

# Suffix Trees

Description follows **Dan Gusfield's book "Algorithms on Strings, Trees and Sequences"**

Slides sources: Pavel Shvaiko, (University of Trento), Haim Kaplan (Tel Aviv University)



# Outline

- Introduction
- Suffix Trees (ST)
- Building STs in linear time: Ukkonen's algorithm
- Applications of ST



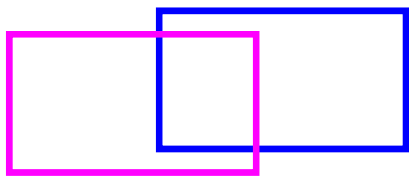
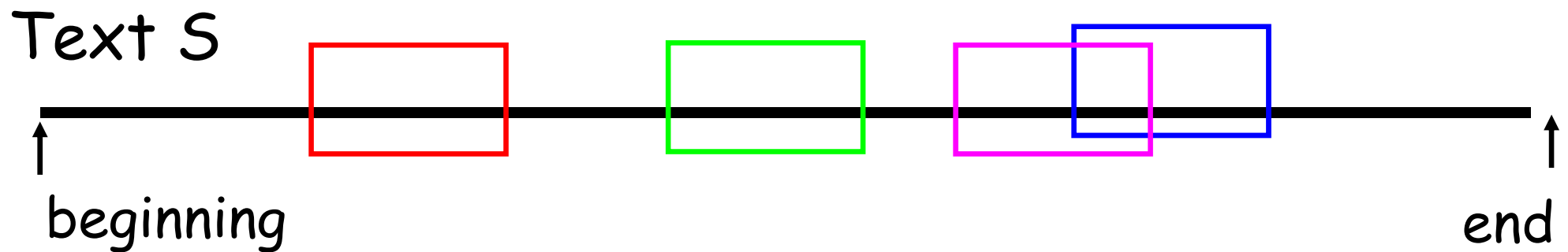
# Introduction



# Exact String/Pattern Matching

$$|S| = m,$$

n different patterns  $p_1 \dots p_n$



Pattern occurrences that overlap



# String/Pattern Matching - I

- Given a **text**  $S$ , answer **queries** of the form: is the **pattern**  $p_i$  a substring of  $S$ ?
- Knuth-Morris-Pratt 1977 (KMP) string matching alg:
  - $O(|S| + |p_i|)$  time per query.
  - $O(n|S| + \sum_i |p_i|)$  time for  $n$  queries.
- Suffix tree solution:
  - $O(|S| + \sum_i |p_i|)$  time for  $n$  queries.

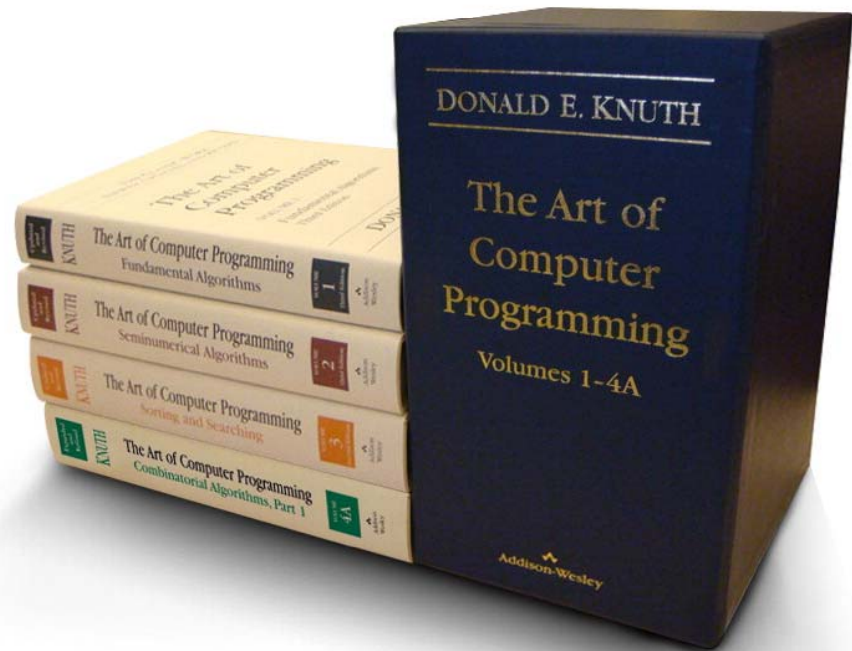


# String/Pattern Matching - II

- KMP preprocesses the patterns  $p_i$ ;
- The suffix tree algorithm:
  - preprocess  $S$  in  $O(|S|)$ : build a suffix tree for  $S$
  - when a pattern of length  $n$  is input, the algorithm searches it in  $O(n)$  time using that suffix tree.



# Donald Knuth



# Prefixes & Suffixes

- **Prefix** of  $S$ : substring of  $S$  beginning at the first position of  $S$
- **Suffix** of  $S$ : substring that ends at its last position
- $S = \text{AACTAG}$ 
  - Prefixes:  $\text{AACTAG}, \text{AACTA}, \text{AACT}, \text{AAC}, \text{AA}, \text{A}$
  - Suffixes:  $\text{AACTAG}, \text{ACTAG}, \text{CTAG}, \text{TAG}, \text{AG}, \text{G}$
- $P$  is a substring of  $S$  iff  $P$  is a prefix of some suffix of  $S$ .
- Notation:  $S[i,j] = S(i), S(i+1), \dots, S(j)$



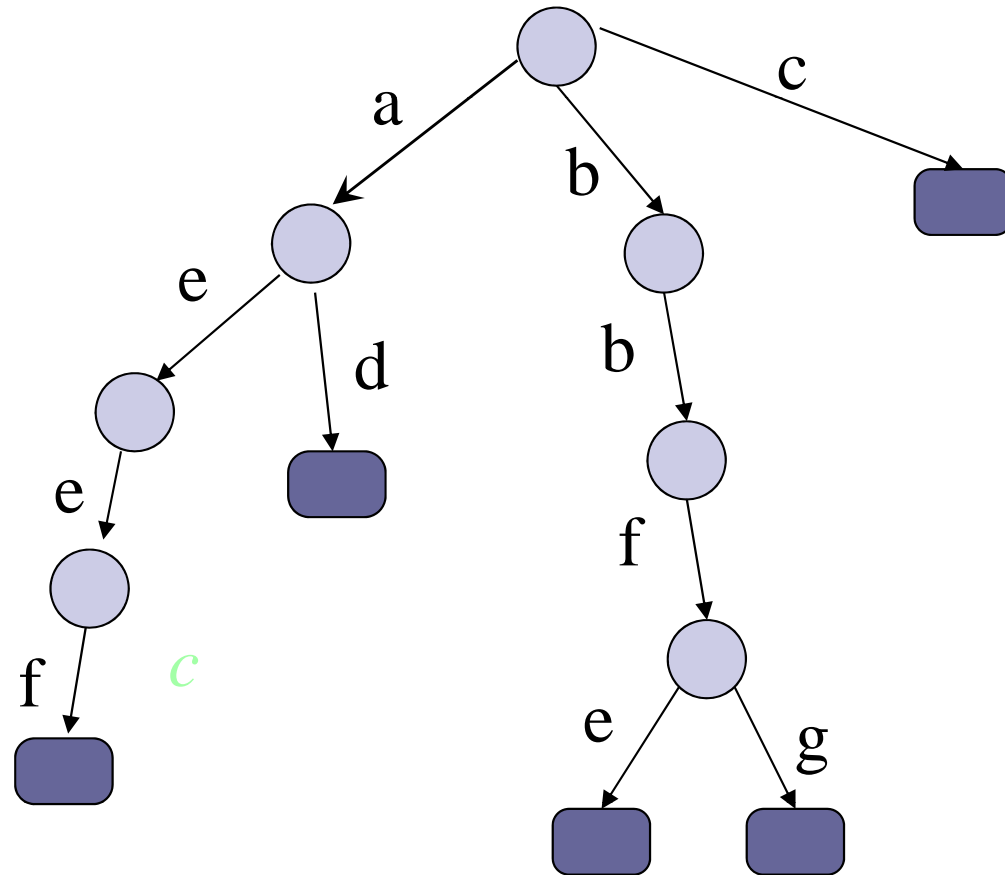
# Suffix Trees



# Trie

- A tree representing a set of strings.

{  
aeef  
ad  
bbfe  
bbfg  
c  
}

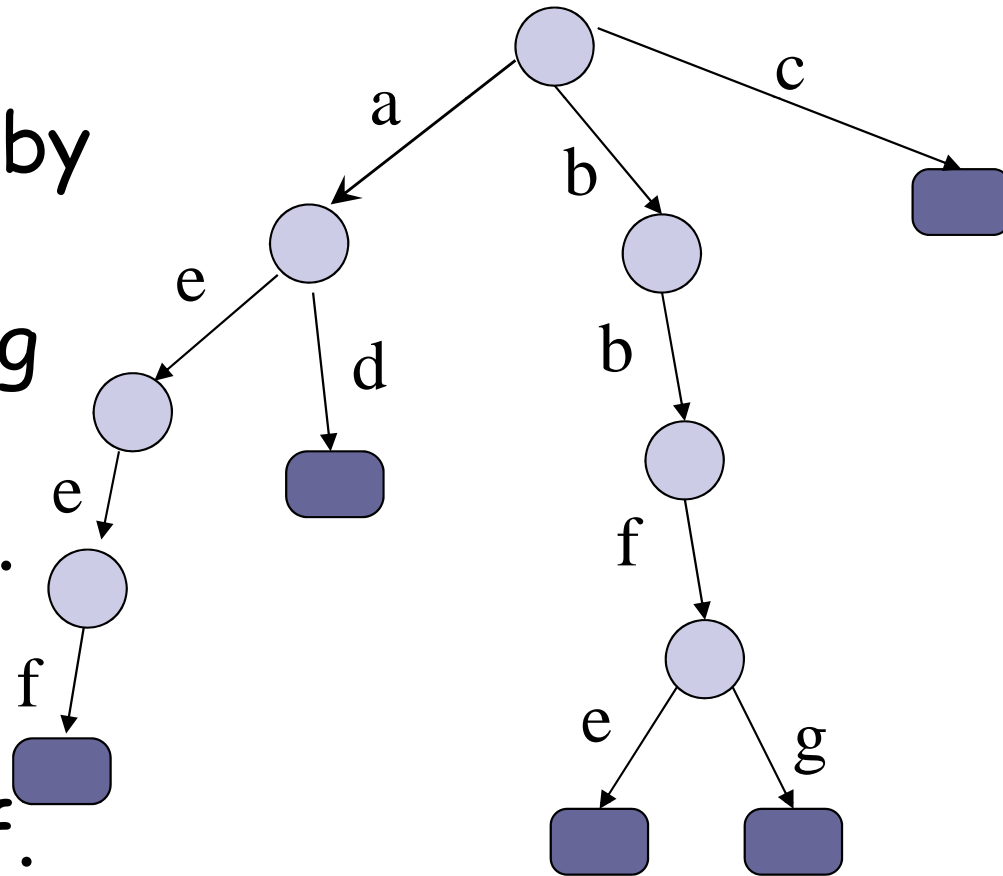


# Trie (Cont)

- Assume no string is a prefix of another

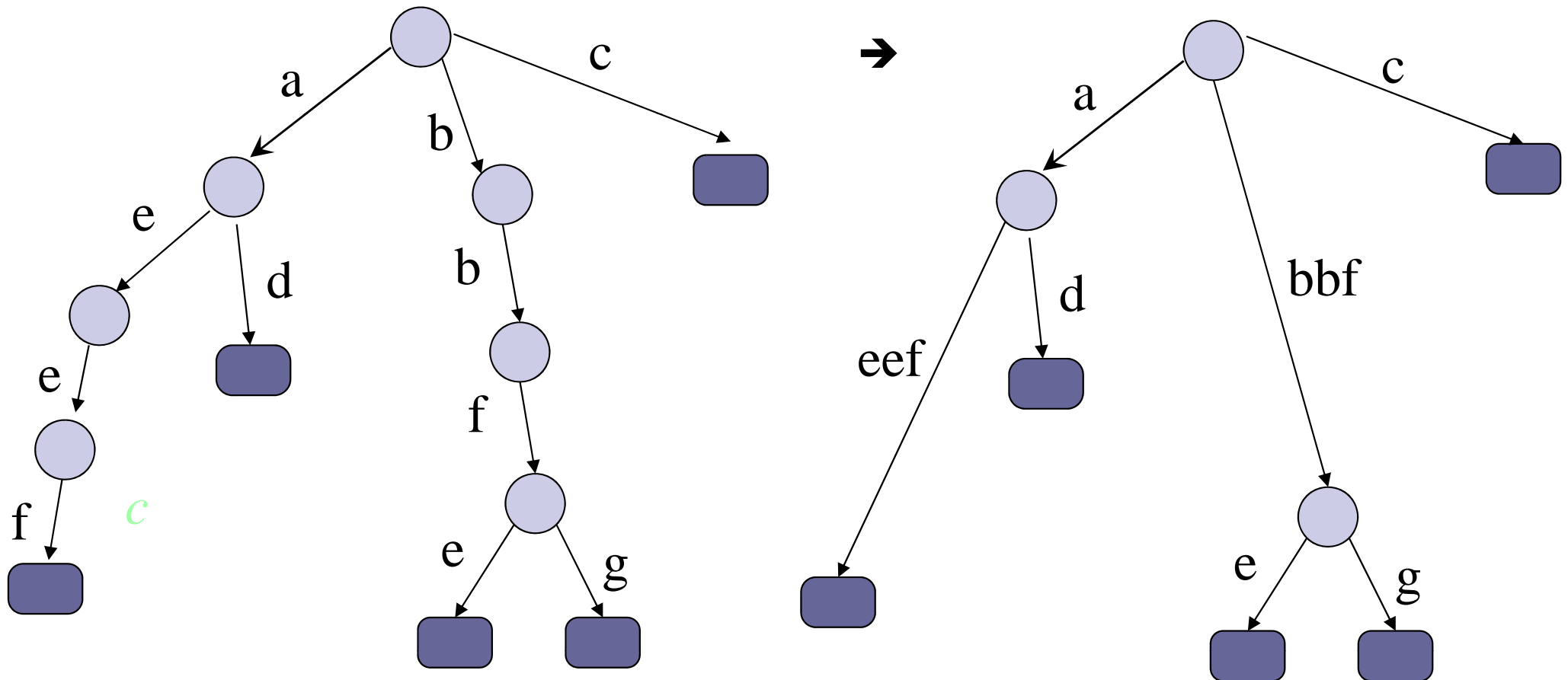
Each edge is labeled by a letter,  
no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.



# Compressed Trie

- Compress unary nodes, label edges by strings

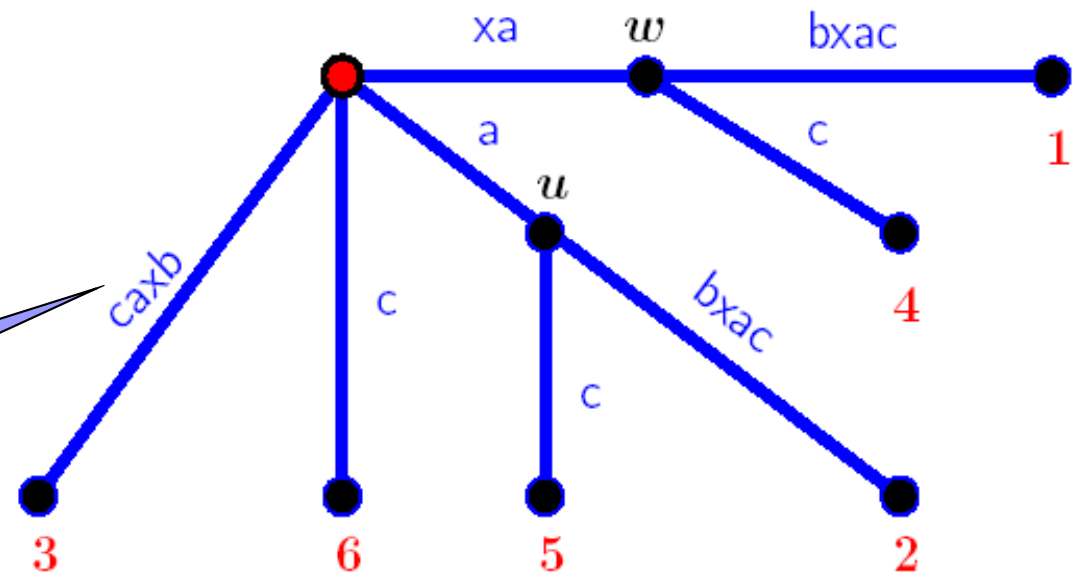


# Def: Suffix Tree for $S$

$|S|=m$

1. A rooted tree  $T$  with  $m$  leaves numbered  $1, \dots, m$ .
2. Each internal node of  $T$ , except perhaps the root, has  $\geq 2$  children.
3. Each edge of  $T$  is labeled with a nonempty substring of  $S$ .
4. All edges out of a node must have edge-labels starting with different characters.
5. For any leaf  $i$ , the concatenation of the edge-labels on the path from the root to leaf  $i$  exactly spells out  $S[i, m]$ , the suffix of  $S$  that starts at position  $i$ .

$S=xabxac$



Caution: this label is read downwards!

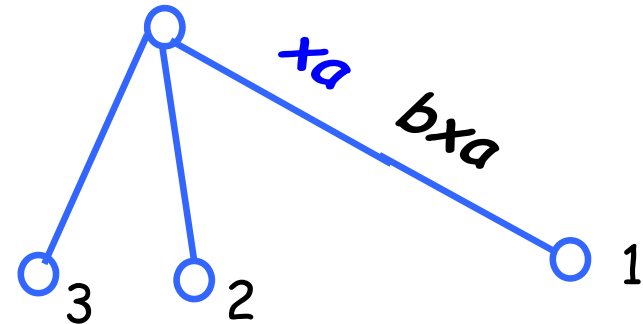


# Existence of a suffix tree $\mathcal{S}$

- If one suffix  $S_j$  of  $\mathcal{S}$  matches a prefix of another suffix  $S_i$  of  $\mathcal{S}$ , then the path for  $S_j$  would not end at a leaf.

- $\mathcal{S} = xabxa$

- $S_1 = xabxa$  and  $S_4 = xa$

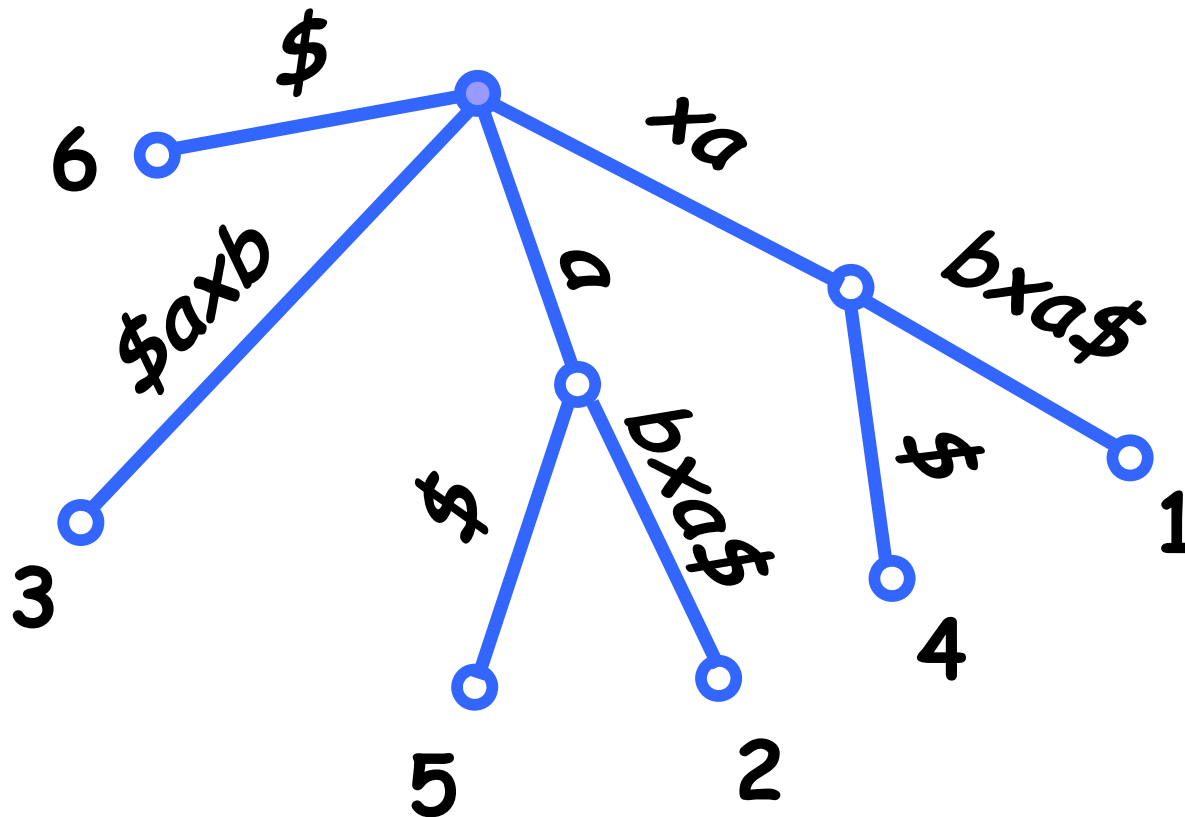


- How to avoid this problem?

- Assume that the last character of  $\mathcal{S}$  appears nowhere else in  $\mathcal{S}$ .
- Add a new character  $\$$  not in the alphabet to the end of  $\mathcal{S}$ .



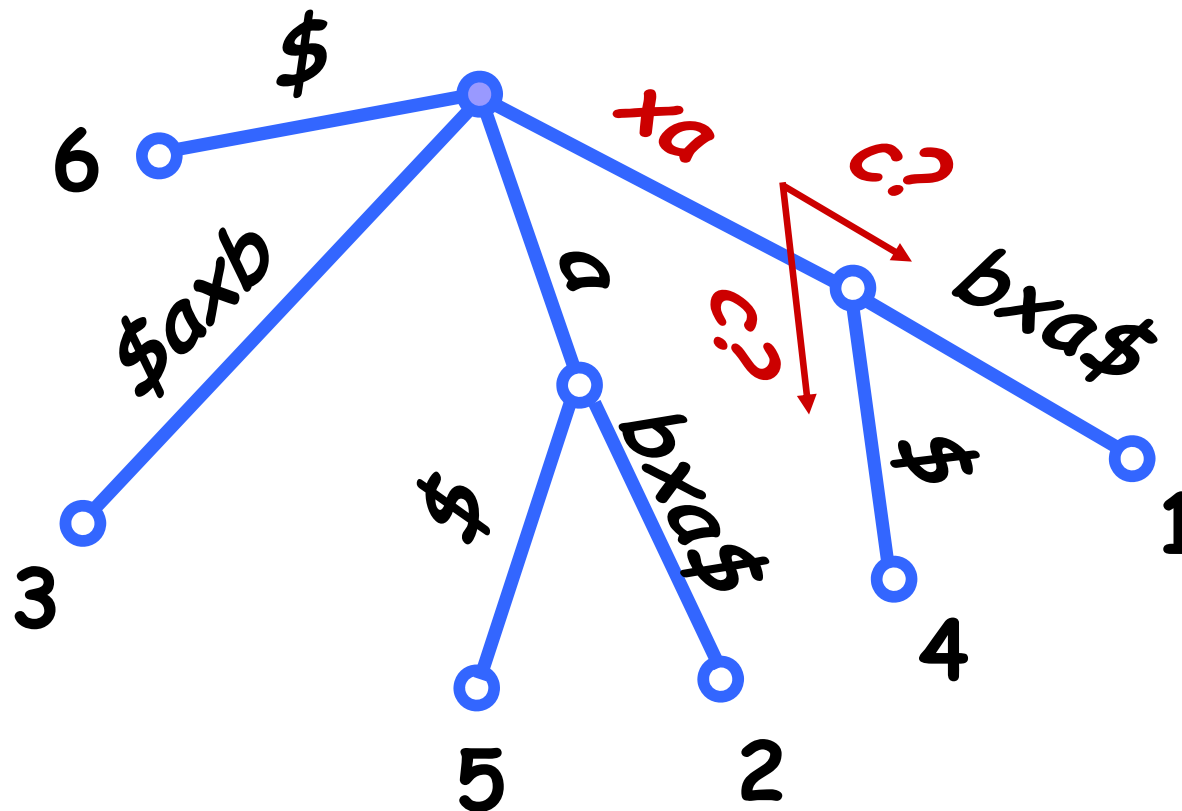
# Example: Suffix Tree for $S=xabxa\$$



# Example: Suffix Tree for $S = xabxa\$$

Query:  $P = xac$

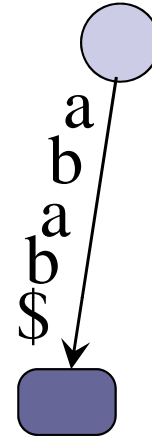
- $P$  is a substring of  $S$  iff  $P$  is a prefix of some suffix of  $S$ .



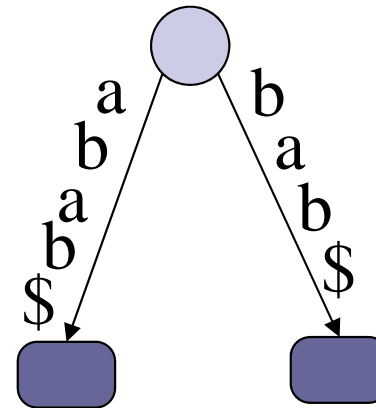
# Trivial algorithm to build a Suffix tree

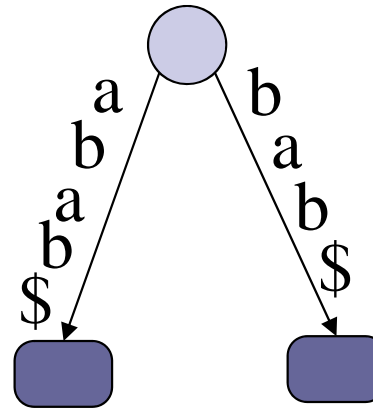
S= abab

Put the largest suffix in

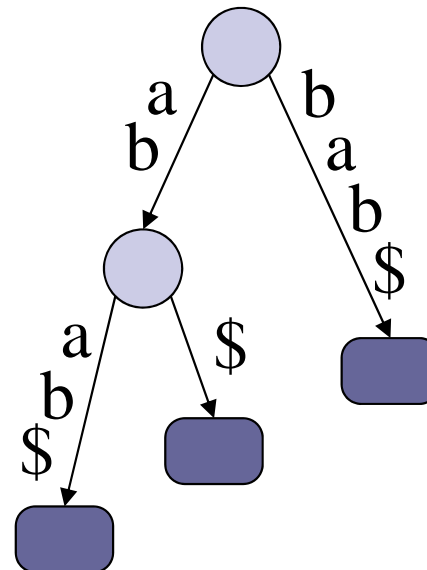


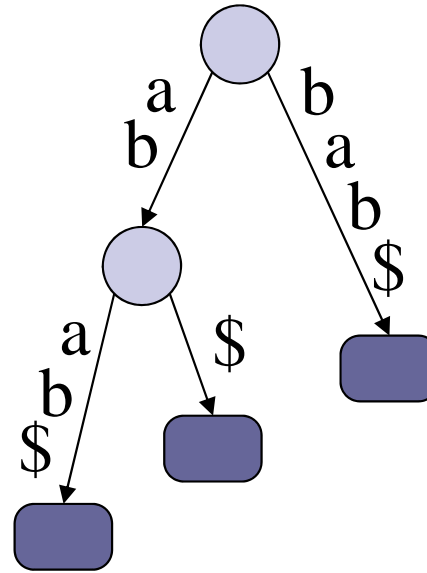
Put the suffix **bab\$** in



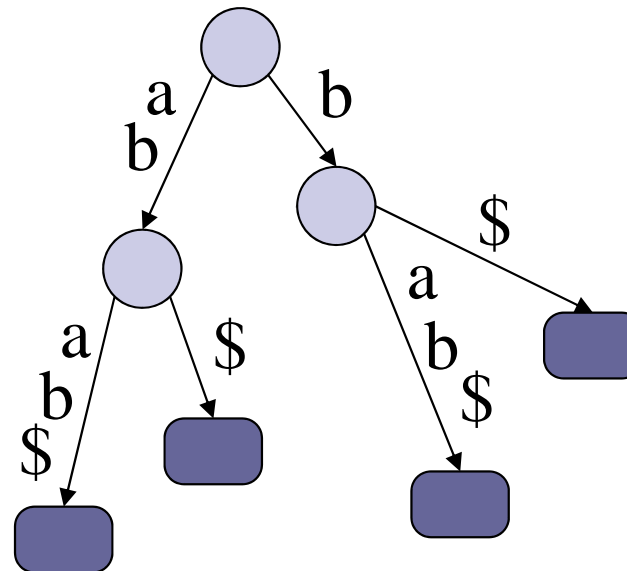


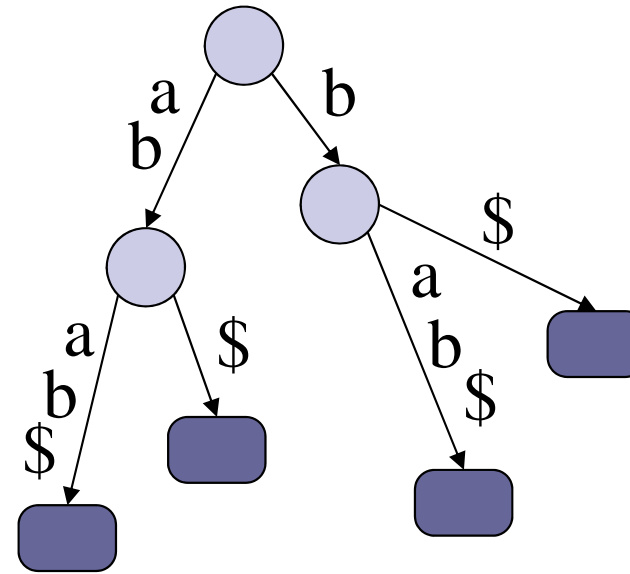
Put the suffix **ab\$** in



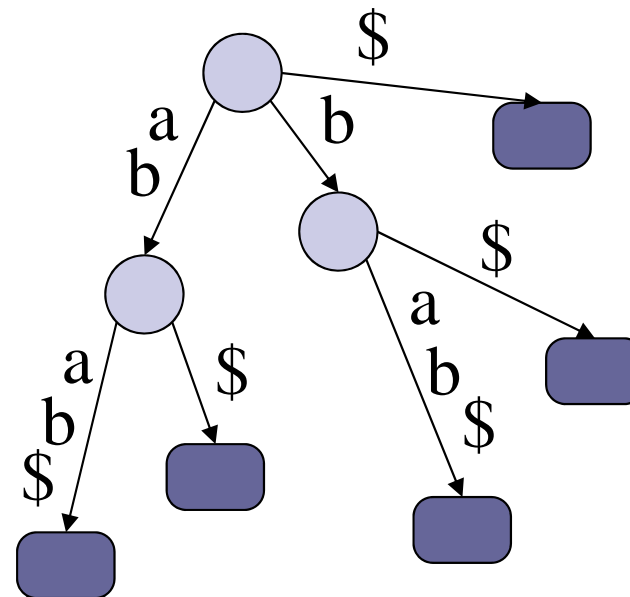


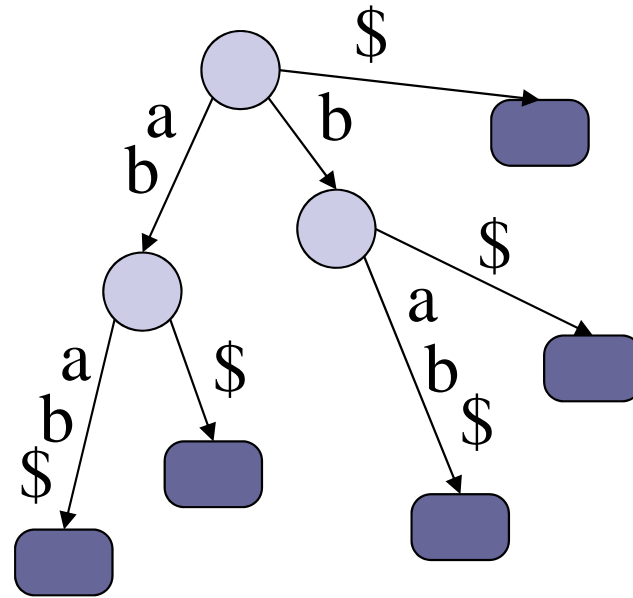
Put the suffix **b\$** in



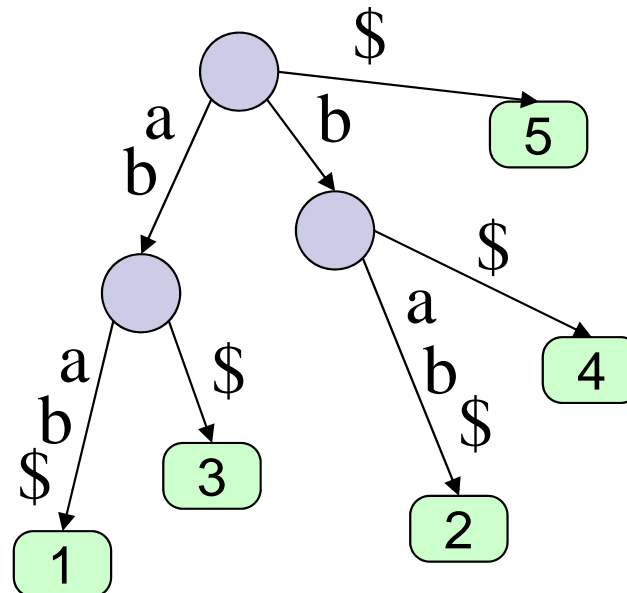


Put the suffix **\$** in





We will also label each leaf with the starting point of the corresponding suffix.



# Analysis

Takes  $O(m^2)$  time to build.

Can be done in  $O(m)$  time - we only sketch the proof here. See the CG class notes or Gusfield's book for the full details.



# Building STs in linear time: Ukkonen's algorithm



# History

- **Weiner's algorithm [FOCS, 1973]**
  - Called by Knuth "The algorithm of 1973"
  - First algorithm of linear time, but much space
- **McCreight's algorithm [JACM, 1976]**
  - Linear time and quadratic space
  - More readable
- **Ukkonen's algorithm [Algorithmica, 1995]**
  - Linear time algorithm and less space
  - This is what we will focus on
- ....



# Esko Ukkonen



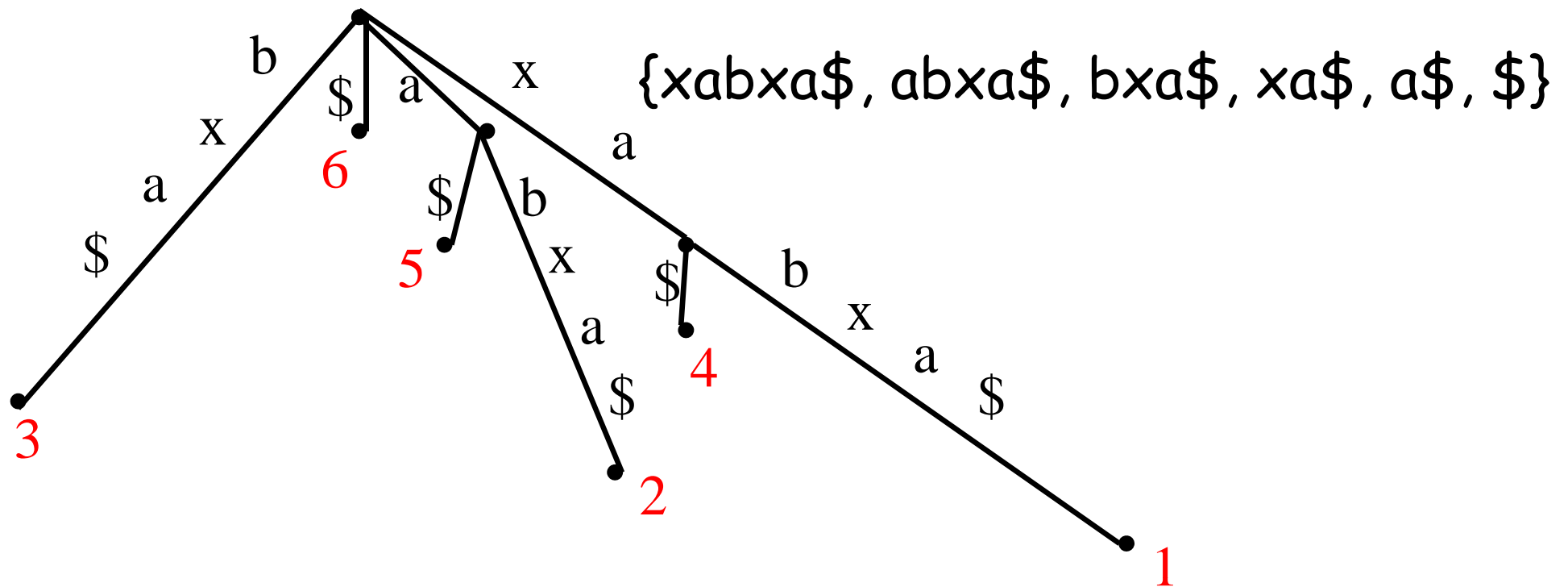
# Implicit Suffix Trees

- Ukkonen's alg constructs a sequence of implicit STs, the last of which is converted to a true ST of the given string.
- An **implicit suffix tree** for string  $S$  is a tree obtained from the suffix tree for  $S\$$  by
  - removing  $\$$  from all edge labels
  - removing any edge that now has no label
  - removing any node with only one child



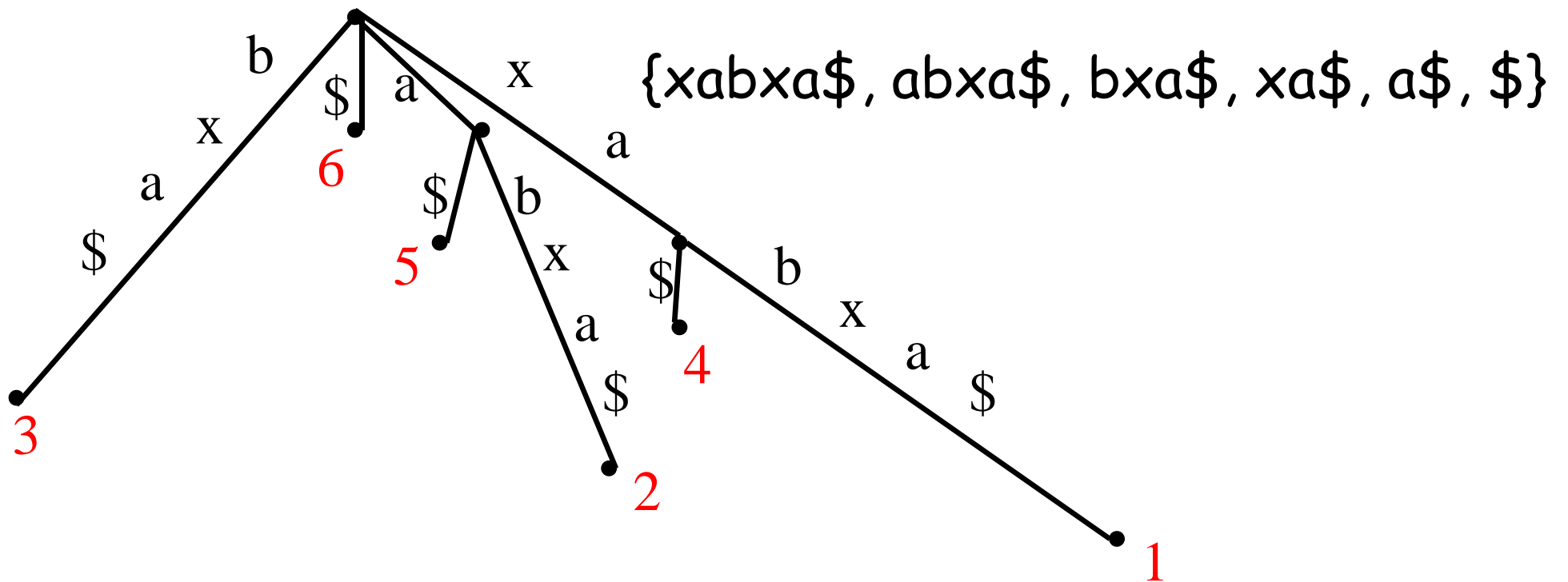
# Example: Construction of the Implicit ST

- The tree for  $xabxa\$$

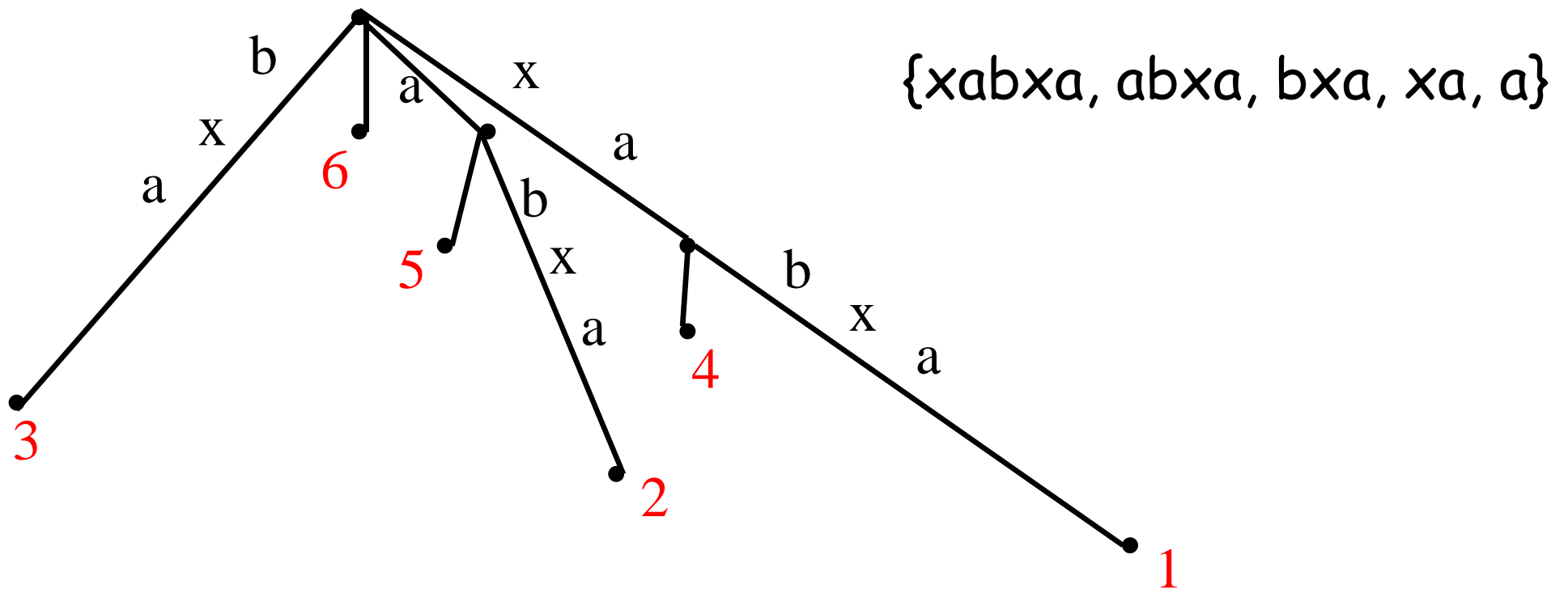


# Construction of the Implicit ST: Remove \$

## ■ Remove \$

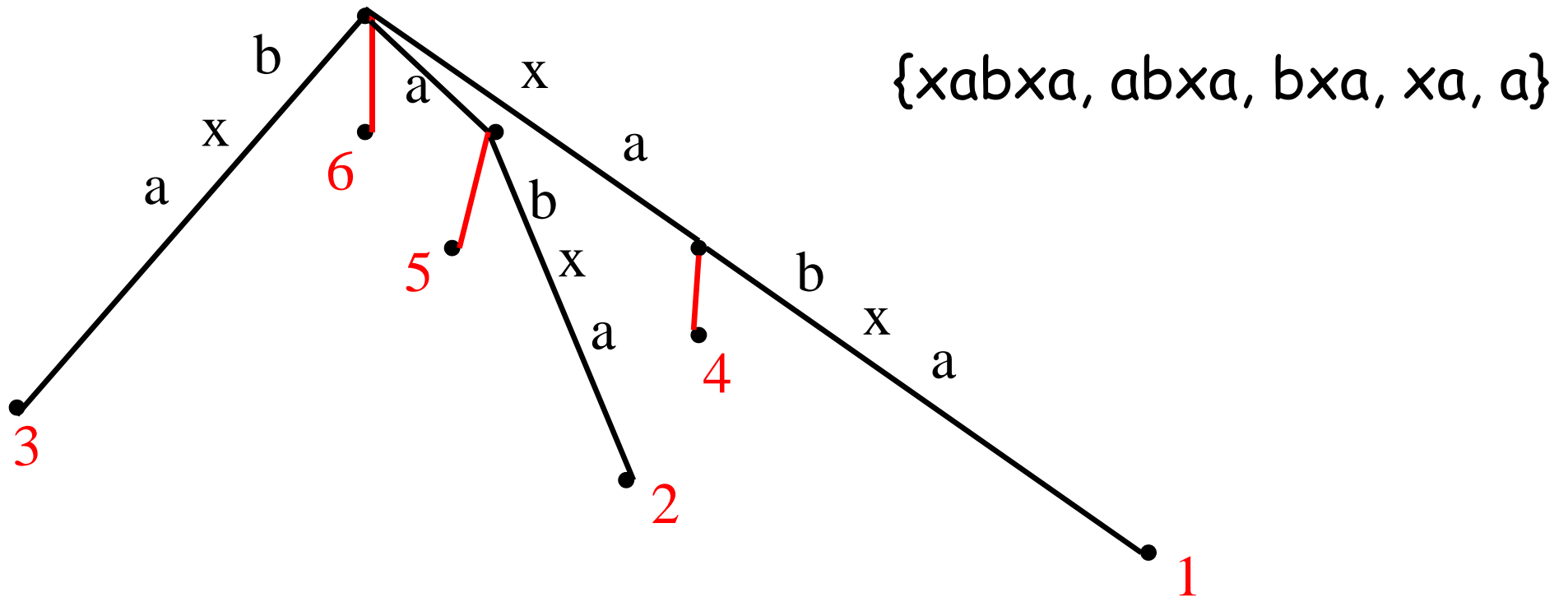


# Construction of the Implicit ST: After the Removal of \$

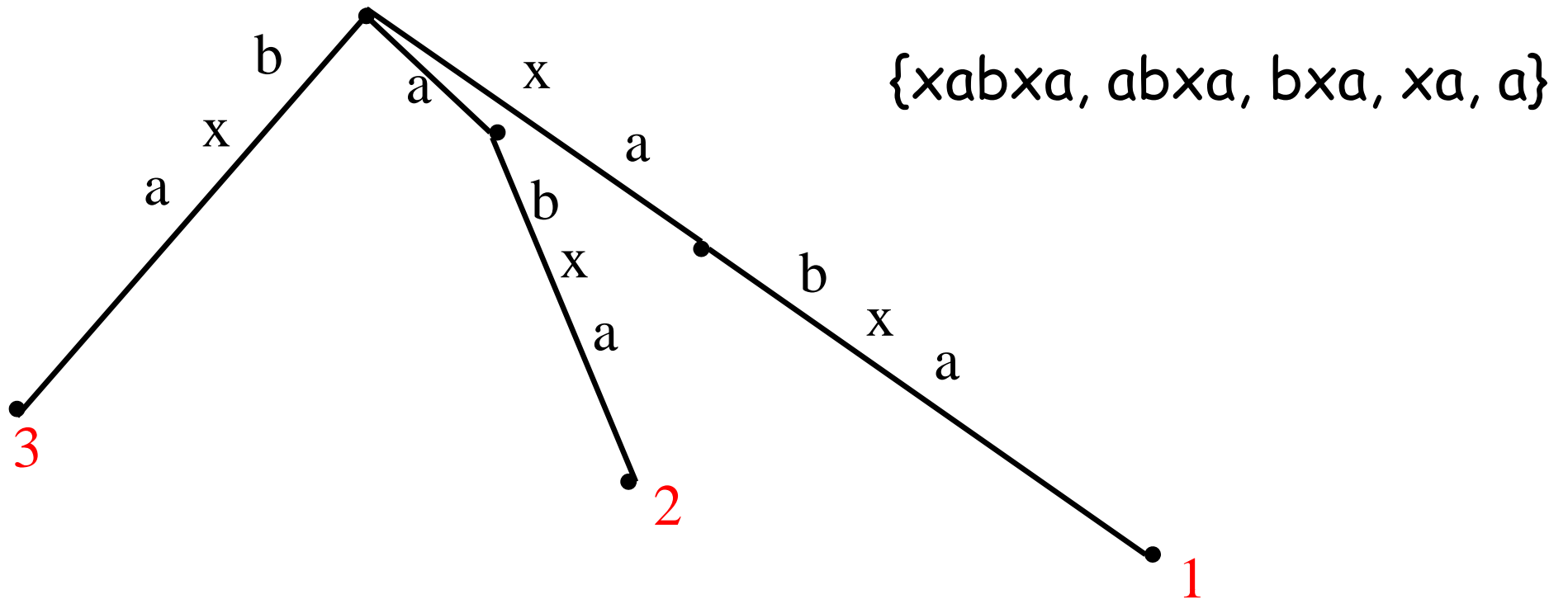


# Construction of the Implicit ST: Remove unlabeled edges

- Remove unlabeled edges

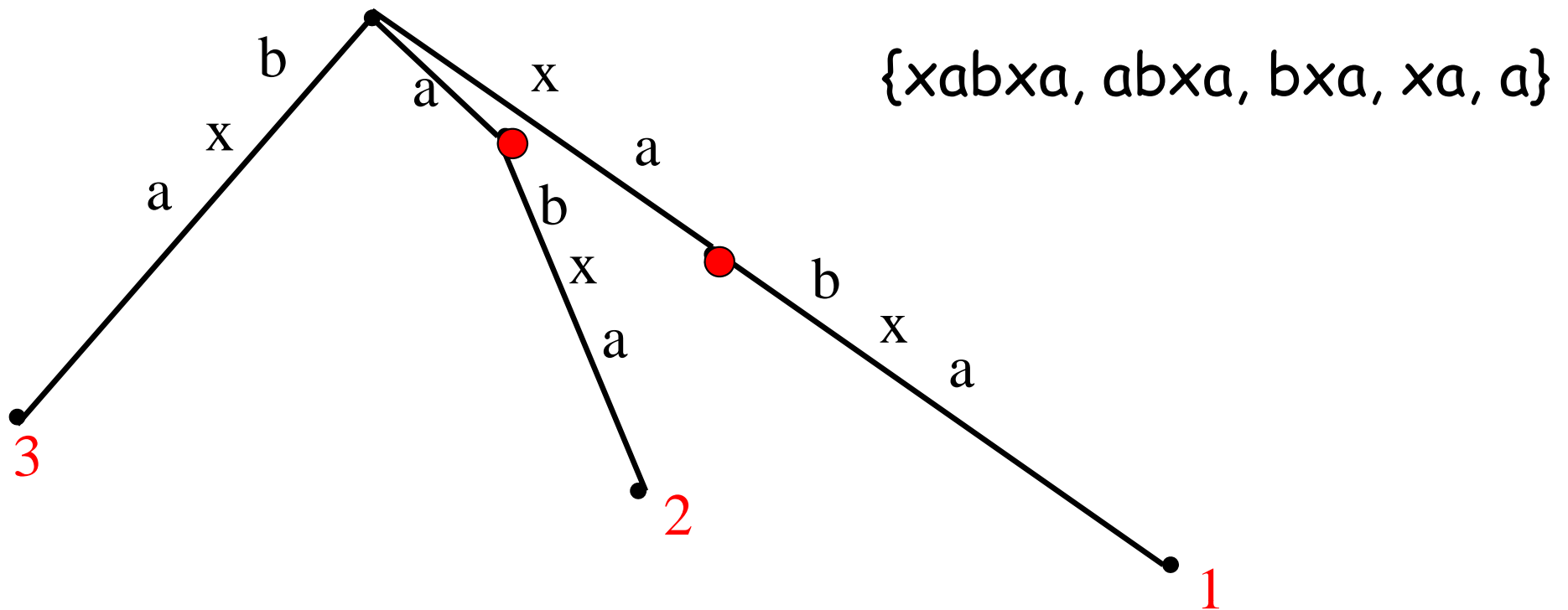


# Construction of the Implicit ST: After the Removal of Unlabeled Edges

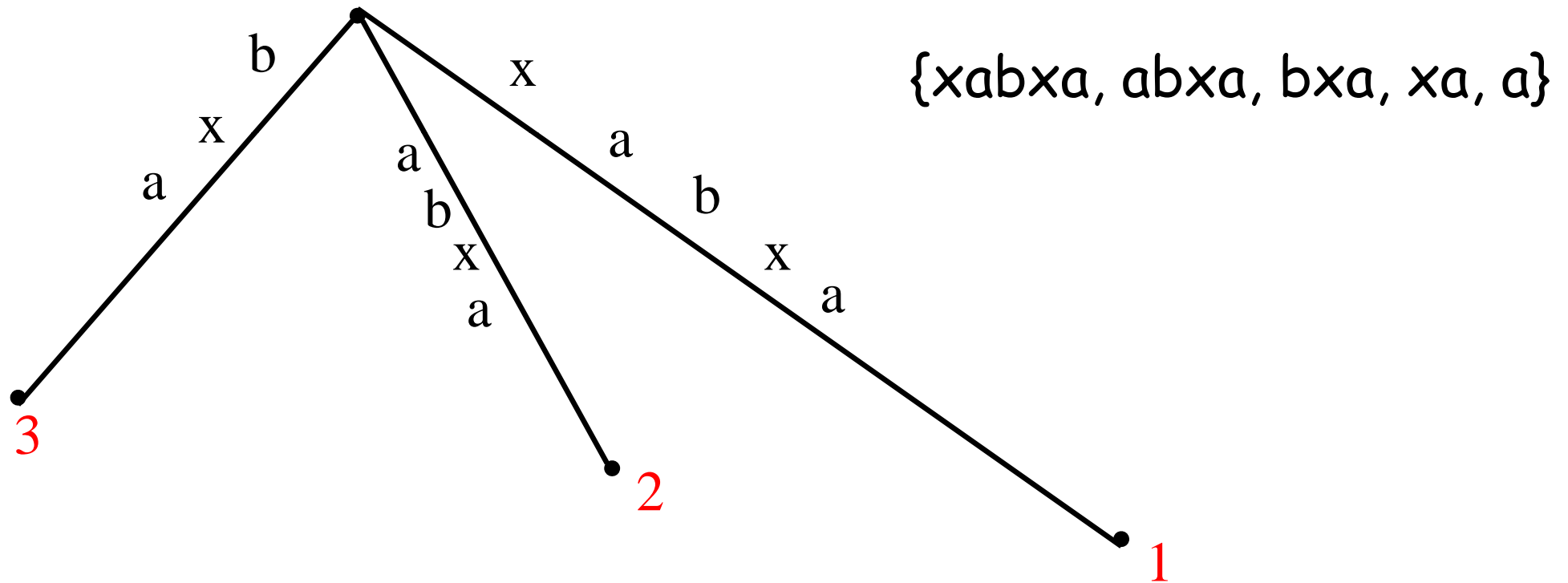


# Construction of the Implicit ST: Remove degree 1 nodes

- Remove internal nodes with only one child



# Construction of the Implicit ST: Final implicit tree



- Each suffix is in the tree, but may not end at a leaf.



# Implicit Suffix Trees (2)

- An implicit suffix tree for prefix  $S[1,i]$  of  $S$  is similarly defined based on the suffix tree for  $S[1,i]\$$ .
- $I_i$  = the implicit suffix tree for  $S[1,i]$ .



# Ukkonen's Algorithm (UA)

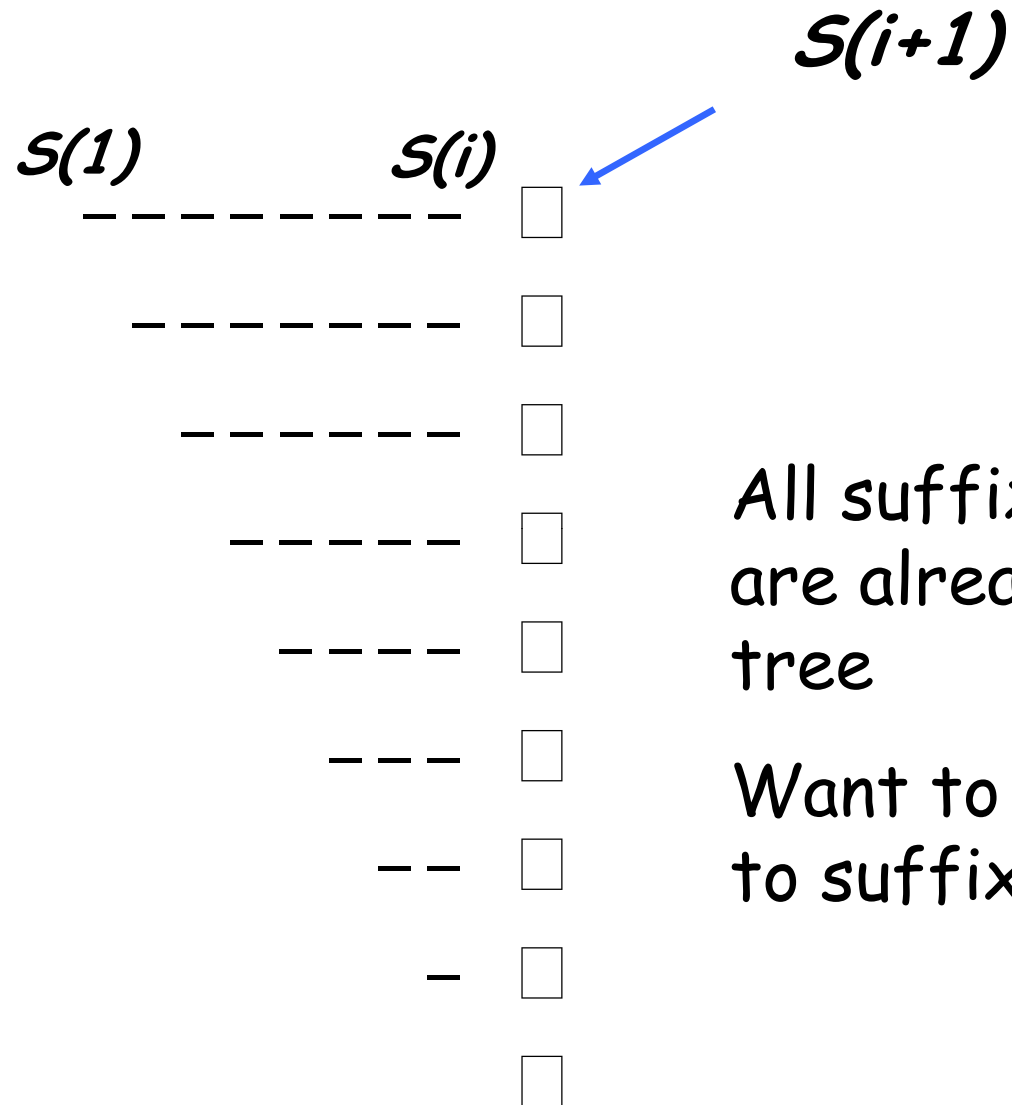
- $I_i$  is the implicit suffix tree of the string  $S[1, i]$
- Construct  $I_1$
- /\* Construct  $I_{i+1}$  from  $I_i$  \*/
- for  $i = 1$  to  $m-1$  do /\* generation  $i+1$  \*/
  - for  $j = 1$  to  $i+1$  do /\* extension  $j$  \*/
    - Find the end of the path  $p$  from the root whose label is  $S[j, i]$  in  $I_i$  and extend  $p$  with  $S(i+1)$  by suffix extension rules;
- Convert  $I_m$  into a suffix tree  $S$



# Example

- $S = xabxa\$$
- (*initialization step*)
  - $x$
- $(i = 1), i+1 = 2, S(i+1) = a$ 
  - extend  $x$  to  $xa$  ( $j = 1, S[1,1] = x$ )
  - $a$  ( $j = 2, S[2,1] = ""$ )
- $(i = 2), i+1 = 3, S(i+1) = b$ 
  - extend  $xa$  to  $xab$  ( $j = 1, S[1,2] = xa$ )
  - extend  $a$  to  $ab$  ( $j = 2, S[2,2] = a$ )
  - $b$  ( $j = 3, S[3,2] = ""$ )
- ...





All suffixes of  $S[1,i]$   
are already in the  
tree

Want to extend them  
to suffixes of  $S[1,i+1]$



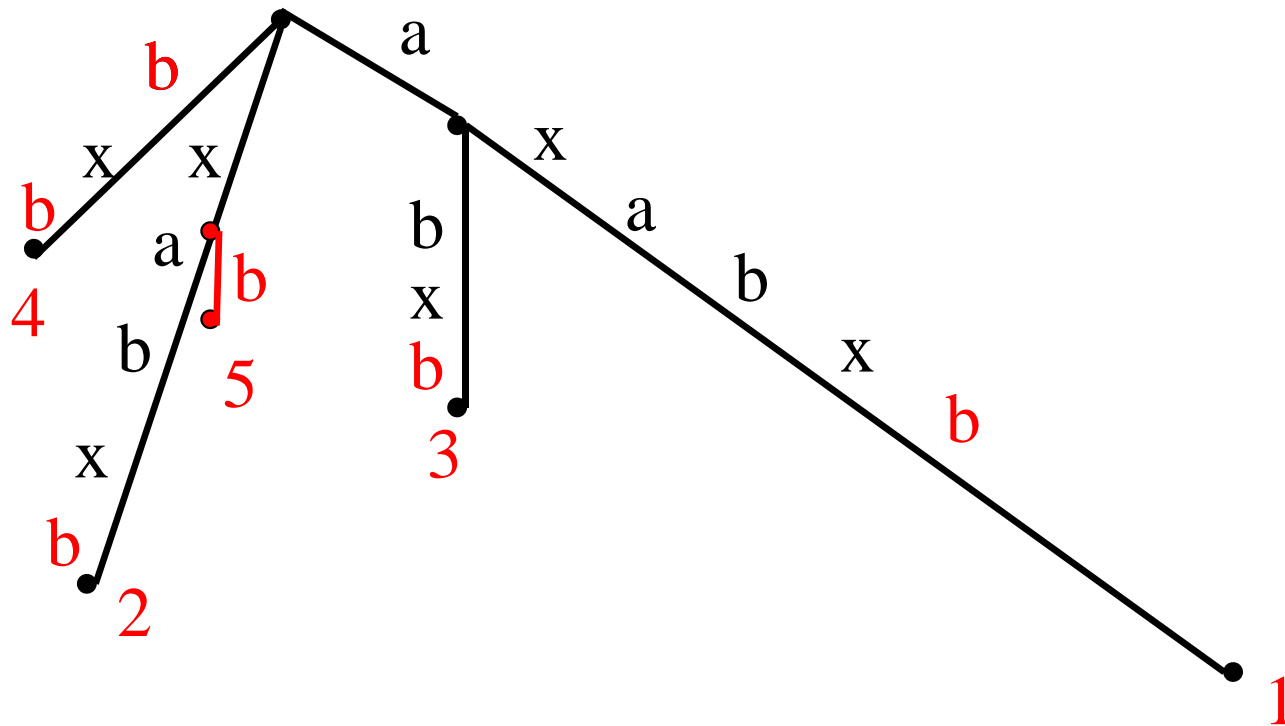
# Extension Rules

- **Goal:** extend each  $S[j,i]$  into  $S[j,i+1]$
- **Rule 1:**  $S[j,i]$  ends at a leaf
  - Add character  $S(i+1)$  to the end of the label on that leaf edge
- **Rule 2:**  $S[j,i]$  doesn't end at a leaf, and the following character is not  $S(i+1)$ 
  - Split a new leaf edge for character  $S(i+1)$
  - May need to create an internal node if  $S[j,i]$  ends in the middle of an edge
- **Rule 3:**  $S[j,i+1]$  is already in the tree
  - No update



# Example: Extension Rules

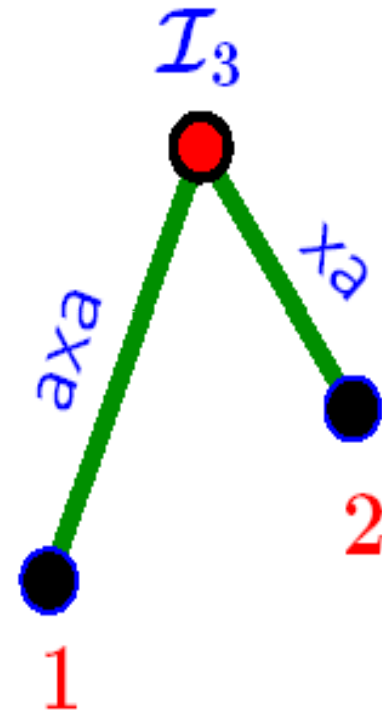
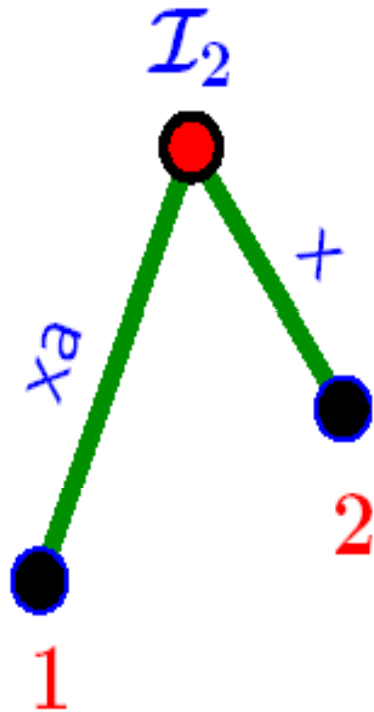
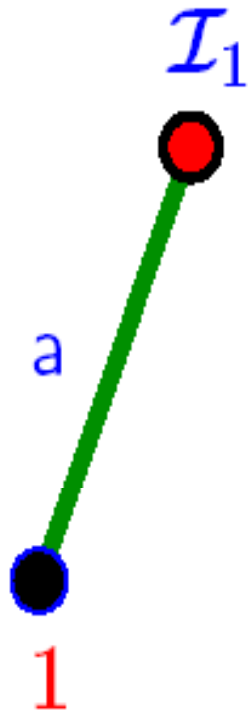
- Constructing the mplicit tree for  $axabxb$  from tree for  $axabx$



**Rule B:** attach a new node (and an interior node)



# UA for $axabxc$ (1)



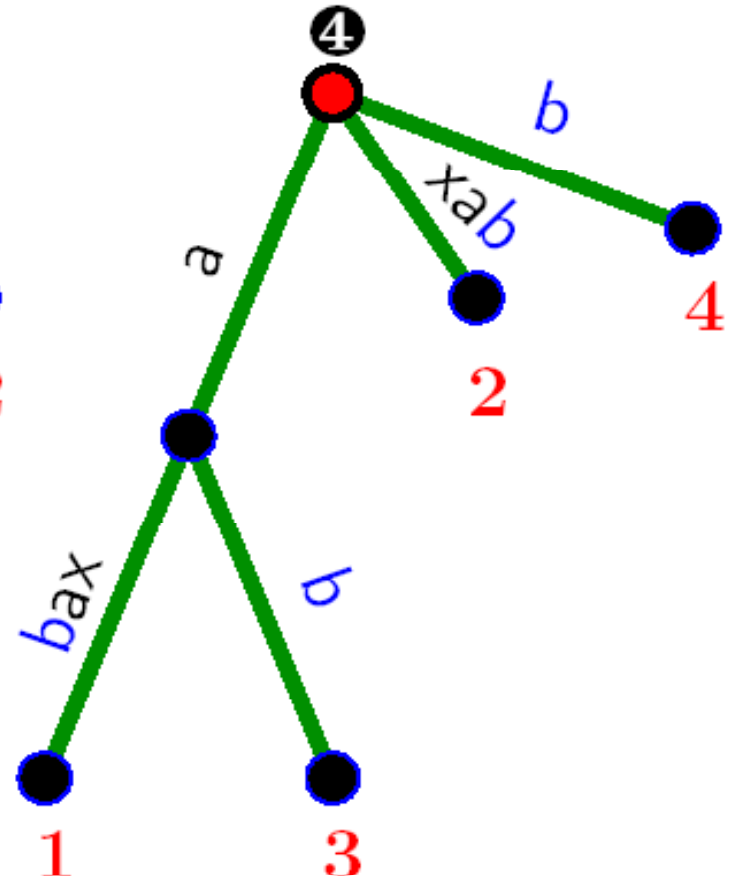
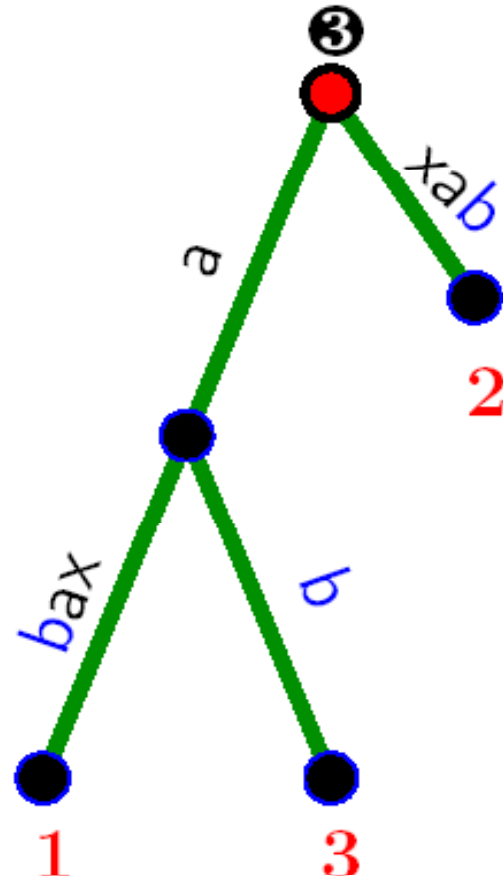
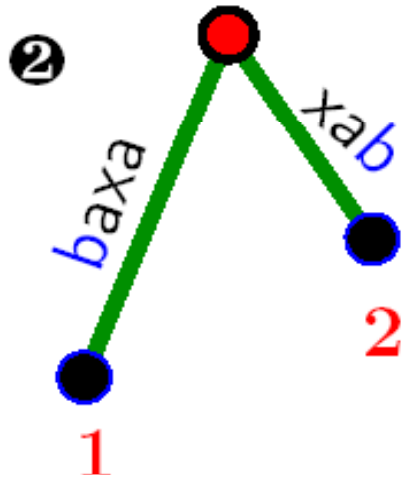
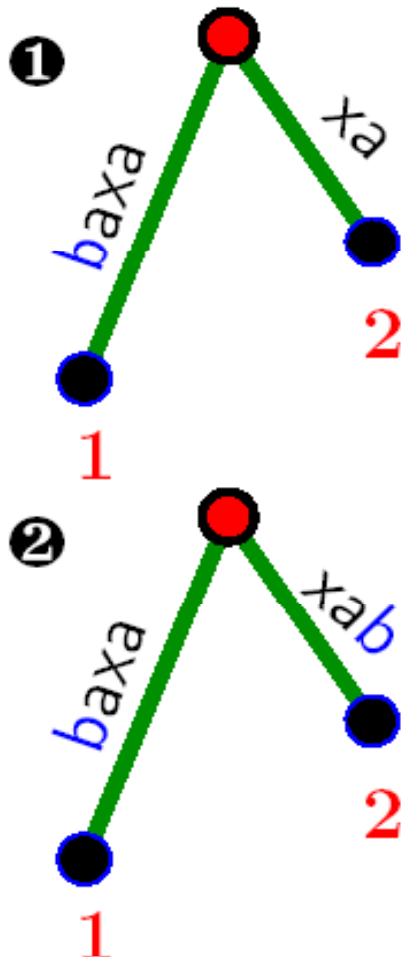
$S[1,3]=axa$		
E	$S(j,i)$	$S(i+1)$
1	ax	a
2	x	a
3		a



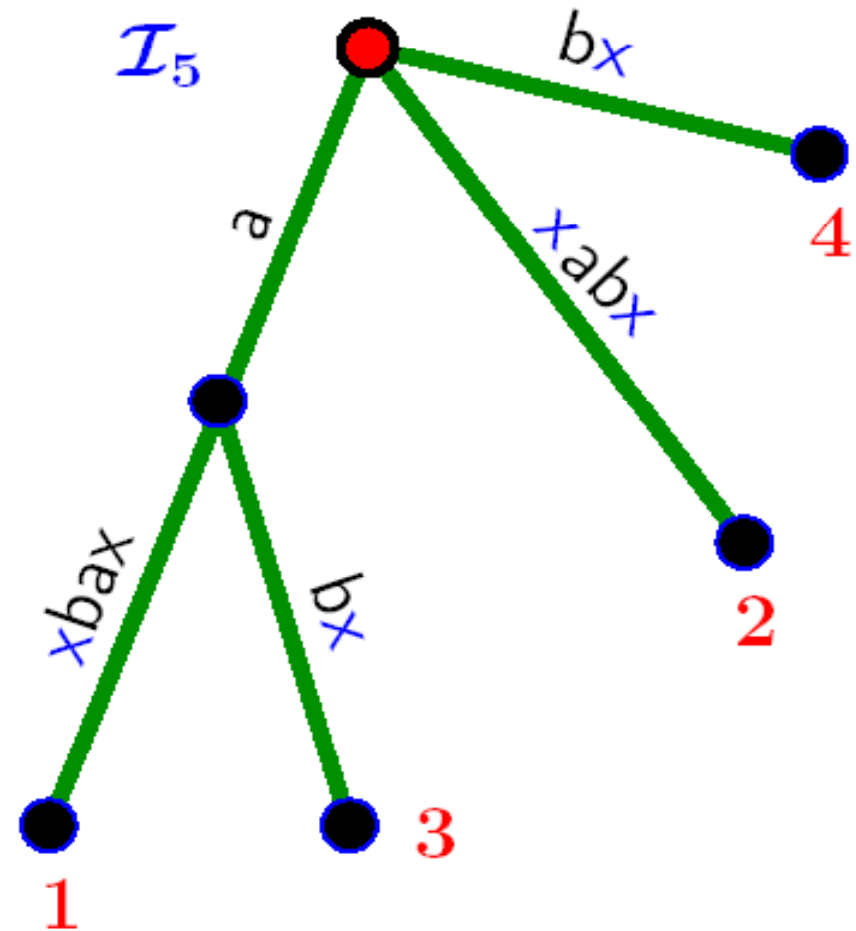
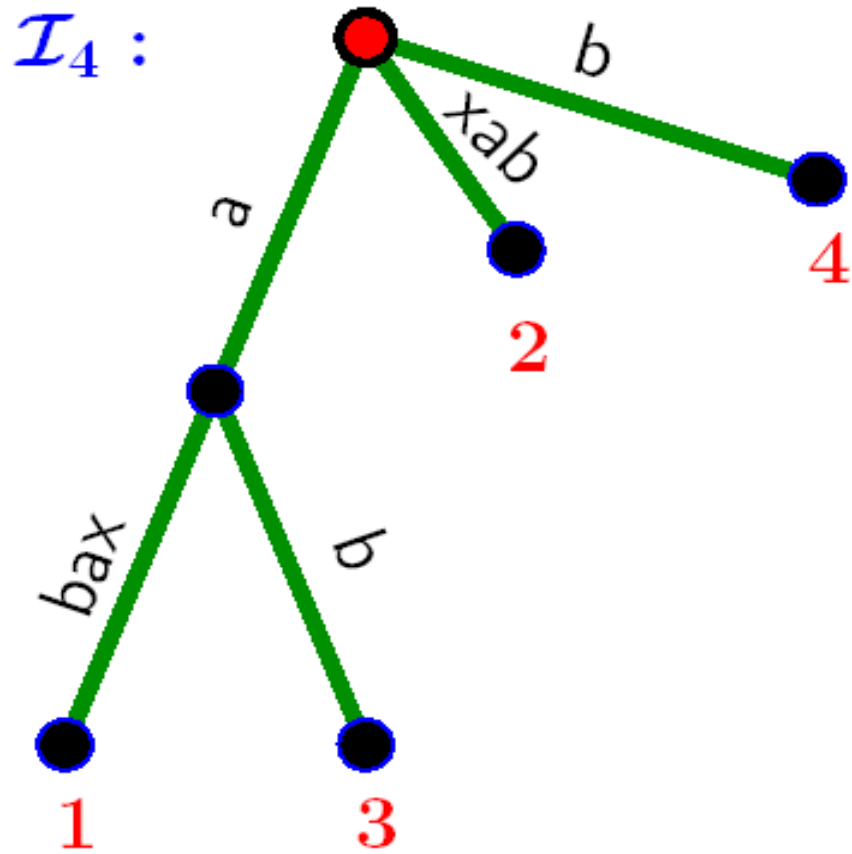
# UA for $axabxc$ (2)



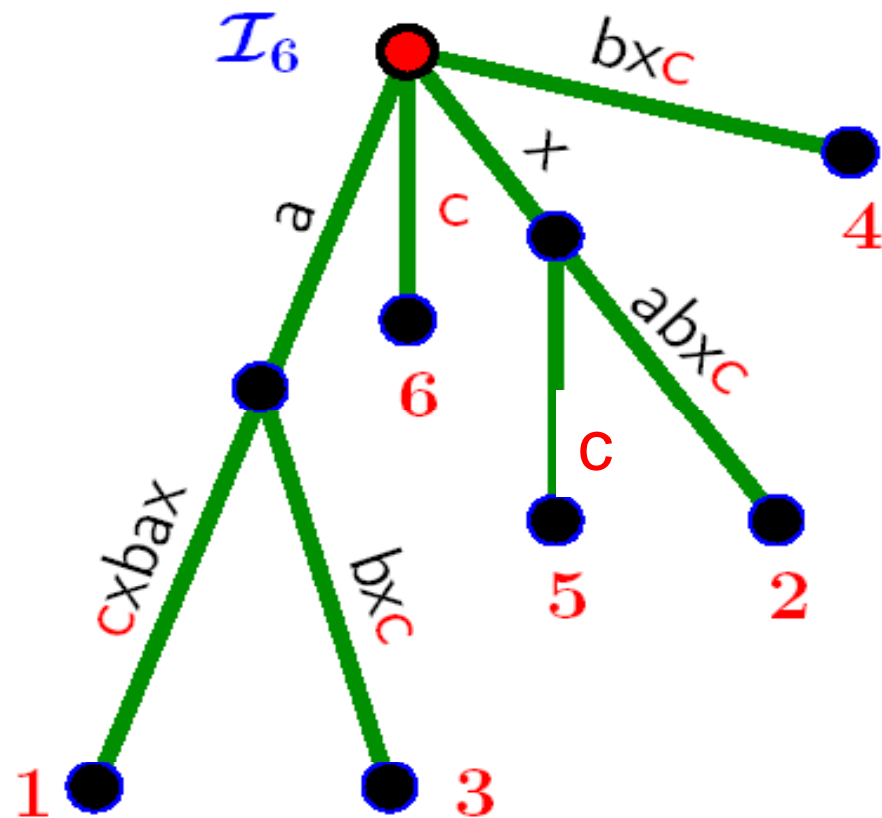
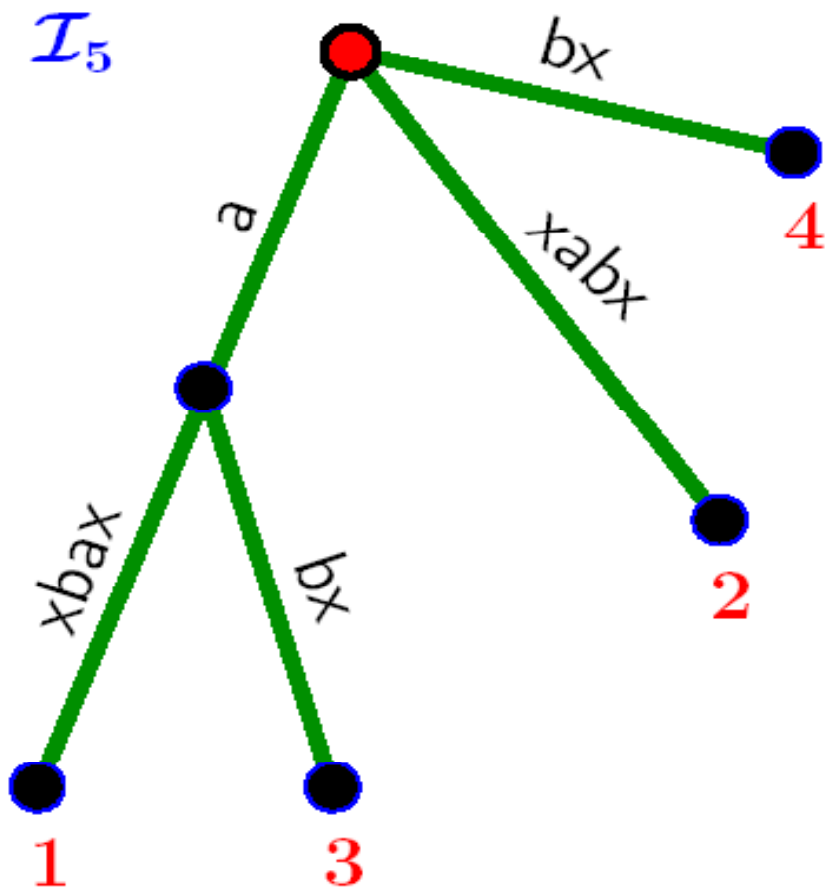
$\mathcal{I}_4$ :



# UA for $axabxc$ (3)



# UA for $axabxc$ (4)



# Observations

- Once  $S[j,i]$  is located in the tree, applying the extension rule takes only constant time
- Naive implementation: find the end of suffix  $S[j,i]$  in  $O(i-j)$  time by walking from the root of the current tree.  $\Rightarrow I_m$  is created in  $O(m^3)$  time.
- Making Ukkonen's algorithm run in  $O(m)$  time is achieved by a set of shortcuts:
  - Suffix links
  - Skip and count trick
  - Edge-label compression
  - A stopper
  - Once a leaf, always a leaf



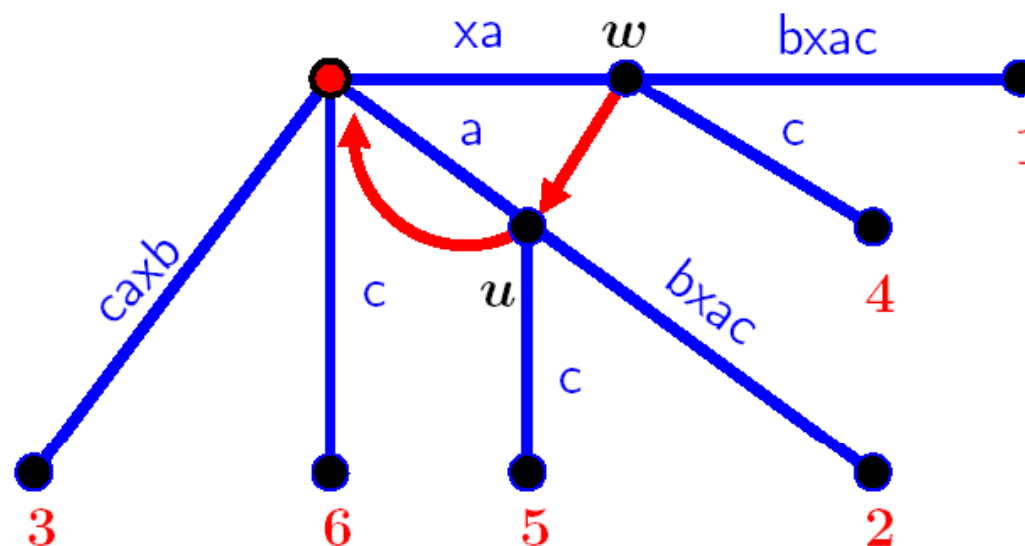
# Ukkonen's Algorithm (UA)

- $I_i$  is the implicit suffix tree of the string  $S[1, i]$
- Construct  $I_1$
- /\* Construct  $I_{i+1}$  from  $I_i$  \*/
- for  $i = 1$  to  $m-1$  do /\* generation  $i+1$  \*/
  - for  $j = 1$  to  $i+1$  do /\* extension  $j$  \*/
    - Find the end of the path  $p$  from the root whose label is  $S[j, i]$  in  $I_i$  and extend  $p$  with  $S(i+1)$  by suffix extension rules;
- Convert  $I_m$  into a suffix tree  $S$



# Suffix Links

- Consider the two strings  $\beta$  and  $x\beta$  (e.g.  $a$ ,  $xa$  in the example below).
- Suppose some internal node  $v$  of the tree is labeled with  $x\beta$  ( $x=\text{char}$ ,  $\beta=\text{string}$ , possibly  $\emptyset$ ) and another node  $s(v)$  in the tree is labeled with  $\beta$
- The edge  $(v, s(v))$  is called the **suffix link** of  $v$
- Do all internal nodes have suffix links?
- (the root is not considered an internal node)



path label of  
node  $v$ :  
concatenation  
of the strings  
labeling edges  
from root to  $v$





# Suffix Link Lemma

If a new internal node  $v$  with path-label  $x\beta$  is added to the current tree in extension  $j$  of some generation  $i+1$ , then either

- the path labeled  $\beta$  already ends at an internal node of the tree, or
- the internal node labeled  $\beta$  will be created in extension  $j+1$  in the same generation  $i+1$ , or
- string  $\beta$  is empty and  $s(v)$  is the root



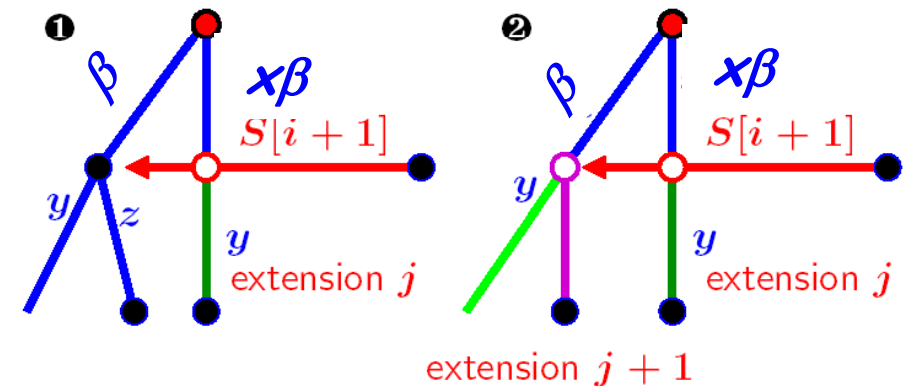
# Suffix Link Lemma

If a new internal node  $v$  with path-label  $x\beta$  is added to the current tree in extension  $j$  of some generation  $i+1$ , then either

- the path labeled  $\beta$  already ends at an internal node of the tree, or
- the internal node labeled  $\beta$  will be created in extension  $j+1$  in the same generation

Pf: A new internal node is created only by **extension rule 2**

- In extension  $j$  the path labeled  $x\beta..$  continued with some  $y \neq S(i+1)$



$\Rightarrow$  In extension  $j+1$ ,  $\exists$  a path  $p$  labeled  $\beta..$

- $p$  continues with  $y$  only  $\Rightarrow$  ext. rule 2 will create a node  $s(v)$  at the end of the path  $\beta..$
- $p$  continues with two different chars.  $\Rightarrow s(v)$  already exists.



# Corollaries

- Every internal node of an implicit suffix tree has a suffix link from it by the end of the next extension
  - Proof by the lemma, using induction.
- In any implicit suffix tree  $I_i$ , if internal node  $v$  has path label  $x\beta$ , then there is a node  $s(v)$  of  $I_i$  with path label  $\beta$

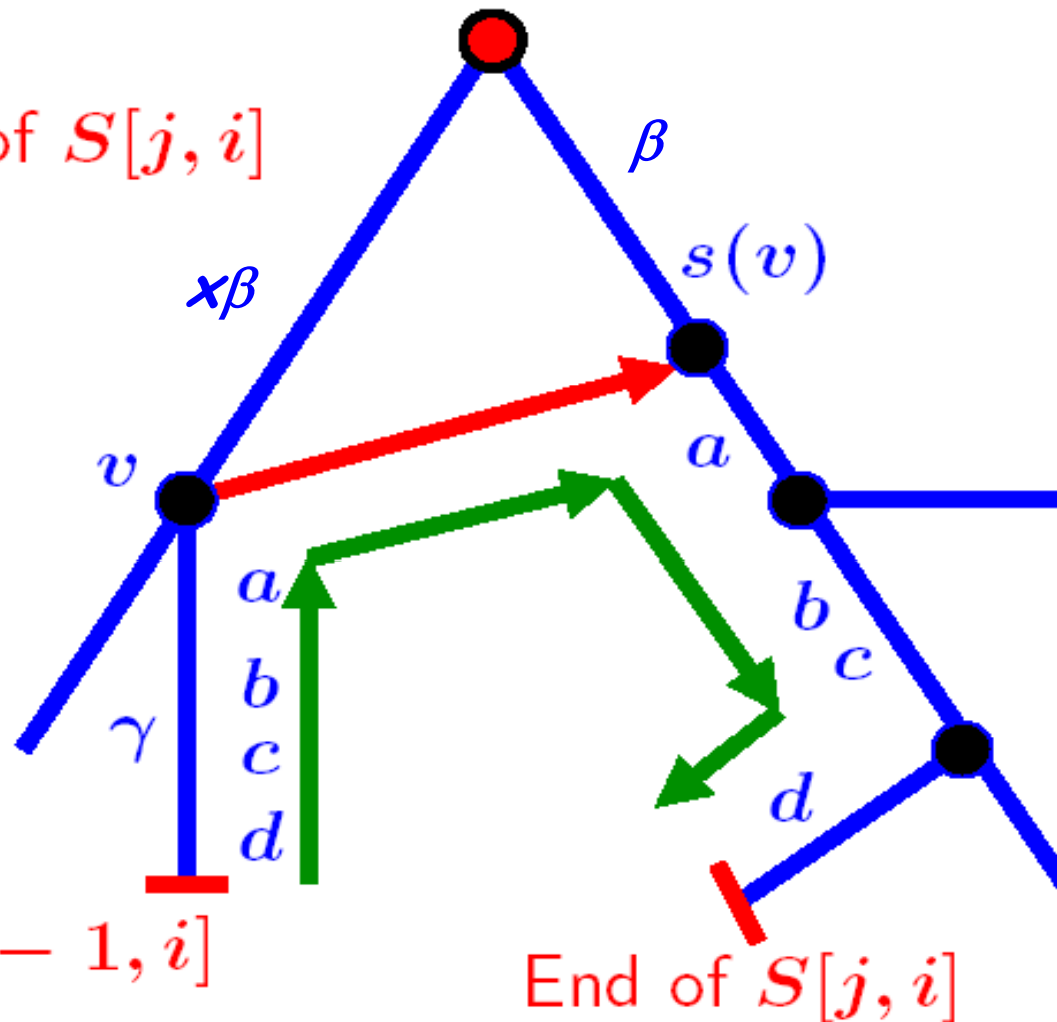


# Building $\mathcal{I}_{i+1}$ with suffix links - 1

- Goal: in extension  $j$  of generation  $i+1$ , find  $S[j, i]$  in the tree and extend to  $S[j, i+1]$ ; add suffix link if needed

Extension  $j$ :

find the end of  $S[j, i]$



End of  $S[j - 1, i]$

End of  $S[j, i]$



# Building $I_{i+1}$ with suffix links - 2

- Goal: in extension  $j$  of generation  $i+1$ , find  $S[j,i]$  in the tree and extend to  $S[j,i+1]$ ; add suffix link if needed
- $S[1,i]$  must end at a leaf since it is the longest string in the implicit tree  $I_i$ 
  - Keep pointer to leaf of full string; extend to  $S[1,i+1]$  (rule 1)
- $S[2,i] = \beta$ ,  $S[1,i] = x\beta$ ; let  $(v, l)$  the edge entering leaf  $l$ :
  - If  $v$  is the root, descend from the root to find  $\beta$
  - Otherwise,  $v$  is internal. Go to  $s(v)$  and descend to find rest of  $\beta$

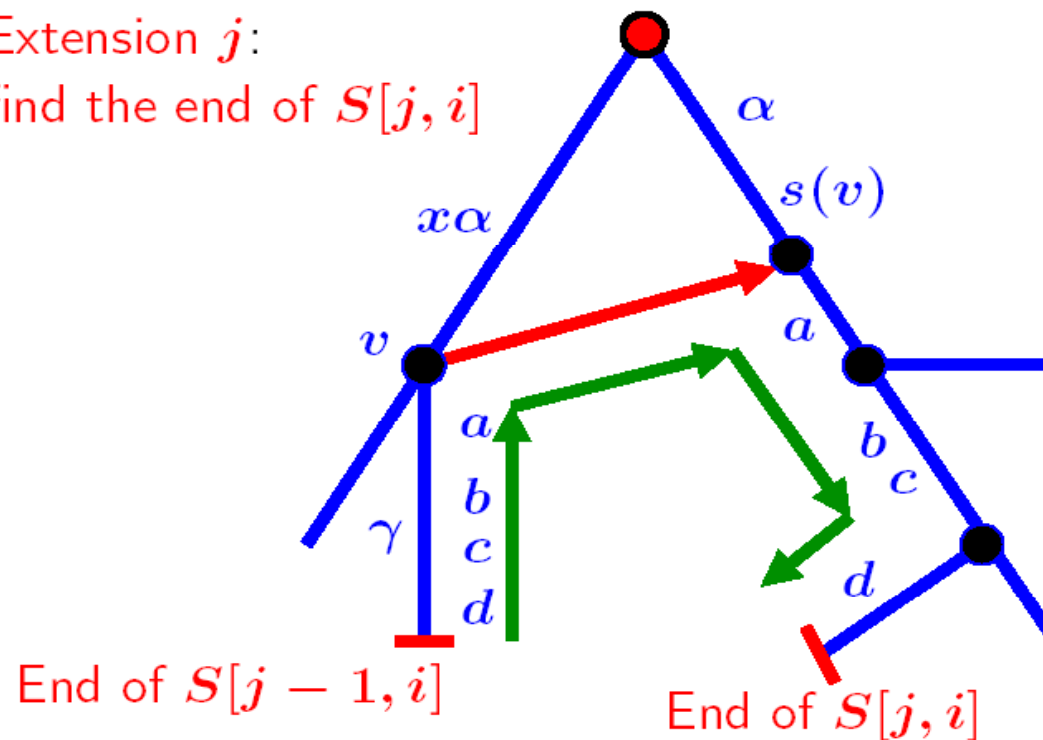


# Building $\mathcal{I}_{i+1}$ with suffix links - 3

- In general: find first node  $v$  at or above  $S[j-1, i]$  that has s.l. or is root; Let  $\gamma$  = string between  $v$  and end of  $S[j-1, i]$ 
  - If  $v$  is internal, go to  $s(v)$  and descend following the path of  $\gamma$
  - If  $v$  is the root, descend from the root to find  $S[j, i]$
  - Extend to  $S[j, i]S(i+1)$
  - If new internal node  $w$  was created in extension  $j-1$ , by the lemma  $S[j, i+1]$  ends in  $s(w)$  => create the suffix link from  $w$  to  $s(w)$ .

Extension  $j$ :

find the end of  $S[j, i]$



# Skip and Count Trick - (1)

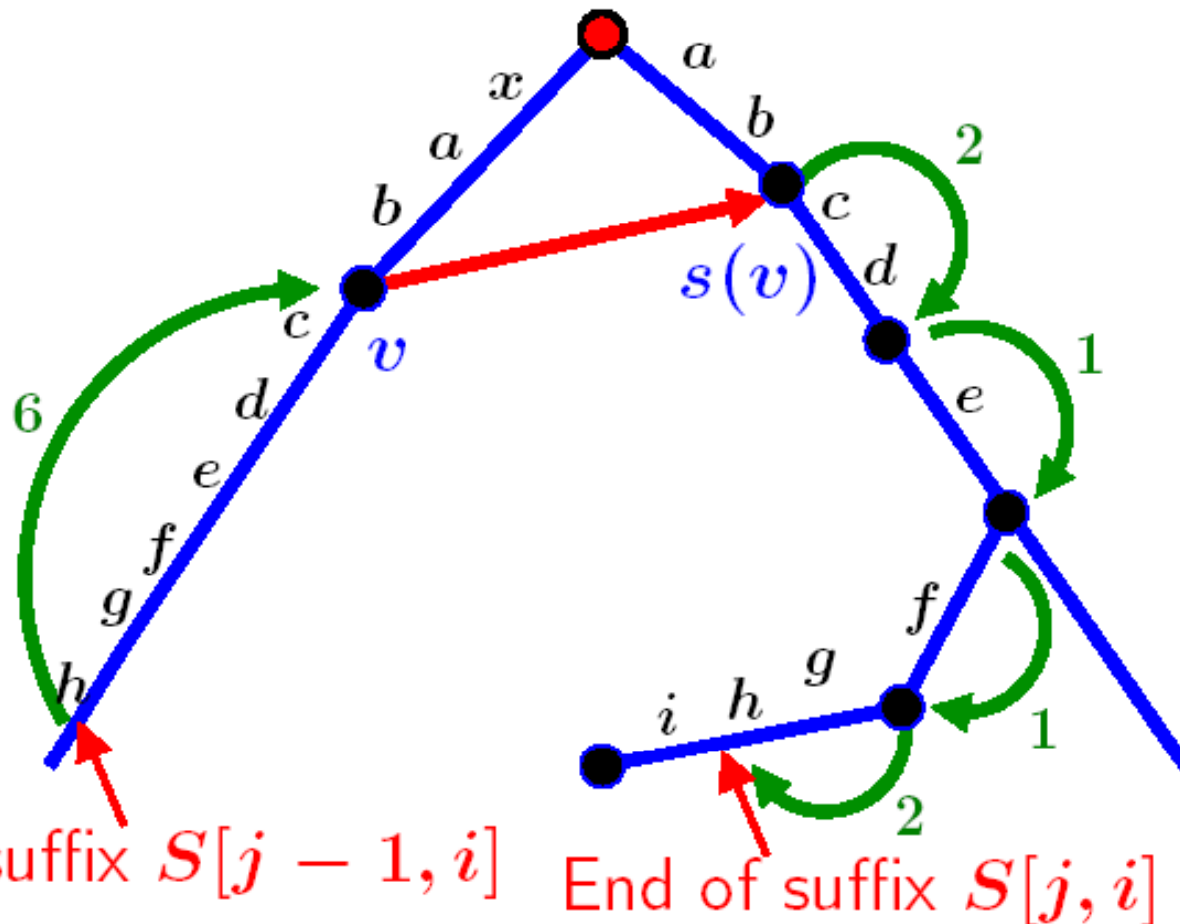
- **Problem:** Moving down from  $s(v)$ , directly implemented, takes time proportional to  $|\gamma|$
- **Solution:** To make running time proportional to the number of nodes in the path searched



# Skip and Count Trick - (2)

- counter=0; On each step from  $s(v)$ , find right edge below, add no. of chars on it to counter and if still  $< |\gamma|$  skip to child
- After 4 skips, the end of  $S[j, i]$  is found.

Can show: with skip & count trick, any generation of Ukkonen's algorithm takes  $\alpha(m)$  time



# Interim conclusion

- Ukkonen's Algorithm can be implemented in  $O(m^2)$  time

A few more smart tricks and we reach  $O(m)$  [see scribe]



# Implementation Issues - (1)

- When the size of the alphabet grows:
  - For large trees suffix links allow an algorithm to move quickly from one part of the tree to another. This is good for **worst-case time bounds**, but bad if the **tree isn't entirely in memory**.
  - Thus, implementing ST to reduce practical space use can be a serious concern.
- The main design issues are how to represent and search the branches out of the nodes of the tree.
- A practical design must balance between constraints of space and need for speed



# Implementation Issues - (3)

- When  $m$  and  $\Sigma$  are large enough, a good design is probably a mixture of the above choices.
  - Nodes near the root of the tree tend to have the most children, so **arrays** are sensible choice at those nodes.
  - (when there are  $k$  very dense levels - use a lookup table of all  $k$ -tuples with pointers to the roots of the corresponding subtrees)
  - For nodes in the middle of a suffix tree, **hashing** or **balanced** trees may be the best choice.



# Implementation Issues - (2)

- basic choices to represent branches:
  - An **array** of size  $\Theta(|\Sigma|)$  at each non-leaf node  $v$
  - A **linked list** at node  $v$  of characters that appear at the beginning of the edge-labels out of  $v$ .
    - If kept in *sorted order* it reduces the average time to search for a given character
    - In the worst case it, adds time  $|\Sigma|$  to every node operation. If the number of children of  $v$  is large, then little space is saved over the **array** while noticeably degrading performance
  - A **balanced tree** implements the **list** at node  $v$ 
    - Additions and searches take  $O(\log k)$  time and  $O(k)$  space, where  $k$  is the number of children of  $v$ . This alternative makes sense only when  $k$  is fairly large.
  - A **hashing scheme**. The challenge is to find a scheme balancing space with speed. For large trees and alphabets hashing is very attractive at least for some of the nodes



# Applications of Suffix Trees



# What can we do with it ?

## Exact string matching:

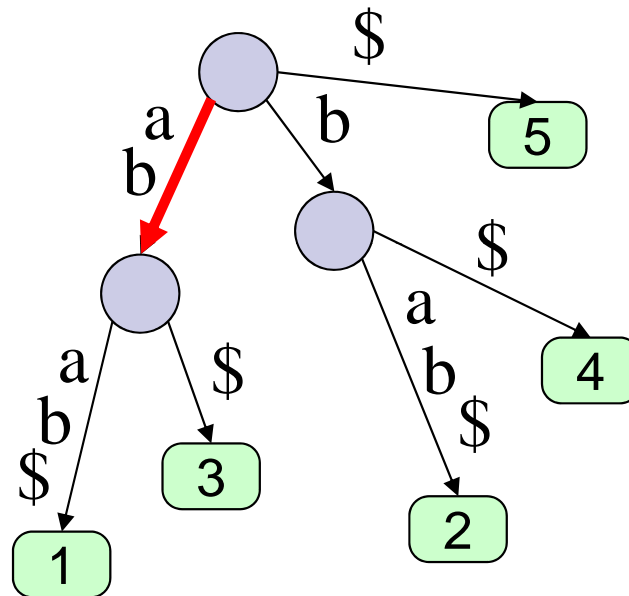
Given a Text  $T$ ,  $|T| = n$ , preprocess it such that when a pattern  $P$ ,  $|P|=m$ , arrives we can quickly decide if it occurs in  $T$ .

We may also want to find all occurrences of  $P$  in  $T$



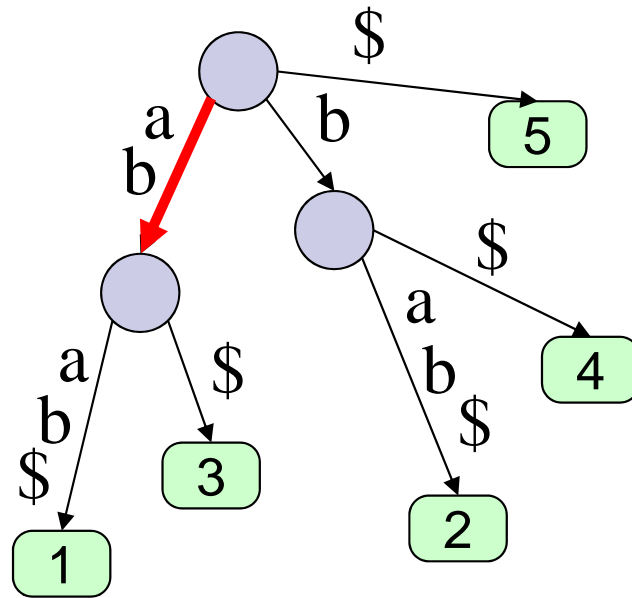
# Exact string matching

In preprocessing we just build a suffix tree in  $O(m)$  time



Given a pattern  $P = ab$  we traverse the tree according to the pattern.





If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all  $k$  occurrences in  $O(n+k)$  time



# Generalized suffix tree

Given a set of strings  $S$ , a **generalized suffix tree** of  $S$  is a compressed trie of all suffixes of  $s \in S$

To make these suffixes prefix-free we add a special char, say  $\$$ , at the end of  $s$

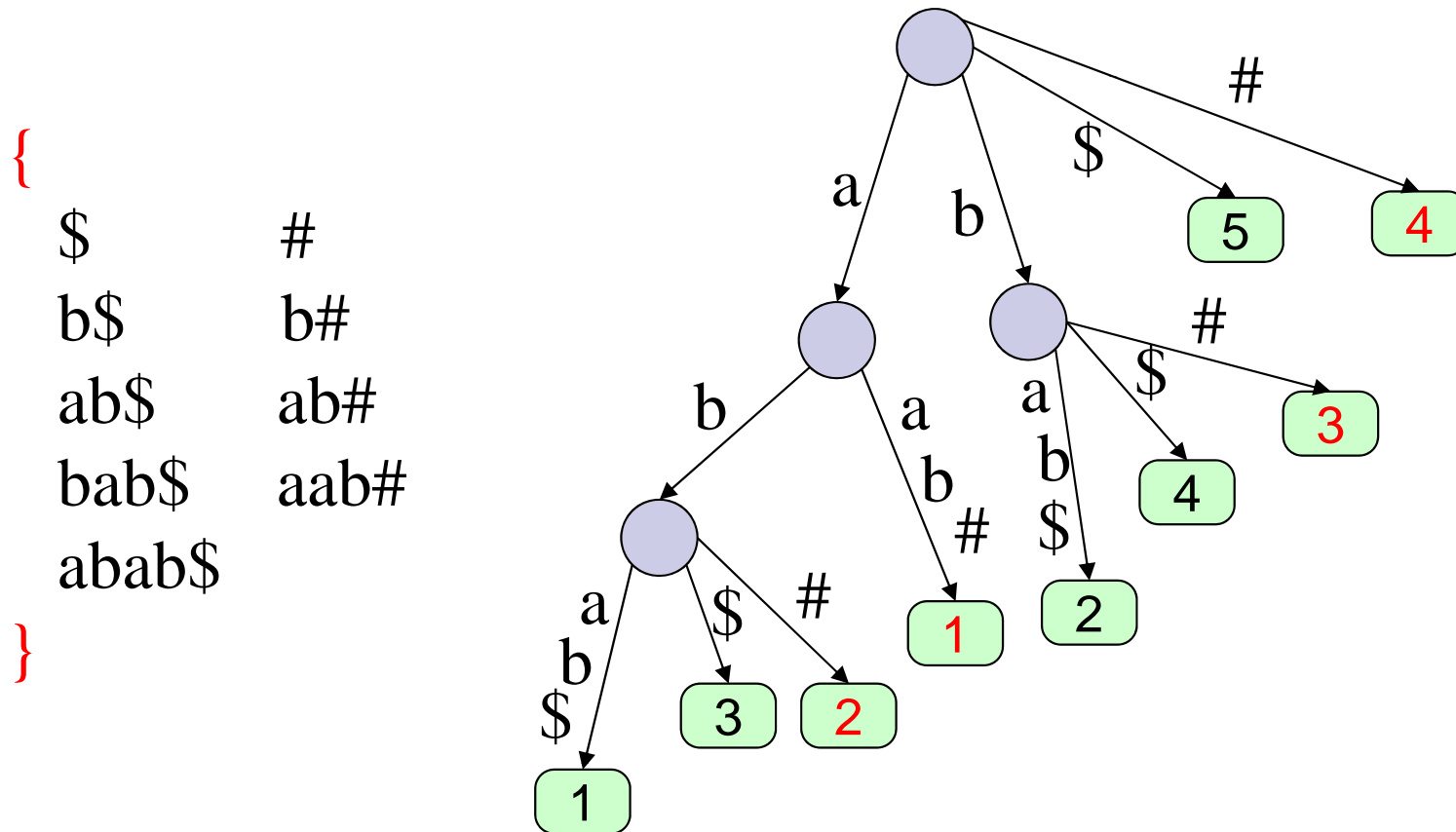
To associate each suffix with a unique string in  $S$  add a different special char  $\$s$  to each  $s$



# Generalized suffix tree (Example)

Let  $s_1=abab$  and  $s_2=aab$

a generalized suffix tree for  $s_1$  and  $s_2$  :



# So what can we do with it ?

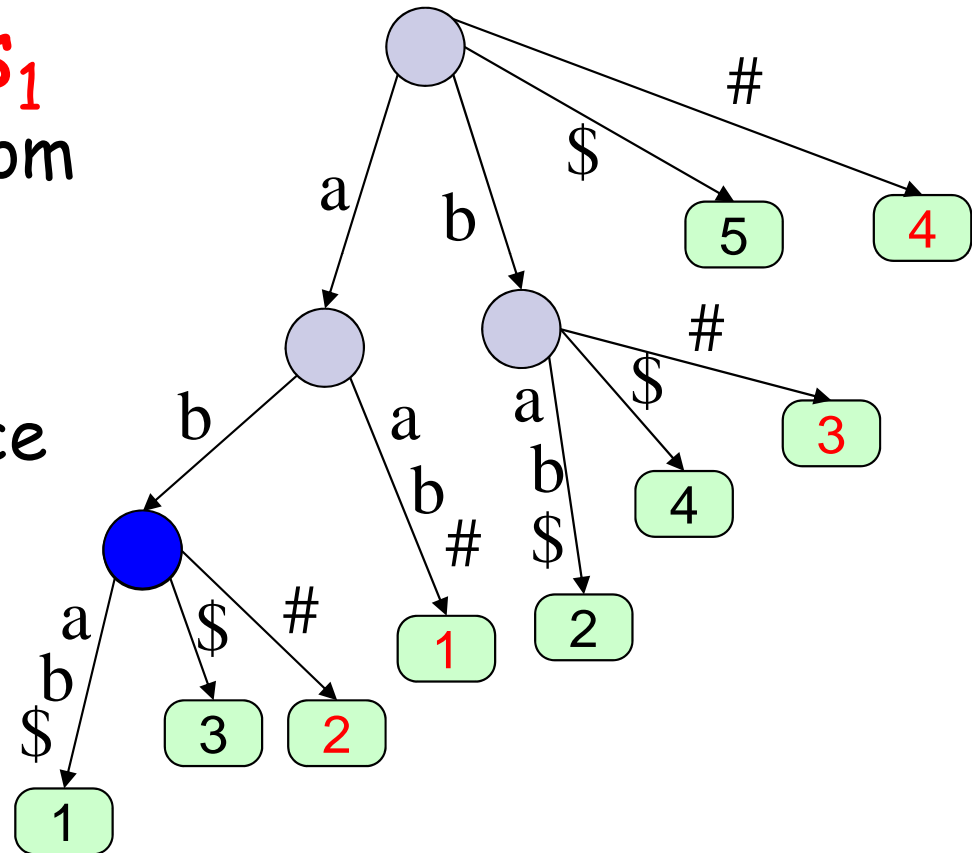
## Matching a pattern against a database of strings



# Longest common substring (of two strings)

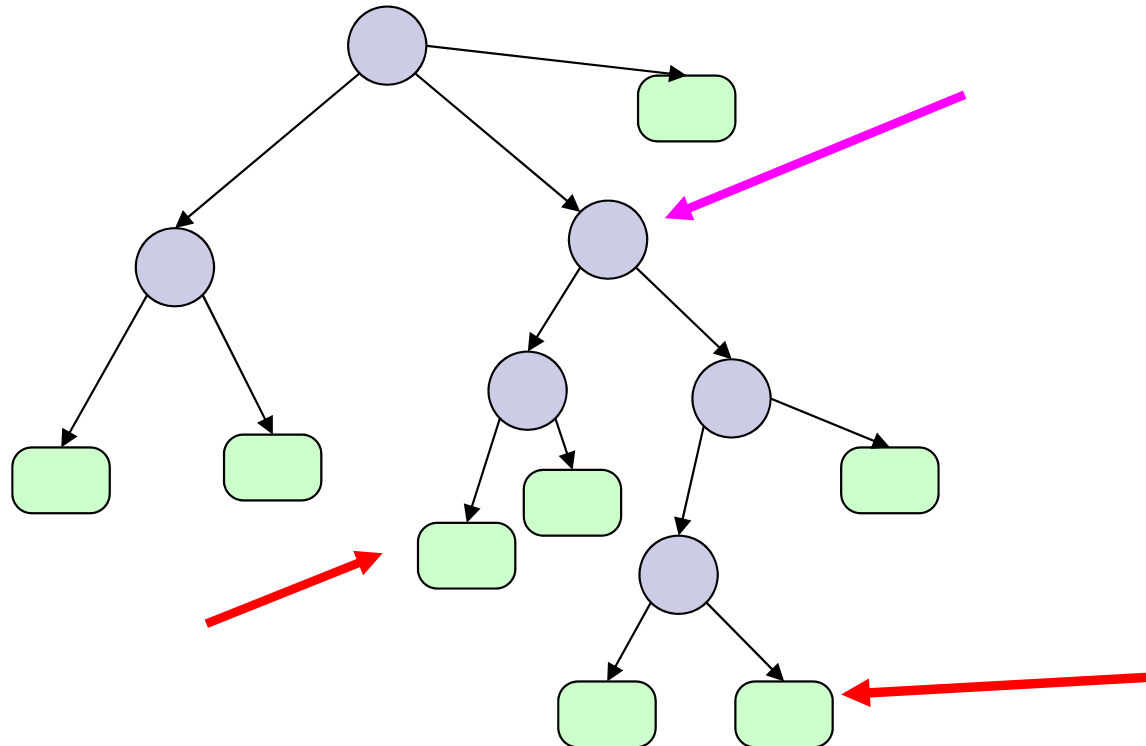
Every node with a leaf descendant from string  $S_1$  and a leaf descendant from string  $S_2$  represents a maximal common substring and vice versa.

Find such node with largest "label depth"



# Lowest common ancestors

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it





# Finding maximal palindromes

- A palindrome: caabaac, cbaabc
- Want to find all maximal palindromes in a string  $s$

Let  $s = cba|aba$

The maximal palindrome with center between  $i-1$  and  $i$  is the LCP of the suffix at position  $i$  of  $s$  and the suffix at position  $m-i+1$  of  $s^r$



# Maximal palindromes algorithm

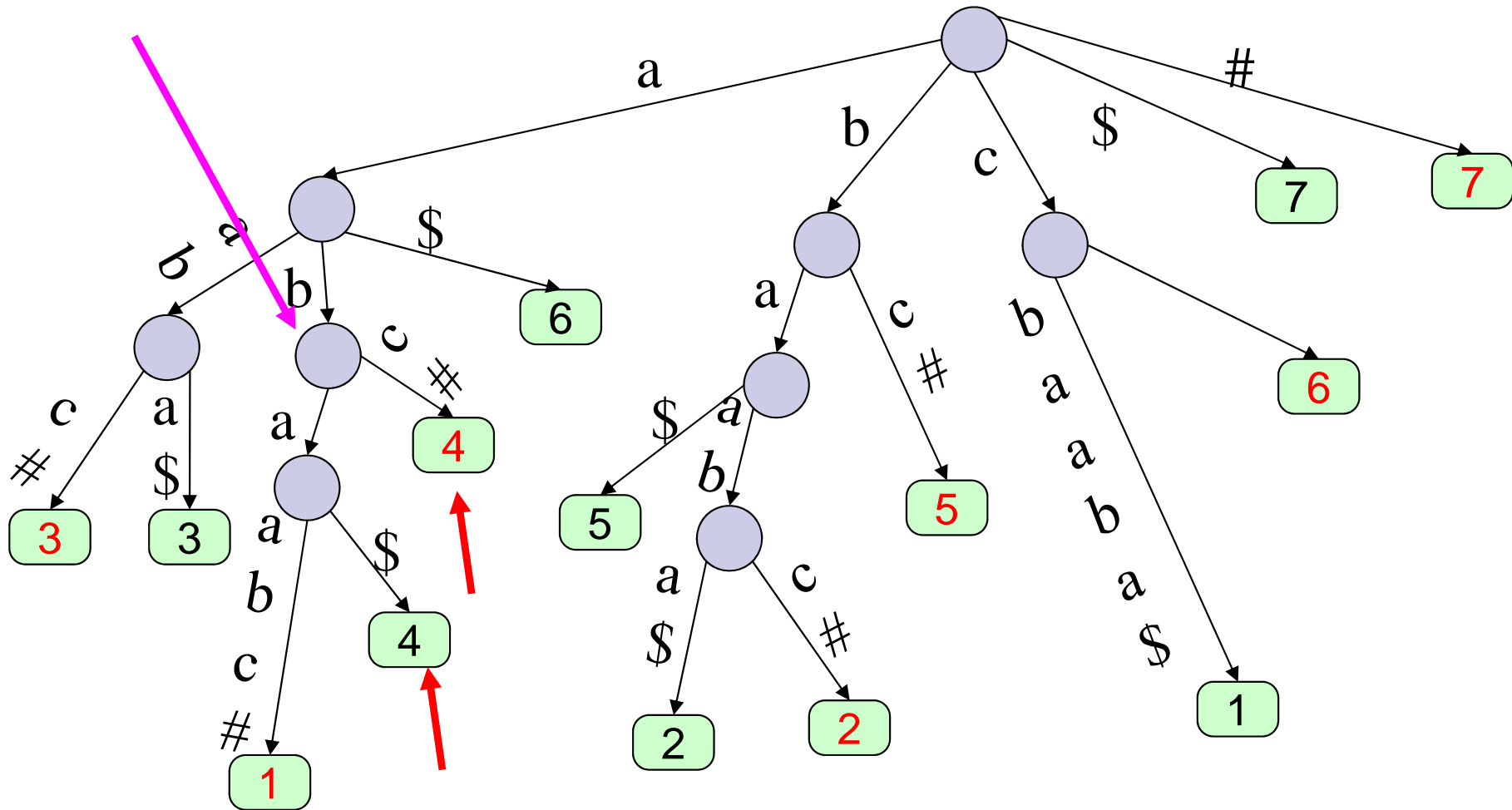
Prepare a generalized suffix tree for

$s = cbaaba\$$  and  $s^r = abaabc\#$

For every  $i$  find the LCA of suffix  $i$  of  $s$   
and suffix  $m-i+1$  of  $s^r$



Let  $s = cbaaba\$$  then  $s^r = abaabc\#$



# Analysis

$O(m)$  time to identify all palindromes



# ST Drawbacks

- Suffix trees consume a lot of space
- It is  $O(m)$  but the constant is quite big



# Suffix arrays (U. Mander, G. Myers '91)

- We lose some of the functionality but we save space.

Let  $s = abab$

Sort the suffixes lexicographically:  
 $ab, abab, b, bab$

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2
---	---	---	---



# How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- $O(m)$  time



# How do we search for a pattern ?

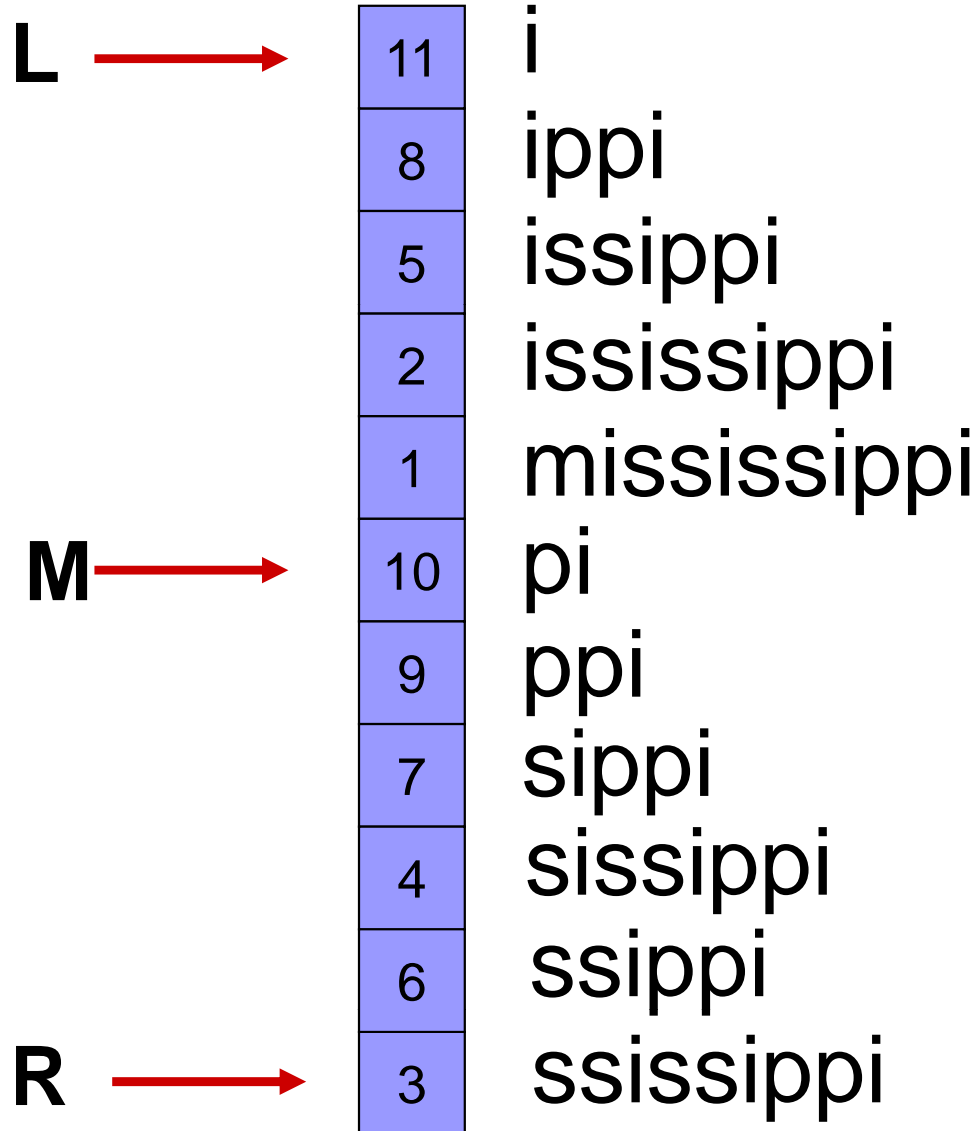
- If  $P$  occurs in  $S$  then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Naive implementation:  $O(n \log m)$  time
- Can show:  $O(n + \log m)$  time



# Example

Let  $S = \text{mississippi}$

Let  $P = \text{issa}$



Udi Manber

Gene Myers

