15-451: Algorithms

Lecture 9: Radix Structures (notes KS)

1 Radix Sort

Recall our lower bound on sorting: $\Omega(n \lg n)$ steps on average to sort input of length n. This bound is predicated on the assumption that only comparisons $a_i < a_j$ are possible, but that the internal structure of the date items a_i cannot be used in any queries.

Clearly, this assumption fails when one is trying to sort numbers. Suppose we have to sort k-digit numbers in base B:

$$a = d_k d_{k-1} \dots d_2 d_1.$$

Radix sort algorithms work by examining single digits in the keys: we have a function $digit(a, i) = a_i$ that returns the ith digit.

1.1 Radix Exchange (left to right)

The obvious approach is to first sort the elements according to their most significant digit, then the second most significant digit, and so on. This is intuitively appealing, but the lists keep braking up into sublists, and bookkeeping becomes a problem. Recursion can be used to handle this. Starting at $i = k$ do

- Place items into B buckets, a goes into $Q(digit(a, i))$.
- Recursively sort each (non-empty) bucket according to digit $i - 1$.
- Concatenate the buckets.

The relative order has to be maintained (much as using dequeue/enqueue operations). The non-recursive part takes $O(n + B)$ steps, hence the whole algorithm is $O(k(n + B))$, which is linear in the total number of digits.

1.2 Straight Radix

Straight radix sort is a bit counter-intuitive: it examines the digits from right to left, starting at the LSD. However, this turns out to be much easier to implement. Abstractly, we sort from LSD to MSD, using a stable sorting algorithm for each digit. Note that use of a stable algorithm is crucial here.

```
for( i = 1; i <= k; i++ )
   sort according to digit i
```

Concretely, the stable sort is usually counting sort. The running time is again $O(k(n + B)$, and thus linear in the total number of digits. Note, though, that the space requirement is $2kn + O(1)$ since the stable sort is not in place.

Since numbers are usually written in binary there is some freedom in the choice of k and B. Suppose we are dealing with 32-bit integers. Allows for the following choices:

buckets	passes
2	k
8	$k/3$
16	$k/4$
2^k	1

2 Tries

A **trie** or **digital tree** is a dictionary data structure that exploits the digital decomposition of the data items, typically words or numbers. In general, a trie stores a finite set of words, where a word is any finite sequence of symbols from some fixed alphabet of size m, the order of the trie. In a trie

- a leaf contains no data,
- an internal node contains an array of pointers of size m.

Thus, we can think of the edges in the trie as being labeled by the alphabet.

The most important application of tries is for word dictionaries. Since one word may be a prefix of another, we have to have an additional bit in each node indicating full words. Note that the trie naturally stores words in lexicographic order. A trie supports not just search, insert and delete, but also finding all words of a given prefix. Indeed, we can list all entries that are extensions of a given prefix in lexicographic order in time linear in the total number of symbols involved. Operations search, insert and delete all in time $O(l)$ where l is the length of the word in question.

If we use 7-bit ASCII, the internal arrays are quite large, and may be very inefficient since usage distribtution may very uneven. This is true in particular for nodes close to leaves (good at the root, bad at the leaves).

Solutions:

(1) Store two arrays of length r, where r is the number of children of the node. First array holds labels of the r out-edges, second array holds the actual pointers.

(2) Use a hash function $hash(i, key)$ that returns some nicely distributed index value, instead of just the ith character of the word. For example, when storing plain English words, this may lead to a better distribution (letter frequencies in English are very uneven). The same hash function is used for insert, search and delete.

2.1 Path Compression

Another way to save space in a trie is to eliminate all nodes of out-degree 1 in the tree. We collapse a non-branching segment of the tree, and store the information at the parent node. Using path compression we can get bounds on the size of a random trie.

For the number of nodes N_n and the height h_n of a trie of order m storing n items (assuming uniform distribution of the digits) we have

$$E[N_n] \approx \frac{n}{\ln m} + O(1)$$
$$E[h_n] = 2\log_m n + o(\lg n)$$

Example: $n = 50000$, $m = 2$: $E[N_n] \approx 72,133$, $E[h_n] \approx 30.3$.

$n = 50000$, $m = 10$: $E[N_n] \approx 21074$, $E[h_n] \approx 9.5$.

2.2 Application: Lempel-Ziv-Welch Compression

In LZW we need to maintain a dictionary with entries $(string, integer)$. Moreover, we need to find the longest prefix $a_1 a_2 \ldots a_r$ of a given lists of symbols that is still in the dictionary. We then have to insert a new element with key $a_1 a_2 \ldots a_r a_{r+1}$. In a trie, this operation is $\Theta(r)$.

2.3 Application: Suffix Trees

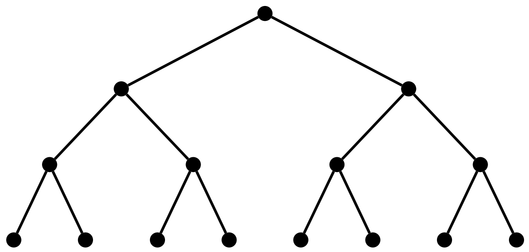
A suffix tree is a trie that stores all the suffixes of a given word. Arguably, suffix trees are the single-most important data structure in string matching.

Note that a priori a suffix tree is only $O(n^2)$ if the length of the word is n. However, by using path compression (and pointers to positions in the original string) one can reduce the size requirement to $O(n)$. Moreover, such a suffix tree can be constructed in time $O(n)$.

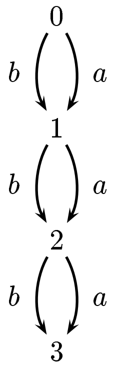
3 DAWGs

A DAWG is a directed acyclic word graph. A trie is an example of a DAWG, but a very limited one: all nodes other than the root have indegree 1. More efficient representations of finite sets of words can be found by admitting arbitrary DAGs (directed acyclic graphs).

For example, the trie for $\{a,b\}^3$ over alphabet $\{a,b\}$ has the form



But there clearly is a better representation: we can simply have 4 nodes in a row, and a double edge from one to the next, labeled by a and b, respectively.



Note, though, that insertions and deletions become more complicated in this setting. One can show that they can still be handled in time linear in the length of the word in question.

Incidentally, the smallest DAWG for all suffixes of a given word has at most $2n$ nodes, but the proof requires some automata theory.

Note: The name trie derives from "re*trie*val", and was originally pronounced like "tree". Since that leads to endless confusions with ordinary trees, it is now proper etiquette to use the pronunciation "try".