

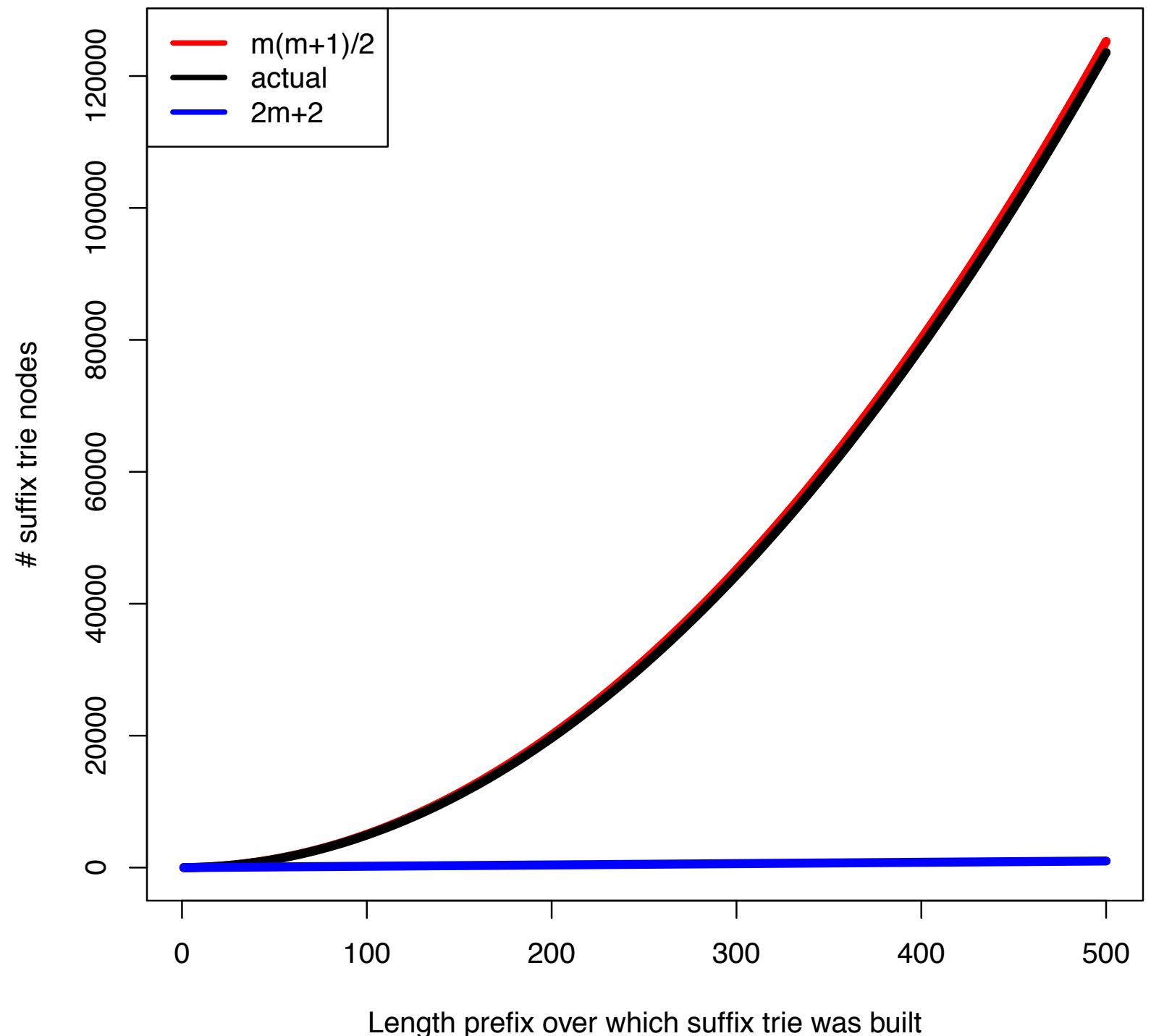
# Suffix Trees

# Suffix trie

We saw the suffix trie, but we also saw its size grows *quadratically* with the length of the string

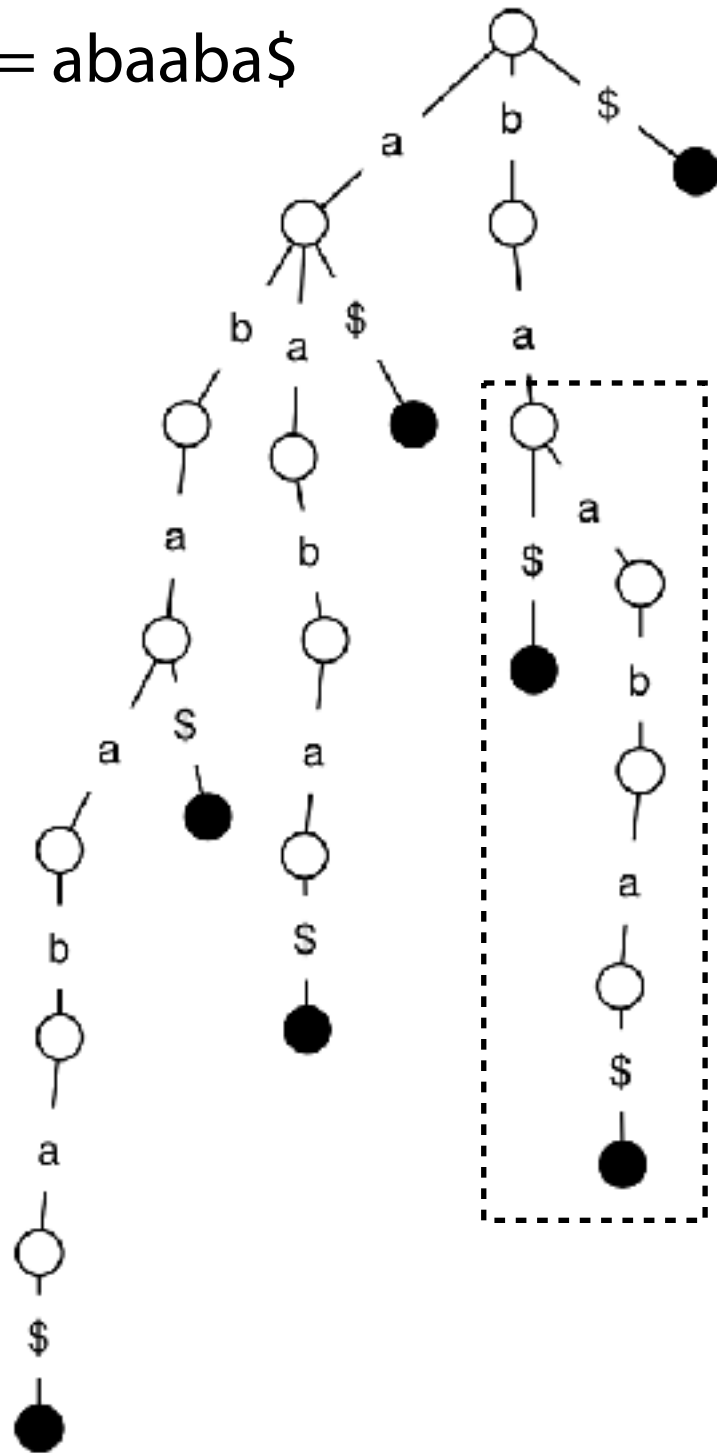
Human genome is  $3 \cdot 10^9$  bases long.

If  $m = 3 \cdot 10^9$ ,  $m^2$  is way huge, far beyond what we can store in memory



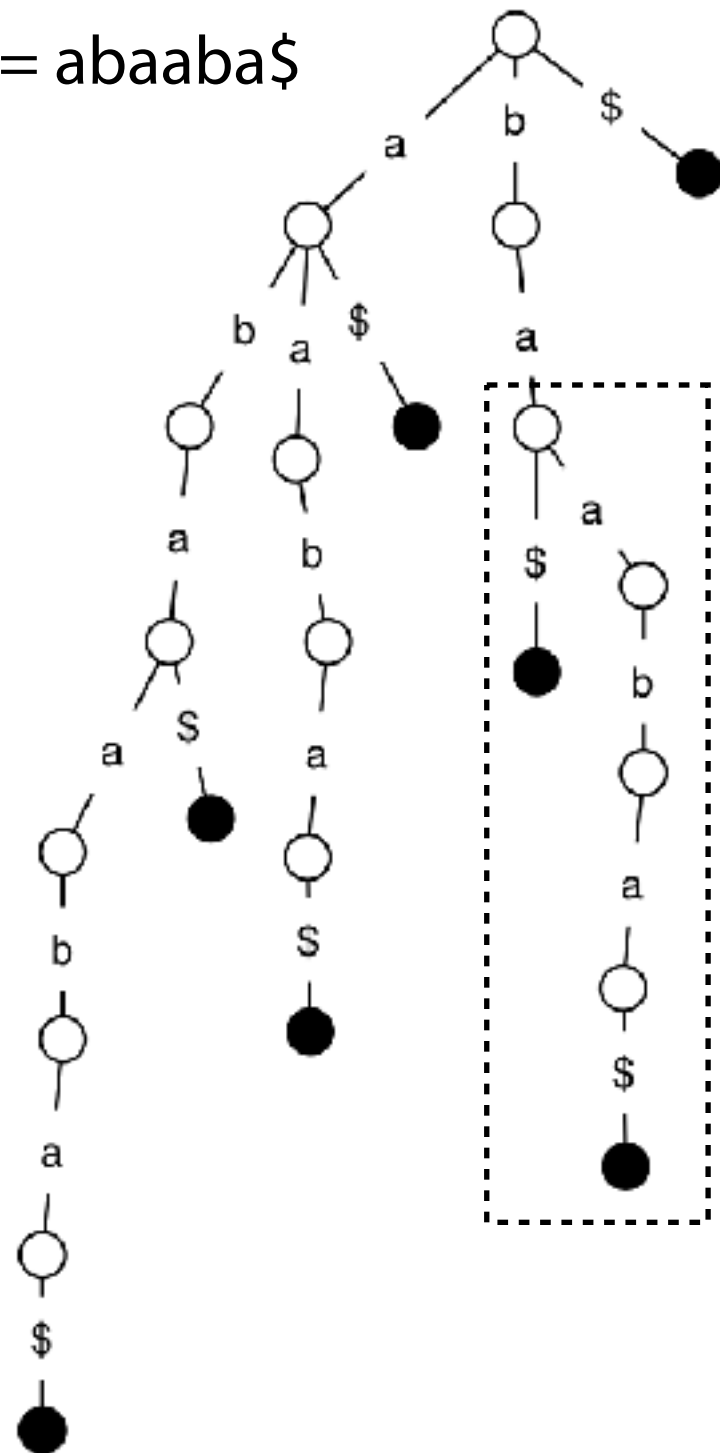
# Suffix trie: making it smaller

$T = \text{abaaba\$}$

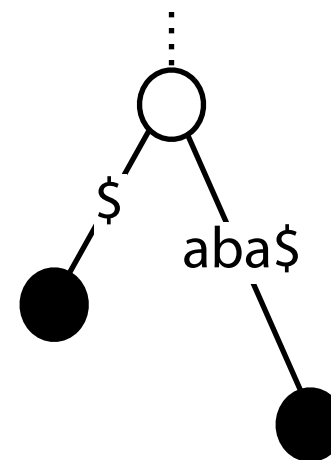


# Suffix trie: making it smaller

$T = \text{abaaba}\$$



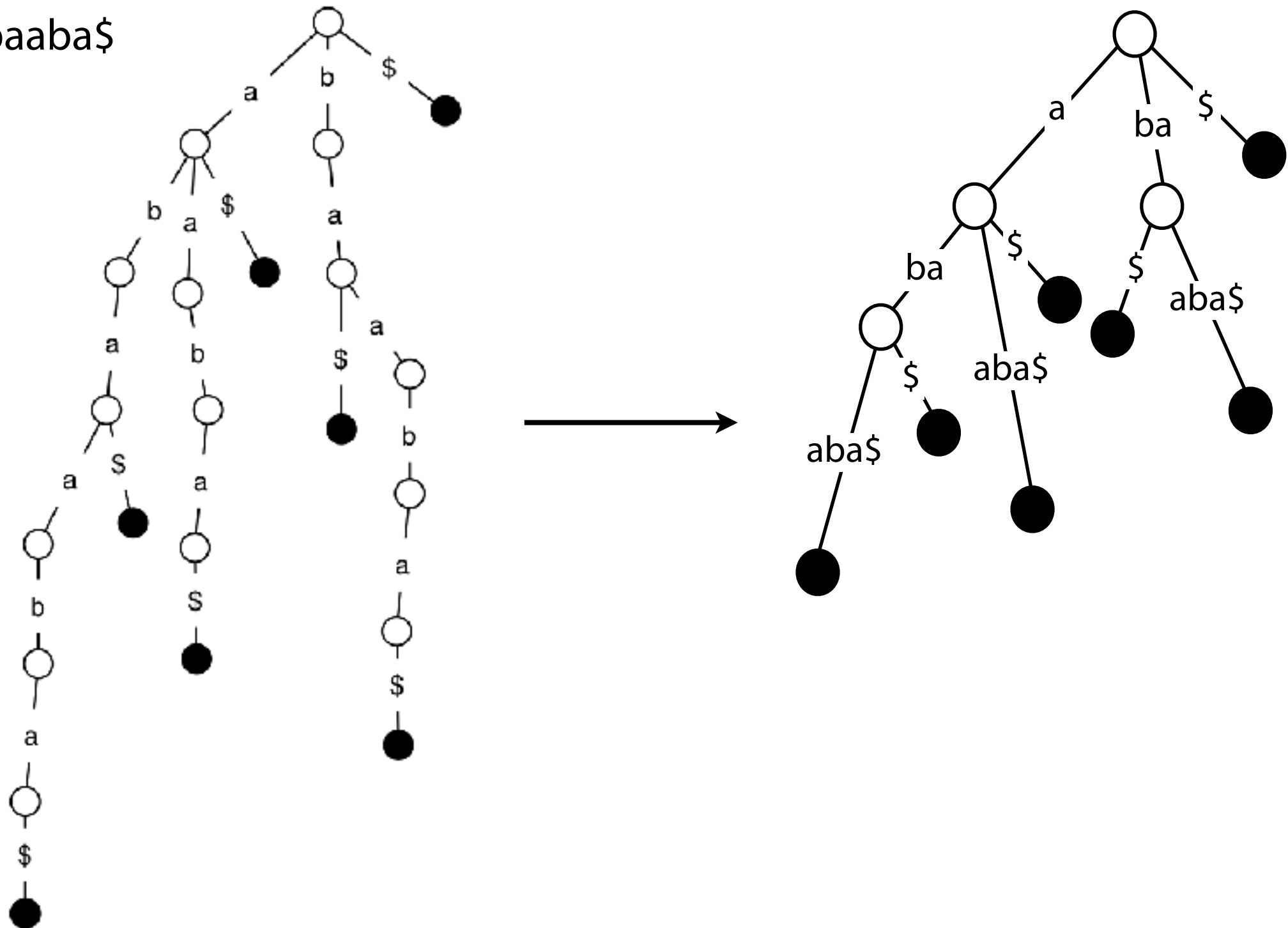
Idea 1: Coalesce non-branching paths into a *single edge* with a *string* label



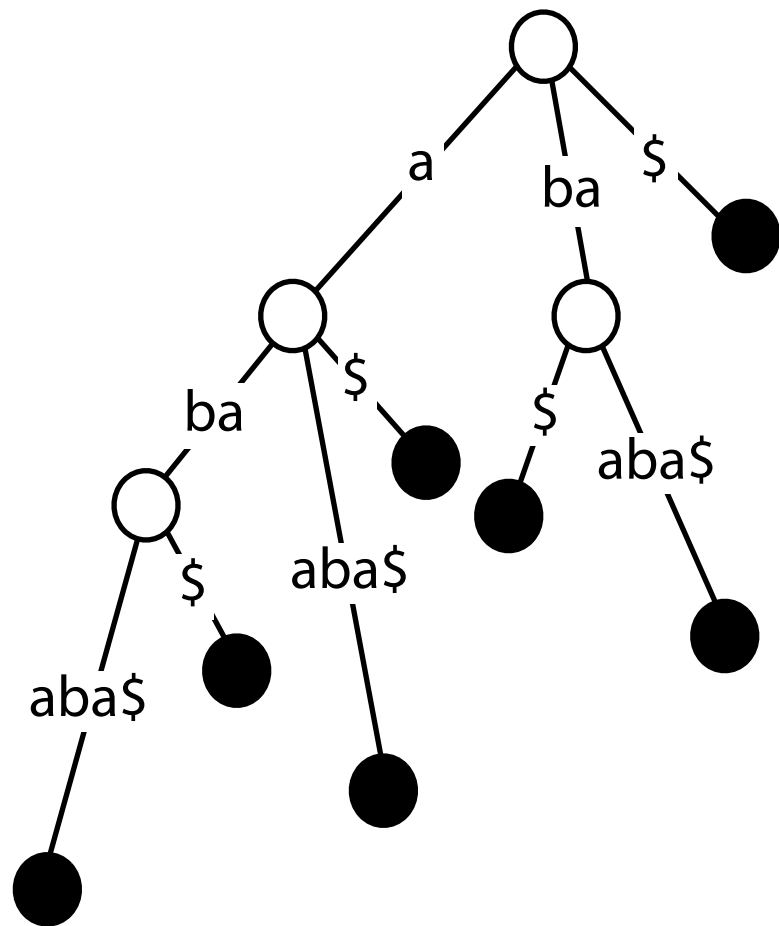
Reduces # nodes, edges,  
guarantees non-leaf nodes have  $>1$  child

# Suffix trie: making it smaller

$T = \text{abaaba\$}$



# Suffix tree

$$T = \text{abaaba\$} \quad |T| = m$$


## # leaves?

*m*

# non-leaf nodes (bound)?  $\leq m - 1$

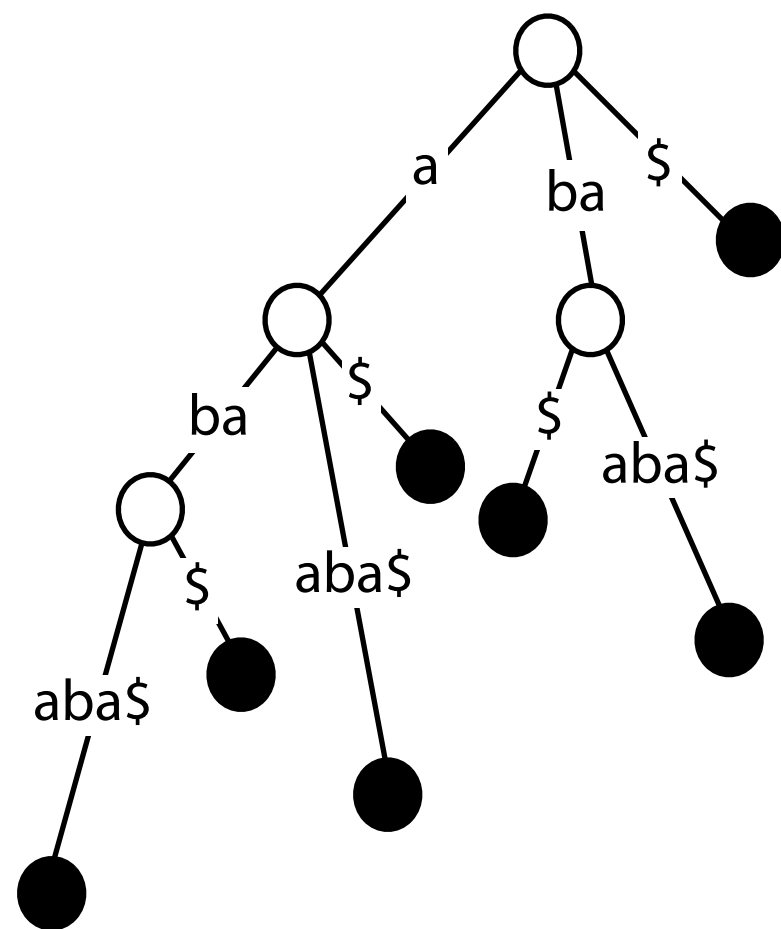
$\leq 2m - 1$  nodes total —  $O(m)$

Is *total size*  $O(m)$  now?

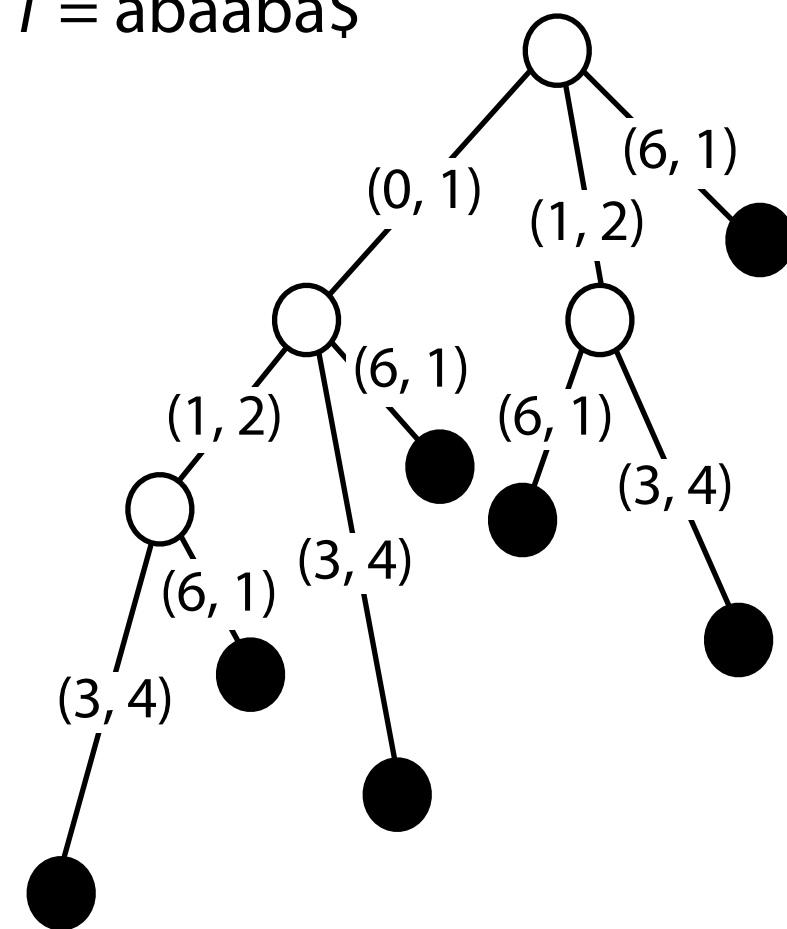
**No:** total length of edge labels grows with  $m^2$

# Suffix tree

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

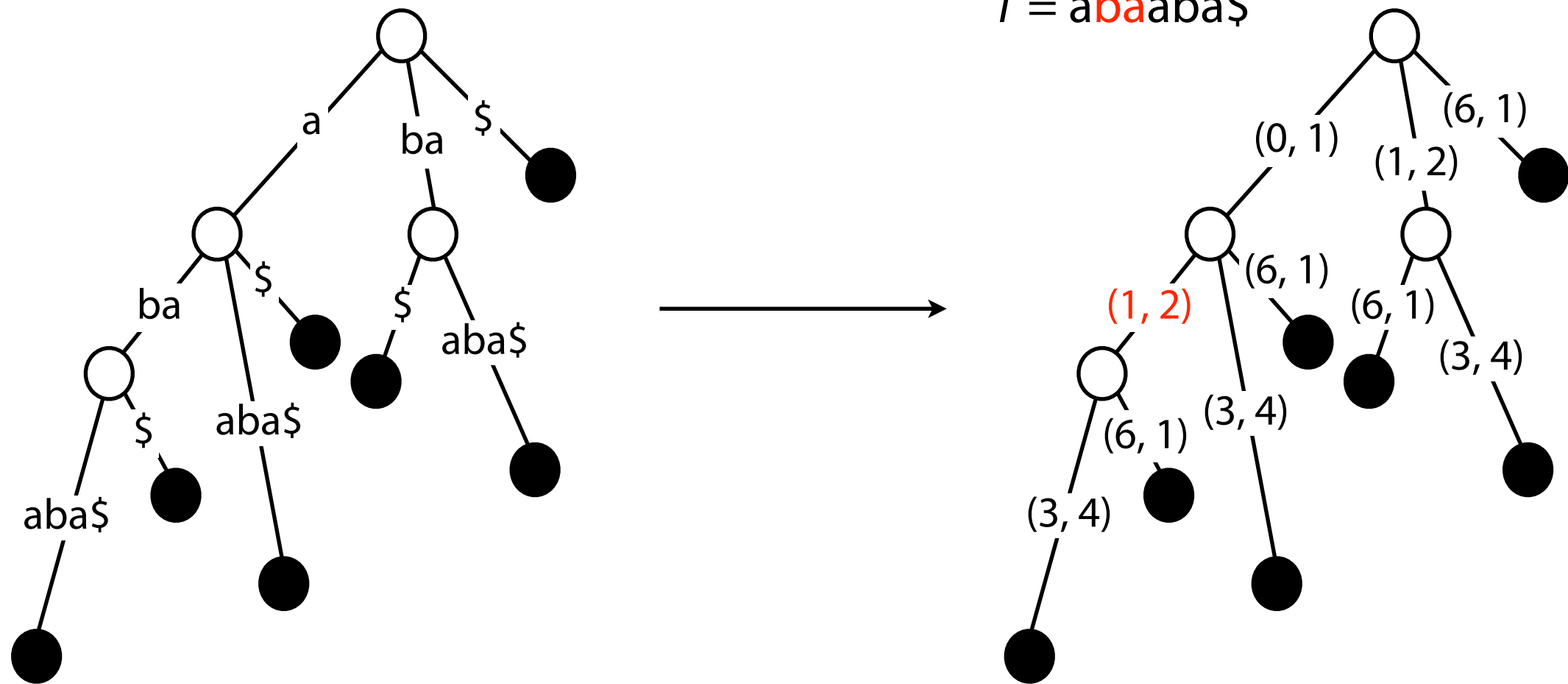


$T = \text{abaaba\$}$



# Suffix tree

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

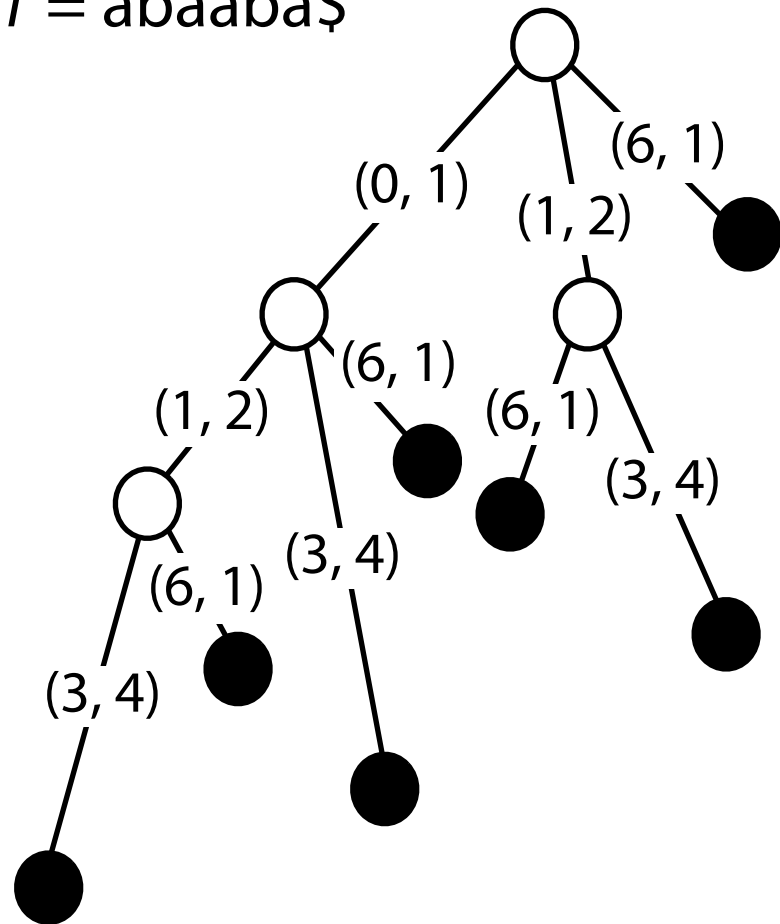


Space is now  $O(m)$     Suffix trie was  $O(m^2)$ !

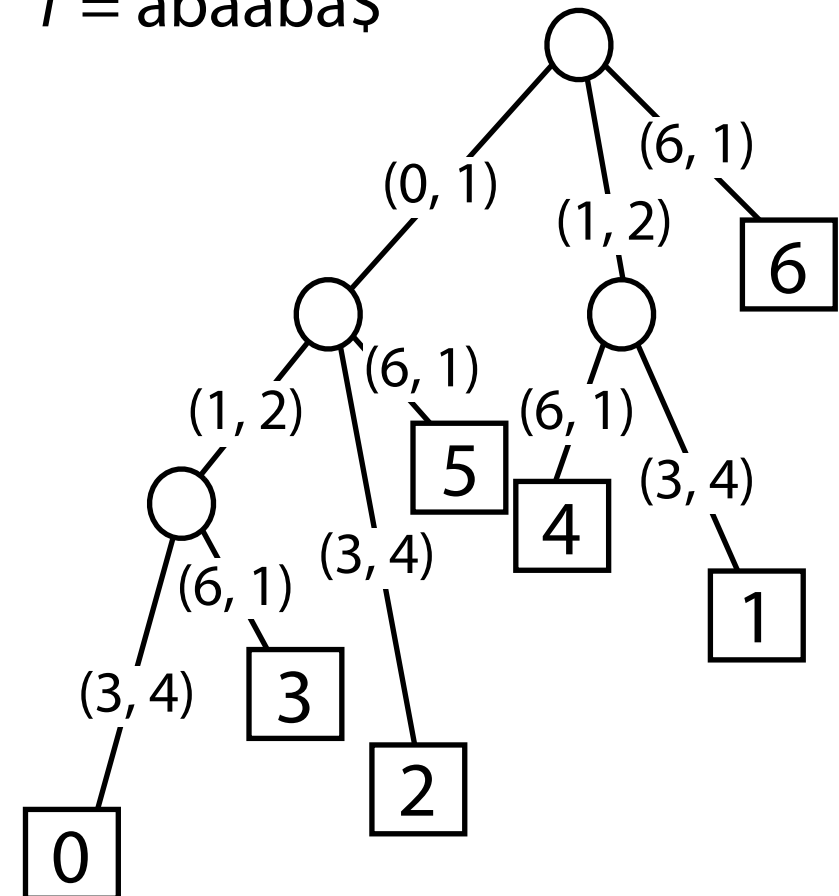


# Suffix tree: leaves hold offsets

$T = \text{abaaba}\$$

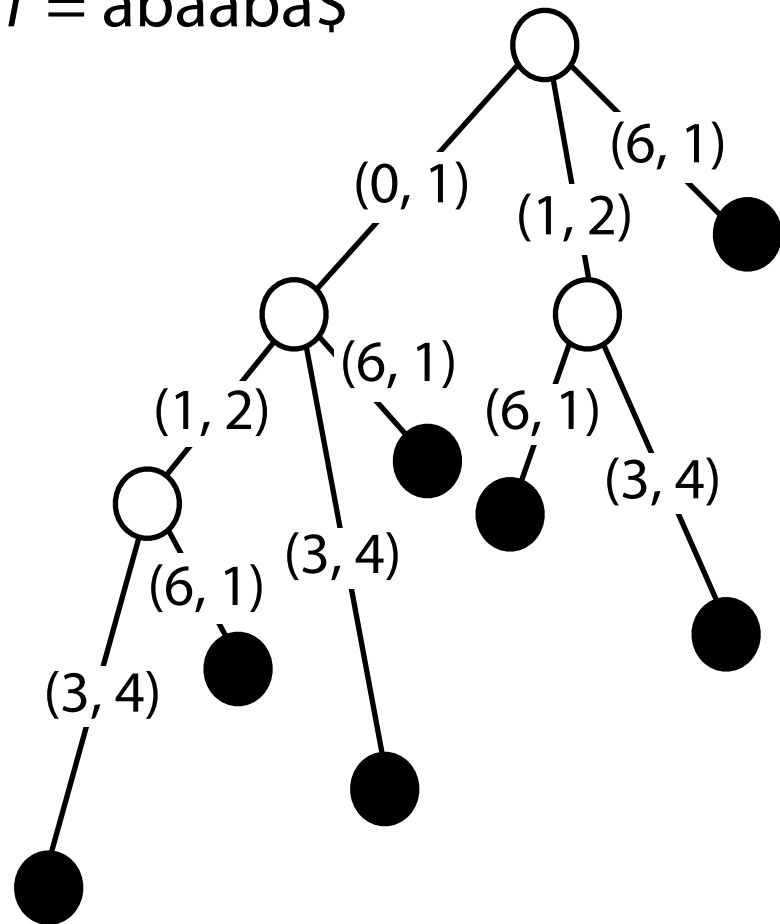


$T = \text{abaaba}\$$

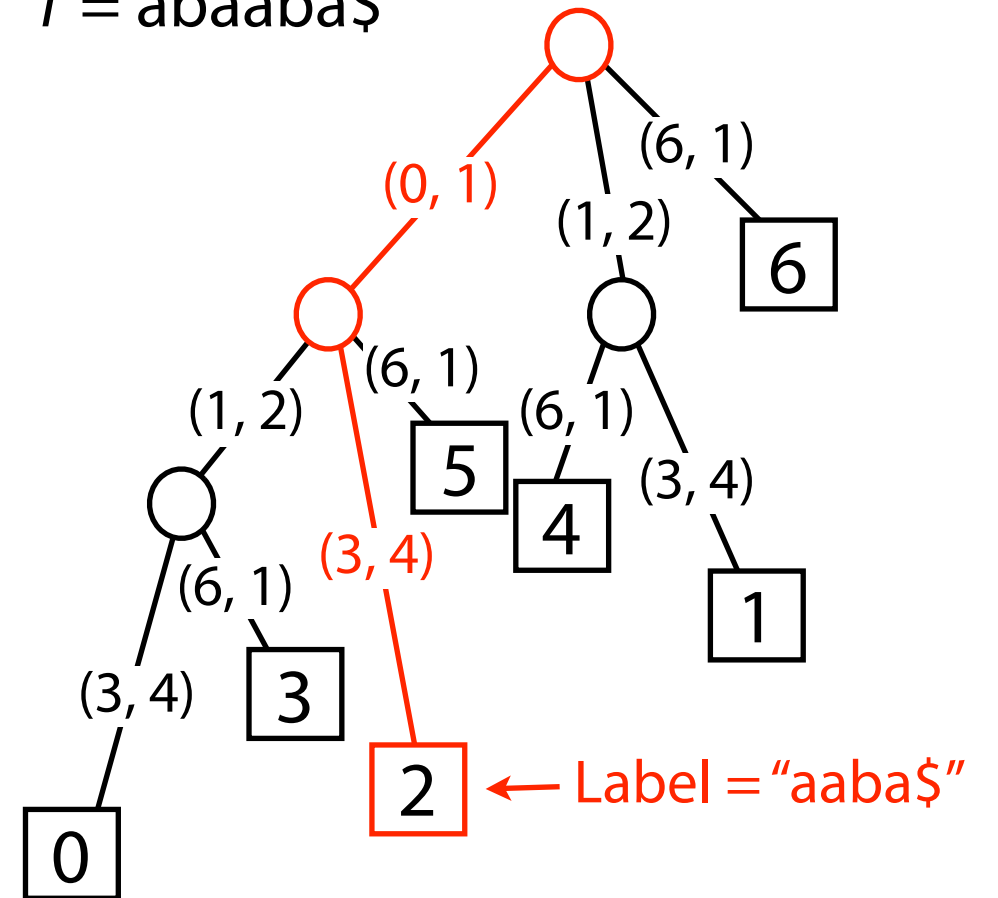


# Suffix tree: leaves hold offsets

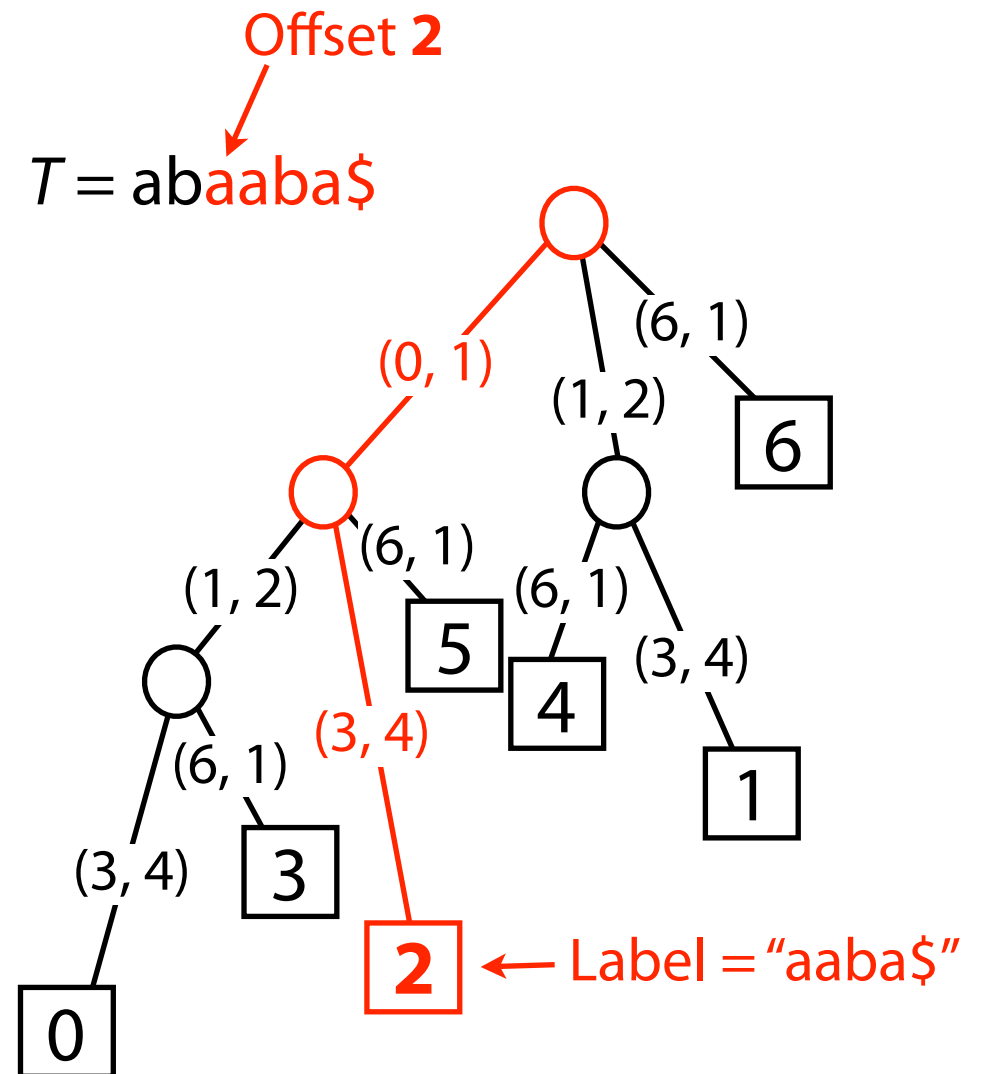
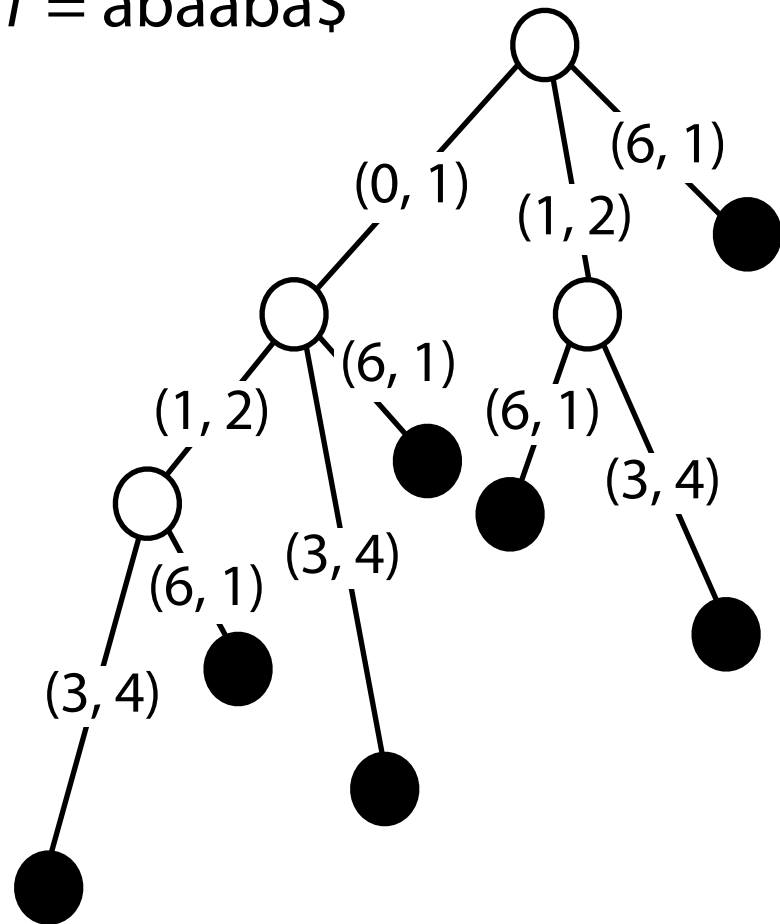
$T = \text{abaaba}\$$



$T = \text{abaaba}\$$

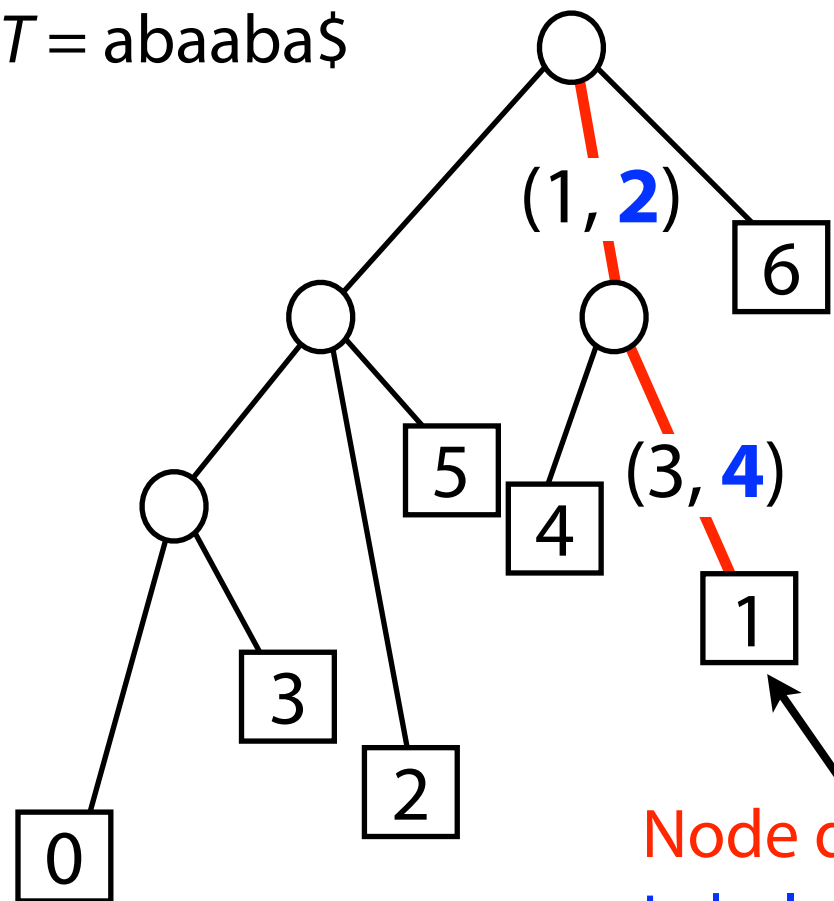


# Suffix tree: leaves hold offsets



# Suffix tree: labels

$T = \text{abaaba}\$$



Node depth = 2

Label depth = 2 + 4 = 6

Two notions of depth:

- **Node** depth: # edges from root to node
- **Label** depth: total length of edge labels from root to node

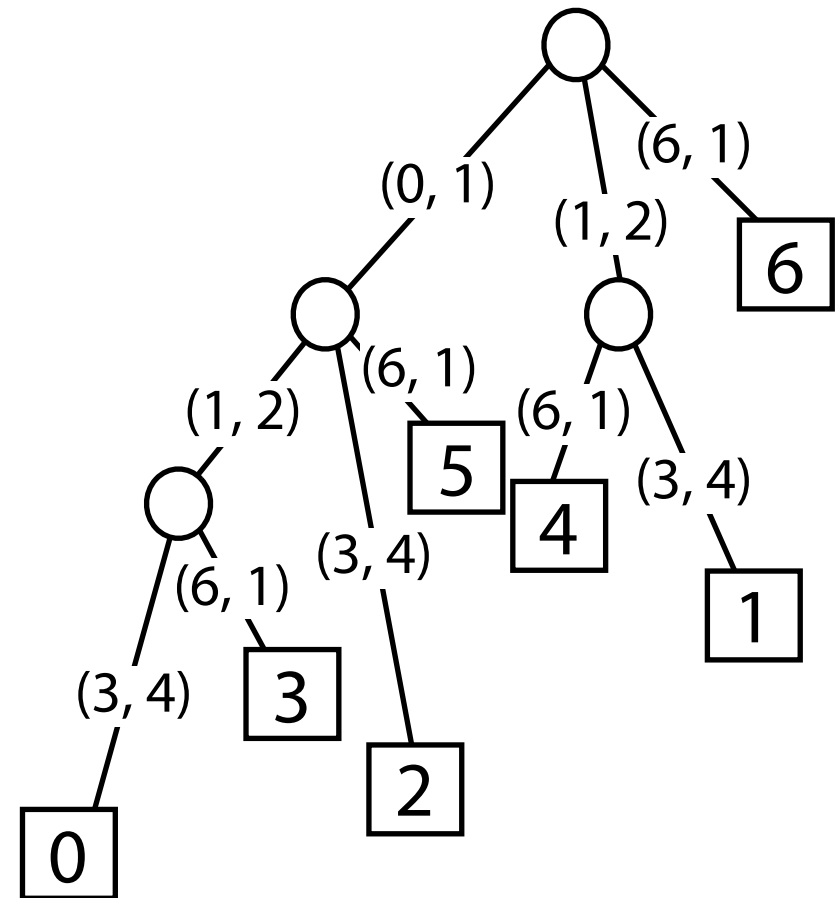
# Suffix tree: building

## Method 1: build suffix trie, coalesce non-branching paths, relabel edges

$O(m^2)$  time,  $O(m^2)$  space

Method 2: build single-edge tree  
representing longest suffix, augment to  
include the 2<sup>nd</sup>-longest, augment to  
include 3<sup>rd</sup>-longest, etc (Gusfield 5.4)

$O(m^2)$  time,  $O(m)$  space



# Suffix tree: implementation

[See suffix-tree notebook](#)

# Suffix tree: building

Canonical method: Ukkonen's algorithm

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

$O(m)$  time and space!

Won't cover it in class; see Gusfield Ch. 6 for details

## On-Line Construction of Suffix Trees<sup>1</sup>

E. Ukkonen<sup>2</sup>

**Abstract.** An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix *tries*. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWG).

**Key Words.** Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

Canonical algorithm for  $O(m)$  time & space suffix tree construction

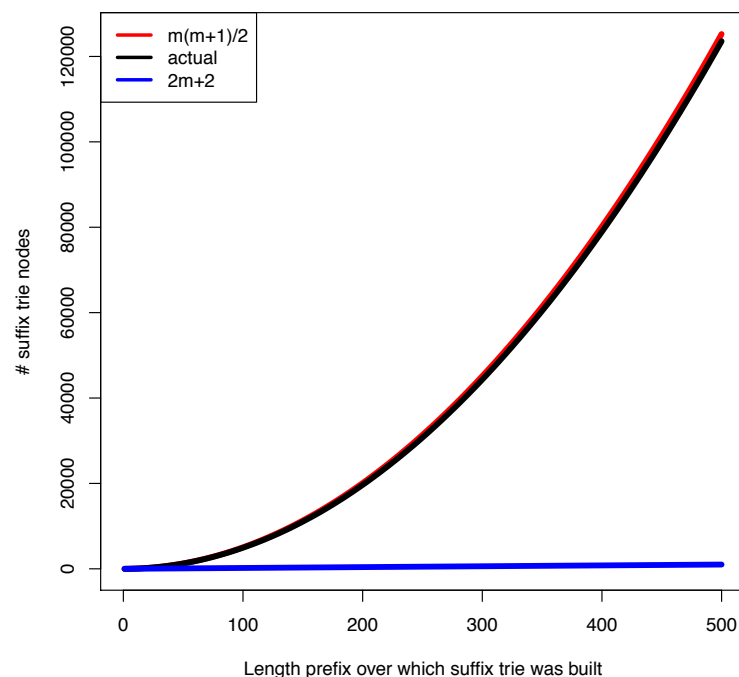


# Suffix tree: actual growth

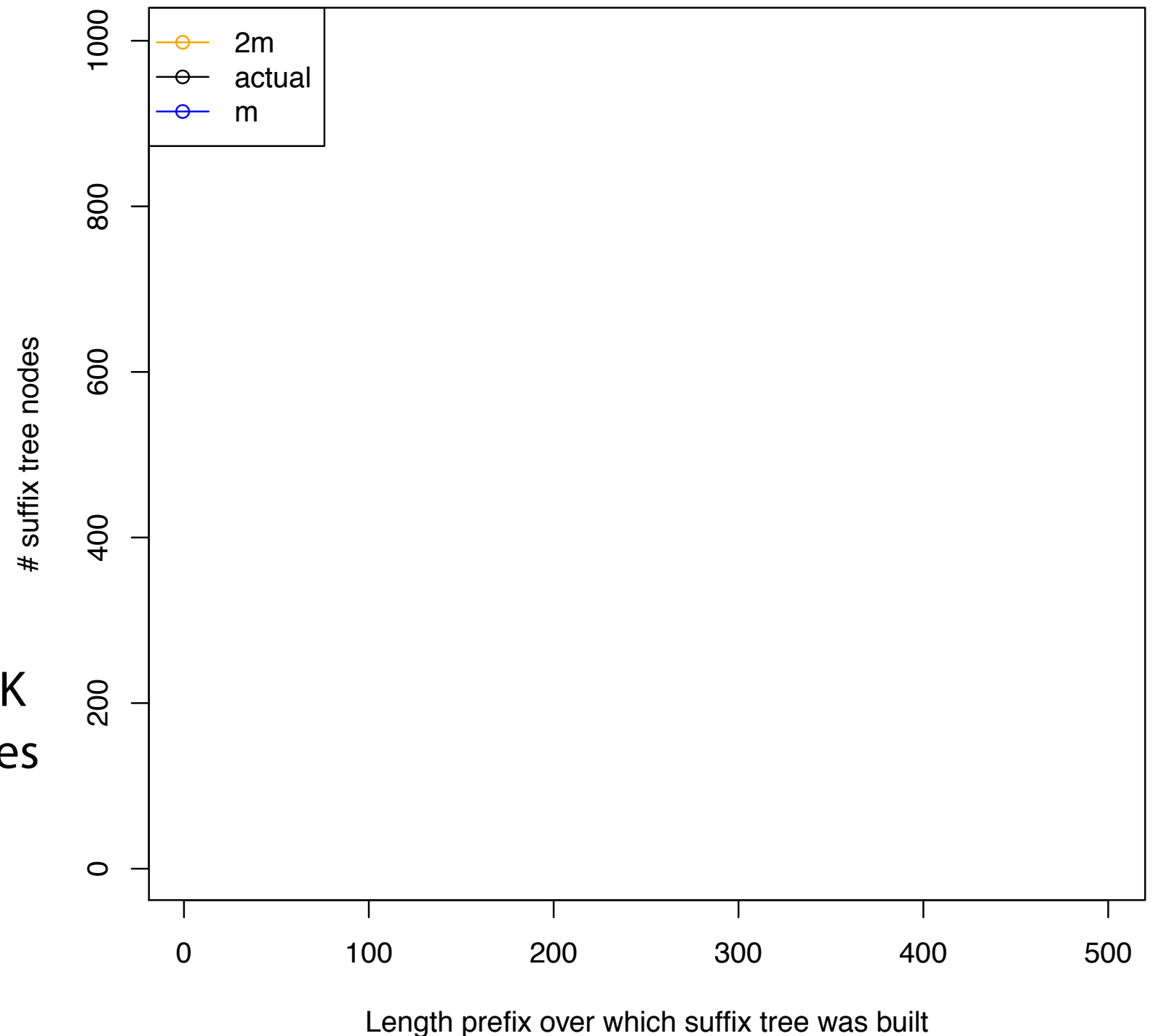
Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Remember suffix trie plot:



123 K  
nodes

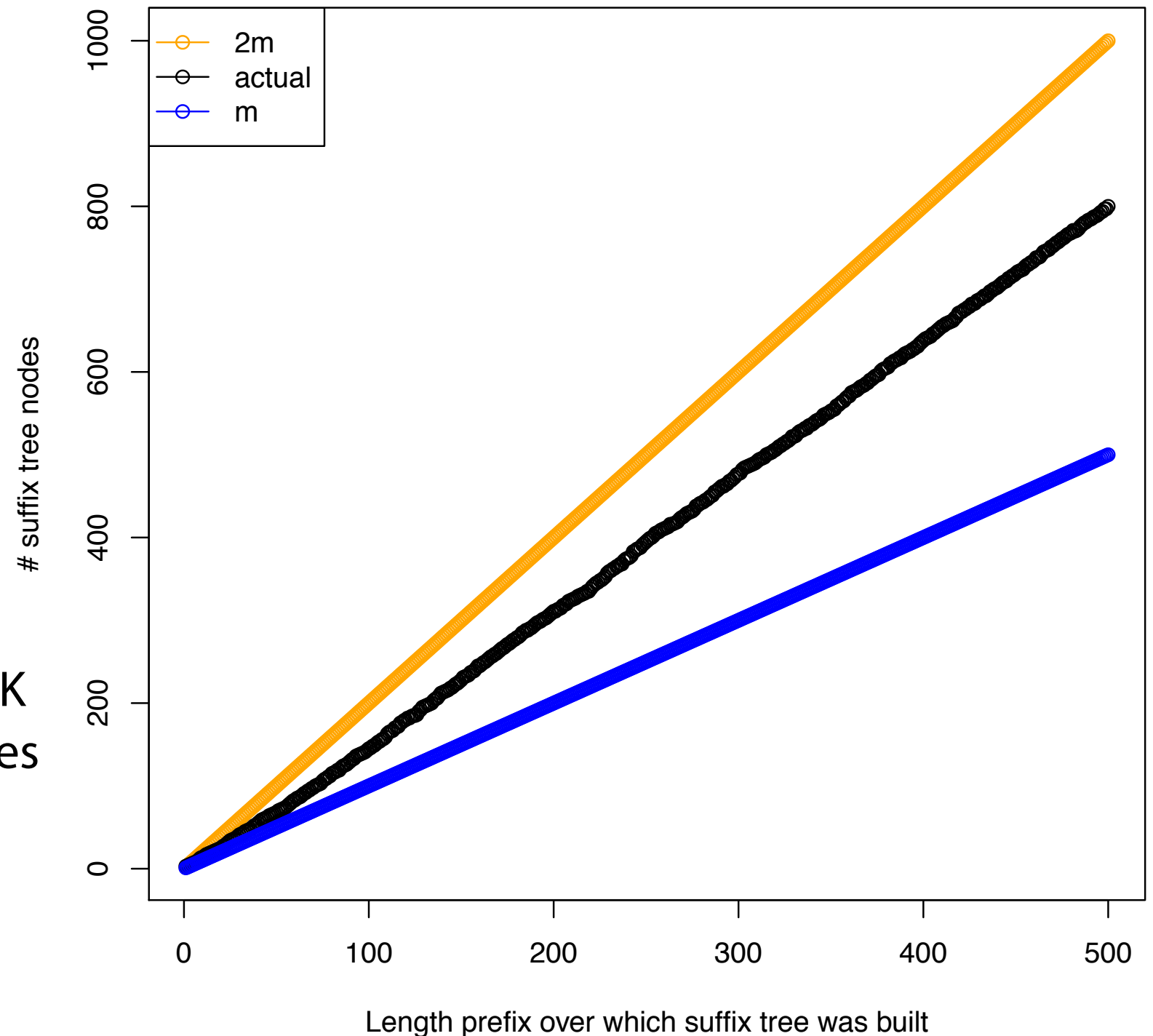
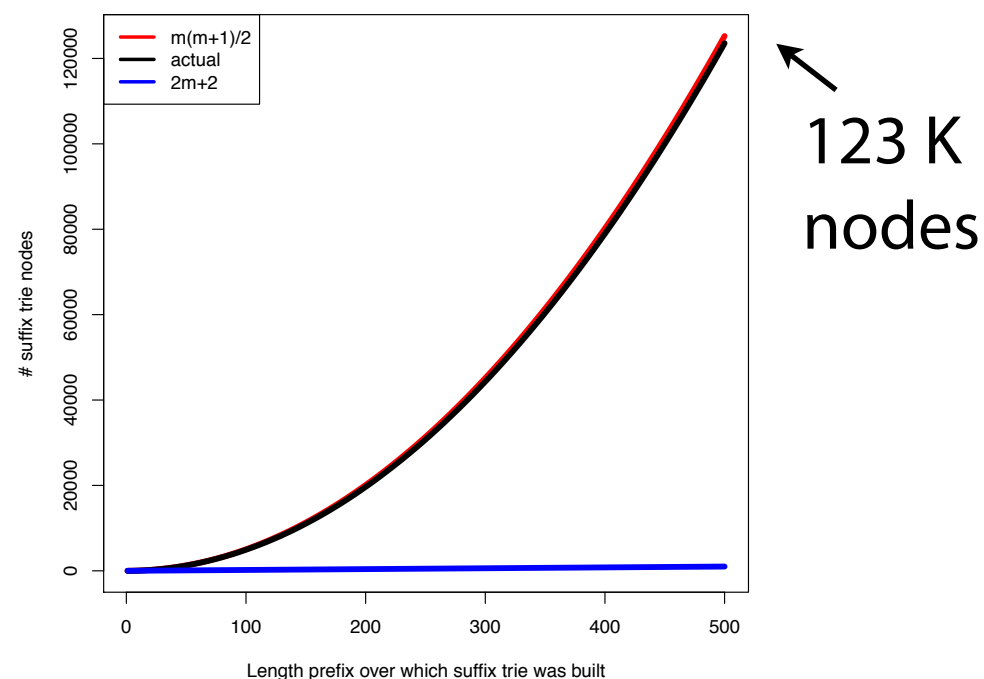


# Suffix tree: actual growth

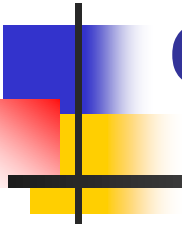
Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Remember suffix trie plot:



# Linear time algorithm for constructing suffix tree



---



# Straightforward construction of suffix tree

---

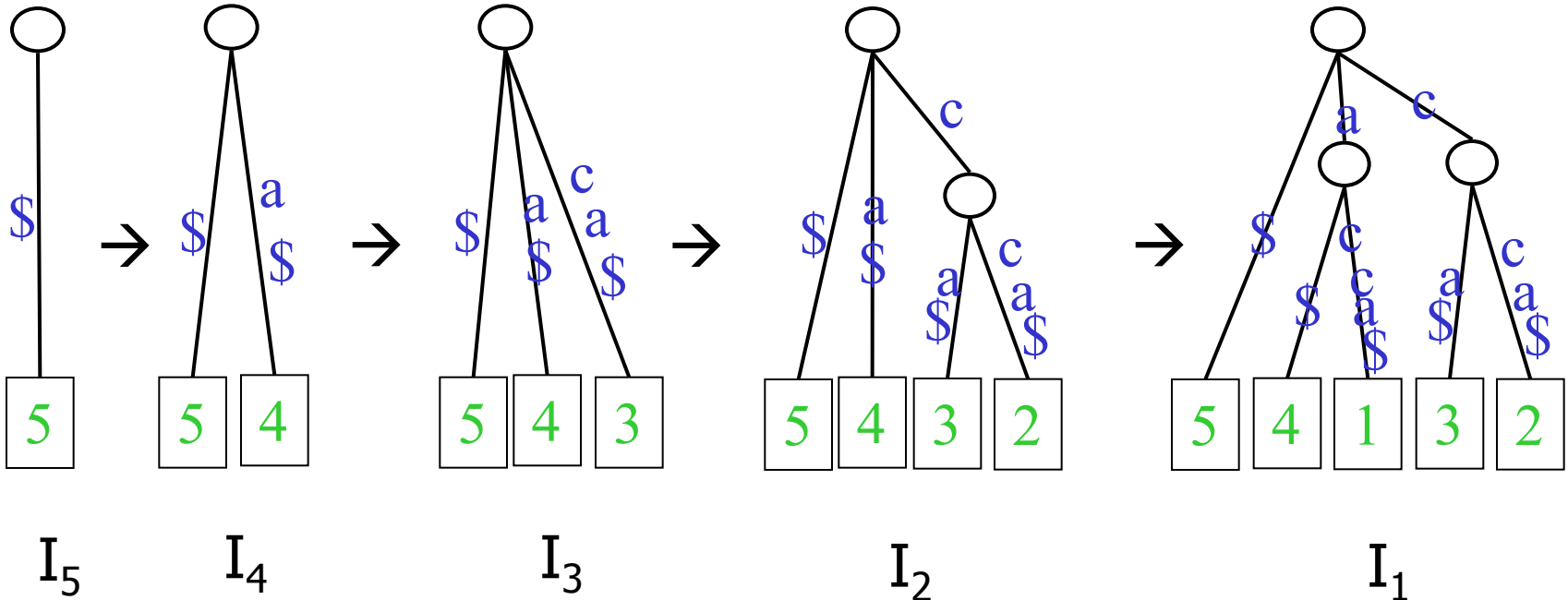
- Consider  $S = s_1s_2\dots s_n$  where  $s_n = \$$
- Algorithm:
  - Initialize the tree with only a root
  - For  $i = n$  to 1
    - Includes  $S[i..n]$  into the tree
- Time:  $O(n^2)$

# Example of construction

■  $S = \text{acca}\$$

Init

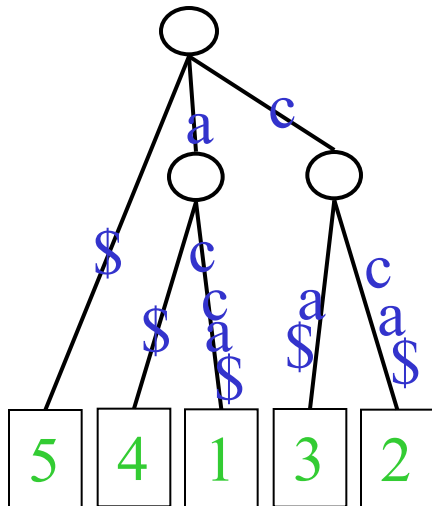
For-loop



# Construction of generalized suffix tree

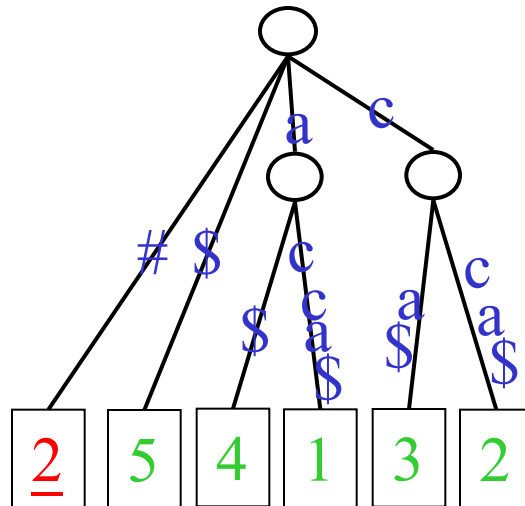
■  $S' = c\#$

Init

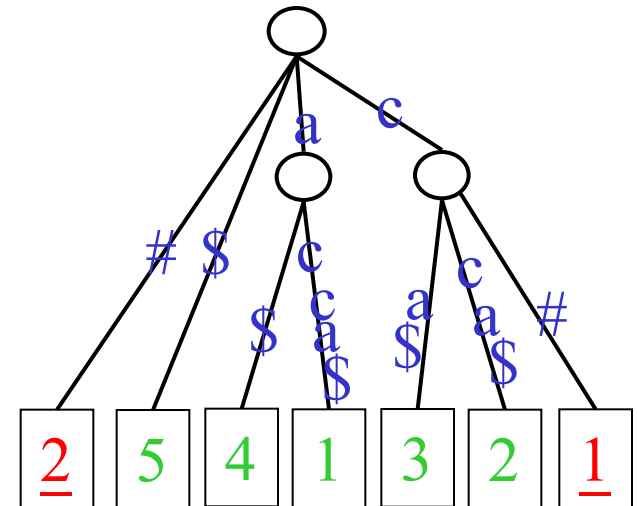


$I_1$

For-loop



$J_2$



$J_1$



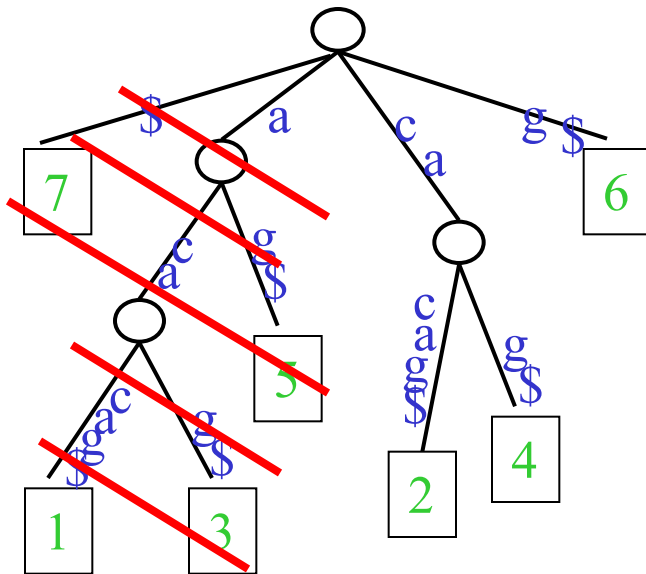
# Can we construct a suffix tree in $o(n^2)$ time?

---

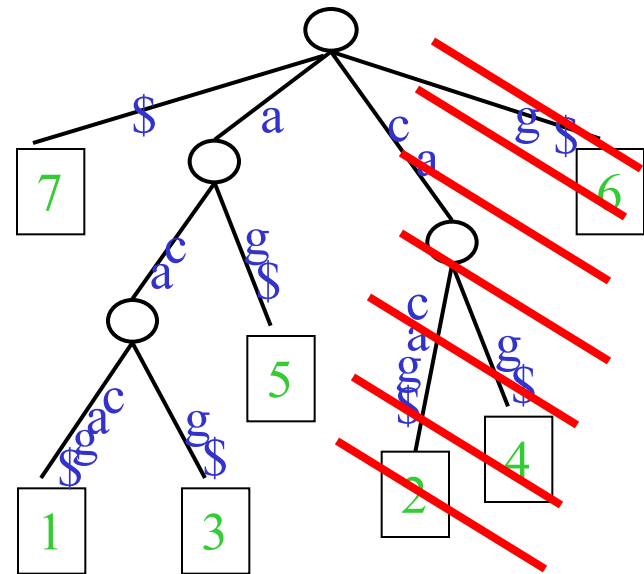
- Yes. We can construct it in  $O(n)$  time.
- Weiner's algorithm [1973]
  - Linear time for constant size alphabet, but much space
- McCreight's algorithm [JACM 1976]
  - Linear time for constant size alphabet, quadratic space
- Ukkonen's algorithm [Algorithmica, 1995]
  - Online algorithm, linear time for constant size alphabet, less space
- Farach's algorithm [FOCS 1997]
  - Linear time for general alphabet
- Hon, Sadakane, and Sung's algorithm [FOCS 2003]
  - $O(n)$  bit space  $O(n \log^n)$  time for  $0 < \epsilon < 1$
  - $O(n)$  bit space  $O(n)$  time for suffix array construction
- We will discuss Farach's algorithm later.

# Idea

- Build Odd Suffix Tree and Even Suffix Tree
- Then, merge odd and even suffix tree.



Even Suffix Tree



Odd Suffix Tree





# Idea

---

- Input: a string  $S$  of length  $n$ 
  1. Recursively compute the suffix tree  $T_o$  of all suffixes beginning at the odd positions.
    - $T_o$  is of size  $n/2$ .
  2. From  $T_o$ , compute  $T_e$  which is the suffix tree for all suffixes beginning at the even positions.
  3. Merge  $T_o$  and  $T_e$  to form the suffix tree for  $S$ .



# Stage 1: Constructing odd suffix tree

---

- Given a string  $S[1..n]$ , we generate a new string  $S'[1..n/2]$  as follows.
  - we map pairs of characters into single characters as follows:
    - $S[1..2], S[3..4], S[5..6], \dots, S[n-1..n]$ .
  - Remove the duplicates from the pairs of characters and sort them by radix sort.
  - $S'[i] = \text{rank of } S[2i-1..2i] \text{ in the sorted list, for } i=1, 2, \dots, n/2.$
- By recursion, we get the suffix tree  $T'$  for  $S'$
- Convert  $T'$  to the odd suffix tree  $T_o$ .



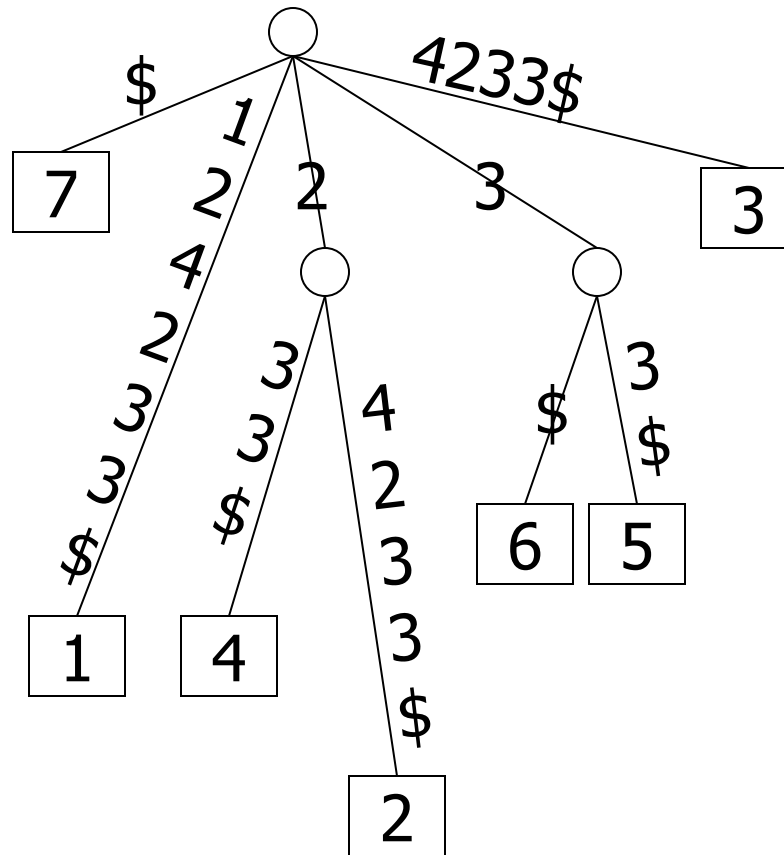
## Example (I)

---

- $S = \text{aaabbbabbaba\$}$
- $S[1..2]=\text{aa}$ ,  $S[3..4]=\text{ab}$ ,  $S[5..6]=\text{bb}$ ,  
 $S[7..8]=\text{ab}$ ,  $S[9..10]=\text{ba}$ ,  $S[11..12]=\text{ba}$ .
- By stable sort,  $\text{aa} < \text{ab} < \text{ba} < \text{bb}$ .
  - $\text{Rank}(\text{aa})=1$ ,  $\text{Rank}(\text{ab})=2$ ,  $\text{Rank}(\text{ba})=3$ ,  
 $\text{Rank}(\text{bb})=4$ .
- So,  $S' = 124233\$$ .

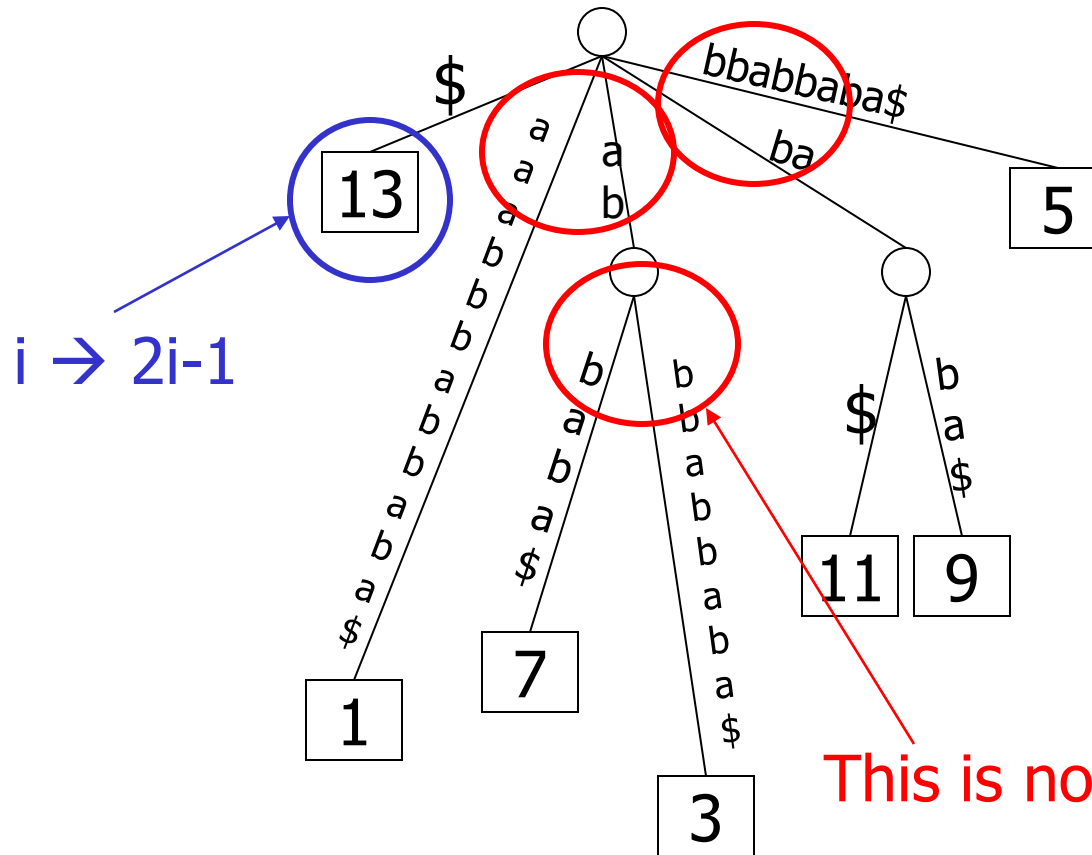
## Example (II)

- By recursion, construct the suffix tree  $T'$  for  $S'$ :



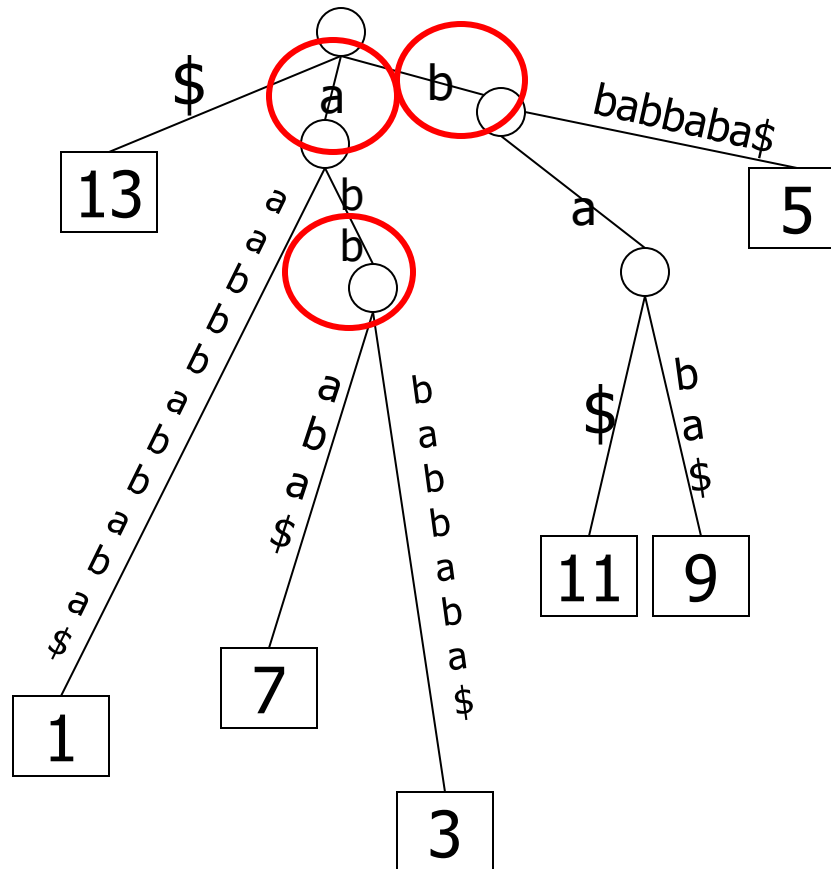
# Example (III)

- Convert  $T'$  to the odd tree:



# Example (IV)

- Refine the odd tree  $T_o$ :





# Time complexity for building the odd tree

---

- Let  $\text{Time}(n)$  be the time to build a suffix tree for a string of length  $n$ .
- Stable sorting and refinement of the odd trees take  $O(n)$  time.
- Build suffix tree for  $S'$  takes  $\text{Time}(n/2)$ .
- So, Stage 1 takes  $\text{Time}(n/2) + O(n)$  time.



## Stage 2: Build the even tree

---

1. Generate the lex-ordering of the leaves in  $T_o$ .
2. For any two adjacent leaves  $2i$  and  $2j$ , we find  $\text{lcp}(2i, 2j)$ .
3. Construct the even tree  $T_e$  from left to right (according to the lex-ordering).





## Build the even tree (Step 1)

---

- We get the lex-ordering of the leaves in  $T_o$ .
- Generate the lex-ordering of the leaves in  $T_o$ .
  - For each leaf  $i$  in  $T_o$ , get the preceding character  $c=S[i-1]$  and form a pair  $(c,i)$ . Each pair represents a even suffix  $i-1$ .
  - Perform stable sorting on those pairs. We get the lex-ordering of the leaves in  $T_e$ .



# Example

$S = \text{aaabbbabbaba\$}$

- Lex-ordering of the leaves in  $T_o$ :
  - $13 < 1 < 7 < 3 < 11 < 9 < 5$
- The pairs are:
  - $(a, 13), (\$, 1), (b, 7), (a, 3), (a, 11), (b, 9), (b, 5).$
- After stable sorting, we have
  - $(\$, 1), (a, 13), (a, 3), (a, 11), (b, 7), (b, 9), (b, 5).$
- Hence, the lex-ordering of the leaves of  $T_e$ :
  - $12 < 2 < 10 < 6 < 8 < 4$



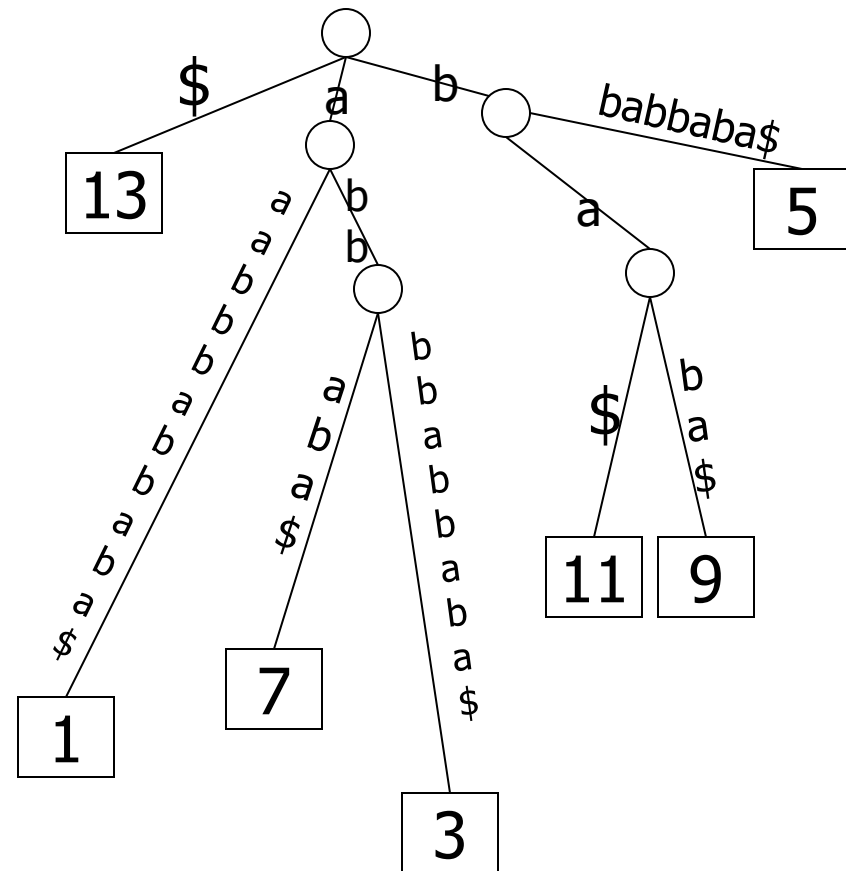
## Build the even tree (Step 2)

---

- For any two adjacent leaves  $2i$  and  $2j$ , we first find  $\text{lcp}(2i, 2j)$ .
- Observation:  $\text{lcp}(2i, 2j) =$ 
  - $\text{lcp}(2i+1, 2j+1)+1$  if  $S[2i]=S[2j]$
  - 0 otherwise
- Proof:
  - If  $S[2i] \neq S[2j]$ ,  $\text{lcp}(2i, 2j)=0$ .
  - Otherwise,  $\text{lcp}(2i, 2j)=1+\text{lcp}(2i+1, 2j+1)$ .

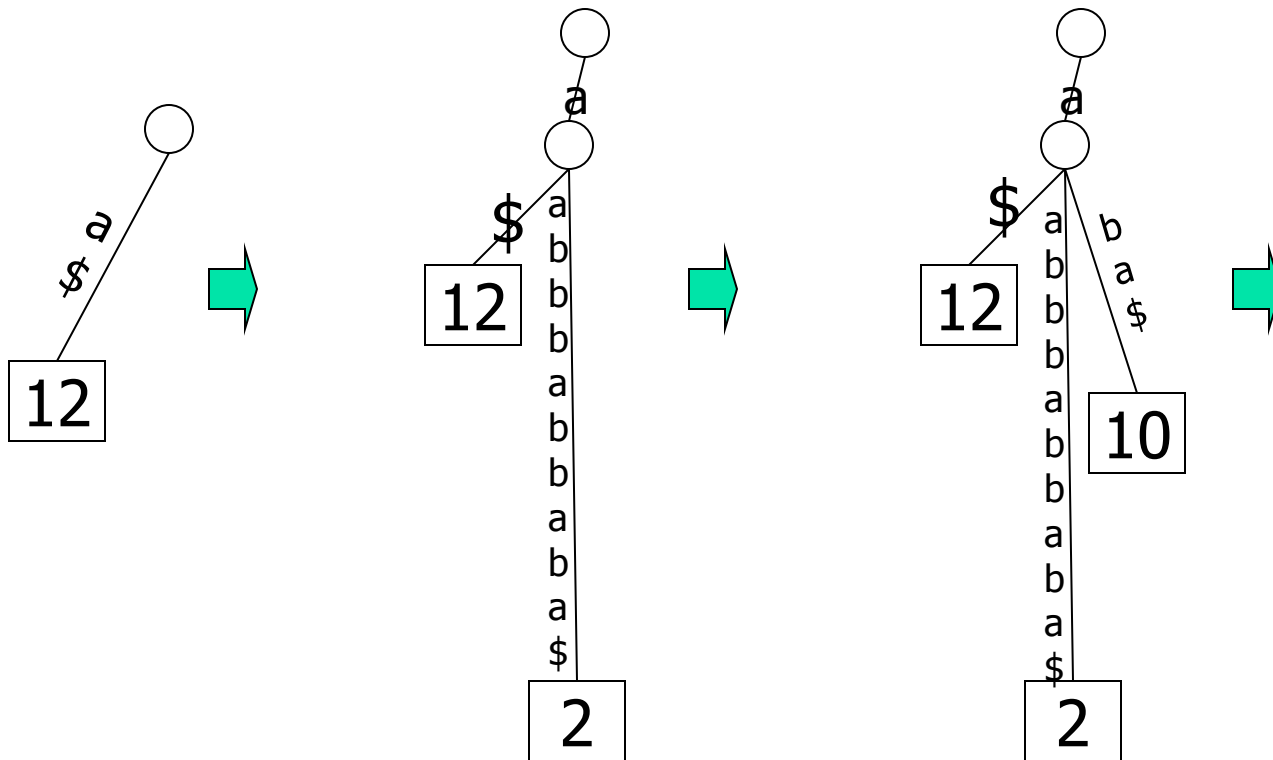
# Example

- Recall that the lex-ordering of leaves:
  - $12 < 2 < 10 < 6 < 8 < 4$ .
- By the previous observation, we have
  - $\text{lcp}(8,4)=\text{lcp}(9,5)+1=2$
- Similarly, we have
  - $\text{lcp}(12,2)=1, \text{lcp}(2,10)=1,$   
 $\text{lcp}(10,6)=0, \text{lcp}(6,8)=1,$   
 $\text{lcp}(8,4)=2$

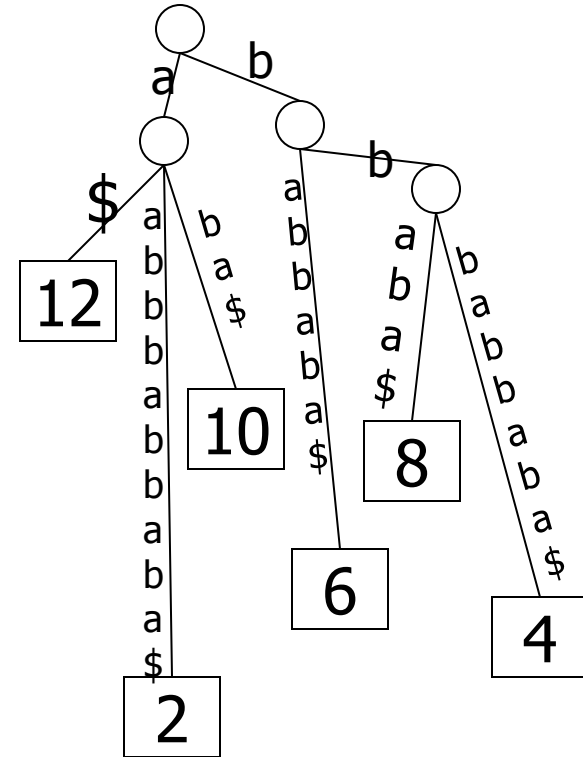
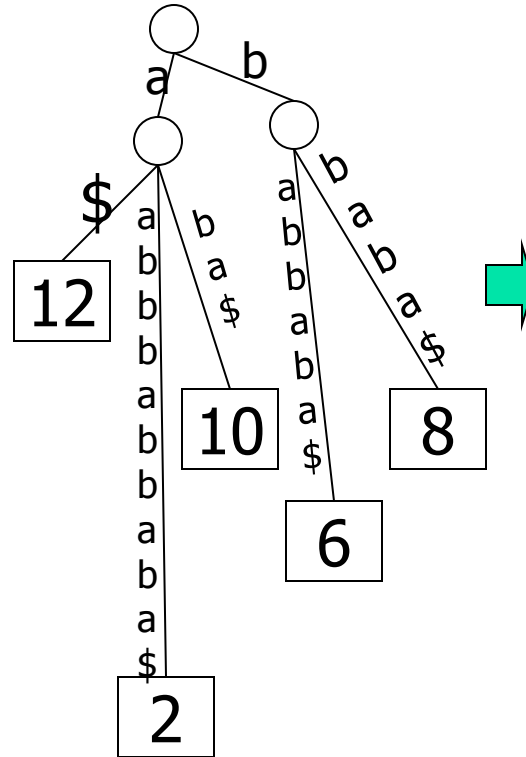
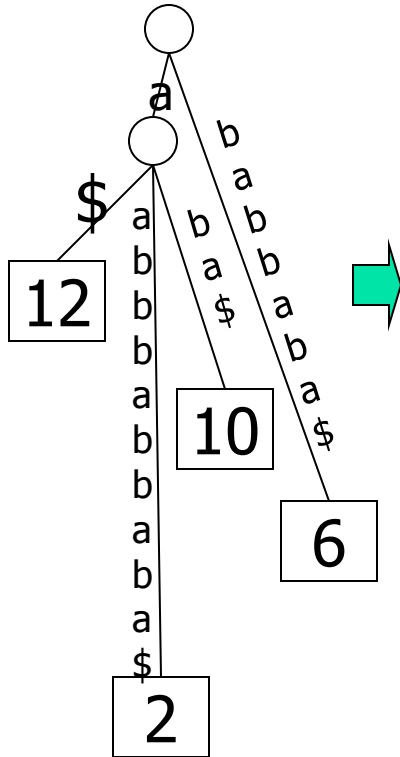


# Build the even tree (Step 3)

- Construct the even tree  $T_e$  from left to right.



# Build the even tree (Step 3)





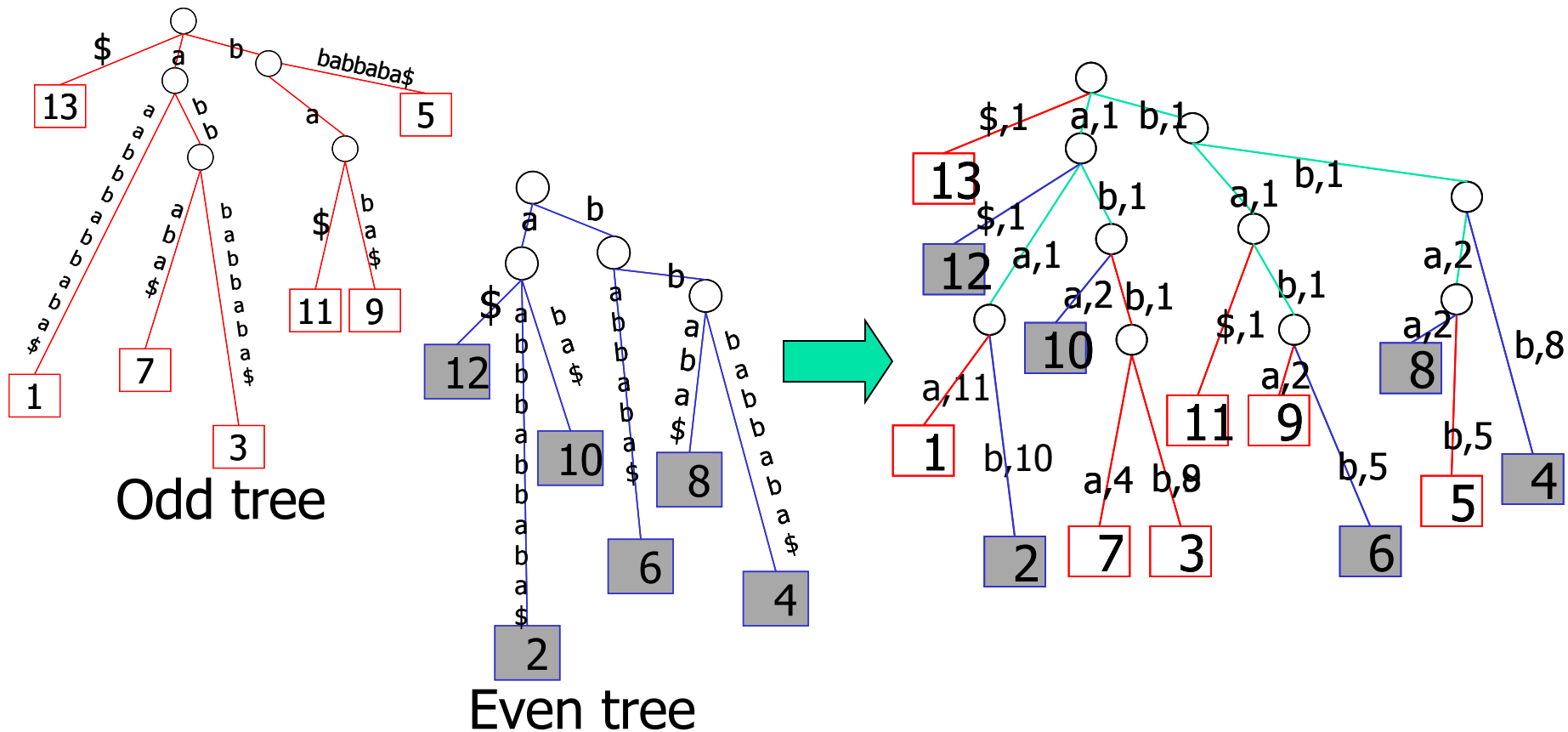
# Time complexity for building the even tree

---

- Step 1:  $O(n)$  time
- Step 2:  $O(n)$  time
- Step 3:  $O(n)$  time

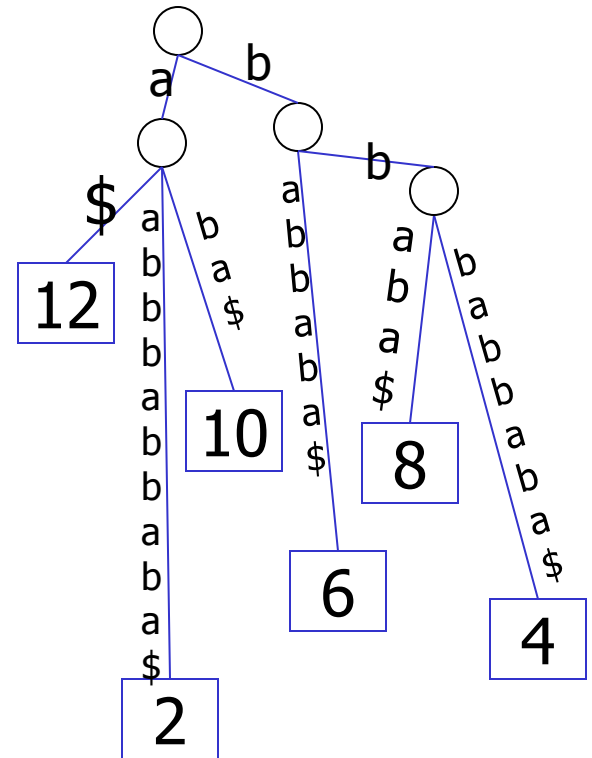
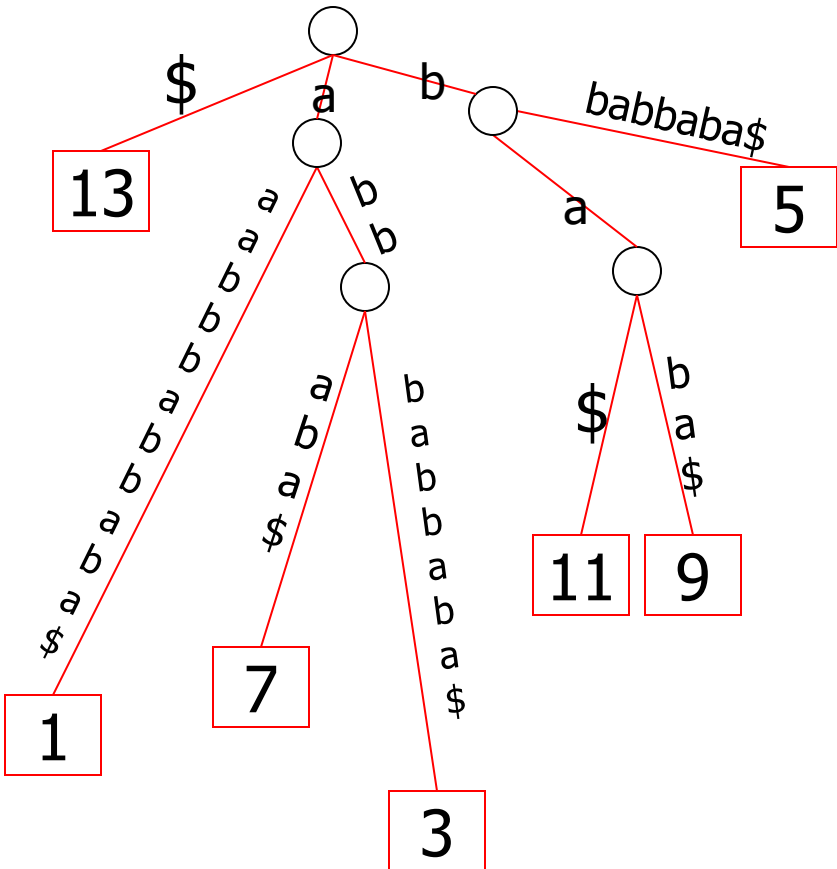
# Stage 3: Merge odd and even trees

- We can merge  $T_o$  and  $T_e$  by DFS. However, it takes  $O(n^2)$  time.



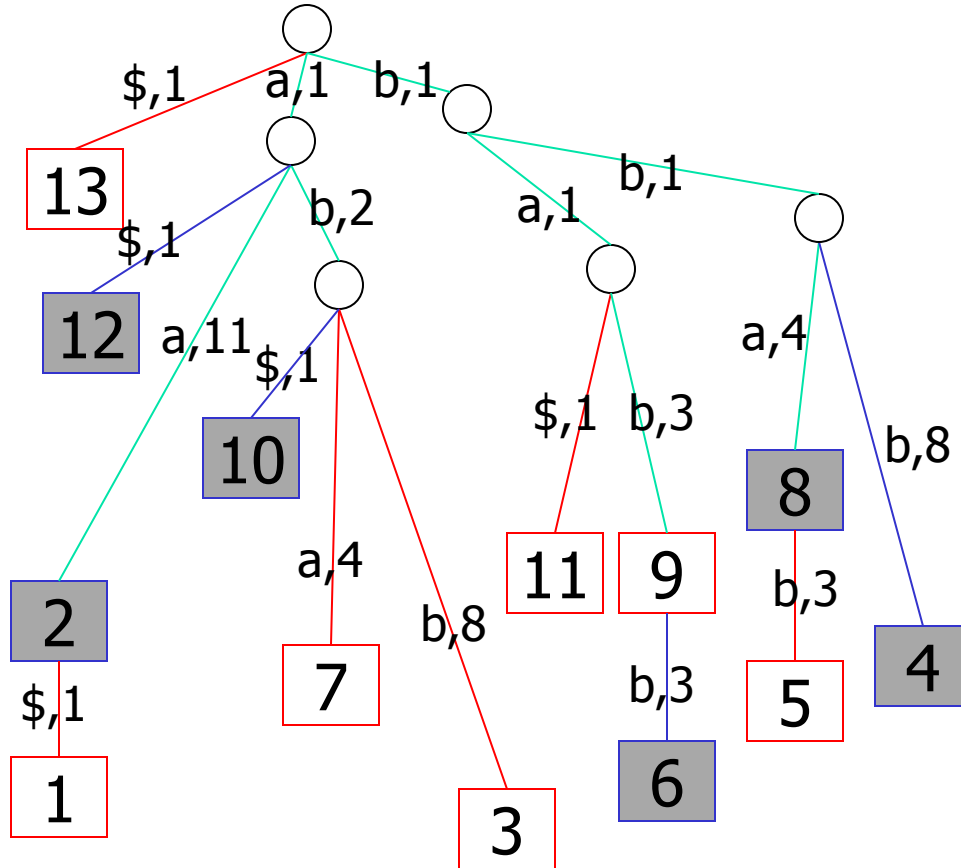


- 



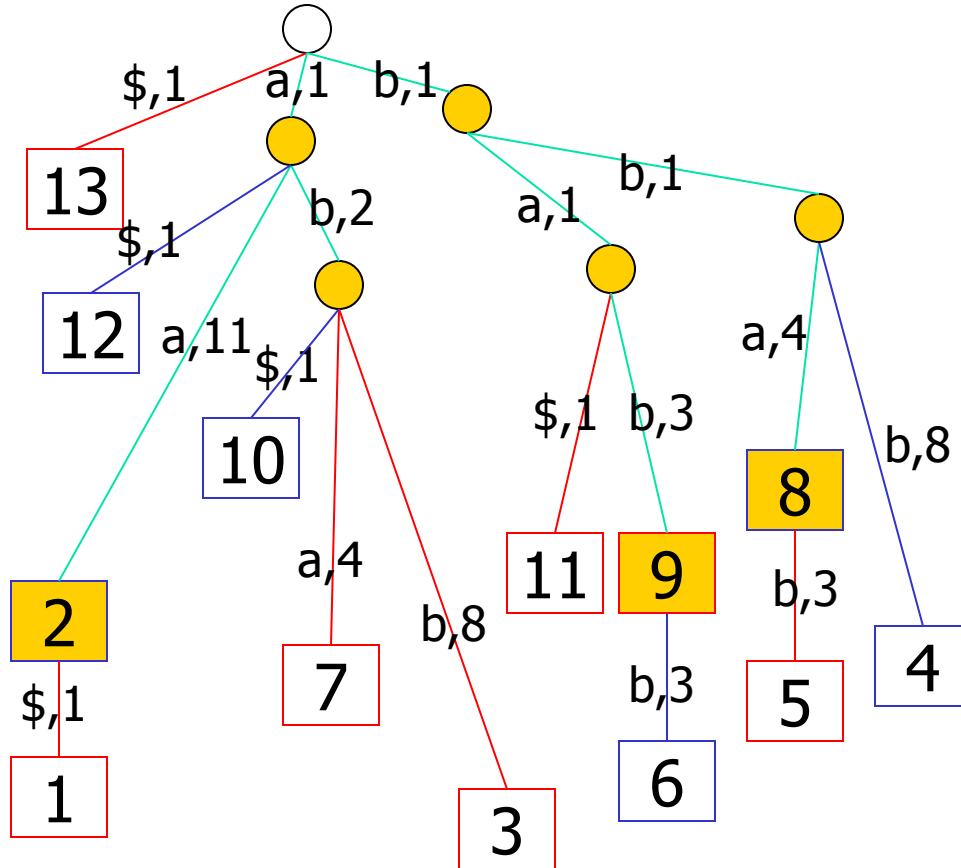
# Merge odd and even trees

- We merge  $T_o$  and  $T_e$  by DFS. We merge two edges as long as they start with the same character. The merge is ended when one edge is longer than the other.



# Merge odd and even trees

- The merging may over-merged some nodes.
- To correct the tree, we need to unmerge some nodes.



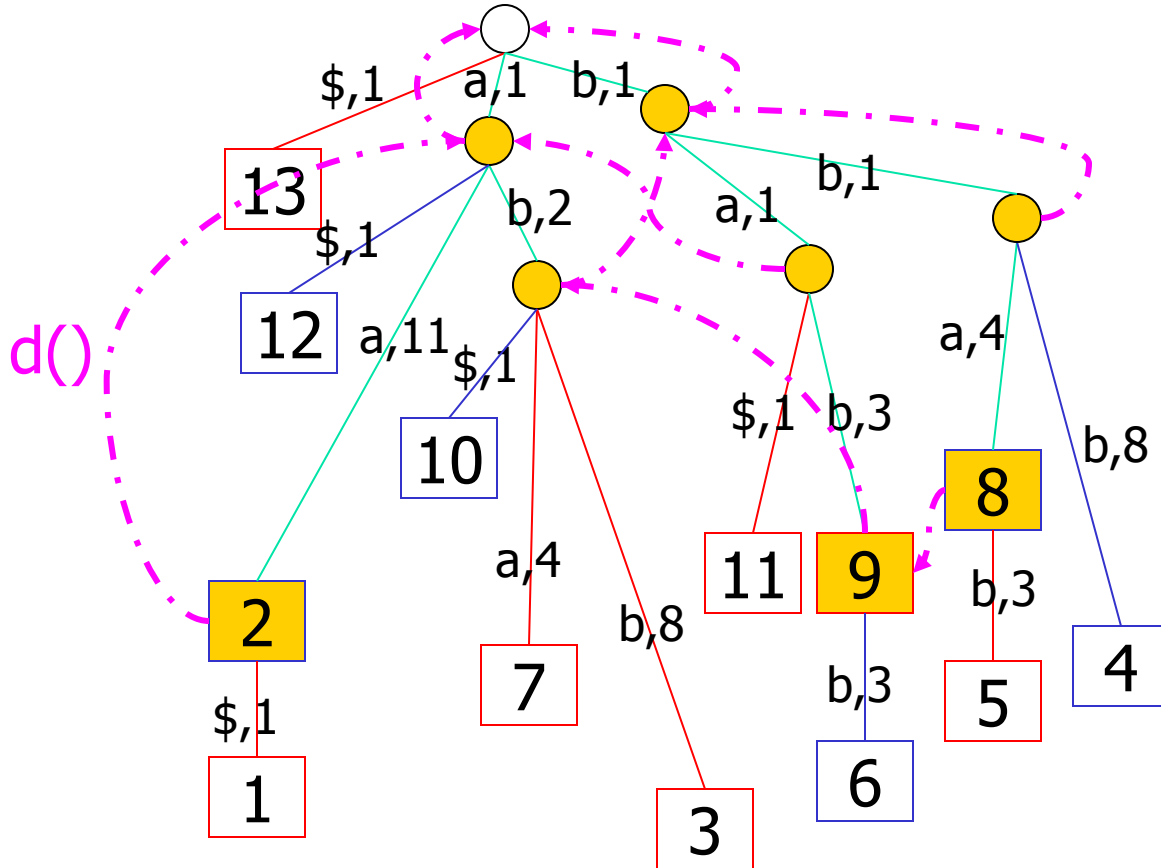


# Definition of $L()$ and $d()$

---

- For every node  $u$  which may be over-merged, there exist two leaves  $2i$  and  $2j-1$  such that  $u = \text{lca}(2i, 2j-1)$ .
  - Denote  $L(u)$  be the correct depth of  $u$ , that is,  $\text{lcp}(2i, 2j-1)$ .
- Note that  $\text{lcp}(2i, 2j-1) = 1 + \text{lcp}(2i+1, 2j)$  if  $S[2i] = S[2j-1]$ ; 0 otherwise.
- Let  $v$  be  $\text{lca}(2i+1, 2j)$ .
- Denote  $d(u) = v$ .
- Note that  $d()$  is equivalent to suffix link!

## d()

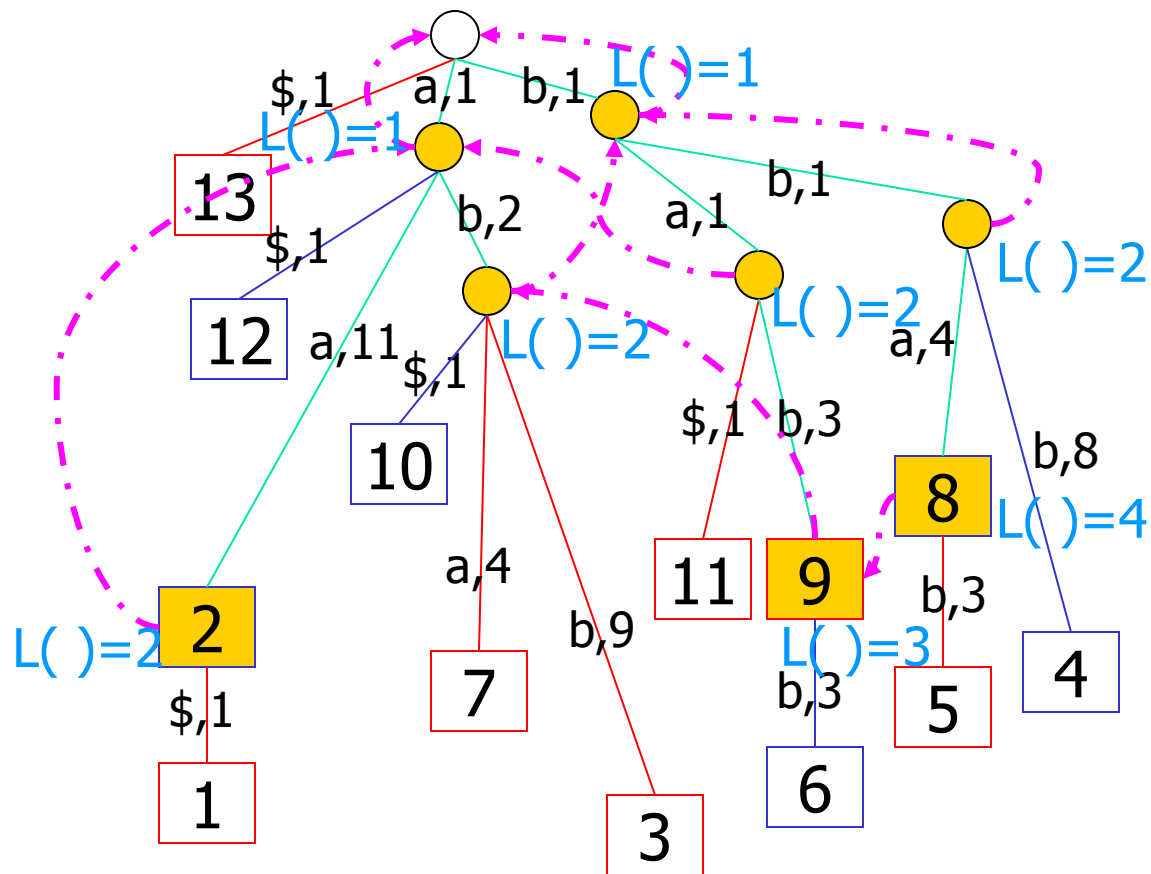




# Relationship between $L()$ and $d()$

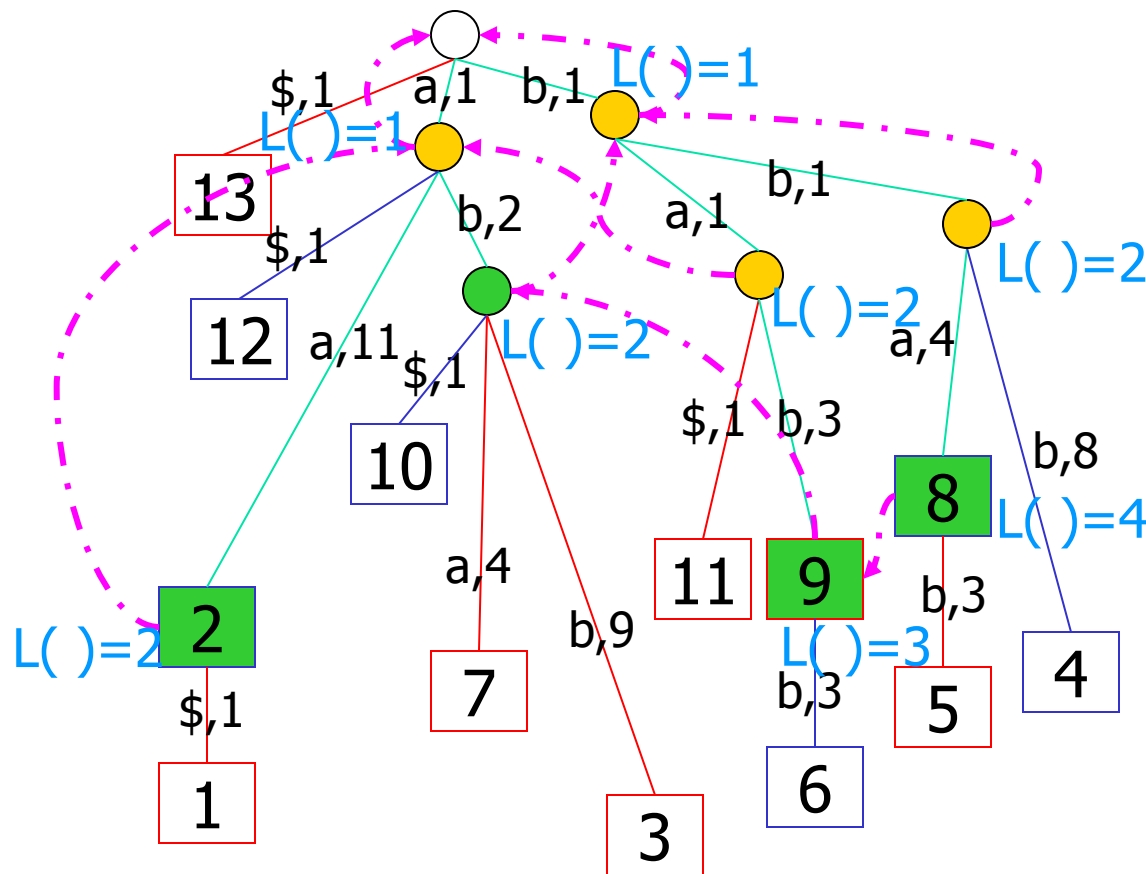
---

- Suppose  $u = \text{lca}(2i, 2j-1)$ .
- Note1: if  $u$  is not the root, then  $S[2i]=S[2j-1]$ .
- Note2:  $\text{lcp}(2i, 2j-1) = 1 + \text{lcp}(2i+1, 2j)$  if  $S[2i]=S[2j-1]$ ; 0 otherwise
- Note3:  $d(u) = \text{lcp}(2i+1, 2j)$
- Hence,  $L(u) = 1 + L(d(u))$  if  $u$  is not the root. Otherwise,  $L(u)=0$ .
- Lemma:  $L(u)$  = the length of the purple path from  $u$  to the root.



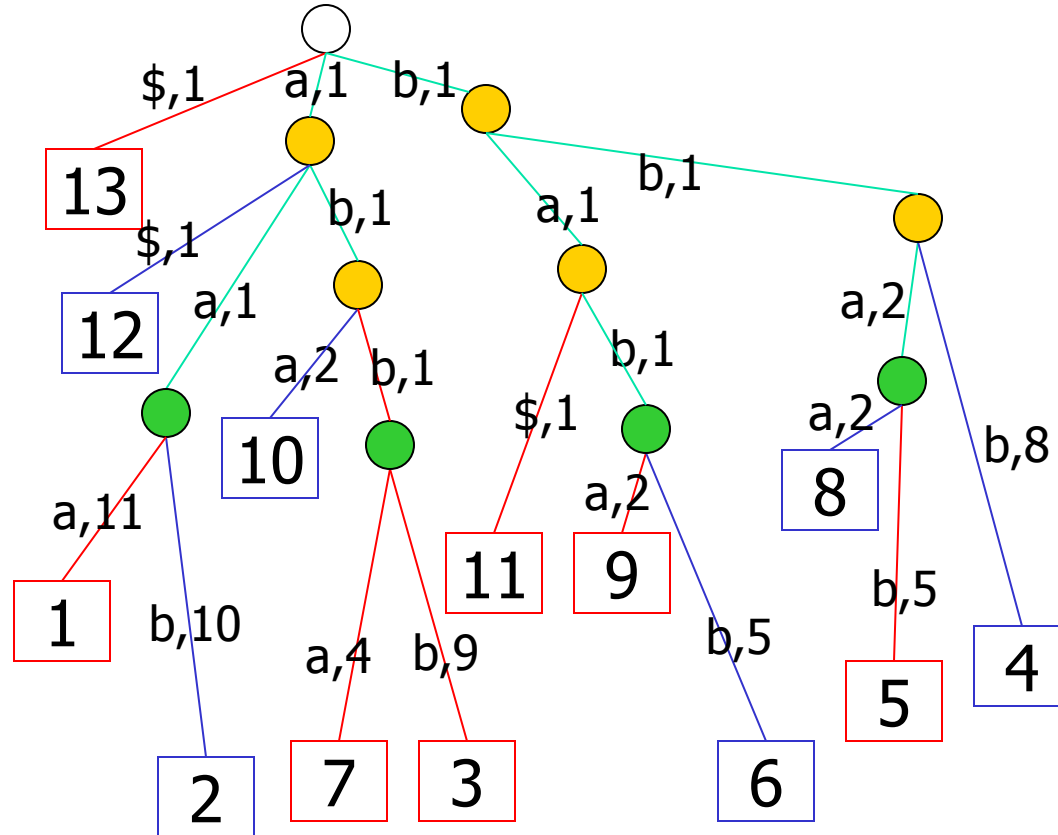
# Unmerge the border nodes based on $L()$ (I)

$S = \text{aaabbbabbaba}\$$





# Unmerge the border nodes based on $L()$ (II)





# Time complexity for merging

---

- Merge the tree using DFS takes  $O(n)$  time.
- Compute the links  $d()$  takes  $O(n)$  time.
- Compute  $L()$  takes  $O(n)$  time.
- Unmerge takes  $O(n)$  time.



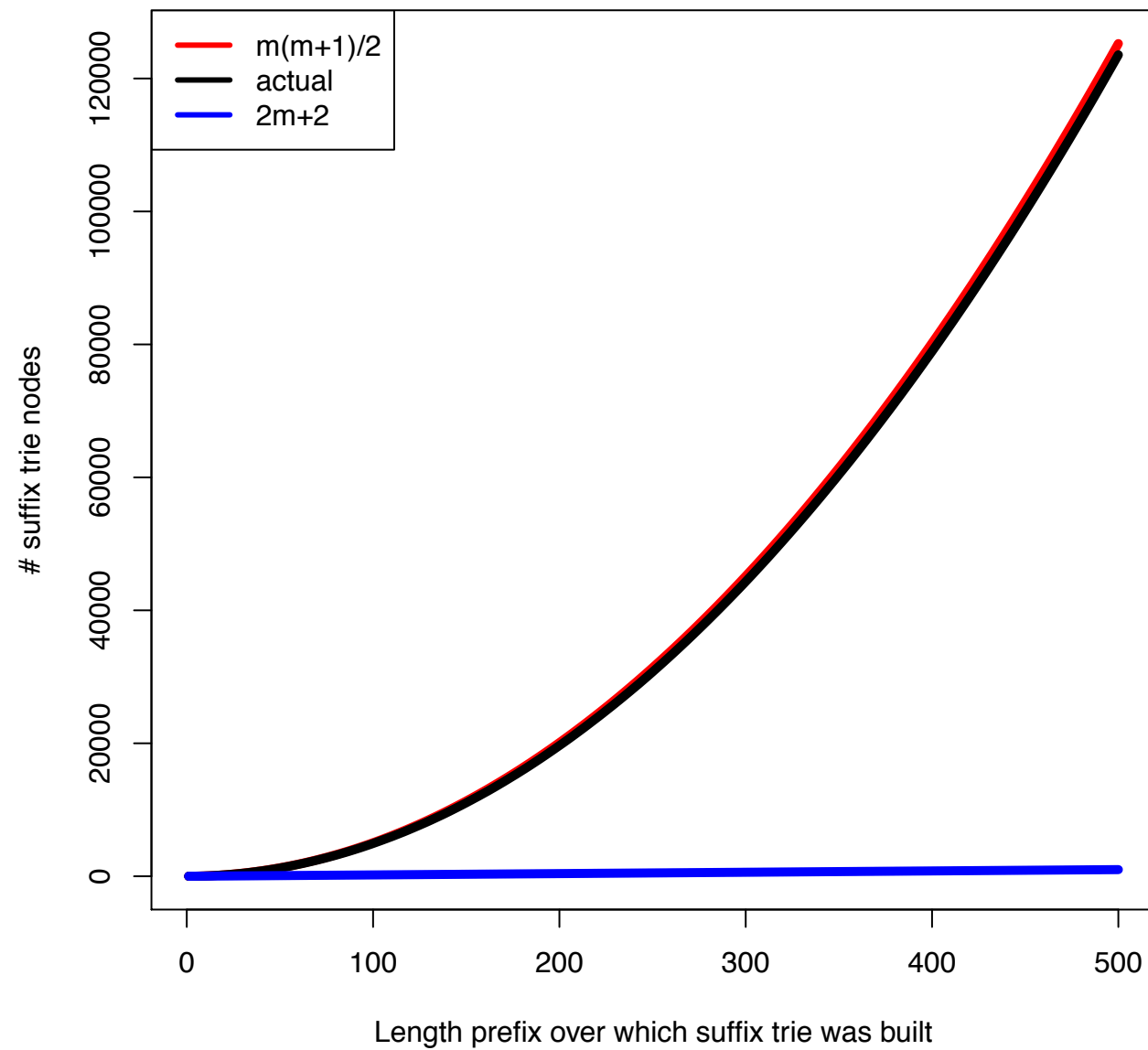
# Total time complexity of Farach's algorithm

---

- Stage 1:  $\text{Time}(n/2) + O(n)$
- Stage 2:  $O(n)$
- Stage 3:  $O(n)$
  
- Thus,  $\text{Time}(n) = \text{Time}(n/2) + O(n)$ .
- By solving the equation,
  - $\text{Time}(n) = O(n)$ .

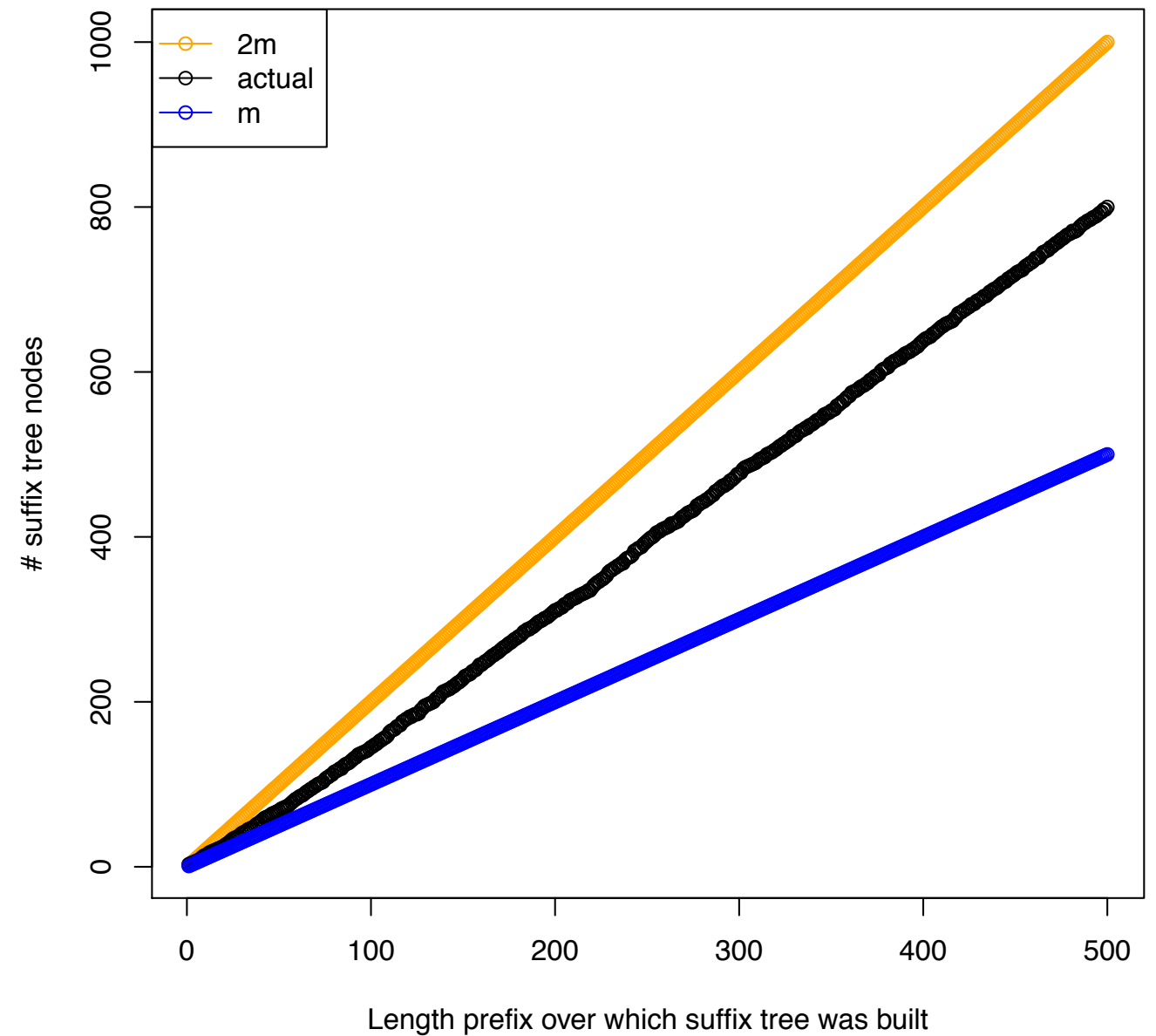
# Suffix trie

>100K nodes



# Suffix tree

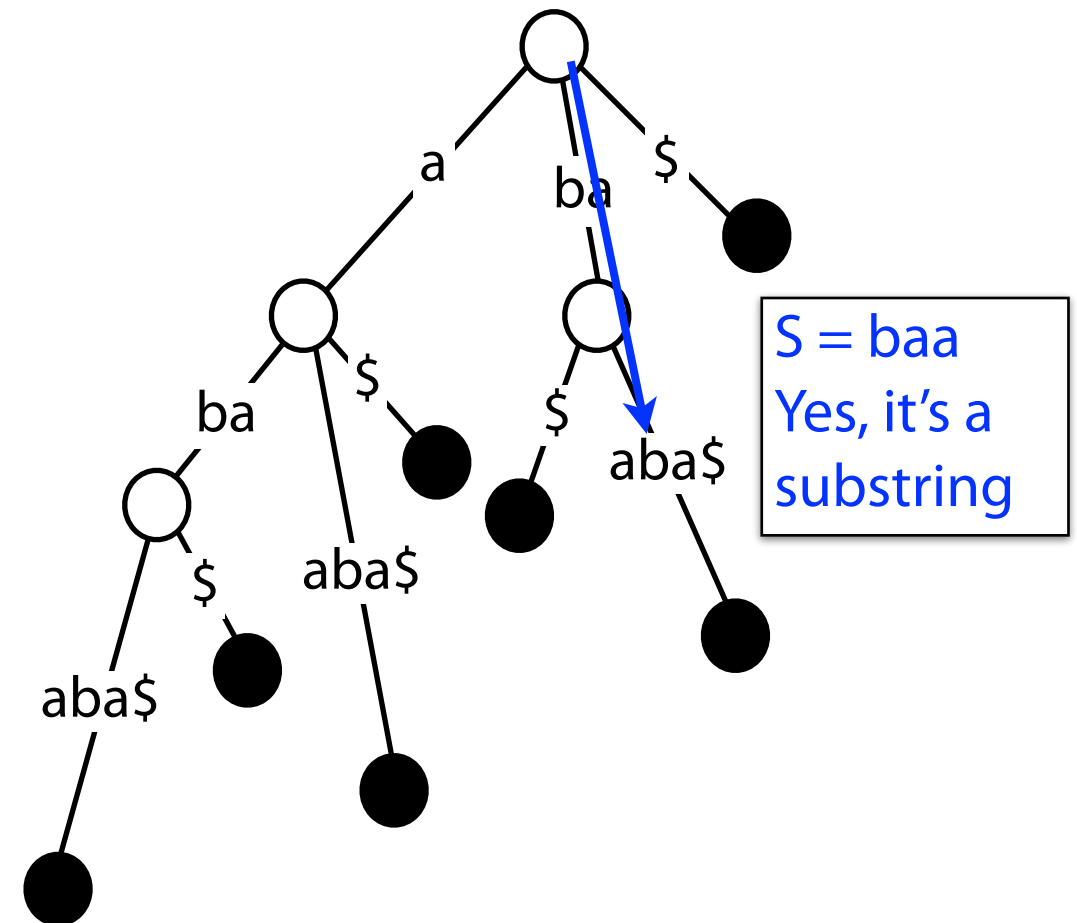
<1K nodes



# Suffix tree

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

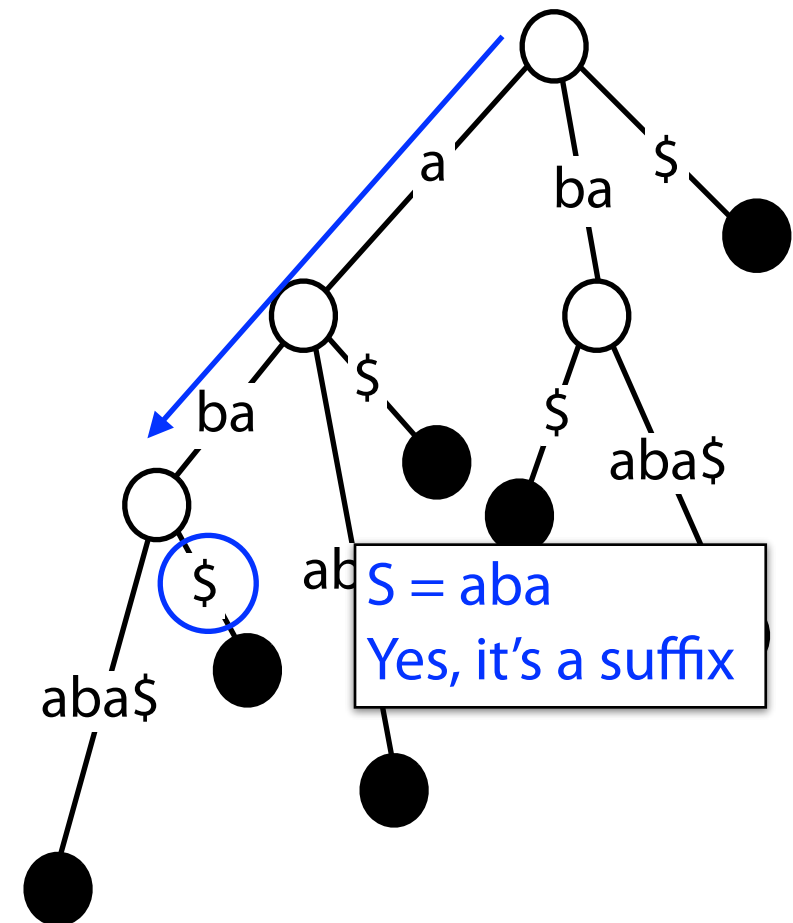
Same procedure as for suffix trie, but we have to deal with coalesced edges



# Suffix tree

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

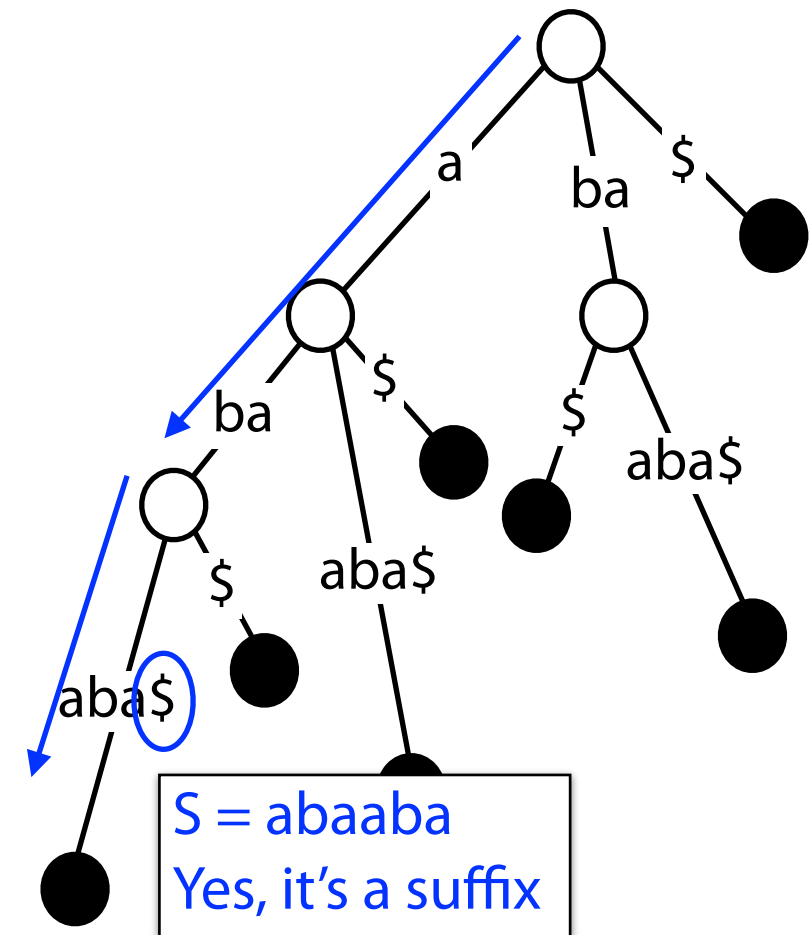
Same procedure as for suffix trie, but we have to deal with coalesced edges



# Suffix tree

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

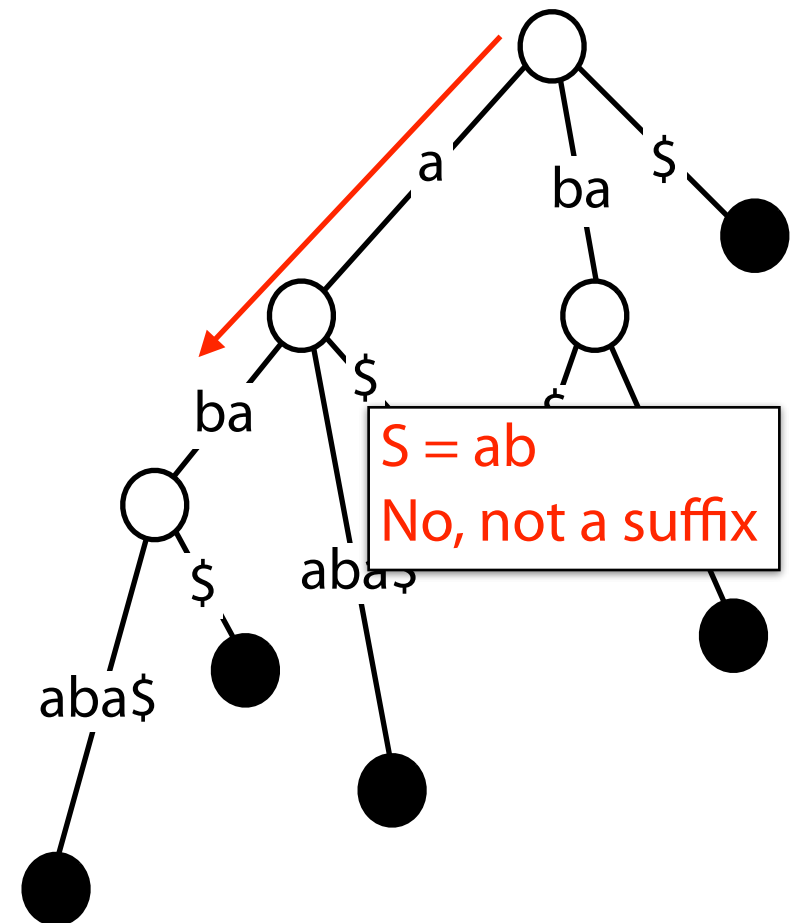
Same procedure as for suffix trie, but we have to deal with coalesced edges



# Suffix tree

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

Same procedure as for suffix trie, but we have to deal with coalesced edges

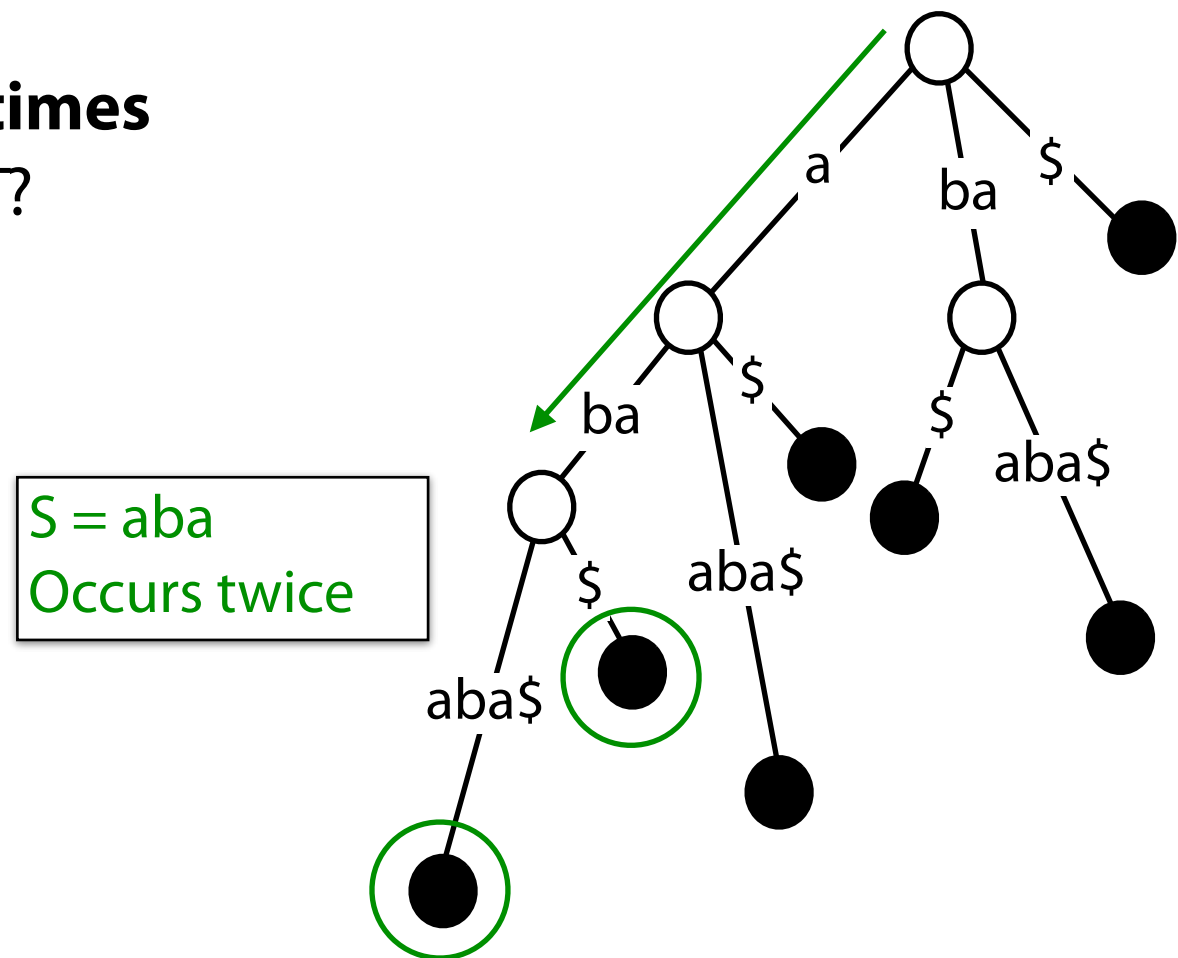




# Suffix tree

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

We can also **count or find** all the matches of  $P$  to  $T$ . Let  $k = \#$  matches.

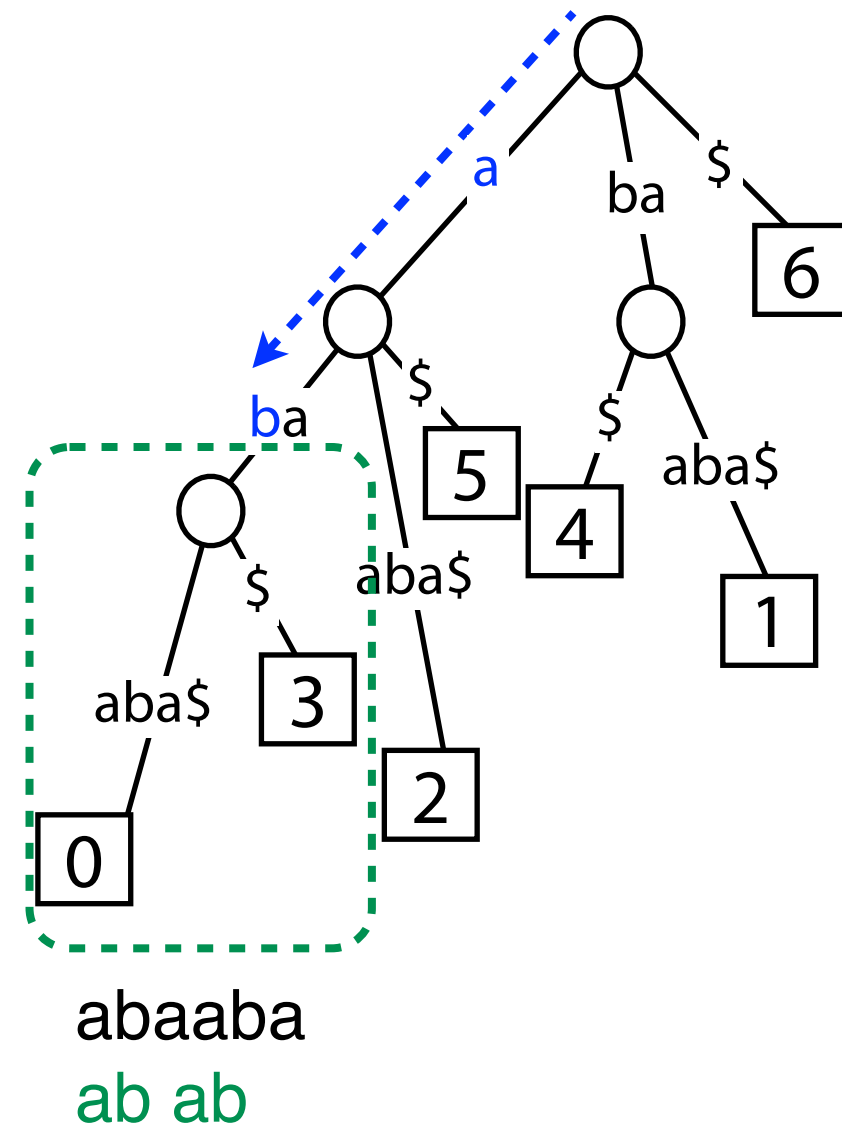
E.g.,  $P = ab$ ,  $T = abaaba\$$

$O(n)$  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

# leaves in subtree is  $k$ ,  
# non-leaves is  $\leq k-1$

$O(n + k)$  time overall



# Suffix tree: some bounds

Suffix tree	
Time: Does $P$ occur?	$O(n)$
Time: Count $k$ occurrences of $P$	$O(n + k)$
Time: Report $k$ locations of $P$	$O(n + k)$
Space	$O(m)$

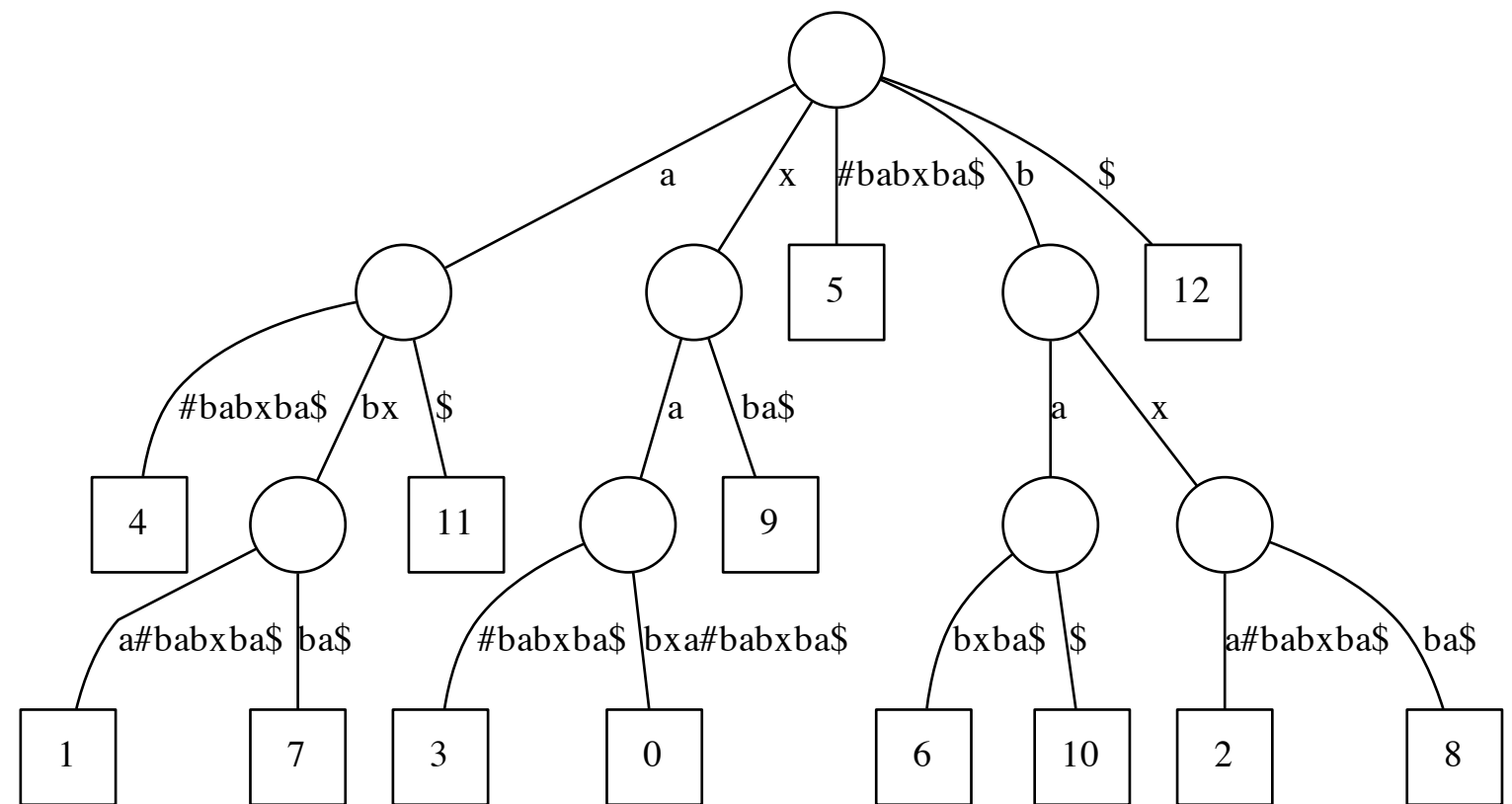
$m = |T|$ ,  $n = |P|$ ,  $k = \#$  occurrences of  $P$  in  $T$

# Suffix tree application: find longest common substring

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

$X = \text{xabxa}$     $Y = \text{babxba}$

$X\#Y\$ = \text{xabxa}\#\text{babxba}\$$



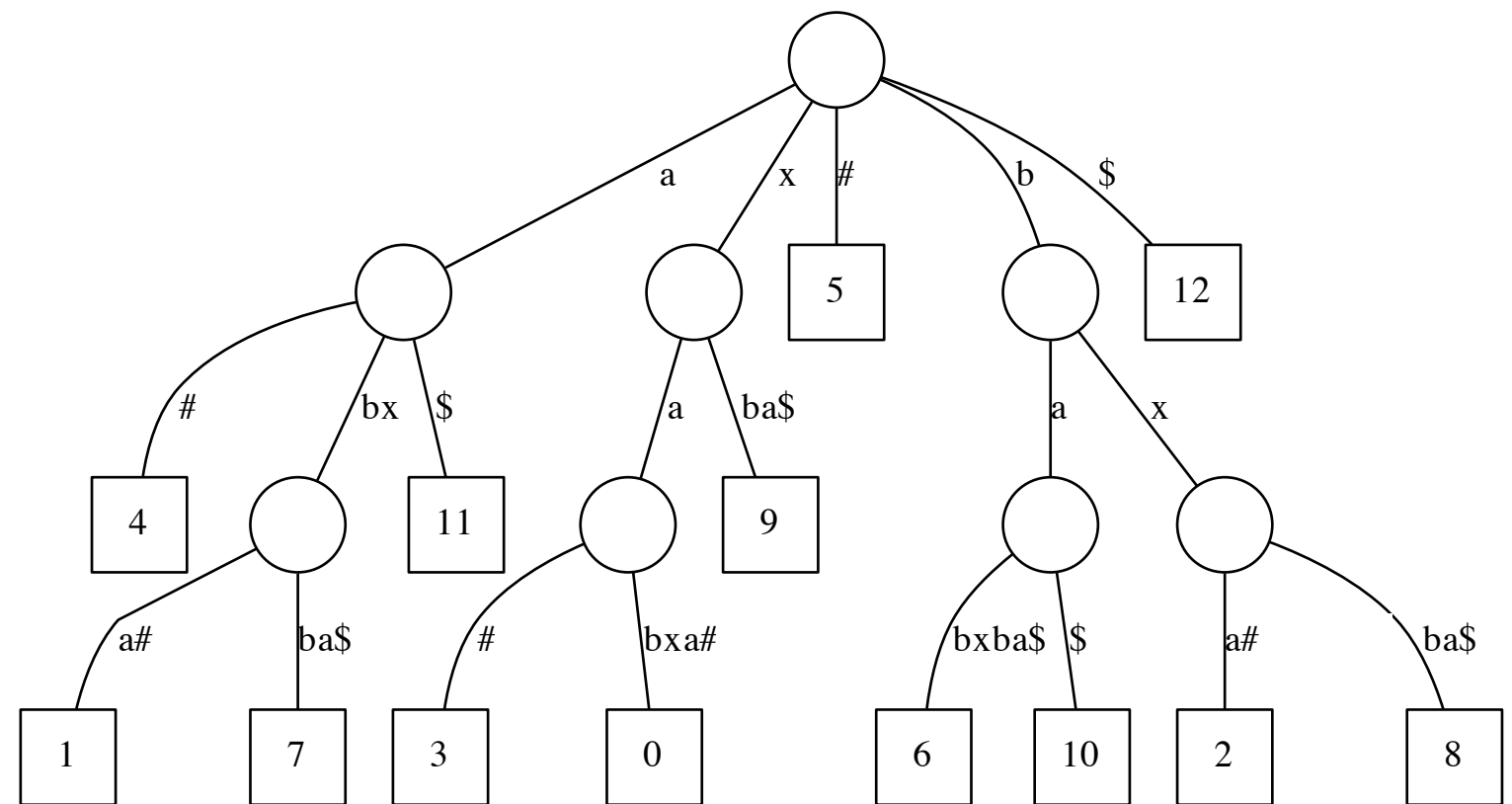
For clarity, if a suffix includes part of both strings, let's hide the portion after the  $\#$

# Suffix tree application: find longest common substring

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

$X = \text{xabxa}$     $Y = \text{babxba}$

$X\#Y\$ = \text{xabxa}\#\text{babxba}\$$



For clarity, if a suffix includes part of both strings, let's hide the portion after the  $\#$

Now suffixes of  $X$  end in  $\#$  and suffixes of  $Y$  end in  $\$$

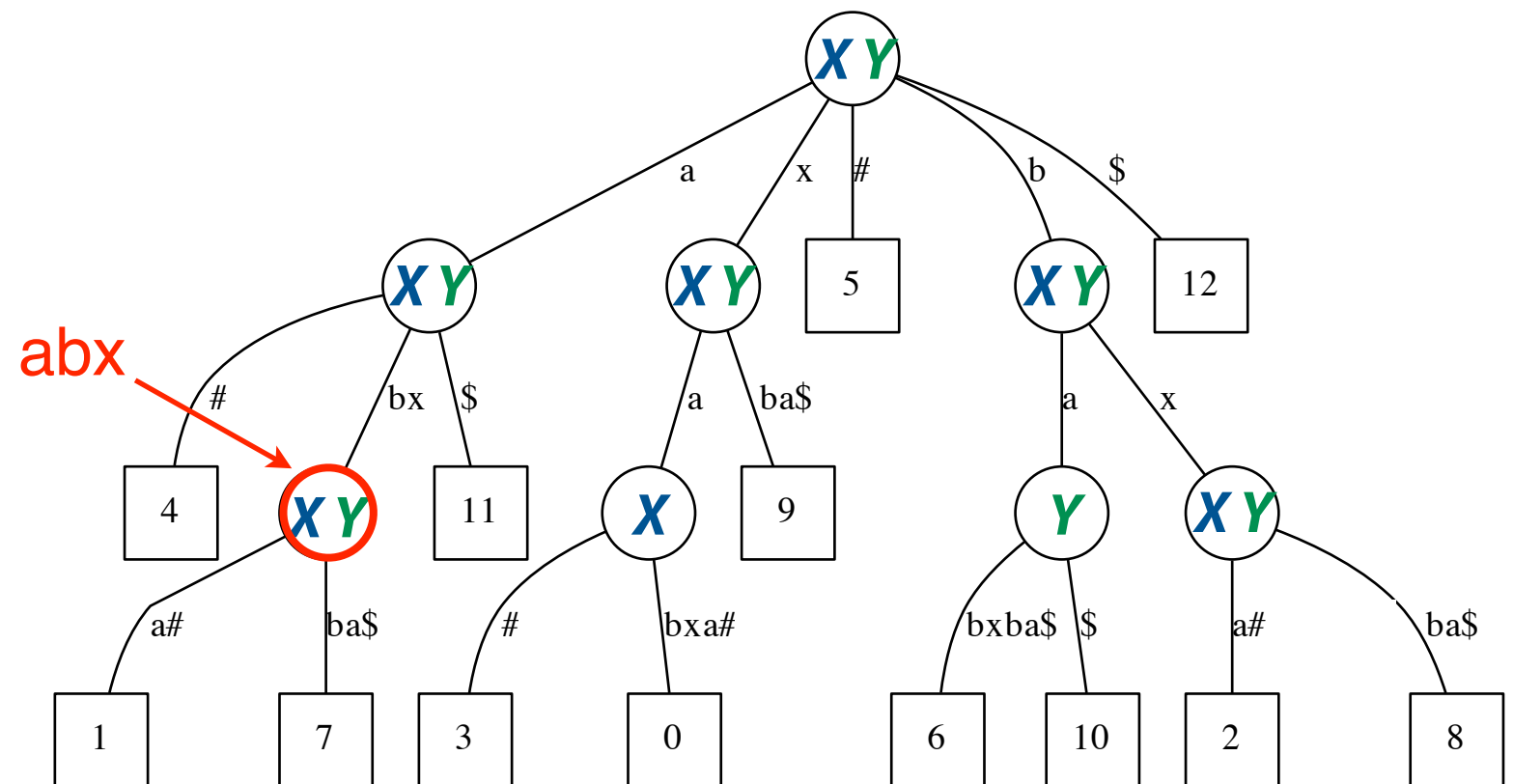
# Suffix tree application: find longest common substring

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

$X = \text{xabxa}$     $Y = \text{babxba}$

$X\#Y\$ = \text{xabxa}\#\text{babxba}\$$

Leaves with labels in  $[0, 5]$  are suffixes of  $X\#$ , labels of  $[6, 12]$  are suffixes of  $Y\$$



Traverse tree, annotating each node according to whether leaves below include suffixes of  $X$ ,  $Y$  or both

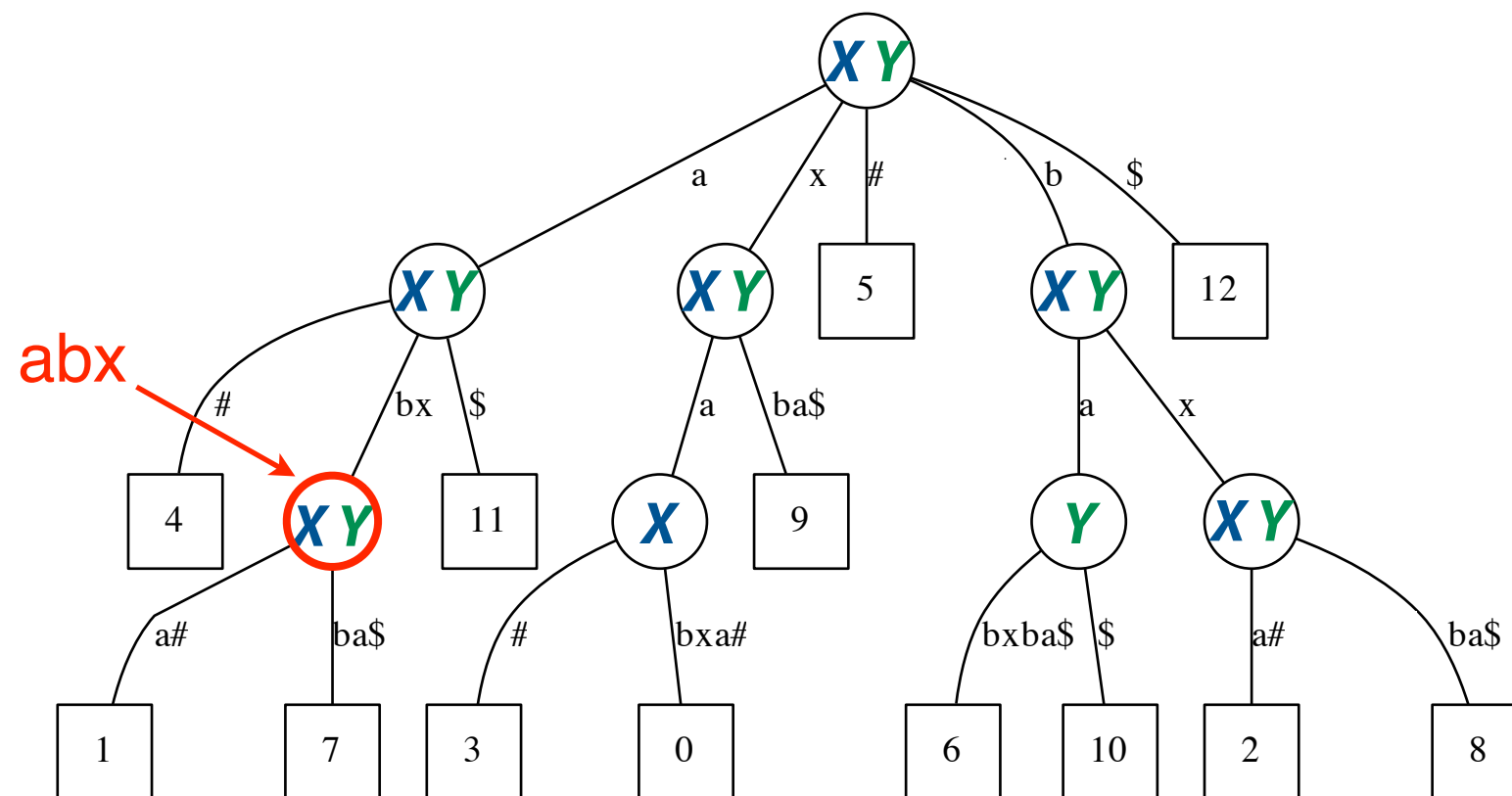
Node w/ greatest label depth annotated  $XY$  corresponds to LCS

$O(|X| + |Y|)$  time and space!

# Suffix tree application: generalized suffix trees

It's often useful to build a suffix tree of many strings at once

This is a *generalized suffix tree*. See *Gusfield* 6.4.



# Suffix trees in the real world

## Alignment of whole genomes (MUMmer):

Delcher, Arthur L., et al. "Alignment of whole genomes." *Nucleic Acids Research* 27.11 (1999): 2369-2376.

Delcher, Arthur L., et al. "Fast algorithms for large-scale genome alignment and comparison." *Nucleic Acids Research* 30.11 (2002): 2478-2483.

Kurtz, Stefan, et al. "Versatile and open software for comparing large genomes." *Genome Biol* 5.2 (2004): R12.

~ 4,000 citations

<http://mummer.sourceforge.net>

## Computing and visualizing repeats in whole genomes (REPuter):

Kurtz, Stefan, and Chris Schleiermacher. "REPuter: Fast computation of maximal repeats in complete genomes." *Bioinformatics* 15.5 (1999): 426-427.

Kurtz, Stefan, et al. "REPuter: the manifold applications of repeat analysis on a genomic scale." *Nucleic acids research* 29.22 (2001): 4633-4642.

> 1,000 citations   <http://bibiserv.techfak.uni-bielefeld.de/reputer>

## Identifying sequence motifs

Marsan, Laurent, and Marie-France Sagot. "Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification." *Journal of Computational Biology* 7.3-4 (2000): 345-362.

Sagot, Marie. "Spelling approximate repeated or common motifs using a suffix tree." *LATIN'98: Theoretical Informatics* (1998): 374-390.

~ 600 citations

Also used in: multiple alignment



# Suffix trees in the real world: MUMmer

FASTA file containing "reference" ("text")

FASTA file containing  
ALU string

Indexing  
phase: ~2  
minutes

Matching  
phase:  
very fast

```
mummer — langmead@igm1:~ — bash — 120x31
Bens-MacBook-Pro:mummer langmead$ cat alu50.fa
>Alu
GCGCGGTGGCTCACGCCTGTAATCCAGCACTTTGGGAGGCCGAGGCGGG
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
```

**Columns:**

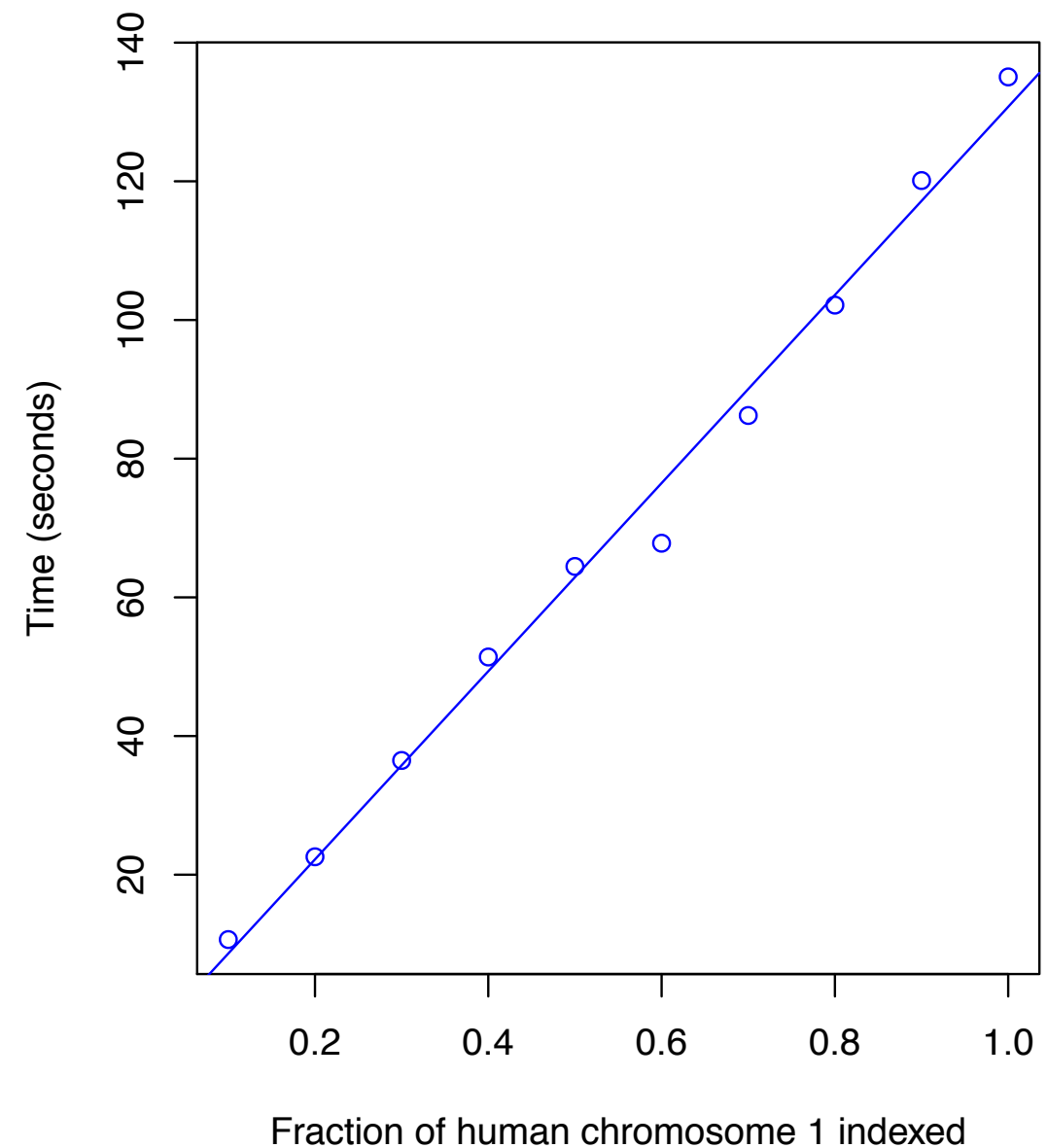
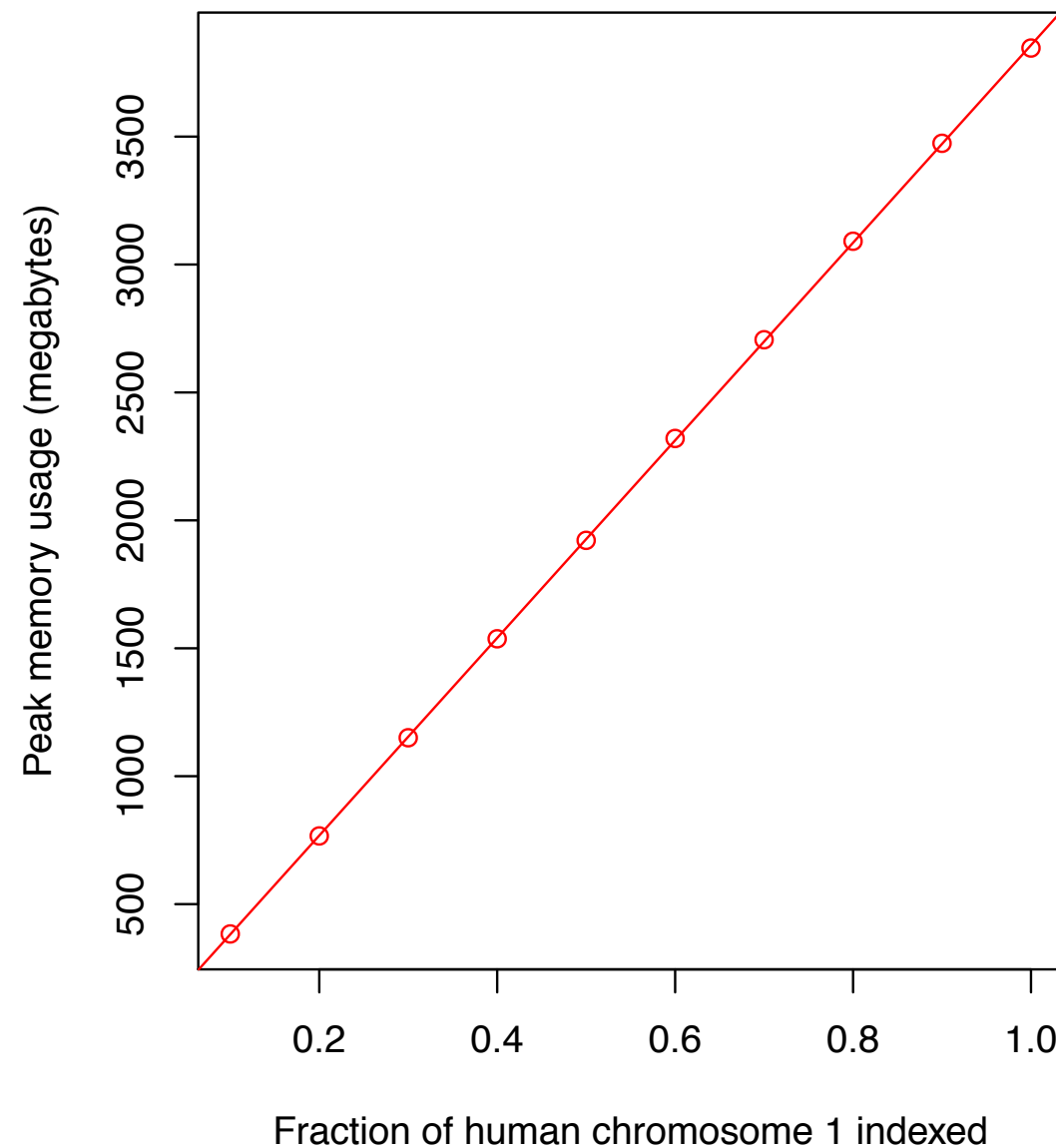
**1. Match offset in T**

**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: the constant factor

$O(m)$  is desirable, but “constant factor” is significant, sometimes making the suffix tree inconvenient

Constant factor varies depending on implementation:

MUMmer constant factor = 3.94 GB / 250 million nt  $\approx$  **15.76 bytes per nt**

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

Suffix tree of human genome will be >45GB, perhaps much larger depending on exact data structures underlying suffix tree nodes/edges