Ternary Search Trees

By Jon Bentley and Bob Sedgewick

Jon is a Member of Technical Staff at Bell Labs. Bob is the William O. Baker Professor of Computer Science at Princeton University. They can be reached at jlb@research.bell-labs.com and rs@cs.princeton.edu, respectively.

When you have to store a set of strings, what data structure should you use? You could use hash tables, which sprinkle the strings throughout an array. Access is fast, but information about relative order is lost. Another option is the use of binary search trees, which store strings in order, and are fairly fast. Or you could use digital search tries, which are lightning fast, but use lots of space.

In this article, we'll examine ternary search trees, which combine the time efficiency of digital tries with the space efficiency of binary search trees. The resulting structure is faster than hashing for many typical search problems, and supports a broader range of useful problems and operations. Ternary searches are faster than hashing and more powerful, too.

We described the theory of ternary search trees at a symposium in 1997 (see "Fast Algorithms for Sorting and Searching Strings," by J.L. Bentley and R. Sedgewick, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1997). In this article, we'll concentrate on what the data structure offers working programmers. *Algorithms in C*, Third Edition, by Robert Sedgewick (Addison-Wesley, 1998) provides yet another view of ternary search trees. For more information (including all the code in this article and the driver software), refer to http://www.cs.princeton.edu/~rs/strings/

A Grove of Trees

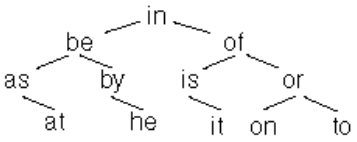

Figure 1

Figure 1 is a binary search tree that represents 12 common two-letter words. For every node, all nodes down the left child have smaller values, while all nodes down the right child have greater values. A search starts at the root. To find the word "on," for instance, we compare it to "in" and take the right branch. We take the right branch at "of" and the left branch at "or," and then arrive at "on." Every comparison could access each character of both words.

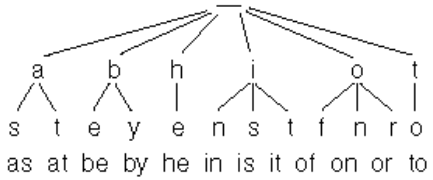

Figure 2

Digital search tries store strings character by character. Figure 2 is a tree that represents the same set of 12 words; each input word is shown beneath the node that represents it. (Two-letter words lead to prettier pictures; all structures that we'll see can store variable-length words.) In a tree representing words of lowercase letters, each node has 26-way branching (though most branches are empty, and not shown in Figure 2). Searches are very fast: A search for "is" starts at the root, takes the "i" branch, then the "s" branch, and ends at the desired node. At every node, we access an array element (one of 26), test for null, and take a branch. Unfortunately, search tries have exorbitant space requirements: Nodes with 26-way branching typically occupy 104 bytes, and 256-way nodes consume a kilobyte. Eight nodes representing the 34,000-character Unicode Standard would together require more than a megabyte!

Ternary search trees combine attributes of binary search trees and digital search tries. Like tries, they proceed character by character. Like binary search trees, they are space efficient, though each node has three children, rather than two. A search compares the current character in the search string with the character at the node. If the search character is less, the search goes to the left child; if the search character is greater, the search goes to the right child. When the search character is equal, though, the search goes to the middle child, and proceeds to the next character in the search string.

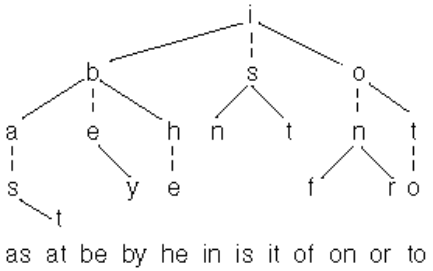

Figure 3

Figure 3 is a balanced ternary search tree for the same set of 12 words. The low and high pointers are shown as solid lines, while equal pointers are shown as dashed lines. Each input word is shown beneath its terminal node. A search for the word "is" starts at the root, proceeds down the equal child to the node with value "s," and stops there after two comparisons. A search for "ax" makes three comparisons to the first letter ("a") and two comparisons to the second letter ("x") before reporting that the word is not in the tree.

The idea behind ternary search trees dates back at least as far as 1964. In "Randomized Binary Searching with Tree Structures" (*Communications of the ACM*, March 1964), H.A. Clampett sketched a primitive version of the structure. Computer scientists have proven many theorems about the trees; for instance, searching for a string of length *k* in a ternary search tree with *n* strings will require at most *O*(*log n*+*k*) comparisons.

The C Structure

As far as we can tell, previous authors have viewed ternary search trees as a theoretical structure for proving theorems. We were delighted to find that the trees also lead to practical computer programs. Although we've chosen to illustrate the data structure in C, we could have just as easily chosen C++, Java, or other languages.

Each node in a ternary search tree is represented by the following structure:

```
typedef struct tnode *Tptr;
typedef struct tnode {
   char splitchar;
   Tptr lokid, eqkid, hikid;
} Tnode;
```

The value stored at the node is *splitchar*, and the three pointers represent the three children. The root of the tree is declared to be *Tptr root*. We will represent every character in each string, including the null character that terminates it.

Membership Searching

We begin with a recursive version of the *search* function. It returns 1 if string *s* is in the subtree rooted at *p*, and 0 otherwise; it is originally called as *rsearch(root, s)*:

```
int rsearch(Tptr p, char *s) {
   if (!p) return 0;
   if (*s < p->splitchar)
      return rsearch(p->lokid, s);
   else if (*s > p->splitchar)
      return rsearch(p->hikid, s);
   else {
      if (*s == 0) return 1;
      return rsearch(p->eqkid, ++s);
   }
}
```

The first *if* returns 0 if the search has run off the end of the tree. The next two *if* statements take the low and high branches as appropriate. The final *else* branch returns 1 if the current character is the end-of-string character 0, and otherwise moves to the next input character and to the equal branch.

Some programmers might feel more comfortable with the iterative version of the *search* function (which has only one argument):

```
int search(char *s) {
   Tptr p;
   p = root;
   while (p) {
      if (*s < p->splitchar)
         p = p->lokid;
      else if (*s == p->splitchar) {
         if (*s++ == 0) return 1;
         p = p->eqkid;
      }
      else
         p = p->hikid;
   }
   return 0;
}
```

This is substantially faster on some compilers, and we'll use it in later experiments. On most optimizing compilers, though, the run time of the recursive version is within a few percent of the iterative version.

Both versions of the search use a pattern that we will see repeatedly: If the search character is less, go to *lokid*; if it is greater, go to *hikid*; if it is equal, go to the next character and *eqkid*. The following code illustrates that shorter is not always cleaner, simpler, faster, or better:

```
int rsearch2(Tptr p, char *s) {
   return (!p ? 0 :
           (*s == p->splitchar ? (*s ? rsearch2(p->eqkid, ++s) : 1) :
           (rsearch2(*s < p->splitchar ? p->lokid : p->hikid, s))))
}
```

Inserting a New String

The *insert* function inserts a new string into the tree, and does nothing if it is already present. We insert the string *s* with the code *root = insert(root, s);*. The first *if* statement detects running off the end of new node, initializes it, and falls through to the standard case. Subsequent code takes the appropriate branch, but branches to *eqkid* only if characters remain in the string.

```
Tptr insert(Tptr p, char *s) {
   if (p == 0) {
      p = (Tptr) malloc(sizeof(Tnode));
      p->splitchar = *s;
      p->lokid = p->eqkid = p->hikid = 0;
   }
   if (*s < p->splitchar)
     p->lokid = insert(p->lokid, s);
   else if (*s == p->splitchar) {
      if (*s != 0)
         p->eqkid = insert(p->eqkid, ++s);
   }
   else
      p->hikid = insert(p->hikid, s);
   return p;
}
```

This code is short but subtle, and worth careful study.

The resulting tree stores the strings themselves, but no other information. Real symbol tables store additional data with each string. To illustrate this problem, we'll store a pointer to every string in the tree; this data will be used by later search algorithms. We could add a new *info* field to each node, but that would be wasteful. Instead, we'll exploit the fact that a node with a null *splitchar* cannot have an *eqkid*, and store the data in that field. We could make a *union* that contains the two possible pointers, but that would be syntactically clumsy (we would have to refer to the union at each reference). We will instead use the sleazy hack of coercing a string into a *Tptr*. The code inside *s == *p->splitchar* test, therefore, becomes the following:

```
if (*s == 0)
   p->eqkid = (Tptr) insertstr;
else
   p->eqkid = insert(p->eqkid, ++s);
```

Faster Insertion Functions

We can improve insertion in two different ways. We'll start by tuning the insertion code, and consider different orders in which we can insert the nodes into the tree.

A major cost of the *insert* function is calling *malloc* to create each node. The function *insert2* in the demo program (available electronically) uses the ancient C technique of allocating a buffer of nodes and dealing them out as needed. Profiling shows that this eliminates most of the time spent in storage allocation. We measured the time to insert the 234,936 words in a dictionary file (one word per line, for a total of 2,486,813 characters). Table 1 lists the run times in seconds on three 200-MHz machines, with the compilers optimizing code for speed. The *insert3* function (also available electronically) uses other common techniques to reduce the run time even further: transforming recursion to iteration, keeping a pointer to a pointer, reordering tests, saving a difference in a comparison, and splitting the single loop into two loops. These changes are beneficial on all machines, and substantial on the SPARC.

Machine	insert1	insert2	insert3
UltraSPARC-2	3.95	2.75	0.49
MIPS R4000	2.40	1.40	1.00
Pentium Pro	7.91	1.10	0.71

Table 1

Better Insertion Orders

In what order should you insert the nodes into a tree? No matter in what order you insert the nodes, you end up with the same digital search trie -- the data structure is totally insensitive to insertion order. Binary search trees are at the opposite end of the spectrum: If you insert the nodes in a good order (middle element first), you end up with a balanced tree. If you insert the nodes in sorted order, the result is a long skinny tree that is very costly to build and search.

Fortunately, if you insert the nodes in random order, a binary search tree is usually close to balanced.

Ternary search trees fall between these two extremes. You can build a completely balanced tree by inserting the median element of the input set, then recursively inserting all lesser elements and greater elements. A simpler approach first sorts the input set. The recursive *build* function inserts the middle string of its subarray, then recursively builds the left and right subarrays. We use this method in our experiments; it is fast and produces fairly well-balanced trees. The cost of inserting all words in a dictionary with function *insert3* is never more than about 10 percent greater than searching for all words. D.D. Sleator and R.E. Tarjan describe theoretical balancing algorithms for ternary search trees in "Self-Adjusting Binary Search Trees" (*Journal of the ACM*, July 1985).

Comparison to Other Data Structures

Symbol tables are typically represented by hash tables. We conducted a simple experiment to compare ternary search trees to that classic data structure; the complete code is available electronically.

To represent *n* strings, our hash code uses a chained table of size *tabsize=n*. The hash function, from Section 6.6 of B. Kernighan and D. Ritchie's *The C Programming Language*, Second Edition (Prentice Hall, 1988), is reasonably efficient and produces good spread:

```
int hashfunc(char *s) {
   unsigned n = 0;
   for ( ; *s; s++)
      n = 31 * n + *s;
   return n % tabsize;
}
```

The body of the *search* function is as follows:

```
for (p = tab[hashfunc(s)]; p; p = p->next)
   if (strcmp(s, p->str) == 0) return 1;
return 0;
```

For fair timing, we replaced the string comparison function *str*cmp with inline code (so the hash and tree *search* functions used the same coding style). We used the same dictionary to compare the performance of hashing and ternary trees. We first built the tree by inserting each word in turn (middle element first, then recurring). Finally, we searched for each element word in the input file. Table 2 presents the times in seconds for the two data structures across three machines. For both building and (successful) searching, the two structures have comparable search times.

Machine	Build		Search	
	Hash	TST	Hash	TST
UltraSPARC-2	0.53	0.49	0.58	0.52
MIPS R4000	0.83	1.00	1.00	0.91
Pentium Pro	0.93	0.71	0.55	0.64

Ternary search trees are usually noticeably faster than hashing for unsuccessful searches. Ternary trees can discover mismatches after examining only a few characters, while hashing always processes the entire key. On some data sets with very long keys and mismatches in the first few characters, ternary trees took less than one-fifth the time of hashing.

Functions *insert3* and *search* combine to yield a time-efficient symbol table. On one dictionary described at our web site, ternary search trees used about three times the space of hashing. An alternative representation of ternary search trees is more space efficient: When a subtree contains a single string, we store a pointer to the string itself (and each node stores three bits telling whether its children point to nodes or strings). This leads to code that is less efficient, but it reduces the number of tree nodes close to the space used by hashing.

We analyzed the worst-case performance of several aspects of ternary search trees in our theoretical paper. In "The Analysis of Hybrid Trie Structures" (*Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998), J. Clement, P. Flajolet, and B. Valle analyze the expected performance of a broad class of hybrid trees that includes ternary trees, and describe extensive experiments to support their theory.

Other Operations on Ternary Trees

Most standard techniques on binary search trees can be applied immediately to their ternary cousins. For instance, we can print the strings in the tree in sorted order with a recursive traversal:

```
void traverse(Tptr p) {
   if (!p) return;
   traverse(p->lokid);
   if (p->splitchar)
      traverse(p->eqkid);
   else
      printf("%s/n", (char *) p->eqkid);
   traverse(p->hikid);
 }
```

Simple recursive searches can find the predecessor or successor of a given element or list all items in a given range. If we add a *count* field to every node, we can quickly count the elements in a given range, count how many words begin with a given substring, or select the *m*th largest element. Most of these operations require logarithmic time in a ternary tree, but linear time in a hash table.

Partial-Match Searching

We turn next to the more subtle problem of "partial-match" or "crossword puzzle" searching: A query string may contain both regular letters and the "don't care" character ".". Searching the dictionary for the pattern ".u.u.u" matches the single word *auhuhu*, while the pattern ".a.a.a"

matches 94 words, including *banana*, *casaba*, and *pajama*. (That pattern does not match *abracadabra*, though. The entire word must match the pattern; we do not look for patterns in the middle of longer words.)

This venerable problem has been studied in many papers, such as in "The World's Fastest Scrabble Program," by A.W. Appel and G.J. Jacobson (*Communications of the ACM, May 1988*). In "Partial-Match Retrieval Algorithms," (*SIAM Journal on Computing, 5*, 1976), R.L. Rivest presents an algorithm for partial-match searching in digital tries: Take the single given branch if a letter is specified, for a don't-care character, recursively search all branches. Function *pmsearch* implements Rivest's method in ternary search trees. The function puts pointers to matching words in *srcharr[0..srchtop-1]*. It is called, for instance, by:

```
srchtop = 0;
pmsearch(root, ".a.a.a");
```

The following is the complete search code:

```
void pmsearch(Tptr p, char *s) {
   if (!p) return;
   nodecnt++;
   if (*s == '.' || *s < p->splitchar)
      pmsearch(p->lokid, s);
   if (*s == '.' || *s == p->splitchar)
      if (p->splitchar && *s)
         pmsearch(p->eqkid, s+1);
   if (*s == 0 && p->splitchar == 0)
      srcharr[srchtop++] = (char *) p->eqkid;
   if (*s == '.' || *s > p->splitchar)
      pmsearch(p->hikid, s);
 }
```

Function *pmsearch* has five *if* statements. The second and fifth *if* statements are symmetric; they recursively search the *lokid* (or *hikid*) when the search character is the don't care "." or when the search string is less (or greater) than the *splitchar*. The third *if* statement recursively searches the *eqkid* if both the *splitchar* and string are nonnull. The fourth *if* statement detects a match to the query and adds the pointer to the complete word (stored in *eqkid* by our sleazy hack).

Rivest states that partial-match search in a trie requires "time about $O(n^{(k-s)/k})$ to respond to a query word with *s* letters specified, given a file of *n* *k*-letter words." Ternary search trees can be viewed as an implementation of his tries (with binary trees implementing multiway branching), so we expected his results to apply immediately to our program. Our experiments, however, led to a surprise: Unspecified positions at the front of the query word are dramatically more costly than unspecified characters at the end of the word. (Rivest suggested shuffling the characters in a word to avoid such problems.) Table 3 gives the flavor of our experiments. The first line says that the search visited 18 nodes to find the single match to the pattern "banana." The tree contains a total of 1,026,033 nodes; searches with some of the first letters specified visit just a tiny fraction of those nodes.

Pattern	Matches	Nodes
banana	1	18
ban...	33	352
.a.a.a	94	5821
...ana	38	24,069
xy.....	7	250
.....xy	16	441,583
television	1	21
tele.....	27	639
t.l.v.s..n	1	254
....vision	6	73,786

Table 3

Near-Neighbor Searching

We finally examine the problem of "near-neighbor searching" in a set of strings: We are to find all words in the dictionary that are within a given Hamming distance of a query word. For instance, a search for all words within distance two of *Dobbs* finds *Debby*, *hobby*, and 14 other words.

Function *nearsearch* performs a near-neighbor search in a ternary search tree. Its three arguments are a tree node, string, and distance. The first *if* statement returns if the node is null or the distance is negative. The second and fourth *if* statements are symmetric: They search the appropriate child if the distance is positive or if the query character is on the appropriate side of *splitchar*. The third *if* statement either checks for a match or recursively searches the middle child.

```
void nearsearch(Tptr p, char *s, int d) {
   if (!p || d < 0) return;
   if (d > 0 || *s < p->splitchar)
      nearsearch(p->lokid, s, d);
   if (p->splitchar == 0) {
      if ((int) strlen(s) <= d)
         srcharr[srchtop++] = (char *) p->eqkid;
   }
   else
      nearsearch(p->eqkid, *s ? s+1:s, (*s == p->splitchar) ? d:d-1);
   if (d > 0 || *s > p->splitchar)
      nearsearch(p->hikid, s, d);
}
```

Our web site contains data on the performance of this search algorithm. It is quite efficient when searching for near neighbors, but searching for distant neighbors grows more expensive. A simple probabilistic model accurately predicts its run time on real data.

Conclusion

The primary challenge in implementing digital search tries is to avoid using excessive memory for trie nodes that are nearly empty. Ternary search trees may be viewed as a trie implementation that gracefully adapts to handle this case, at the cost of slightly more work for full nodes. Ternary search trees combine the best of two worlds: the low space overhead of binary search trees and the character-based time efficiency of digital search tries.

Ternary search trees have been used for several years to represent English dictionaries in a commercial optical character recognition (OCR) system built at Bell Labs. The trees were faster than hashing for the task, and they gracefully handle the 34,000-character set of the Unicode Standard. The designers have also experimented with using partial-match for word lookup: Replace letters with low probability of recognition with the "don't care" character.

Ternary search trees are efficient and easy to implement. They offer substantial advantages over both binary search trees and digital search tries. We feel that they are superior to hashing in many applications for the following reasons:

- Ternary trees do not incur extra overhead for insertion or successful searches.
- Ternary trees are usually substantially faster than hashing for unsuccessful searches.
- Ternary trees gracefully grow and shrink; hash tables need to be rebuilt after large size changes.
- Ternary trees support advanced searches, such as partial-match and near-neighbor search.
- Ternary trees support many other operations, such as traversal to report items in sorted order.