CS 5968: Data Str & Alg Scalable Comp	Spring 2023
Lecture 5 — Jan. 25, 2023	
Prof. Prashant Pandey	*Scribe: Aaron Schindler*

1 Overview

1.1 Logistics

- Assignment is due next Wednesday February 1
- There are 2 guest lectures coming up, one on LSH (Feb 15) and one on ANN (March 1)
- The scribe schedule is now up to date and can be found on the course website
- If you did not send your name for scribbing, you were auto-assigned

In the last lecture we finished up vEB-Trees, and went over succinct data structures.

In this lecture we will go over efficient data structures that deal with strings (tries, compact tries, suffix trees).

2 Tries, Compact Tries, Suffix Tree

2.1 A quick background on a current problem

A current problem in the field of genomics is gene assembly. The process of reading in genes is called **reads**. Current instruments can only read in limited amounts of information at a time. There are two types of reads:

1. *short reads* – instrument can read $150 - 200$ base pairs at a time
2. *long reads* – instrument can read $\approx 10{,}000$ base pairs at a time

For reference, each person has approximately 3 billion base pairs. In addition to the massive scale of information being handled at a time, the instruments that take biological samples do not read accurately. There are three main types of errors:

1. *Insert* – reading an extra pair
2. *Delete* – removing/skipping a pair
3. *Substitute* – reading wrong pair

2.2 String matching

Problem: Given text **T** and pattern **P** that are strings built from alphabet Σ, find **some** or **all** occurrences of $\mathbf{P} \in \mathbf{T}$.

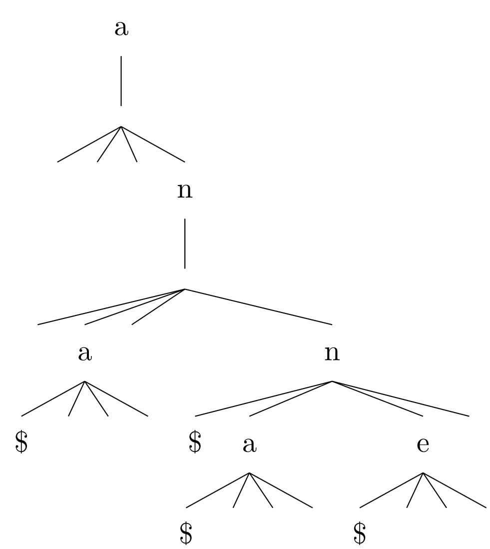
Naive solution takes $O(PT)$ time for each P.

GOAL: Get $O(P)$ query time and $O(T)$ space complexity

2.3 Warm up: Find predecessor among strings

- Given strings $(T_1, T_2, \ldots T_k)$
- Build rooted trie with child branches labeled with letters in Σ
- Represent strings as root-to-leaf paths in the trie
- Add a new letter \$ to the end of each string

Example: $\Sigma = \{a, e, n, \$\}$, Commands = $\{\text{ana}, \text{ann}, \text{anna}, \text{anne}\}$ Here is the trie built for this example:



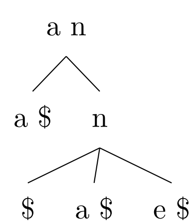
2.4 Trie Representation

Representation	Query Cost	Space Complexity
Array	$O(P)$	$O(T\Sigma)$
Balanced Search Tree	$O(P \lg \Sigma)$	$O(T)$
Hash Table*	$O(P)$	$O(T)$
VEB/Y-Fast Tree	$O(P \lg \lg \Sigma)$	$O(T)$
Trays**	$O(P + \lg \Sigma)$	$O(T)$

Table 1: Trie representations and their space/query complexities. *Cannot do predecessor queries. **Can do predecessor queries

2.5 Compact Tries

A **compact** or **compressed trie** cuts out nodes that are useless, (i.e. if only one path branches, then there is no need to create the other branches). A representation of a compact trie can be derived from the above example with the command set $\{\text{ana}, \text{ann}, \text{anna}, \text{anne}\}$. The resulting compact trie is displayed below:



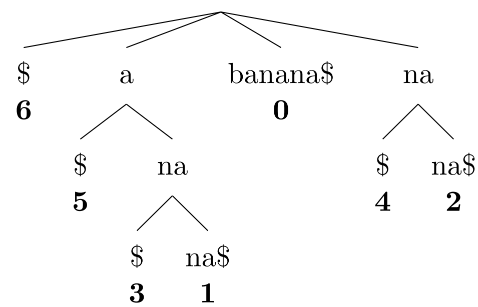
2.6 Suffix Trees

A **Suffix Tree** is composed of compact tries of size $|T|$ suffixes where $T[i{:}]$ of T denotes the branches.

**Example: banana$** Given the word banana$ we can build 7 different suffixes:

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

To build the suffix tree, we build the tree based on suffix that starts with letter i. If two or more unique suffixes share the same i^{th} letter, then a branch is created. An example of the suffix tree is given below:



Note that the bold numbers on each leaf in the tree correspond to the suffix listings given in the list above

Using the suffix tree we can achieve a query time of $O(P)$ and a space complexity of $O(T)$.