12.5 Tries

The pattern matching algorithms presented in the previous section speed up the search in a text by preprocessing the pattern (to compute the failure function in the KMP algorithm or the last function in the BM algorithm). In this section, we take a complementary approach, namely, we present string searching algorithms that preprocess the text. This approach is suitable for applications where a series of queries is performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query (for example, a Web site that offers pattern matching in Shakespeare's *Hamlet* or a search engine that offers Web pages on the *Hamlet* topic).

A ***trie*** (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "re*trie*val." In an information retrieval application, such as a search for a certain DNA sequence in a genomic database, we are given a collection S of strings, all defined using the same alphabet. The primary query operations that tries support are pattern matching and ***prefix matching***. The latter operation involves being given a string X, and looking for all the strings in S that contain X as a prefix.

12.5.1 Standard Tries

Let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string. A ***standard trie*** for S is an ordered tree T with the following properties (see Figure 12.9):

- Each node of T, except the root, is labeled with a character of Σ.
- The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ.
- T has s external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node v of T yields the string of S associated with v.

Thus, a trie T represents the strings of S with paths from the root to the external nodes of T. Note the importance of assuming that no string in S is a prefix of another string. This ensures that each string of S is uniquely associated with an external node of T. We can always satisfy this assumption by adding a special character that is not in the original alphabet Σ at the end of each string.

An internal node in a standard trie T can have anywhere between 1 and d children, where d is the size of the alphabet. There is an edge going from the root r to one of its children for each character that is first in some string in the collection S. In addition, a path from the root of T to an internal node v at depth i corresponds to

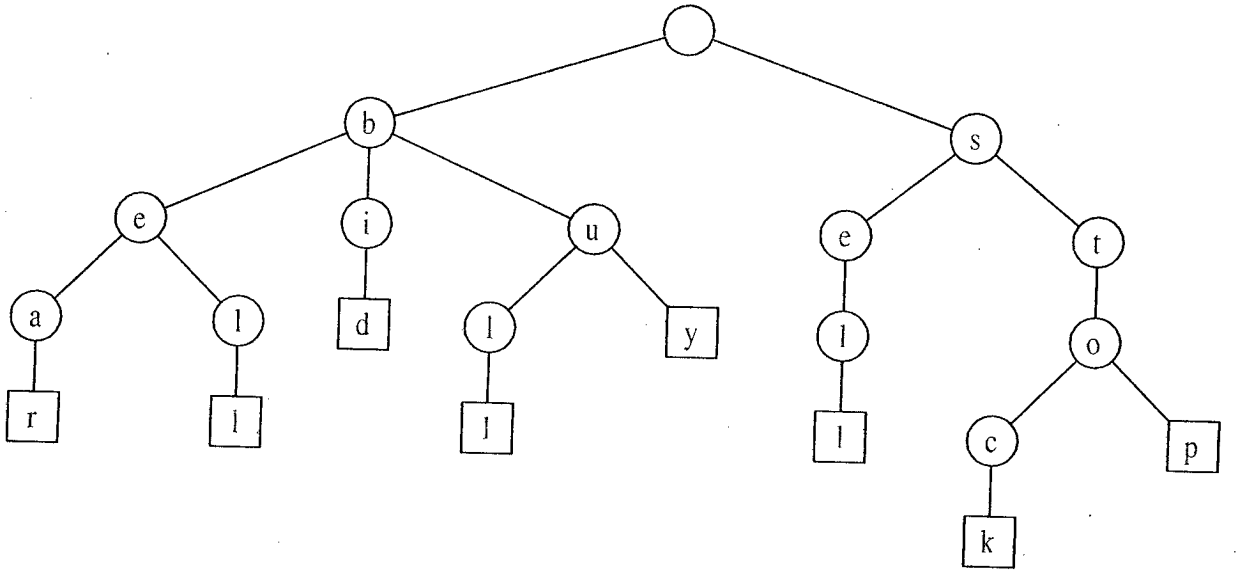


Figure 12.9: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

an i-character prefix $X[0..i-1]$ of a string X of S. In fact, for each character c that can follow the prefix $X[0..i-1]$ in a string of the set S, there is a child of v labeled with character c. In this way, a trie concisely stores the common prefixes that exist among a set of strings.

If there are only two characters in the alphabet, then the trie is essentially a binary tree, with some internal nodes possibly having only one child (that is, it may be an improper binary tree). In general, if there are d characters in the alphabet, then the trie will be a multi-way tree where each internal node has between 1 and d children. In addition, there are likely to be several internal nodes in a standard trie that have fewer than d children. For example, the trie shown in Figure 12.9 has several internal nodes with only one child. We can implement a trie with a tree storing characters at its nodes.

The following proposition provides some important structural properties of a standard trie:

Proposition 12.9: *A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:*

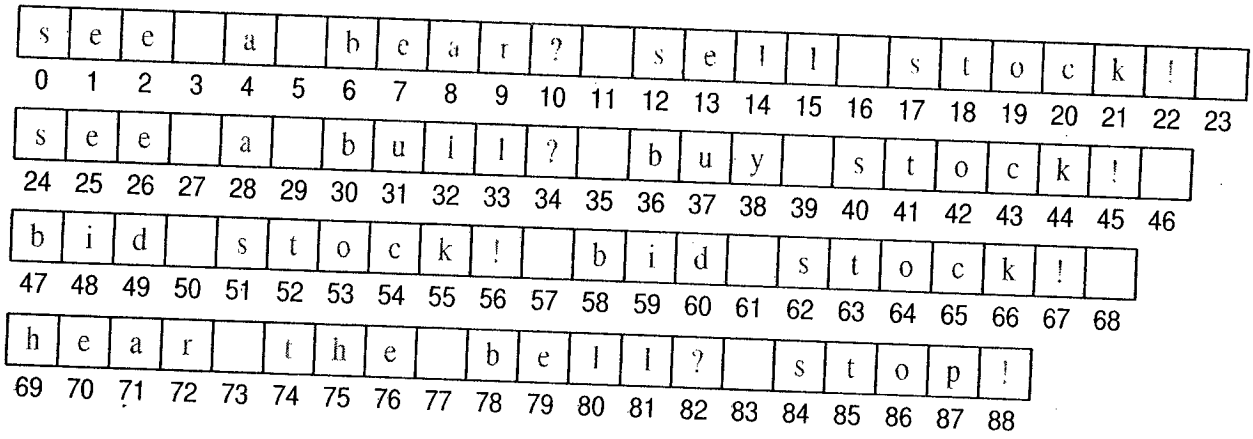
- *Every internal node of T has at most d children.*
- *T has s external nodes.*
- *The height of T is equal to the length of the longest string in S.*
- *The number of nodes of T is $O(n)$.*

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix; that is, except for the root, all internal nodes have one child.

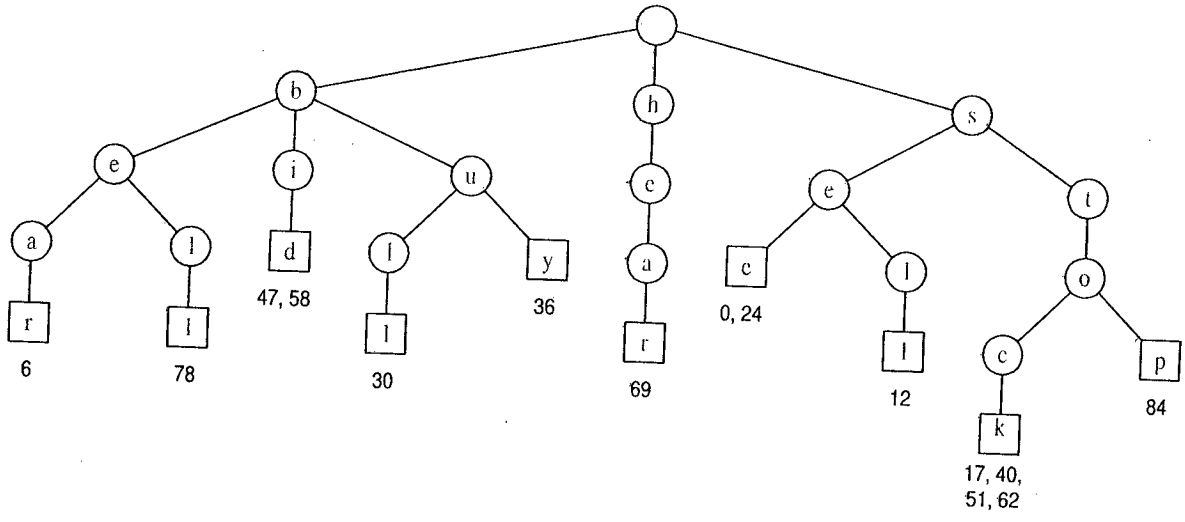
A trie T for a set S of strings can be used to implement a dictionary whose keys are the strings of S. Namely, we perform a search in T for a string X by tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at an external node, then we know X is in the dictionary. For example, in the trie in Figure 12.9, tracing the path for "bull" ends up at an external node. If the path cannot be traced or the path can be traced but terminates at an internal node, then X is not in the dictionary. In the example in Figure 12.9, the path for "bet" cannot be traced and the path for "be" ends at an internal node. Neither such word is in the dictionary. Note that in this implementation of a dictionary, single characters are compared instead of the entire string (key). It is easy to see that the running time of the search for a string of size m is $O(dm)$, where d is the size of the alphabet. Indeed, we visit at most $m+1$ nodes of T and we spend $O(d)$ time at each node. For some alphabets, we may be able to improve the time spent at a node to be $O(1)$ or $O(\log d)$ by using a dictionary of characters implemented in a hash table or search table. However, since d is a constant in most applications, we can stick with the simple approach that takes $O(d)$ time per node visited.

From the discussion above, it follows that we can use a trie to perform a special type of pattern matching, called ***word matching***, where we want to determine whether a given pattern matches one of the words of the text exactly. (See Figure 12.10.) Word matching differs from standard pattern matching since the pattern cannot match an arbitrary substring of the text, but only one of its words. Using a trie, word matching for a pattern of length m takes $O(dm)$ time, where d is the size of the alphabet, independent of the size of the text. If the alphabet has constant size (as is the case for text in natural languages and DNA strings), a query takes $O(m)$ time, proportional to the size of the pattern. A simple extension of this scheme supports prefix matching queries. However, arbitrary occurrences of the pattern in the text (for example, the pattern is a proper suffix of a word or spans two words) cannot be efficiently performed.

To construct a standard trie for a set S of strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no string of S is a prefix of another string. To insert a string X into the current trie T, we first try to trace the path associated with X in T. Since X is not already in T and no string in S is a prefix of another string, we will stop tracing the path at an ***internal*** node v of T before reaching the end of X. We then create a new chain of node descendents of v to store the remaining characters of X. The time to insert X is $O(dm)$, where m is the length of X and d is the size of the alphabet. Thus, constructing the entire trie for set S takes $O(dn)$ time, where n is the total length of the strings of S.



(a)



(b)

Figure 12.10: Word matching and prefix matching with a standard trie: (a) text to be searched; (b) standard trie for the words in the text (articles and prepositions, which are also known as ***stop words***, excluded), with external nodes augmented with indications of the word positions.

There is a potential space inefficiency in the standard trie that has prompted the development of the ***compressed trie***, which is also known (for historical reasons) as the ***Patricia trie***. Namely, there are potentially a lot of nodes in the standard trie that have only one child, and the existence of such nodes is a waste. We discuss the compressed trie next.