Understanding ternary trees

Ternary trees are the fastest way to search for data strings, but how do they work?

In this issue...

▶ WHAT'S COVERED

In many applications, there's a requirement for a list of name-value pairs, where we have some kind of interesting information (a record or an object) retrieved using a suitable name. Traditionally, this is done with a dictionary or hash table, but these methods mean you can't easily get at the values. Enter the ternary tree: sorted by name, and faster than a hash table.

Name-value pairs occur all over the place in software. If you have a large set of named values, you'll want to be able to get the value for a particular name (or 'key') at a moment's notice. Examples of this include configuration files, the Windows Registry, the filesystem on your PC, the internet and so forth.

It's instructive to think about how you might do this in the case of an in-memory container. The first thought might be to use an array or linked list. To add a new name-value pair, you'd add it on the end of the list. To find the value for a particular name, you'd start at the front, look at each item in the list and compare its name to the one you're looking for. If you read through the entire list without finding it, the name-value pair wasn't there; otherwise you're guaranteed to find it. This is known as a 'sequential search'.

Although this algorithm would work perfectly well, the problem with it is efficiency. The time taken to find a given name (or not find it, for that matter) is proportional to the number of items you have in the list. For 10 items or so, there's not much to worry about. For a million, however, there's a world of hurt waiting for you, especially if you want to perform the find operation frequently.

The problem is that we're not taking advantage of the fact that we could sort the names and keep the name-value pairs in the sorted order. If we had that system, we could use a 'phone book' search algorithm to find the name we want. This works by starting the search in the middle of the range. If the name we're looking for is early in the alphabet, it'll be in the first half of the book, otherwise it'll be in the second half. We can ignore the half we're not interested in, and start over in the middle of the half we have remaining, repeating as required. This technique is usually known as 'binary search'. The start point is called a 'node', and the two directions in which the search looks are called its 'children'.

Making a hash of it

Since we're halving the search space with every comparison, the number of comparisons reduces dramatically at each step. For a million name-value pairs, on average we'd only need about 20 comparisons before we hit gold, since 2^{20} is about a million.

The issue here is not with finding what we want quickly, but with the addition of new name-value pairs. To do this, we'd have to supplement our simple data structure with ▶

Retrieving a sorted list

Like a binary search tree, the strings stored in a ternary tree can be retrieved in sorted order. This is a little hard to see at first, since the strings themselves aren't in the tree, but we can easily devise a recursive routine that will follow all the links in a 'pre-order traversal' (left, equal, then right) and output each string when we get to the null markers that indicate the end of a string. The pre-order traversal means that the output strings will be in sorted order. ■

> "A trie works by getting the next character in our string and following the link pointed to by it. It's extremely fast"

▲ The structure of the ternary tree is simple, but very powerful. Somewhat like the real thing.

some balancing algorithms to avoid the problem of it degenerating into a linked list. Once this happens, we're reduced to using a sequential search once more.

The solution might be to use a hash table, where we 'hash' (or reduce) the name we're looking for into an integer value, and use that number to index into an array. This is a very fast method, but we lose all information about the name in doing so, and we can't then 'see' the names in sorted order.

Another problem with hash tables is what happens when two names hash to the same value (known as a 'collision'). There are several strategies used to deal with collisions, categorised into two main types: 'probing' and 'chaining'. In either case, we end up reduced to a sequential set of comparisons to sort this out. Obviously, if the hash table gets too loaded, we lose the speed of the 'perfect' hash table and are reduced again to sequential search.

The other problem with all of these proposals, and one that tends to get ignored, is that at some point we will have to compare two strings for equality. In fact, we generally have to do this an awful lot with these solutions (for every item in the sequential and the binary tree, and for every item when we're resolving collisions for the hash table). The longer the string, the more time this takes. Even worse, the more similar the strings are, especially at the start of the string, the more time it takes.

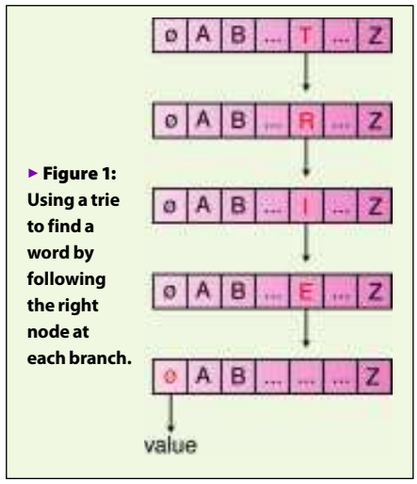
This is where 'tries' come in. A trie (which is officially pronounced 'tree' since it's from the word retrieval, but practically everyone says 'try') is a data structure that stores not whole strings, but the characters in strings.

Suppose our strings just consisted of the uppercase characters A to Z. Instead of using nodes with two children like we did with binary trees, let's use nodes with 27 children, one for each letter of the alphabet and one for an end-of-word terminator. Each node encodes a single letter in a string. To traverse the tree, we pick off letters one by one from our target string and walk through the nodes.

Trie, trie and trie again

Let's assume we were looking for the word 'TRIE'. To see how this works, see Figure 1. Take the first letter 'T' and the root node. Follow the T branch to the next node (also containing 27 children). Take the next letter 'R' and follow the R branch to the next node, which again has 27 children. Do the same for 'I' and then 'E'. Finally, take the end-of-word character link (otherwise you might be following the links for the word 'TRIED' or 'TRIENNIAL', for example). You can see that you'll either eventually have a branch that is null (indicating that the target string is not present) or you'll find the value at the end of the journey 'T', 'R', 'I', 'E', 'end-of-word-marker'.

Note that we're not comparing string values at any point here. All we're doing is getting the next character of our search string and following the link pointed to by it. This is extremely fast, but I'm sure you've spotted the fly in this particular ointment. Suppose we stored all one- to five-letter words in a trie.



▸ **Figure 1: Using a trie to find a word by following the right node at each branch.**

Different languages

Although we can retrieve the strings from a ternary tree in sorted order, this is only true for languages where the string collating sequence can be described in terms of individual characters being 'less than' or 'greater than' others. Some languages sort differently for certain letters or sequences, and in these languages the ternary tree will not produce a correctly ordered list of strings. For example, in German the character 'ß' is sorted as the double character pair 'ss'. In cases such as this, an alternate method is advised. ■

How big would the data structure be? Well, the first thing we can say is that each node is 27 times the size of a link in size: 108 bytes on a 32-bit machine. The first level of the trie only has one node: the root. The second level could have up to 27 nodes, one for each letter of the alphabet plus the terminator. The third level could have 27×27 nodes; the fourth, 27^3 nodes; the fifth, 27^4 nodes. That's a grand total of 551,881 nodes maximum, for at most 59,603,148 bytes in size.

Now obviously, there aren't many small words that would lead to this maximum figure. For example, in the English language the next letter after a 'Q' is always 'U' (ignoring foreign imported words like 'Iraq' or 'qanat'). If we took account of that, we've already saved 26 potential subtrees at the next level under a Q. Despite this qualification, a trie is still very wasteful of space. It may be fast, but that speed is at the expense of space requirements, and these can prove impossible to meet. Just imagine the space requirements for a trie that stored strings with a full set of lower-case letters, digits and punctuation marks – or, worst of all, Unicode characters.

In 1997, aware of these difficulties, Jon Bentley and Robert Sedgewick devised a new data structure that maintained the speed of the trie but reduced its memory footprint dramatically. They called it a ternary tree because it looks a little like a binary tree, but has three links per node instead of two.

In the standard binary search tree, the two children at each node are for all nodes less than the current key (to the left) and all nodes greater than the current key (to

Spotlight on… Partial searches

A fascinating operation we can do on a ternary search tree is partial match searching. This is what we do to solve crossword puzzles. Let's say that there's a five-letter word at 12 across in which the second letter is O and the fourth letter is V, and we need to find out all the words that will fit. This operation is simple to perform with a ternary search tree, mainly because all the nodes are single characters.

Let's suppose the pattern format we use consists of literal characters for the letters we know and a 'don't know' character (say the full stop) for those we don't. So, the pattern for our example crossword clue would be '.O.V.'.

The algorithm is recursive. We peel off the characters one by one from the pattern and perform the normal search operation, with the exception of when the current character is a full stop. For this character, we follow all of the links: left, equal, then right. When we reach the end of a word in the tree, we output it and continue where we left off (because it's entirely possible we'll find several hits for a given pattern, giving us a choice of words). ■

A ternary tree performs better than a balanced binary tree, and is faster at the find operation than a hash table

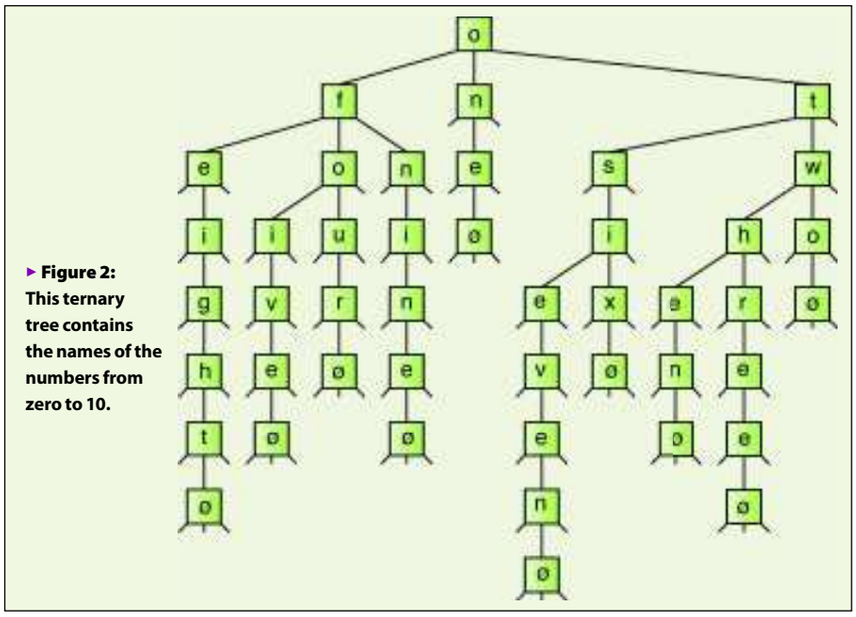


Figure 2: This ternary tree contains the names of the numbers from zero to 10.

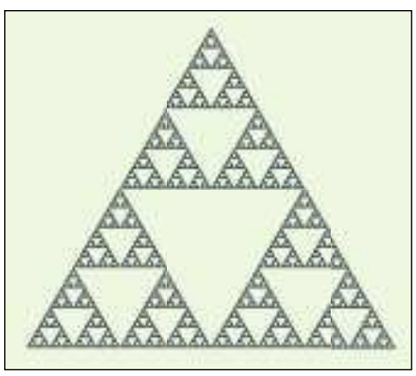
the right). In a ternary search tree, two nodes are used for the same purpose, and the third is used for the 'equality' case. Unlike the binary search tree we briefly discussed above, the ternary search tree doesn't store a complete string at each node. Instead, it stores a single character, just like a trie. Traversing the tree is then a process of picking off characters from our search string. For each character, we compare it against the character in the current node. If it's less than the node character, we take the left child path; if equal, the middle; if greater, the right path.

We only get the next character from the string when we take an equal path. Paths that lead nowhere are represented by null pointers. So, if there's no path for a 'less than' test, the left pointer will be null. Each string is assumed to be terminated with a null character, so that we know when we reach the end of a string.

Figure 2 shows a ternary tree containing the names of the numbers from one to 10. The null links are those going nowhere. Using ternary scheme, let's trace how we would find the value for the word 'TEN' in this tree.

As before, we pick off the characters from our search string, one by one. The first is 'T', and we start by looking at the root node of the tree. The character in that node is 'O'. We compare T against O: it's greater, so we need to take the right-hand path. With the next node, we compare our T against the character, which is a T as well. The characters are equal, and so we move along the 'equal' path. Since we moved down the equal link, we now need to get the next character from the string (E). We compare this against the next node (W). This results in a 'less than' score, so we take the left-hand path to the 'H' node. We continue this way (taking the relevant path, and only getting the next letter from the string when we take the equal route) until we get to the null terminator at the end of our TEN string. If the string is present in the ternary search tree, we shall get to a node whose character is null. If the string is not present, we shall drop off the end of the tree by hitting a null link pointer.

Searching the network for the string was easy enough (after we assumed that a ternary search tree was already set up for us), but how do we create a ternary tree in the first place? Or, more importantly, how do we insert a new string into a ternary tree?



The Sierpinski gasket is a good – and rather pretty – visual representation of a ternary tree.

Seeding the tree

First, it's important to note that the ternary tree structure does not allow us to put the same string into the tree more than once. Duplicates are not allowed.

We start off by using the same method as for the search. We follow the links in the tree until we reach a null link. Once we get to this null link, we add as many nodes as we need in order to express the remaining characters in the string. These nodes are all joined up by the 'equal' links, and each node has a character from the string. So, if we were inserting the word 'TRIE' into an empty ternary search tree (that is, the root node is nil), we'd create a root node with the letter 'T', we'd link that through its equal link to a node with the letter 'R', and link that through its equal link to a node with the letter 'I' and so on. Finally we'd have a long thin ternary search tree with five nodes for T, R, I, E and null. Inserting the word 'TREE' into this tree, would cause a split off to the left at the 'I' node for the first 'E' in TREE, and we'd add three more nodes after that.

Bentley and Sedgewick found that a ternary tree performed better than a balanced binary tree. It was faster to populate than a hash table, and faster at the find operation too. Since, in theory, once a search data structure is set up it is used over and over for various search operations, the ternary tree has become the structure to use for name-value pairs. ■

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

Similar strings

In many applications, the names within the name-value pairs are very similar, in the sense that they start with the same characters and only differ towards the end. For example, a red-black tree program I wrote some time ago had the following set of identifiers: 'RedBlackNode', 'RedBlackTree' and 'RedBlackTreePainter'. The ternary tree performs just as well in cases such as this (after all, it's merely a case of picking off characters and following links), but other algorithms would spend a lot of time comparing the equal front segment of the strings over and over again. ■