13.3 Tries

The pattern-matching algorithms presented in Section 13.2 speed up the search in a text by preprocessing the pattern (to compute the *last* function in the Boyer-Moore algorithm or the *failure* function in the Knuth-Morris-Pratt algorithm). In this section, we take a complementary approach, namely, we present string searching algorithms that preprocess the text, rather than the pattern. This approach is suitable for applications in which many queries are 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 website that offers pattern matching in Shakespeare's *Hamlet* or a search engine that offers Web pages containing the term *Hamlet*).

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 being with X.

13.3.1 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 13.7):

- Each node of T, except the root, is labeled with a character of Σ.
- The children of an internal node of T have distinct labels.
- T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf 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 leaves 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 a leaf of T. (This is similar to the restriction for prefix codes with Huffman coding, as described in Section 13.4.) 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 $|\Sigma|$ children. 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 k corresponds to a k-character prefix

www.it-ebooks.info

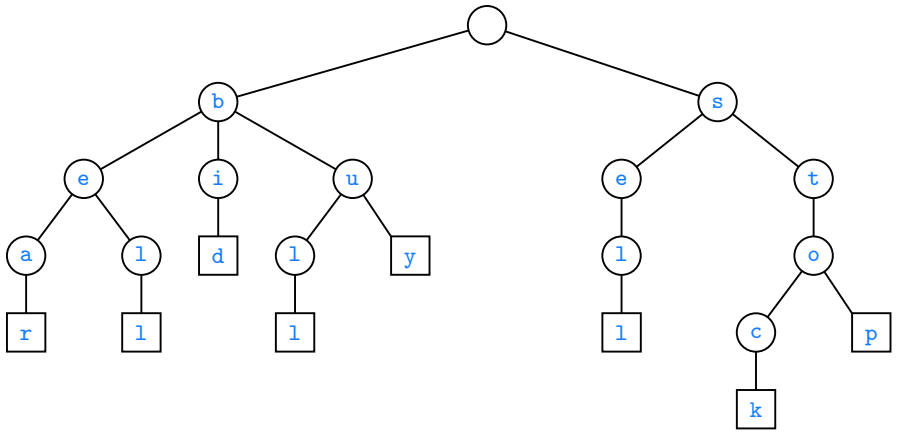


Figure 13.7: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

$X[0..k-1]$ of a string X of S. In fact, for each character c that can follow the prefix $X[0..k-1]$ 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.

As a special case, 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, although it is possible that an internal node has up to $|\Sigma|$ children, in practice the average degree of such nodes is likely to be much smaller. For example, the trie shown in Figure 13.7 has several internal nodes with only one child. On larger data sets, the average degree of nodes is likely to get smaller at greater depths of the tree, because there may be fewer strings sharing the common prefix, and thus fewer continuations of that pattern. Furthermore, in many languages, there will be character combinations that are unlikely to naturally occur.

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

Proposition 13.4: *A standard trie storing a collection S of s strings of total length n from an alphabet Σ has the following properties:*

- *The height of T is equal to the length of the longest string in S.*
- *Every internal node of T has at most $|\Sigma|$ children.*
- *T has s leaves.*
- *The number of nodes of T is at most $n+1$.*

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.

www.it-ebooks.info

A trie T for a set S of strings can be used to implement a set or map 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 a leaf node, then we know X is a string in S. For example, in the trie in Figure 13.7, tracing the path for "bull" ends up at a leaf. If the path cannot be traced or the path can be traced but terminates at an internal node, then X is not a string in S. In the example in Figure 13.7, the path for "bet" cannot be traced and the path for "be" ends at an internal node. Neither such word is in the set S.

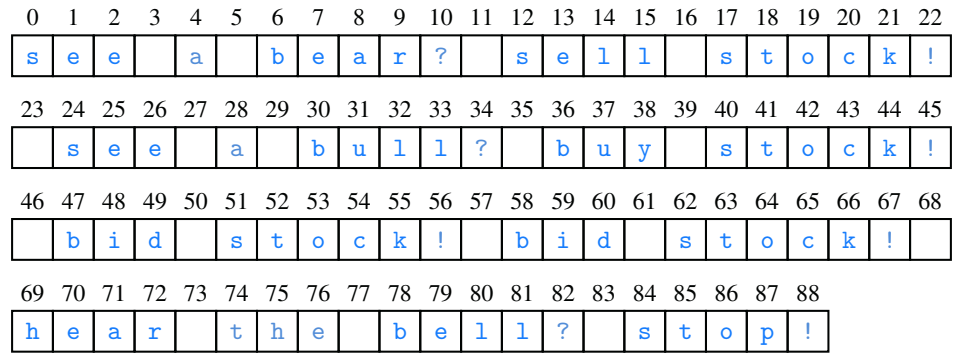
It is easy to see that the running time of the search for a string of length m is $O(m \cdot |\Sigma|)$, because we visit at most $m+1$ nodes of T and we spend $O(|\Sigma|)$ time at each node determining the child having the subsequent character as a label. The $O(|\Sigma|)$ upper bound on the time to locate a child with a given label is achievable, even if the children of a node are unordered, since there are at most $|\Sigma|$ children. We can improve the time spent at a node to be $O(\log |\Sigma|)$ or expected $O(1)$, by mapping characters to children using a secondary search table or hash table at each node, or by using a direct lookup table of size $|\Sigma|$ at each node, if $|\Sigma|$ is sufficiently small (as is the case for DNA strings). For these reasons, we typically expect a search for a string of length m to run in $O(m)$ time.

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. Word matching differs from standard pattern matching because the pattern cannot match an arbitrary substring of the text—only one of its words. To accomplish this, each word of the original document must be added to the trie. (See Figure 13.8.) 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.

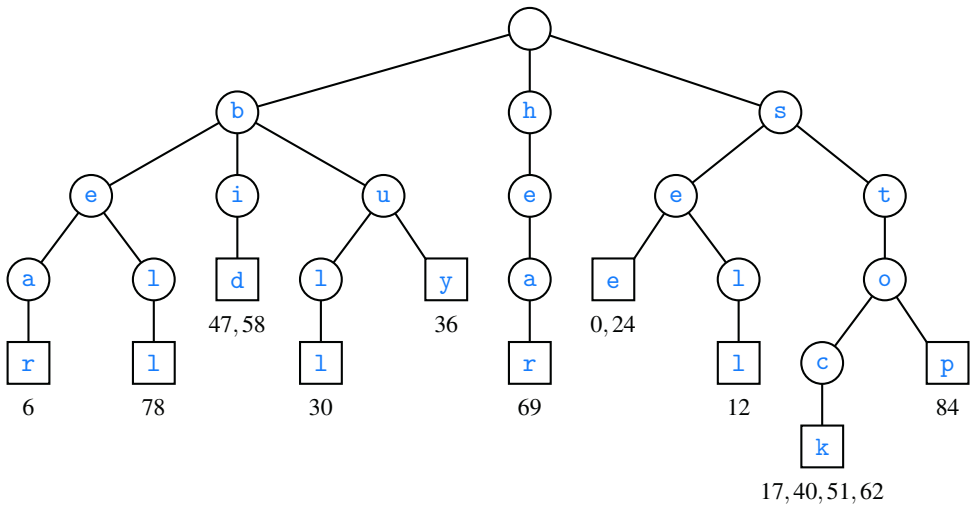
To construct a standard trie for a set S of strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no string of S is a prefix of another string. To insert a string X into the current trie T, we trace the path associated with X in T, creating a new chain of nodes to store the remaining characters of X when we get stuck. The running time to insert X with length m is similar to a search, with worst-case $O(m \cdot |\Sigma|)$ performance, or expected $O(m)$ if using secondary hash tables at each node. Thus, constructing the entire trie for set S takes expected $O(n)$ time, where n is the total length of the strings of S.

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.

www.it-ebooks.info



(a)



(b)

Figure 13.8: Word matching with a standard trie: (a) text to be searched (articles and prepositions, which are also known as ***stop words***, excluded); (b) standard trie for the words in the text, with leaves augmented with indications of the index at which the given work begins in the text. For example, the leaf for the word "stock" notes that the word begins at indices 17, 40, 51, and 62 of the text.

www.it-ebooks.info

13.3.2 Compressed Tries

A ***compressed trie*** is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. (See Figure 13.9.) Let T be a standard trie. We say that an internal node v of T is ***redundant*** if v has one child and is not the root. For example, the trie of Figure 13.7 has eight redundant nodes. Let us also say that a chain of $k \geq 2$ edges,

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k),$$

is ***redundant*** if:

- v_i is redundant for $i = 1, \ldots, k-1$.
- v_0 and v_k are not redundant.

We can transform T into a compressed trie by replacing each redundant chain $(v_0, v_1) \cdots (v_{k-1}, v_k)$ of $k \geq 2$ edges into a single edge (v_0, v_k), relabeling v_k with the concatenation of the labels of nodes $v_1, \ldots, v_k$.

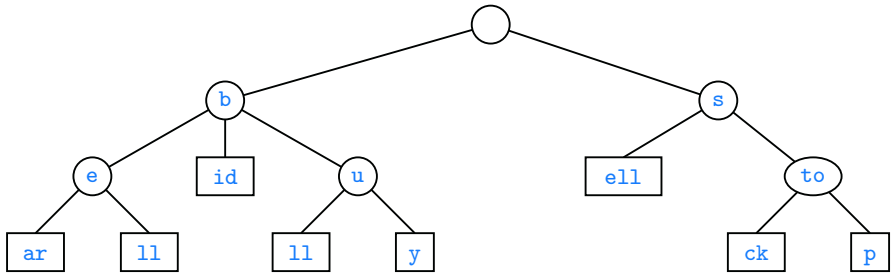


Figure 13.9: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. (Compare this with the standard trie shown in Figure 13.7.) Notice that, in addition to compression at the leaves, the internal node with label "to" is shared by words "stock" and "stop".

Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters. The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length, as shown in the following proposition (compare with Proposition 13.4).

Proposition 13.5: *A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties:*

- *Every internal node of T has at least two children and most d children.*
- *T has s leaves nodes.*
- *The number of nodes of T is $O(s)$.*

www.it-ebooks.info

The attentive reader may wonder whether the compression of paths provides any significant advantage, since it is offset by a corresponding expansion of the node labels. Indeed, a compressed trie is truly advantageous only when it is used as an ***auxiliary*** index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection.

Suppose, for example, that the collection S of strings is an array of strings $S[0]$, $S[1]$, ..., $S[s-1]$. Instead of storing the label X of a node explicitly, we represent it implicitly by a combination of three integers (i, j, k), such that $X = S[i][j..k]$; that is, X is the substring of $S[i]$ consisting of the characters from the j^{th} to the k^{th} inclusive. (See the example in Figure 13.10. Also compare with the standard trie of Figure 13.8.)

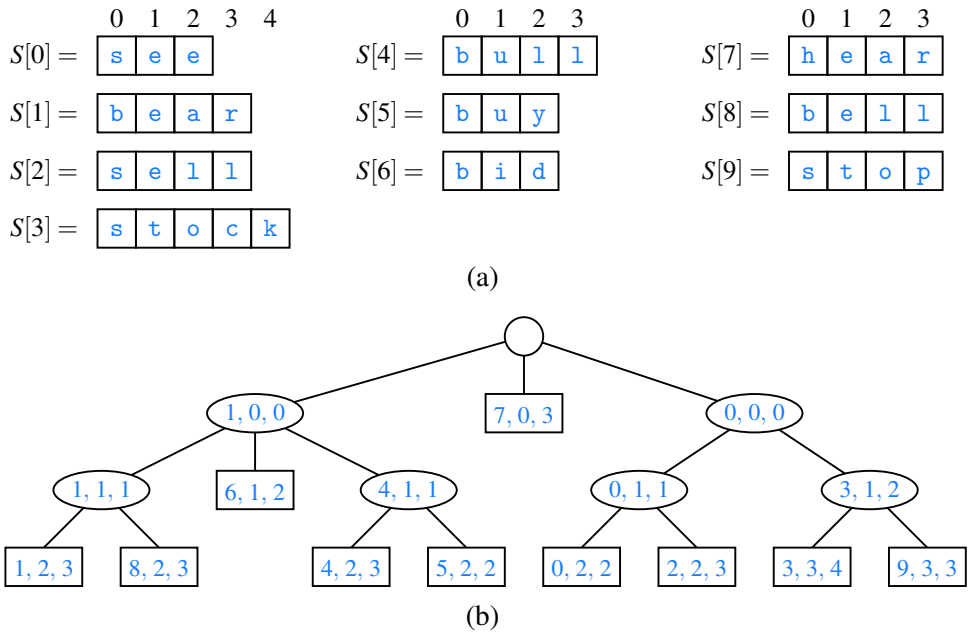


Figure 13.10: (a) Collection S of strings stored in an array. (b) Compact representation of the compressed trie for S.

This additional compression scheme allows us to reduce the total space for the trie itself from $O(n)$ for the standard trie to $O(s)$ for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S. We must still store the different strings in S, of course, but we nevertheless reduce the space for the trie.

Searching in a compressed trie is not necessarily faster than in a standard tree, since there is still need to compare every character of the desired pattern with the potentially multicharacter labels while traversing paths in the trie.

www.it-ebooks.info

13.3.3 Suffix Tries

One of the primary applications for tries is for the case when the strings in the collection S are all the suffixes of a string X. Such a trie is called the ***suffix trie*** (also known as a ***suffix tree*** or ***position tree***) of string X. For example, Figure 13.11a shows the suffix trie for the eight suffixes of string "minimize." For a suffix trie, the compact representation presented in the previous section can be further simplified. Namely, the label of each vertex is a pair "j..k" indicating the string $X[j..k]$. (See Figure 13.11b.) To satisfy the rule that no suffix of X is a prefix of another suffix, we can add a special character, denoted with \$, that is not in the original alphabet Σ at the end of X (and thus to every suffix). That is, if string X has length n, we build a trie for the set of n strings $X[j..n-1]$\$, for $j = 0,\ldots,n-1$.

Saving Space

Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

The advantage of the compact representation of tries now becomes apparent for suffix tries. Since the total length of the suffixes of a string X of length n is

$$1+2+\cdots+n = \frac{n(n+1)}{2},$$

storing all the suffixes of X explicitly would take $O(n^2)$ space. Even so, the suffix trie represents these strings implicitly in $O(n)$ space, as formally stated in the following proposition.

Proposition 13.6: *The compact representation of a suffix trie T for a string X of length n uses $O(n)$ space.*

Construction

We can construct the suffix trie for a string of length n with an incremental algorithm like the one given in Section 13.3.1. This construction takes $O(|\Sigma|n^2)$ time because the total length of the suffixes is quadratic in n. However, the (compact) suffix trie for a string of length n can be constructed in $O(n)$ time with a specialized algorithm, different from the one for general tries. This linear-time construction algorithm is fairly complex, however, and is not reported here. Still, we can take advantage of the existence of this fast construction algorithm when we want to use a suffix trie to solve other problems.

www.it-ebooks.info

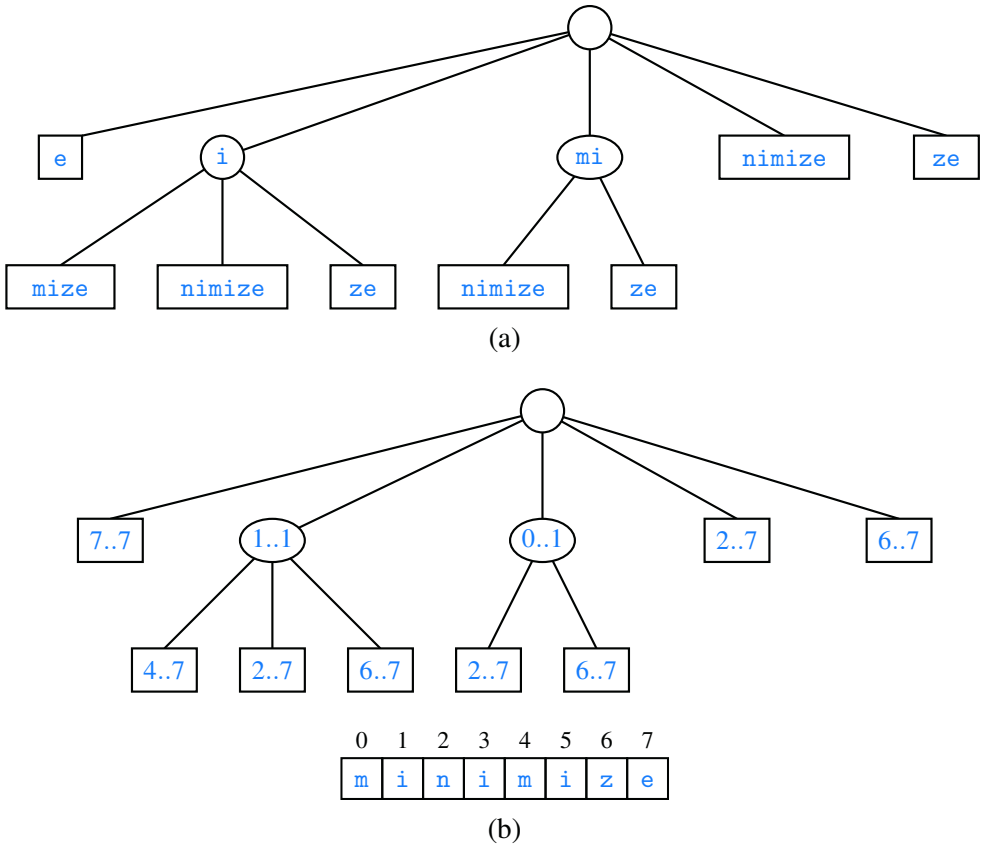


Figure 13.11: (a) Suffix trie T for the string $X =$ "minimize". (b) Compact representation of T, where pair $j..k$ denotes the substring $X[j..k]$ in the reference string.

Using a Suffix Trie

The suffix trie T for a string X can be used to efficiently perform pattern-matching queries on text X. Namely, we can determine whether a pattern is a substring of X by trying to trace a path associated with P in T. P is a substring of X if and only if such a path can be traced. The search down the trie T assumes that nodes in T store some additional information, with respect to the compact representation of the suffix trie:

> If node v has label j..k and Y is the string of length y associated with the path from the root to v (included), then $X[k-y+1..k] = Y$.

This property ensures that we can compute the start index of the pattern in the text when a match occurs in $O(m)$ time.

www.it-ebooks.info