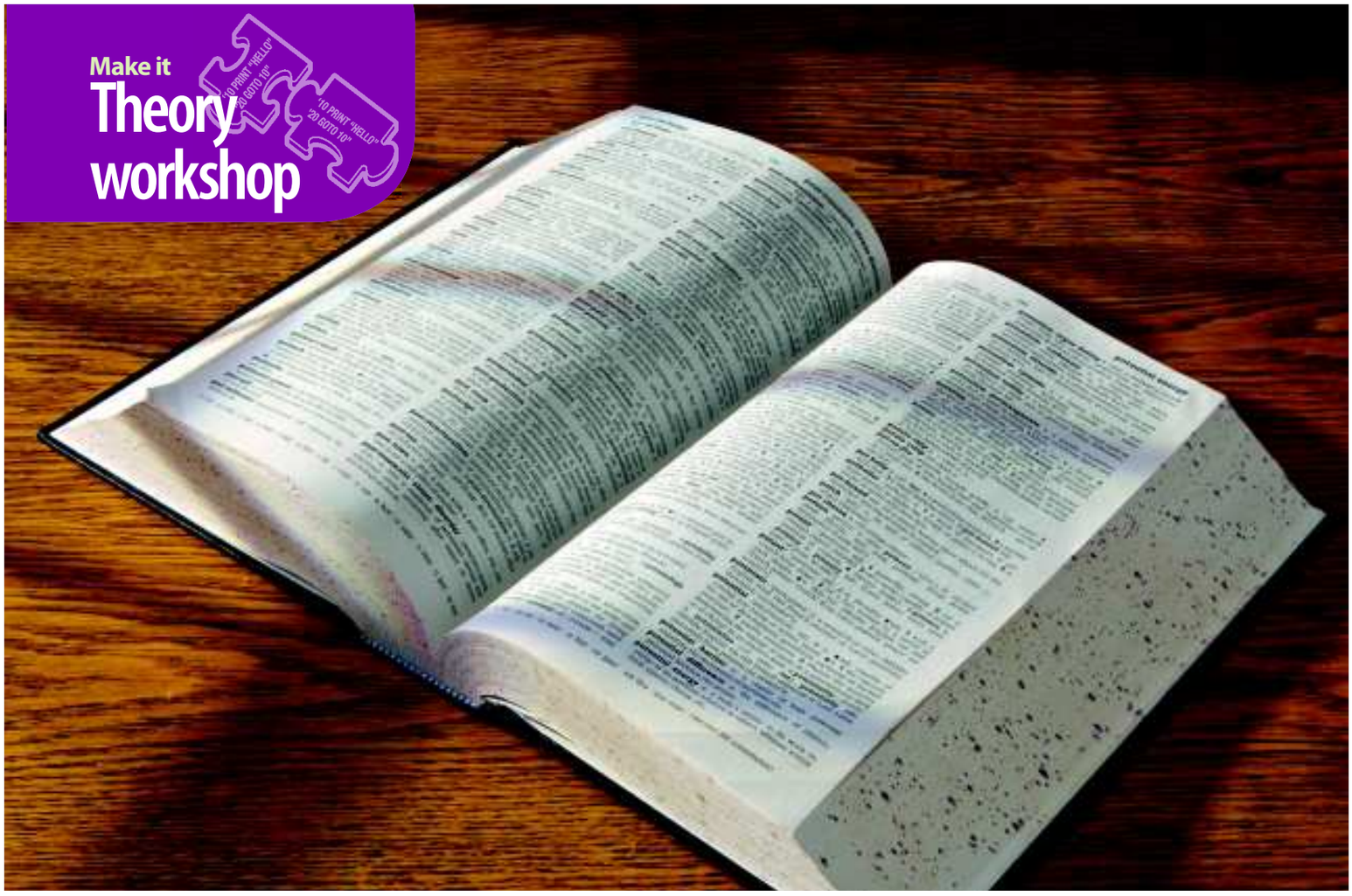


Building an efficient dictionary

How to categorise algorithms by their manner of execution

When we need to select an algorithm for a particular purpose, we should pay attention to its runtime characteristics: how fast it is; how much memory it uses; whether there's a worst case for the algorithm's execution speed; and so on. All these answers are expressed with the big-Oh notation, which I'll describe later.

A common abstract data structure that's used all the time in programming is the dictionary or associative array, which is sometimes known as a map. I call it an abstract structure because it can be implemented in myriad different ways, but it always has a specific interface. We'll use the dictionary to investigate the runtime efficiency of various algorithms that can be used to implement it.

But first, a definition: a dictionary is a structure that holds name-value pairs. A name-value pair is an object that has a name – that's used both to describe its value and as a key to find it – and a value, which can be anything at all. The classic example is a real-world dictionary, where the name is a word and its value is the word's definition. However, don't limit yourself to assuming the name is always some kind of text string. In reality, names can be integer values, bit strings, 128-bit GUIDs, dates or anything at all. That said, it's helpful to assume that they're text strings for now.

The dictionary has various operations that define its external interface. There's the 'Create' operation, which creates a new dictionary, and the 'Destroy' operation, which releases any resources the dictionary is using and destroys the structure. A dictionary can only be used after 'Create' has been called, and once 'Destroy' is executed, it no longer exists. Since these operations are only used once each per dictionary, they won't have much effect on the overall runtime and so we won't discuss them any further.

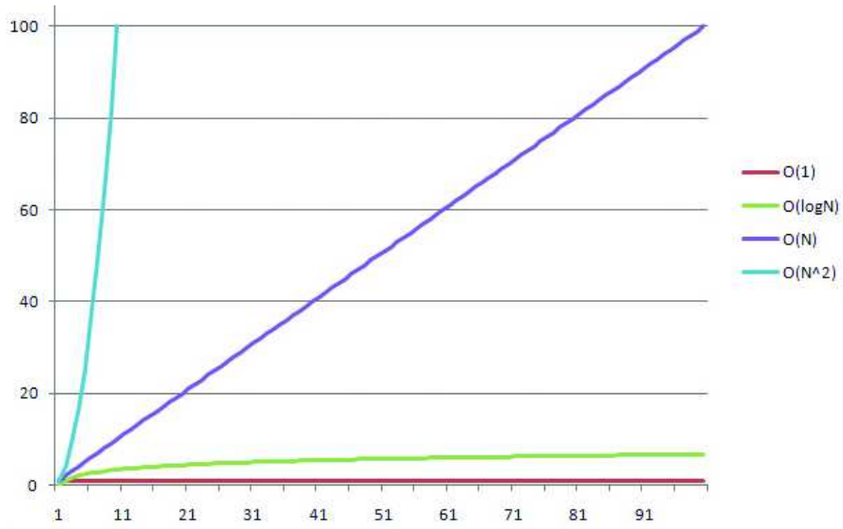
When given a name, 'Find' will search for the name-value pair that matches and return its value or an error if the name is not found. 'Exists' will do the same, except it will merely return true or false according to whether the name is present or not. Since they're virtually identical, apart from what they return, we'll ignore 'Find' from now on.

Finally we have 'Insert' and 'Delete', which do what you'd expect: add a new name-value pair to the dictionary (returning an error if the name already exists), and remove the name-value pair that matches a given name, respectively. In general, 'Delete' won't return an error if the name is not found, and sometimes 'Insert' will merely replace the value if the name already exists.

Now that we have our abstract data structure, let's investigate first how to implement it and second analyse the efficiency of our implementations. We'll look at a total of four implementations.

Name-value pairs

The first implementation is the most obvious: use an array of name-value pairs. 'Exists' is the first operation to think about. In essence, to see whether the given name is present, you would check every pair in the dictionary sequentially ▸



▲ **Figure 1: Graphing some common big-Oh expressions (O(N^2) is cut off so we can see the others).**

and stop when you found it. If the given name isn't present, you would compare the name of every name-value pair to the given name. The more pairs there are, the longer it would take, but you can be even more precise than that. Suppose there were N pairs in the dictionary and each comparison took the same (constant) length of time – say *t*. Then it would take *t*N time units to find out the given name wasn't present. Another way of putting this is that the time taken for the nonexistence check is proportional to N. In computer science, without going into too much rigorous mathematics, we say the runtime efficiency is O(N), pronounced 'big-Oh of N', although you can read it as 'is proportional to N'.

So if it took so many seconds to find out that a given name wasn't in a dictionary of 1,000 pairs, it would take twice as long for a dictionary of 2,000 pairs, and 10 times as long for a dictionary of 10,000 pairs.

What if the given name was in the dictionary? What could we say then? Well, it could be that the matching pair was the first item checked. In that scenario, we say the best case efficiency for 'Exists' is O(1), which you read as 'is constant' (in other words, it doesn't depend at all on the number of items in the dictionary). But, of course, for that to happen, you'd have to be extremely lucky. You could be completely unlucky and be looking for the final item. Here the worst case efficiency is O(N) – the time taken would be proportional to the number of items in the dictionary.

On average, though, if you searched for every name in the dictionary, the efficiency would be O(N/2). Now comes the fun bit with big-Oh notation: since it essentially means 'is proportional to', you can take the 1/2 (a constant) out of the parentheses into the implied proportionality constant and say that the efficiency is O(N). We say that searching through the dictionary-as-array is O(N): twice as many items, twice as long.

'Insert' is simple: we add the new name-value pair to the end of the array, a constant O(1) operation. Hold on there though – we first have to search the array to find out if the name is already present or not. 'Insert' then degenerates to O(N), just like 'Exists'. We get no benefits at all from the constant, quick, add-it-to-the-end operation; we still have to search.

'Delete', as I'm sure you can see, is at least O(N) as well – we have to do the search. There's something else about 'Delete' that we have to take into account: we have to physically remove the name-value pair from the array. The simplest way of doing this is to simply take the final pair in the array and put it in the slot vacated by the pair that was removed: a constant O(1) operation. So, overall, 'Delete' is O(N); the search time will swamp the move-an-item time.

Sorted pairs

Let's move on to the second implementation. This one is again an array, except this time we maintain the pairs in sorted order. This has the assumed requirement that the names are sortable and that, given any two unequal names, we can say that the first is smaller or greater than the second.

We'll start off by analysing 'Exists' again. The array is in sorted order, so we can use binary search to try and find the name-value pair that matches. With binary search, we look at the middle item in the array. If it's the one we want, we stop. If the one we want is less than this middle item, we know that, if it's present at all, it'll be in the first half of the array. If the one we want is greater than the middle item, we know it will be in the second half. We repeat this process with the half array we selected. We'll either find the item immediately again, or we'll have reduced the number of items we have to search to a quarter of the array. Ditto the next step, except we reduce the space we have to search to an eighth of the original array. And so on.

Again, consider the doesn't-exist case. Say we start out with an array with 1,023 items. After one step, we'll have discarded one item and will have identified a subarray of 511 items for the next step. After this next step, we'll have reduced the search space to 255 items, and so on. At the 10th step we'll have a tiny array of just one item, which we can easily compare. So all in all, we'll have made 10 comparisons to find out that the given name is not present. What's so special about 10? Well, it's the logarithm to base two of 1024 (that is, 2^10 = 1024). Again, without being too rigorous mathematically, we say 'Exists' is O(logN) when the name isn't present.

Think of O(logN) this way: if it takes a particular length of time to find out that a given name isn't present in a sorted array of 1,000 items, it will only take twice as long for an array of 1,000,000 items. If you square the number of items, you double the time taken. This is an extremely significant result, showing the importance of binary search.

Ternary trees

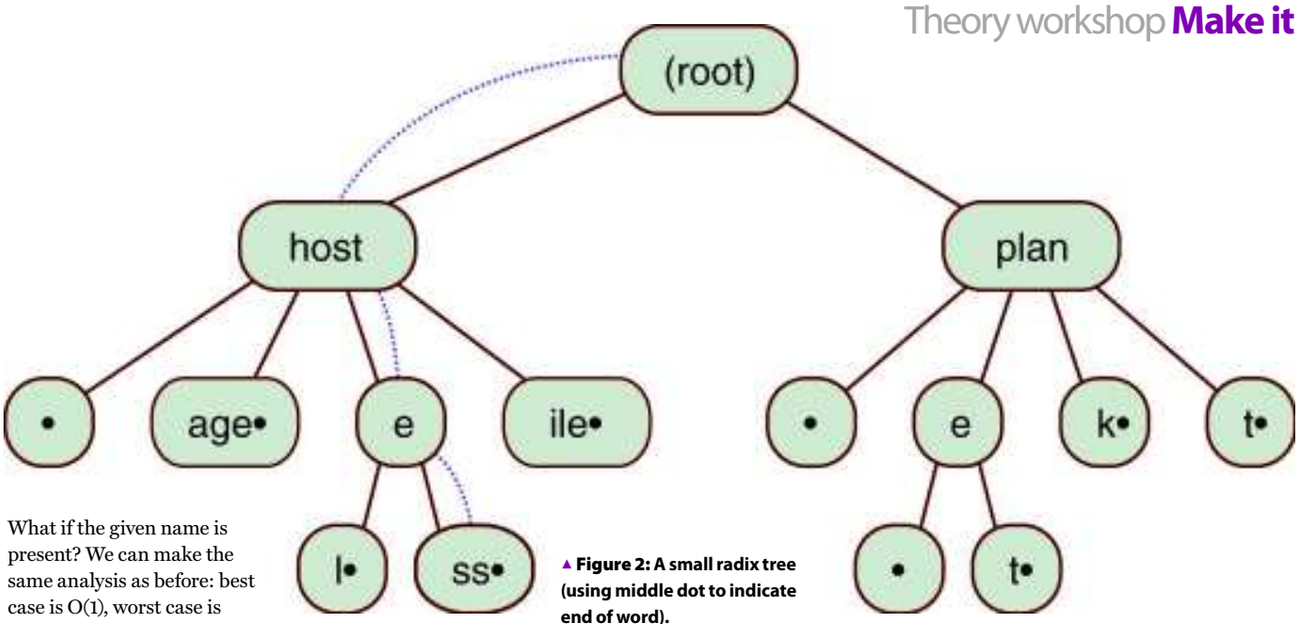
Back in issue 282, I cited ternary search trees as a strong candidate for the data structure behind a dictionary. Ternary trees, like radix trees, have a runtime efficiency that's dictated by the length of the keys rather than their number, but are much easier to implement. Ternary search trees and radix trees also have a further benefit: using them means you can easily produce a sorted list of names in the dictionary, as well as produce a prefix list (a list of names with a particular prefix). ■

Spotlight on… Radix trees

Radix trees offer a further data structure that can be used for a dictionary. A radix tree stores prefixes to keys rather than complete keys in its nodes, and each node can have many children. A key is then found as a complete path through the tree from root to leaf – at each step down the tree, you compare another small part of the name to the next node.

Figure 2 shows an example radix tree storing a small set of words. In searching for 'hostess', we follow the left link from the root, matching host, then follow the middle link matching the 'e' and finally matching the 'ss' in the right node.

Unlike the other data structures we've looked at, the efficiency of a radix tree doesn't depend on the number of name-value pairs, but instead on the length of the keys. All operations are essentially O(*k*), where *k* is the maximum name length in the radix tree. This can be greater than the balanced binary tree's O(logN), for example, but in practice we find that the comparisons needed in a binary tree are also significant, so the radix tree can be a viable alternative. ■



▲ **Figure 2: A small radix tree (using middle dot to indicate end of word).**

What if the given name is present? We can make the same analysis as before: best case is O(1), worst case is going to be the same as not finding it: O(logN), and so we say that, overall, 'Exists' is O(logN).

What about 'Insert' and 'Delete'? Again, we have to search for the name, so it would seem that they're both O(logN). But this time, consider what we must do to add (or remove) the name-value pair. For 'Insert', we have to make a hole in the array to put the new pair in, shuffling all the items greater than it along by one. For 'Delete', we have to shuffle the remaining pairs to close up the hole vacated by the removed pair. If we're lucky, in both cases, we don't have to move any items (that is, best case is O(1)); if we're unlucky we have to move all of the remaining pairs (that is, worst case is O(N)). On average, it's O(N) for all the shuffling we need to do. Since O(N) is bigger than O(logN) – for very large values of N the (in)efficiency of the moving of the items will swamp the efficiency of the search – we ignore the smaller proportionality and just use the larger one. We say 'Insert' and 'Delete' are both O(N).

Hash table

Now for the next implementation: the hash table. Without going into full detail, we have an array as the basic data structure.

Again, we analyse 'Exists' first. To find an item in a hash table, we hash the given name to produce an index into the array. The hash is produced by a randomising type function that takes the name, chops it up and combines the parts to produce an integer value. That integer value is then reduced to a possible array index value by use of the mod operator. The hash function is designed so that similar names produce very different hash values.

Best case is that 'Exists' is O(1). That is, we create the hash for the given name, convert it to an index, go to that element in the array, and the pair we need is there and matches. No matter how many items are in the array, that process is constant. (Actually, the hash function is usually O(k) where k is the length of the name, but we're ignoring that for now.)

What about worst case? Well, in practice we'll find that many names will hash to the same array index value. These are called collisions and we need to implement a collision resolution strategy to deal with them. The simplest is known as chaining, where we chain the name-value pairs as, say, a linked list at each array element. In this case, once we've calculated the index, we then do a sequential search through the chain at that index.

To ensure that the chain is never too long, hash tables grow themselves periodically when their load factor (the number of pairs present divided by the number of array elements) reaches a particular value. To do this, a new array is created, and all the pairs are rehashed and inserted into the new array. This ensures that chains never grow beyond a few items, say five or 10. Since this isn't dependent on the total number of items, it's still constant and we say 'Exists' in a hash table is O(1) on average.

'Insert' is a more difficult operation to analyse. On the face of it, it's O(1) – both the 'Search' and 'Add' functions are constant time operations in general – but every now and then, a reorganisation will take place on an insertion operation. In general, hash tables are written such that they double in size when they grow. This is a O(N) operation, but we can amortise it over all previous insertion operations, so that, overall, 'Insert' remains O(1). Best case then is O(1), worst case is O(N), amortised case is O(1).

The same types of arguments can be made about 'Delete', although in general we tend not to shrink a hash table anywhere near as often as we make it bigger. 'Delete' is then O(1), meaning that the amortised use of a hash table over all its operations is O(1). There is, of course, still that warning that every now and then you will hit the O(N) worst case on an insertion.

Binary tree

The next data structure we can use is a balanced binary search tree, such as a red-black tree. This, like the sorted array version, makes the assumption that names can be sorted.

In a binary tree, the efficiency of search operations is O(d), where d is the maximum depth of the tree (the number of levels from the root of the tree to the furthest leaf). Since a perfectly balanced binary search tree is equivalent to binary search on a sorted array (every link you decide to follow will enable you to ignore a whole chunk of the tree), 'Exists' is on average O(logN). Best case is still O(1), but what about worst case? That depends on the algorithm used to balance the binary tree. Balancing is never perfect but, using red-black trees as an example, we can prove that they're constructed such that the longest path is a maximum of twice the length of the shortest path. If you like, O(2logN). Since 2 is a constant, we can take it out, making red-black trees O(logN) in the worst case for 'Exists'.

For 'Insert' and 'Delete', there's a lot of mathematics that can prove that they're both O(logN) as well. In essence, the search is O(logN), and the addition of the new node or removal of the old node is O(1) on average.

So, overall, a red-black tree is O(logN) in all its operations. Perhaps more importantly, it has guaranteed O(logN) time even in the worst case. This means that some people will prefer to use a red-black tree for their dictionary instead of a hash table because they don't want to hit the possibility of O(N) insertion.

From this discussion, you should now have a basic understanding of how to read and understand big-Oh expressions and how to evaluate algorithms and data structures based on them. Figure 1 illustrates the runtime for various common big-Oh expressions. PCP

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

Profiling

All of the efficiency results quoted in this article are theoretical. They are all of the form 'for large values of N the efficiency is proportional to some expression in N', but make no mention of the size of the constant of proportionality. Therefore, when deciding on which data structure to use in your dictionary, you should profile actual code running on your actual data. It's pointless worrying, for example, about the efficiency of millions of items in a dictionary when you'll only have 100. ■