CSE 549: Suffix Tries & Suffix Trees

Stony Brook University

All slides in this lecture **not** marked with “*” of Ben Langmead.

KMP is great, but

|T| = m |P| = n   (**note**: m,n are opposite from previous lecture)

	Without pre-processing (KMP)	**Given pre-processing (KMP)**	**Without pre-processing (ST)**	**Given pre-processing (ST)**
Find **an** occurrence of P	$O(m+n)$	$O(m)$	$O(m+n)$	$O(n)$
Find **all** occurrences of P	$O(m+n)$	$O(m)$	$O(m + n + k)$	$O(n+k)$
Find **an** occurrence of $P_1,\dots,P_\ell$	$O(\ell(m+n))$	$O(\ell(m))$	$O(m + \ell n)$	$O(\ell n)$

If the text is constant over many patterns, pre-processing the text rather than the pattern is better (and allows other efficient queries).

*

Tries

A trie (pronounced "try") is a rooted tree representing a collection of strings with one node per common prefix

Smallest tree such that:

Each edge is labeled with a character $c \in \Sigma$

A node has at most one outgoing edge labeled c, for $c \in \Sigma$

Each key is "spelled out" along some path starting at the root

Natural way to represent either a *set* or a *map* where keys are strings

This structure is also known as a Σ-tree

Tries: example

Represent this map with a trie:

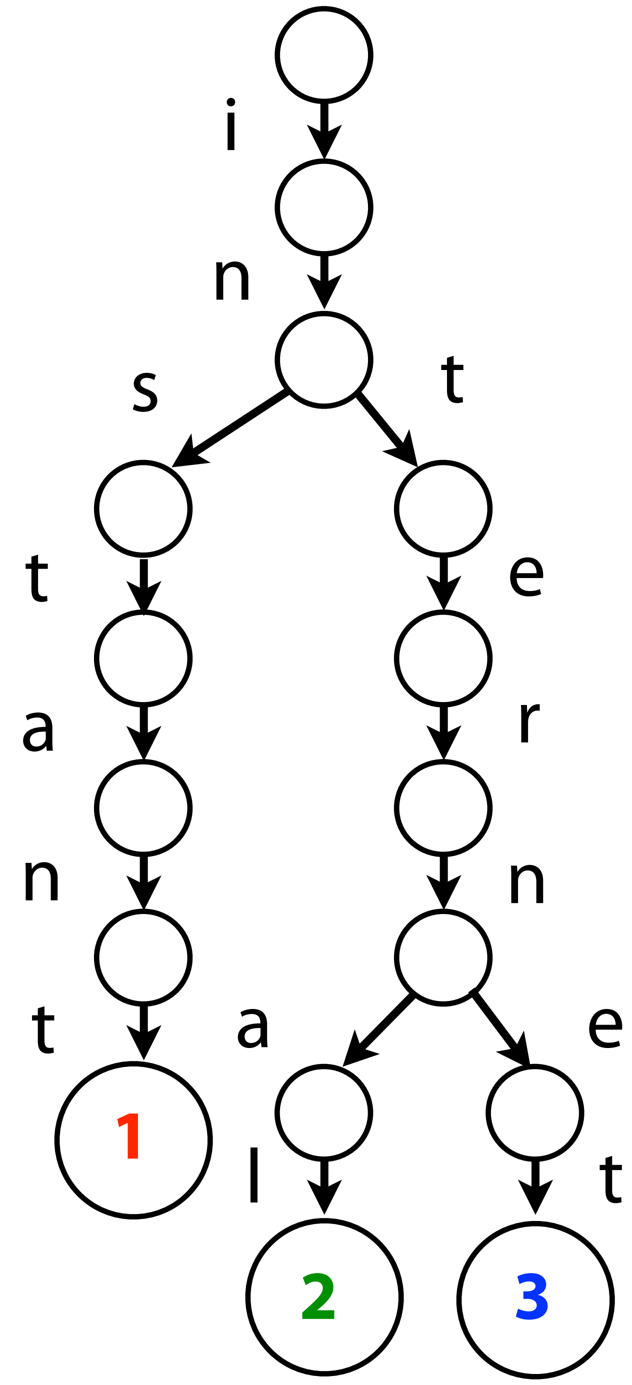
Key	Value
instant	1
internal	2
internet	3

The smallest tree such that:

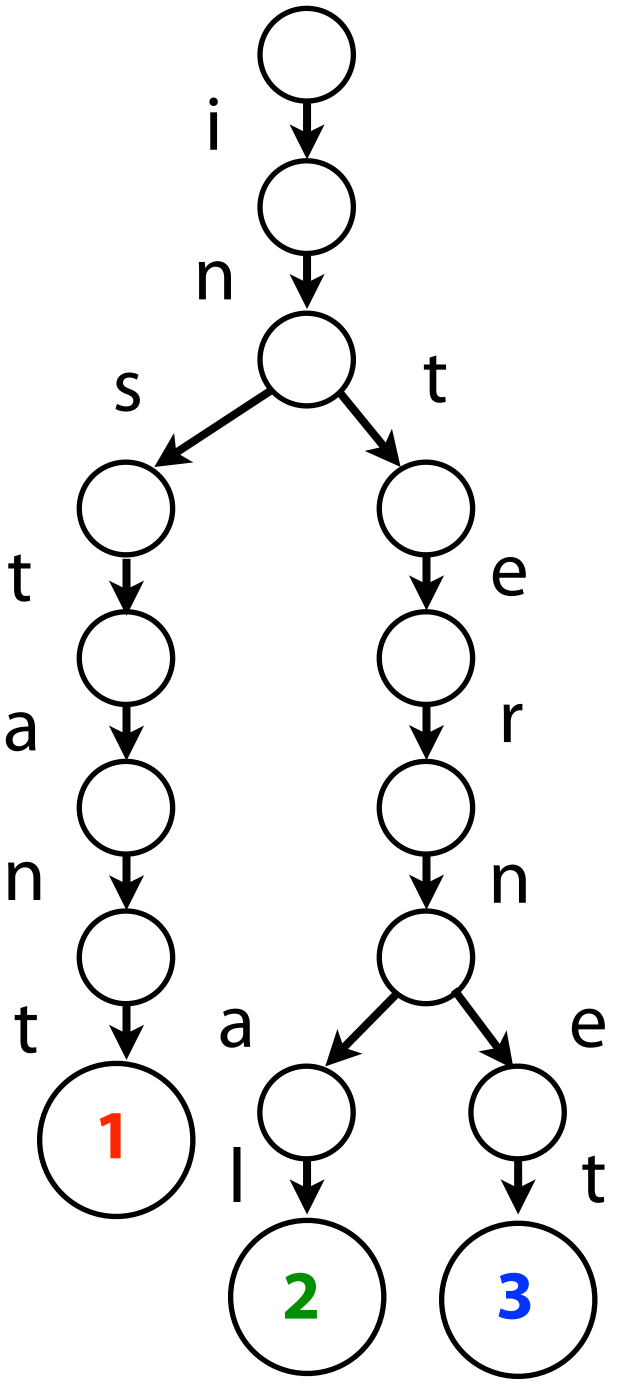
Each edge is labeled with a character $c \in \Sigma$

A node has at most one outgoing edge labeled c, for $c \in \Sigma$

Each key is "spelled out" along some path starting at the root



Tries: example



Checking for presence of a key *P*, where $n = |P|$, is ***O*(*n*)** time

If total length of all keys is *N*, trie has ***O*(*N*)** nodes

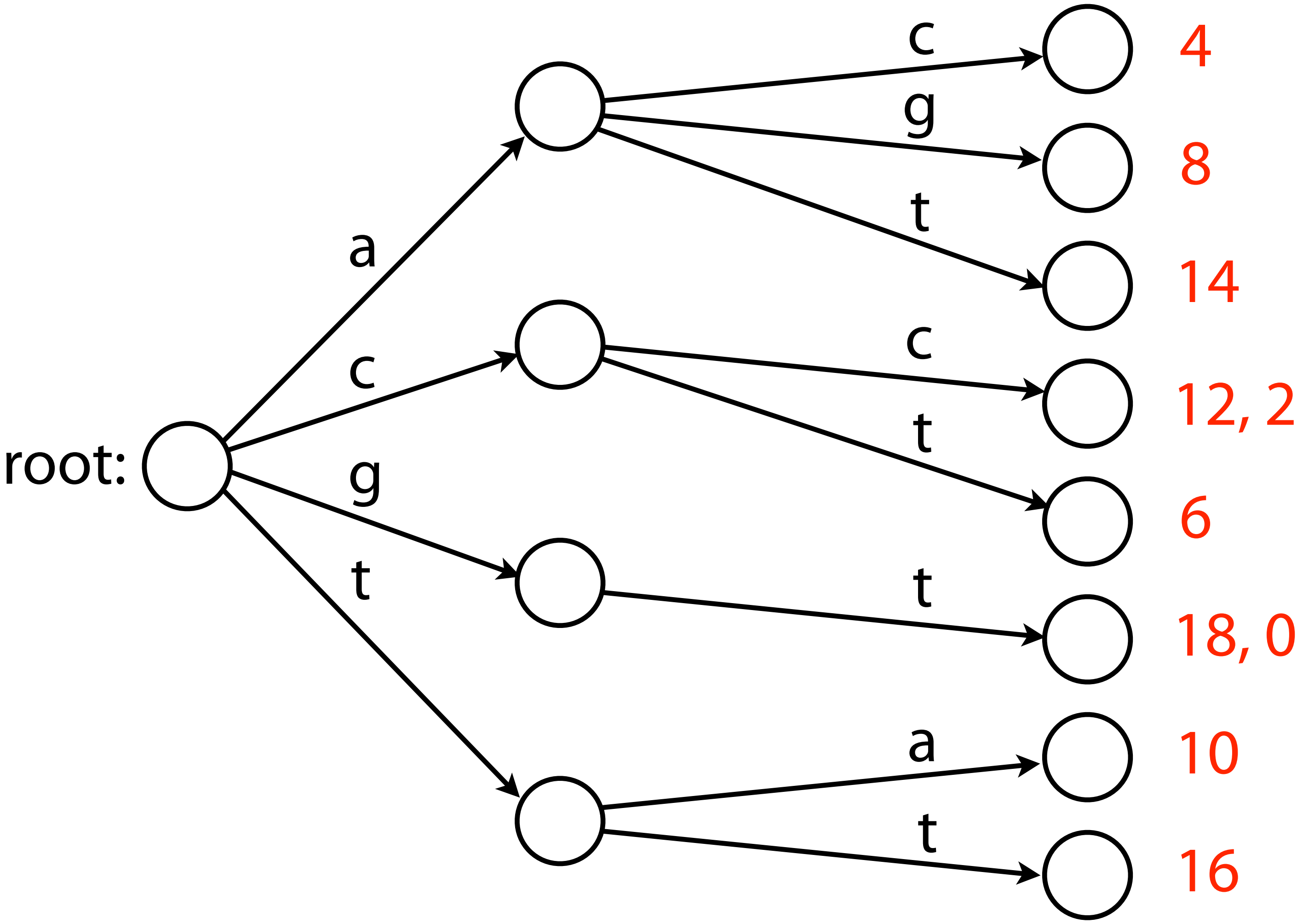
What about $|\Sigma|$?

Depends how we represent outgoing edges. If we don't assume $|\Sigma|$ is a small constant, it shows up in one or both bounds.

Tries: another example

We can index *T* with a trie. The trie maps substrings to offsets where they occur

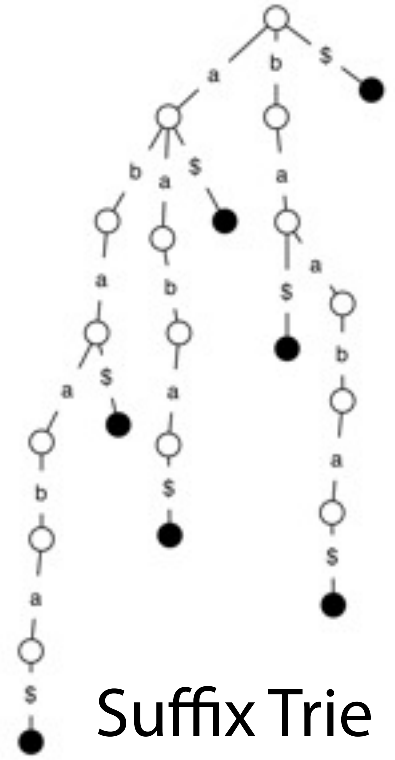
ac	4
ag	8
at	14
cc	12
cc	2
ct	6
gt	18
gt	0
ta	10
tt	16



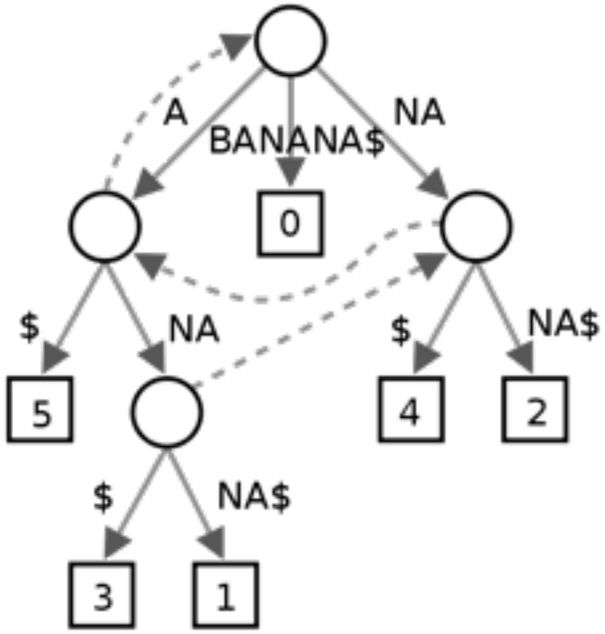
Indexing with suffixes

Some indices (e.g. the inverted index) are based on extracting substrings from *T*

A very different approach is to extract *suffixes* from *T*. This will lead us to some interesting and practical index data structures:



Suffix Trie



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

$ BANAN**A**
A $BANA**N**
A NA$BA**N**
A NANA$**B**
B ANANA**$**
N A$BAN**A**
N ANA$B**A**

FM Index

Trie Definitions

A Σ-tree (trie) is a rooted tree where each edge is labeled with a single character c ϵ Σ, such that no node has two outgoing edges labeled with the same character.

- for a node v in T, **depth**(v) or **node-depth**(v) is the distance from v to the root.
- **node-depth**(r) = 0
- **string**(v) = concatenation of all characters on the path r ⇝ v
- **string-depth**(v) = |**string**(v)| (note: **string-depth**(v) ≥ **node-depth**(v))
- for a string x, if ∃ node v with **string**(v) = x, we say **node**(x) = v
- T **displays** string x if ∃ node v and string y such that xy = **string**(v)
- **words**(T) = { x | T **displays** x}
- A **suffix trie** of string s is a Σ-tree such that **words**(T) = {s' | s' is a substring of s}
- An **internal**/**leaf** edge leads to an **internal**/**leaf** node

*

Defs. from: http://profs.sci.univr.it/~liptak/ALBioinfo/files/sequence_analysis.pdf

Suffix trie

Build a **trie** containing all **suffixes** of a text *T*

T: GTTATAGCTGATCGCGGCGTAGCGG

GTTATAGCTGATCGCGGCGTAGCGG
TTATAGCTGATCGCGGCGTAGCGG
TATAGCTGATCGCGGCGTAGCGG
ATAGCTGATCGCGGCGTAGCGG
TAGCTGATCGCGGCGTAGCGG
AGCTGATCGCGGCGTAGCGG
GCTGATCGCGGCGTAGCGG
CTGATCGCGGCGTAGCGG
TGATCGCGGCGTAGCGG
GATCGCGGCGTAGCGG
ATCGCGGCGTAGCGG
TCGCGGCGTAGCGG
CGCGGCGTAGCGG
GCGGCGTAGCGG
CGGCGTAGCGG
GGCGTAGCGG
GCGTAGCGG
CGTAGCGG
GTAGCGG
TAGCGG
AGCGG
GCGG
CGG
GG
G

$m(m+1)/2$ chars

Suffix trie

First add special *terminal character* **$** to the end of *T*

$ is a character that does not appear elsewhere in *T*, and we define it to be less than other characters (for DNA: **$** < **A** < **C** < **G** < **T**)

$** enforces a rule we're all used to using: e.g. "as" comes before "ash" in the dictionary. **$ also guarantees no suffix is a prefix of any other suffix.

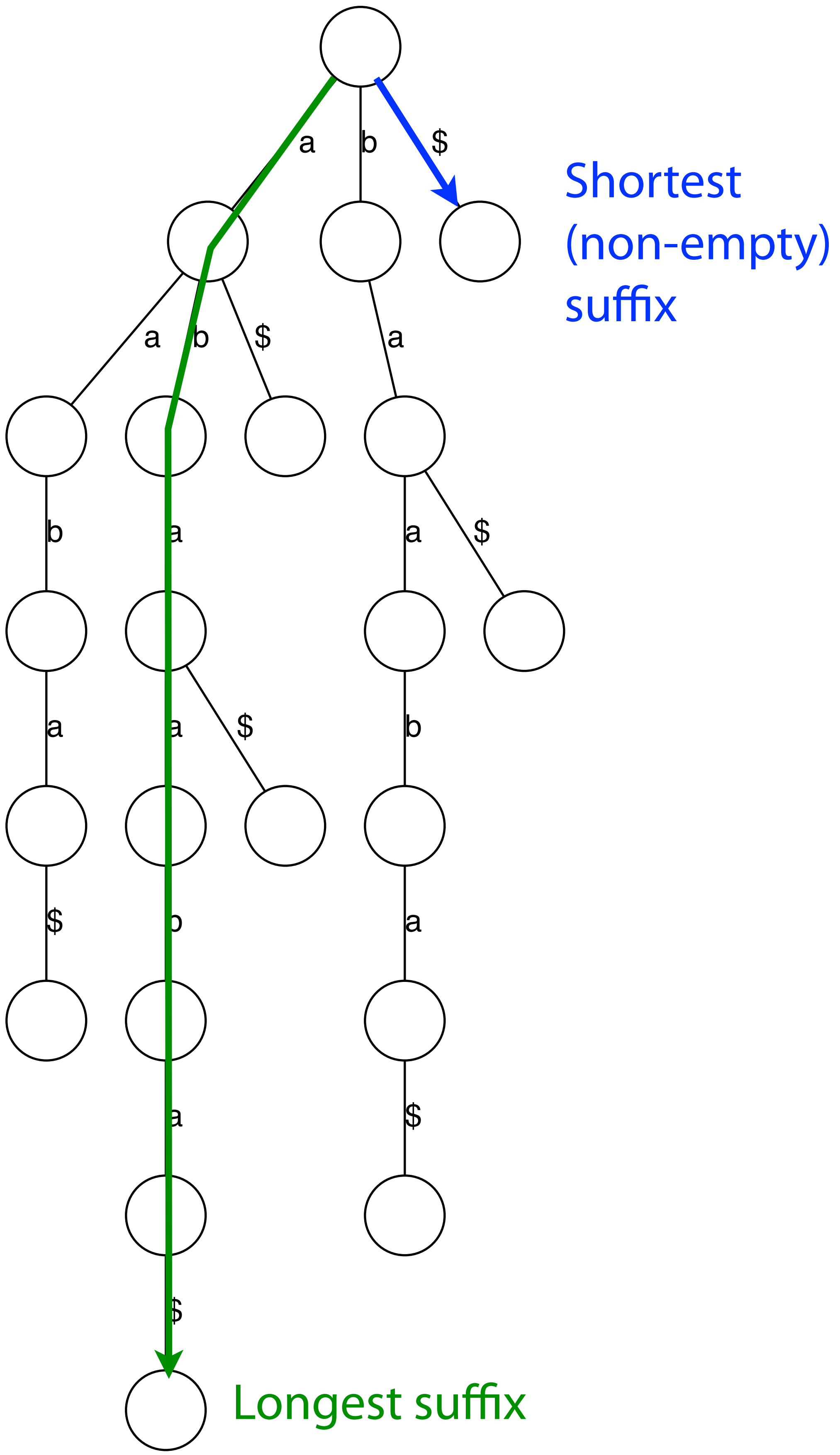
```
T: GTTATAGCTGATCGCGGCGTAGCGG$
   GTTATAGCTGATCGCGGCGTAGCGG$
    TTATAGCTGATCGCGGCGTAGCGG$
     TATAGCTGATCGCGGCGTAGCGG$
      ATAGCTGATCGCGGCGTAGCGG$
       TAGCTGATCGCGGCGTAGCGG$
        AGCTGATCGCGGCGTAGCGG$
         GCTGATCGCGGCGTAGCGG$
          CTGATCGCGGCGTAGCGG$
           TGATCGCGGCGTAGCGG$
            GATCGCGGCGTAGCGG$
             ATCGCGGCGTAGCGG$
              TCGCGGCGTAGCGG$
               CGCGGCGTAGCGG$
                GCGGCGTAGCGG$
                 CGGCGTAGCGG$
                  GGCGTAGCGG$
                   GCGTAGCGG$
```

Suffix trie

T: abaaba *T***$***:* abaaba**$**

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**?

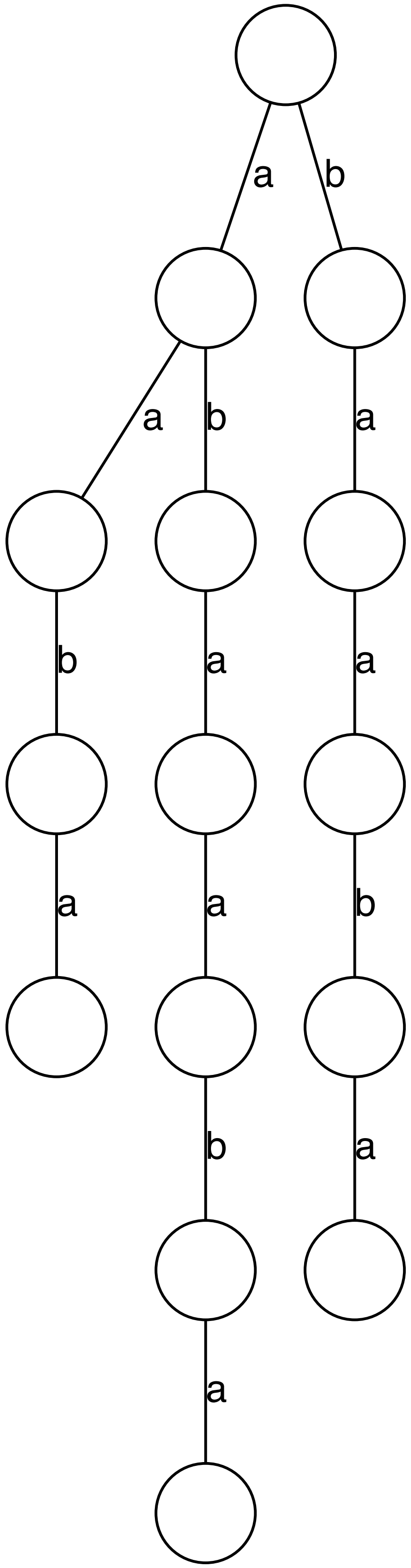


Suffix trie

T: abaaba

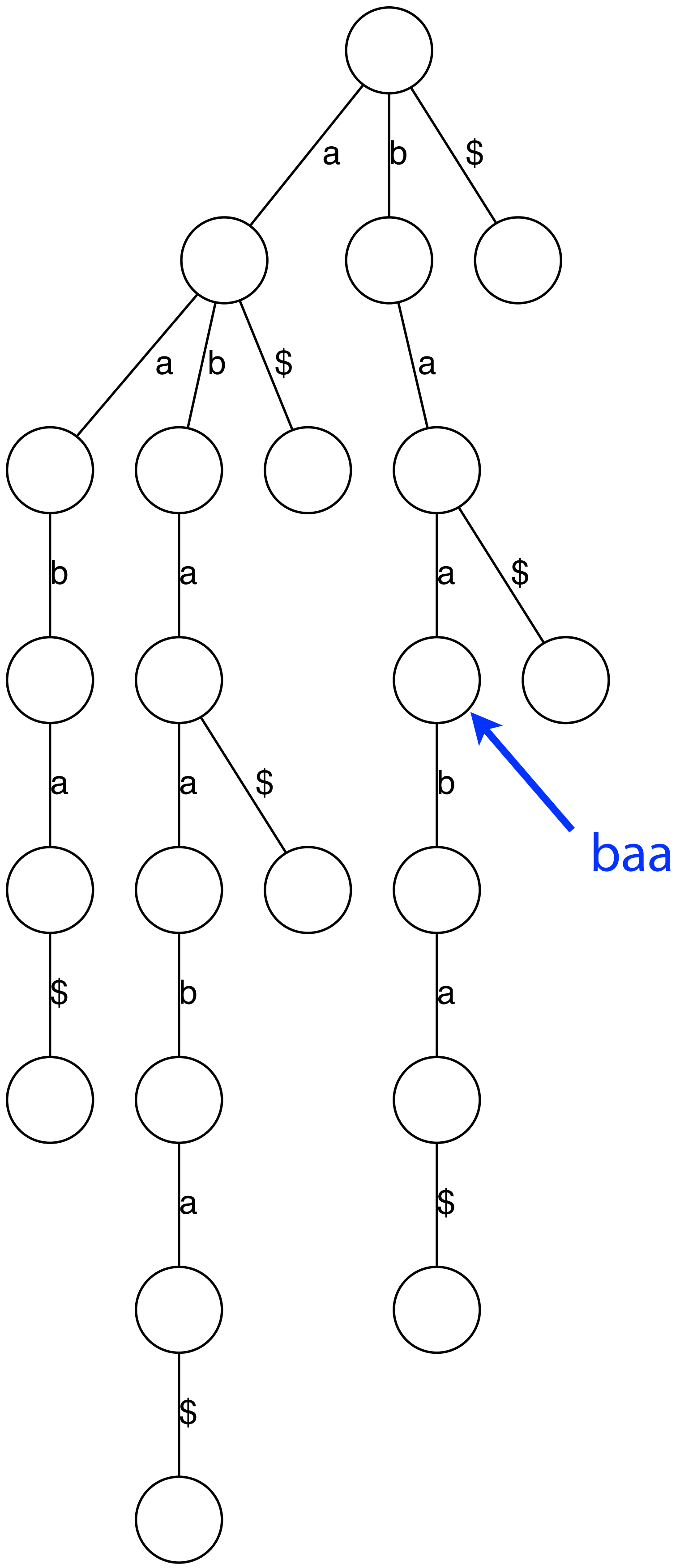
Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**? **No**



Suffix trie

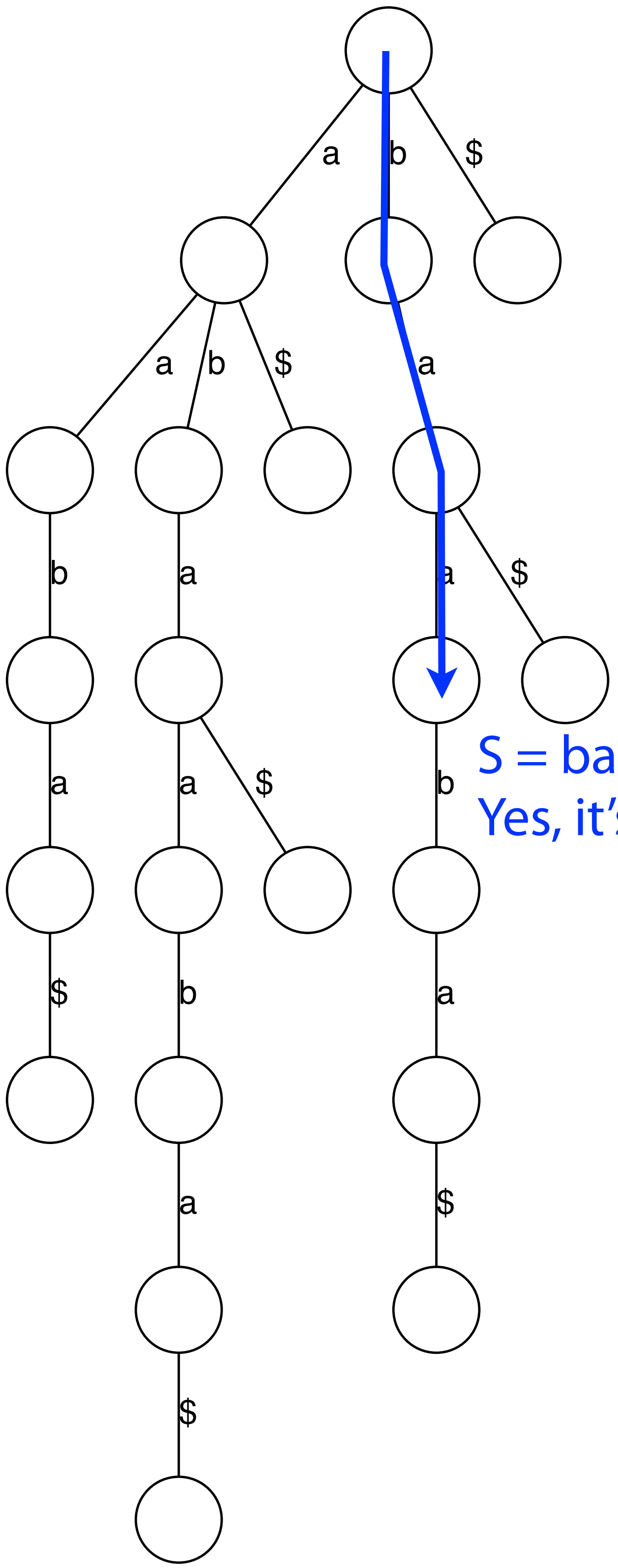
We can think of nodes as having **labels**, where the label spells out characters on the path from the root to the node



Suffix trie

How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.



S = baa
Yes, it's a substring

Suffix trie

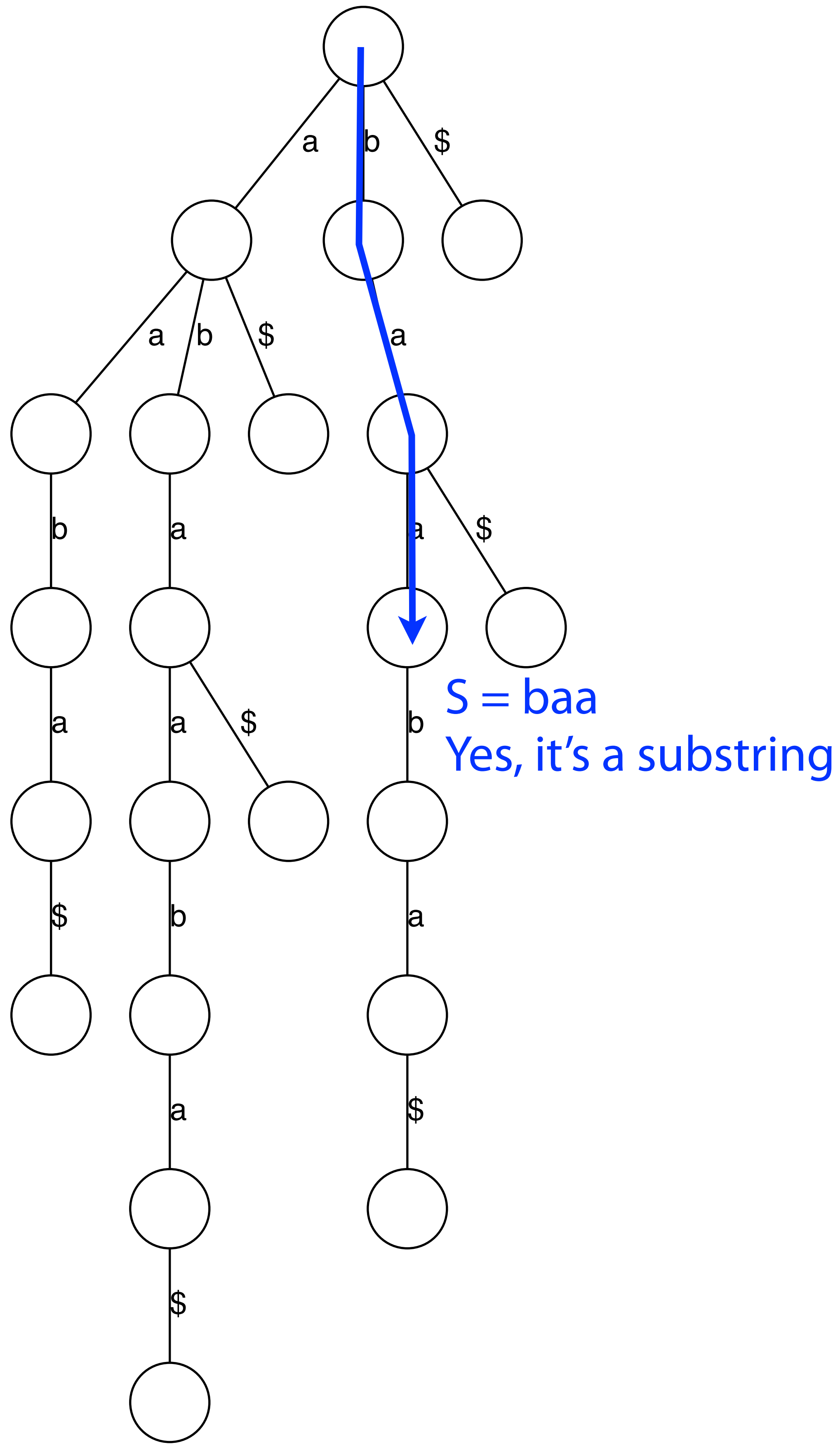
How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

If we exhaust *S* without falling off, *S* is a substring of *T*



Suffix trie

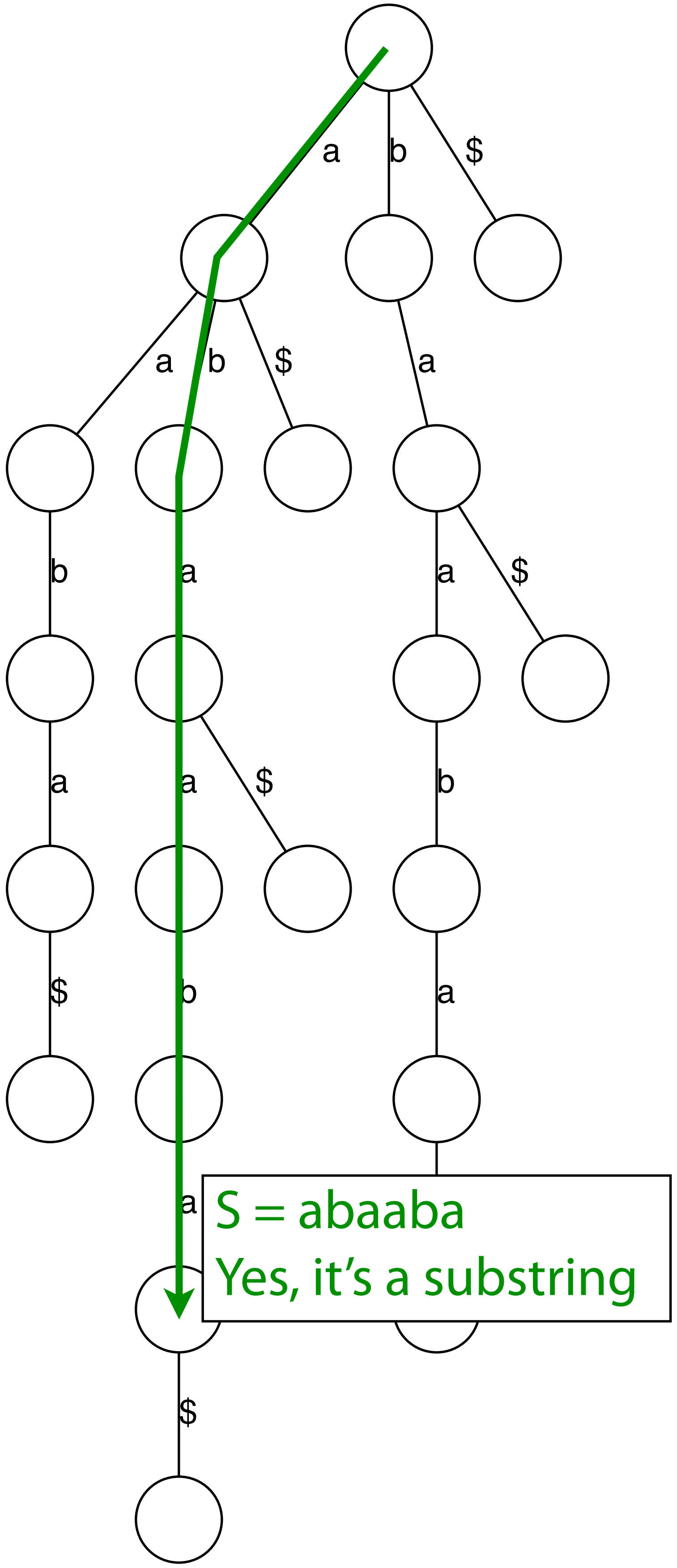
How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

  If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

  If we exhaust *S* without falling off, *S* is a substring of *T*



S = abaaba
Yes, it's a substring

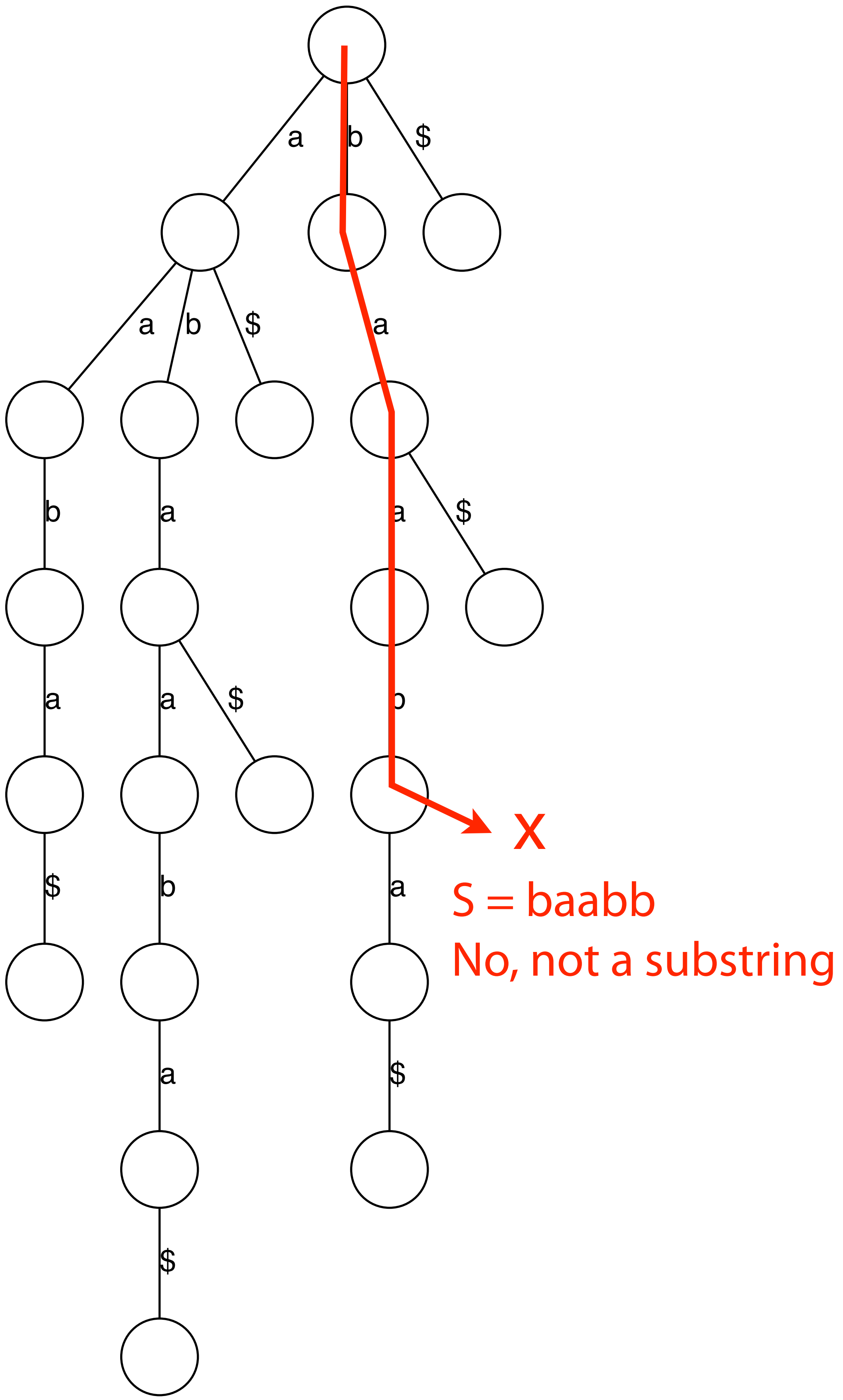
Suffix trie

How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

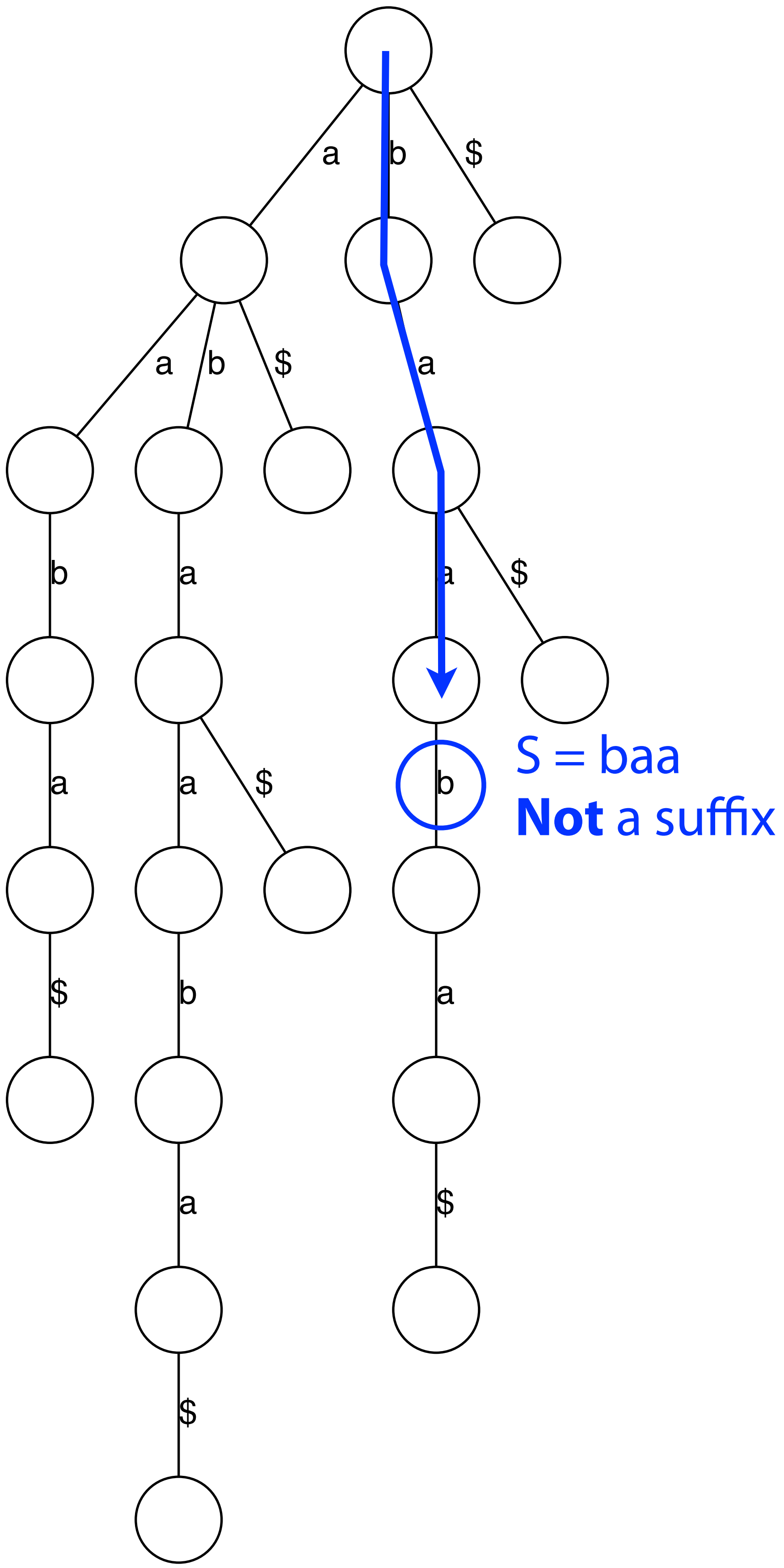
Start at the root and follow the edges labeled with the characters of *S*

- If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*
- If we exhaust *S* without falling off, *S* is a substring of *T*



Suffix trie

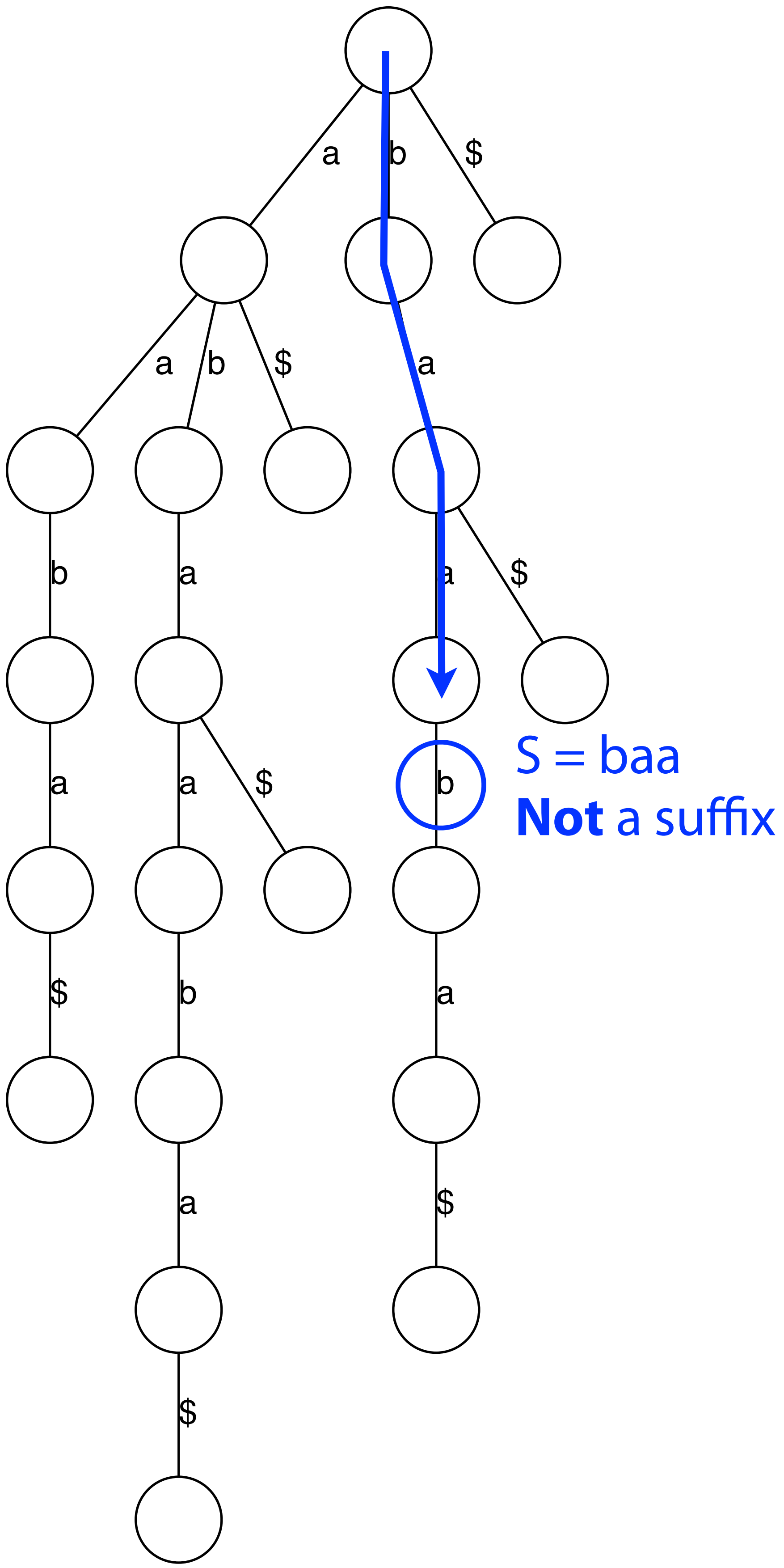
How do we check whether a string *S* is a **suffix** of *T*?



Suffix trie

How do we check whether a string *S* is a **suffix** of *T*?

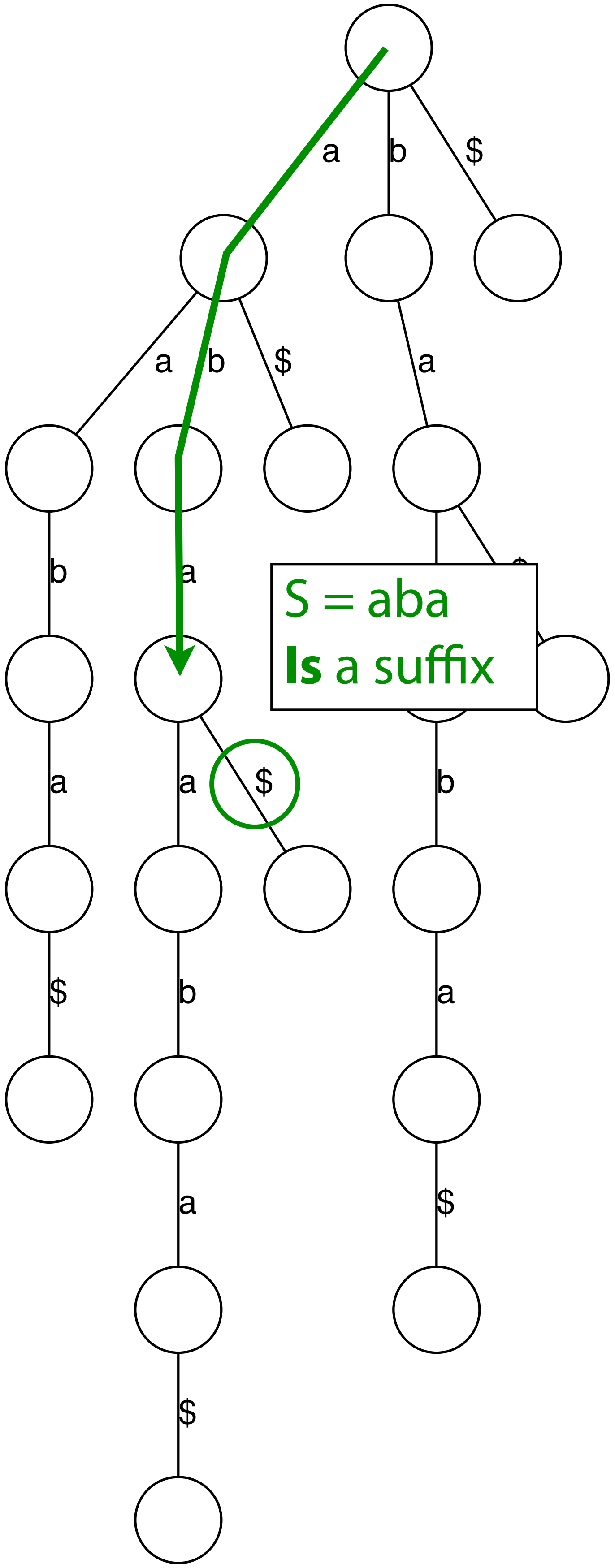
Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled **$**



Suffix trie

How do we check whether a string *S* is a **suffix** of *T*?

Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled **$**

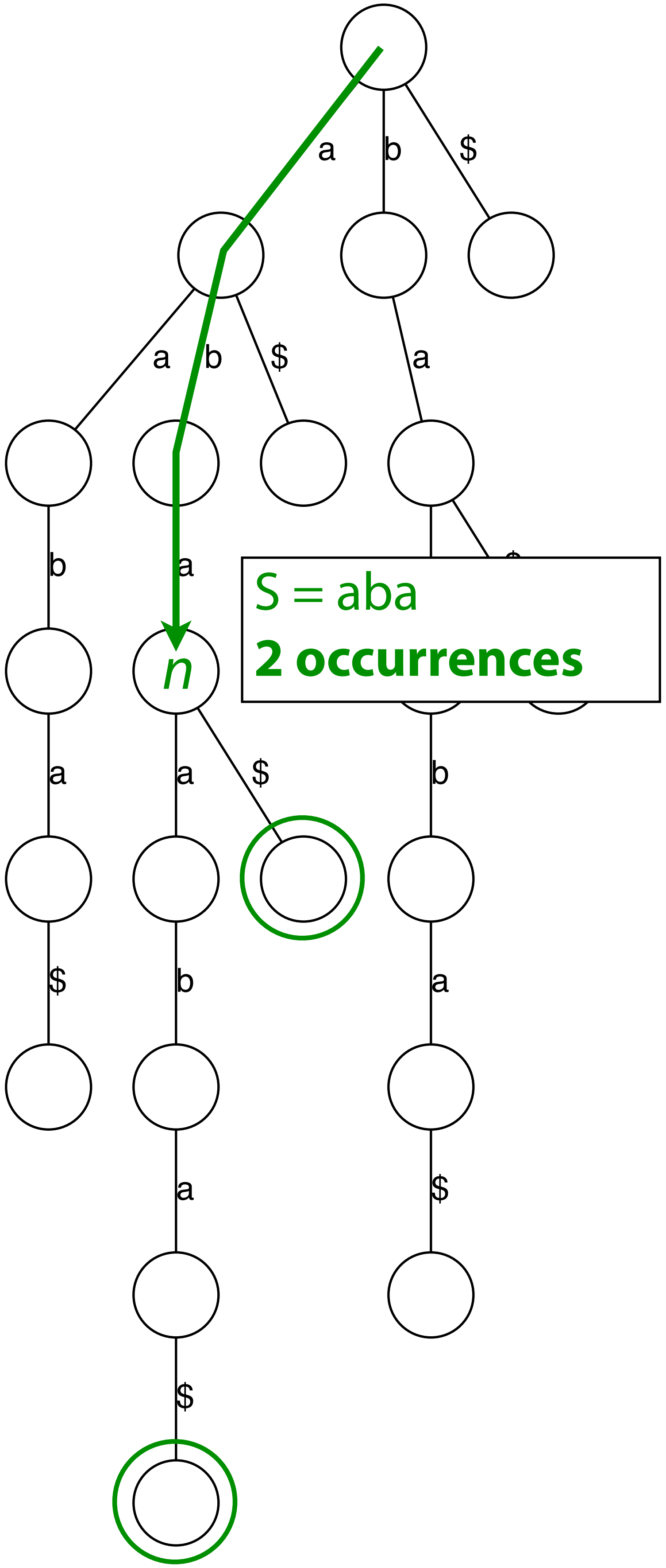


S = aba

Is a suffix

Suffix trie

How do we count the **number of times** a string *S* occurs as a substring of *T*?



S = aba

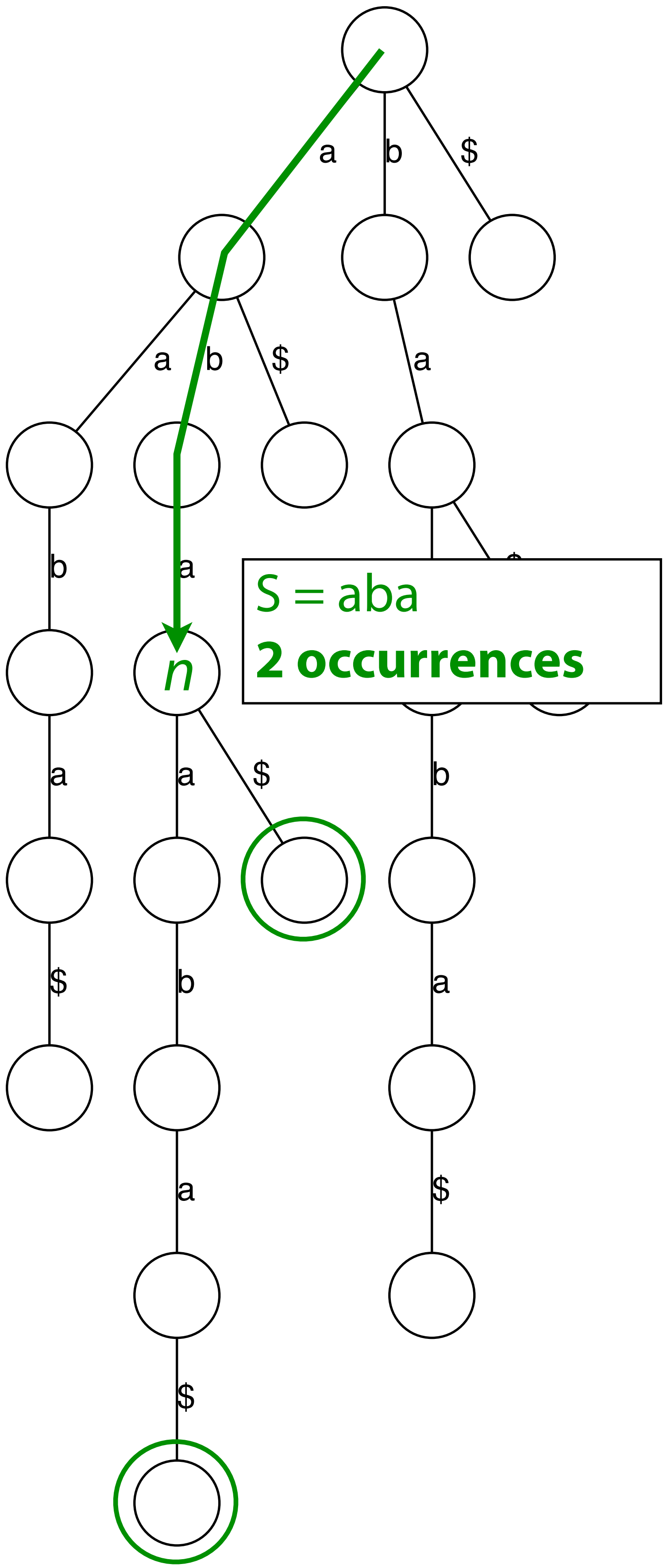
2 occurrences

Suffix trie

How do we count the **number of times** a string *S* occurs as a substring of *T*?

Follow path corresponding to *S*. Either we fall off, in which case answer is 0, or we end up at node *n* and the answer = # of leaf nodes in the subtree rooted at *n*.

Leaves can be counted with depth-first traversal.

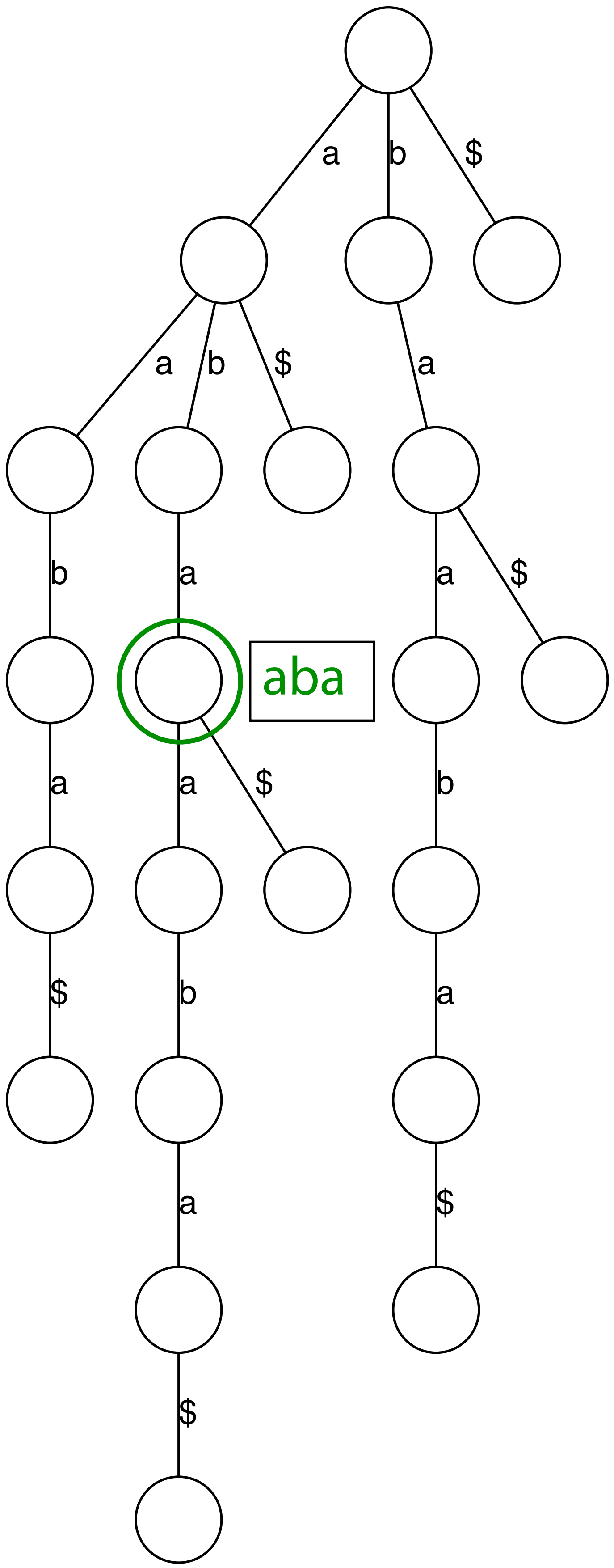


S = aba

2 occurrences

Suffix trie

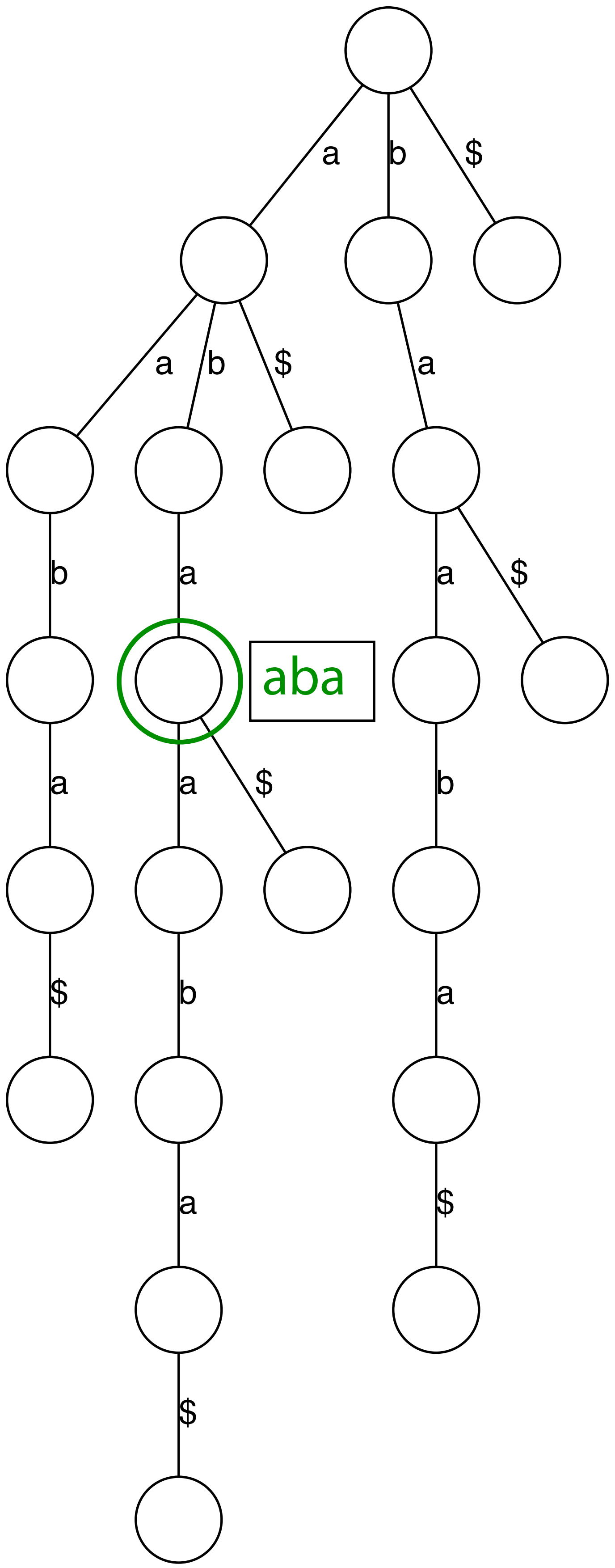
How do we find the **longest repeated substring** of *T*?



Suffix trie

How do we find the **longest repeated substring** of *T*?

Find the deepest node with more than one child



Suffix trie

How many nodes does the suffix trie have?

Is there a class of string where the number of suffix trie nodes grows linearly with *m*?

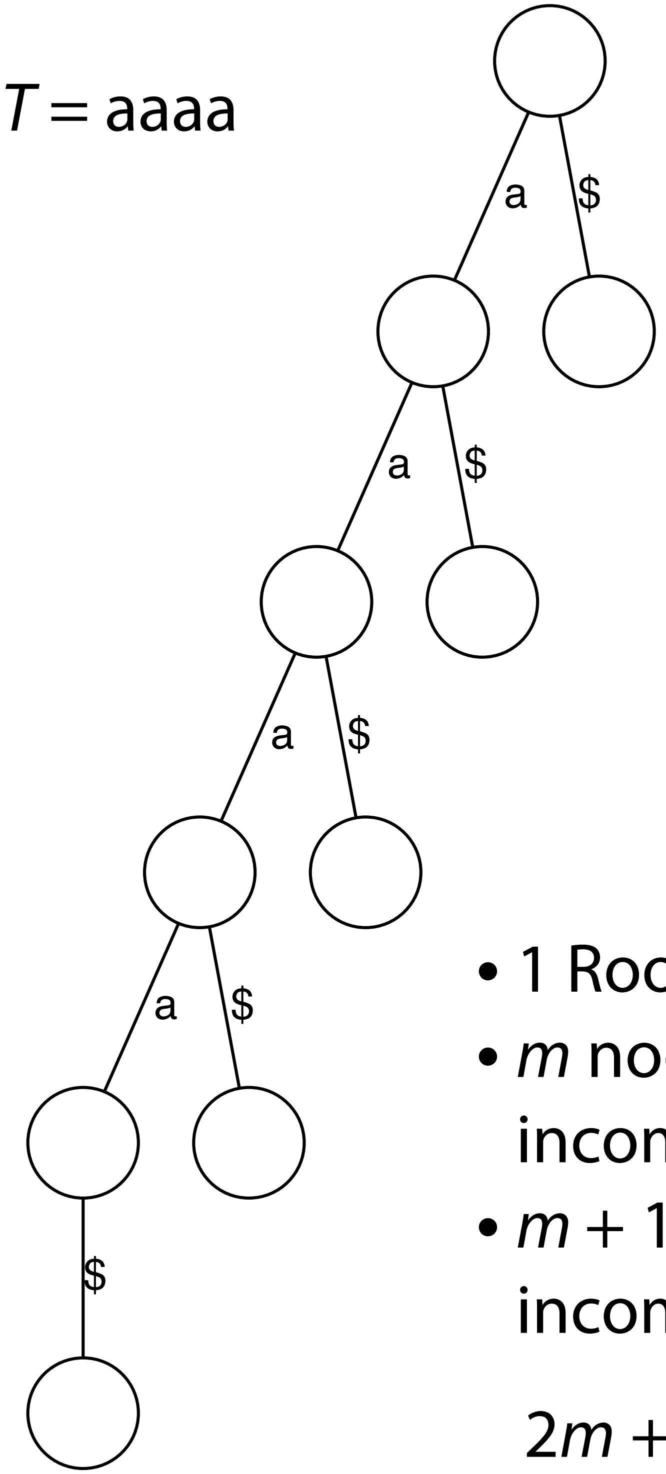
Suffix trie

How many nodes does the suffix trie have?

Is there a class of string where the number of suffix trie nodes grows linearly with m?

Yes: e.g. a string of m a's in a row (a^m)

T = aaaa



- 1 Root
- m nodes with incoming **a** edge
- $m + 1$ nodes with incoming **$** edge

$2m + 2$ nodes

Suffix trie

Is there a class of string where the number of suffix trie nodes grows with m^2?

Suffix trie

Is there a class of string where the number of suffix trie nodes grows with m^2?

Yes: $a^n b^n$

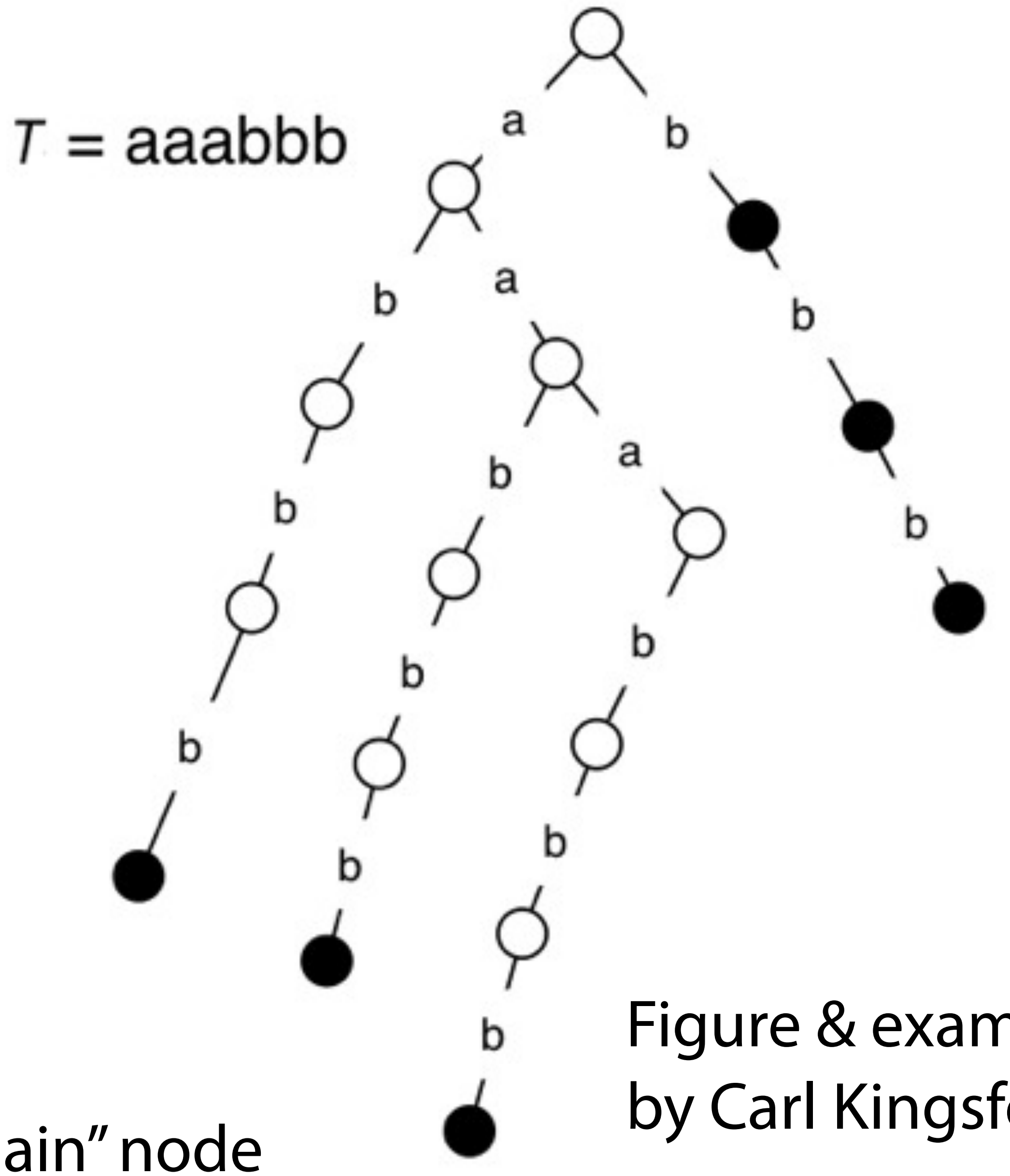


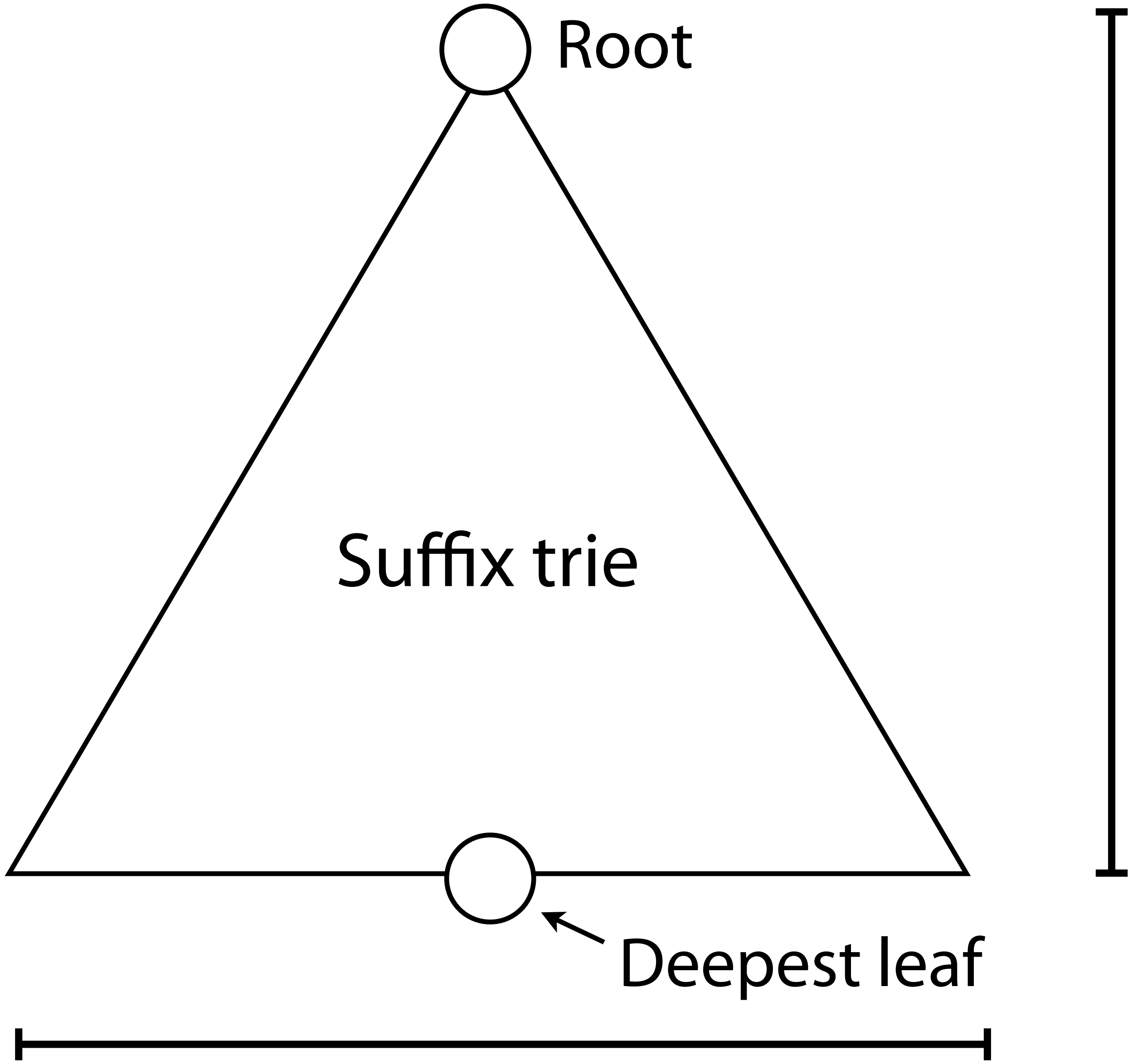
Figure & example by Carl Kingsford

- 1 root
- n nodes along "b chain," right
- n nodes along "a chain," middle
- n chains of n "b" nodes hanging off each "a chain" node
- $2n + 1$ **$** leaves (not shown)

$n^2 + 4n + 2$ nodes, where $m = 2n$

Suffix trie: upper bound on size

Could worst-case # nodes be worse than $O(m^2)$?



Root

Suffix trie

Deepest leaf

Max # nodes from *top to bottom*
= length of longest suffix + 1
= $m + 1$

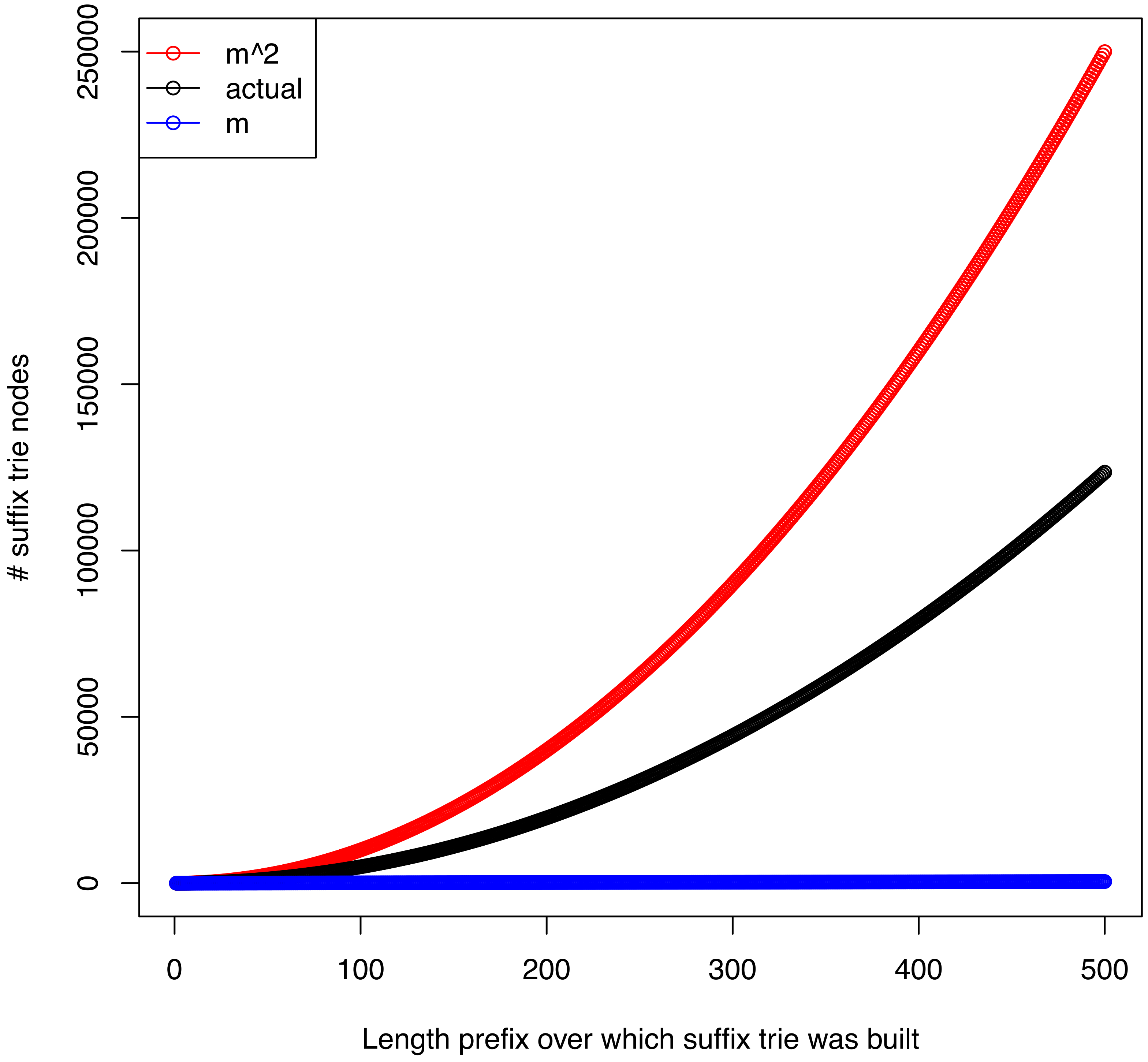
Max # nodes from *left to right*
= max # distinct substrings of any length
≤ m

$O(m^2)$ is worst case

Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length



Suffix tries -> Suffix trees

Suffix Tree Definitions

A **Σ^+**-tree is a rooted tree, T, where each edge is labeled with *non-empty strings*, where no node has two outgoing edges labeled with strings having the same *first* character. T is **compact** if all internal nodes have ≥ 2 children.

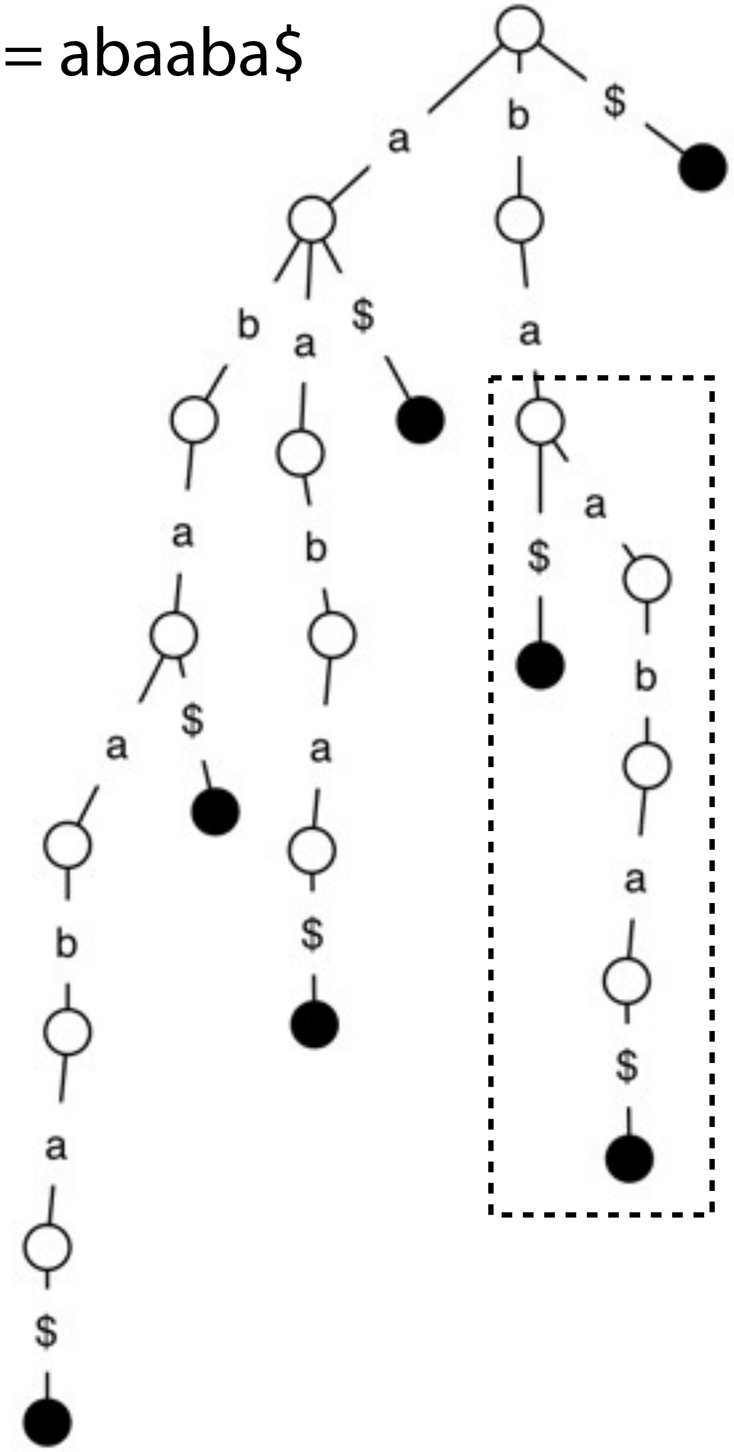
- for a node v in T, **depth**(v) or **node-depth**(v) is the distance from v to the root.
- **node-depth**(r) = 0
- **string**(v) = concatenation of all characters on the path $r \rightsquigarrow v$
- **string-depth**(v) = |**string**(v)| (note: **string-depth**(v) ≥ **node-depth**(v))
- for a string x, if ∃ node v with **string**(v) = x, we say **node**(x) = v
- T **displays** string x if ∃ node v and string y such that xy = **string**(v)
- **words**(T) = { x | T **displays** x}
- A **suffix tree** of string s is a compact Σ^+-tree such that **words**(T) = {s’ | s’ is a substring of s}

*

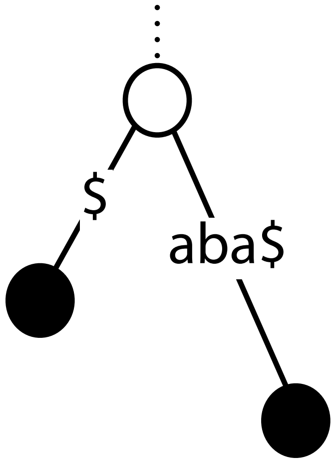
Defs. from: http://profs.sci.univr.it/~liptak/ALBioinfo/files/sequence_analysis.pdf

Suffix trie: making it smaller

T = abaaba$



Idea 1: Coalesce non-branching paths into a *single edge* with a *string* label

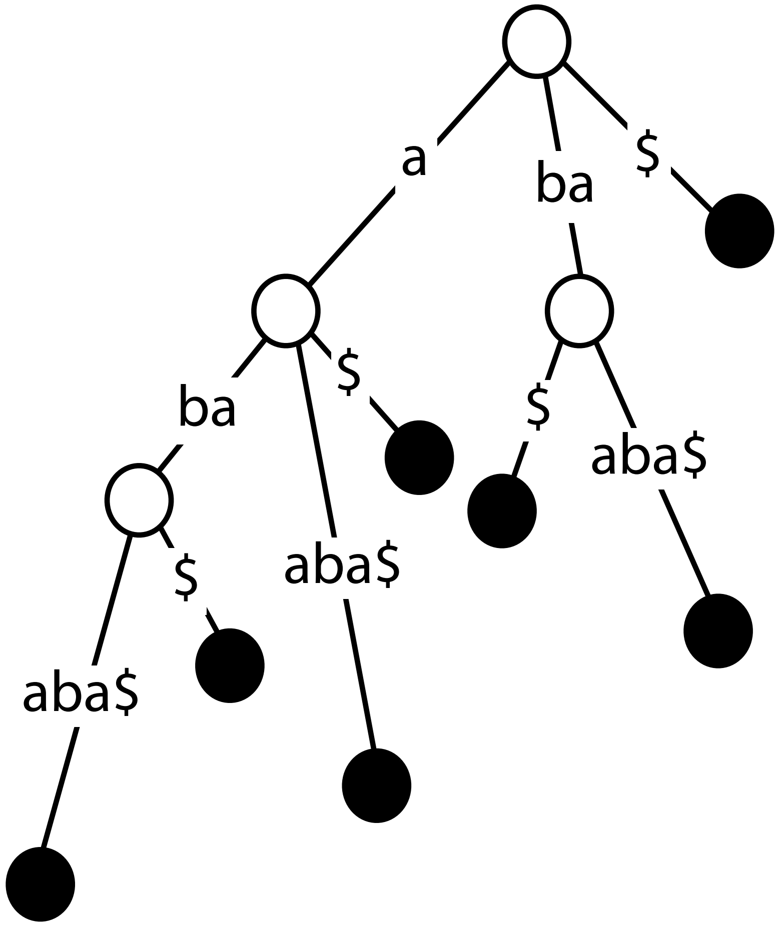


Reduces # nodes, edges,
guarantees internal nodes have >1 child

Suffix tree

L leaves, I internal nodes, E edges

T = abaaba$



$E = L + I - 1$

$E \geq 2I$ (each internal node branches)

$L + I - 1 \geq 2I \Rightarrow I \leq L - 1$

but

$L \leq m$ (at most m suffixes)

$I \leq m - 1$

$E = L + I - 1 \leq 2m - 2$

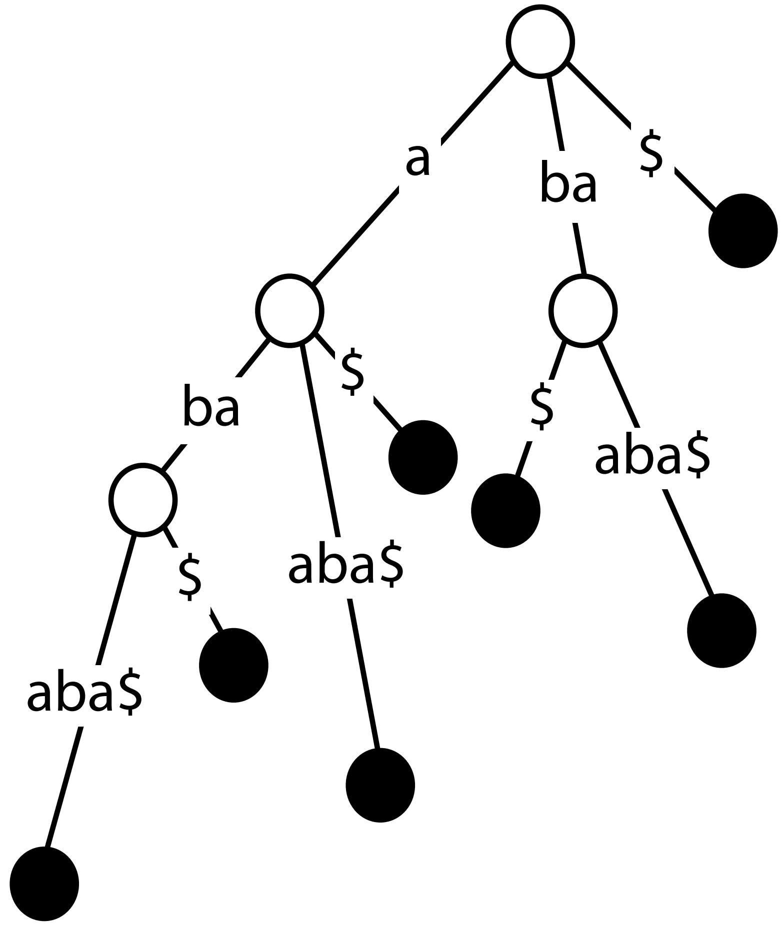
$E + L + I \leq 4m - 3 \in O(m)$

Is the total size O(m) now?

Suffix tree

L leaves, I internal nodes, E edges

T = abaaba$



$E = L + I - 1$

$E \geq 2I$ (each internal node branches)

$L + I - 1 \geq 2I \Rightarrow I \leq L - 1$

but

$L \leq m$ (at most m suffixes)

$I \leq m - 1$

$E = L + I - 1 \leq 2m - 2$

$E + L + I \leq 4m - 3 \in O(m)$

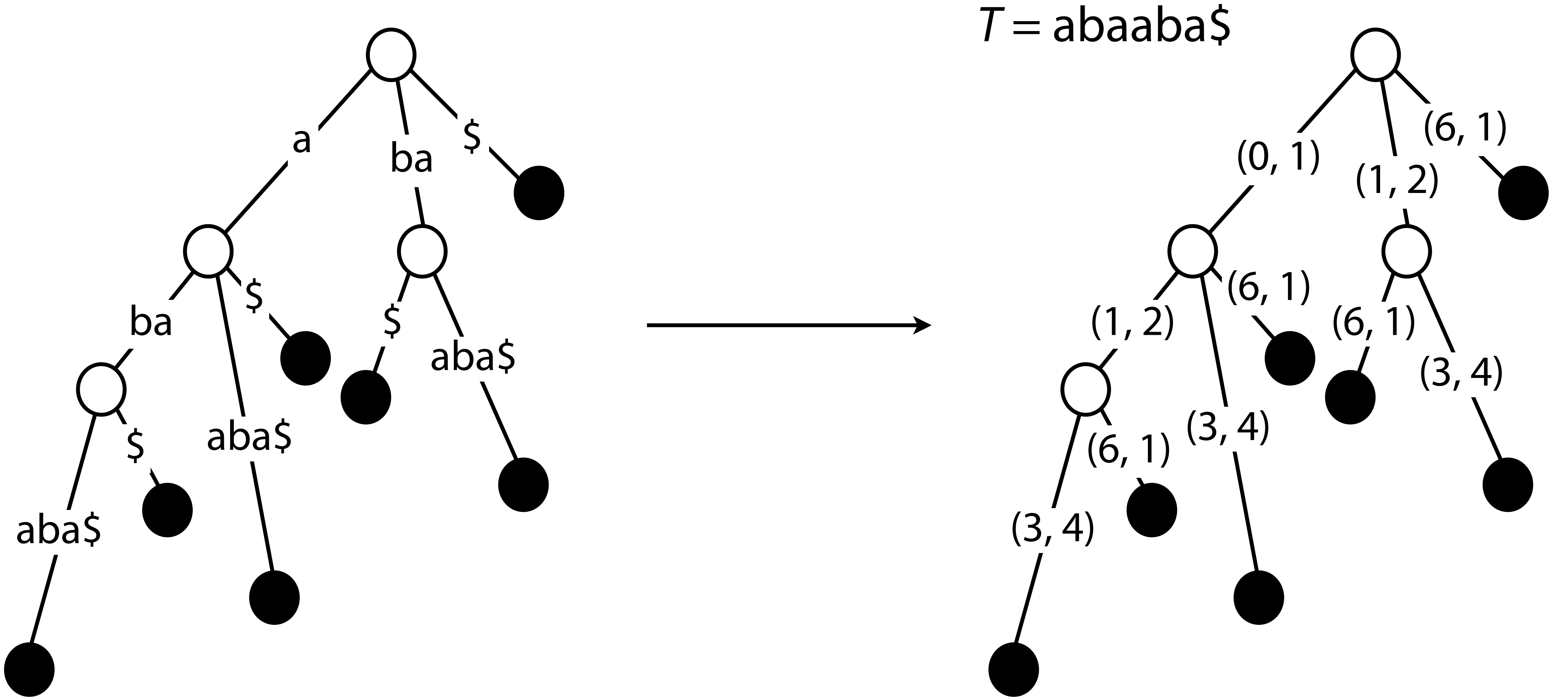
Is the total size O(m) now?

NO: The total length of edge labels is quadratic in m.

Suffix tree

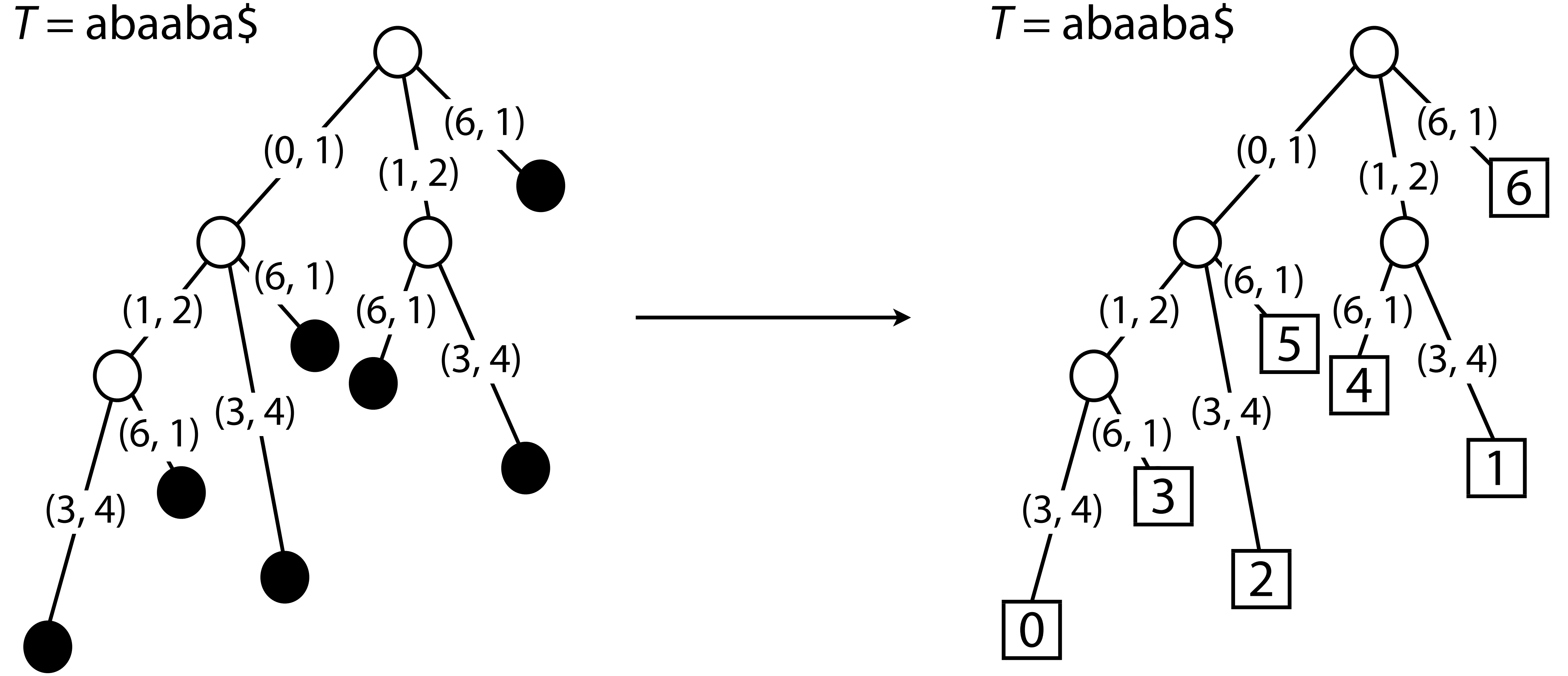
T = abaaba$

Idea 2: Store *T* itself in addition to the tree. Convert tree's edge labels to (offset, length) pairs with respect to *T*.

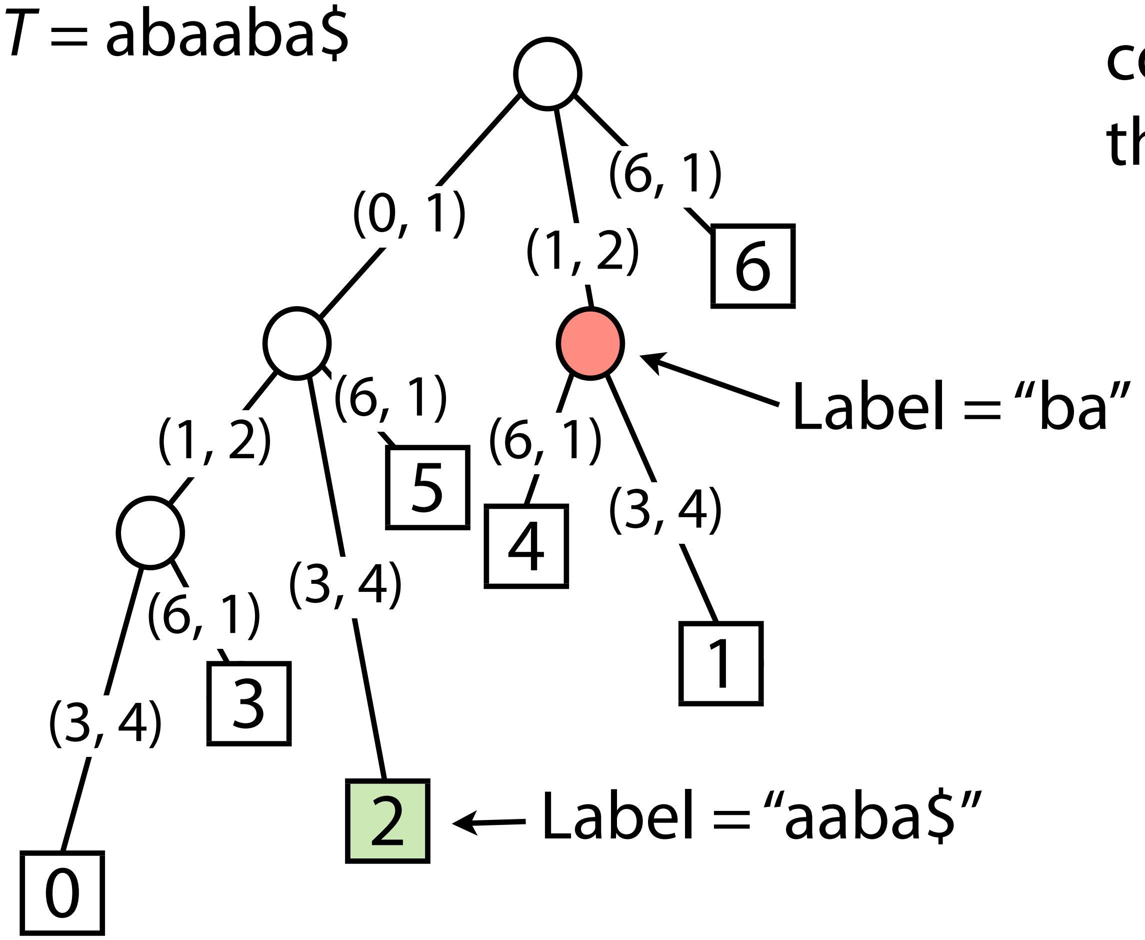


Space required for suffix tree is now *O*(*m*)

Suffix tree: leaves hold offsets where suffixes **begin**



Suffix tree: labels

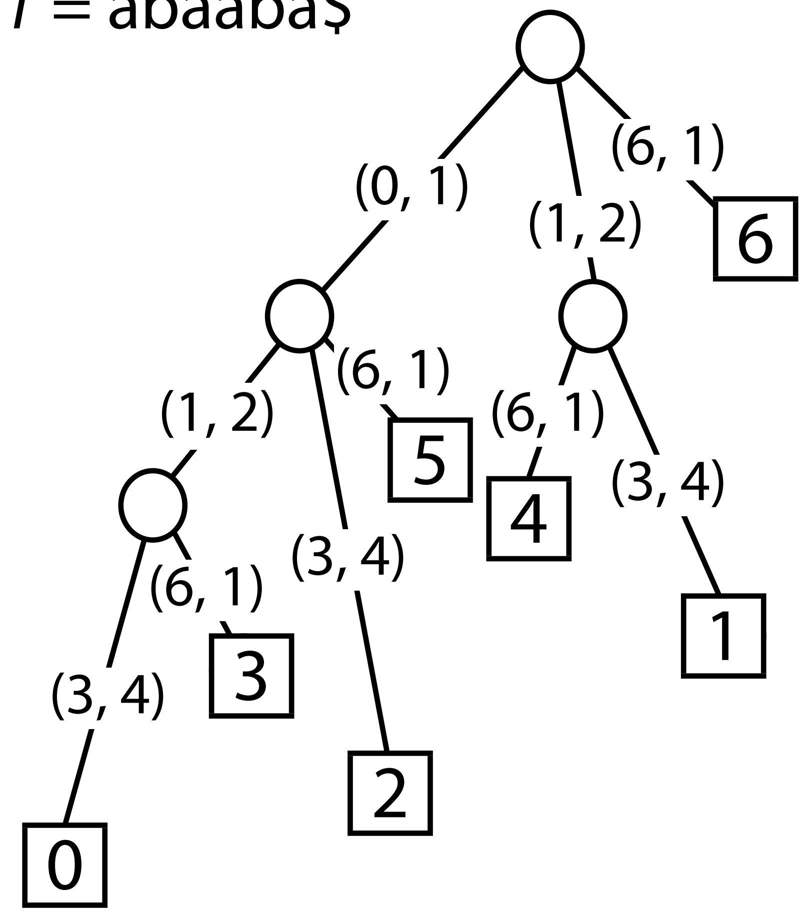

T = abaaba$

Label = “ba”

Label = “aaba$”

Again, each node’s *label* equals the concatenated edge labels from the root to the node. These aren’t stored explicitly.

Suffix tree: labels

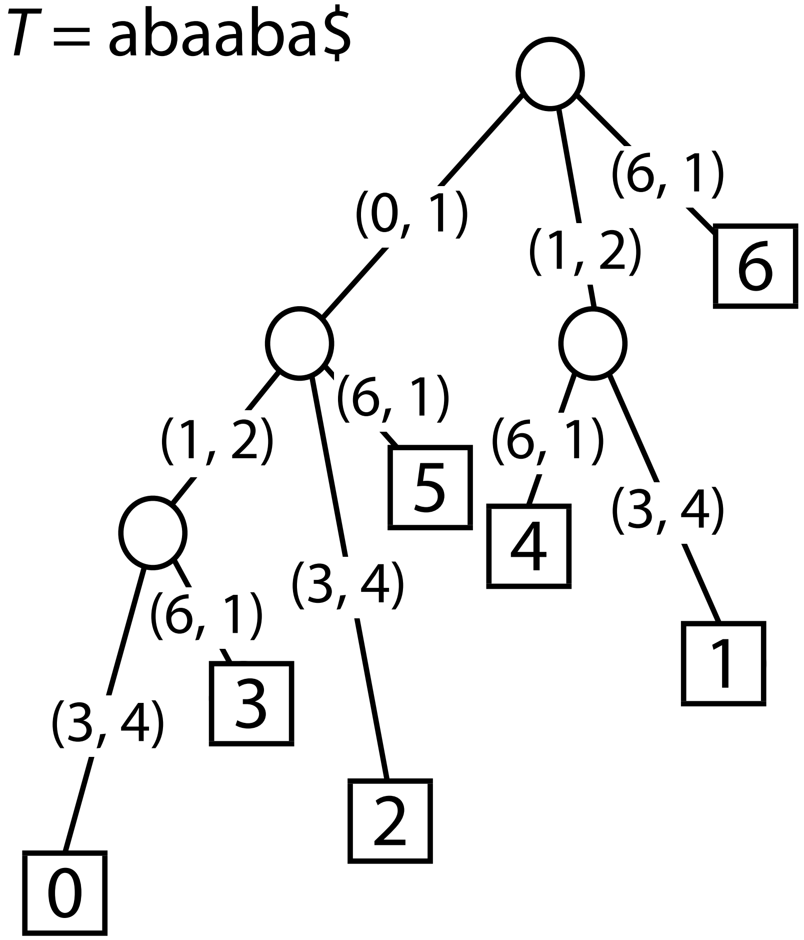


Because edges can have string labels, we must distinguish two notions of “depth”

- **Node** depth: how many edges we must follow from the root to reach the node

- **Label** depth: total length of edge labels for edges on path from root to node

Suffix tree: space caveat



Minor point:

We say the space taken by the edge labels is O(m), because we keep 2 integers per edge and there are O(m) edges

To store one such integer, we need enough bits to distinguish m positions in T, i.e. ceil($\log_2 m$) bits. We usually ignore this factor, since 64 bits is plenty for all practical purposes.

Similar argument for the pointers / references used to distinguish tree nodes.

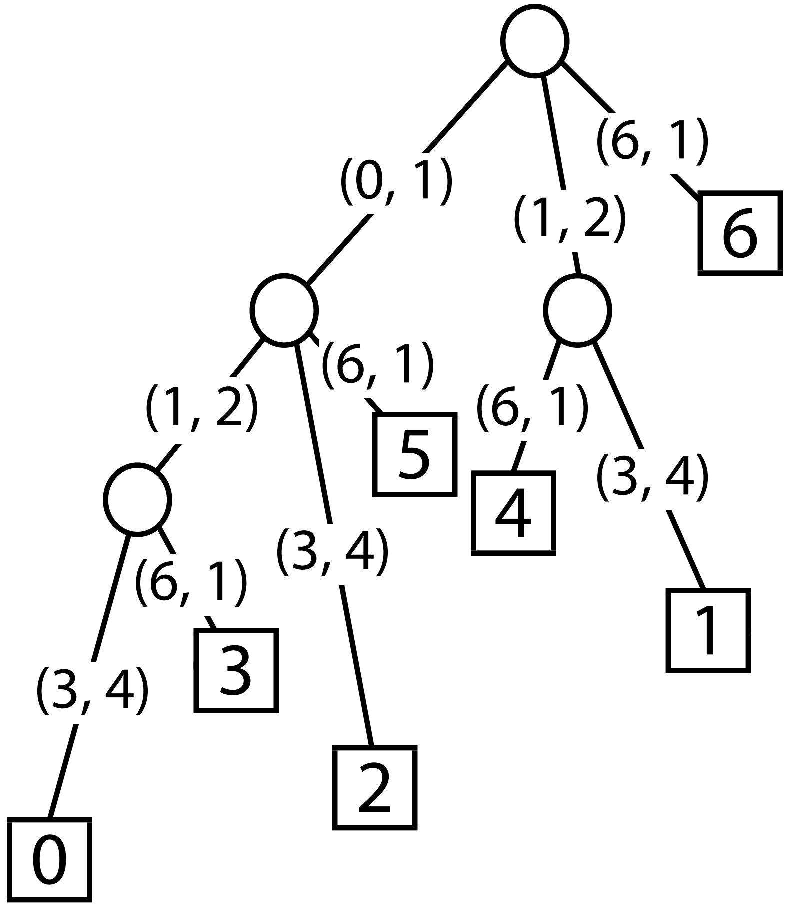
Suffix tree: building

Naive method 1: build a suffix trie, then coalesce non-branching paths and relabel edges

Naive method 2: build a single-edge tree representing only the longest suffix, then augment to include the 2nd-longest, then augment to include 3rd-longest, etc

Both are $O(m^2)$ time, but first uses $O(m^2)$ space while second uses $O(m)$

Naive method 2 is described in Gusfield 5.4



Python implementation at: http://nbviewer.ipython.org/6665861

WOTD (Write-Only Top-Down) Construction

Giegerich, Robert, and Stefan Kurtz. "A comparison of imperative and purely functional suffix tree constructions." Science of Computer Programming 25.2 (1995): 187-218.

Build a suffix tree for string s$

Recursive construction:

For every branching node **node**(u), subtree of **node**(u) is determined by all suffixes of s$ where u is a prefix.

Recursively construct subtree for all suffixes where u is a prefix.

Definition: *remaining suffixes* of u

R(node(u)) = { v | uv is a suffix of s$ }

WOTD (Write-Only Top-Down) Construction

Build a suffix tree for string s$

Recursive construction:

For every branching node **node**(u), subtree of **node**(u) is determined by all suffixes of s$ where u is a prefix.

Recursively construct subtree for all suffixes where u is a prefix.

Definition: *remaining suffixes* of u

$R(\text{node}(u)) = \{\ v \mid uv \text{ is a suffix of } s\$\ \}$

Definition: *c-group* of node(u)

$\text{group}(\text{node}(u), c) = \{\ w \in \Sigma^* \mid cw \in R(\text{node}(u))\}$

*

WOTD (Write-Only Top-Down) Construction

```
def WOTD(T : tree, node(u): node):
   for each c ∈ Σ ∪ {$}:
      G = group(node(u), c)
      ucv = lcp(G)
      if |G| == 1:
         add leaf node(ucv) as a child of node(u)
      else:
         add inner node(ucv) as a child of node(u)
         WOTD(T, node(ucv))
```

non-branching suffix

branching suffix

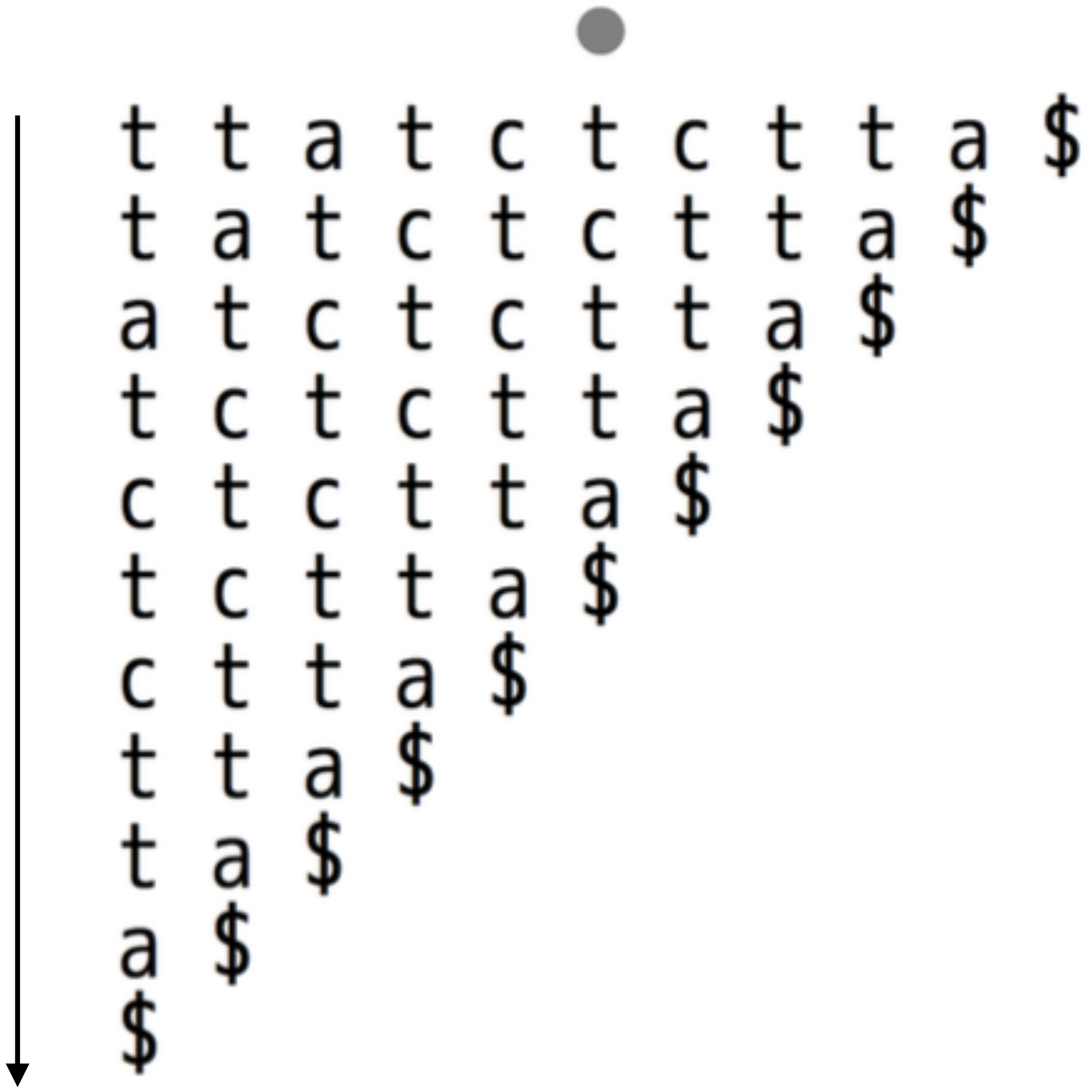
Start the algorithm by calling WOTD(T, node(ϵ))

root node

WOTD Example

s = ttatctctta$

suffixes are
read top-to-bottom



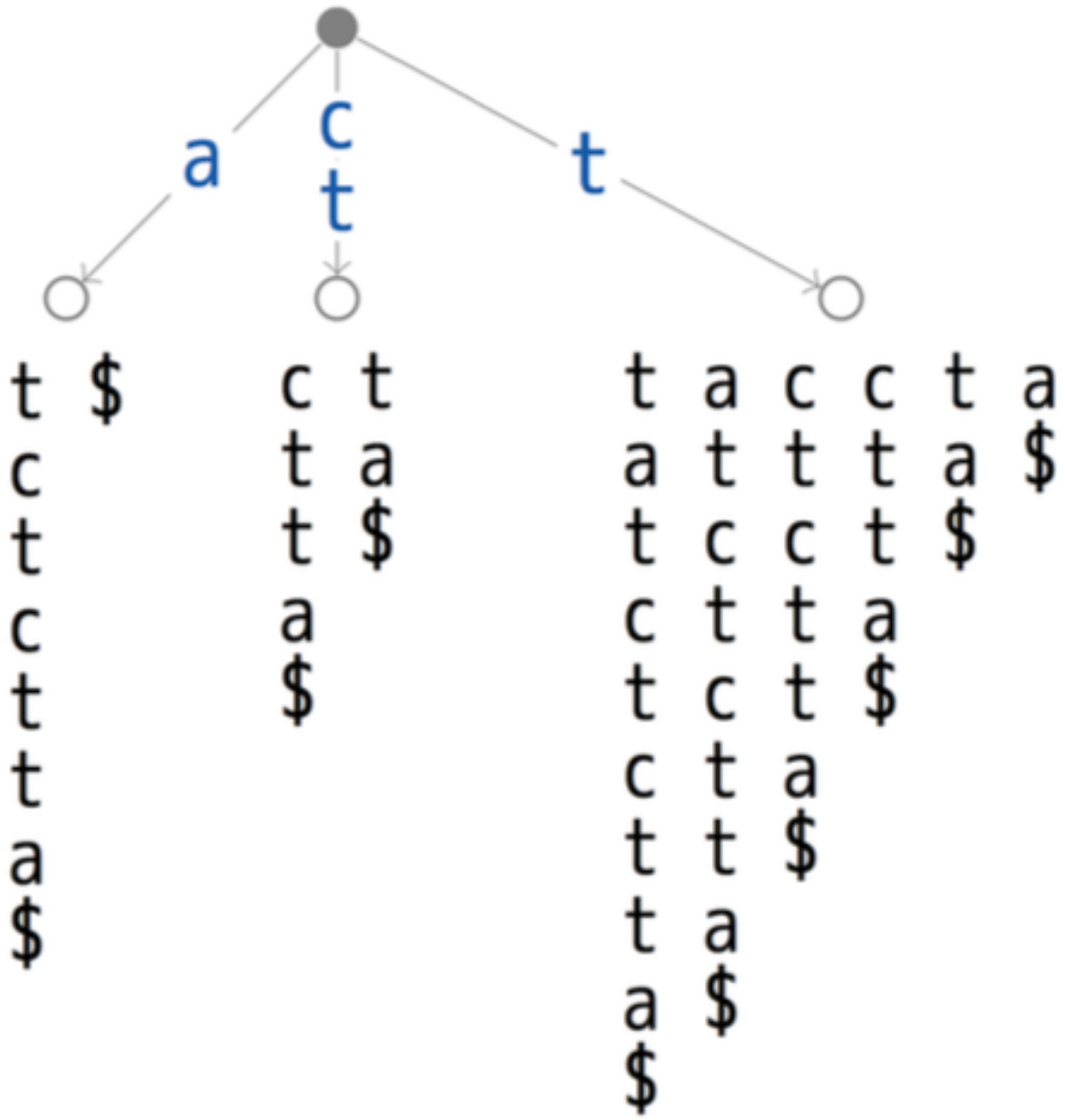
```
t t a t c t c t t a $
t a t c t c t t a $
a t c t c t t a $
t c t c t t a $
c t c t t a $
t c t t a $
c t t a $
t t a $
t a $
a $
$
```

example from: http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture4Materials/wotd.pdf

*

WOTD Example

s = ttatctctta

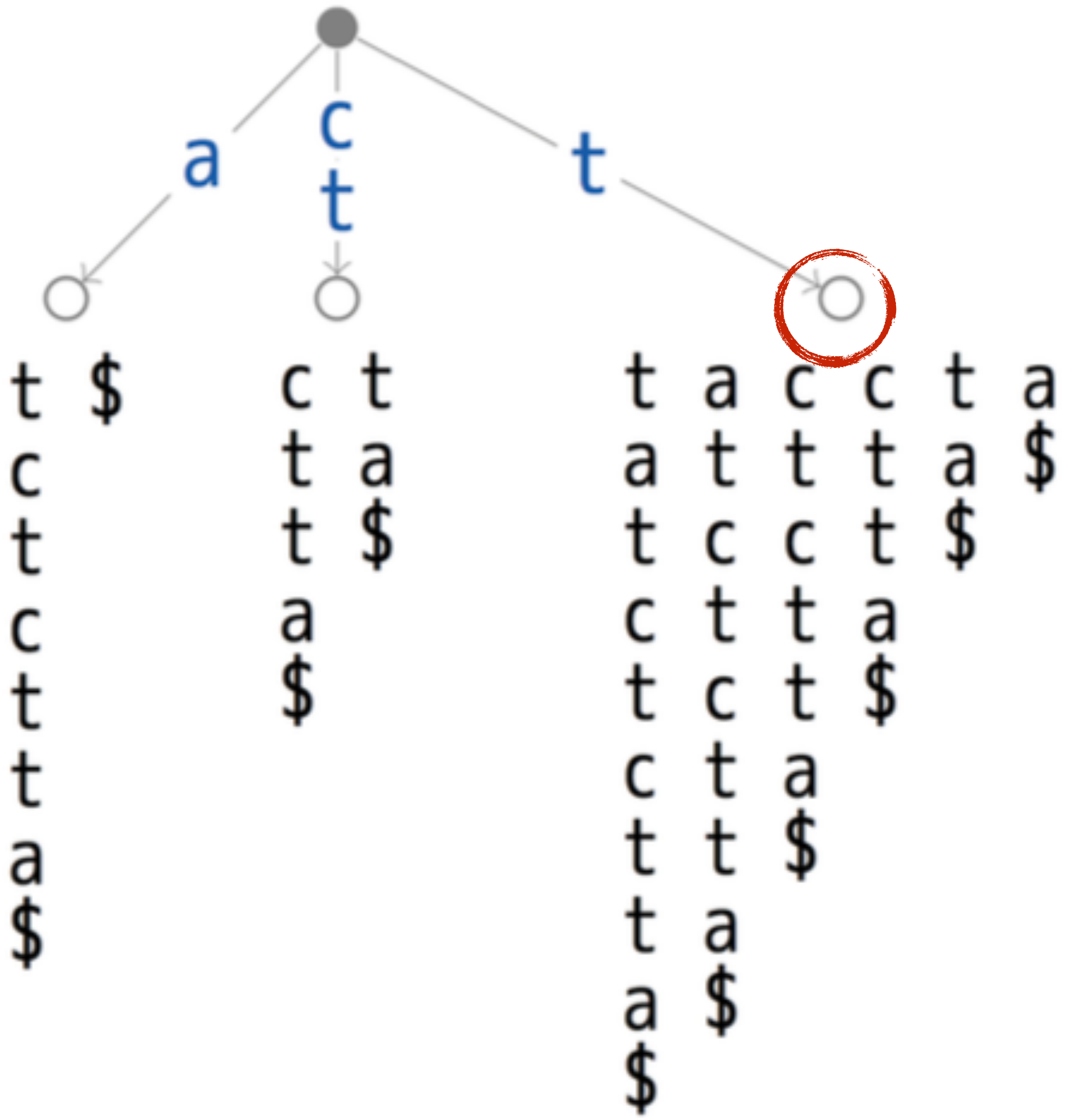


*

example from: http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture4Materials/wotd.pdf

WOTD Example

s = ttatctctta

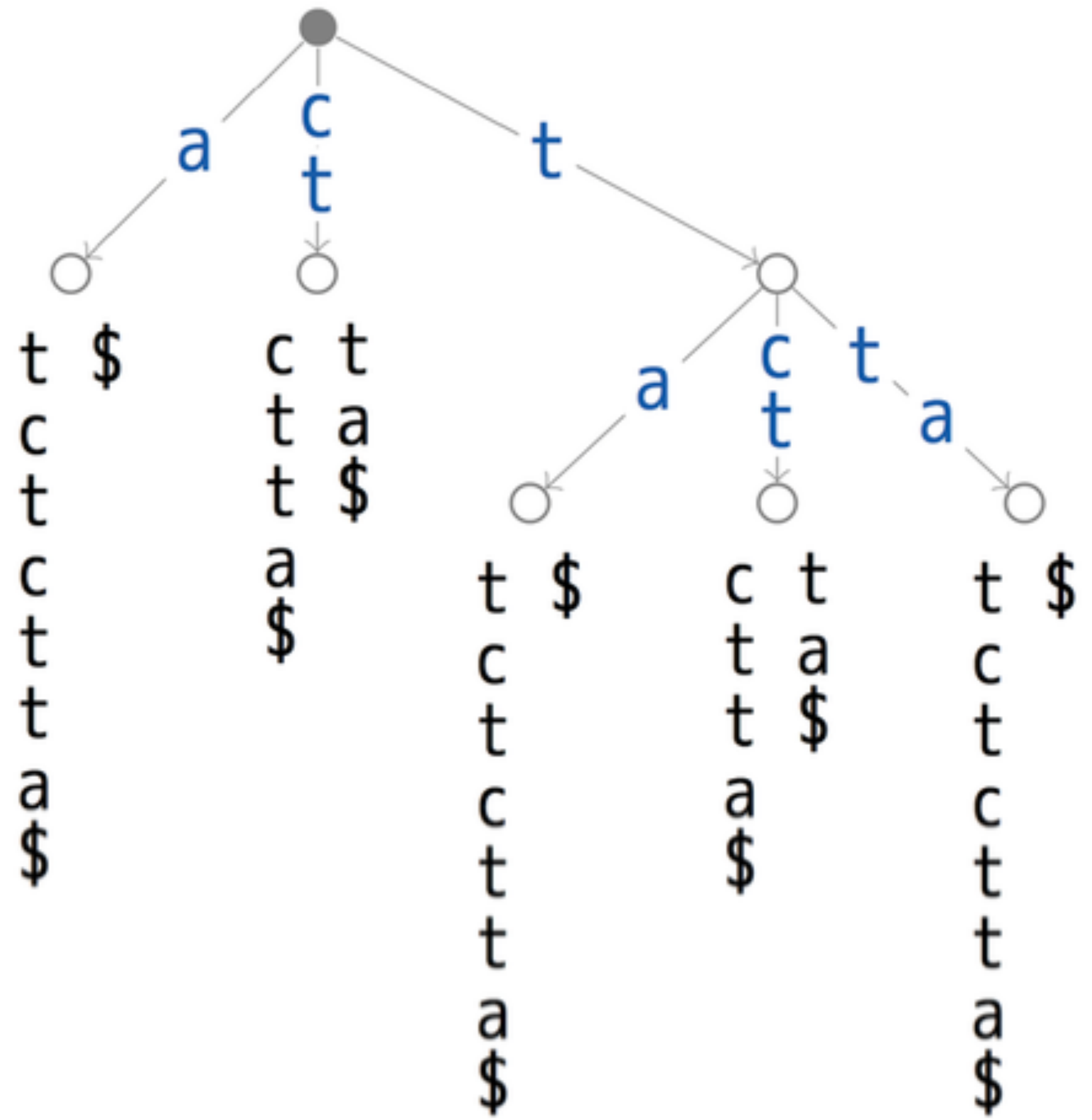


example from: http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture4Materials/wotd.pdf

*

WOTD Example

s = ttatctctta

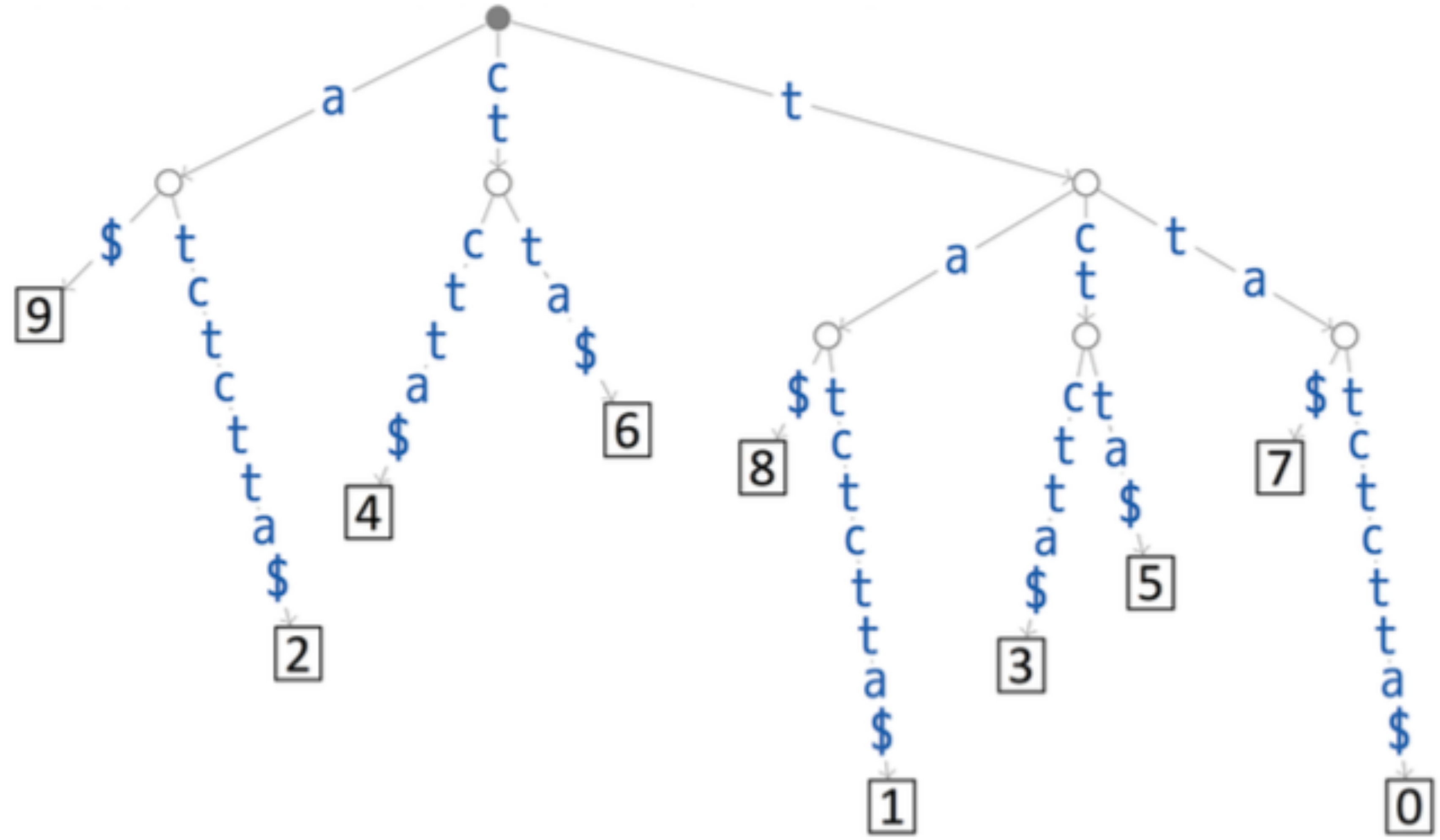


example from: http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture4Materials/wotd.pdf

*

WOTD Example

s = ttatctctta



example from: http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture4Materials/wotd.pdf

WOTD Properties

- Worst case time still ∊ $O(|T|^2)$
- Expected case time ∊ $O(|T| \log |T|)$
- Write-only property & recursive construction lends itself well to parallelism
- Good caching properties (locality of reference for substrings belonging to a subtree)
- Top-down construction order allows lazy construction as discussed in:

Giegerich, Robert, Stefan Kurtz, and Jens Stoye. "Efficient implementation of lazy suffix trees." Software: Practice and Experience 33.11 (2003): 1035-1049.

Suffix tree: building

Other methods for construction:

Ukkonen, Esko. "On-line construction of suffix trees."
Algorithmica 14.3 (1995): 249-260.

$O(m)$ time and space

Has *online* property: if T arrives one character at a time, algorithm efficiently updates suffix tree upon each arrival

We won’t cover it here; see Gusfield Ch. 6 for details

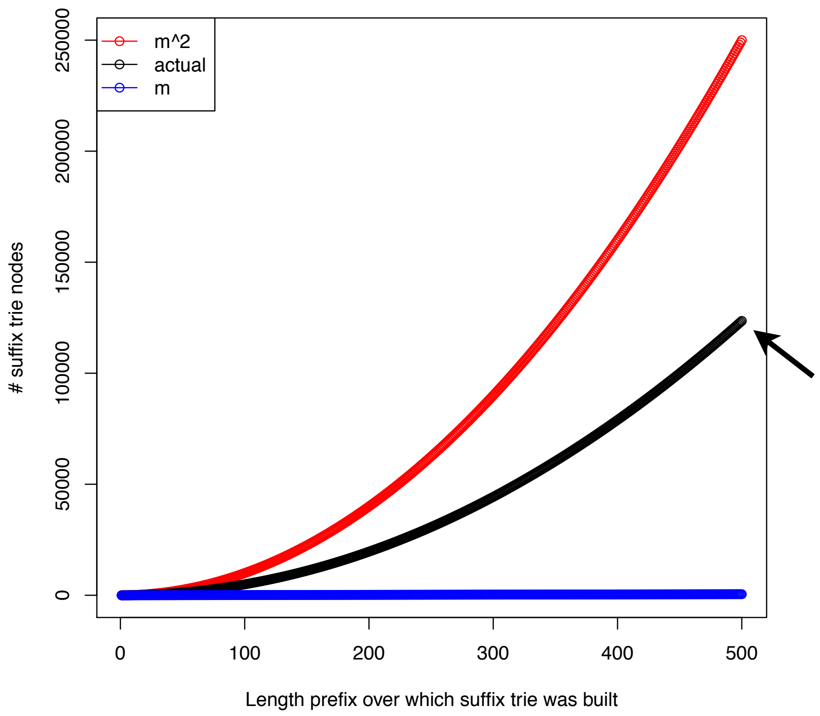
Or just Google “Ukkonen’s algorithm”

Suffix tree: actual growth

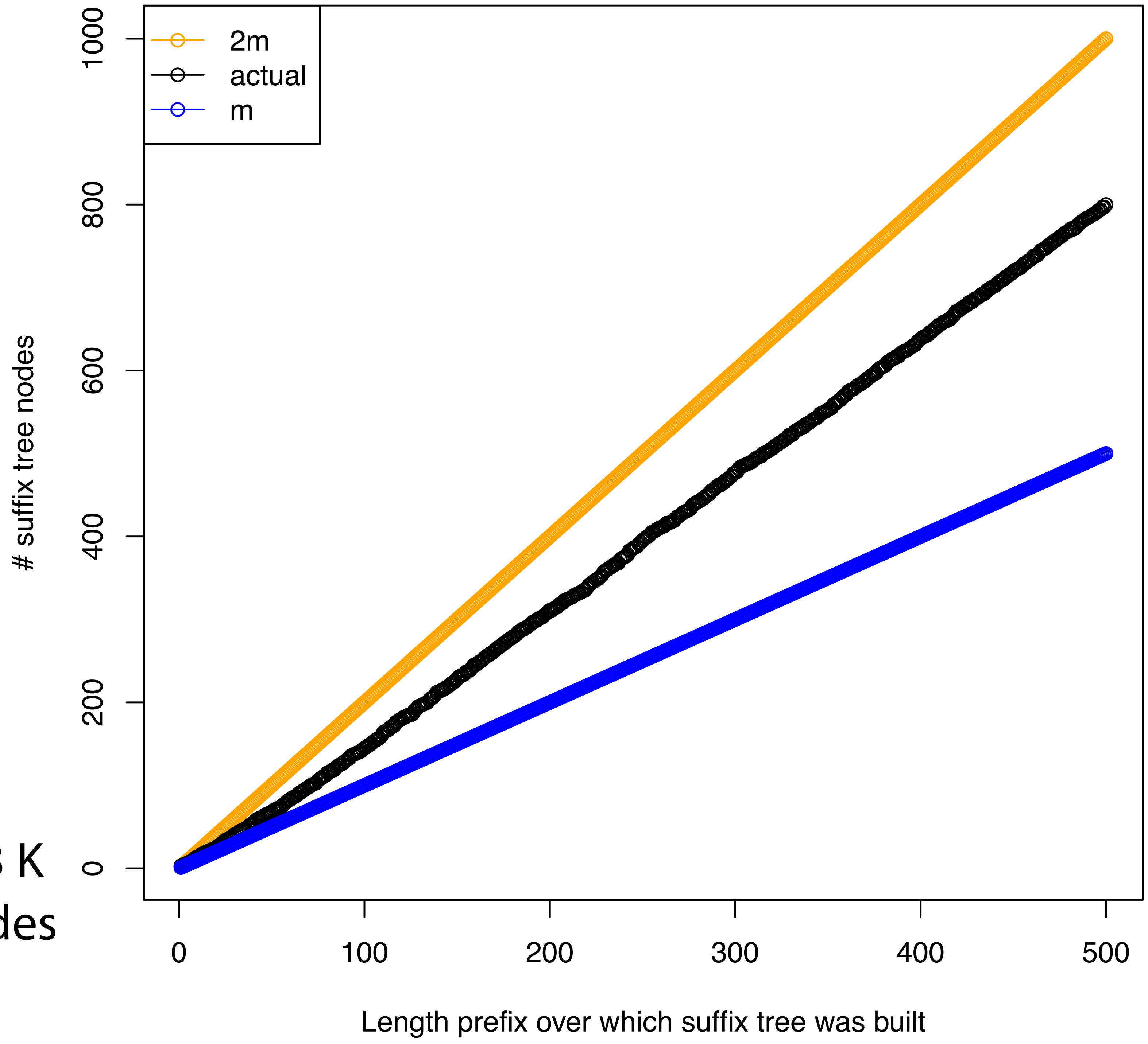
Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Compare with suffix trie:

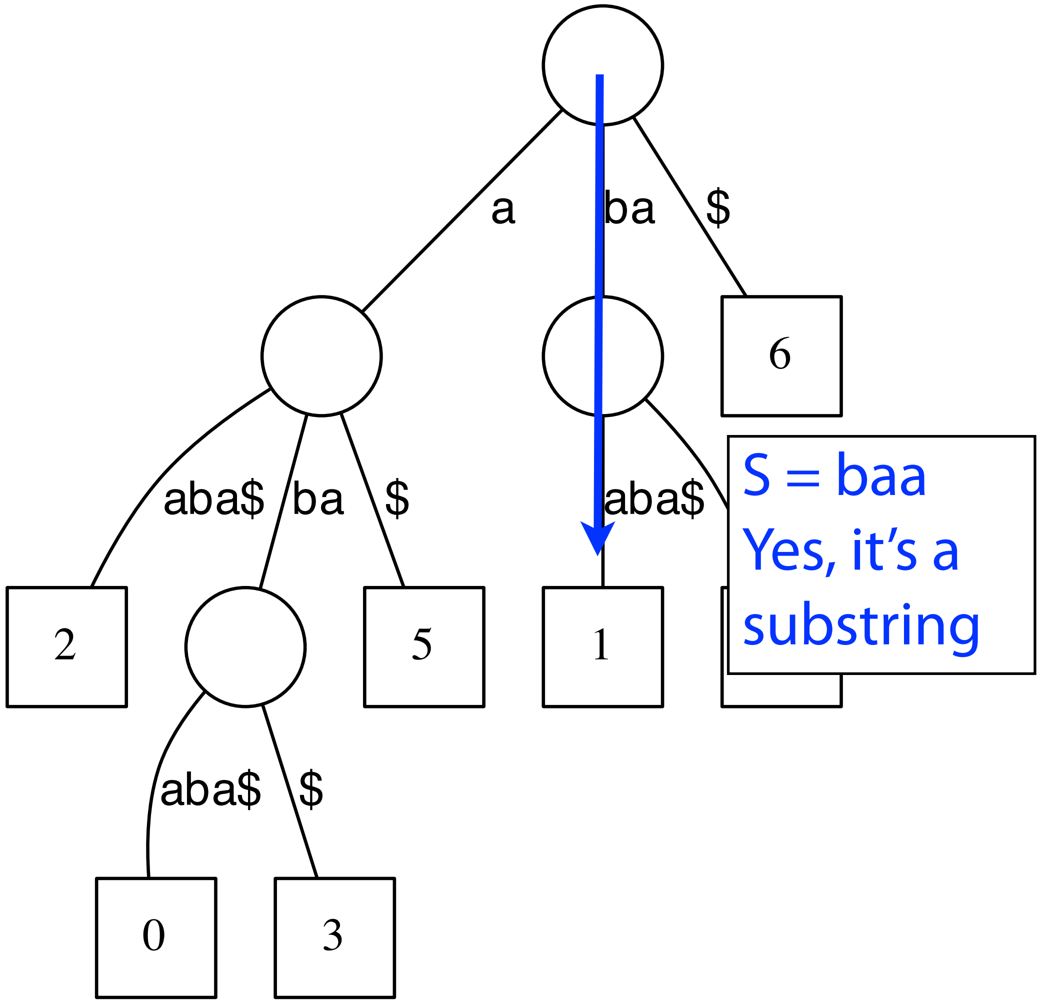


123 K nodes



Suffix tree

How do we check whether a string *S* is a substring of *T*?

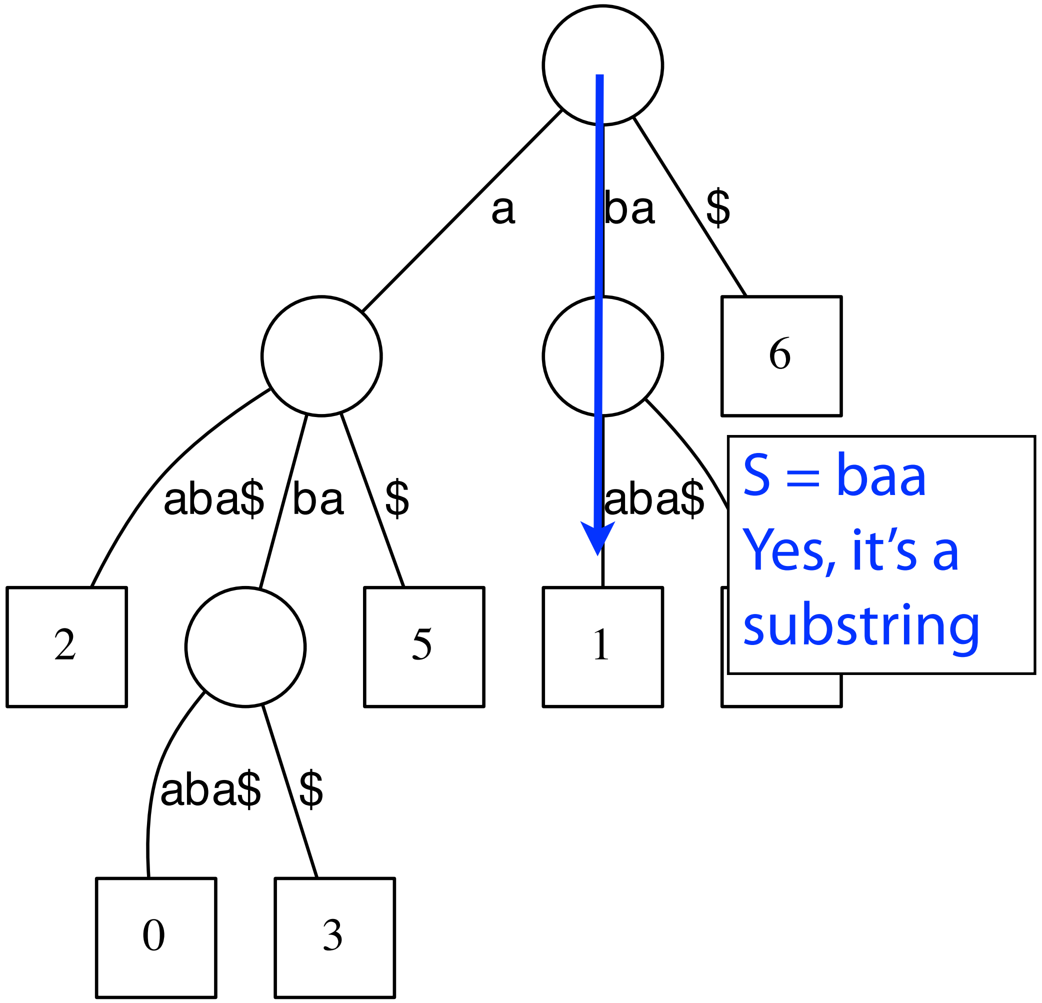


S = baa
Yes, it's a substring

Suffix tree

How do we check whether a string *S* is a substring of *T*?

Essentially same procedure as for suffix trie, except we have to deal with coalesced edges

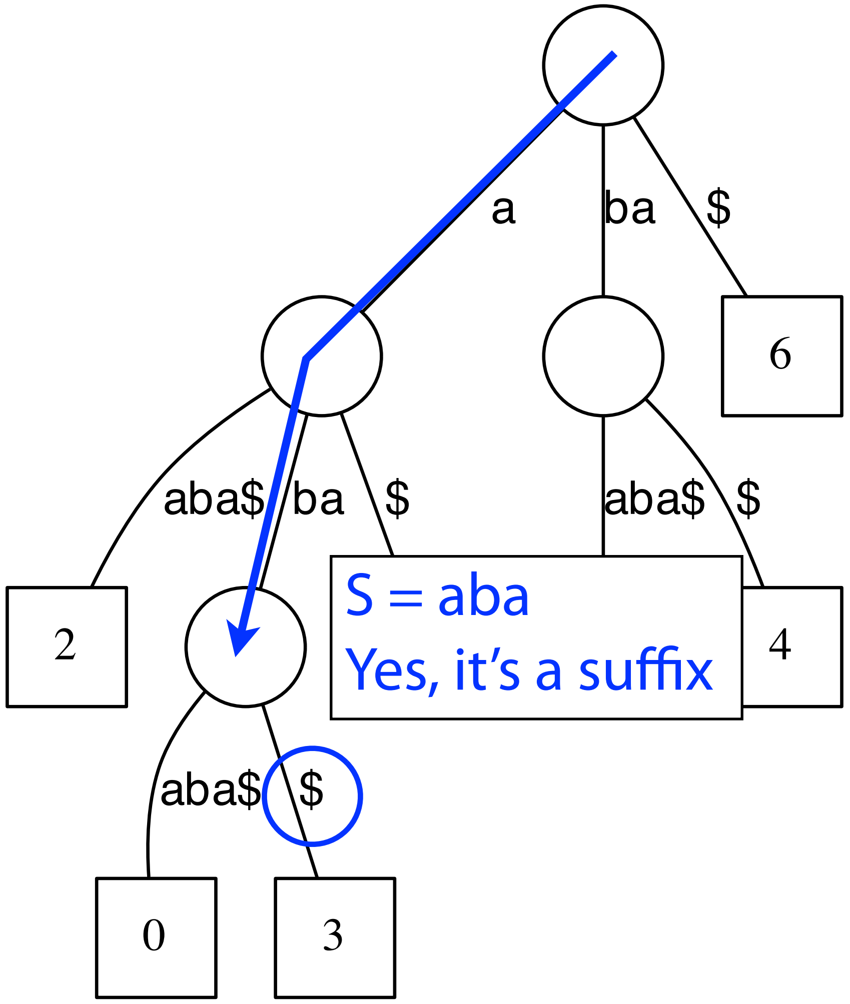


S = baa
Yes, it's a substring

Suffix tree

How do we check whether a string *S* is a suffix of *T*?

Essentially same procedure as for suffix trie, except we have to deal with coalesced edges

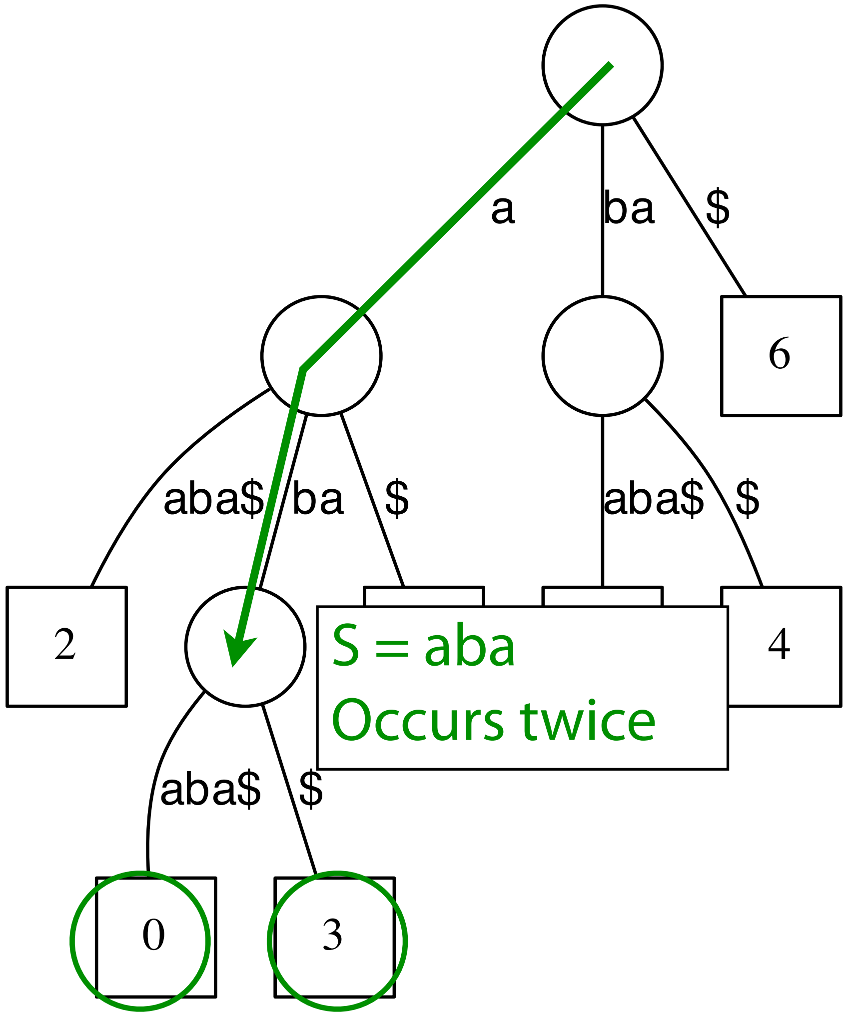


S = aba
Yes, it's a suffix

Suffix tree

How do we count the **number of times** a string *S* occurs as a substring of *T*?

Same procedure as for suffix trie



S = aba
Occurs twice

Suffix tree: applications

With suffix tree of *T*, we can find all matches of *P* to *T*. Let *k* = # matches.

E.g., *P* = ab, *T* = abaaba$

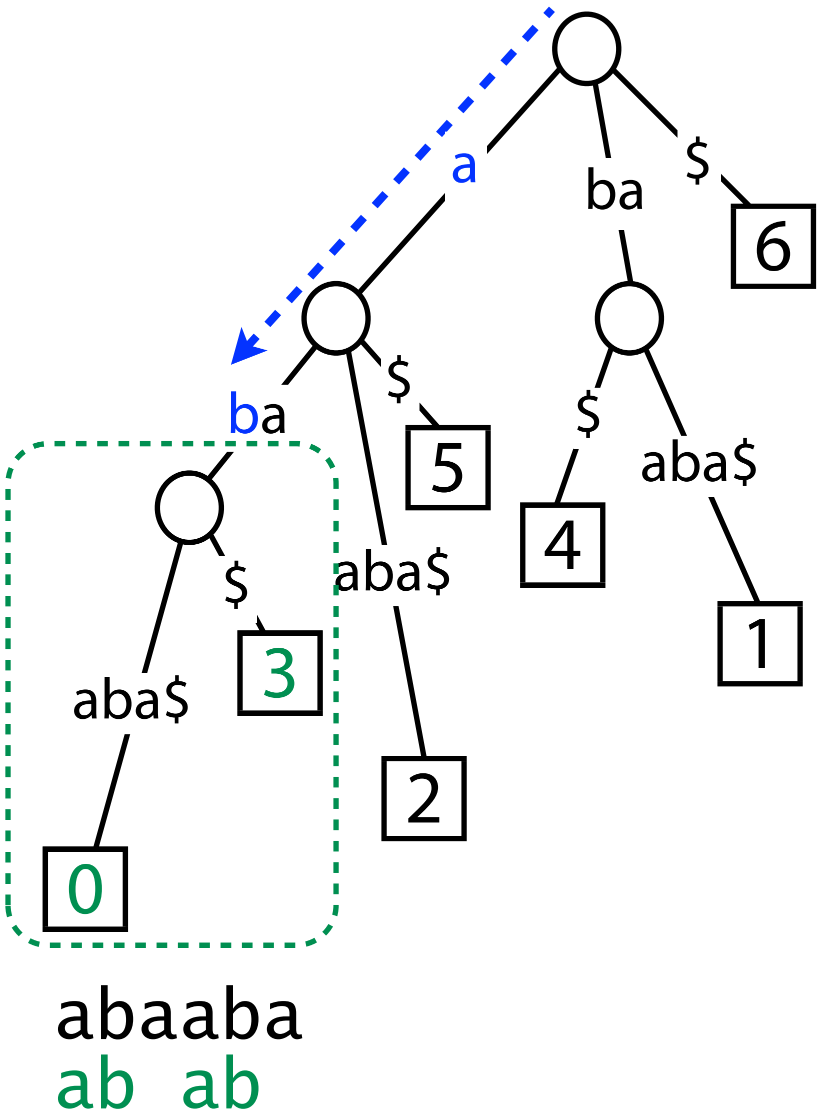
O(*n*) Step 1: walk down ab path

If we "fall off" there are no matches

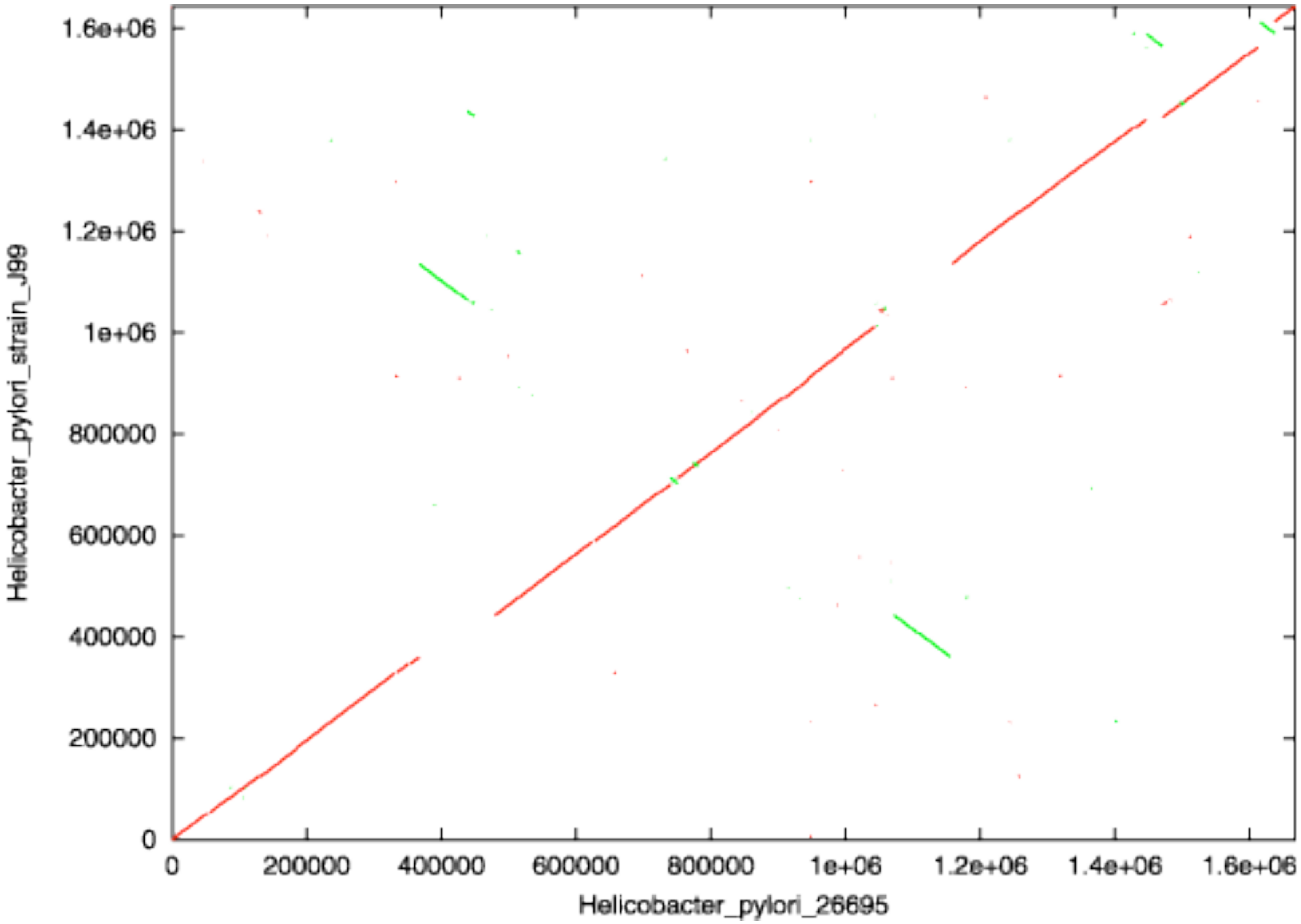
O(*k*) Step 2: visit all leaf nodes below

Report each leaf offset as match offset

O(*n* + *k*) time



Suffix tree application: find long common substrings



Dots are *maximal unique matches* (*MUMs*), a kind of long substring shared by two sequences

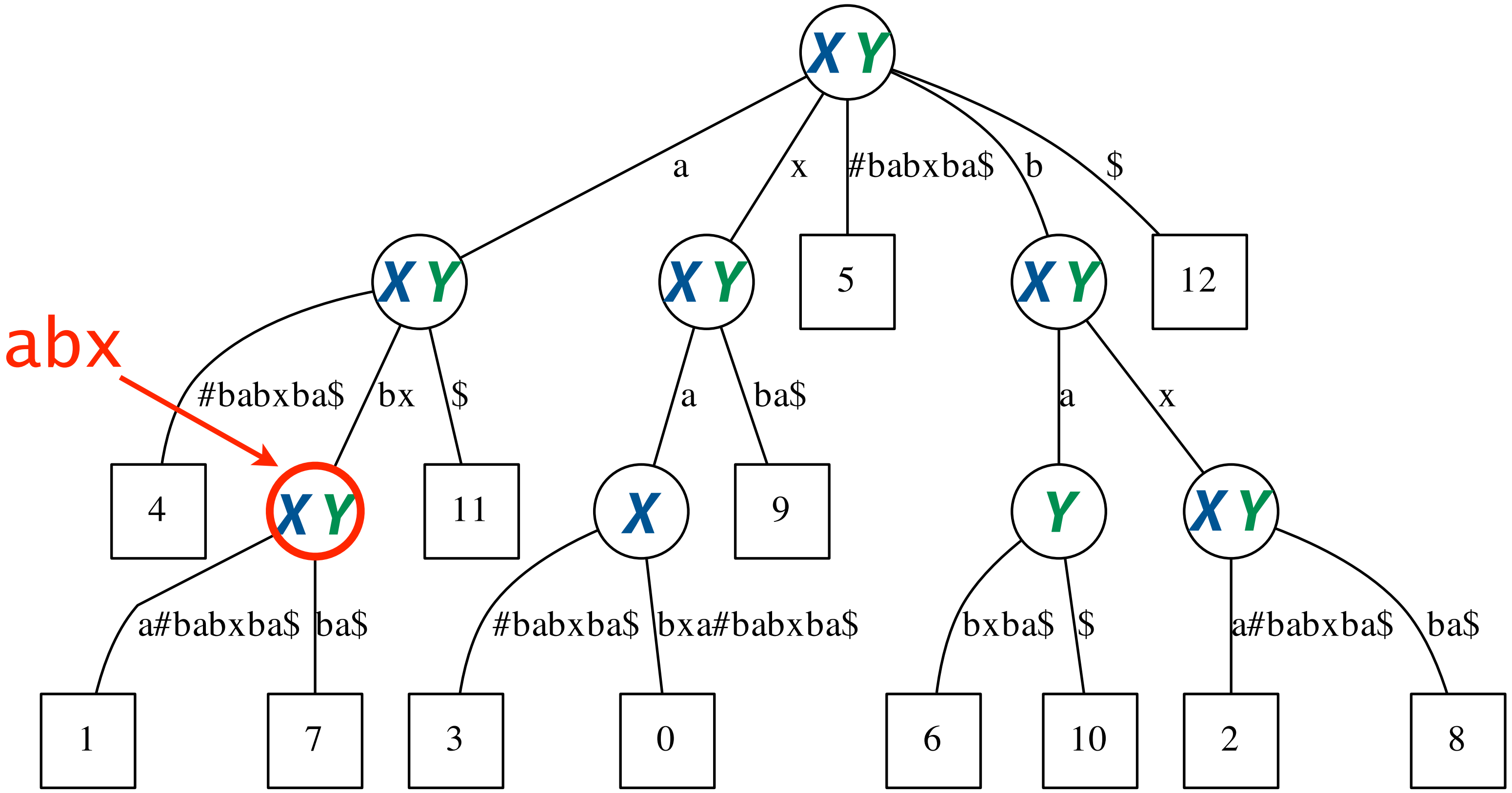
Red = match was between like strands, green = different strands

Axes show different strains of Helicobacter pylori, a bacterium found in the stomach and associated with gastric ulcers

Suffix tree application: find longest common substring

To find the longest common substring (LCS) of *X* and *Y*, make a new string $X\#Y\$$ where $\#$ and $\$$ are both terminal symbols. Build a suffix tree for $X\#Y\$$.

X = xabxa
Y = babxba
X#***Y***$ = xabxa#babxba$



Consider leaves:
offsets in [0, 4] are suffixes of ***X***, offsets in [6, 11] are suffixes of ***Y***

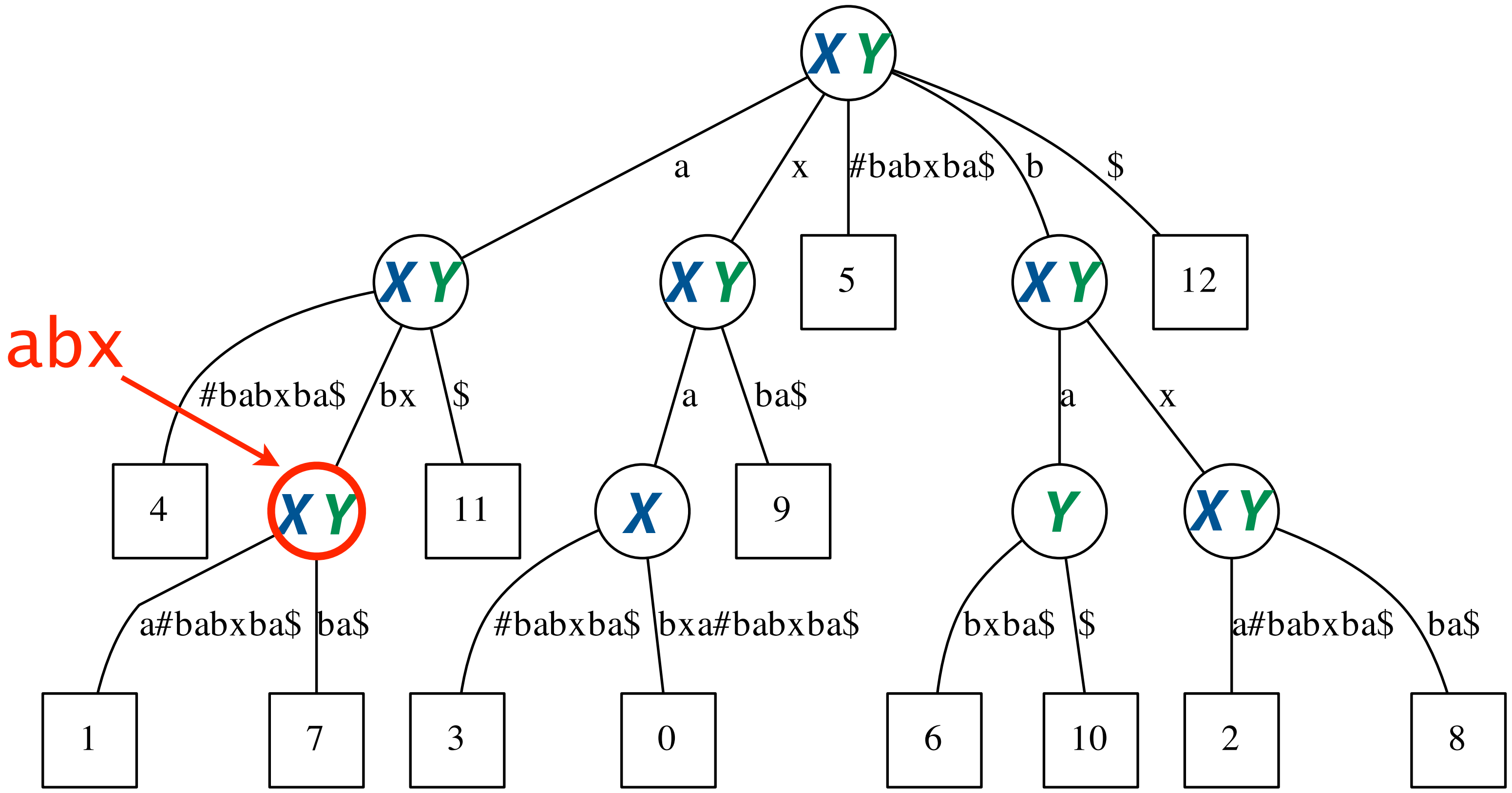
Traverse the tree and annotate each node according to whether leaves below it include suffixes of ***X***, ***Y*** or both

The deepest node annotated with both ***X*** and ***Y*** has LCS as its label. $O(|X| + |Y|)$ time and space.

Suffix tree application: generalized suffix trees

This is one example of many applications where it is useful to build a suffix tree over many strings at once

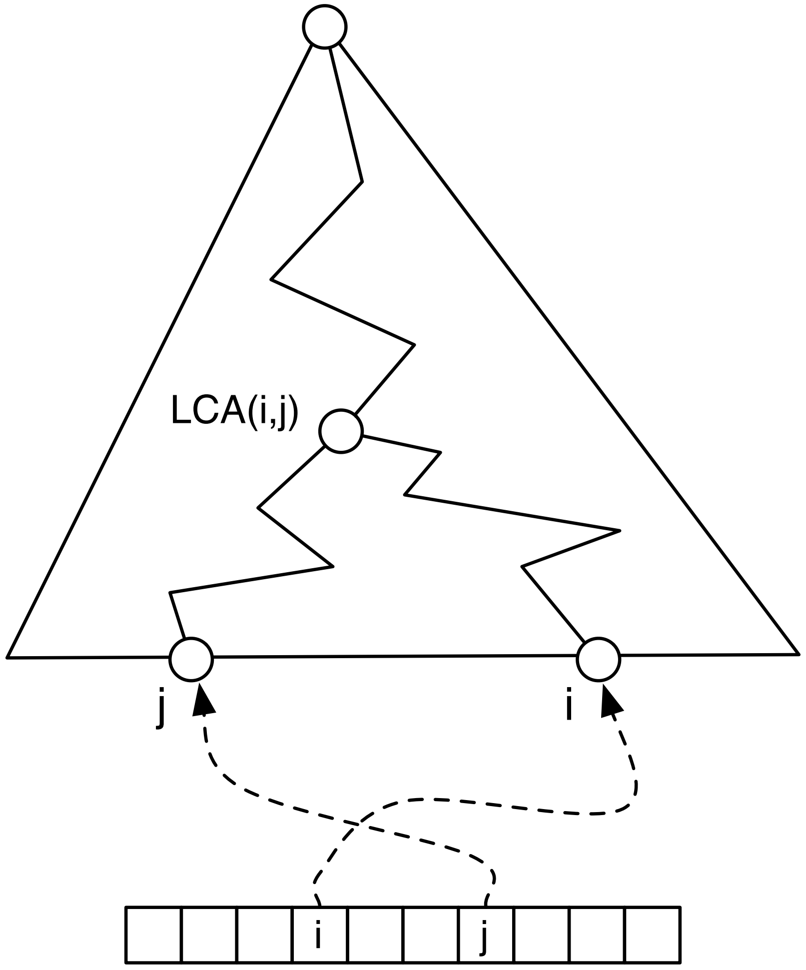
Such a tree is called a *generalized suffix tree.* These are introduced in *Gusfield* 6.4.



Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.

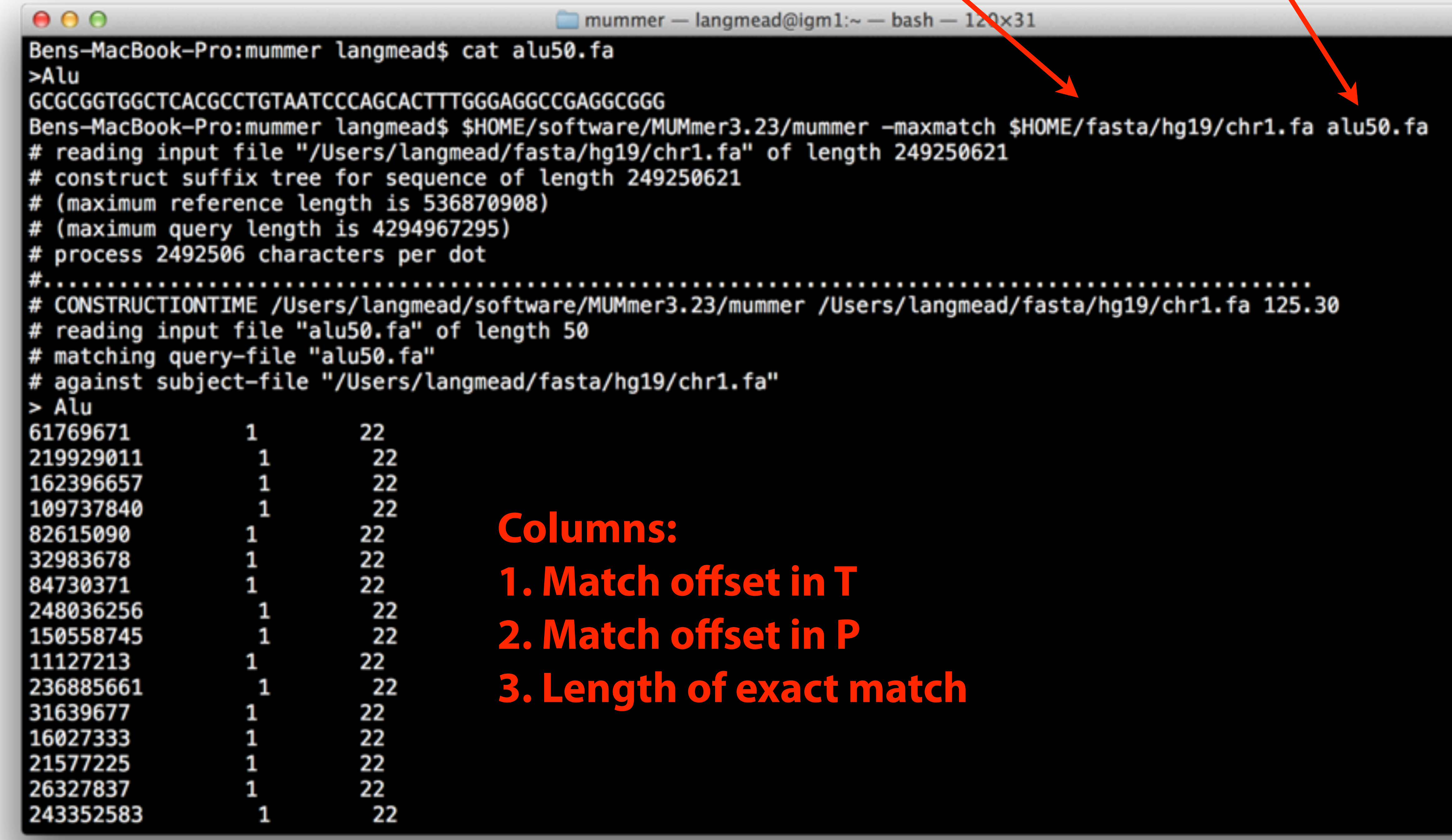


Build generalized suffix tree for S and T.	O(|S| + |T|)
Preprocess tree so that lowest common ancestors (LCA) can be found in constant time. This can be done using range-minimum queries (RMQ)	O(|S| + |T|)
Create an array mapping suffix numbers to leaf nodes.	O(|S| + |T|)
Given query (i,j):	
Find the leaf nodes for i and j	O(1)
Return string of LCA for i and j	O(LCE(i,j))

Suffix trees in the real world: MUMmer

FASTA file containing "reference" ("text")

FASTA file containing ALU string



```
Bens-MacBook-Pro:mummer langmead$ cat alu50.fa
>Alu
GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG
Bens-MacBook-Pro:mummer langmead$ $HOME/software/MUMmer3.23/mummer -maxmatch $HOME/fasta/hg19/chr1.fa alu50.fa
# reading input file "/Users/langmead/fasta/hg19/chr1.fa" of length 249250621
# construct suffix tree for sequence of length 249250621
# (maximum reference length is 536870908)
# (maximum query length is 4294967295)
# process 2492506 characters per dot
#....................................................................................................
# CONSTRUCTIONTIME /Users/langmead/software/MUMmer3.23/mummer /Users/langmead/fasta/hg19/chr1.fa 125.30
# reading input file "alu50.fa" of length 50
# matching query-file "alu50.fa"
# against subject-file "/Users/langmead/fasta/hg19/chr1.fa"
> Alu
61769671         1        22
219929011         1        22
162396657         1        22
109737840         1        22
82615090         1        22
32983678         1        22
84730371         1        22
248036256         1        22
150558745         1        22
11127213         1        22
236885661         1        22
31639677         1        22
16027333         1        22
21577225         1        22
26327837         1        22
243352583         1        22
```

Indexing phase: ~2 minutes

Matching phase: very fast

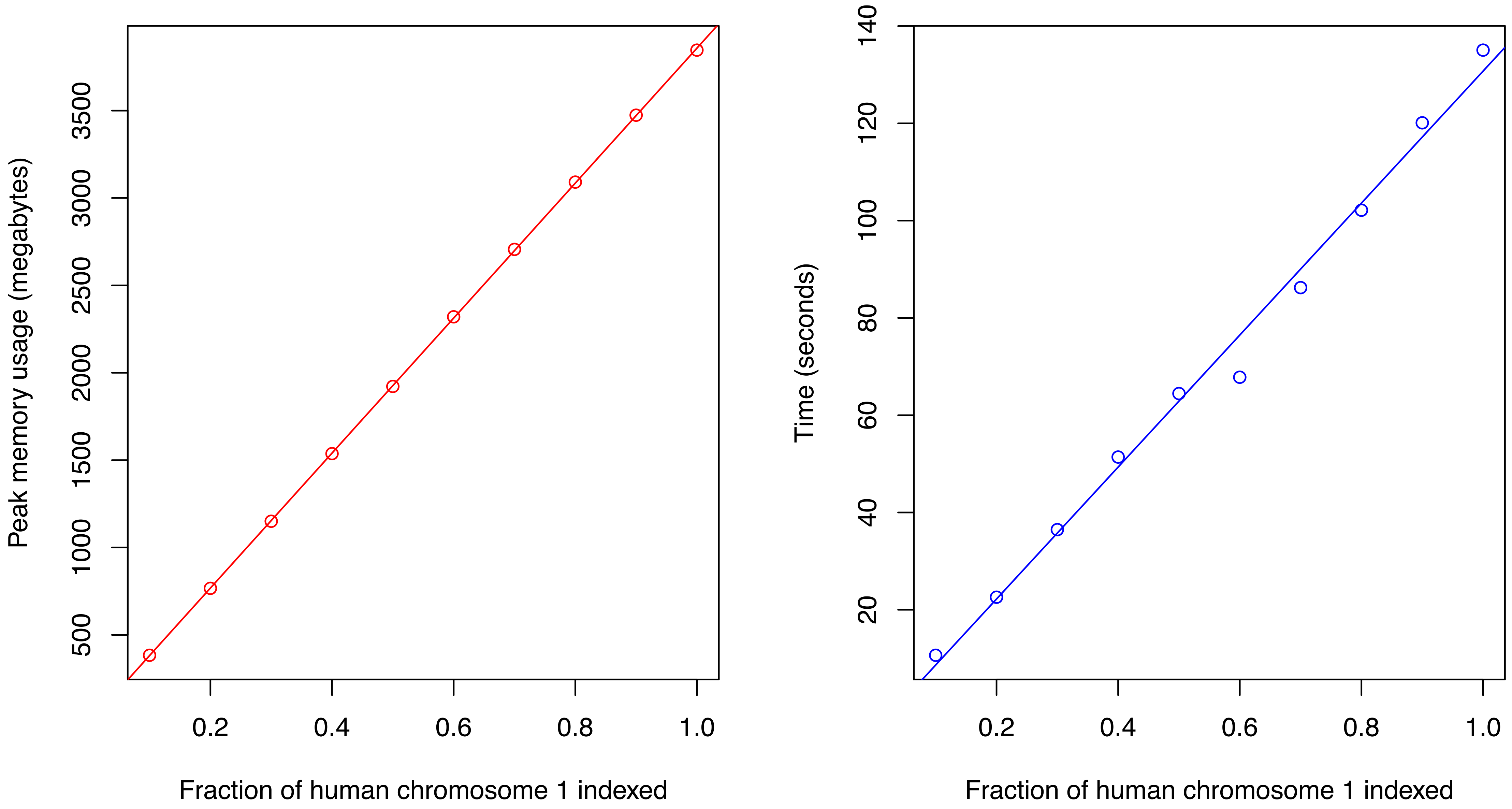
⋮

Columns:

1. **Match offset in T**
2. **Match offset in P**
3. **Length of exact match**

Suffix trees in the real world: MUMmer

MUMmer v3.32 time and memory scaling when indexing increasingly larger fractions of human chromosome 1



For whole chromosome 1, took 2m:14s and used 3.94 GB memory

Suffix trees in the real world: MUMmer

Attempt to build index for whole human genome reference:

```
mummer: suffix tree construction failed: textlen=3101804822
larger than maximal textlen=536870908
```

We can predict it would have taken about 47 GB of memory

Suffix trees in the real world: the constant factor

While *O*(*m*) is desirable, the constant in front of the *m* limits wider use of suffix trees in practice

Constant factor varies depending on implementation:

Estimate of MUMmer's constant factor = 3.94 GB / 250 million nt ≈ **15.75 bytes per node**

Literature reports implementations achieving as little as 8.5 bytes per node, but no implementation used in practice that I know of is better than ≈ **12.5 bytes per node**

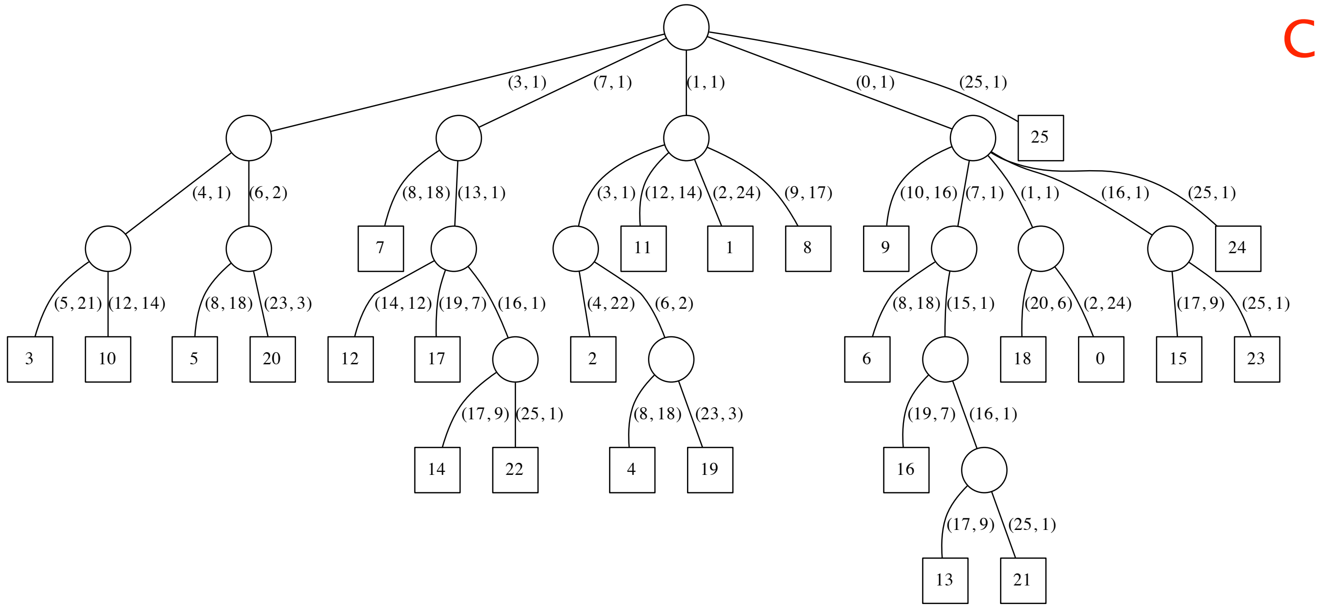
Kurtz, Stefan. "Reducing the space requirement of suffix trees." *Software Practice and Experience* 29.13 (1999): 1149-1171.

Suffix tree: summary

Organizes all suffixes into an incredibly useful, flexible data structure, in $O(m)$ time and space

A naive method (e.g. suffix trie) could easily be quadratic or worse

Used in practice for whole genome alignment, repeat identification, etc



Actual memory footprint (bytes per node) is quite high, limiting usefulness

GTTATAGCTGATCGCGGCGTAGCGG$ *m* chars

GTTATAGCTGATCGCGGCGTAGCGG$
TTATAGCTGATCGCGGCGTAGCGG$
TATAGCTGATCGCGGCGTAGCGG$
ATAGCTGATCGCGGCGTAGCGG$
TAGCTGATCGCGGCGTAGCGG$
AGCTGATCGCGGCGTAGCGG$
GCTGATCGCGGCGTAGCGG$
CTGATCGCGGCGTAGCGG$
TGATCGCGGCGTAGCGG$
GATCGCGGCGTAGCGG$
ATCGCGGCGTAGCGG$
TCGCGGCGTAGCGG$
CGCGGCGTAGCGG$
GCGGCGTAGCGG$
CGGCGTAGCGG$
GGCGTAGCGG$
GCGTAGCGG$
CGTAGCGG$
GTAGCGG$
TAGCGG$
AGCGG$
GCGG$
CGG$
GG$
G$
$

m(m+1)/2 chars