Litterature

At the lecture today we will talk about string *indexing* using tries. You should read the lecture notes about tries (section 1-3).

Exercises

1 Compact tries [*w*] Construct the compact trie for the words: "freedom, trie, tree, try, tire, car, cat, fire, free". You don't have to replace the labels by indexes into the strings.

Illustrate how a prefix search for "ca" works.

2 Search Engine Your friends have a company "Shoes, Sandals and Socks". They would like a search engine where it is possible to search for their products. Often the customer only remember the beginning of the name of the product, so they would like the following feature: If you search for a product with name s (where s is a string), then the search engine return all products that begins with s.

Give a data structure that solves this problem and analyze the space and query time.

3 Database of Recipes You want a data structure that allows you to search in your recipes. It should support the following operations:

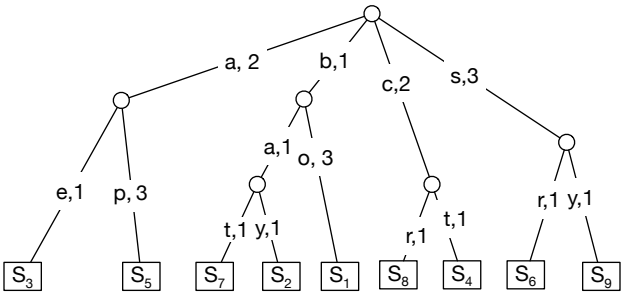
- Search(P): Return a list of all recipes that contain the word P.
- Update(R): Insert a new recipe.

The list returned by the searches should not contain the same recipe more than once. Give a data structure that solves this problem and analyze the space and query time.

4 Time for construction of tries Let S be a set of k strings of total length N over an alphabet of size σ. How much time does it take to construct the trie using the following data structures for navigating edges in the nodes in the trie:

- Hashtable with universal hashing.
- Balanced search trees.
- Sorted arrays.
- Linked lists.

5 Blind Trie A blind trie over a set of strings is a compact trie, where instead of indexes into the string, we store for each edge the first letter of the label of the edge and the length of the label. Here is the blind trie for the set strings $S = \{$`boat, bay, ape, cat, apple, star, bat, car, stay`$\}$.



5.1 [*w*] Draw the blind trie for the set of strings from Exercise 1.

5.2 We search after a pattern P in a blind trie almost as in the compact trie. Start from the root and follow edges according to the next character on the edge to the corresponding character in P. Let α be the first character in P. Assume we follow an edge from the root to some node v labeled $(\alpha, 5)$. Which character in P should we then compare to next, to find the correct edge out of v to continue the match?

5.3 Argue that if there is a string in S that is equal to P then we find it doing the search as described above.

5.4 We might end in a leaf corresponding to a string $S_i \neq P$ when matching in the blind trie. Why? Explain how to check if $S_i = P$. How much time does this take?

6 Multiple Pattern Matching using the Aho-Corasick Automaton In the multiple pattern matching problem we are given a set of patterns $P = \{P_1, \ldots, P_k\}$ of total length $\sum_{i=1}^{k} m_i = m$ and a string S of length n from an alphabet of size σ. The problem is to return all ending positions of occurrences of patterns from P in S. That is, return the endpoint of all occurrences of P_i in S for all $i = 1, \ldots, k$.

6.1 Solve the multiple string matching problem of the following instance: $P = \{\texttt{abac}, \texttt{abba}, \texttt{aca}, \texttt{bbaa}, \texttt{ba}\}$ and $S = \texttt{abcaabacabbaabbacbabbacabbabbabaccc}$.

6.2 Explain how to use KMP to solve the multiple pattern matching problem. What is the running time of the algorithm? Run your algorithm on the example.

6.3 Draw the (uncompacted) trie T_P of the patterns P from the example and direct all edges away from the root.

6.4 For a node $v \in T_P$, let $s(v)$ be the string corresponding to v (the root corresponds to the empty string). For each node v in the trie T_P, find the node u in the trie such that $s(u)$ is the longest proper suffix of $s(v)$ among all nodes in the trie. Create a failure link from v to u. Mark all nodes that corresponds to a pattern.

6.5 We will use a matching algorithm similar to the one in KMP. We start in the root and process each character in S one by one from left to right following edges and failure links. Argue that if we are in node v after processing $S[1 \ldots i]$ then $s(v)$ is the longest suffix of $S[1 \ldots i]$ matching a prefix of a string in P.

6.6 Just returning the position in S each time we reach a node that is marked does not always return the ending position of *all* occurrences. Explain why.

6.7 Add *matching links* from each node the following way. From each node $v \in T_P$, create a matching link to the *marked* node u such that $s(u)$ is the longest proper suffix of $s(v)$ among all the marked nodes (if such a node exists).

6.8 We add the following to the matching algorithm. After processing a character from S we do the following. Let v be the node we are in after processing the character. Follow the matching link from v if it exist, return the matched node that it points to, and continue recursively following matching link and reporting occurrences as long as the node we are in have a matching link. Then we return to node v and are ready to process the next character in S.

Try running the algorithm on the text S from exercise 6.1.

The automaton constructed in the exercise is called the Aho-Corasick automaton. The automaton can be constructed in $O(m)$ time and the total time to solve the multiple pattern matching problem is then $O(m + n + \text{occ})$, where occ is the number of occurrences of patterns from P in S.