CS 7800/4810: Data Str & Alg Scalable Comp Spring 2026

Lecture 6 — Jan 28, 2026

Prof. Prashant Pandey *Scribe: Zikun Wang*

1 Overview

In this lecture we talked about the string matching problem, and the corresponding data structures to solve it.

Goal: Given a text T and pattern P (both are strings over a given alphabet), fins some/all occurances of P in T. We target $O(|P|)$ for query time complexity and $O(|T|)$ space complexity.

2 Naive solution: bruteforce

Look at every possible position of P in T (sliding window with length of $|P|$ and offset of 1). Time complexity is $O(|P| \cdot |T|)$.

3 Alternative solution: rolling hash

Intuitively, each adjacent slinding window is differ from each other only by the first letter and the second letter. We could use a hash function that takes $O(|P|)$ time on the initial window, and only spend $O(1)$ constant time updating it for each subsequent window. We are going to discuss this more in future lectures.

Related reading R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," in IBM Journal of Research and Development, vol. 31, no. 2, pp. 249-260, March 1987, doi: 10.1147/rd.312.0249.

4 Predecessor queries among strings using trie.

Goal: Given k strings $\{T_1, T_2, ..., T_k\}$, find if P is a prefix for any string in the list.

Trie is a rooted tree with child branch labels from $\sum$ (alphabet).

- Strings are represented as root-to-leaf paths.
- Add the symbol \$ to mark the end of each string; otherwise prefixes cannot be distinguished.

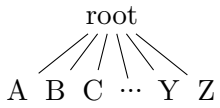


Figure 1: An example trie. Note the number of children is bounded by $\sum$(alphabet).

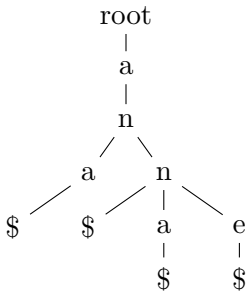


Figure 2: Example prefix trie.

Example. Given a word list $\{\texttt{ana}, \texttt{ann}, \texttt{anna}, \texttt{anne}\}$, where $k = 4$.

In practice, we can construct the trie using hash tables for each node's children list.

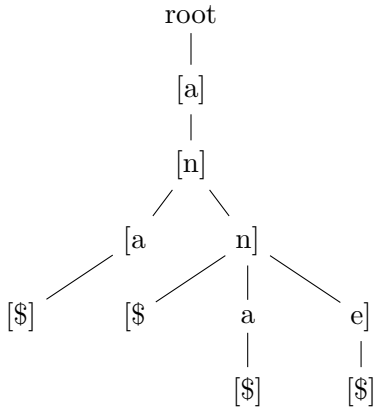


Figure 3: Example prefix trie with hash table, where each pair of brackets correspond to a hash table.

5 Trie representation

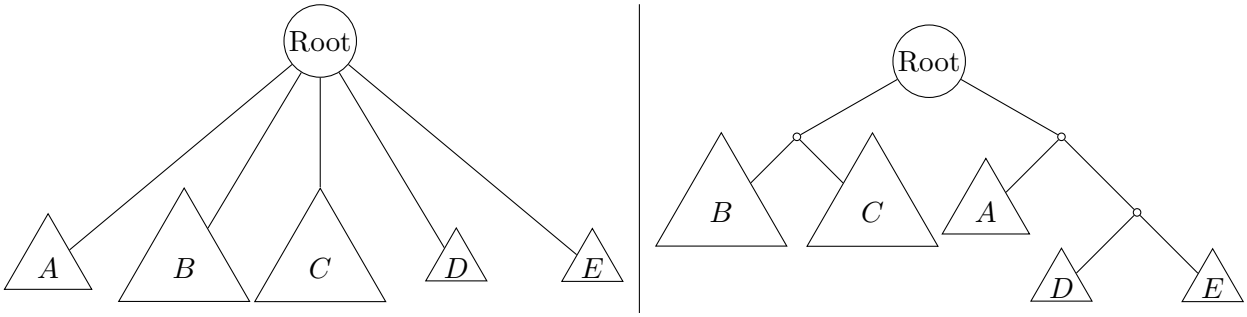
Let's define T as

$$T = \#\text{ nodes in trie} \leq \sum_{i=1}^{k} |T_i|$$

Data type	Query time complexity	Space complexity
Array	$O(P)$	$O(T\|\Sigma\|)$
BST	$O(P \cdot \log \|\Sigma\|)$	$O(T)$
Hash table	$O(P)$	$O(T)$
vEB/y-fast	$O(P \cdot \log\log \|\Sigma\|)$	$O(T)$
Weight balance BST	$O(P + \log k)$	$O(T)$
Leaf trimming	$O(P + \log \|\Sigma\|)$	$O(T)$
vEB (only when you fall off)	$O(P + \log\log \|\Sigma\|)$	$O(T)$
String sorting	$O(P \cdot \log k)$	$O(T)$

- Weight balance BST is good when k is small.
- Leaf trimming and vEB tree (only when fall off) is good when $|\Sigma|$ is small.

Weight balanced BST Given a BST, we balance the tree based on the number of descendant leaves, optimizing the average cost over the tree traversal. We place the larger subtrees closer to the root. This data structure guarantees that between every two edges you follow, you either: Advance by 1 letter OR reduce the number of candidate T_i by $\frac{2}{3}$.



Compressed trie We can reduce the amount of node in a trie by contract all non-branching path to a single edge. Using example from Figure 2:

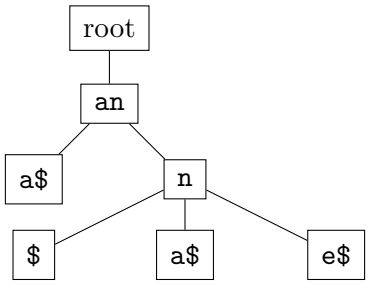


Figure 4: Compressed prefix trie.

6 Suffix trees

Goal Now we want to find some/all occurrences of P in T. We will do so by creating a compressed trie of the text T for all possible suffixes in T. We can formally write this set as:

$T[i:]$ of T for $i \in [0, |T|]$ with \$ appended to T

This tree will have

- $|T| + 1$ leaves.
- each edge label as T's substring $T[i:j]$ store as $[i, j]$.
- space complexity of $O(T)$.

Example Take the word `banana`, we append \$ sign to the end and construct a compressed trie on all the possible suffixes.

idx	substring
0	banana$
1	anana$
2	nana$
3	ana$
4	na$
5	a$
6	$

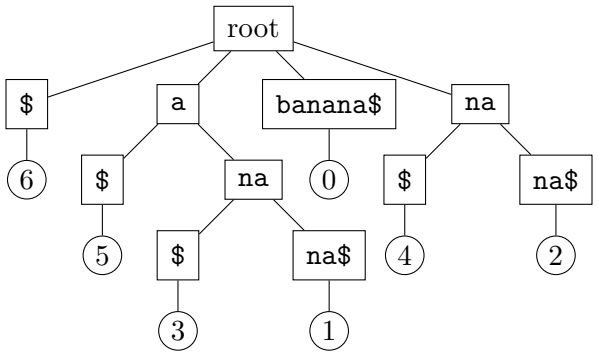


Figure 5: Compressed suffix trie on the word `banana`.

Application

- Search for P gives sub-tree whose level corresponds to all occurrences of P ($O(l)$ time complexity).
- List first k occurrences in $O(k)$.
- Every node corresponds to left most descend leaf.
- We can connect leaves via a linked list.
- We can calculate the number of P's occurrences in $O(1)$ time by counting the sub-tree size.

- Similarly, we can find the longest repeated sub-string in T by searching for branching node of maximum letter depth. ($O(T)$ time complexity)