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 "retrieval." 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.

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 11.6):

- 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 from 1 to d children, where d is the size of the alphabet. There is an edge going from the root r to one

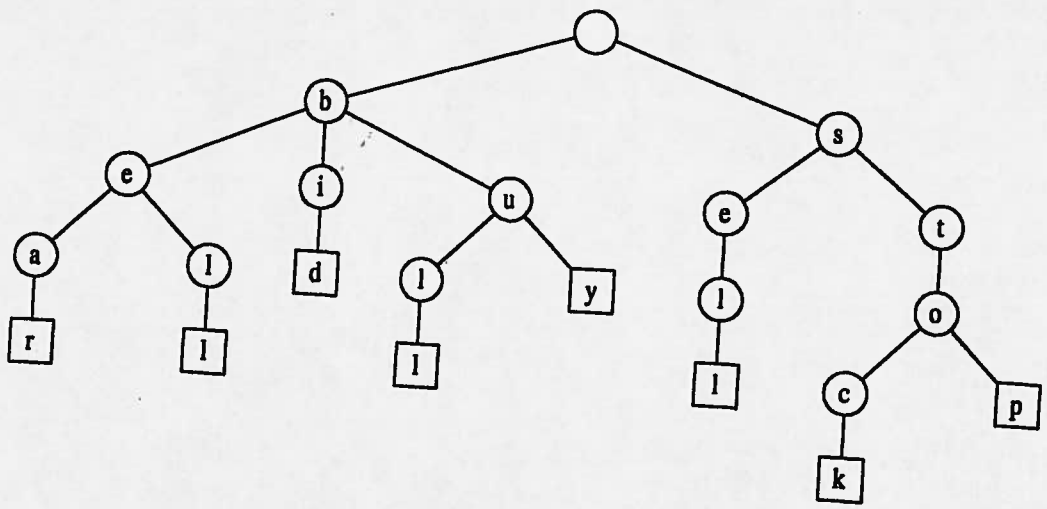


Figure 11.6: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

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 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, although some internal nodes may have 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 11.6 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 11.5: *A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:*

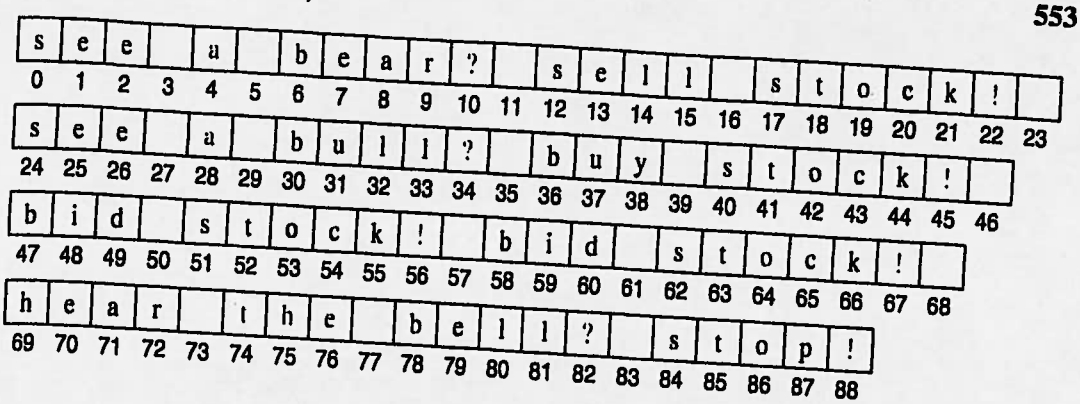
- *Every internal node of T has at most d children*
- *T has s external nodes*
- *The height of T is equal to the length of the longest string in S*
- *The number of nodes of T is $O(n)$.*

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.

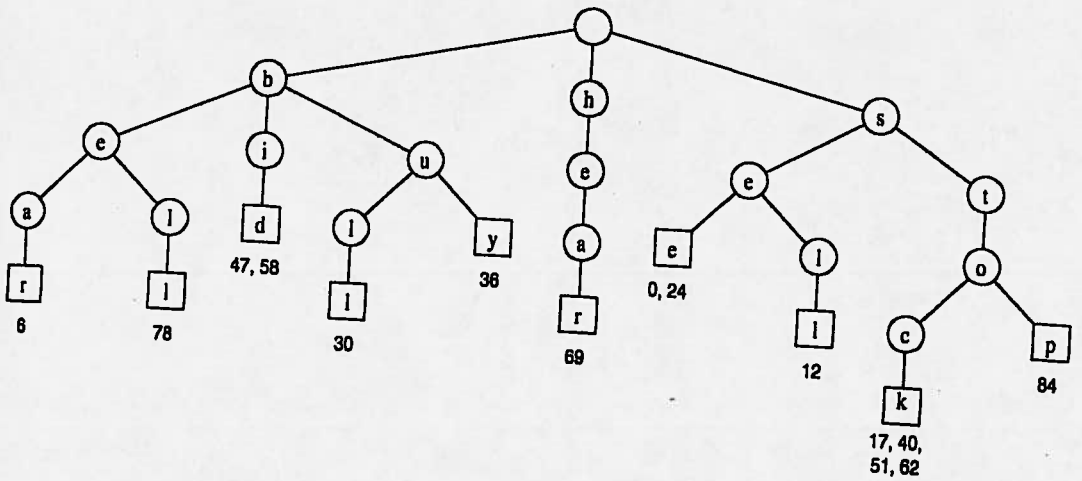
A trie T for a set S of strings can be used to implement a dictionary whose keys are the strings of S. Namely, we perform a search in T for a string X by tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at an external node, then we know X is in the dictionary. For example, in the trie in Figure 11.6, 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 11.6, the path for "bet" cannot be traced and the path for "be" ends at an internal node. Neither 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 look-up 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 above discussion, 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 11.7.) 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 11.7: 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.

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 11.8.) 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 11.6 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 $v_1, \ldots, v_k$.

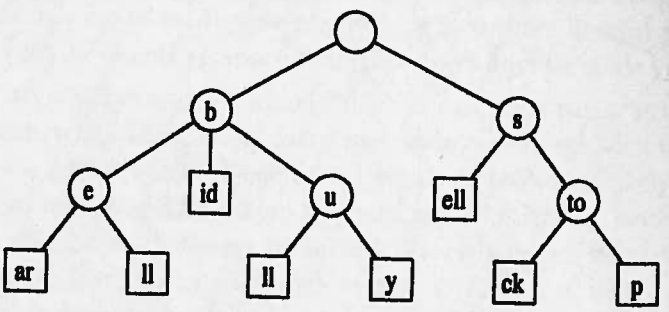


Figure 11.8: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Compare this with the standard trie shown in Figure 11.6.

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 11.5).

Proposition 11.6: *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 at most d children*
- *T has s external nodes*
- *The number of nodes of T is $O(s)$.*

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 triplet of 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 jth to the kth included. (See the example in Figure 11.9. Also compare with the standard trie of Figure 11.7.)

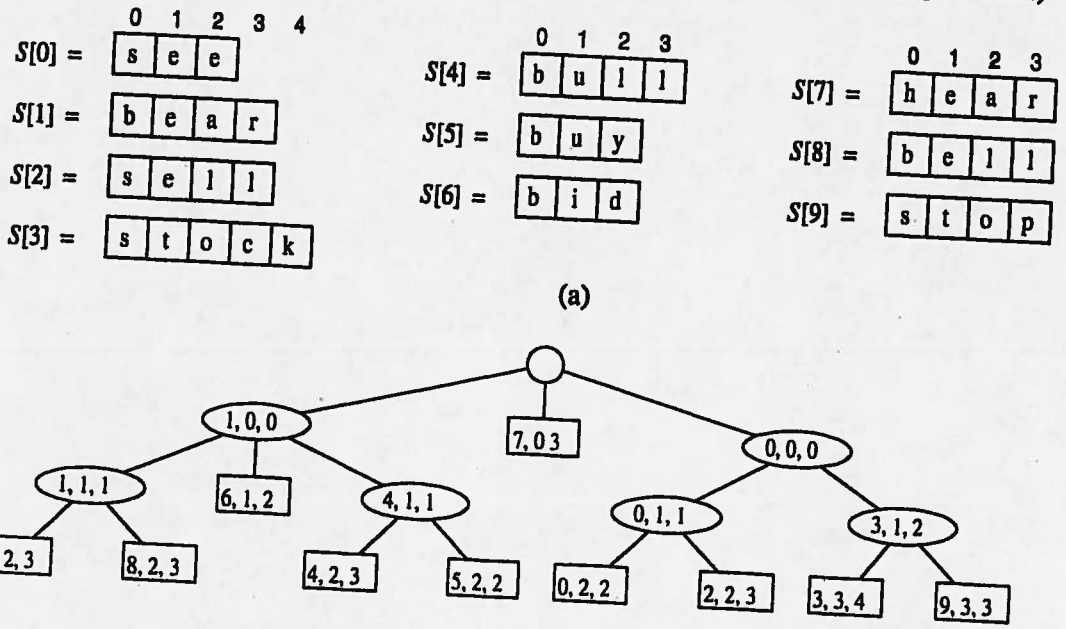


Figure 11.9: (a) Collection S of strings stored in an array. (b) Compact representation of the compressed trie for S.

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. In the next section, we present an application where the collection of strings can also be stored compactly.

11.3.3 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 11.10a 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 (i, j) indicating the string $X[i..j]$. (See Figure 11.10b.) 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 $X[i..n-1]\$$, for $i = 0, \ldots, n-1$.

Caution

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

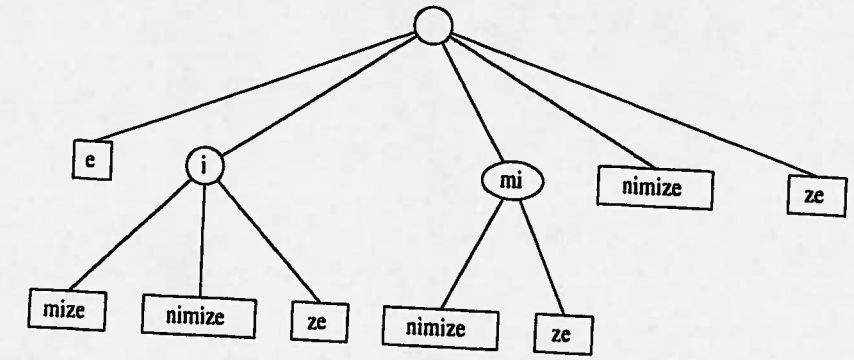
$$1+2+\cdots+n = \frac{n(n+1)}{2},$$

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.

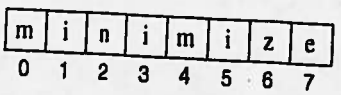
Proposition 11.7: *The compact representation of a suffix trie T for a string X of length n uses $O(n)$ space.*

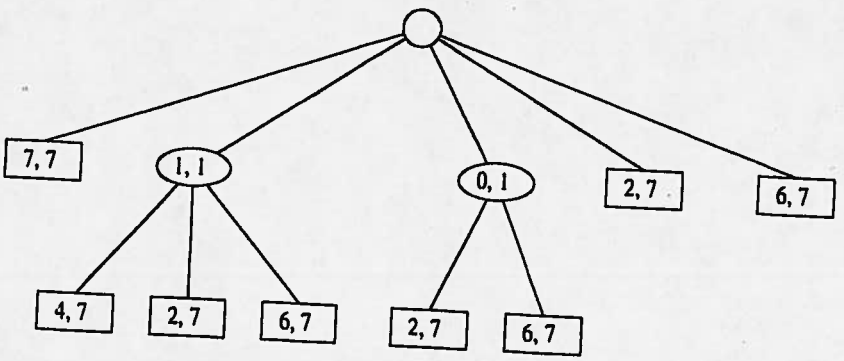
Construction

We can construct the suffix trie for a string of length n with an incremental algorithm like the one given in Section 11.3.1. This construction takes $O(dn^2)$ 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.



(a)





(b)

Figure 11.10: (a) Suffix trie T for the string $X =$ `"minimize"`. (b) Compact representation of T, where pair (i, j) denotes $X[i..j]$.

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 P 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 details of the pattern matching algorithm are given in Code Fragment 11.7, which assumes the following additional property on the labels of the nodes in the compact representation of the suffix trie:

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

This property ensures that we can easily compute the start index of the pattern in the text when a match occurs.

```
Algorithm suffixTrieMatch(T,P):
    Input: Compact suffix trie T for a text X and pattern P
    Output: Starting index of a substring of X matching P or an indication that P
      is not a substring of X
    p ← P.length() { length of suffix of the pattern to be matched }
    j ← 0 { start of suffix of the pattern to be matched }
    v ← T.root()
    repeat
      f ← true { flag indicating that no child was successfully processed }
      for each child w of v do
        i ← start(v)
        if P[j] = T[i] then
          { process child w }
          x ← end(w) − i + 1
          if p ≤ x then
            { suffix is shorter than or of the same length of the node label }
            if P[j..j+p−1] = X[i..i+p−1] then
              return i − j { match }
            else
              return "P is not a substring of X"
          else
            { suffix is longer than the node label }
            if P[j..j+x−1] = X[i..i+x−1] then
              p ← p − x { update suffix length }
              j ← j + x { update suffix start index }
              v ← w
              f ← false
              break out of the for loop
    until f or T.isExternal(v)
    return "P is not a substring of X"
```

Code Fragment 11.7: Pattern matching with a suffix trie. We denote the label of a node v with $(\mathsf{start}(v), \mathsf{end}(v))$, that is, the pair of indices specifying the substring of the text associated with v.

The correctness of algorithm suffixTrieMatch() follows from the fact that we search down the trie T, matching characters of the pattern P one at a time until one of the following events occurs:

- We completely match the pattern P
- We get a mismatch (caught by the termination of the for-loop without a break out)
- We are left with characters of P still to be matched after processing an external node.

Let m be the size of pattern P and d be the size of the alphabet. In order to determine the running time of algorithm suffixTrieMatch(), we make the following observations:

- We process at most $m+1$ nodes of the trie
- Each node processed has at most d children
- At each node v processed, we perform at most one character comparison for each child w of v to determine which child of v needs to be processed next (which may possibly be improved by using a fast dictionary to index the children of v)
- We perform at most m character comparisons overall in the processed nodes
- We spend $O(1)$ time for each character comparison.

Performance

We conclude that algorithm suffixTrieMatch() performs pattern matching queries in $O(dm)$ time (and would possibly run even faster if we used a dictionary to index children of nodes in the suffix trie). Note that the running time does not depend on the size of the text X. Also, the running time is linear in the size of the pattern, that is, it is $O(m)$, for a constant-size alphabet. Hence, suffix tries are suited for repetitive pattern matching applications, where a series of pattern matching queries is performed on a fixed text.

We summarize the results of this section in the following proposition.

Proposition 11.8: *Let X be a text string with n characters from an alphabet of size d. We can perform pattern matching queries on X in $O(dm)$ time, where m is the length of the pattern, with the suffix trie of X, which uses $O(n)$ space and can be constructed in $O(dn)$ time.*

We explore another application of tries in the next subsection.

11.4 Search Engines

The World Wide Web contains a huge collection of text documents (Web pages). Information about these pages are gathered by a program called a ***Web crawler***, which then stores this information in a special dictionary database. A Web ***search engine*** allows users to retrieve relevant information from this database, thereby identifying relevant pages on the Web containing given keywords. In this section, we present a simplified model of a search engine.

Inverted Files

The core information stored by a search engine is a dictionary, called an ***inverted index*** or ***inverted file***, storing key-value pairs (w, L), where w is a word and L is a collection of pages containing word w. The keys (words) in this dictionary are called ***index terms*** and should be a set of vocabulary entries and proper nouns as large as possible. The elements in this dictionary are called ***occurrence lists*** and should cover as many Web pages as possible.

We can efficiently implement an inverted index with a data structure consisting of:

1. An array storing the occurrence lists of the terms (in no particular order).
2. A compressed trie for the set of index terms, where each external node stores the index of the occurrence list of the associated term.

The reason for storing the occurrence lists outside the trie is to keep the size of the trie data structure sufficiently small to fit in internal memory. Instead, because of their large total size, the occurrence lists have to be stored on disk.

With our data structure, a query for a single keyword is similar to a word matching query (see Section 11.3.1). Namely, we find the keyword in the trie and we return the associated occurrence list.

When multiple keywords are given and the desired output are the pages containing ***all*** the given keywords, we retrieve the occurrence list of each keyword using the trie and return their intersection. To facilitate the intersection computation, each occurrence list should be implemented with a sequence sorted by address or with a dictionary (see, for example, the generic merge computation discussed in Section 10.2).

In addition to the basic task of returning a list of pages containing given keywords, search engines provide an important additional service by ***ranking*** the pages returned by relevance. Devising fast and accurate ranking algorithms for search engines is a major challenge for computer researchers and electronic commerce companies.