear probing to allocate slots. The orange squares are direct insertions and the red squares involve one or more collisions. At the end, we have two growing clusters.

Clusters affect both the average number of probes required to find an existing item (known as a hit) and also the average number of probes required to show that an item does not exist in the dictionary (known as a miss).

In fact, in his seminal work *The Art of Computer Programming*, Knuth derived some fairly simple expressions based on the load factor of a dictionary (the load factor is the proportion of occupied slots in the dictionary's array).

For a dictionary that is about half full, a hit requires about 1.5 probes, whereas a miss requires 2.5 probes on average. If the table is two-thirds full, a hit requires two probes and a miss requires five probes. If the table is 90 per cent full, a hit requires on average 5.5 probes, but a miss requires an amazing 55.5 probes.

As you can see, using linear probing as your dictionary's collision-resolution scheme requires you to keep the table at most two-thirds full. This should reduce the number of problems caused by clustering and hence increase your dictionary's efficiency. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk

Why use a dictionary?

The reason that dictionaries are so popular is because they are the ideal implementation for a build-and-lookup scenario. As we saw above, if the dictionary is nowhere near full, a hit takes one or two probes. This operation takes hardly any time at all to complete.

An example of a program that uses a dictionary all the time is a compiler. This program has to maintain information about the various identifiers it comes across as it's reading the code. It builds up a dictionary with the identifiers as keys and information such as type, usage and so on as the values. When the compiler comes across another reference to the type, it can look it up in its dictionary to find out about it.

Up until relatively recently, programming languages required the use of a library in order to gain the benefits of using a dictionary. For example, the dictionaries of both C# and Java are found in the frameworks that accompany them, not in the language itself.

Most of the recent dynamic languages – such as Perl, Python and Ruby – have dictionaries built into the language in the sense that there is special syntax for declaring and using a dictionary.

However, even though C# has no syntactic support for dictionaries, C#'s compiler is clever enough to be able to use them in the compiled code for certain code structures, such as the switch statement. ■