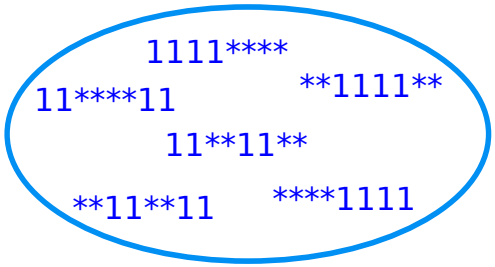
# Next generation sequencing

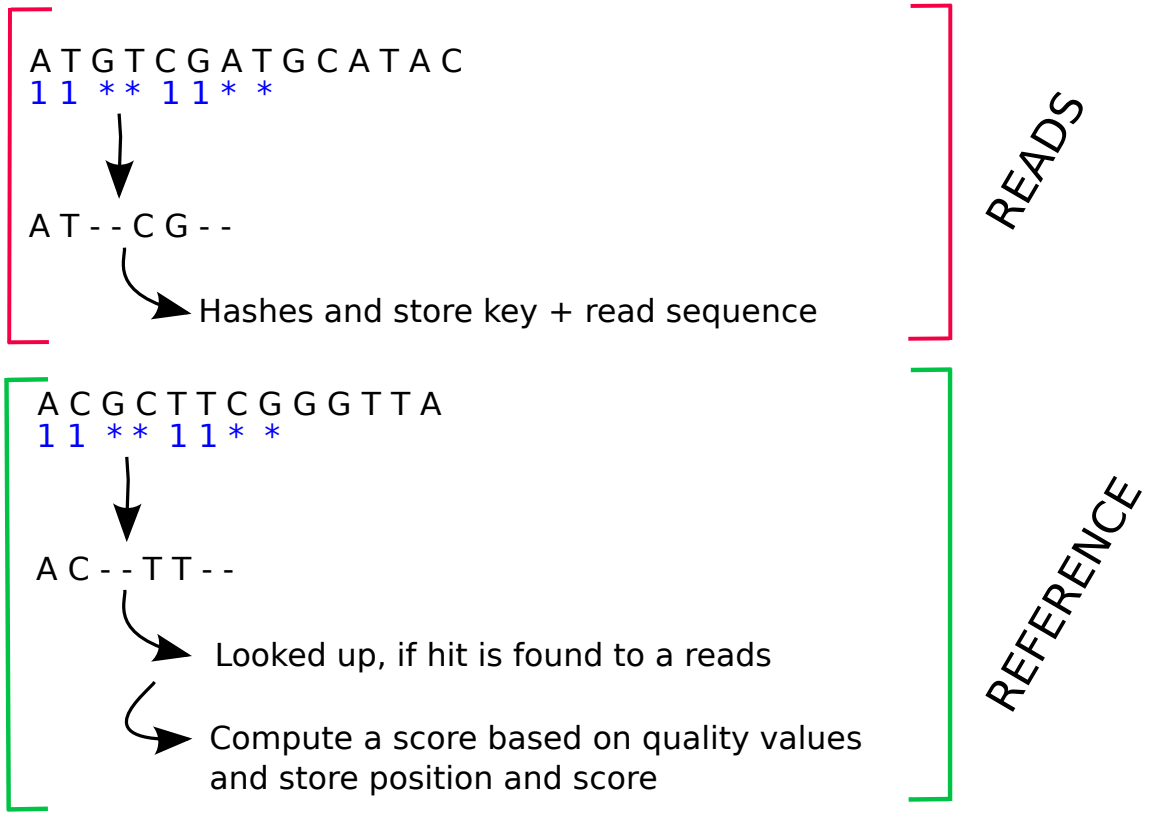
# Data structure in NGS mapping algorithm

- Hash

- Seed consecutive and not consecutive

# Hash and seed




Full accurancy on the first 28 positions of the read

# Second Generation Mappers have Leveraged the Burrows Wheeler Transformation

Software

**Ultrafast and memory-efficient alignment of short DNA sequences to the human genome**

Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg

Abstract

Bowtie is an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. For the human genome, Burrows-Wheeler indexing allows Bowtie to align more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes. Bowtie extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches. Multiple processor cores can be used simultaneously to achieve even greater alignment speeds. Bowtie is open source http://bowtie.cbcb.umd.edu.

BIOINFORMATICS APPLICATIONS NOTE

**SOAP: short oligonucleotide alignment program**



BIOINFORMATICS APPLICATIONS NOTE

**SOAP2: an improved ultrafast tool for short read alignment**

"…35 times faster than Maq and 300 times faster than SOAP under the same conditions"

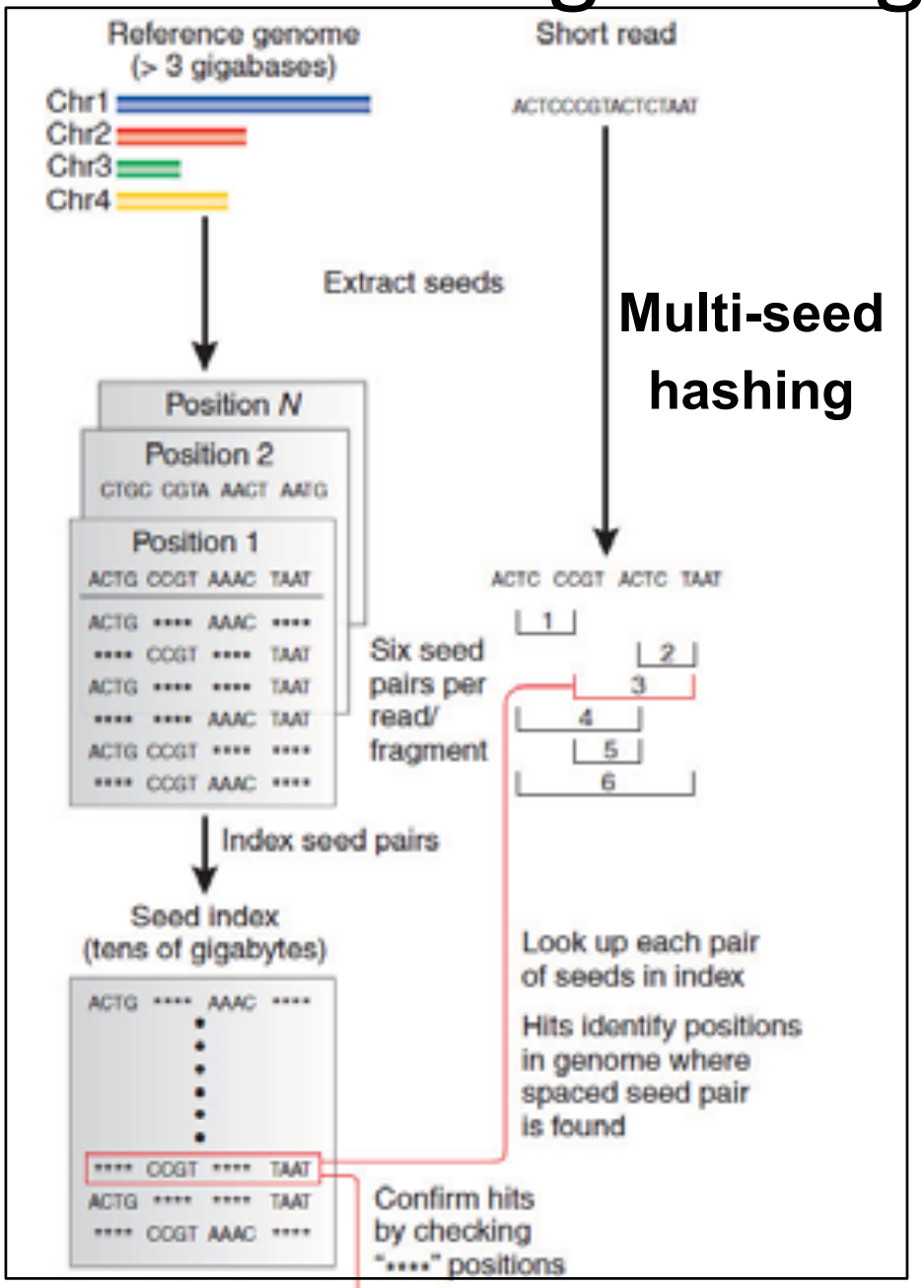
**Mapping short DNA sequencing reads and calling variants using mapping quality scores**

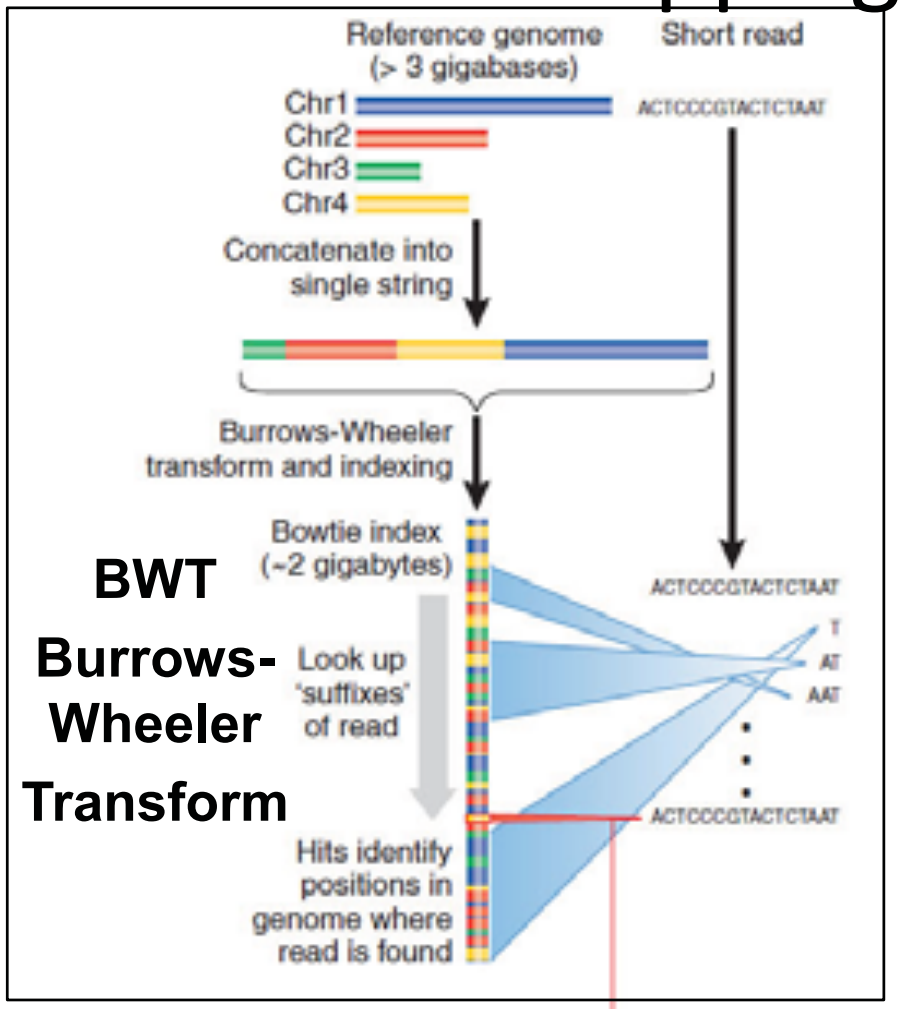


BIOINFORMATICS ORIGINAL PAPER

**Fast and accurate short read alignment with Burrows-Wheeler transform**

# Two indexing strategies for read mapping




**Today: How does the BW transform actually work?**

Multiple Pattern Matching  Problem

*given a set of patterns and a text, find all occurrences of any of the patterns in the text*

**input**: A set of k patterns  P1, P2, …, Pk and text *t=t1…tm*

**output**: All positions *1 <= i <= m* such that a substring of t starting at position i coincides with a pattern Pj for *1 <= j <= k*

t = ATGGTCGGT

P1=GGT

P2=GGG

P3=CG

P4=ATG

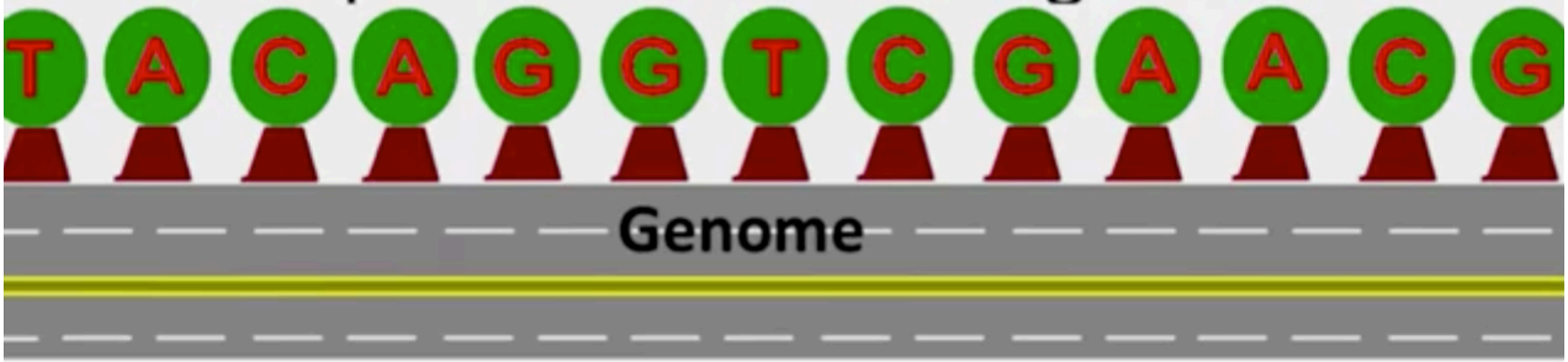
Positions = {1,3,6,7}

This algorithm is resolved in  **O(knm)** time where n is the length of the longest of the k patterns, by k application s of the PATTERN MATCHING algorithm

# Brute Force Is Too Slow

- The runtime of the brute force approach is too high!
  - Single *Pattern*: O(|*Genome*| * |*Pattern*|)
  - Multiple *Patterns*: O(|*Genome*| * |*Patterns*|)
  - |*Patterns*| = combined length of *Patterns*

# Multiple Pattern Matching Problem



# Packing Patterns onto a Bus

# Tries

**Patterns**

banana
pan
nab
antenna
bandana
ananas
nana

- **Trie:** a data structure for representing a collection of strings.
- Tries support fast pattern matching.

**Idea**:

- Combine patterns into a rooted-tree with branches labeled by letters in the alphabet.
- Every string in Patterns is presented as a root-to-leaf path.

## Keyword Tree

The keyword tree for a set of patterns P1, P2, P3, …, Pk is a rooted labeled tree satisfying the following condition:

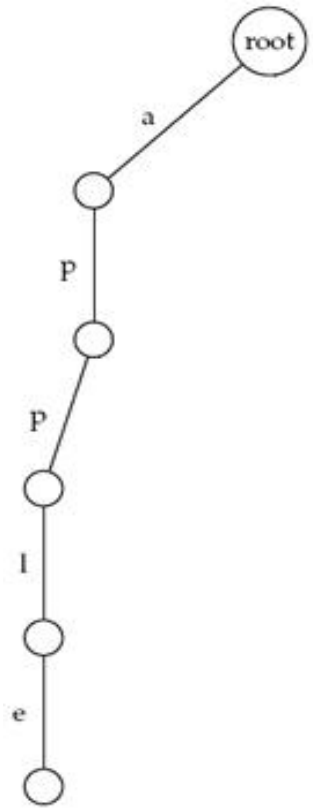
- Each edge of the tree is labeled with a letter of the alphabet
- Any two edges out of the same vertex have distinct labels
- Every pattern Pi from the set of patterns is spelled on some path from the root to the leaf

The time to build a keyword tree is O(N) where N is the total length of patterns P1, P2, P3, …, Pk

The time to solve the *Multiple Pattern Matching Problem* is O(N + nm)

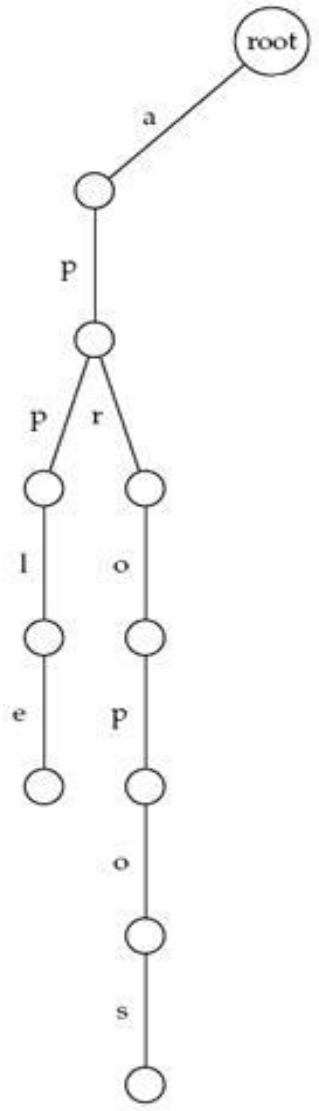
# Keyword Trees: Example

- ***Keyword tree***:
  - Apple



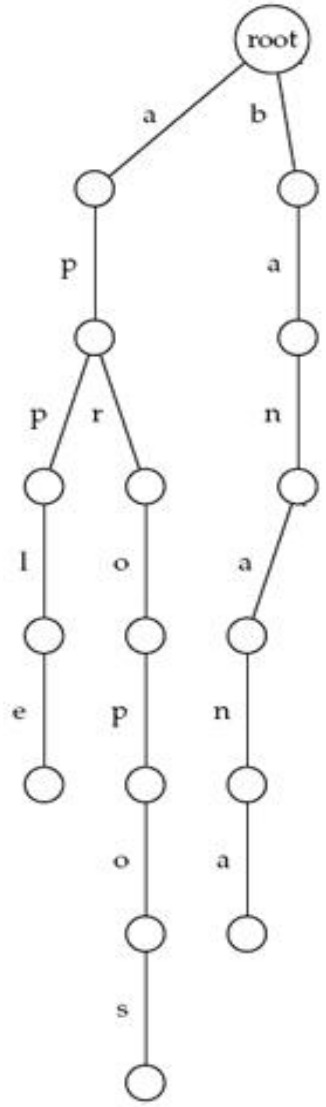
# Keyword Trees: Example

- ***Keyword tree***:
  - Apple
  - Apropos



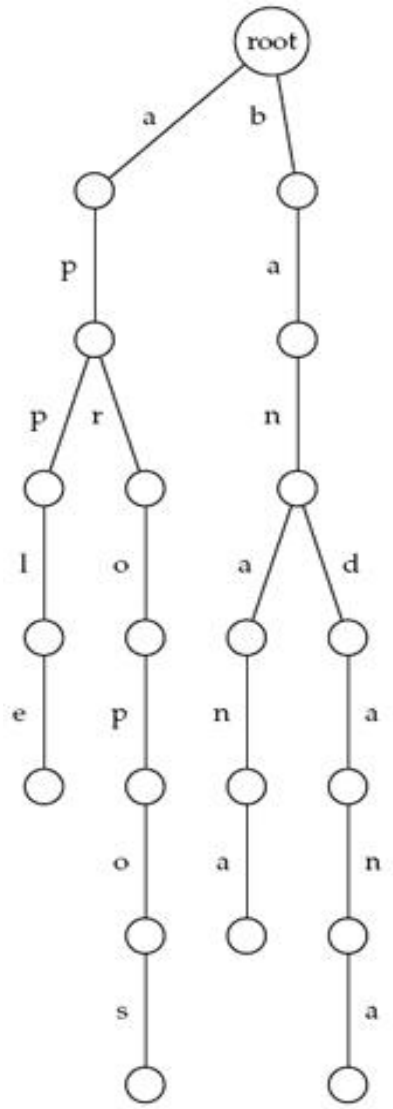
# Keyword Trees: Example

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana



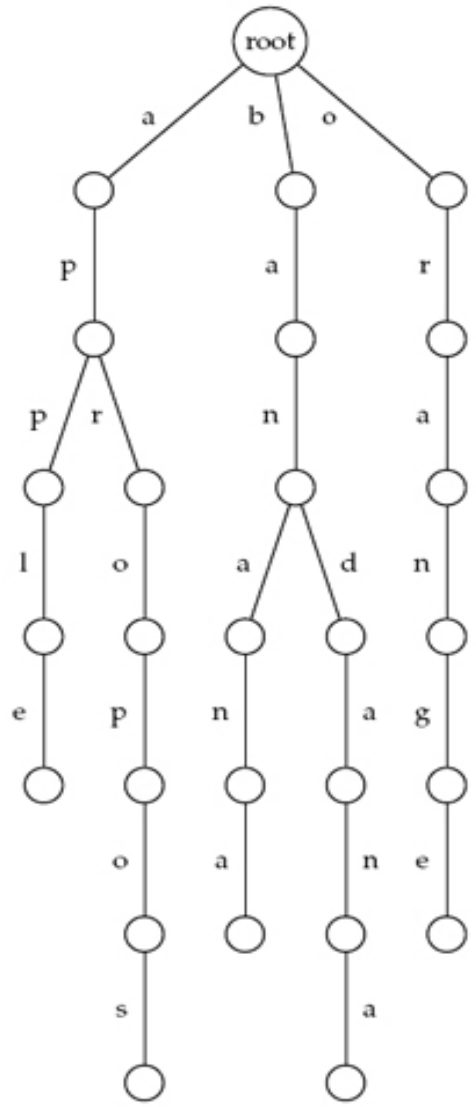
# Keyword Trees: Example

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana



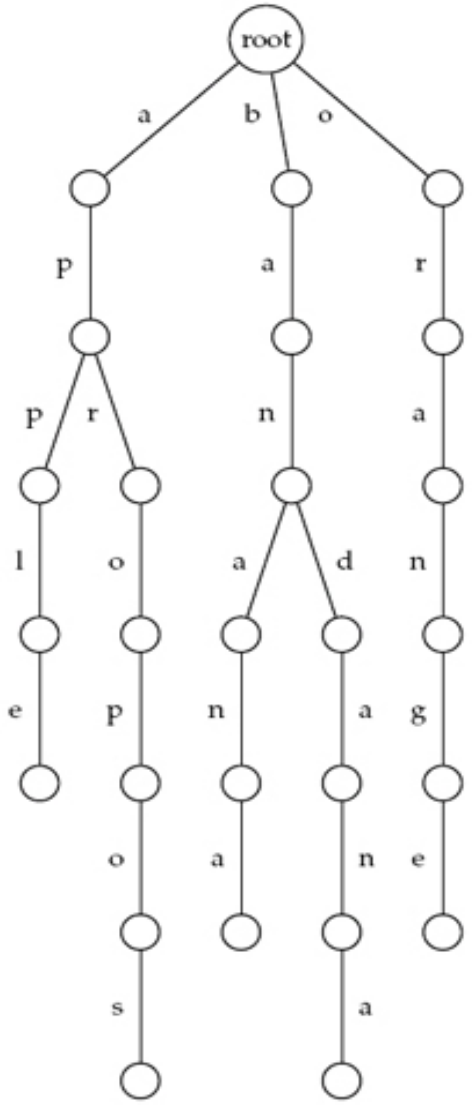
# Keyword Trees: Example

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana
  - Orange



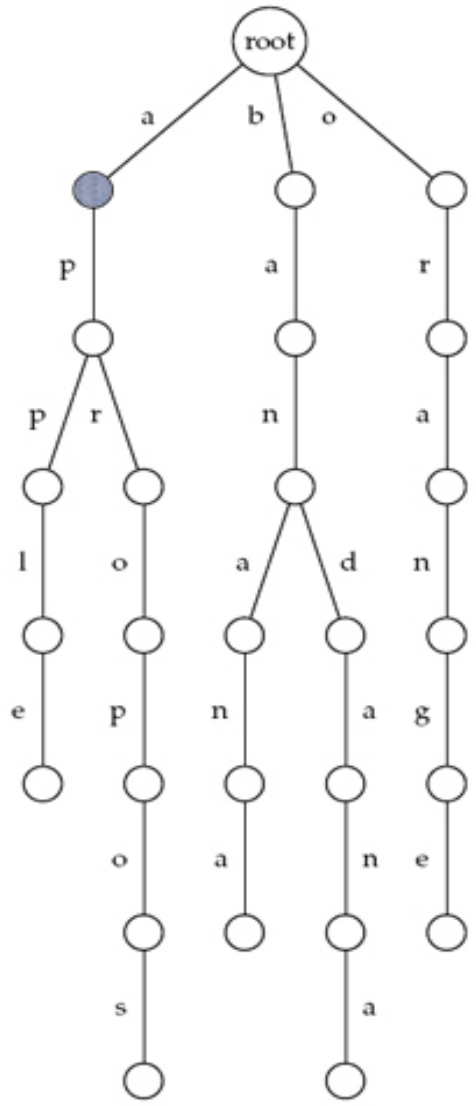
# Keyword Trees: Properties

- Stores a set of keywords in a rooted labeled tree.
- Each edge is labeled with a letter from an alphabet.
- Any two edges coming out of the same vertex have distinct labels.
- Every keyword stored can be spelled on a path from root to some leaf.
- Furthermore, every path from root to leaf gives a keyword.



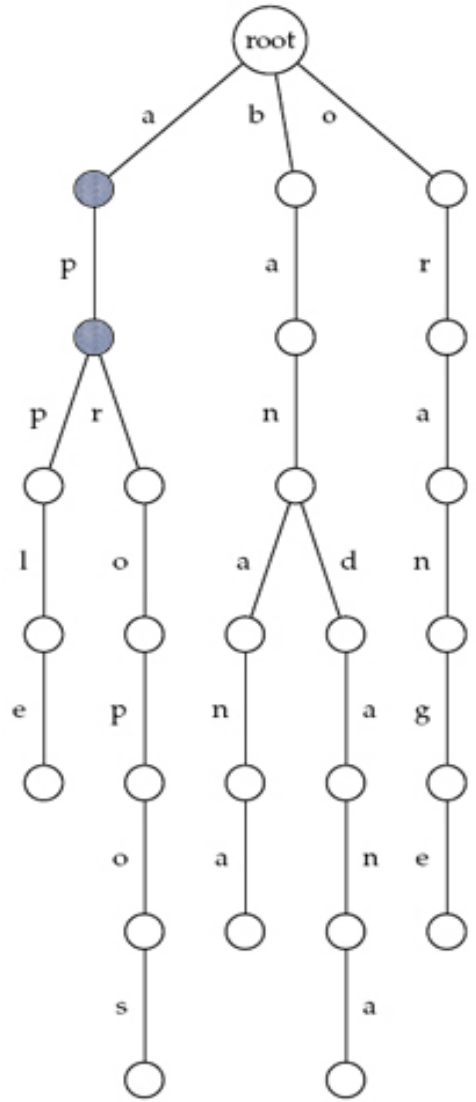
# Keyword Trees: Threading

- Thread "appeal"
  - appeal



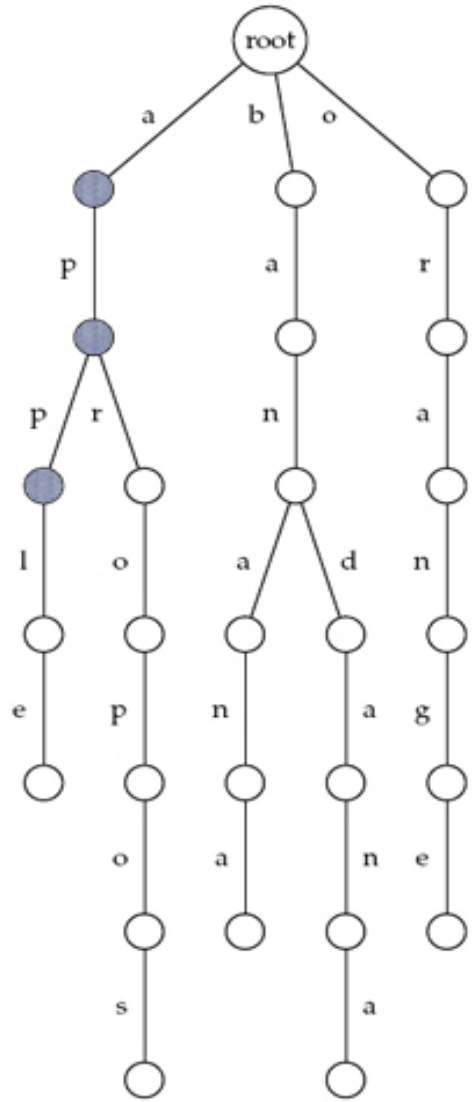
# Keyword Trees: Threading

- Thread “appeal”
  - appeal



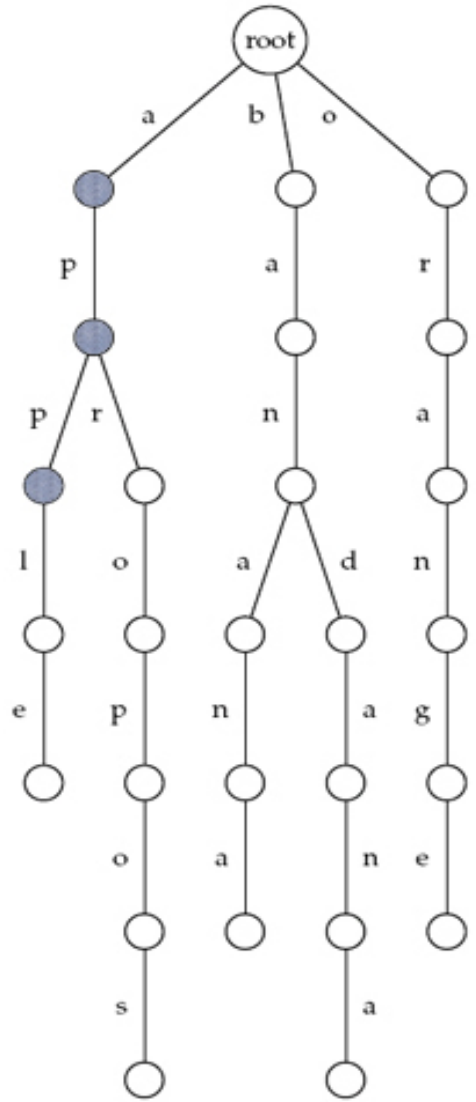
# Keyword Trees: Threading

- Thread “appeal”
  - appeal



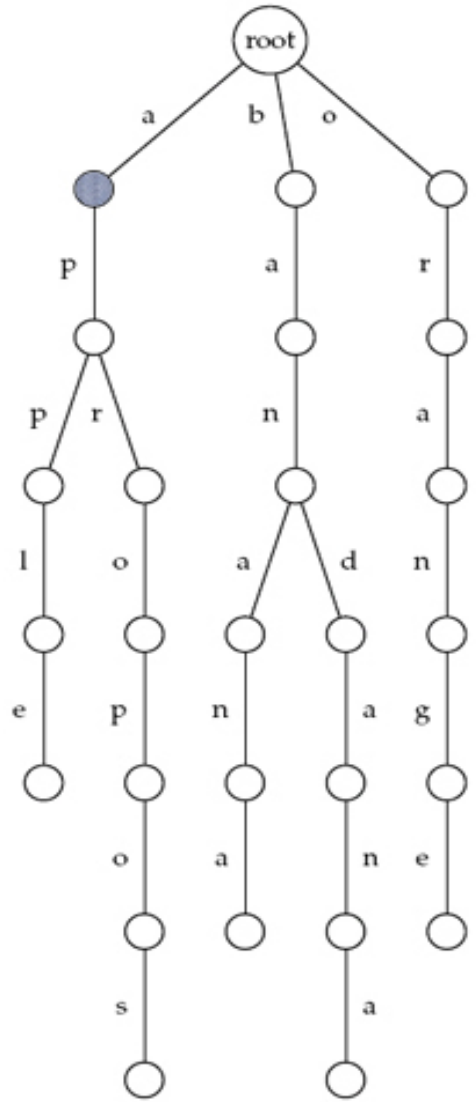
# Keyword Trees: Threading

- Thread “appeal”
  - appeal



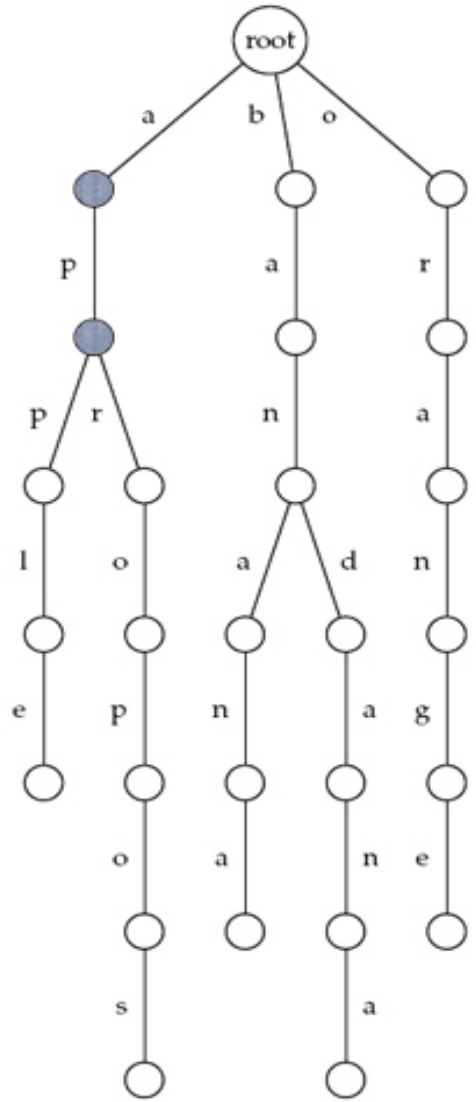
# Keyword Trees: Threading

- Thread “apple”
  - apple



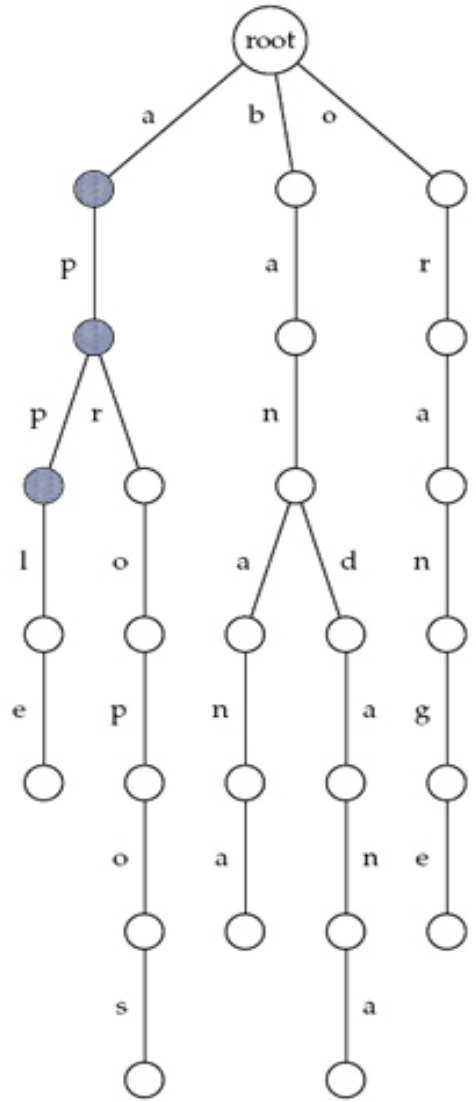
# Keyword Trees: Threading

- Thread “apple”
  - apple



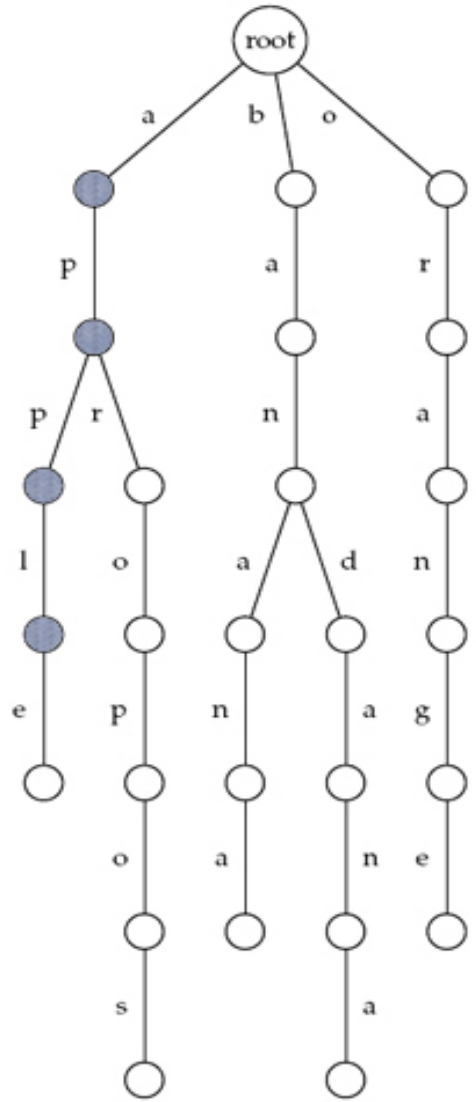
# Keyword Trees: Threading

- Thread “apple”
  - apple



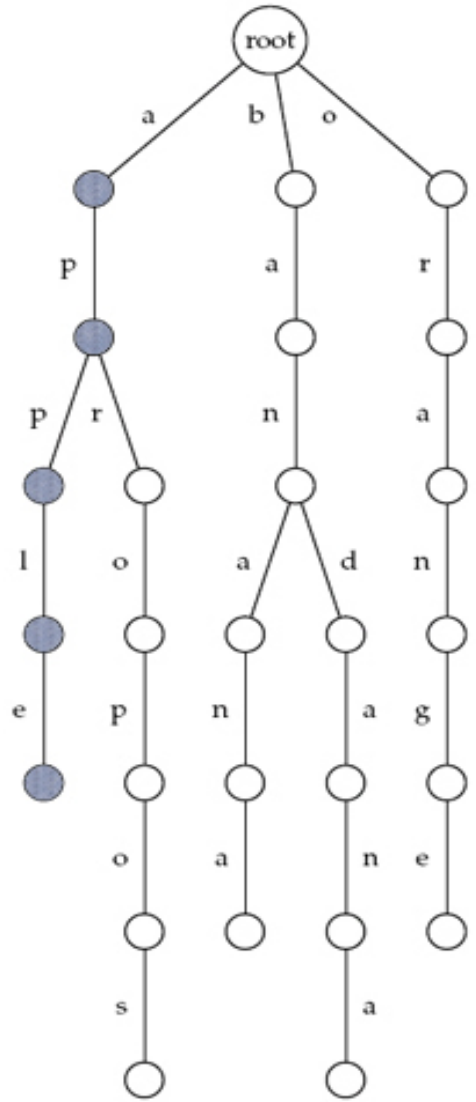
# Keyword Trees: Threading

- Thread “apple”
  - apple



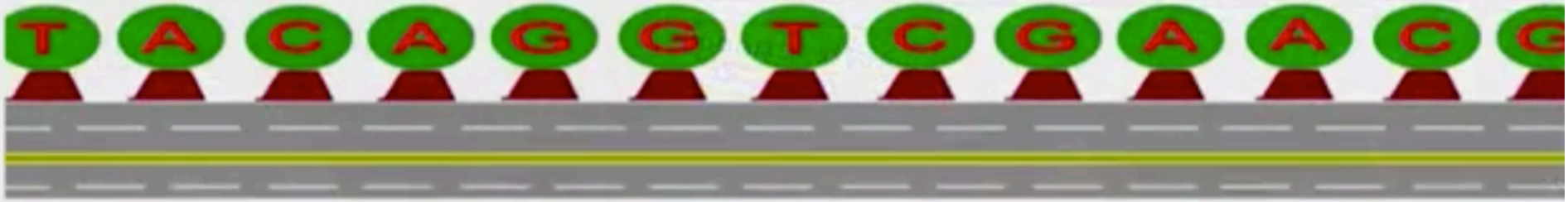
# Keyword Trees: Threading

- Thread “apple”
  - apple



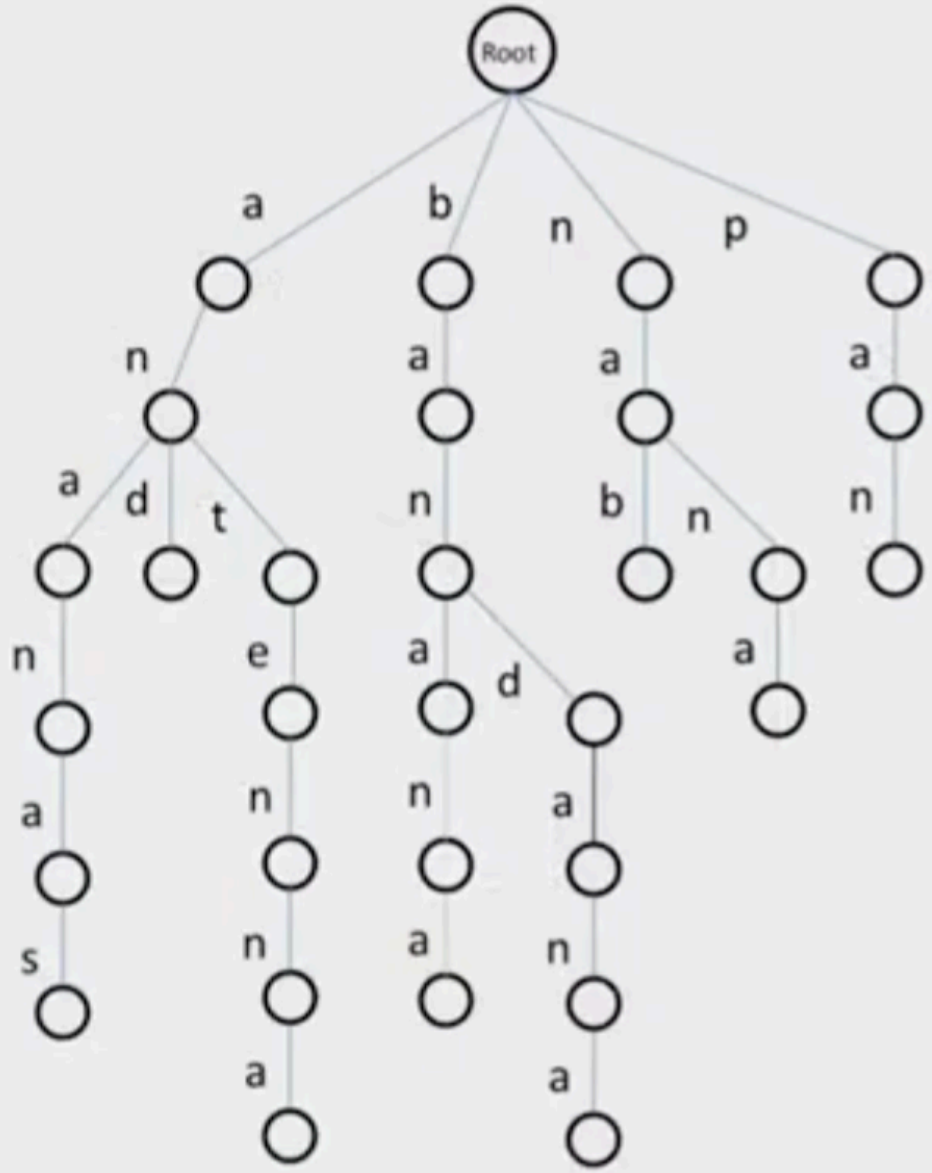
# Using the Trie for Pattern Matching

- **TrieMatching**: Slide the trie down the genome.

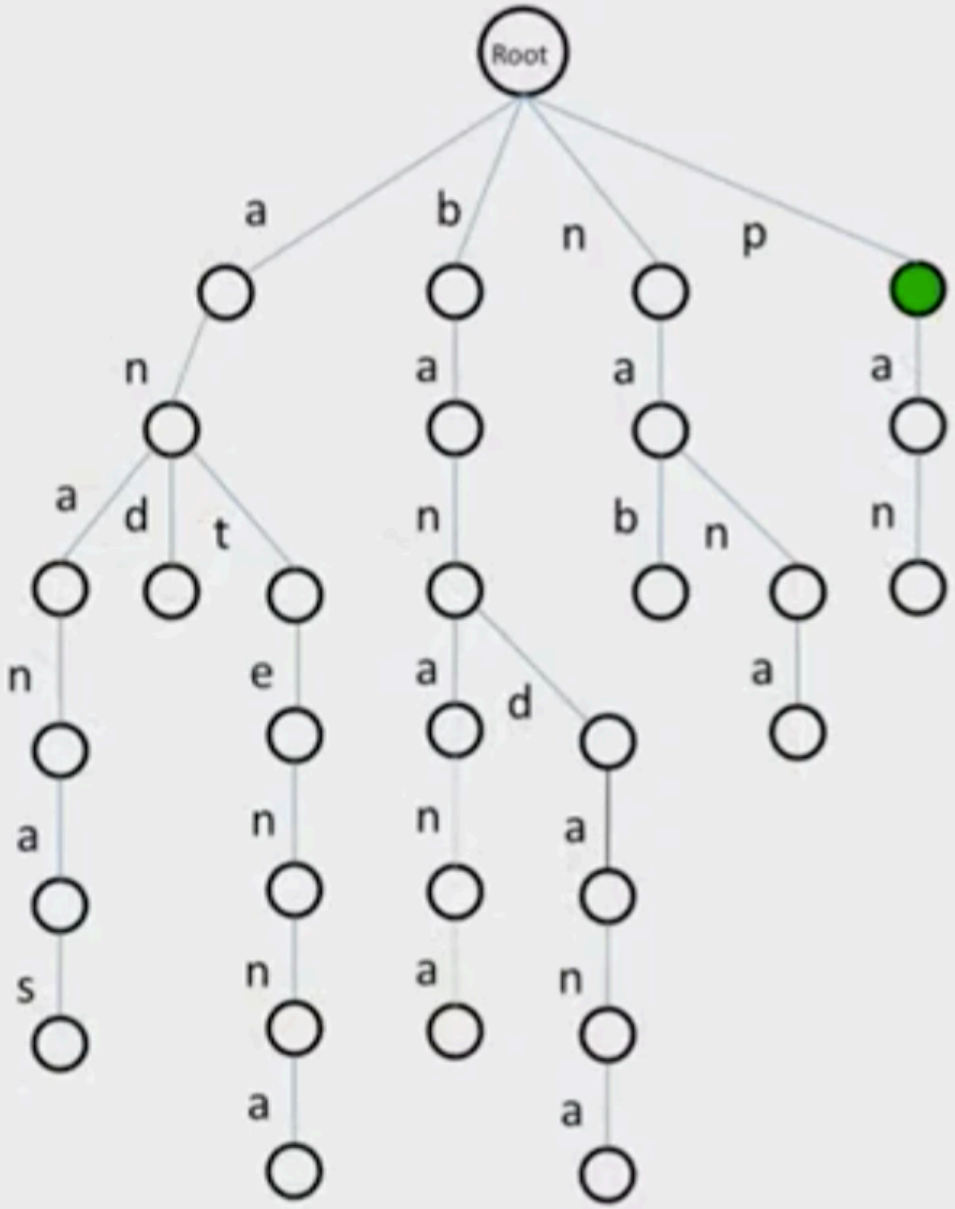



- At each position, walk down the trie and see if we can reach a leaf by matching symbols.

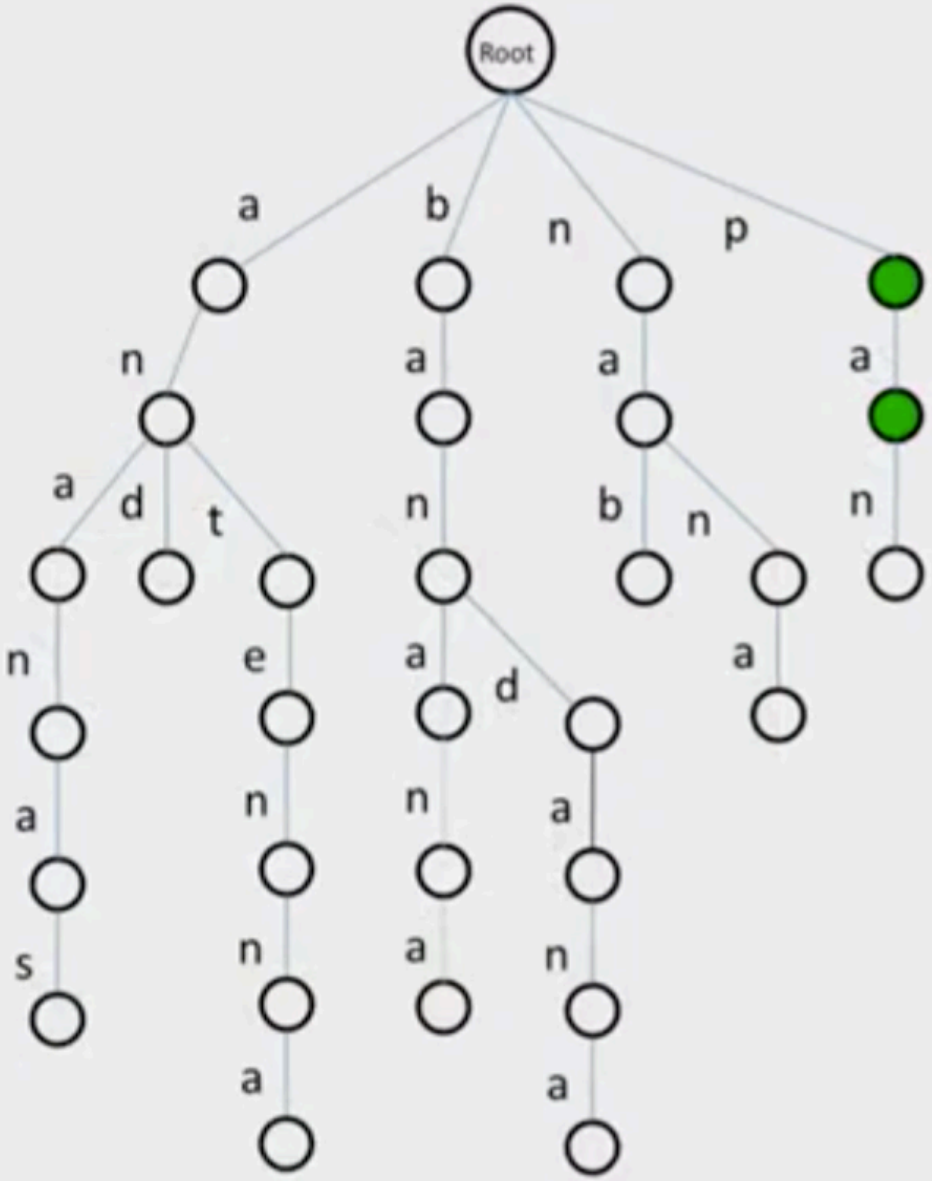
# p a n a m a b a n a n a s



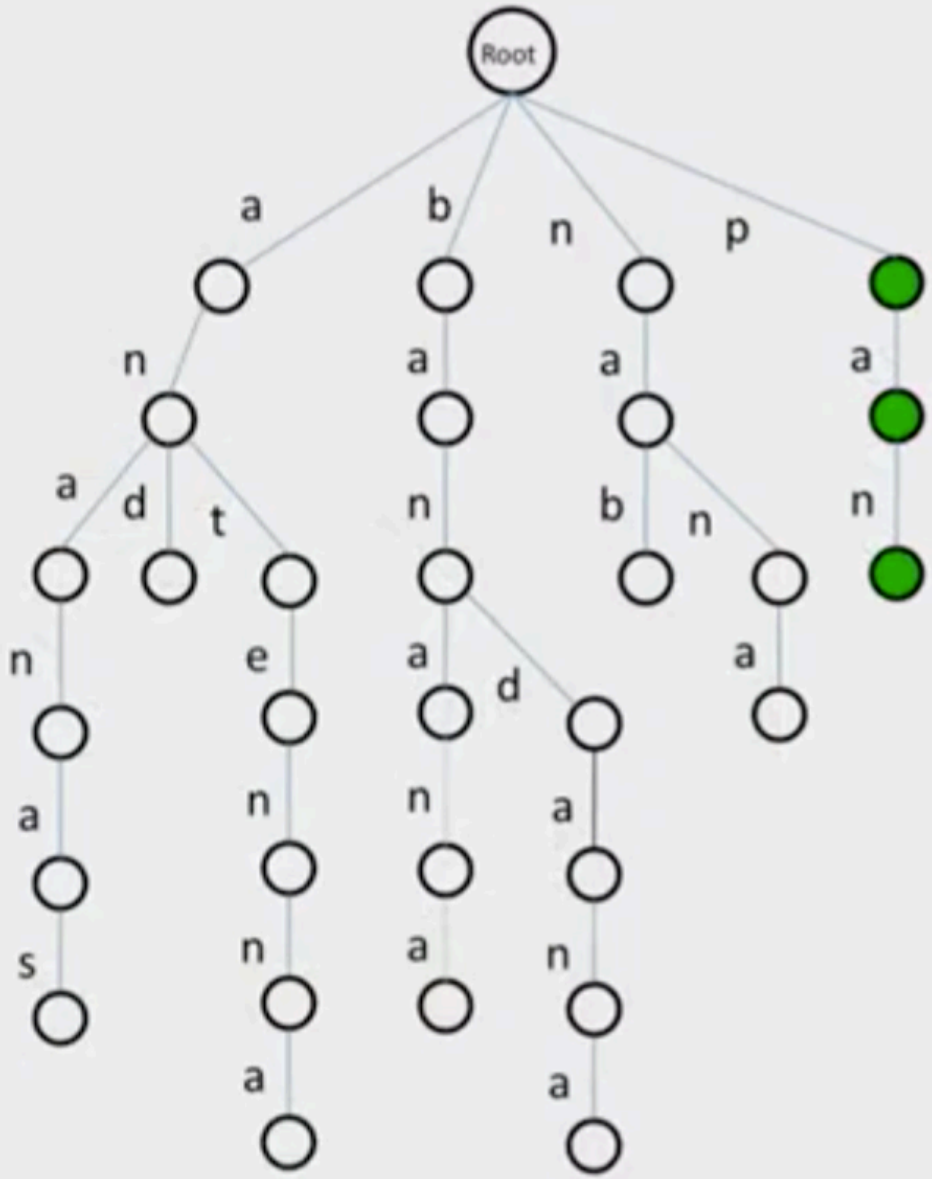
# panamabananas



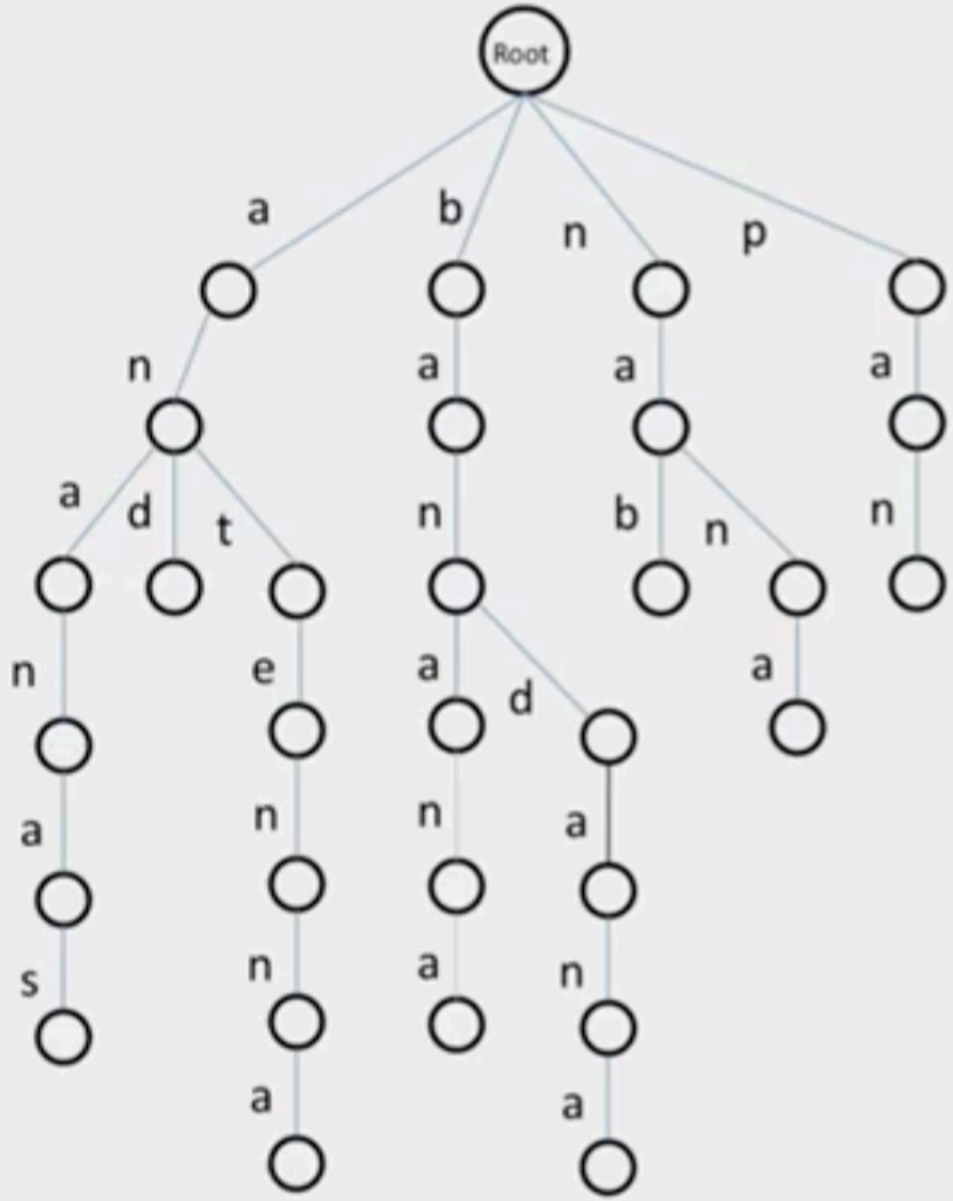
# p a n a m a b a n a n a s



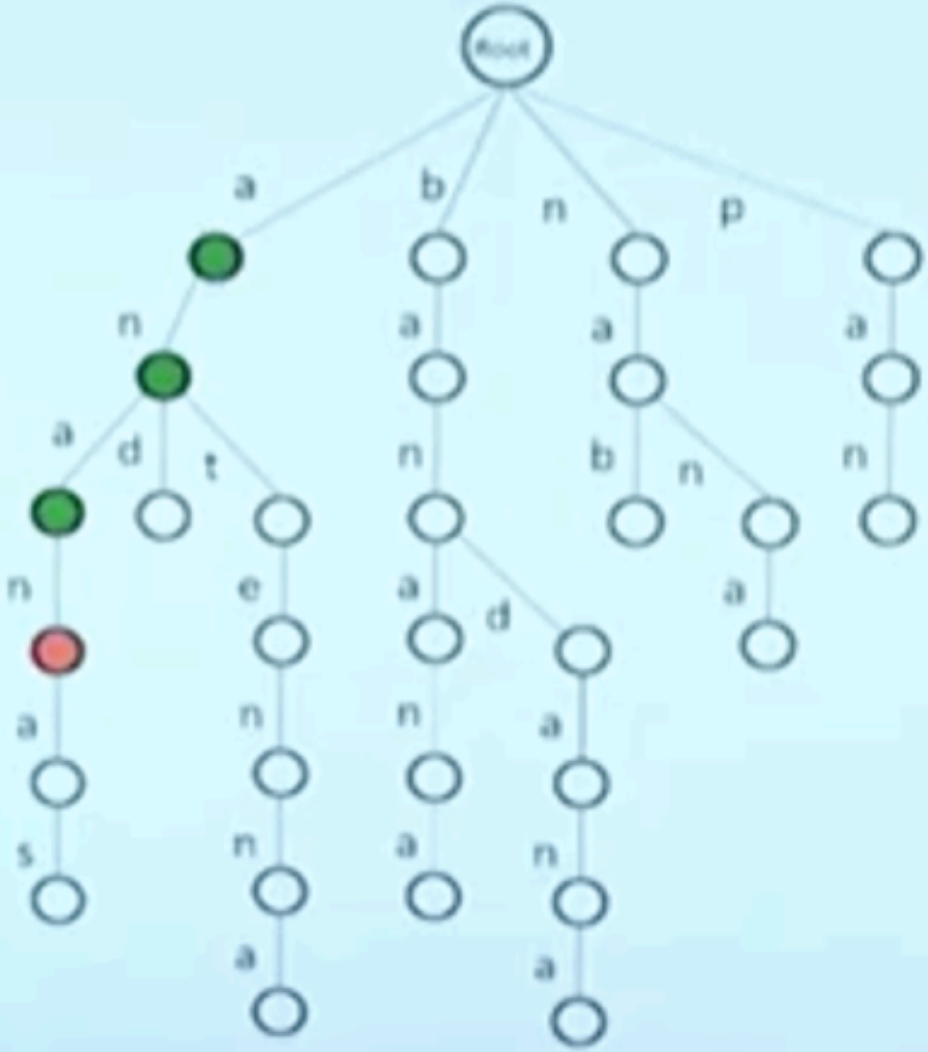
# panamabananas



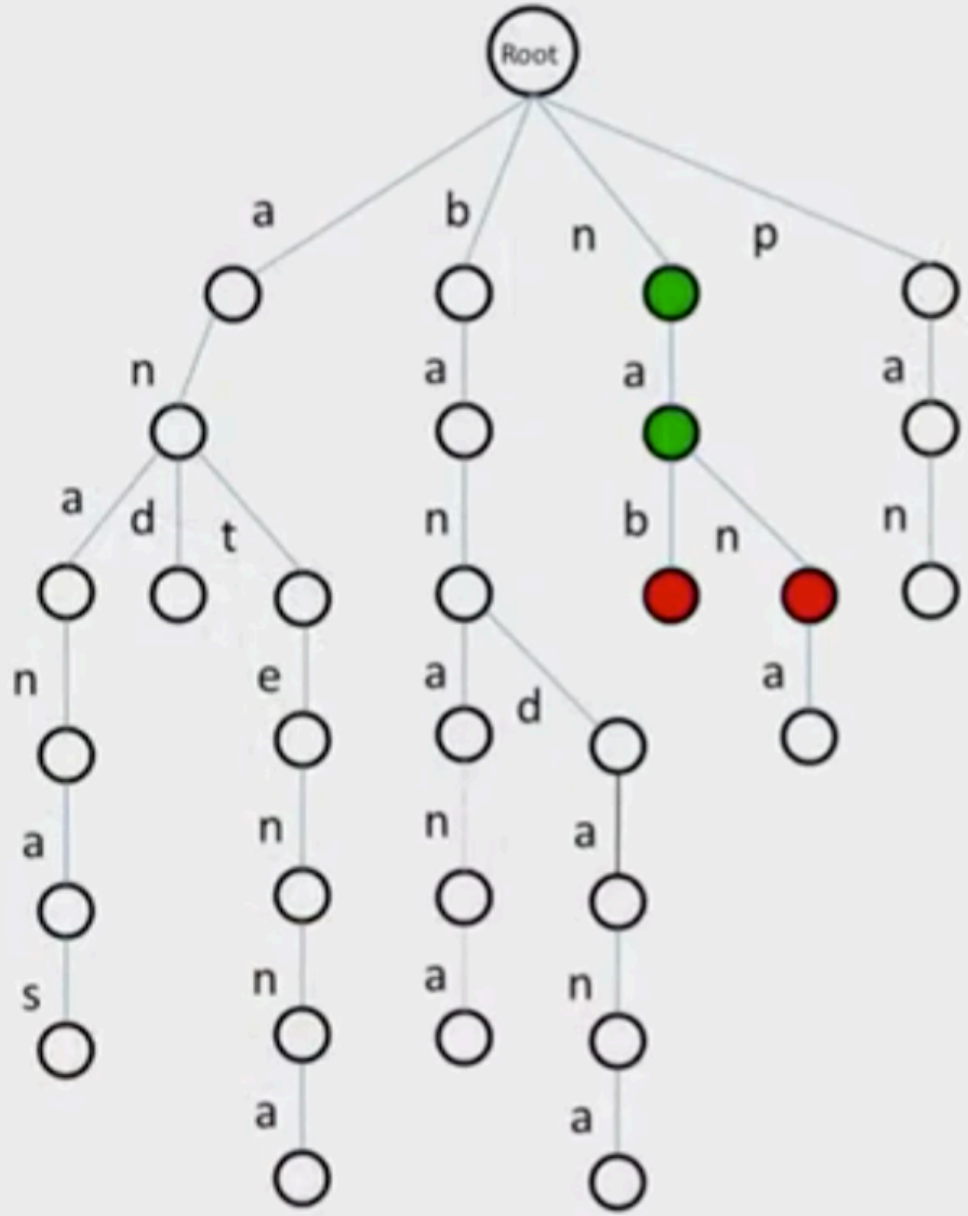
# panamabanananas



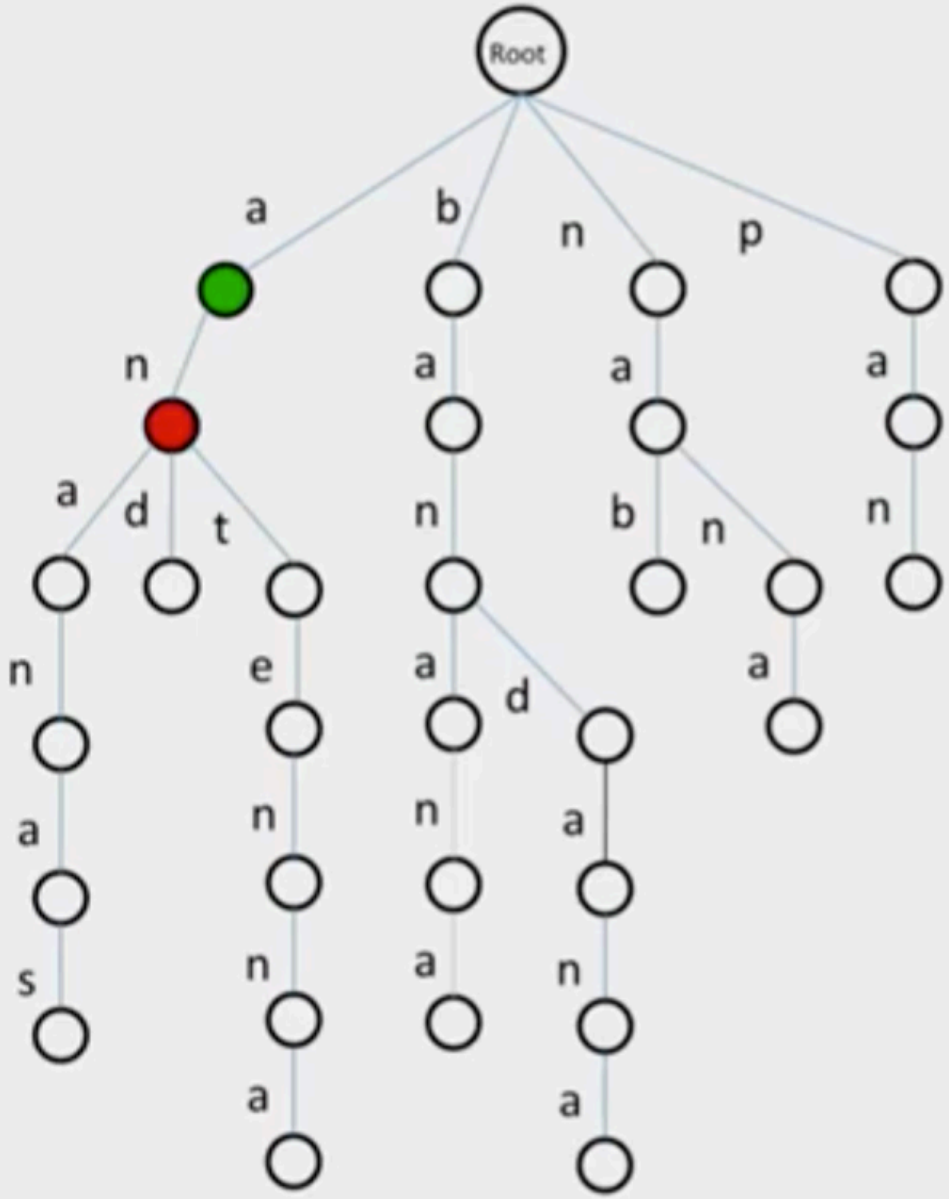
# p a n a m a b a n a n a s



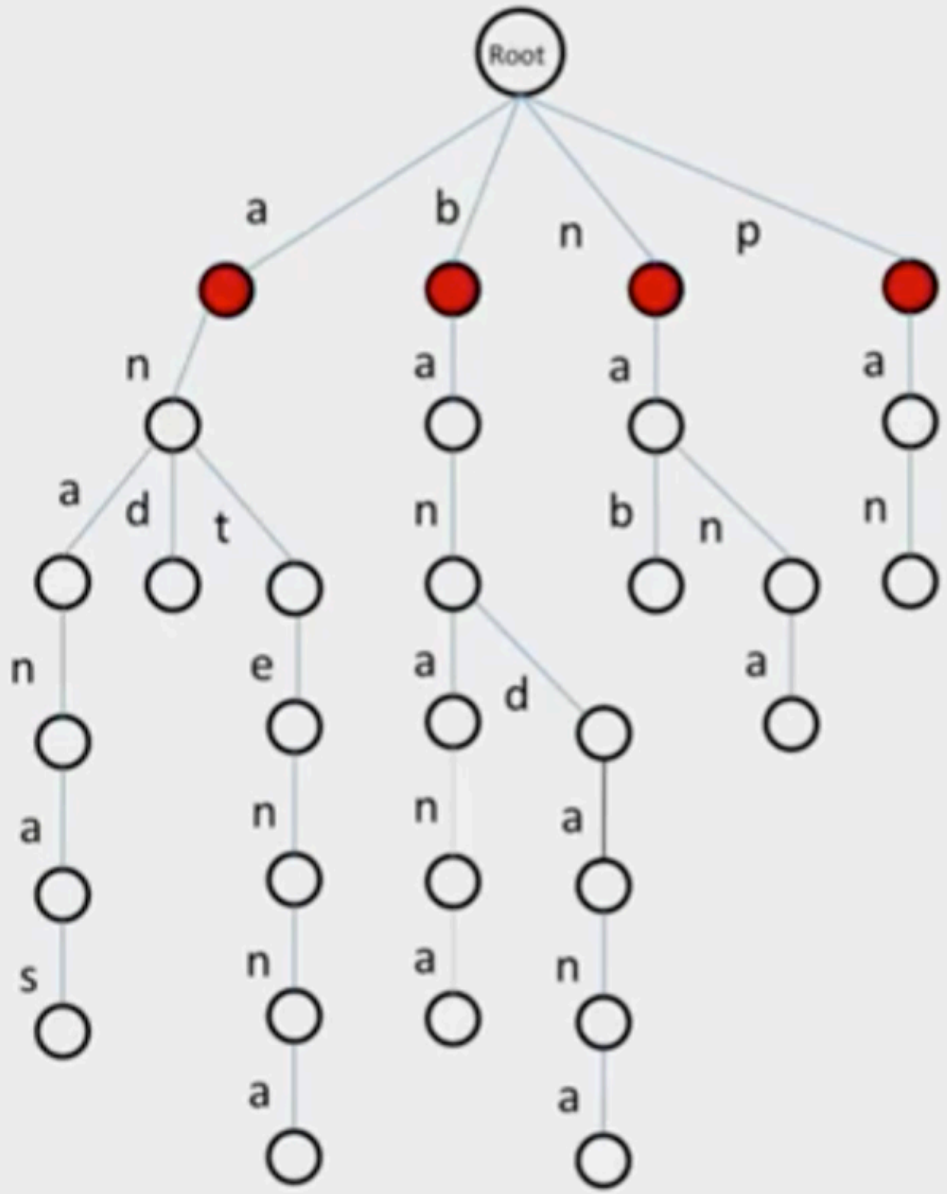
# p a n a m a b a n a n a s



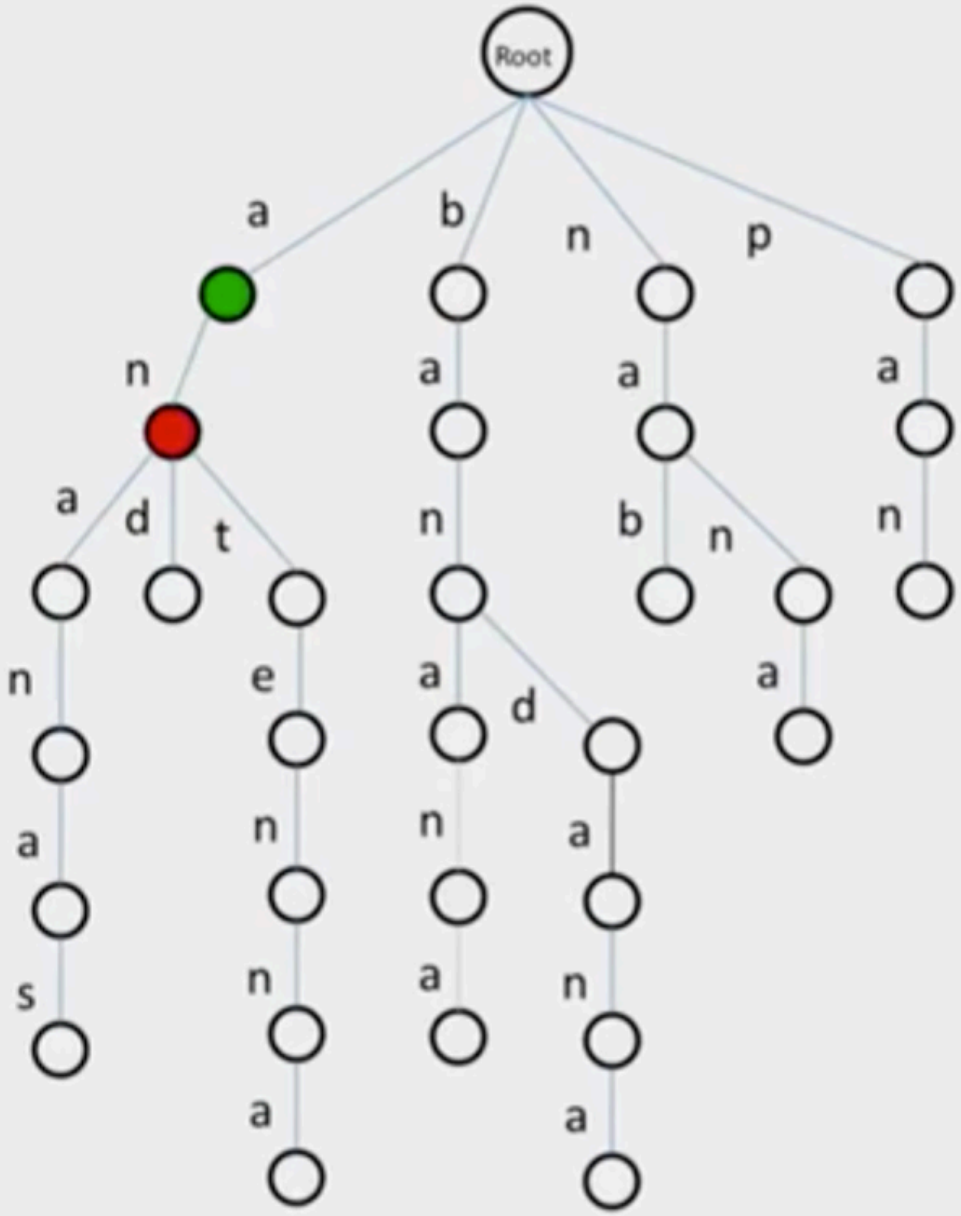
# p a n a m a b a n a n a s



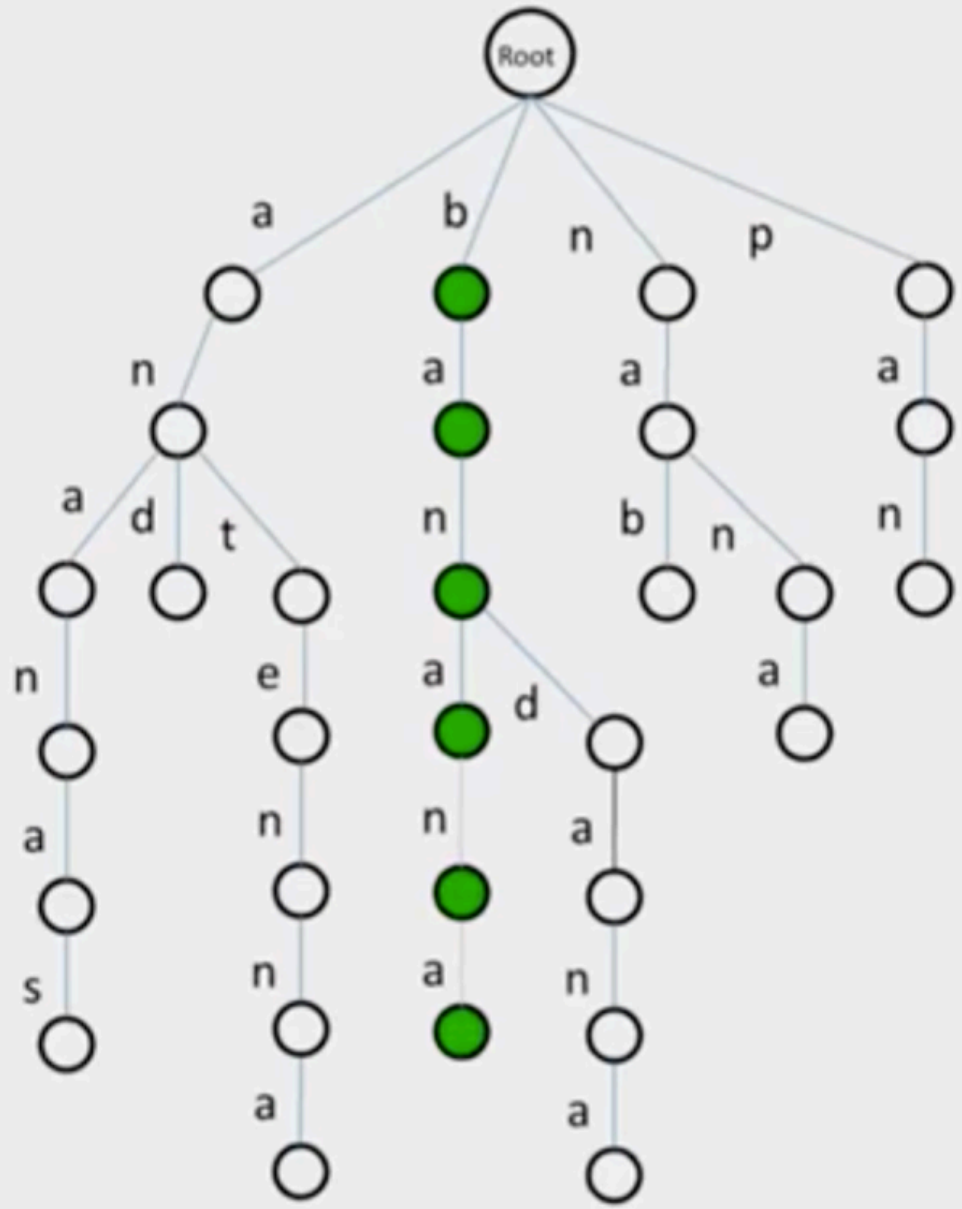
panamabananas



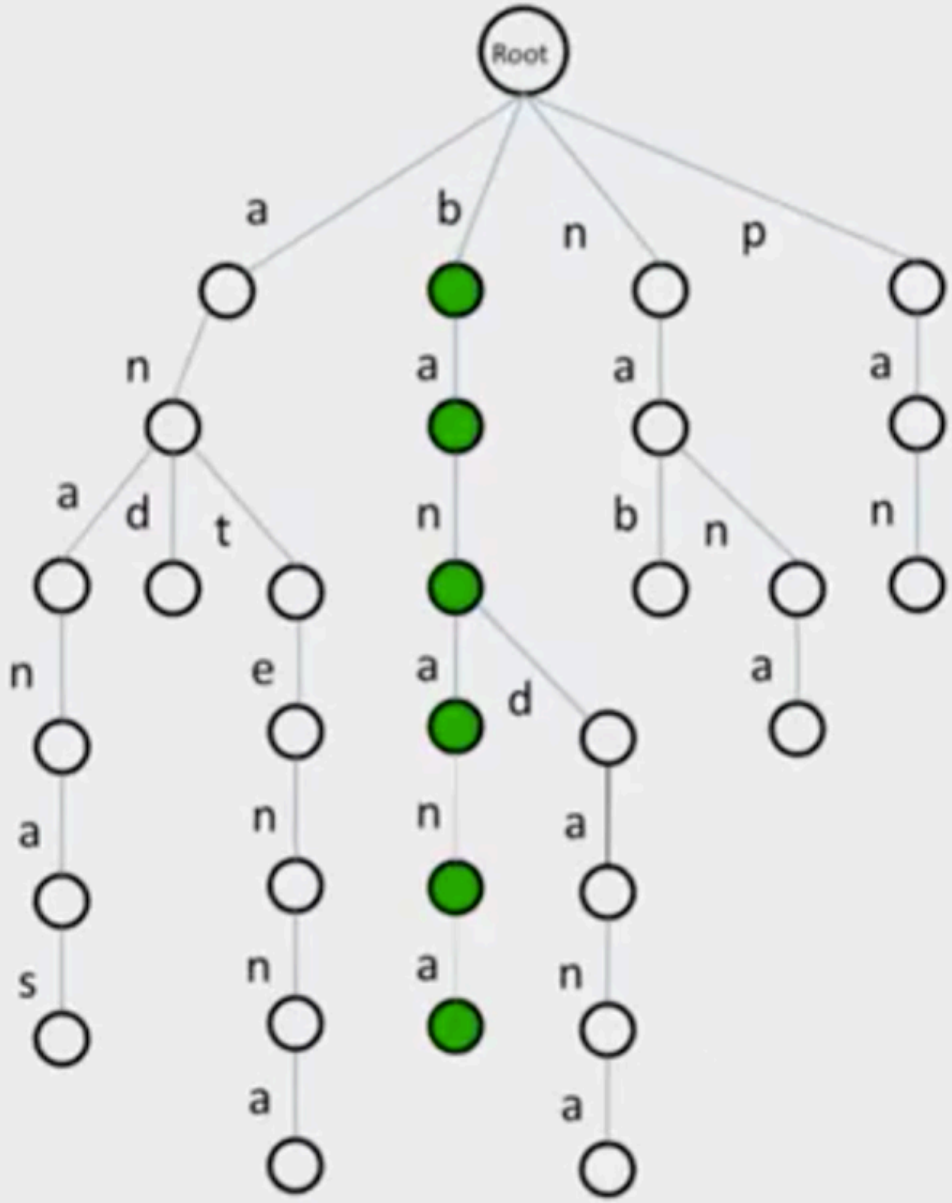
panamabananas



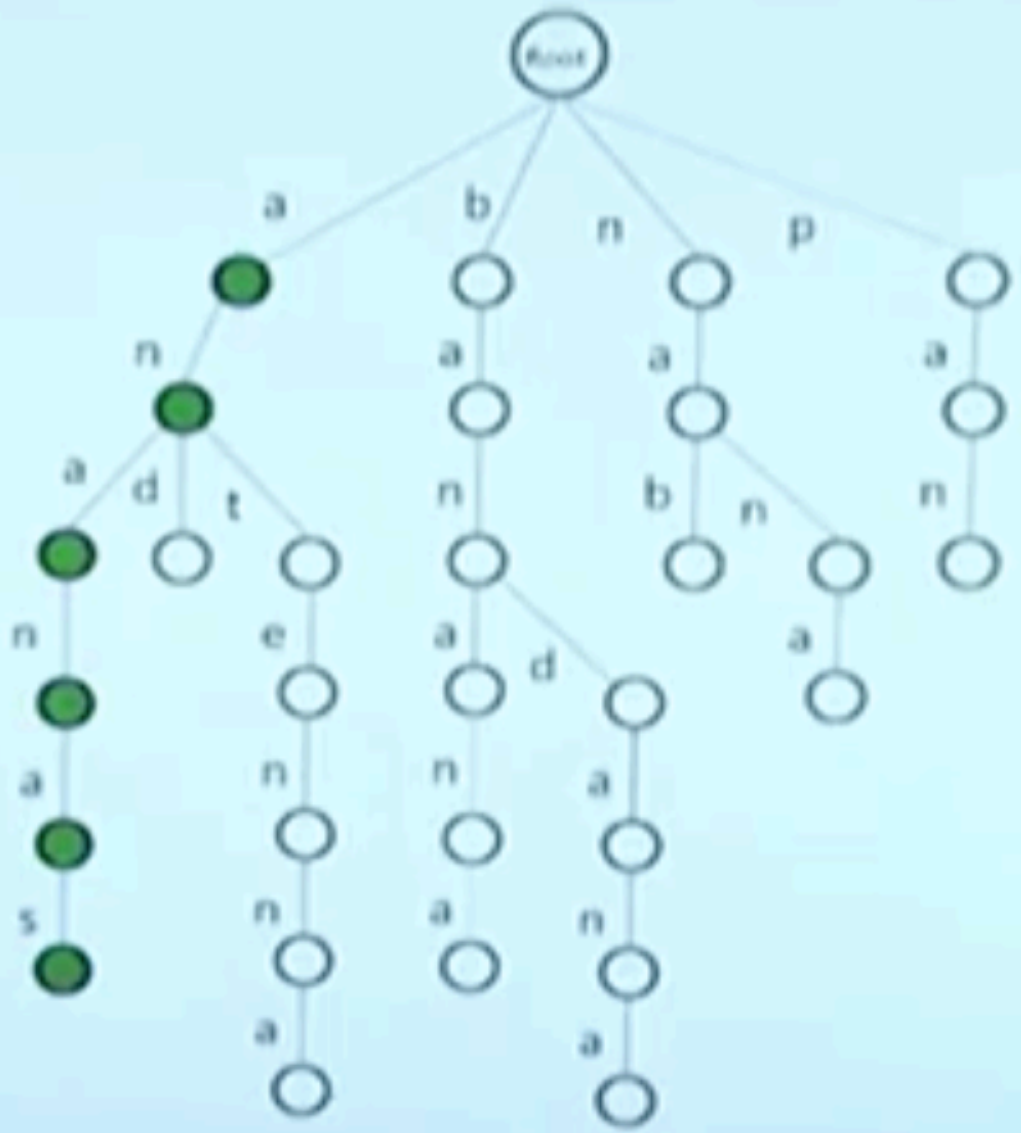
panamabananas



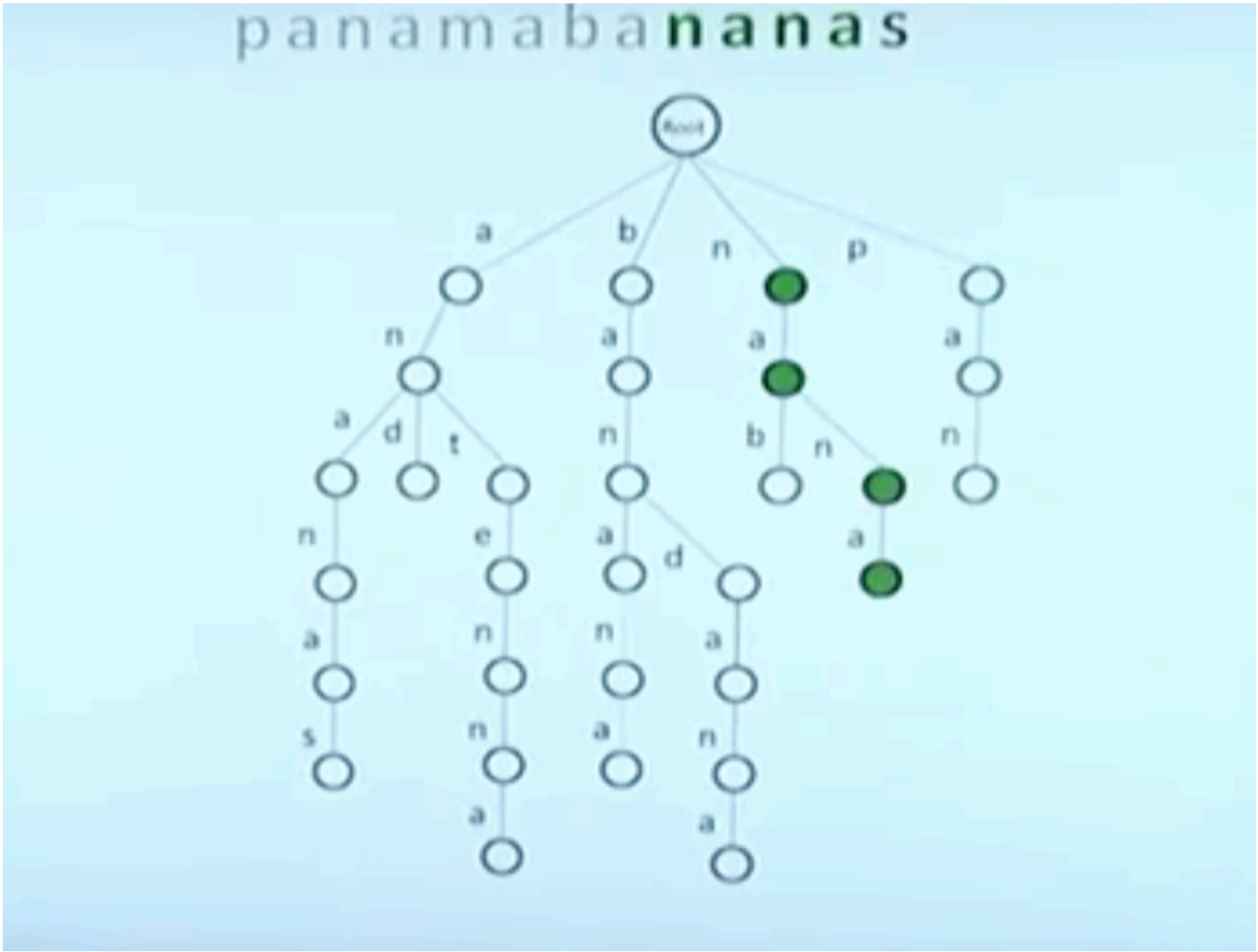
panamabananas



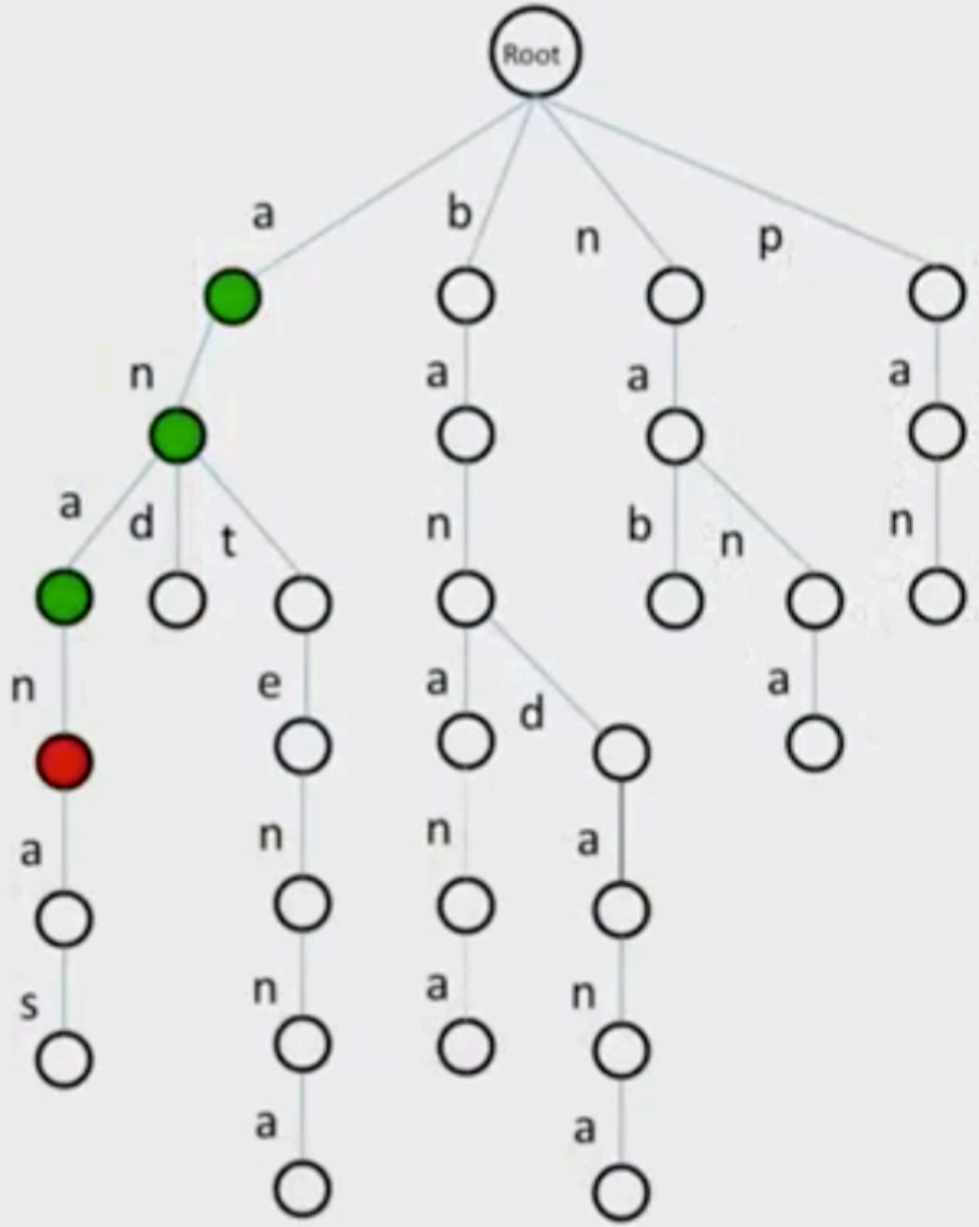
panamabananas



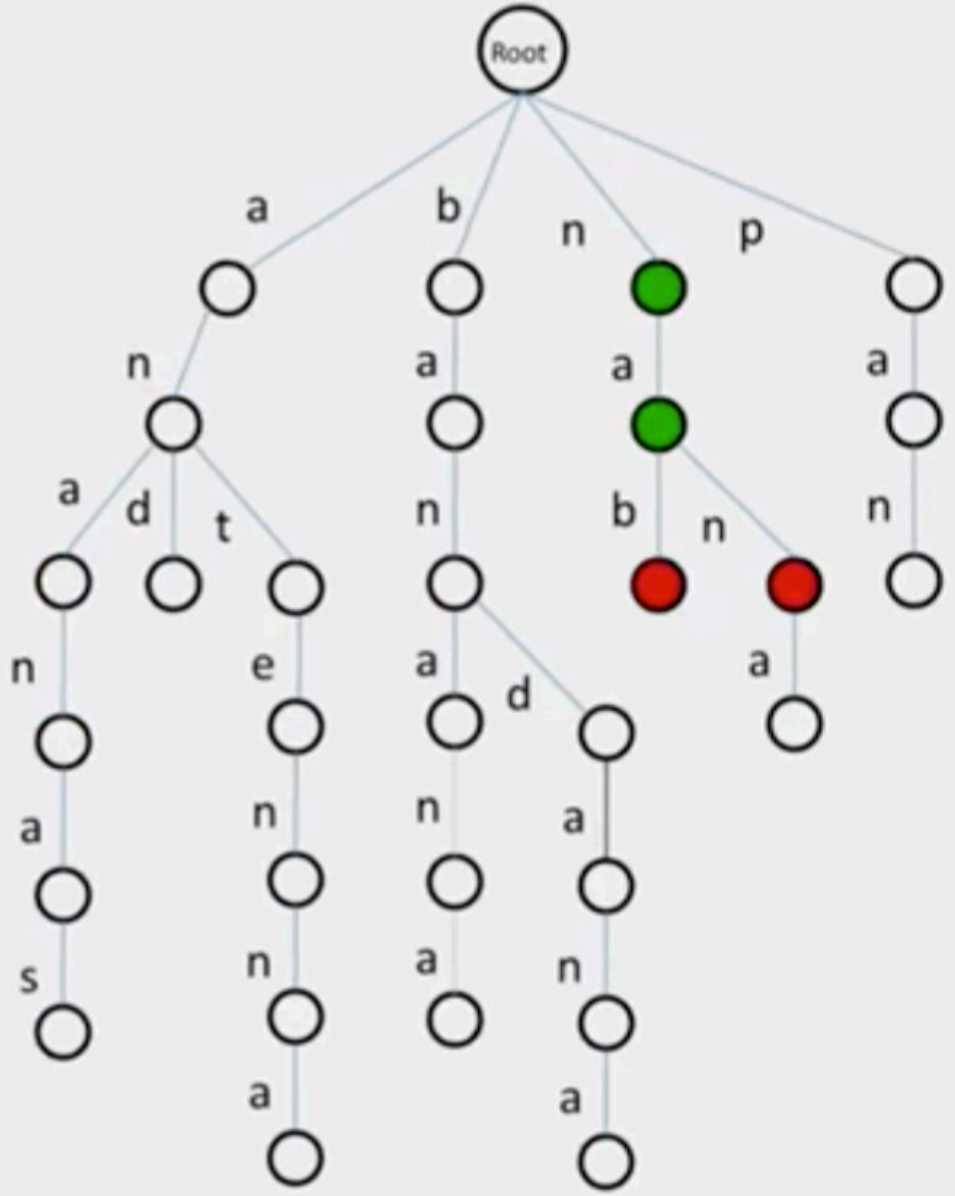
p a n a m a b a **n a n a s**



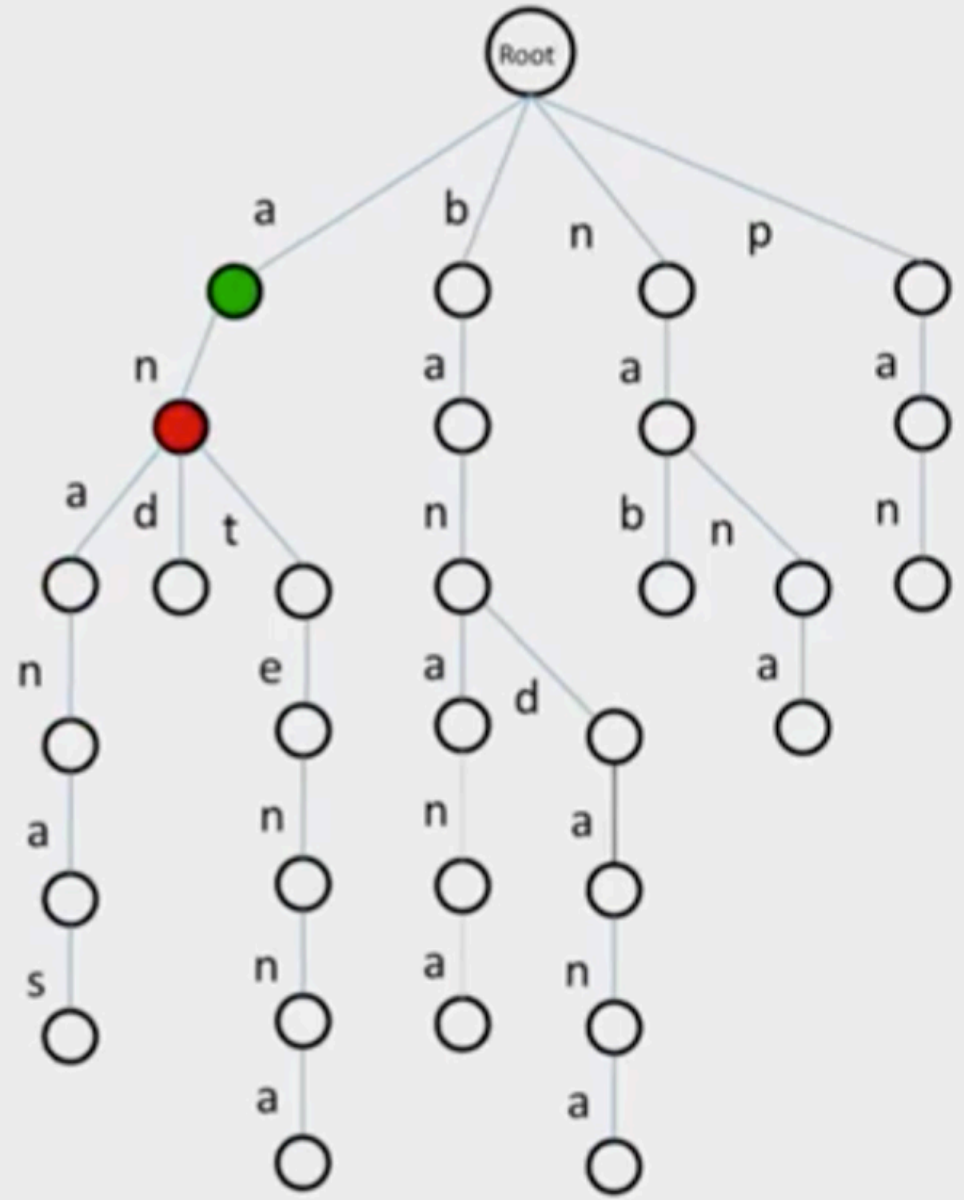
panamabananas



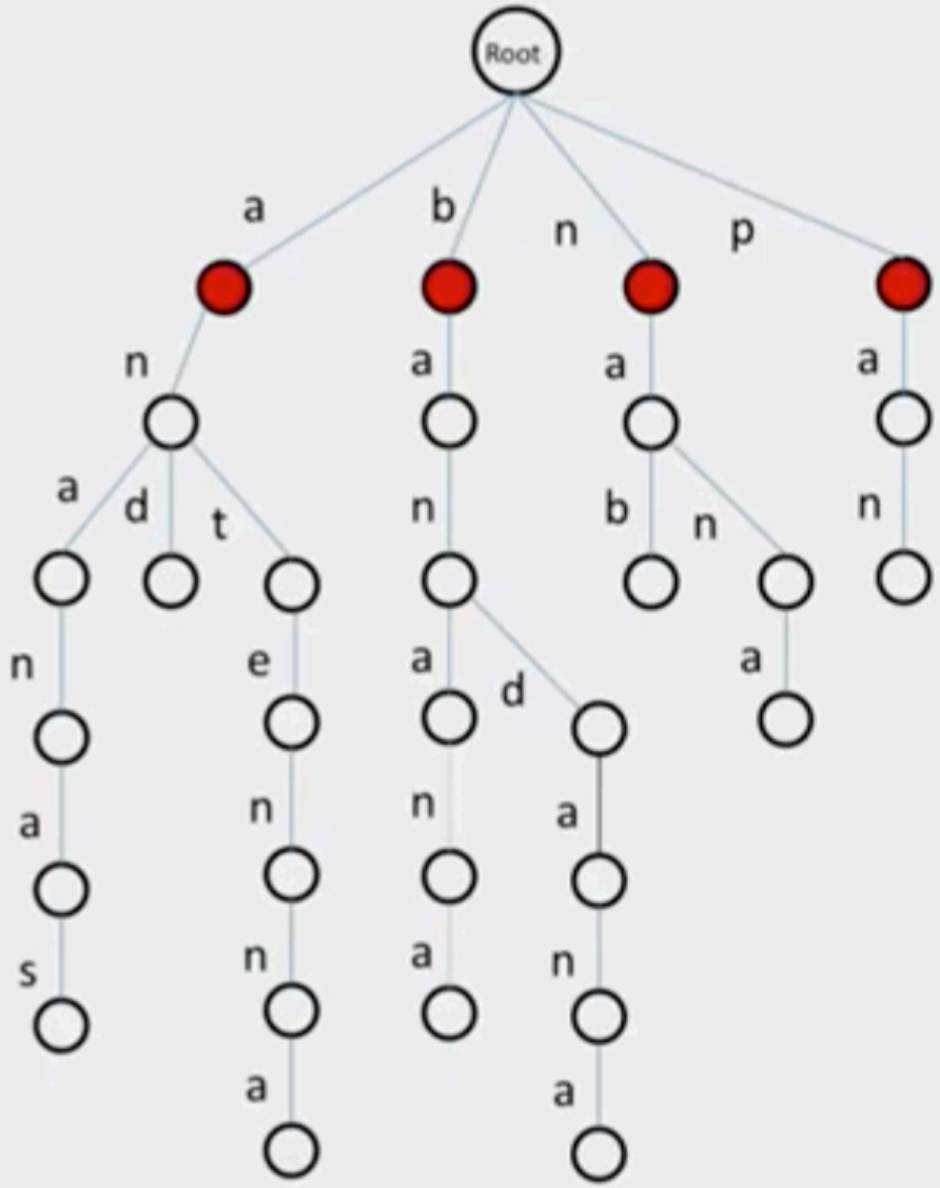
p a n a m a b a n a **n a s**



p a n a m a b a n a n a s



p a n a m a b a n a n a **s**



# Success!

- Runtime of Brute Force:
  - Total: O(|*Genome*|*|*Patterns*|)
- Runtime of Trie Matching:
  - Trie Construction: O(|*Patterns*|)
  - Pattern Matching: O(|*Genome*| * |*LongestPattern*|)

# Bowtie

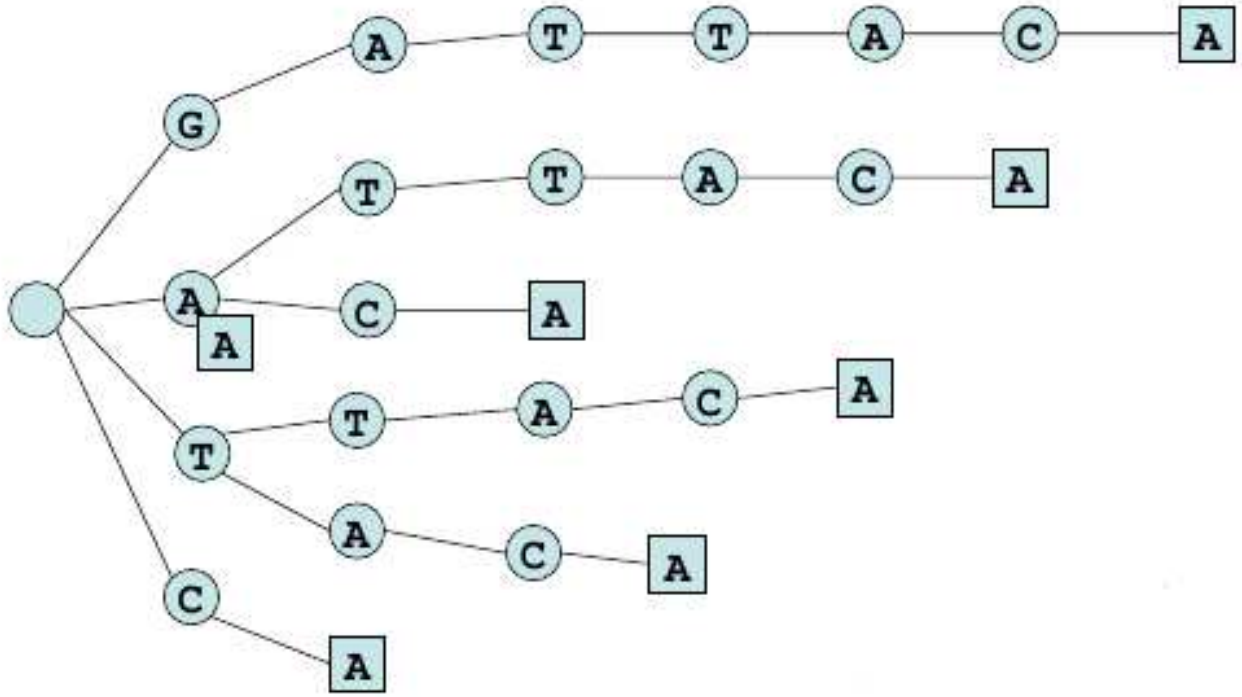
Bowtie is based on **Burrows Wheeler Transformation** (BWT) paired with a **Compressed Suffix Array** (CSA)

# Suffix trie

- Fact: $y$ occurs in $x$ if $y$ is a prefix of a suffix of $x$
- We build a trie with all the suffixes of the text $x$
- Example: $x =$ **GATTACA** we build the tree on:
  - **GATTACA**
  - **ATTACA**
  - **TTACA**
  - **TACA**
  - **ACA**
  - **CA**
  - **A**

# Suffix trie

**GATTACA**




The suffix trie collects in the internal nodes all the substrings of $x$

# Pattern Matching with Suffix Trees

- To find any pattern in a text:
  1. Build suffix tree for text of length $m$—O($m$) time
  2. Thread the pattern of length $n$ through the suffix tree of the text—O($n$) time.

- Therefore the runtime of the Pattern Matching Problem is only O($m + n$)!

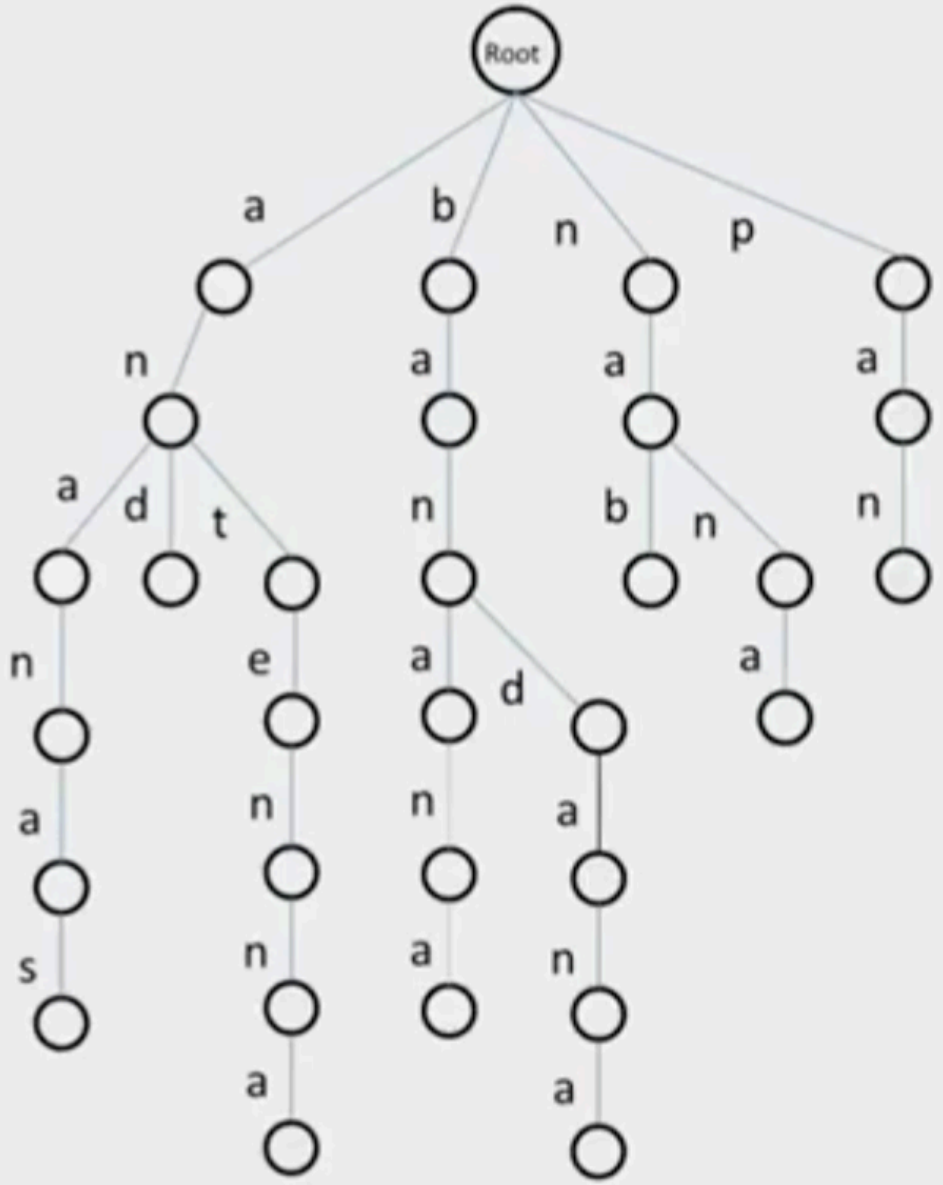
# Pattern Matching with Suffix Trees

SuffixTreePatternMatching(***p***,***t***)

1. Build **suffix tree** for text ***t***
2. Thread pattern ***p*** through **suffix tree**
3. **if** threading is complete
4. **output** positions of all ***p***-matching leaves in the tree
5. **else**
6. **output** "Pattern does not appear in text"

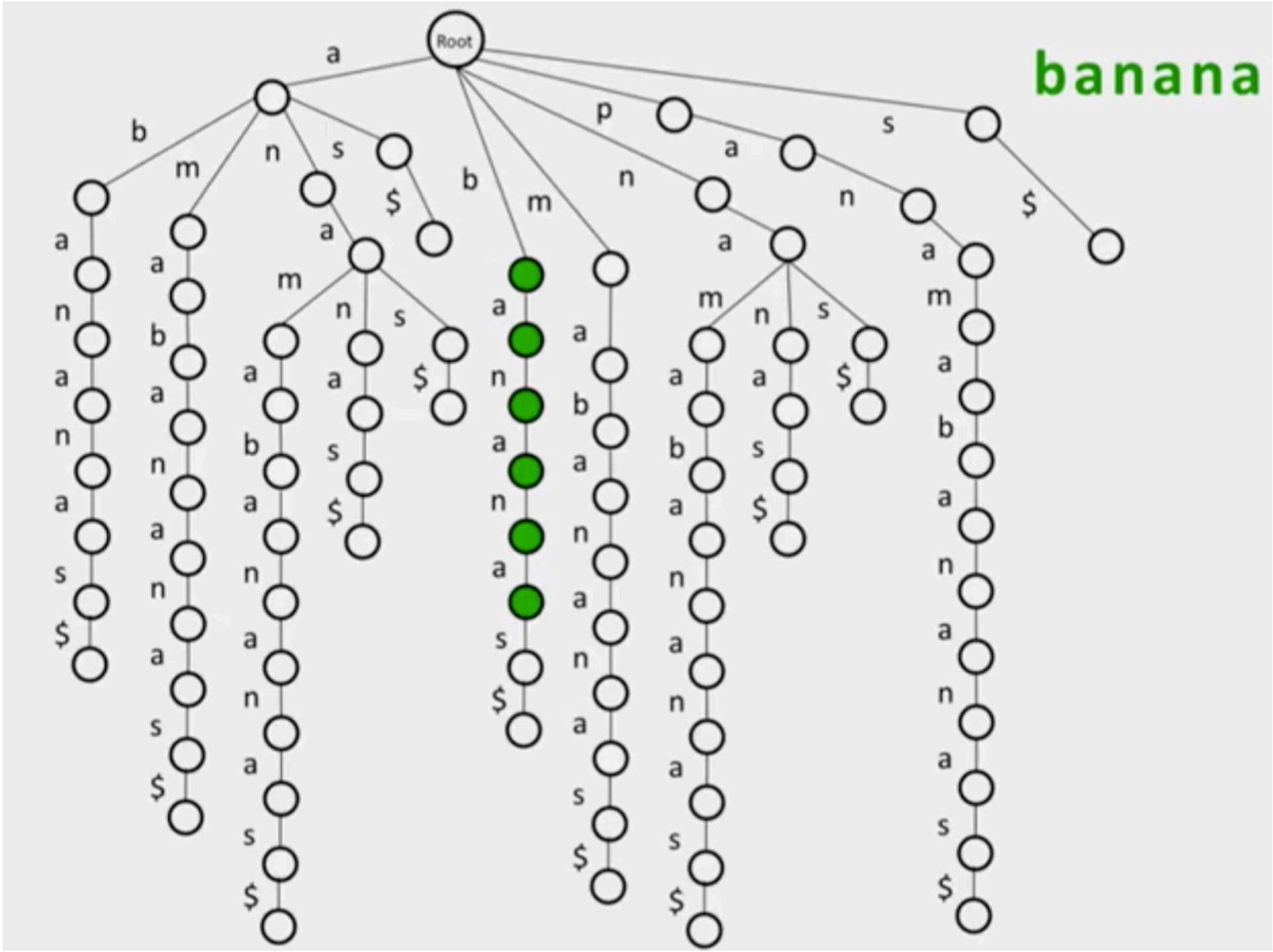
# Memory Analysis of TrieMatching

- Son completely forgot about memory!
- Worst case: # edges = O(|*Patterns*|)



# Preprocessing the Genome

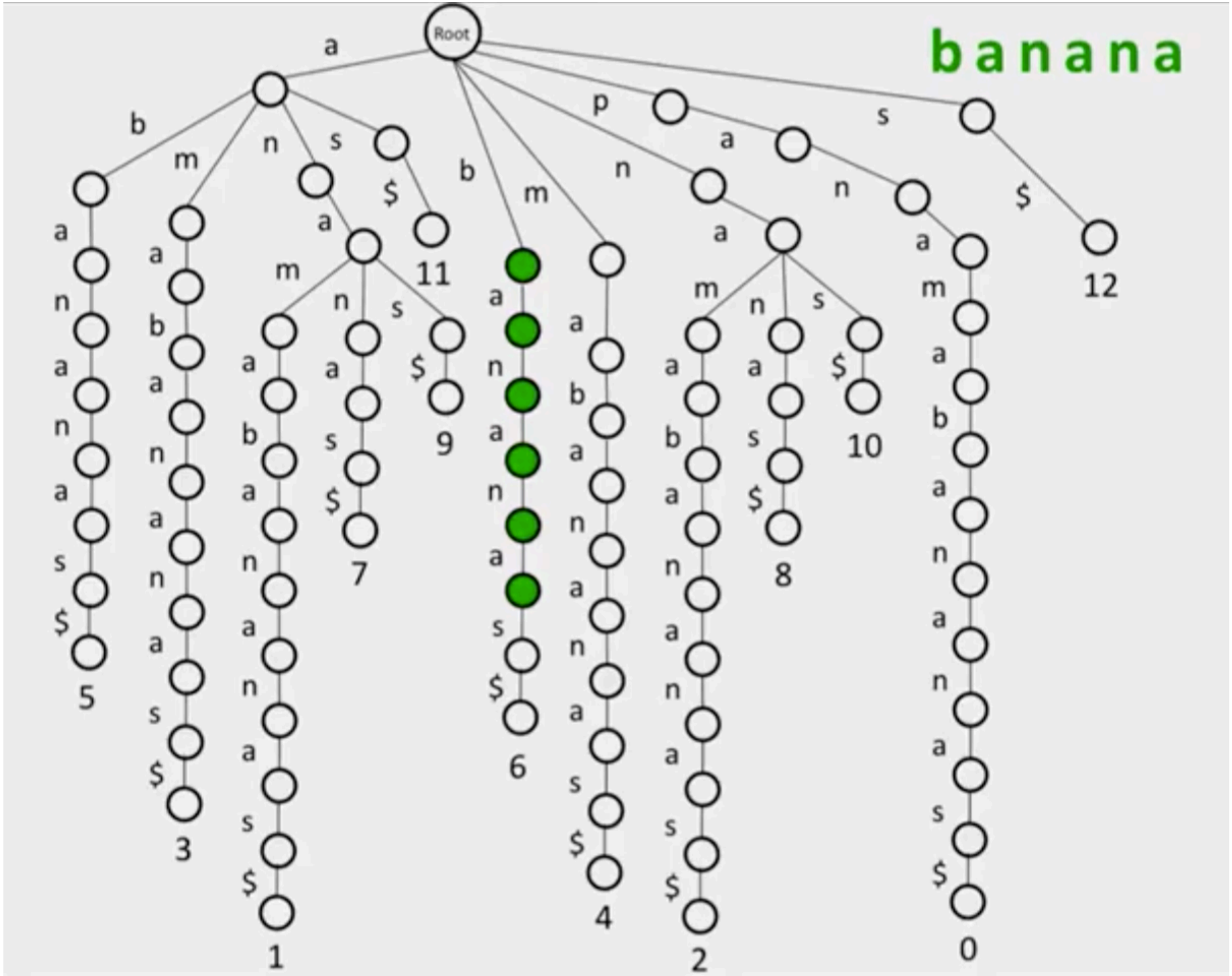
- Form all suffixes of *Genome.*
- How can we combine these suffixes into a data structure?
- Let's use a trie!

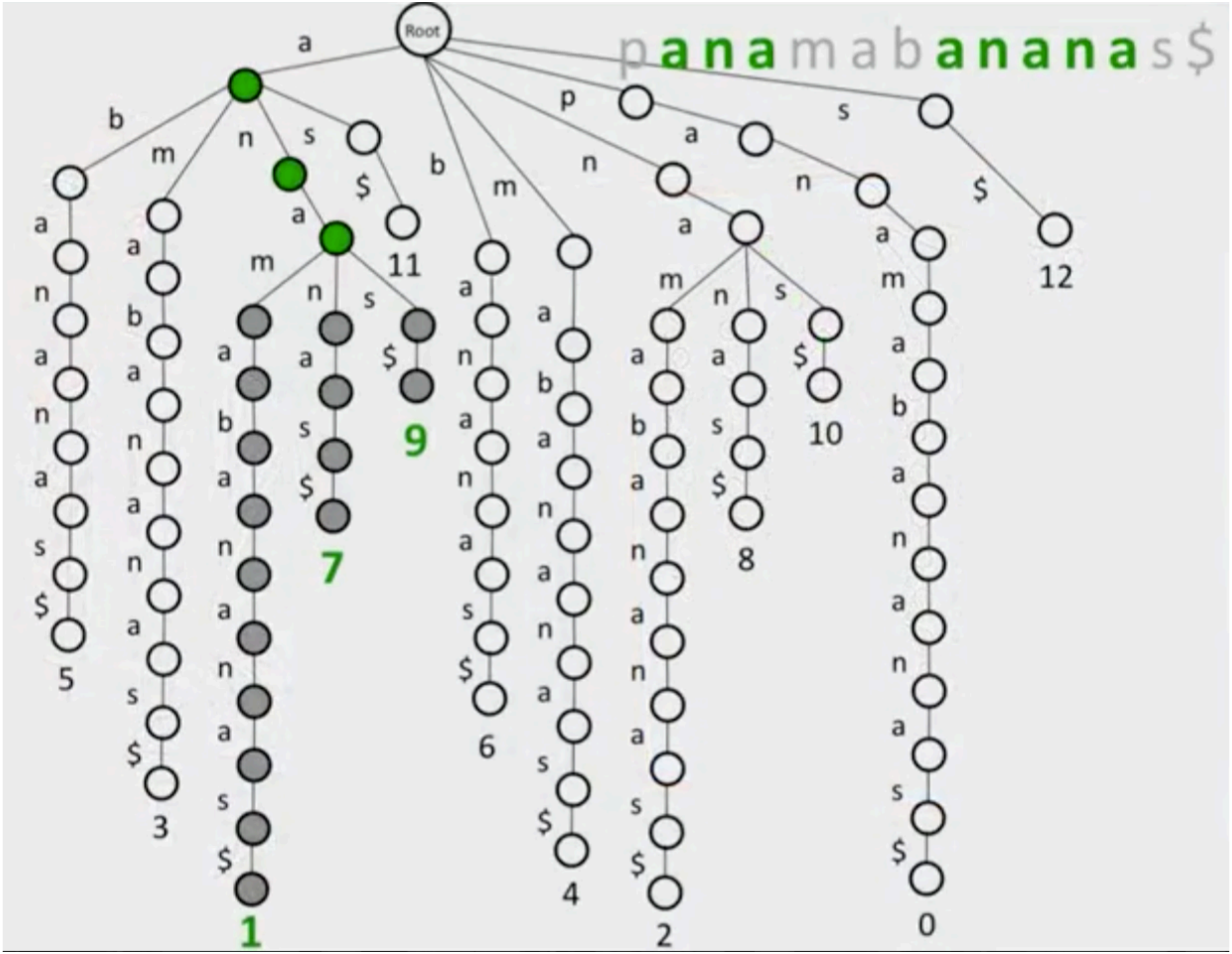

banana
Root
a
b
m
n
s
$
p

# Where Are the Matches?

- To find where the pattern matches are, we need to add a little more information to the suffix trie.

- At each leaf ($), we add the starting position in *Genome* of the suffix ending at that leaf.


banana
Root
a
b
m
n
s
$
p
a
n
s
$
a
n
a
n
a
s
$
5
a
b
a
n
a
n
a
s
$
3
m
a
n
a
n
a
s
$
1
a
n
a
s
$
7
n
s
$
9
11
b
m
a
n
a
n
a
s
$
6
a
b
a
n
a
n
a
s
$
4
m
a
n
a
n
a
s
$
2
n
a
s
$
8
s
$
10
a
n
a
n
a
s
$
0
$
12


Root
panamabanana s$
a
b
m
n
s
$
p
a
n
a
m
a
b
a
n
a
n
a
s
$
1
3
5
7
9
11
6
4
2
8
10
12
0

# Memory Trouble Once Again

- Worst case: the suffix trie holds O(|*Suffixes*|) nodes.

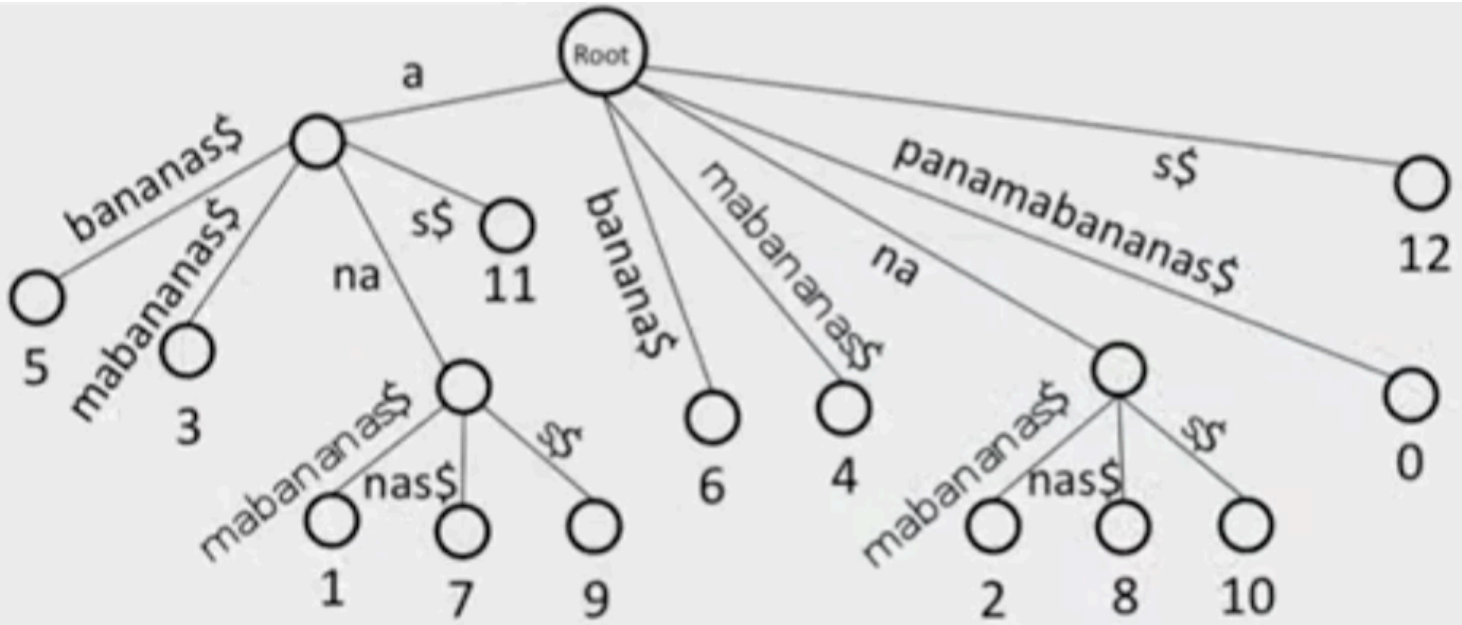
- For a *Genome* of length *n*, $|Suffixes| = n(n-1)/2 = O(n^2)$

*Suffixes*

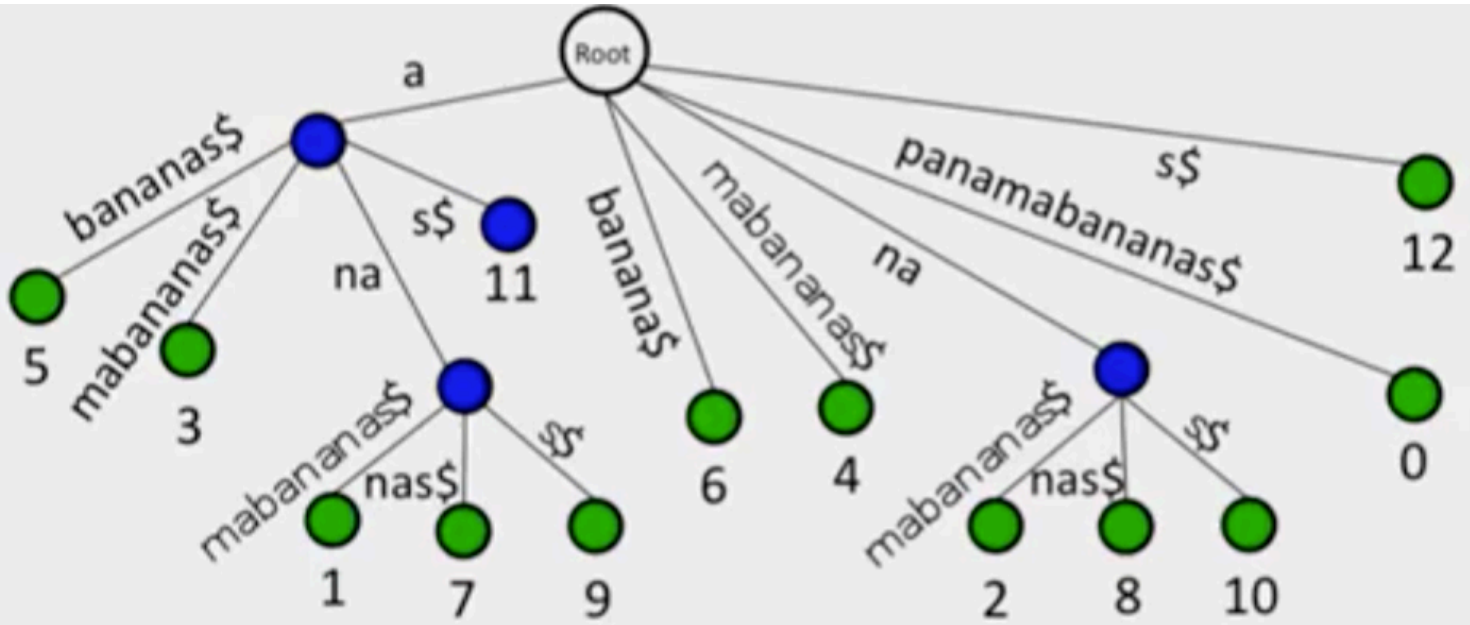
```
panamabananas$
anamabananas$
namabananas$
amabananas$
mabananas$
abananas$
bananas$
ananas$
nanas$
anas$
nas$
as$
s$
$
```

# Compressing the Trie

- This doesn't mean that our idea was bad!
- To reduce memory, we can compress each "nonbranching path" of the tree into an edge.




- This data structure is called a **suffix tree**.




- This data structure is called a **suffix tree**.

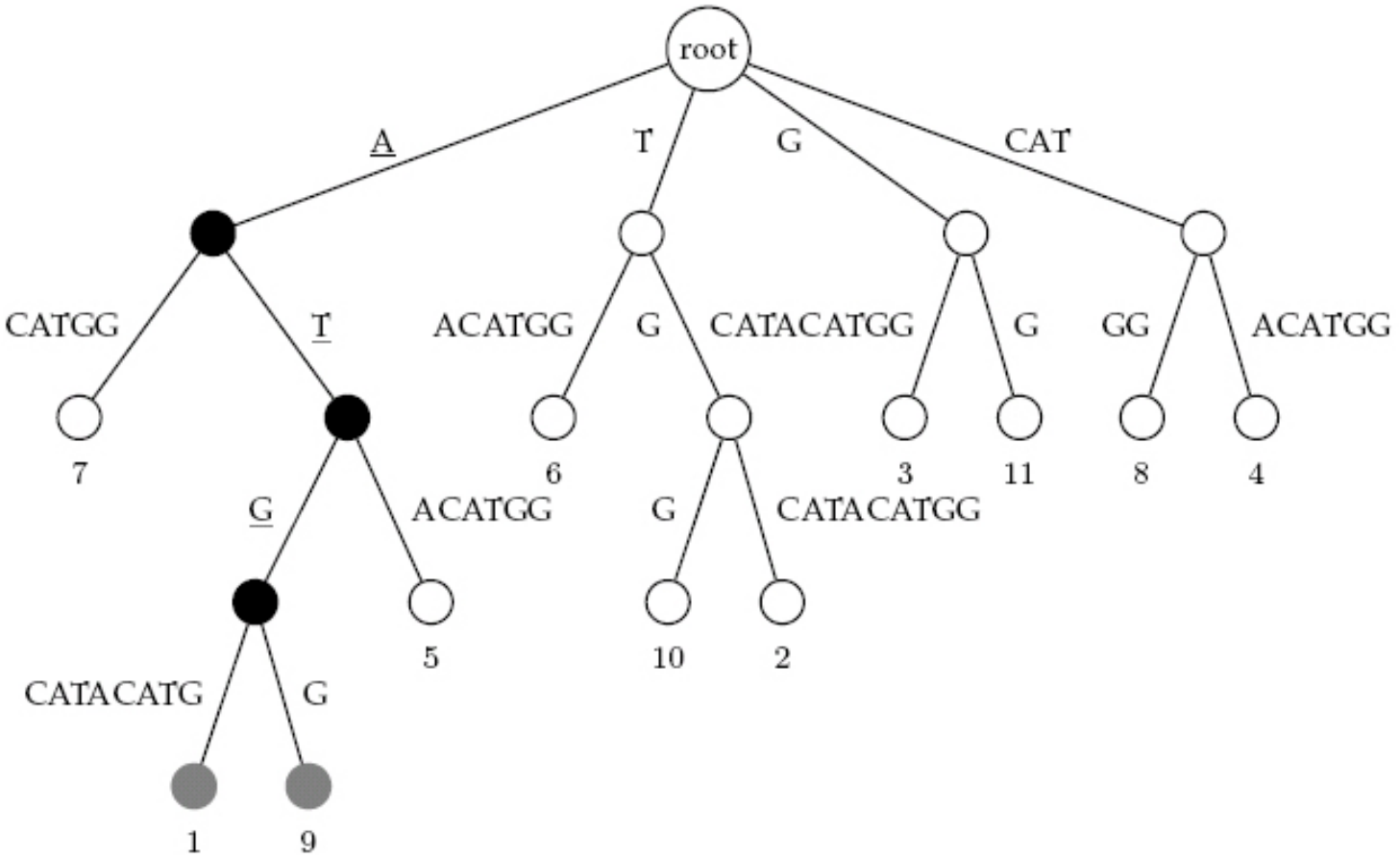
- For any *Genome*, # nodes < 2|*Genome*|.
  - **# leaves** = |*Genome*|;
  - **# internal nodes** < |*Genome*| − 1

# We are Not Finished Yet

- I am happy with the suffix tree, but I am not completely satisfied.
  - Runtime: O(|*Genome*| + |*Patterns*|)
  - Memory: O(|*Genome*|)

- However, big-O notation ignores constants!
  - The best known suffix tree implementations require ~ 20 times the length of |*Genome*|.
  - Can we reduce this constant factor?

# Threading ATG through a Suffix Tree



**Suffix Trees.** We can reduce the number of edges in the **suffix trie** by combining the edges on any non-branching path into a single edge. The resulting data structure is called ***suffix tree.***