Data Structures for Strings

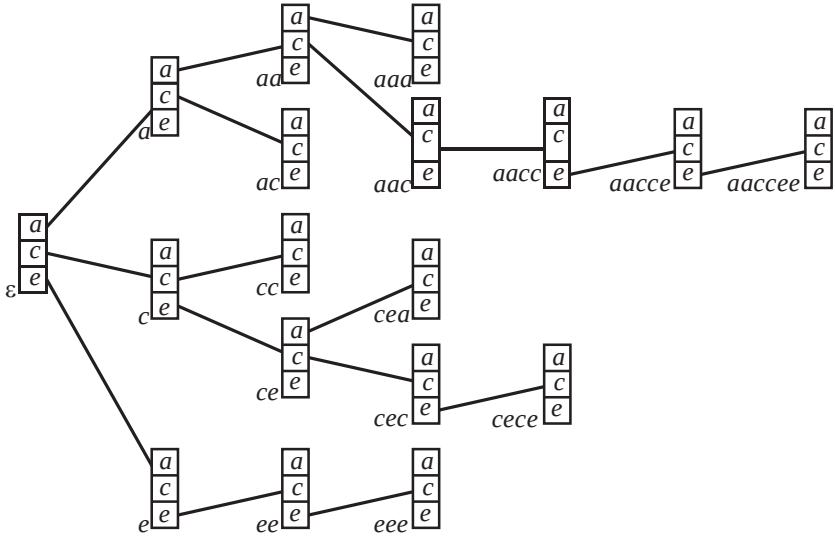
Up to now we always assumed that the data items are of constant size, and key values can be compared in constant time, so essentially that they are numbers. A very important class of objects for which these assumptions fail are strings. In real applications, text processing is more important than the processing of numbers.

We have an underlying alphabet A, for example, the ASCII codes, and strings are sequences of characters from this alphabet. But for use in the computer, we need an important further information: how to recognize where the string ends. There are two solutions for this: we can have an explicit termination character, which is added at the end of each string, but may not occur within the string, or we can store together with each string its length. The first solution is the `'\0'`-terminated strings used in the C language, and the other model is followed, for example, in the Pascal language and its descendants.

Tries and Compressed Tries

The basic tool for string data structures, similar in role to the balanced binary search tree, is called "trie," which is said to derive from "retrieval." This structure was invented by de la Briandais (1959); the first easily accessible reference, which also introduced this unfortunate name, is Fredkin (1961). The underlying idea is very simple – again a tree structure is used to store a set of strings. But in this tree, the nodes are not binary; instead, they contain potentially one outgoing edge for each possible character, so the degree is at most the alphabet size $|A|$. Each node in this tree structure corresponds to a prefix of some strings of the set; if the same prefix occurs several times, there is only one node to represent it. The root of the tree structure is the node corresponding to the empty prefix. The node corresponding to the prefix σ_1 contains for each character $a \in A$ a pointer to the node corresponding to the prefix $\sigma_1 a$ if such a node exists, that is, if there is a string $\sigma_1 a \sigma_2$ in the set.

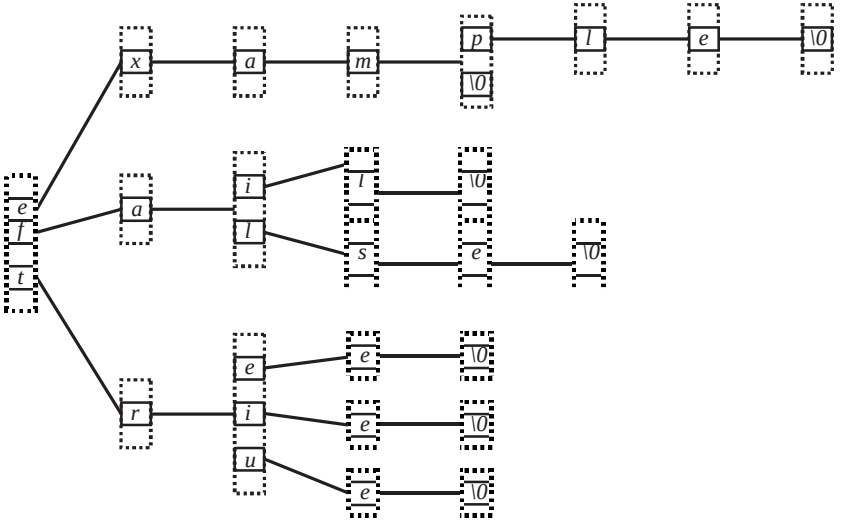
To perform a `find` operation in this structure, we start in the node corresponding to the empty prefix and then read the query string, following for each read character the outgoing pointer corresponding to that character to the next node. After we read the query string, we arrived at a node corresponding to that string as prefix. If the query string is contained in the set of strings stored in



TRIE OVER ALPHABET $\{a, c, e\}$ WITH NODES FOR THE WORDS *aaa*, *aaccee*, *ac*, *cc*, *cea*, *cece*, *eee*, AND THEIR PREfiXES

the trie, and that set is prefix-free, then this node belongs to that unique string. And we can assume that the set of strings is prefix free if we use the model of `'\0'`-terminated strings: if the character `'\0'` occurs only as termination character in the last position of each string, then no string can be a prefix of another string. With this assumption, we can now write the basic version of the trie structure. Each node has the following form:

```
typedef struct trie_n_t {
              struct trie_n_t   *next[256];
            /* possibly additional information*/
                       } trie_node_t;
```



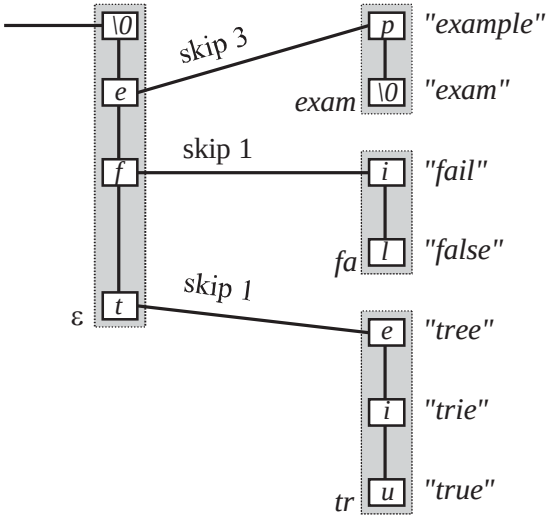
TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:*
IN EACH ARRAY NODE, ONLY THE USED FIELDS SHOWN

This structure looks very simple and extremely efficient; the one problem is the dependence on the size of the alphabet that determines the size of the nodes. In this basic implementation, each node contains 256 pointers, one for each character, and a pointer might be 4–8 bytes, so the size of each node is at least 1 kB. And, unless the strings we wish to store have very much overlap, we need approximately as many nodes as the total length of all strings together is: almost all nodes will contain only one valid pointer because almost all prefixes have only one possible continuation. So the space requirement is enormous.

A different type of tries is path compression, which is the idea that instead of explicitly storing nodes with just one outgoing edge, we skip these nodes and keep track of the number of skipped characters. So the path compressed trie contains only nodes with at least two outgoing edges, and together with each edge it contains a number, which is the number of characters that should be skipped before the next relevant character is looked at. This reduces the required number of nodes from the total length of all strings to the number words in our structure. But, as we skip all those intermediate nodes, we need in each access a second pass over the string to check all those skipped characters of the found string against the query string. This structure is known as **Patricia tree** (Morrison 1968), which is an acronym for “Practical

algorithm to retrieve information coded in alphanumeric." The idea of path compression can be combined with any of the aforementioned variants of tries; originally it was described for bit strings, but for a two-element alphabet the space overhead is so small that today there is no need for path compression;

this technique to reduce the number of nodes is justified only if the alphabet is large.

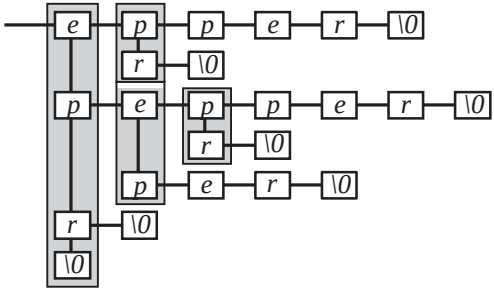


PATRICIA TREE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:* NODES IMPLEMENTED AS LISTS; EACH LEAF CONTAINS ENTIRE STRING

As a static data structure, the Patricia tree seems to be quite straightforward, but the insertion and deletion operations create significant difficulties. To insert a new string, we need to find where to insert a new branching node, but this requires that we know the skipped characters. It seems an obvious solution to attach to each node the skipped substring that led to it, but then we have to allocate and deallocate many small strings of varying sizes; even if we group them in a few standard sizes, this is a procedure with high overhead.

Suffix Trees

The suffix tree is a static structure that preprocesses a long string *s* and answers for a query string *q*, if and where it occurs in the long string. Thus, it solves the substring matching problem, as do the classical string-matching algorithms. The difference is that the time to answer a substring query is not dependent on the length of the long string, but only on the length of the query string. The query time is $O(\text{length}(q))$ for a query string *q*. The idea is very simple at least on the query side: each substring of *s* is prefix of a suffix of *s*, and the nodes of any trie correspond to the prefixes of the strings stored in the trie, so if we construct a trie that stores all suffixes of the long string *s*, then its nodes correspond to the substrings of *s*, and we can decide for any query *q* in $O(\text{length}(q))$ whether it is a substring of *s*.



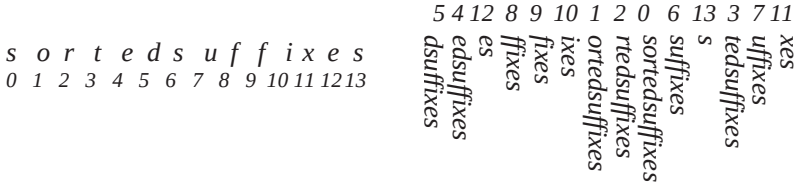
TRIE of THE SUFFIXES of *pepper*

Because any realization of the suffix tree has a trie as underlying structure, the space requirements of tries, especially for large alphabets, are also a problem for suffix trees.

Suffix Arrays

It supports the same operations as the suffix tree: it preprocesses a long string and then answers for a query string whether it occurs as substring in the preprocessed string. The possible advantage of the suffix array structure is that its size does not depend on the size of the alphabet and that it offers a quite different tool to attack the same type of string problems. It is said to be smaller than suffix trees, but that somewhat depends on various compact encoding tricks; in its most straightforward implementation, it requires three integers per character of the long string, whereas an implementation of the suffix tree with list nodes requires five pointers per node, and the number of nodes is at most the length of the string, but possibly smaller.

The underlying idea of the suffix array structure is to consider all suffixes of the preprocessed string s in lexicographic order and perform binary search on them to find a given query string. This already shows one disadvantage of the structure: the query time to find a string q in the long string s alsodepends on length(s); to find the right one among the length(s) possible suffixes, we need O(log length(s)) lexicographic comparisons between q and some suffix of s. Without additional information, each comparison takes O(length(q)) time for a total of O(length(q) log length(s)); if some additional information on the length of common prefixes of the suffixes of s is available, this reduces to O(length(q) + log length(s)). The suffix tree needs only O(length(q)) query time, independent of s.



THE SUFFIXES of *sortedsuffixes* IN LEXICOGRAPHIC ORDER WITH THEIR STARTING INDICES IN THE STRING

We need to represent the suffixes of the string s in a way that they are sorted in lexicographic order and we can perform binary search on them. The most natural way is to have one big array in which the starting indices of the lexicographically sorted suffixes are stored. So we need an integer array of the same length of the string. This is another disadvantage. There might be a problem of allocating an array of length(s) integers if length(s) is very large. And for the common prefix information, we need another two such arrays. The structure does not fit into our pointer-machine model, which allows only fixed-size arrays.