Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Preprocess Pattern or Preoprocess Text?
We have discussed the following algorithms in the previous posts:

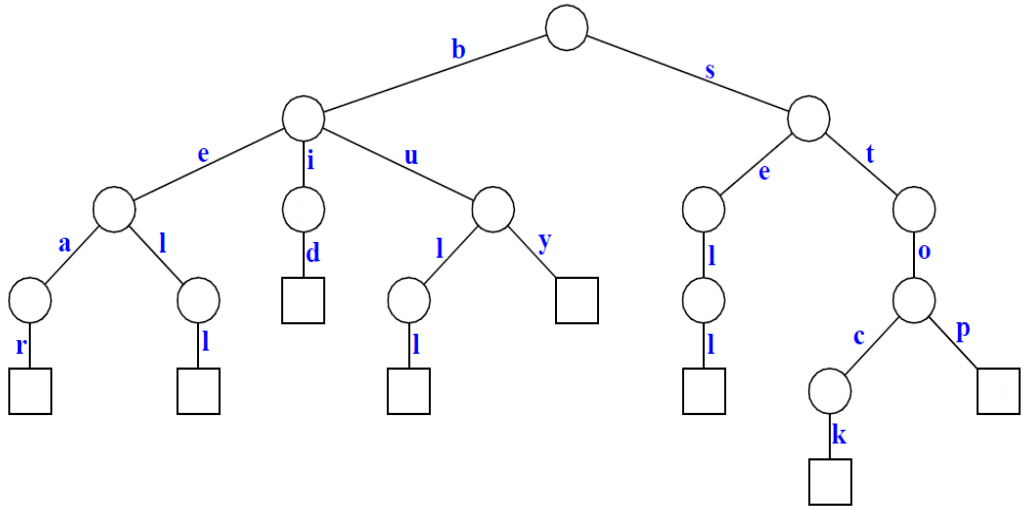
KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm
Boyer Moore Algorithm

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is O(n) where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in O(m) time where m is length of the pattern.
Imagine you have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.
Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

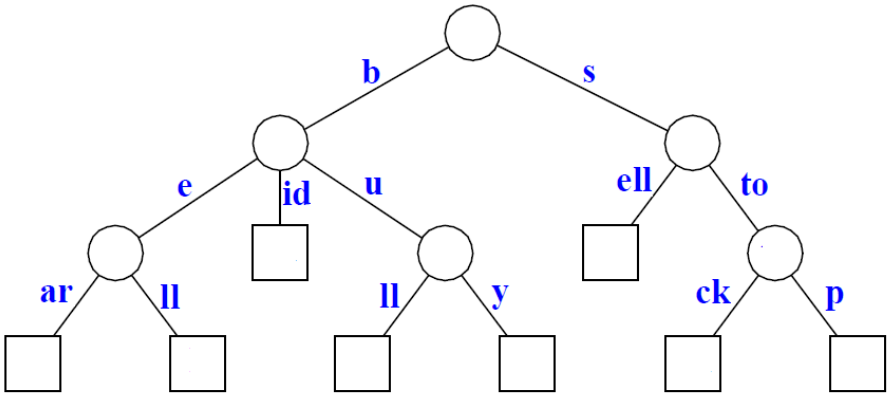
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text. We have discussed Standard Trie. Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



How to build a Suffix Tree for a given text?

As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

1) Generate all suffixes of given text.

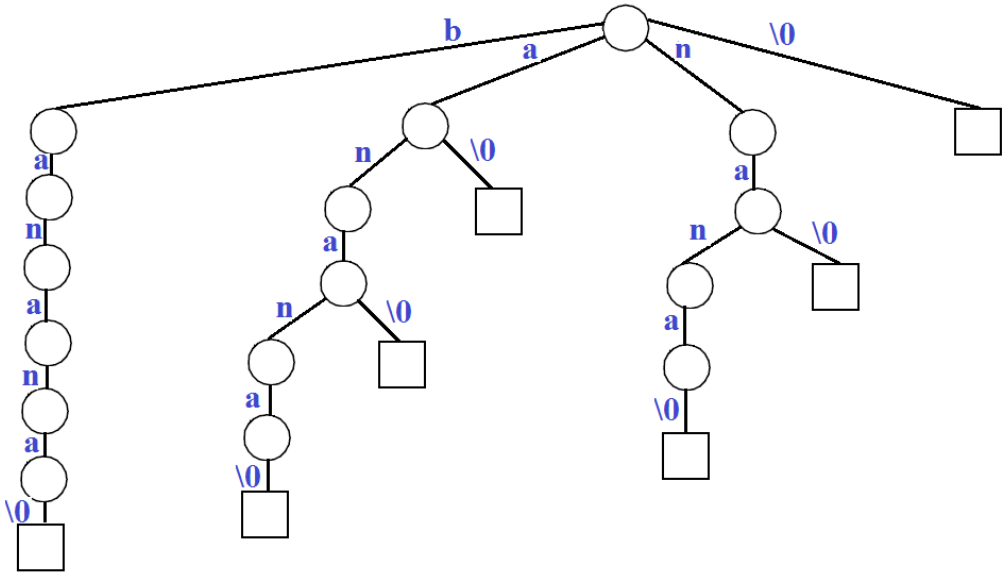
2) Consider all suffixes as individual words and build a compressed trie.

Let us consider an example text “banana\0″ where ‘\0′ is string termination character. Following are all suffixes of “banana\0″

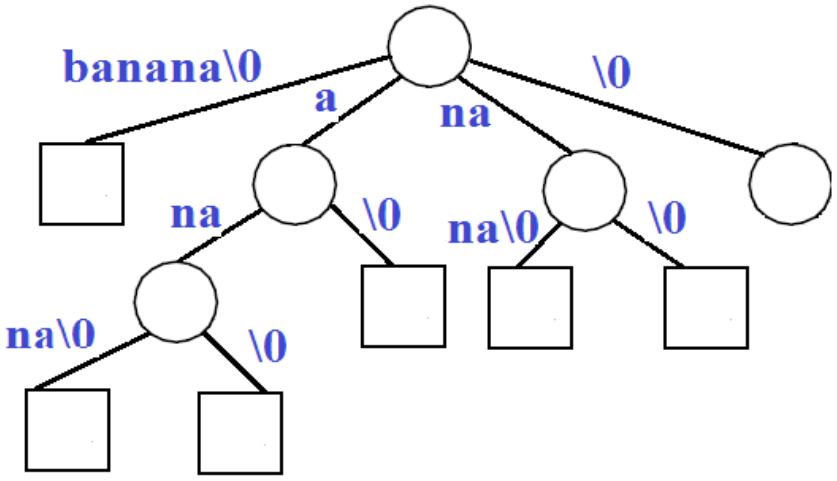
```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
```

```
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text “banana\0″



Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?

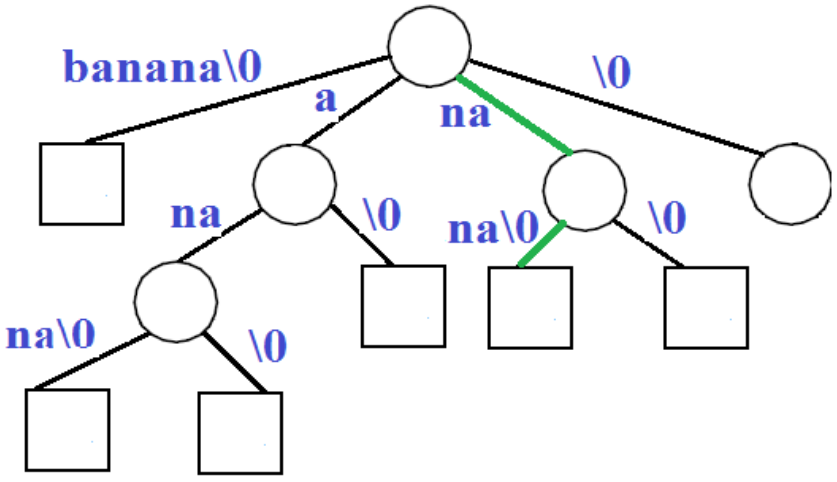
We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.

…..**a)** For the current character of pattern, if there is an edge from the current

node of suffix tree, follow the edge.
…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print “Pattern found”.

Let us consider the example pattern as “nan” to see the searching process. Following diagram shows the path followed for searching “nan” or “nana”.



How does this work?
Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree
Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

There are many more applications. See this for more details.

Ukkonen’s Suffix Tree Construction is discussed in following articles:
Ukkonen’s Suffix Tree Construction – Part 1
Ukkonen’s Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

References:

http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf
http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf
http://www.allisons.org/ll/AlgDS/Tree/Suffix/