Suffix-based indexing

Dr. Jared Simpson
Ontario Institute for Cancer Research
&
Department of Computer Science
University of Toronto

Introduction to suffix indexing structures

- So far we've discussed substring indices
- Today we'll cover data structures that index the suffixes of a string: suffix tries, suffix tree, suffix array

Tries

A trie (pronounced “try”) is a 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 a *set* or a *map* where keys are strings

Tries: example

Make this map into a trie:

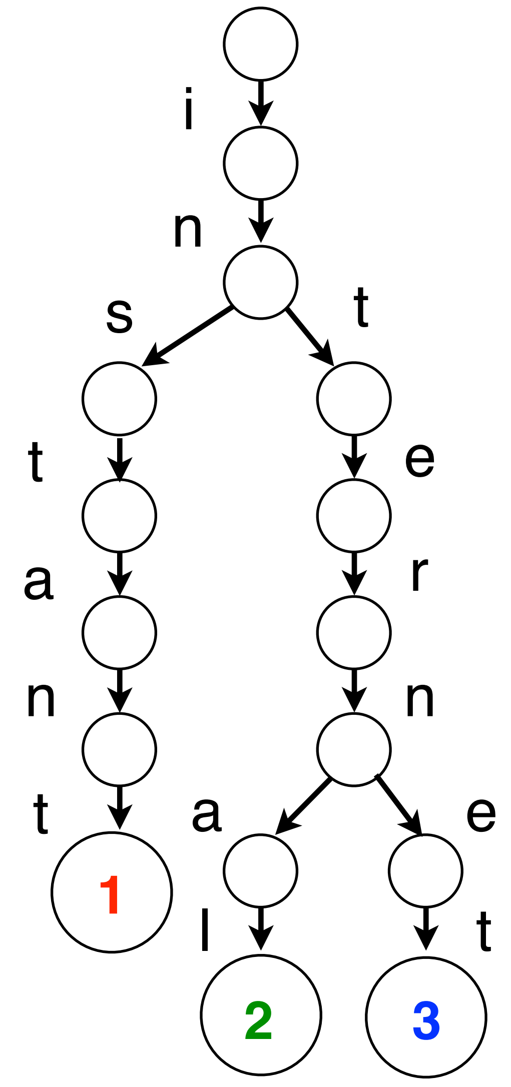
Key	Value
instant	1
internal	2
internet	3

The smallest tree such that:

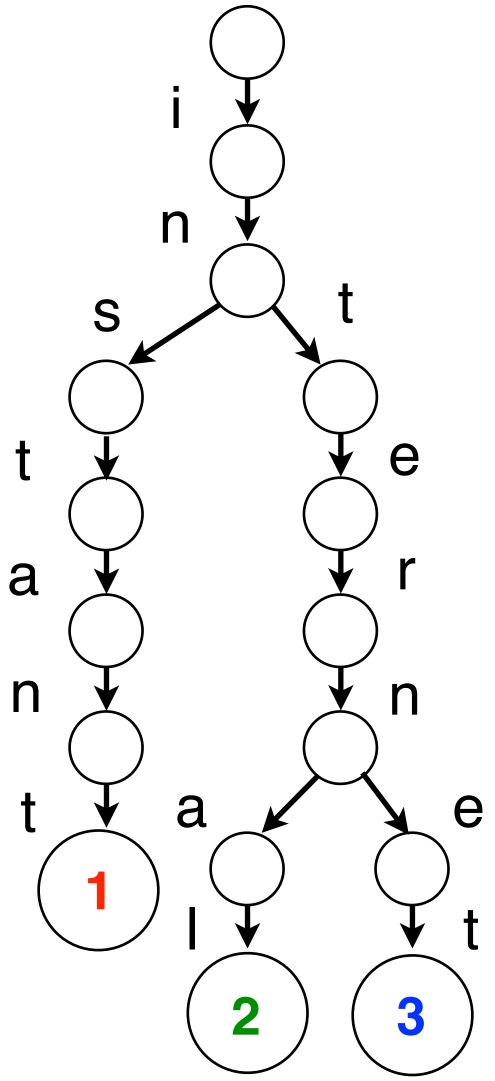
Each edge is labeled with a character $c \in \Sigma$

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

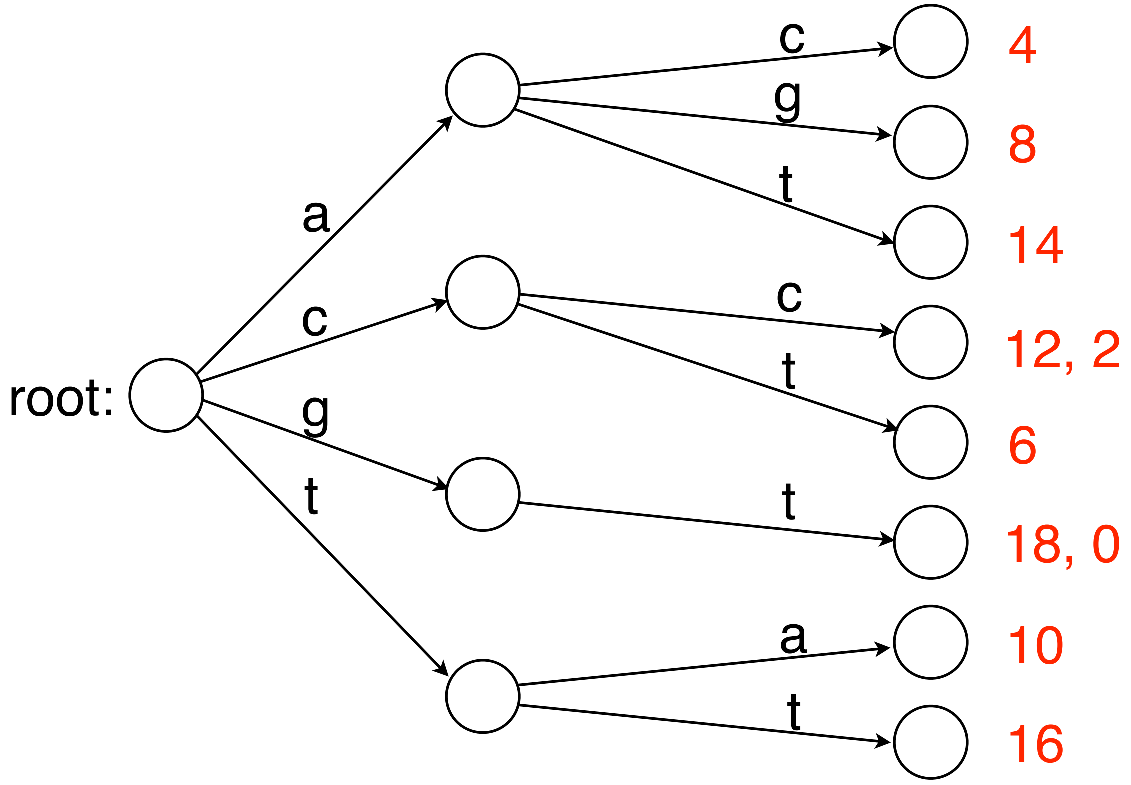
If total length of all keys is *N*, trie has ***O*(*N*)** nodes

Tries: another example

JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

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

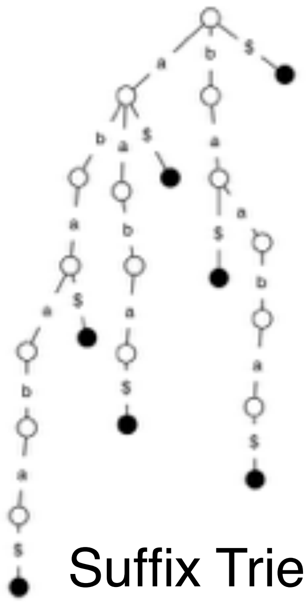
ac	4
ag	8
at	14
cc	12
cc	2
ct	6
gt	18
gt	0
ta	10
tt	16



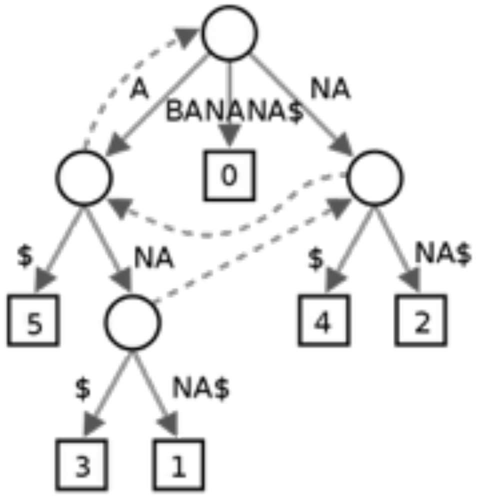
Indexing with suffixes

Until now, our indexes have been 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

$	B	A	N	A	N	**A**
A	$	B	A	N	A	**N**
A	N	A	$	B	A	**N**
A	N	A	N	A	$	**B**
B	A	N	A	N	A	**$**
N	A	$	B	A	N	**A**
N	A	N	A	$	B	**A**

FM Index

Suffix trie

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

T:

```
G T T A T A G C T G A T C G C G G C G T A G C G G $
G T T A T A G C T G A T C G C G G C G T A G C G G $
  T T A T A G C T G A T C G C G G C G T A G C G G $
    T A T A G C T G A T C G C G G C G T A G C G G $
      A T A G C T G A T C G C G G C G T A G C G G $
        T A G C T G A T C G C G G C G T A G C G G $
          A G C T G A T C G C G G C G T A G C G G $
            G C T G A T C G C G G C G T A G C G G $
              C T G A T C G C G G C G T A G C G G $
                T G A T C G C G G C G T A G C G G $
                  G A T C G C G G C G T A G C G G $
                    A T C G C G G C G T A G C G G $
                      T C G C G G C G T A G C G G $
                        C G C G G C G T A G C G G $
                          G C G G C G T A G C G G $
                            C G G C G T A G C G G $
                              G G C G T A G C G G $
                                G C G T A G C G G $
                                  C G T A G C G G $
                                    G T A G C G G $
                                      T A G C G G $
                                        A G C G G $
                                          G C G G $
                                            C G G $
                                              G G $
                                                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:

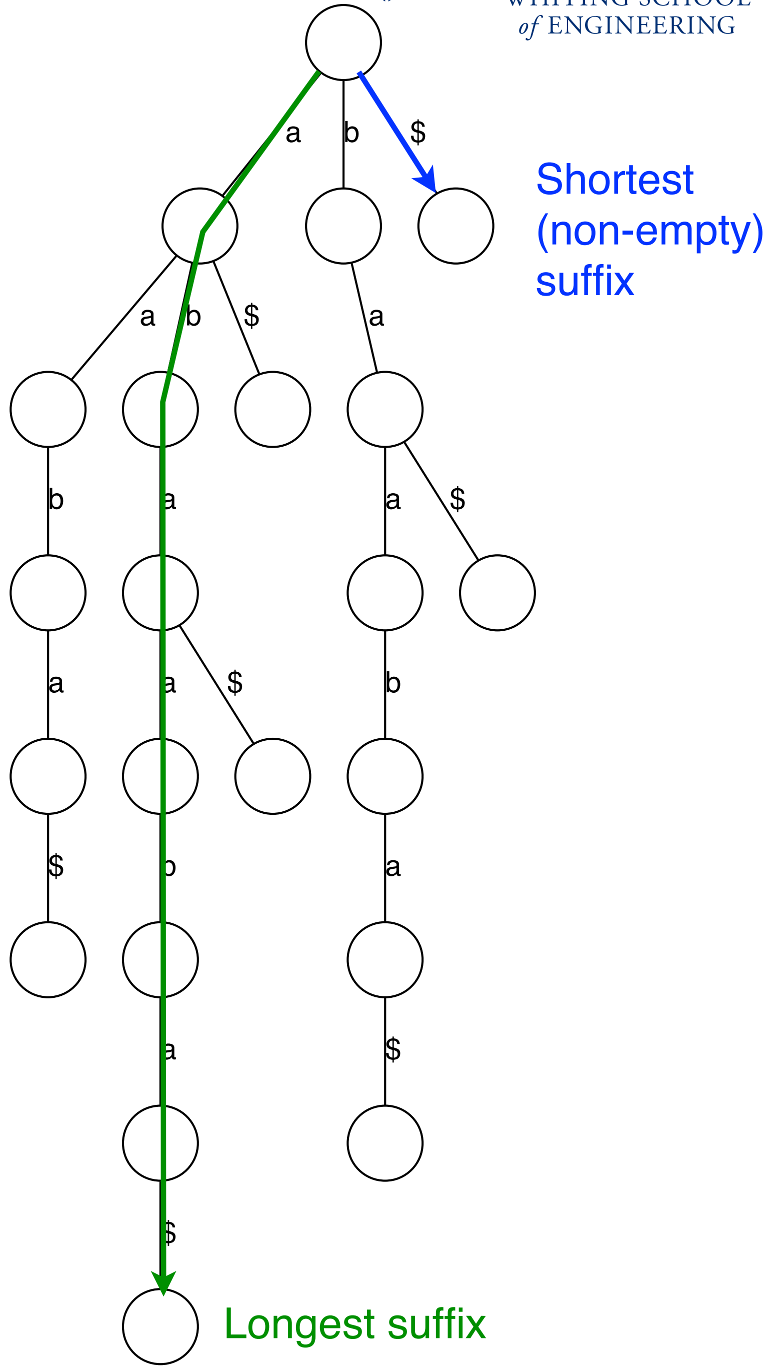
```
G T T A T A G C T G A T C G C G G C G T A G C G G $
G T T A T A G C T G A T C G C G G C G T A G C G G $
  T T A T A G C T G A T C G C G G C G T A G C G G $
    T A T A G C T G A T C G C G G C G T A G C G G $
      A T A G C T G A T C G C G G C G T A G C G G $
        T A G C T G A T C G C G G C G T A G C G G $
          A G C T G A T C G C G G C G T A G C G G $
            G C T G A T C G C G G C G T A G C G G $
              C T G A T C G C G G C G T A G C G G $
                T G A T C G C G G C G T A G C G G $
                  G A T C G C G G C G T A G C G G $
                    A T C G C G G C G T A G C G G $
                      T C G C G G C G T A G C G G $
                        C G C G G C G T A G C G G $
                          G C G G C G T A G C G G $
                            C G G C G T A G C G G $
                              G G C G T A G C G G $
                                G C G T A G C G G $
```

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 **$**?

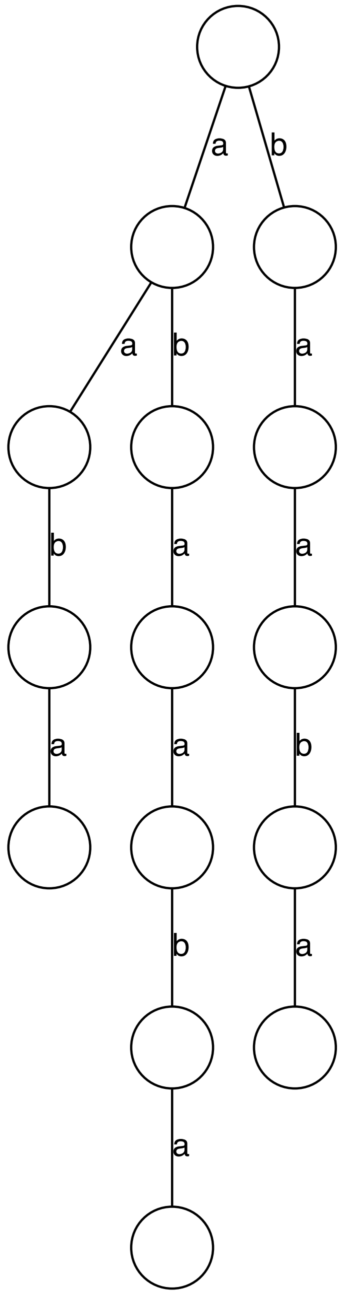


Suffix trie

T: abaaba

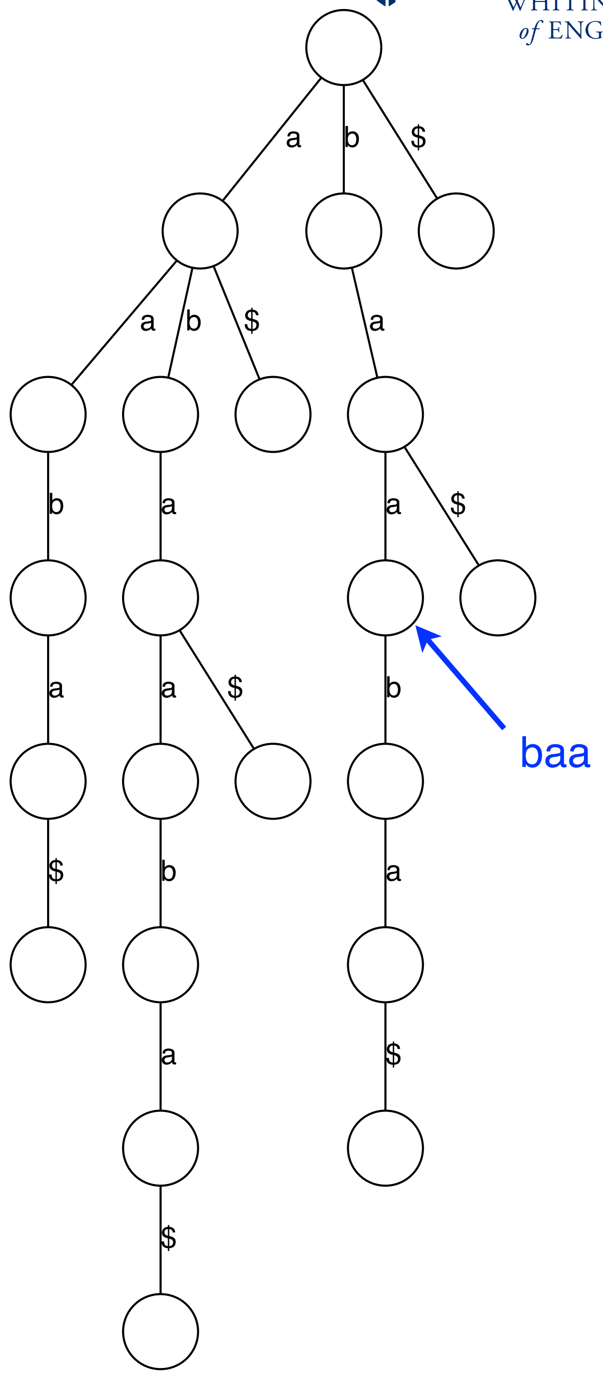
Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

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

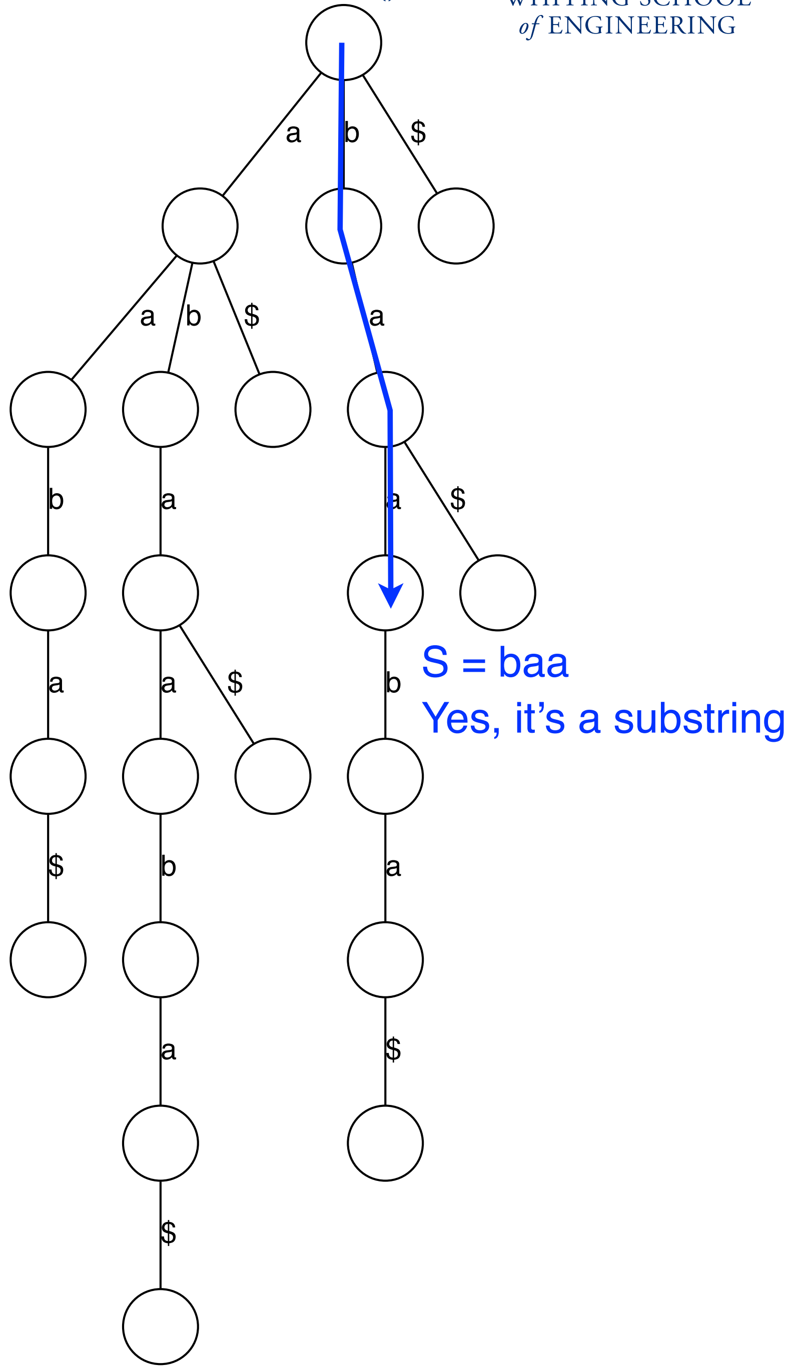
How do we check whether a string *S* is a substring of *T*?

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.

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

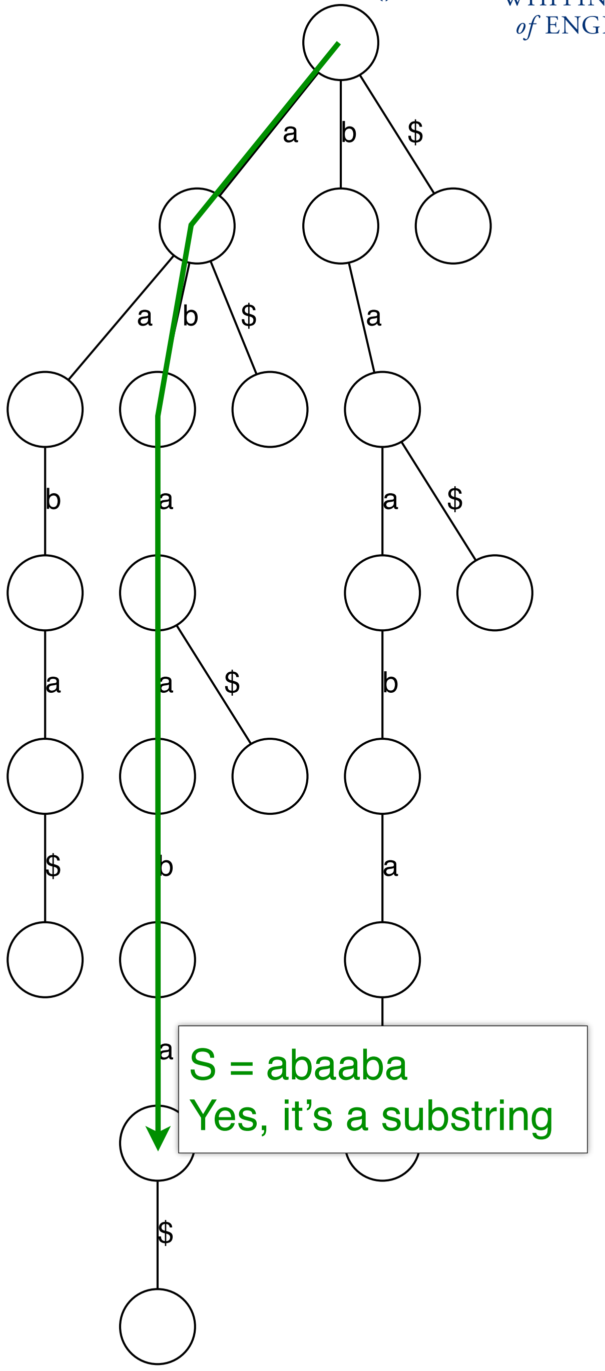
How do we check whether a string *S* is a substring of *T*?

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*



Suffix trie

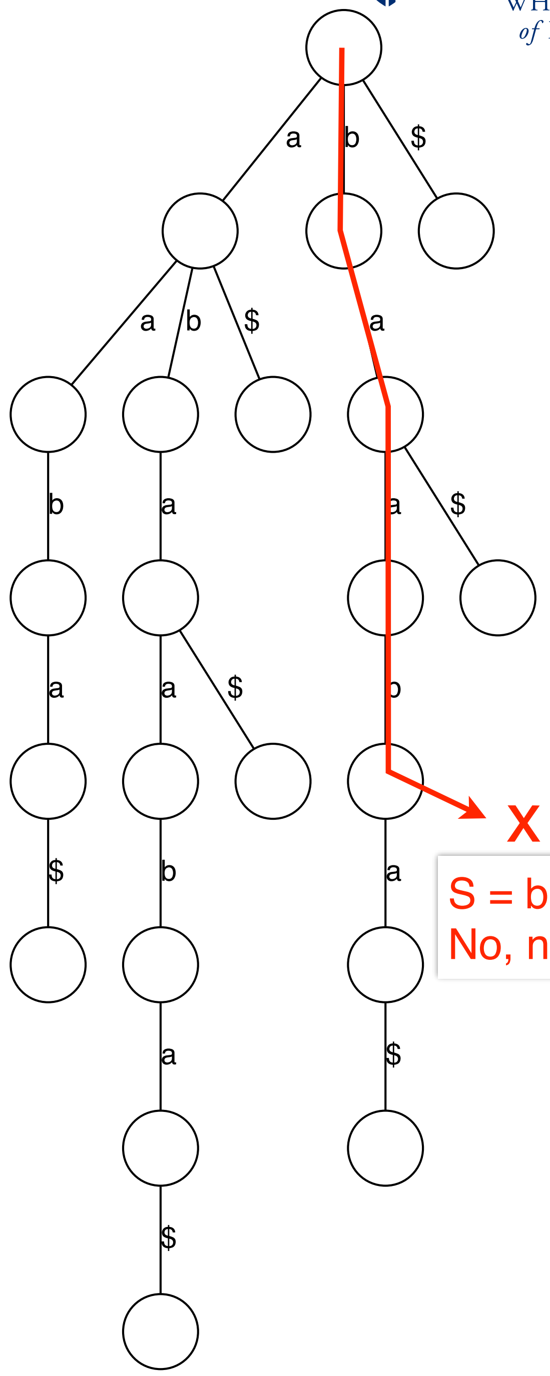
How do we check whether a string *S* is a substring of *T*?

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*

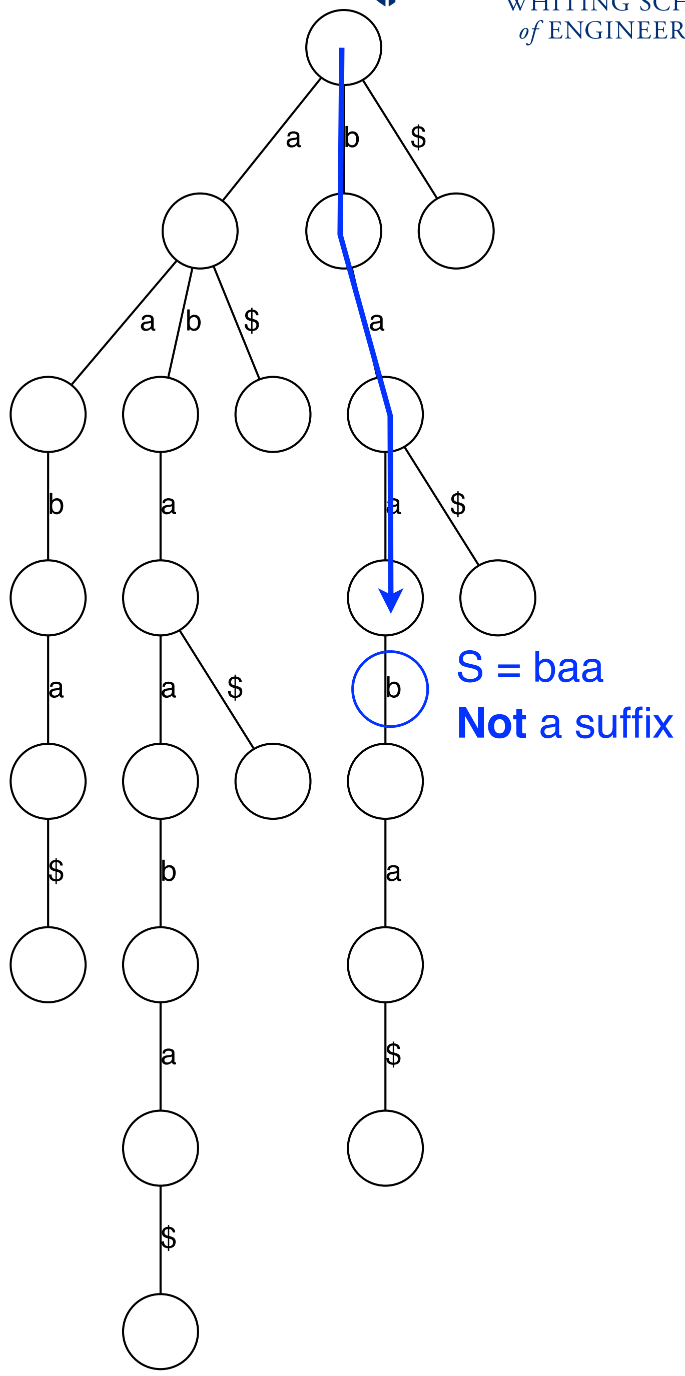


S = baabb
No, not a substring

Suffix trie

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

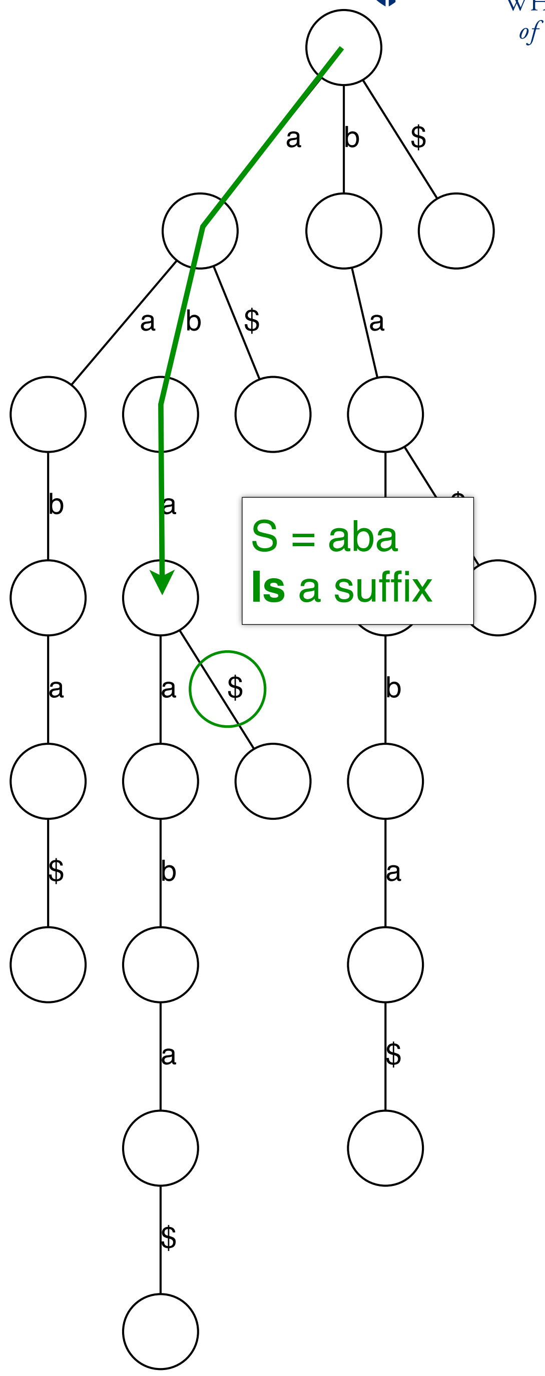
Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled **$**



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 **$**

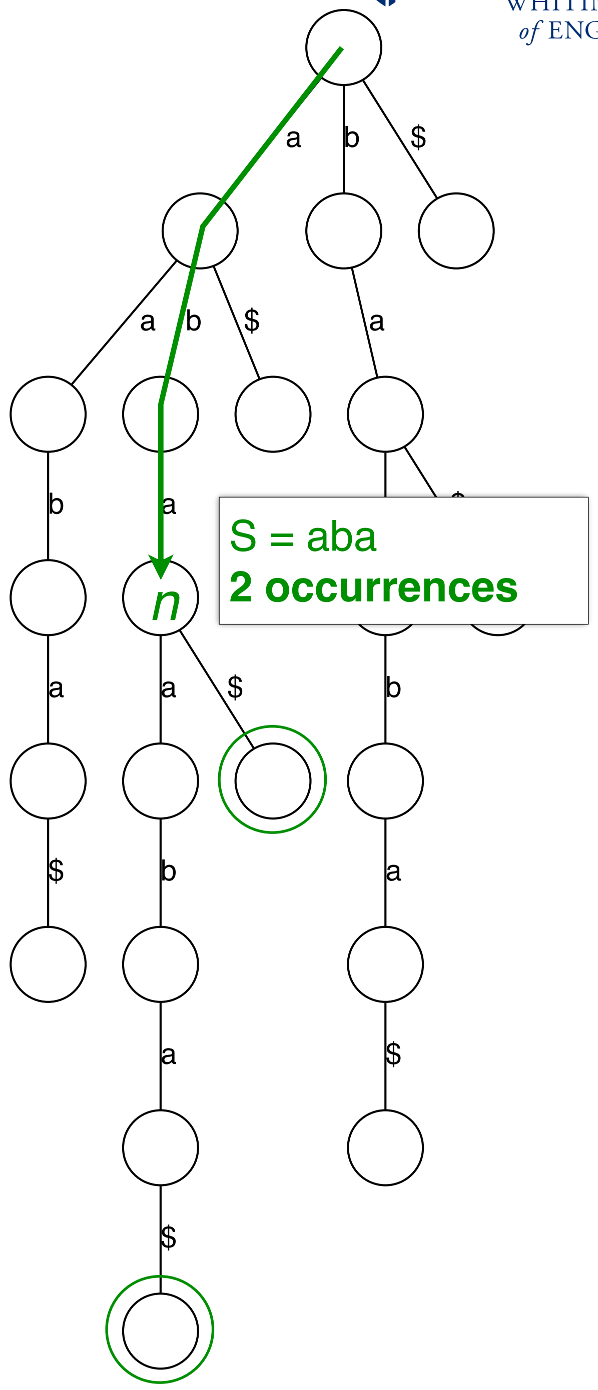


Suffix trie

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*.

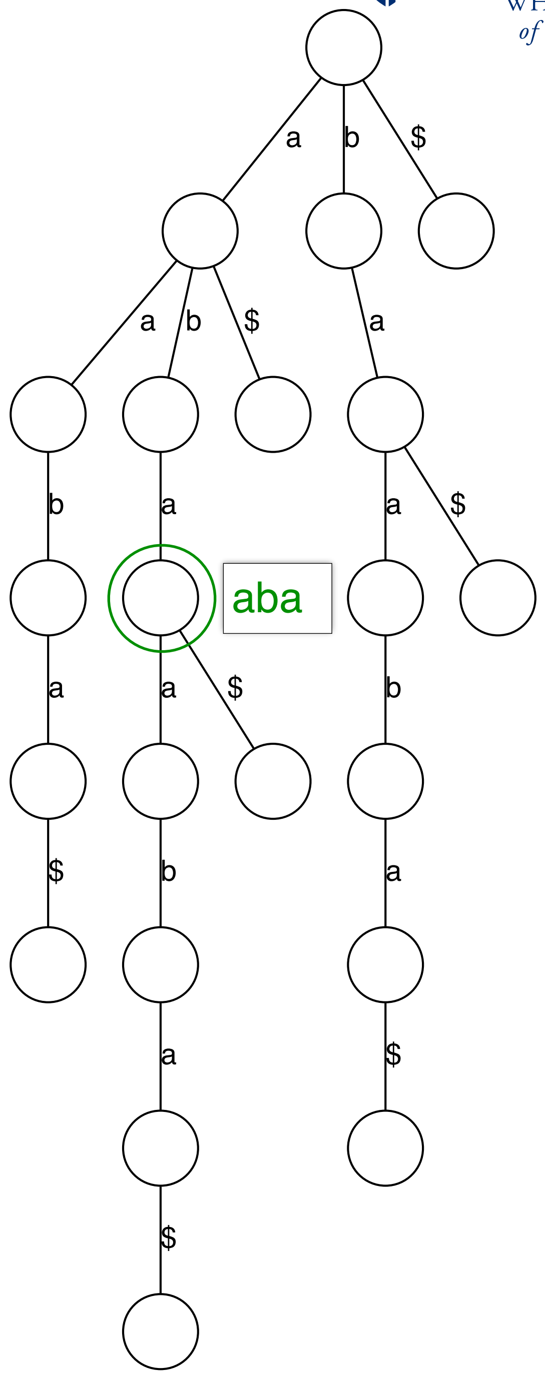
Leaves can be counted with depth-first traversal.



Suffix trie

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

Find the deepest node with more than one child



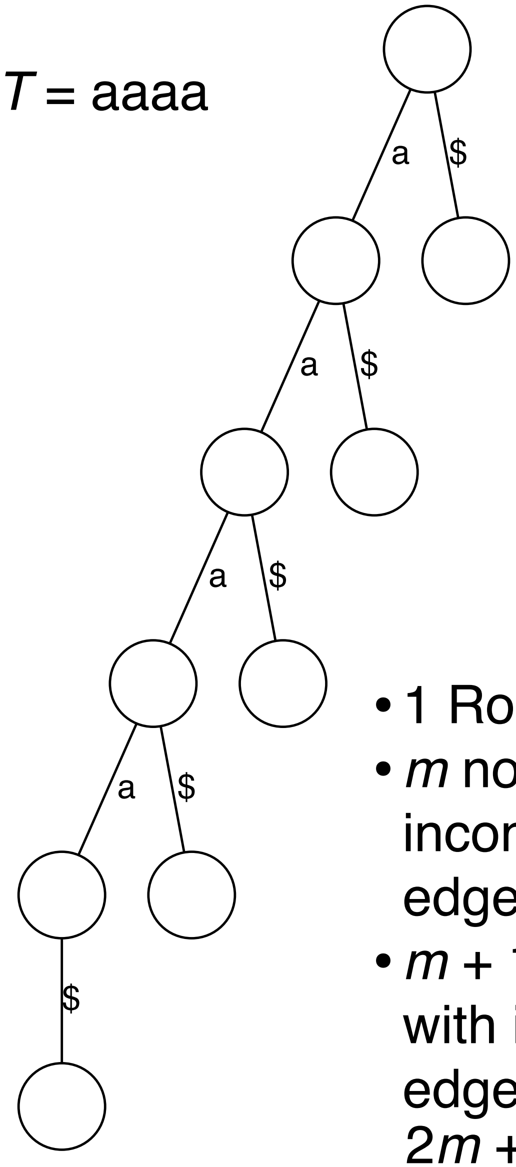
Suffix trie

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?

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

T = aaaa



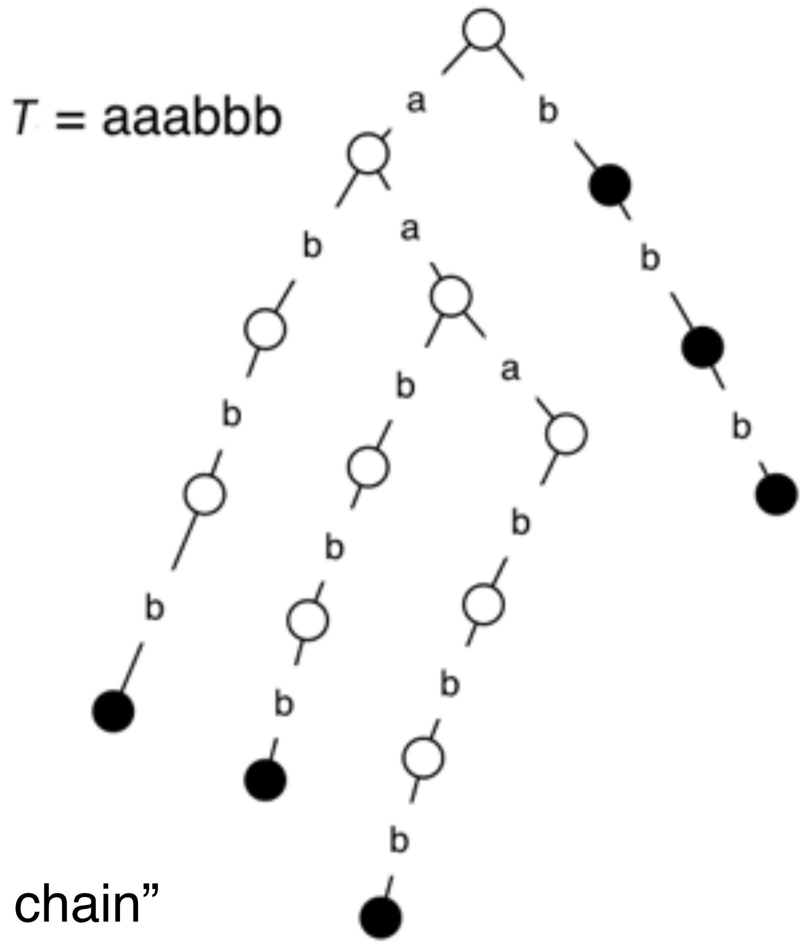
- 1 Root
- m nodes with incoming **a** edge
- $m + 1$ nodes with incoming **$** edge

$2m + 2$ nodes

Suffix trie

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

Yes: $a^n b^n$



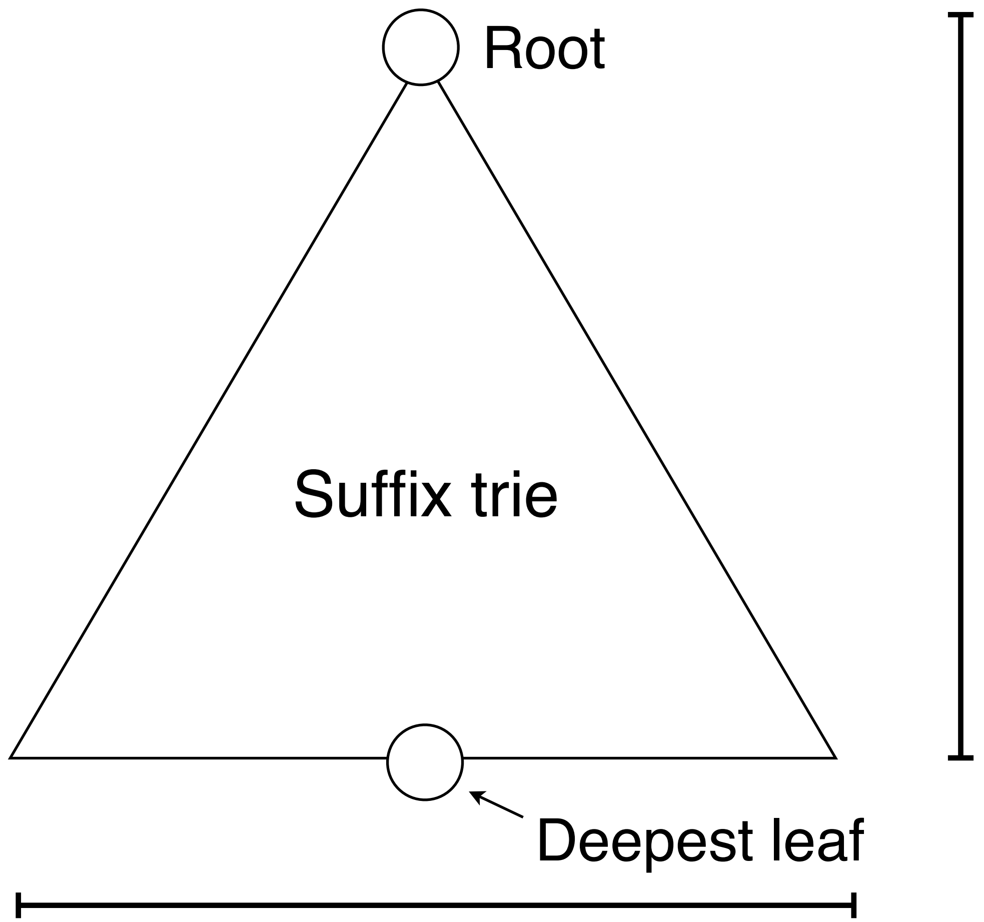
- 1 root
- n nodes along "b chain," right
- n nodes along "a chain," middle
- n chains of n "b" nodes hanging off each"a chain" node
- $2n + 1$ \$ leaves (not shown)

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

Figure & example by Carl Kingsford

Suffix trie: upper bound on size

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



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

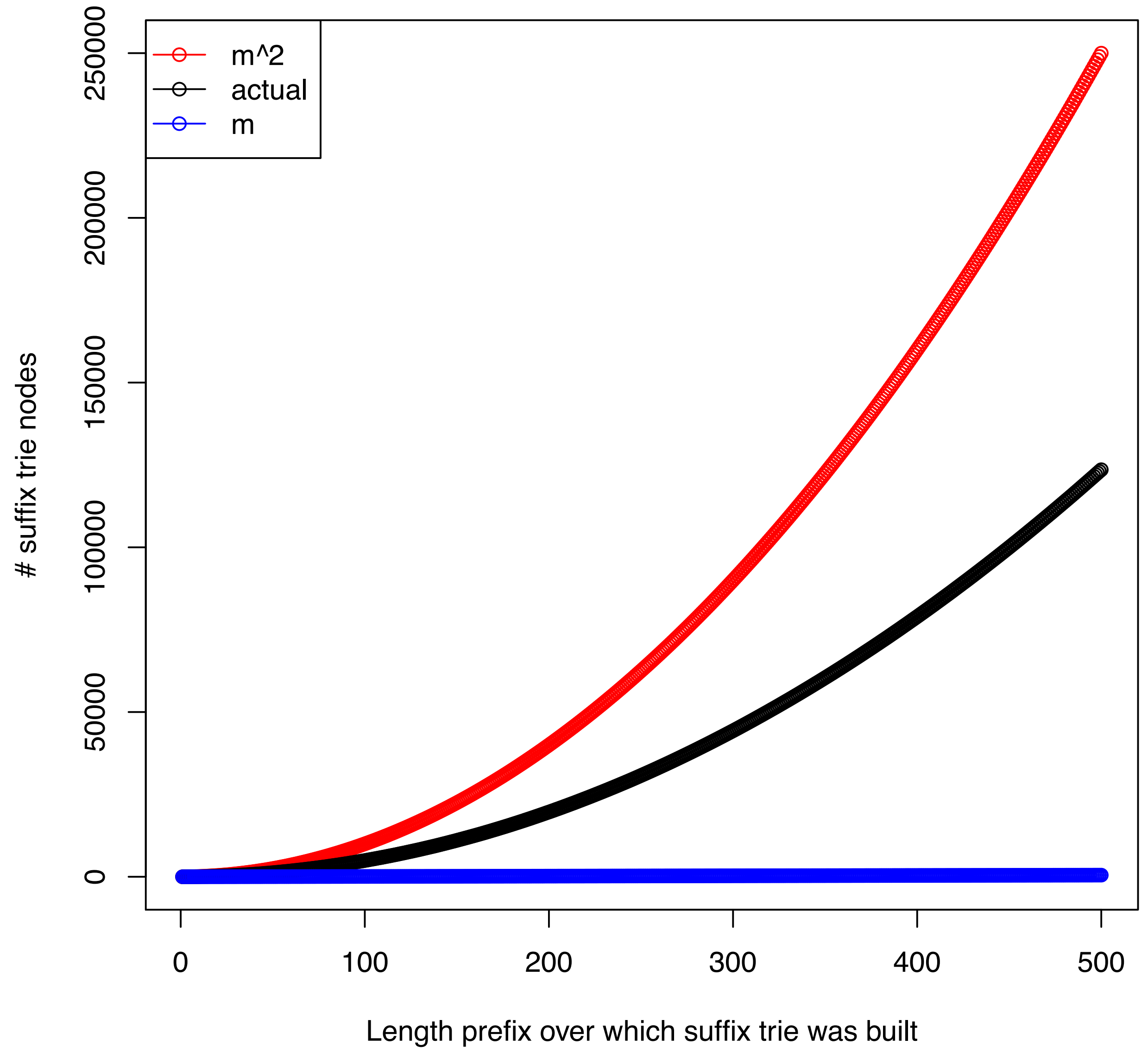
Max # nodes from *left to right*
= max # distinct substrings of any length ≤ m

$O(m^2)$ is worst case

Suffix trie: actual growth

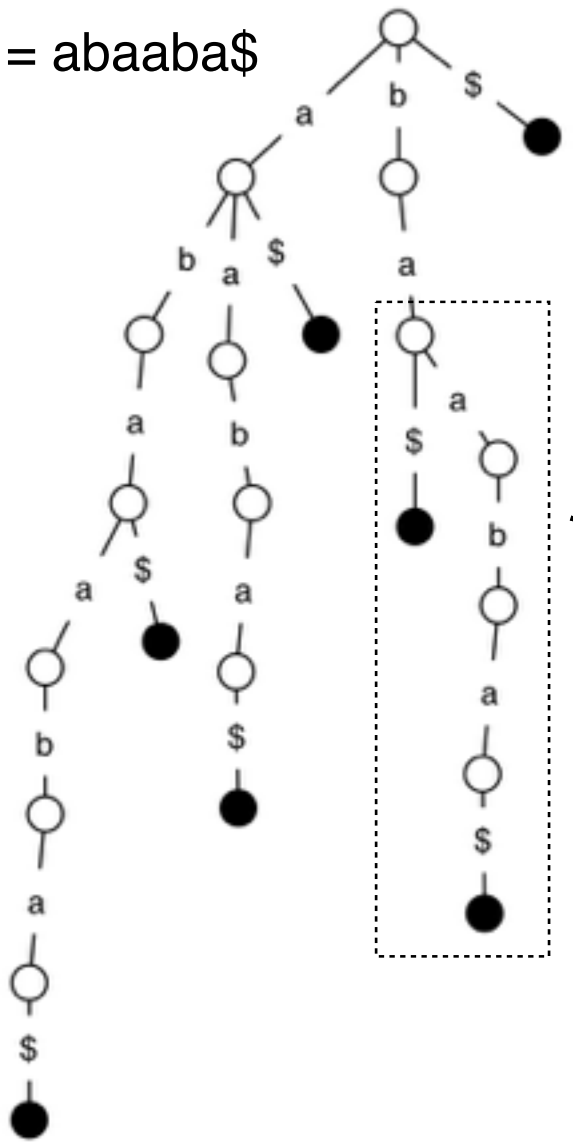
Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length

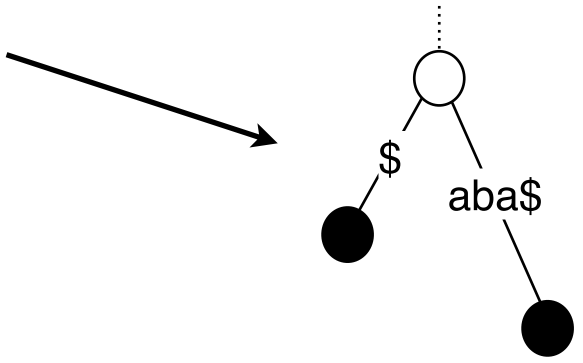


Suffix trie: making it smaller

T = abaaba$



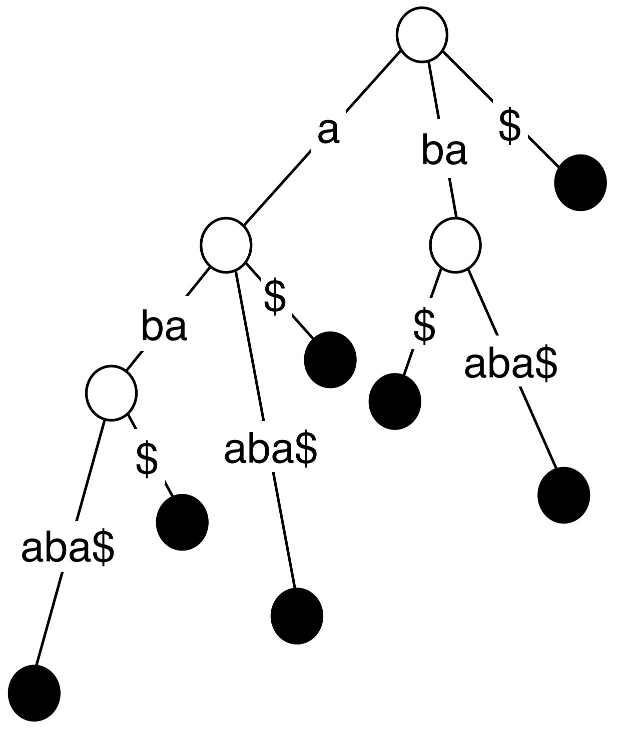
Idea 1: Coalesce non-branching paths into a *single edge* with a *string* label



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

Suffix tree

T = abaaba$



With respect to *m*:

How many leaves? ***m***

How many non-leaf nodes? ***≤ m - 1***

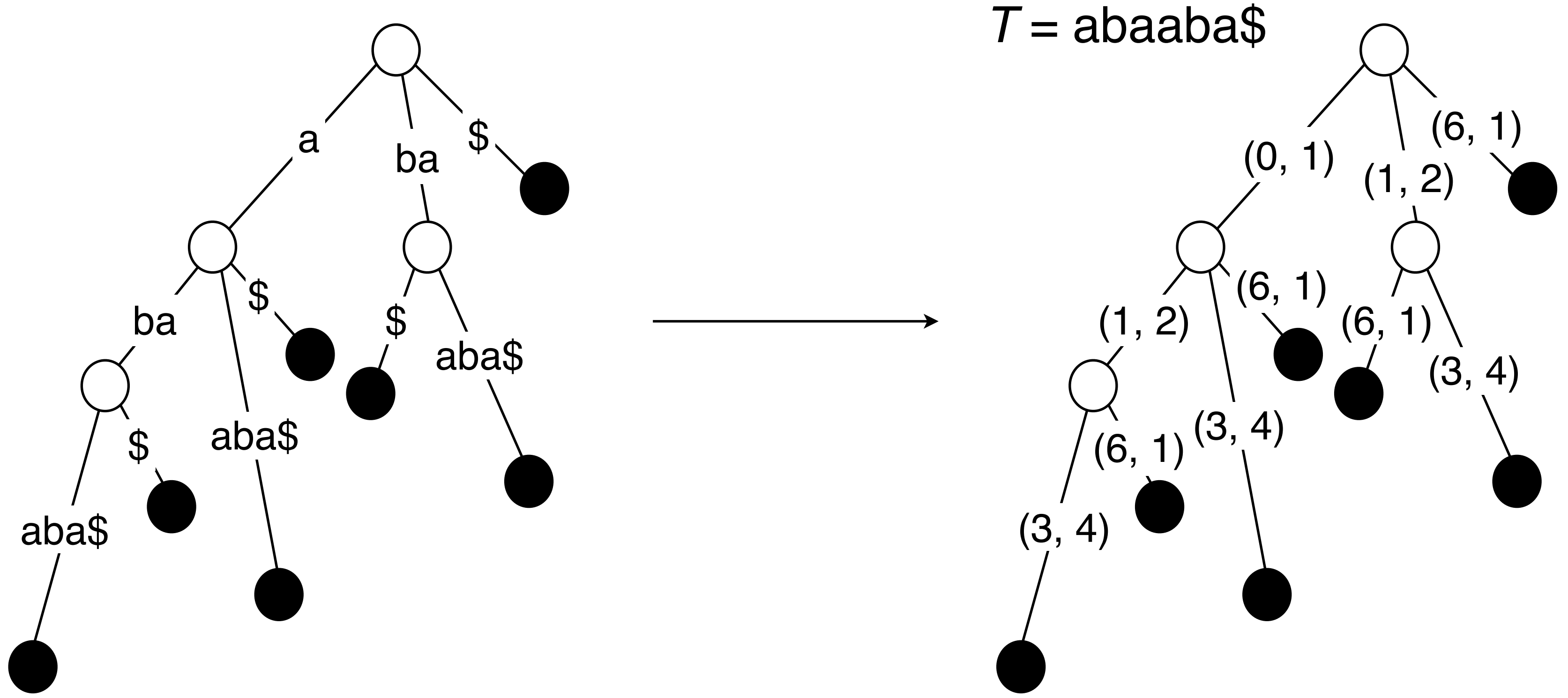
≤ 2*m* -1 nodes total, or *O*(*m*) nodes

Is the total size *O*(*m*) now? **No**: total length of edge labels is quadratic in *m*

Suffix tree

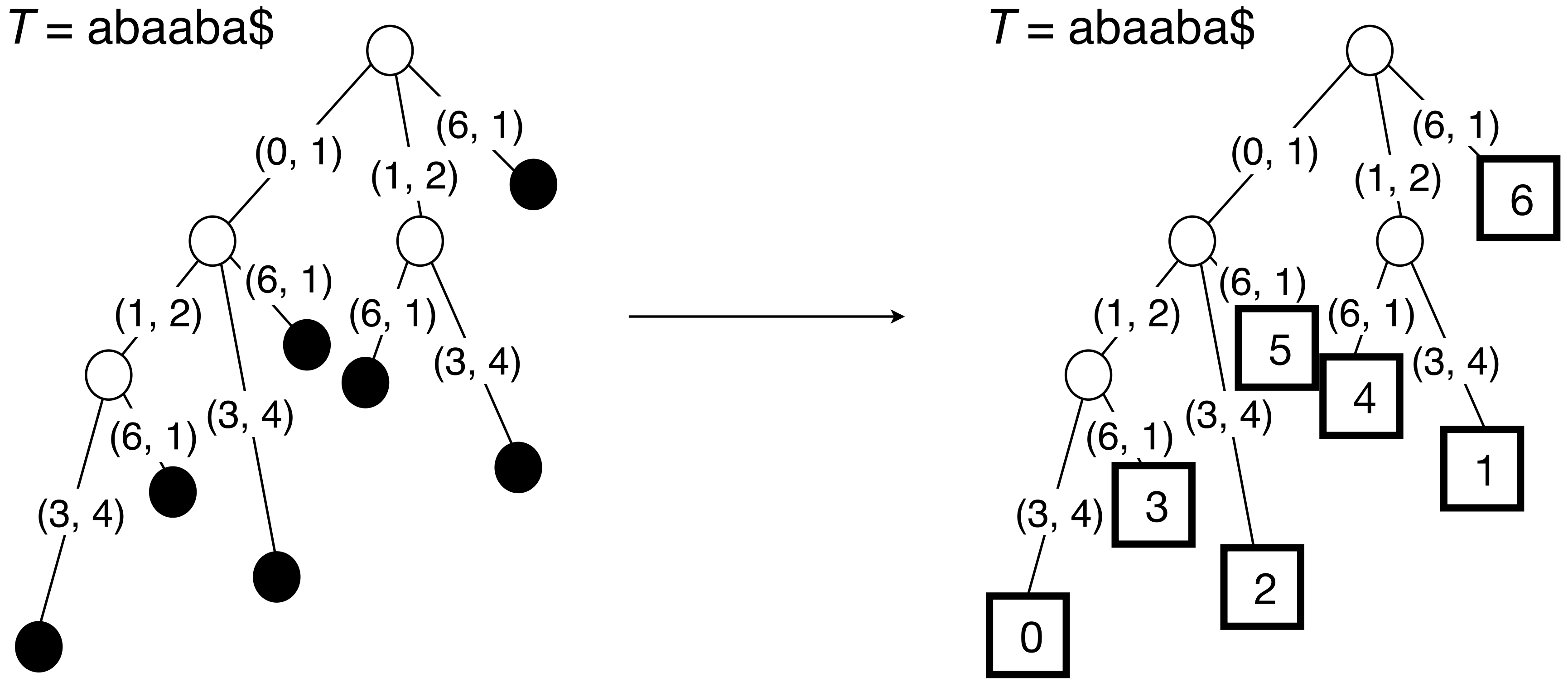
T = abaaba$

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



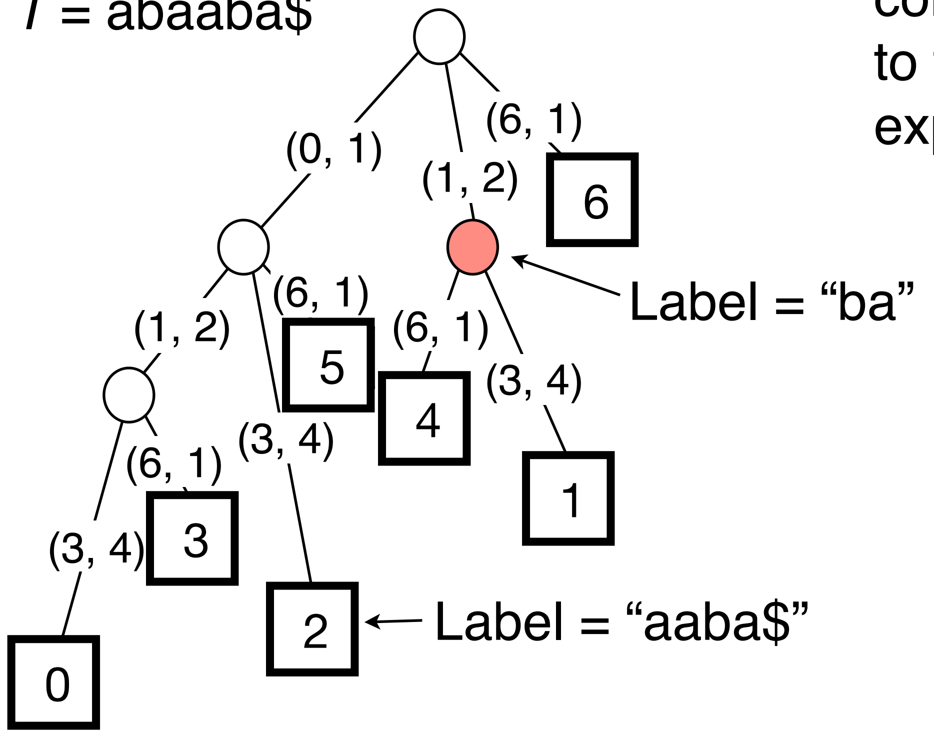
Space required for suffix tree is now *O*(*m*)

Suffix tree: leaves hold offsets



Suffix tree: labels

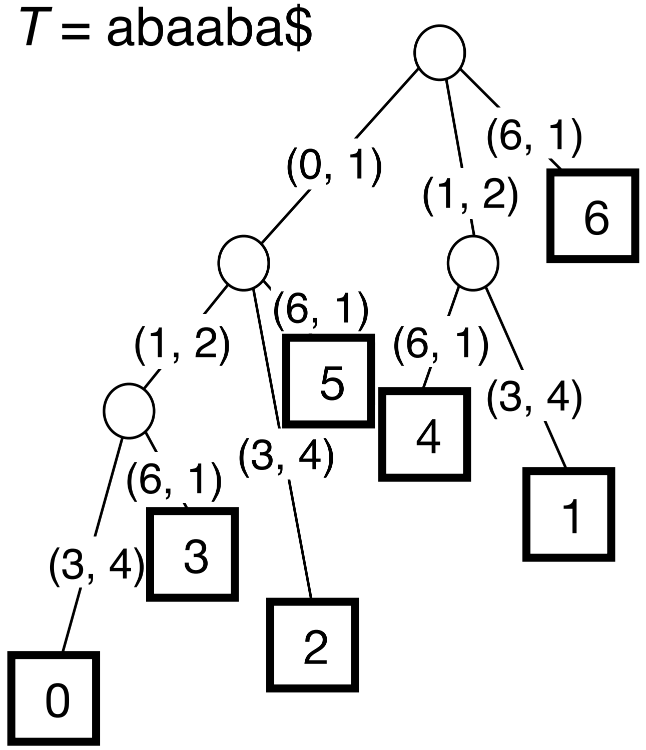
T = abaaba$



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

Suffix tree: labels

T = abaaba$



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

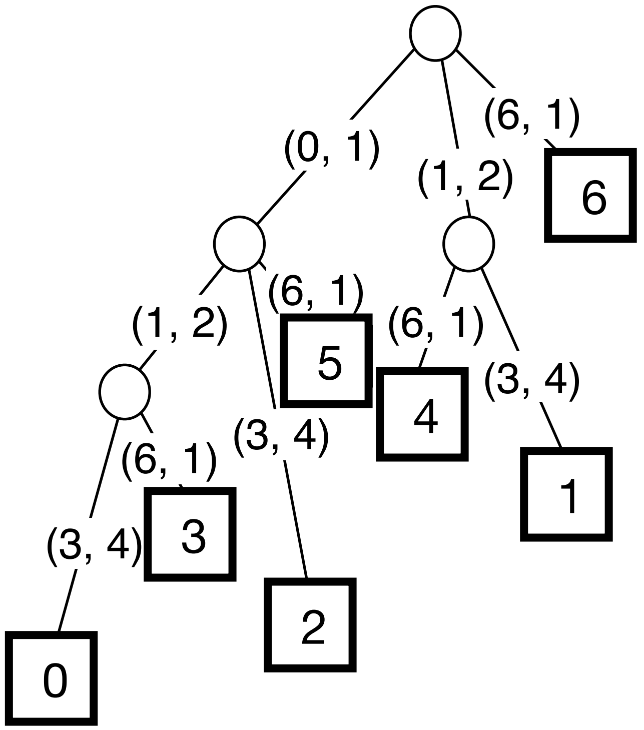
Suffix tree: building

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

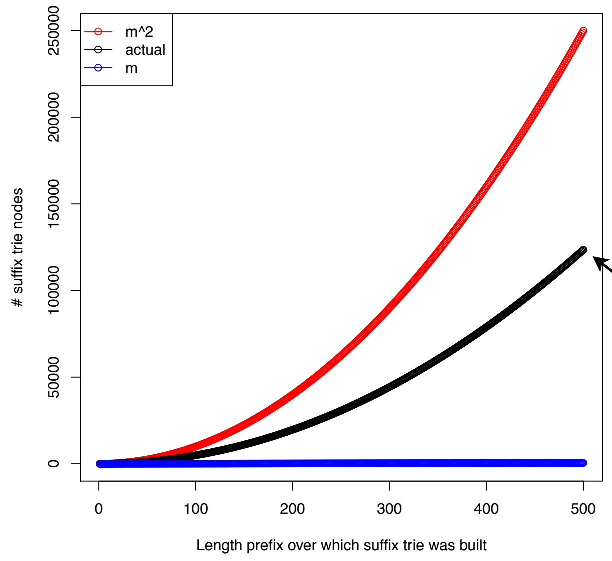


Suffix tree: actual growth

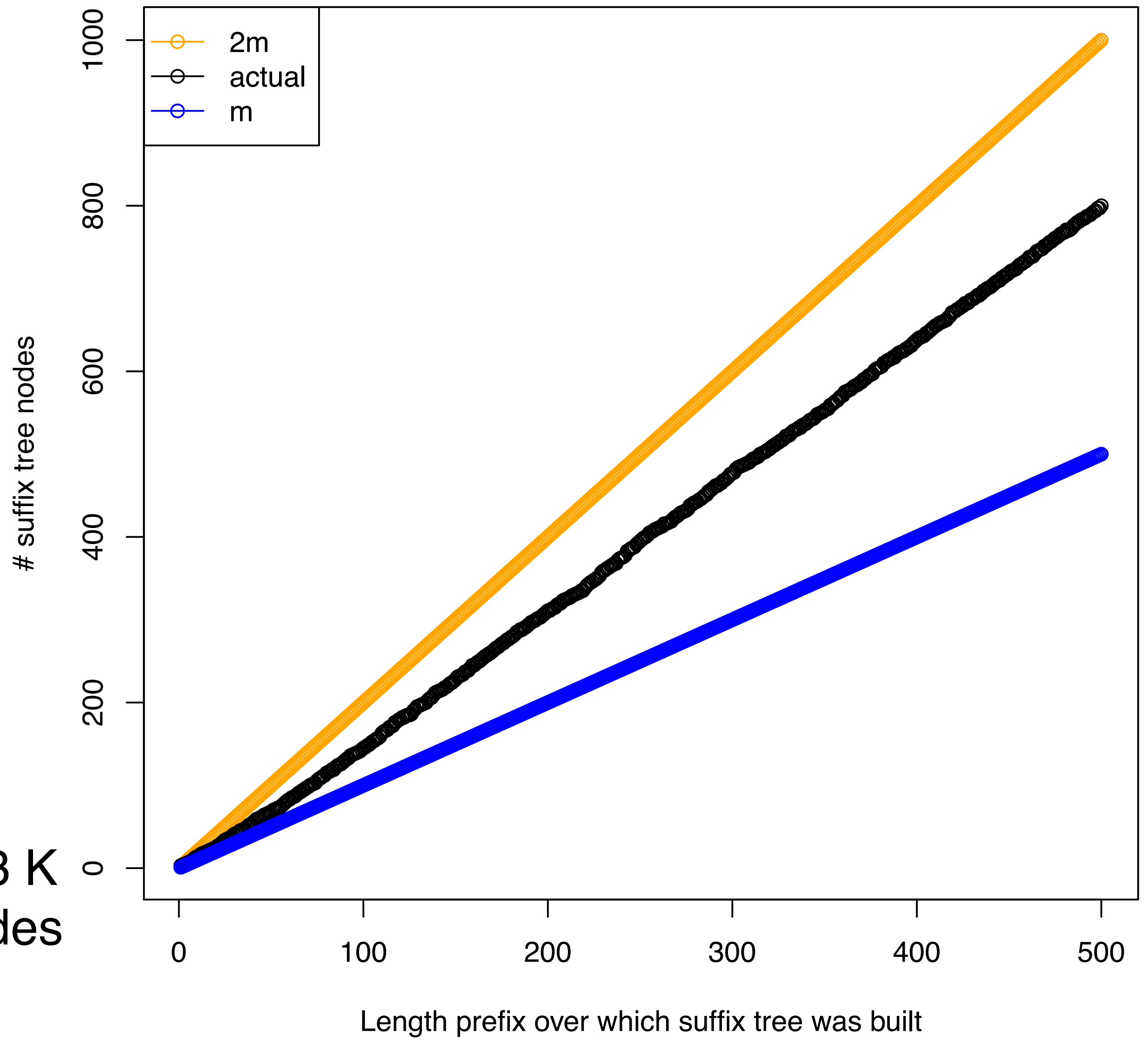
Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Compare with suffix trie:



123 K nodes



Suffix tree: building

Method of choice: Ukkonen's algorithm

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

$O(m)$ time and space

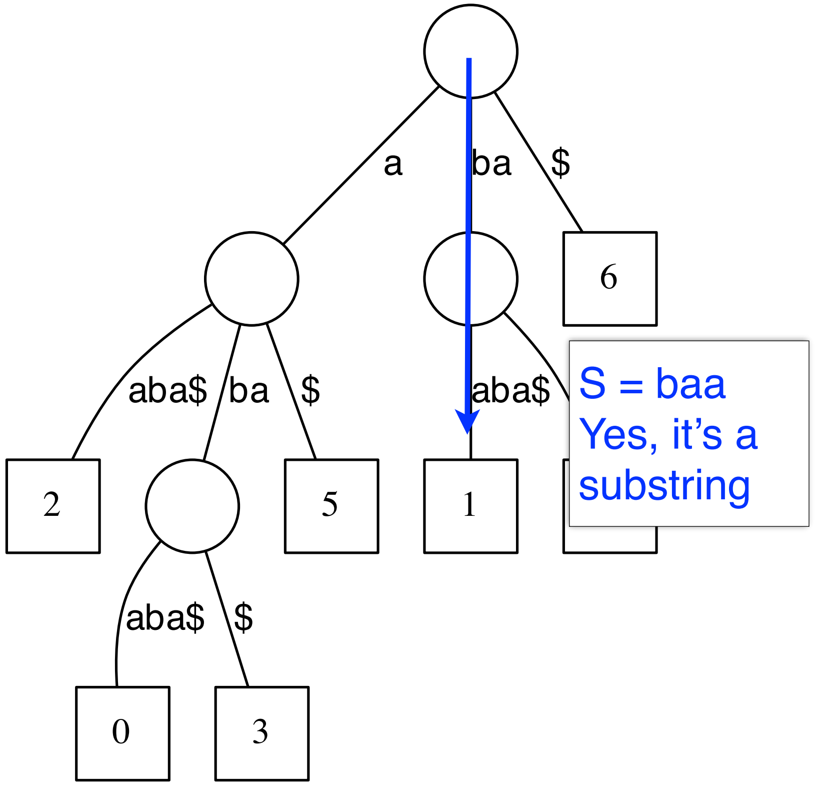
Has *online* property: if T arrives one character at a time, algorithm efficiently updates suffix tree upon each arrival

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

Suffix tree

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

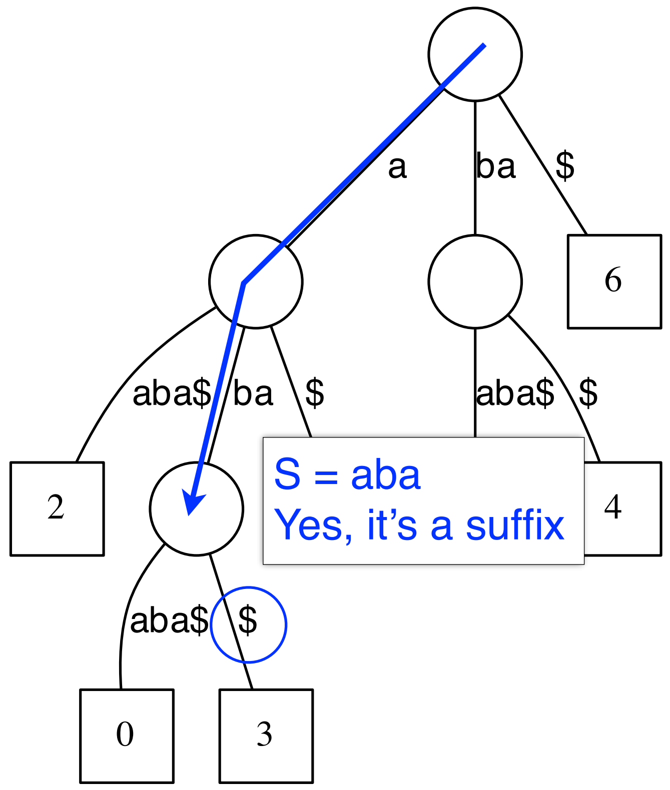
Essentially same procedure as for suffix trie, except we have to deal with coalesced edges



Suffix tree

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

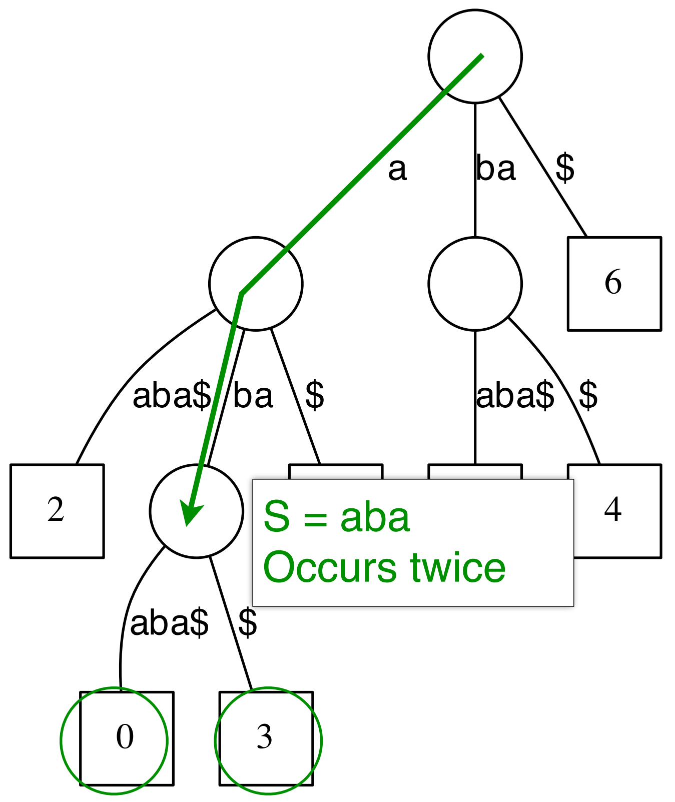
Essentially same procedure as for suffix trie, except we have to deal with coalesced edges



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



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$

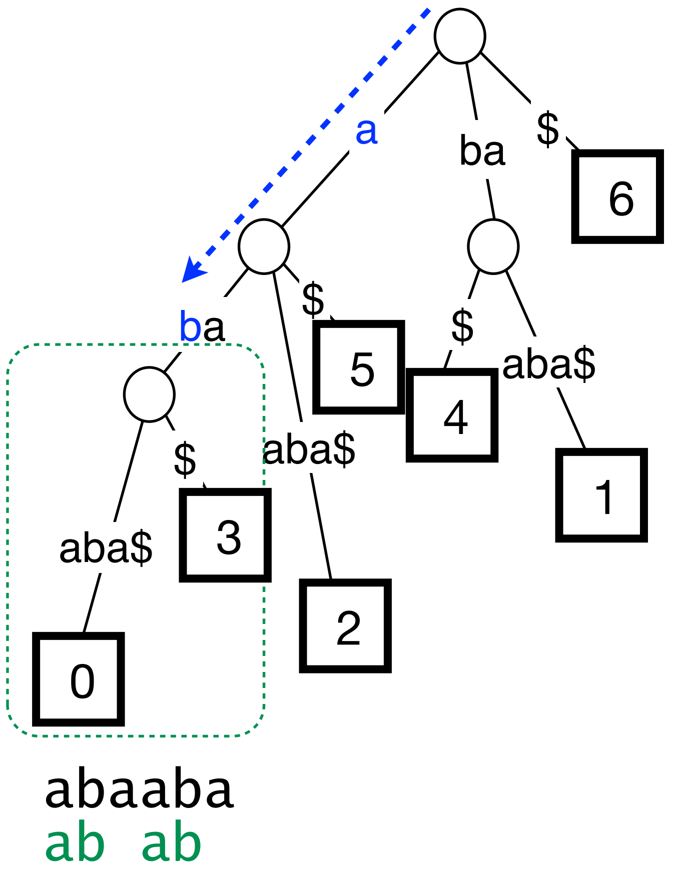
$O(n)$ Step 1: walk down ab path

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 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

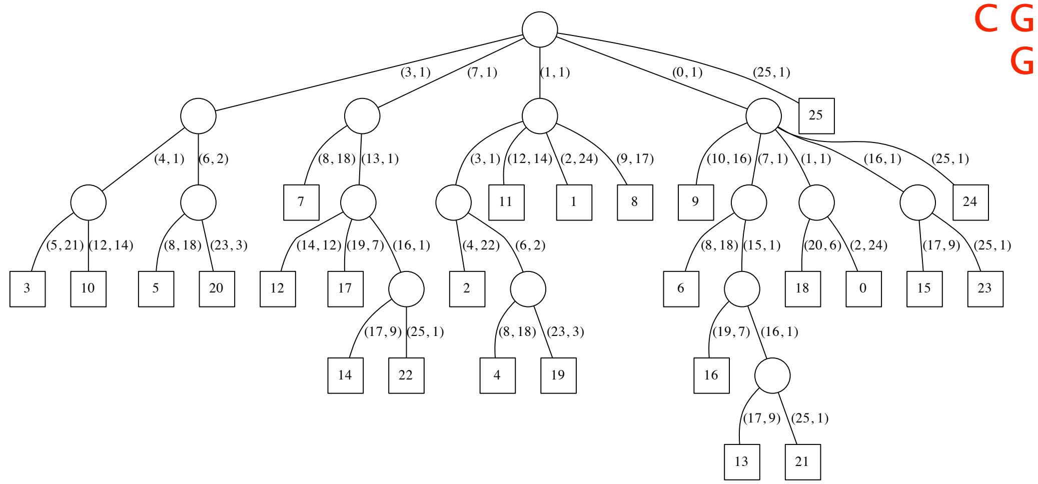
A naive method (e.g. suffix trie) could easily be quadratic

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

m chars

```
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



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

Suffix array

JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

T\$ = abaaba\$ ← As with suffix tree, *T* is part of index

SA(*T*) =
(SA = "Suffix Array")

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

m + 1 integers

Suffix array of *T* is an array of integers in [0, *m*] specifying the lexicographic order of *T*\$'s suffixes

Suffix array

$O(m)$ space, same as suffix tree. Is constant factor smaller?

32-bit integer can distinguish characters in the human genome, so suffix array is ~12 GB, smaller than MUMmer's 47 GB suffix tree.

Suffix array: querying

Is *P* a substring of *T*?

1. For *P* to be a substring, it must be a prefix of ≥1 of *T*'s suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: binary search

Python has `bisect` module for binary search

`bisect.bisect_left(a, x)`: Leftmost offset where we can insert `x` into `a` to maintain sorted order. `a` is already sorted!

`bisect.bisect_right(a, x)`: Like `bisect_left`, but returning *rightmost* instead of leftmost offset

```
from bisect import bisect_left, bisect_right

a = [1, 2, 3, 3, 3, 4, 5]
print(bisect_left(a, 3), bisect_right(a, 3)) # output: (2, 5)

a = [2, 4, 6, 8, 10]
print(bisect_left(a, 5), bisect_right(a, 5)) # output: (2, 2)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

We can straightforwardly use binary search to find a range of elements in a sorted list that *equal* some query:

```
from bisect import bisect_left, bisect_right

strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# Get range of elements that equal query string ‘awl’
st, en = bisect_left(strls, 'awl'), bisect_right(strls, 'awl')

print(st, en) # output: (2, 3)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

Can also use binary search to find a range of elements in a sorted list with some query as a *prefix*:

```
from bisect import bisect_left, bisect_right

strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# Get range of elements with ‘aw’ as a prefix
st, en = bisect_left(strls, 'aw'), bisect_left(strls, 'ax')

print(st, en) # output: (1, 4)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

We can do the same thing for a sorted list of suffixes:

```
from bisect import bisect_left, bisect_right

t = 'abaaba$'
suffixes = sorted([t[i:] for i in xrange(len(t))])

st, en = bisect_left(suffixes, 'aba'),
         bisect_left(suffixes, 'abb')

print(st, en) # output: (3, 5)
```

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Python example: http://nbviewer.ipython.org/6753277

Suffix array: querying

Is *P* a substring of *T*?

Do binary search, check whether *P* is a prefix of the suffix there

How many times does *P* occur in *T*?

Two binary searches yield the range of suffixes with *P* as prefix; size of range equals # times *P* occurs in *T*

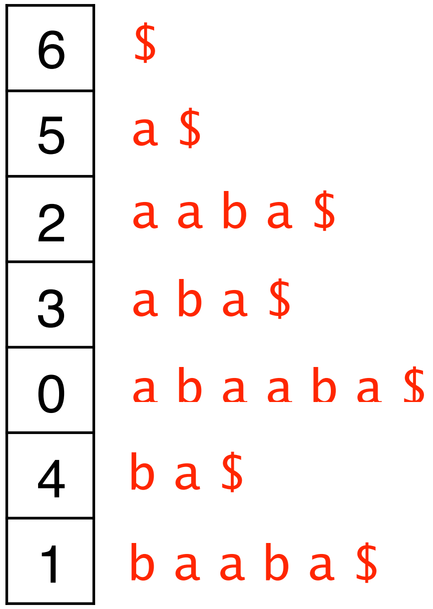
Worst-case time bound?

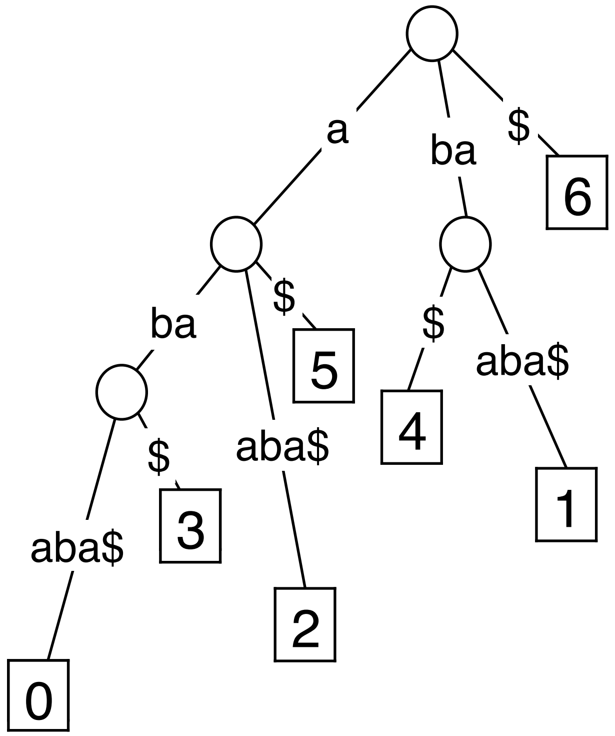
O(log$_2$ *m*) bisections, *O*(*n*) comparisons per bisection, so O(*n* log *m*)

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Contrast suffix array: O(*n* log *m*) with suffix tree: O(*n*)





But we can improve bound for suffix array...

Suffix array: querying

Consider further: binary search for suffixes with *P* as a prefix

Assume there's no **$** in *P*. So *P* can't be equal to a suffix.

Initialize $\underset{\substack{\uparrow \\ \text{“left”}}}{\boldsymbol{l}} = 0$, $\underset{\substack{\uparrow \\ \text{“center”}}}{\boldsymbol{c}} = \mathit{floor}(m/2)$ and $\underset{\substack{\uparrow \\ \text{“right”}}}{\boldsymbol{r}} = m$ (just past last elt of SA)

Notation: We'll use use **SA[*l*]** to refer to the suffix corresponding to suffix-array element ***l***. We could write *T*[**SA[*l*]**:], but that's too verbose.

Throughout the search, invariant is maintained:

$$\mathbf{SA}[\boldsymbol{l}] < P < \mathbf{SA}[\boldsymbol{r}]$$

Suffix array: querying

Throughout search, invariant is maintained:

$\mathbf{SA}[l] < P < \mathbf{SA}[r]$

What do we do at each iteration?

Let c = floor(($r + l$) / 2)
If $P < \mathbf{SA}[c]$, either stop or let $r = c$ and iterate
If $P > \mathbf{SA}[c]$, either stop or let $l = c$ and iterate

When to stop?

$P < \mathbf{SA}[c]$ and $c = l + 1$ - answer is c

$P > \mathbf{SA}[c]$ and $c = r - 1$ - answer is r

Suffix array: querying

```
def binarySearchSA(t, sa, p):
    assert t[-1] == '$' # t already has terminator
    assert len(t) == len(sa) # sa is the suffix array for t
    if len(t) == 1: return 1
    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]
    while True:
        c = (l + r) // 2
        # determine whether p < T[sa[c]:] by doing comparisons
        # starting from left-hand sides of p and T[sa[c]:]
        plt = True # assume p < T[sa[c]:] until proven otherwise
        i = 0
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far
        if plt:
            if c == l + 1: return c
            r = c
        else:
            if c == r - 1: return r
            l = c
```

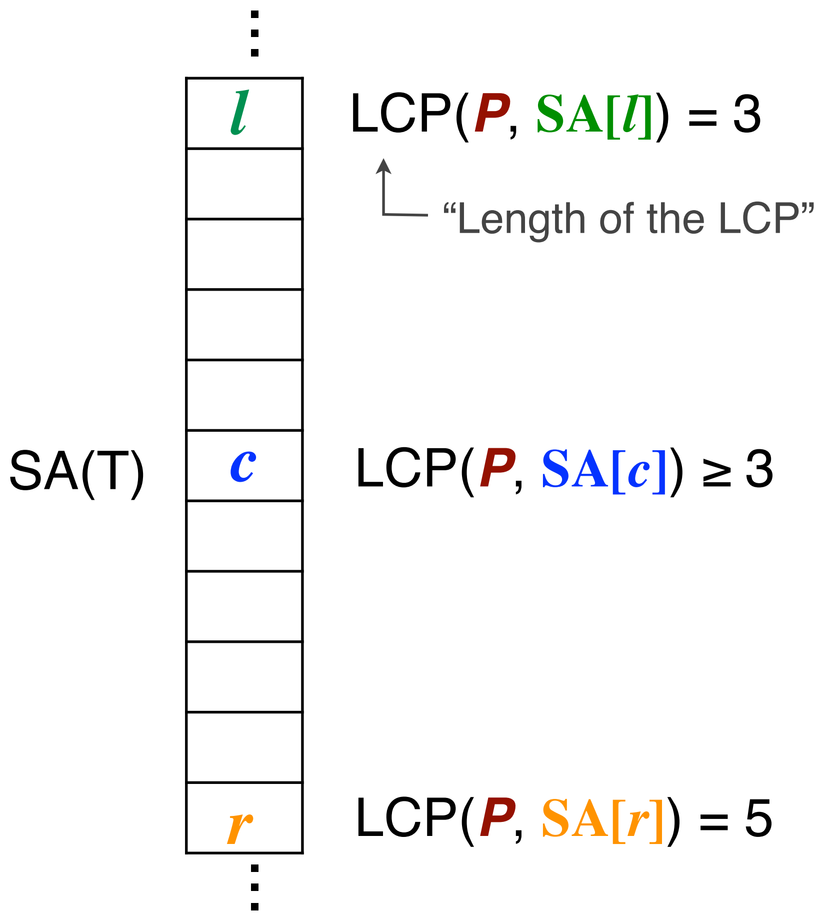
# loop iterations ≈ length of Longest Common Prefix (LCP) of *P* and **SA[*c*]**

If we already know something about LCP of *P* and **SA[*c*]**, we can save work

Python example: http://nbviewer.ipython.org/6765182

Suffix array: querying

Say we're comparing ***P*** to **SA[*c*]** and we've already compared *P* to **SA[*l*]** and **SA[*r*]** in previous iterations.



SA(T)
l
c
r
LCP(***P***, **SA[*l*]**) = 3
"Length of the LCP"
LCP(***P***, **SA[*c*]**) ≥ 3
LCP(***P***, **SA[*r*]**) = 5

More generally:

LCP(***P***, **SA[*c*]**) ≥
min(LCP(***P***, **SA[*l*]**), LCP(***P***, **SA[*r*]**)**)**

We can skip character comparisons

Suffix array: querying

```
def binarySearchSA_lcp1(t, sa, p):
    if len(t) == 1: return 1
    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]
    lcp_lp, lcp_rp = 0, 0
    while True:
        c = (l + r) // 2
        plt = True
        i = min(lcp_lp, lcp_rp)
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far
        if plt:
            if c == l + 1: return c
            r = c
            lcp_rp = i
        else:
            if c == r - 1: return r
            l = c
            lcp_lp = i
```

Worst-case time bound is still $O(n \log m)$, but we're closer

Python example: http://nbviewer.ipython.org/6765182

Suffix array: querying review

We saw 2 ways to query (binary search) the suffix array:

1. Typical binary search. Ignores LCPs. $O(n \log m)$.
2. Binary search with some skipping using LCPs between P and T's suffixes. Still $O(n \log m)$ worst case, but near $O(n + \log m)$ in practice.

Gusfield:
"Simple Accelerant"

If we precompute external LCP arrays we can accelerate the search to worst case $O(n + \log m)$ at the additional space cost of 2m integers. "Super Accelerant" algorithm

Suffix array: performance comparison

	Super accelerant	Simple accelerant	No accelerant
python -O	68.78 s	69.80 s	102.71 s
pypy -O	5.37 s	5.21 s	8.74 s
# character comparisons	99.5 M	117 M	235 M

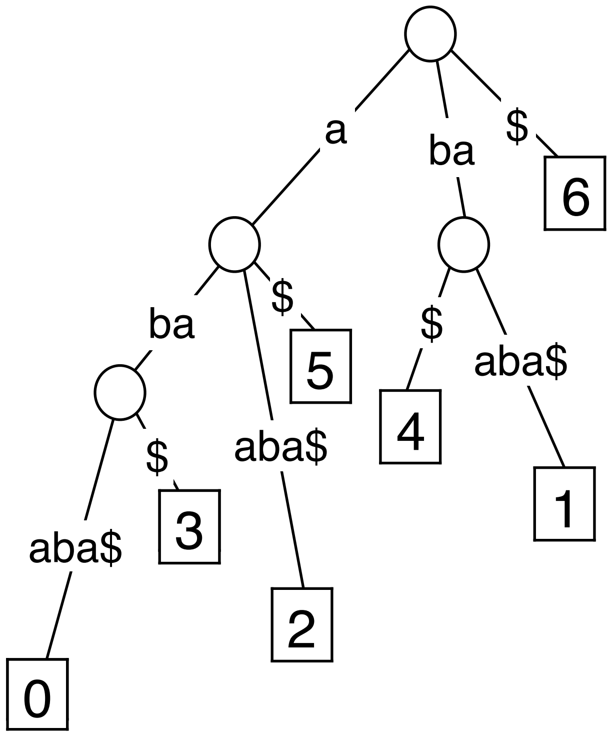
Matching 500K 100-nt substrings to the ~ 5 million nt-long *E. coli* genome. Substrings drawn randomly from the genome.

Index building time not included

Suffix array: building

Given *T*, how to we efficiently build *T*'s suffix array?

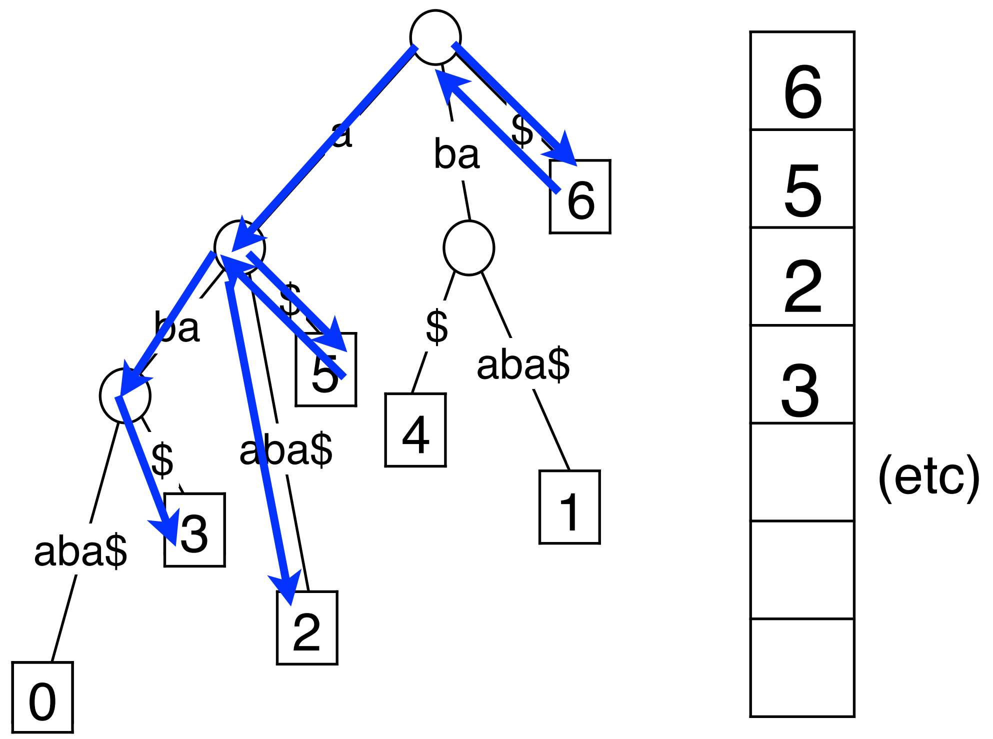
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $



Suffix array: building

Idea: Build suffix tree, do a lexicographic depth-first traversal reporting leaf offsets as we go

Traverse $O(m)$ nodes and emit m integers, so $O(m)$ time assuming edges are already ordered



Suffix array: building

Suffix trees are big. Given *T*, how do we efficiently build *T*'s suffix array *without* first building a suffix tree?

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: sorting suffixes

One idea: Use your favorite sort, e.g., quicksort

0	a b a a b a $
1	b a a b a $
2	a a b a $
3	a b a $
4	b a $
5	a $
6	$

```
def quicksort(q):
    lt, gt = [], []
    if len(q) <= 1:
        return q
    for x in q[1:]:
        if x < q[0]:
            lt.append(x)
        else:
            gt.append(x)
    return quicksort(lt) + q[0:1] + quicksort(gt)
```

Expected time: O($m^2 \log m$)

Not $O(m \log m)$ because a suffix comparison is $O(m)$ time

Suffix array: sorting suffixes

One idea: Use a sort algorithm that's aware that the items being sorted are strings, e.g. "multikey quicksort"

0	a b a a b a $
1	b a a b a $
2	a a b a $
3	a b a $
4	b a $
5	a $
6	$

Essentially $O(m^2)$ time

Bentley, Jon L., and Robert Sedgewick. "Fast algorithms for sorting and searching strings." *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1997

Suffix array: sorting suffixes

Another idea: Use a sort algorithm that's aware that the items being sorted are all suffixes of the same string

Original suffix array paper suggested an $O(m \log m)$ algorithm

Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.

Other popular $O(m \log m)$ algorithms have been suggested

Larsson NJ, Sadakane K. Faster suffix sorting. Technical Report LU-CS-TR: 99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

More recently $O(m)$ algorithms have been demonstrated!

Kärkkäinen J, Sanders P. "Simple linear work suffix array construction." Automata, Languages and Programming (2003): 187-187.

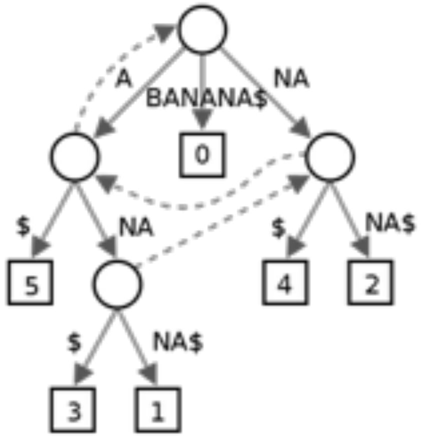
Ko P, Aluru S. "Space efficient linear time construction of suffix arrays." *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 2003.

Suffix array: summary

Suffix array gives us index that is:

(a) Just m integers, with $O(n \log m)$ worst-case query time, but close to $O(n + \log m)$ in practice

or **(b)** $3m$ integers, with $O(n + \log m)$ worst case



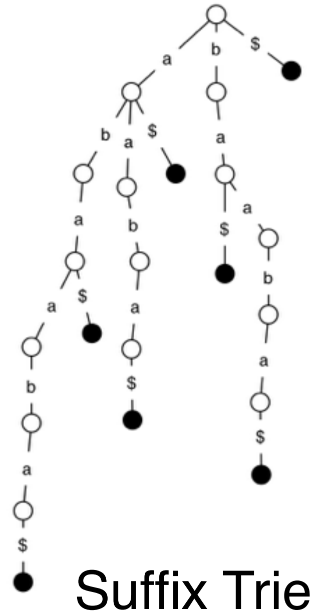
Suffix Tree

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

Suffix Array

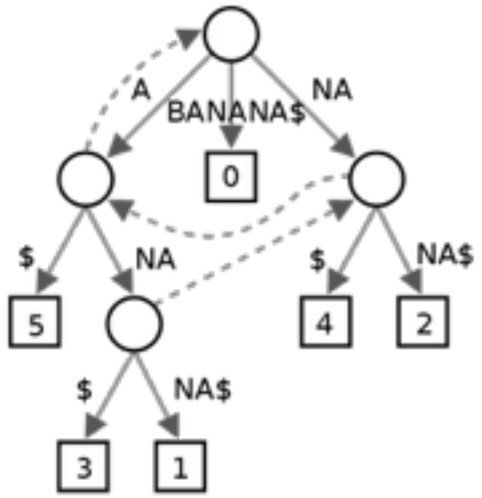
(a) will often be preferable: index for entire human genome fits in ~12 GB instead of > 45 GB

Summary



Suffix Trie

O(n) queries
O(m^2) space



Suffix Tree

O(n) queries
O(m) space

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

Suffix Array

O(n log m) queries
O(m) space

Paper discussion: MUMmer

- Short break then I’ll present the main ideas from the MUMmer paper