CS340 Lecture Notes, Lecture 12A

Tries and digital search trees

Reference: Denenberg and Lewis, Data Structures and Their Algorithms, Section 8.2

Goal: Store data where the keys are represented by character strings.

In a binary search tree, a two-way decision is made: is the key less than or greater than the current node. However, when character data is stored, one can perform multi-way branching based on the character. Such a structure is called a trie (pronounced "try", based on the middle syllable of "retrieval").

Tries were invented by E. Fredkin in 1960.

If there are k possible character values, then a trie is a $(k+1)$-ary tree. Each node is an array of $k+1$ pointers:

one for each possible character, and one for an "end-of-string" ($\triangle$) indicator. The root node arranges keys according to their first character of the key; each key beginning with character numbered i goes in the i'th subtree. At the second level of the tree, one deals with the second character of the key, and so on. If a key has exactly m characters, then the search path for that key ends in a node at depth m at the $\triangle$ indicator. You can also store at the leaf any information associated with the key.

Example: A trie containing the following keys: MAXWELL, MENDEL, MENDELEEV, PASTEUR, PAVLOV, PEANO, POINCARE, POISSON, TURING.

Advantage: a key can be looked up in $O(t)$ time, where t is the length of the longest string stored in the trie.

Disadvantage: if the trie has s nodes, then $s(k+1)$ pointers must be stored, independent of the number of keys. This can be very high. Example, if you store just one word "IMPOSSIBILITY" in a trie, it will use $13 \times 27 = 351$ pointers, all but 13 of which are nil pointers.

There are many ways around this problem of excessive storage utilization.

1. Store the trie as an $(s \times k+1)$-array of numbers. Each row represents a node. In position i of row s is stored the row number that represents the child, or 0. Now the number of bits of storage is governed by the number of bits it takes to represent s, the number of nodes. In general, this is much less than the number of bits needed to store a pointer (typically 32 bits or even more). Thus, the table can be packed.

2. de la Briandais trees. Instead of using arrays to represent nodes, one can use a linked list of character/pointer pairs. One might want to use a move-to-front strategy to minimize access time.

3. **Patricia trees**

Patricia stands for "Practical algorithm to retrieve information coded in alphanumeric", invented by D. R. Morrison, 1968.

The idea is to take a trie and get rid of any nodes that only have one child. Instead, each remaining node is labeled with a character *position number* which would have given that node's depth in the original uncompressed trie. We now have the problem that keys are no longer uniquely specified by the search path, so we have to store the key itself in the appropriate leaf. The storage requirement is now kn pointers, where n is the number of keys and k is the size of the alphabet. This is often significantly less than $s(k+1)$, particularly if the keys are very long.

The New OED project here at Waterloo was originally based on Patricia trees, now it is combined with the suffix array, a more advanced data structure. (Note, Patricia tree alone is far from efficient: OED too large, must minimize number of disk accesses.)

4. One last approach is *digital search trees*, treat a key as binary string.

Invented by Coffman and Eve in 1970.

Here we take a string of letters and code it over an alphabet of size $k = 2$, for example, A = 00001, B = 00010, ..., Z = 11010. We construct a binary tree where left = 0, right = 1, and in each node of the search path so determined we store keys for which the prefix might be a match. Digital search trees have the advantage that they can grow dynamically, and deletions can also be supported. The expected search time in such a tree with n keys is $O(\log n)$, assuming the keys are reasonably distributed.