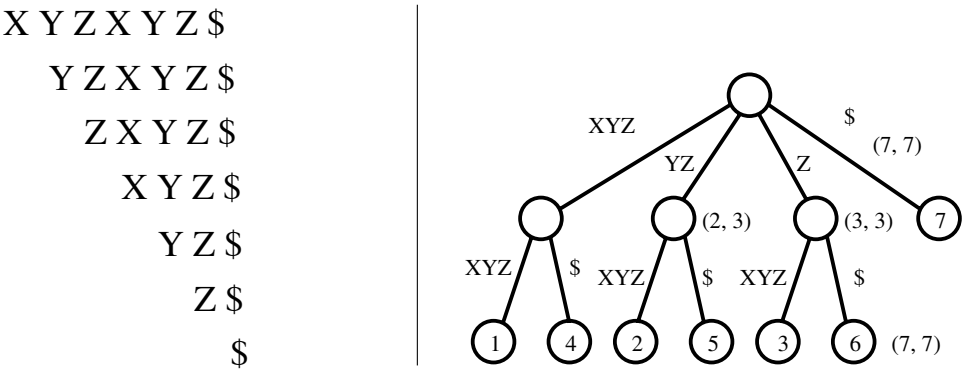

INPUT OUTPUT

12.3 Suffix Trees and Arrays

Input description: A reference string S.

Problem description: Build a data structure to quickly find all places where an arbitrary query string q occurs in S.

Discussion: Suffix trees and arrays are phenomenally useful data structures for solving string problems elegantly and efficiently. Proper use of suffix trees often speeds up string processing algorithms from $O(n^2)$ to linear time—likely the answer. Indeed, suffix trees are the hero of the war story reported in Section 3.9 (page 94).

In its simplest instantiation, a suffix tree is simply a *trie* of the n suffixes of an n-character string S. A trie is a tree structure, where each edge represents one character, and the root represents the null string. Thus, each path from the root represents a string, described by the characters labeling the edges traversed. Any finite set of words defines a trie, and two words with common prefixes branch off from each other at the first distinguishing character. Each leaf denotes the end of a string. Figure 12.1 illustrates a simple trie.

Tries are useful for testing whether a given query string q is in the set. We traverse the trie from the root along branches defined by successive characters of q. If a branch does not exist in the trie, then q cannot be in the set of strings. Otherwise we find q in $|q|$ character comparisons *regardless* of how many strings are in the trie. Tries are very simple to build (repeatedly insert new strings) and very fast to search, although they can be expensive in terms of memory.

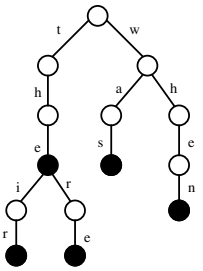

Figure 12.1: A trie on strings *the, their, there, was,* and *when*

A *suffix tree* is simply a trie of all the proper suffixes of S. The suffix tree enables you to test whether q is a substring of S, because any substring of S is the prefix of some suffix (got it?). The search time is again linear in the length of q.

The catch is that constructing a full suffix tree in this manner can require $O(n^2)$ time and, even worse, $O(n^2)$ space, since the average length of the n suffixes is $n/2$. However, linear space suffices to represent a full suffix tree, if we are clever. Observe that most of the nodes in a trie-based suffix tree occur on simple paths between branch nodes in the tree. Each of these simple paths corresponds to a substring of the original string. By storing the original string in an array and collapsing each such path into a single edge, we have all the information of the full suffix tree in only $O(n)$ space. The label for each edge is described by the starting and ending array indices representing the substring. The output figure for this section displays a collapsed suffix tree in all its glory.

Even better, there exist $O(n)$ algorithms to construct this collapsed tree, by making clever use of pointers to minimize construction time. These additional pointers can also be used to speed up many applications of suffix trees.

But what can you do with suffix trees? Consider the following applications. For more details see the books by Gusfield [Gus97] or Crochemore and Rytter [CR03]:

- *Find all occurrences of q as a substring of S* – Just as with a trie, we can walk from the root to the node n_q associated with q. The positions of all occurrences of q in S are represented by the descendents of n_q, which can be identified using a depth-first search from n_q. In collapsed suffix trees, it takes $O(|q| + k)$ time to find the k occurrences of q in S.

- *Longest substring common to a set of strings* – Build a single collapsed suffix tree containing all suffixes of all strings, with each leaf labeled with its original string. In the course of doing a depth-first search on this tree, we can label each node with both the length of its common prefix and the number of distinct strings that are children of it. From this information, the best node can be selected in linear time.

- *Find the longest palindrome in S* – A *palindrome* is a string that reads the same if the order of characters is reversed, such as *madam*. To find the longest palindrome in a string S, build a single suffix tree containing all suffixes of S and the reversal of S, with each leaf identified by its starting position. A palindrome is defined by any node in this tree that has forward and reversed children from the same position.

Since linear time suffix tree construction algorithms are nontrivial, I recommend using an existing implementation. Another good option is to use suffix arrays, discussed below.

Suffix arrays do most of what suffix trees do, while using roughly four times less memory. They are also easier to implement. A suffix array is in principle just an array that contains all the n suffixes of S in sorted order. Thus a binary search of this array for string q suffices to locate the prefix of a suffix that matches q, permitting an efficient substring search in $O(\lg n)$ string comparisons. With the addition of an index specifying the common prefix length of all bounding suffixes, only $\lg n + |q|$ *character* comparisons need be performed on any query, since we can identify the next character that must be tested in the binary search. For example, if the lower range of the search is *cowabunga* and the upper range is *cowslip*, all keys in between must share the same first three letters, so only the fourth character of any intermediate key must be tested against q. In practice, suffix arrays are typically as fast or faster to search than suffix trees.

Suffix arrays use less memory than suffix trees. Each suffix is represented completely by its unique starting position (from 1 to n) and read off as needed using a single reference copy of the input string.

Some care must be taken to construct suffix arrays efficiently, however, since there are $O(n^2)$ characters in the strings being sorted. One solution is to first build a suffix *tree*, then perform an in-order traversal of it to read the strings off in sorted order! However, recent breakthroughs have lead to space/time efficient algorithms for constructing suffix arrays directly.

Implementations: There now exist a wealth of suffix array implementations available. Indeed, all of the recent linear time construction algorithms have been implemented and benchmarked [PST07]. Schürmann and Stoye [SS07] provide an excellent C implementation at *http://bibiserv.techfak.uni-bielefeld.de/bpr/*.

No less than eight different C/C++ implementations of compressed text indexes appear at the *Pizza&Chili corpus* (*http://pizzachili.di.unipi.it/*). These data structures go to great lengths to minimize space usage, typically compressing the input string to near the empirical entropy while still achieving excellent query times!

Suffix tree implementations are also readily available. A `SuffixTree` class is provided in BioJava (*ttp://www.biojava.org/*)—an open source project providing a Java framework for processing biological data. `Libstree` is a C implementation of Ukkonen's algorithm, available at *http://www.icir.org/christian/libstree/*.

Nelson's C++ code [Nel96] is available from *http://marknelson.us/1996/08/01/suffix-trees/*.

Strmat is a collection of C programs implementing exact pattern matching algorithms in association with [Gus97], including an implementation of suffix trees. It is available at *http://www.cs.ucdavis.edu/~gusfield/strmat.html*.

Notes: Tries were first proposed by Fredkin [Fre62], the name coming from the central letters of the word "retrieval." A survey of basic trie data structures with extensive references appears in [GBY91].

Efficient algorithms for suffix tree construction are due to Weiner [Wei73], McCreight [McC76], and Ukkonen [Ukk92]. Good expositions on these algorithms include Crochmore and Rytter [CR03] and Gusfield [Gus97].

Suffix arrays were invented by Manber and Myers [MM93], although an equivalent idea called *Pat trees* due to Gonnet and Baeza-Yates appears in [GBY91]. Three teams independently emerged with linear-time suffix array algorithms in 2003 [KSPP03, KA03, KSB05], and progress has continued rapidly. See [PST07] for a recent survey covering all these developments.

Recent work has resulted in the development of compressed full text indexes that offer essentially all the power of suffix trees/arrays in a data structure whose size is proportional to the *compressed* text string. Makinen and Navarro [MN07] survey these remarkable data structures.

The power of suffix trees can be further augmented by using a data structure for computing the *least common ancestor (LCA)* of any pair of nodes x, y in a tree in constant time, after linear-time preprocessing of the tree. The original data structure due to Harel and Tarjan [HT84], has been progressively simplified by Schieber and Vishkin [SV88] and later Bender and Farach [BF00]. Expositions include Gusfield [Gus97]. The least common ancestor of two nodes in a suffix tree or trie defines the node representing the longest common prefix of the two associated strings. That we can answer such queries in constant time is amazing, and proves useful as a building block for many other algorithms.

Related Problems: String matching (see page 628), text compression (see page 637), longest common substring (see page 650).