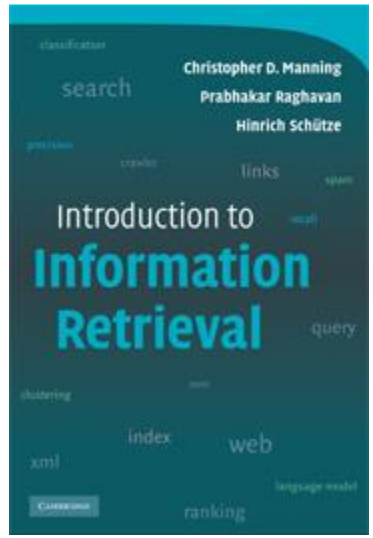
Information Retrieval and Organisation



Suffix Trees

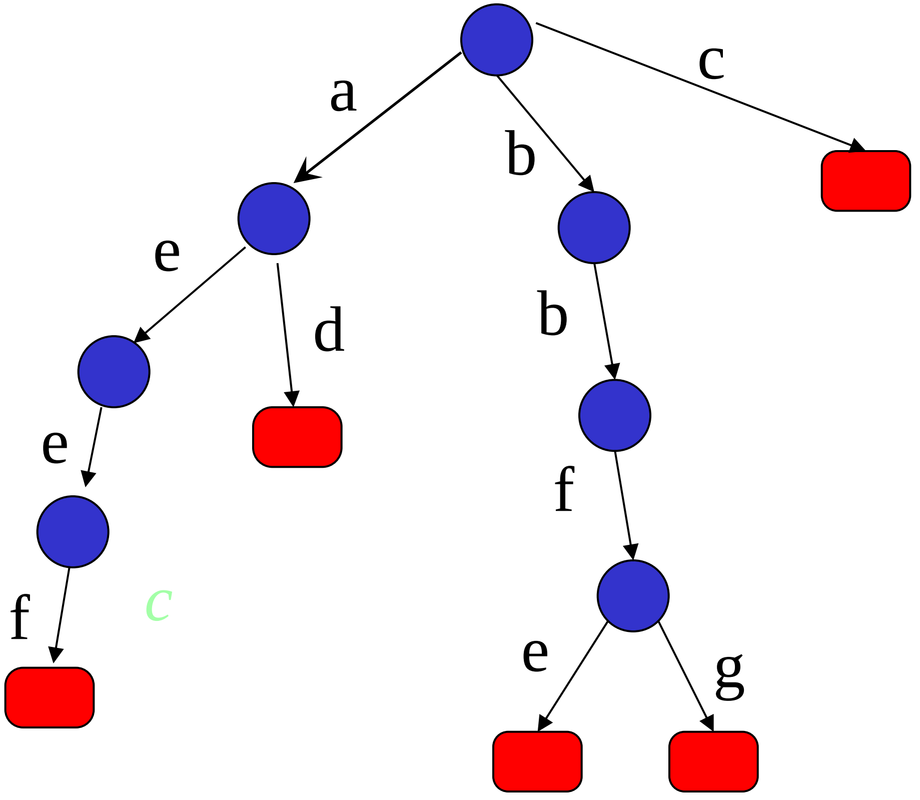
adapted from
http://www.math.tau.ac.il/~haimk/seminar02/suffixtrees.ppt

Dell Zhang
Birkbeck, University of London

Trie

- A tree representing a set of strings

{
aeef
ad
bbfe
bbfg
c
}



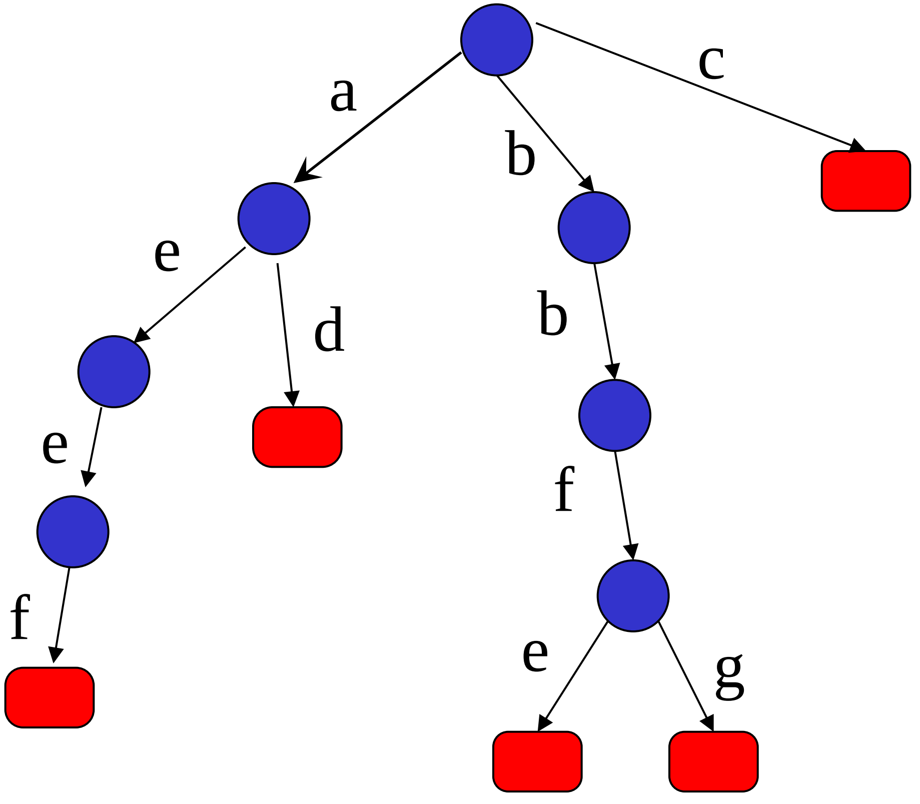
Trie

- Assume no string is a prefix of another

1) Each edge is labeled by a letter.

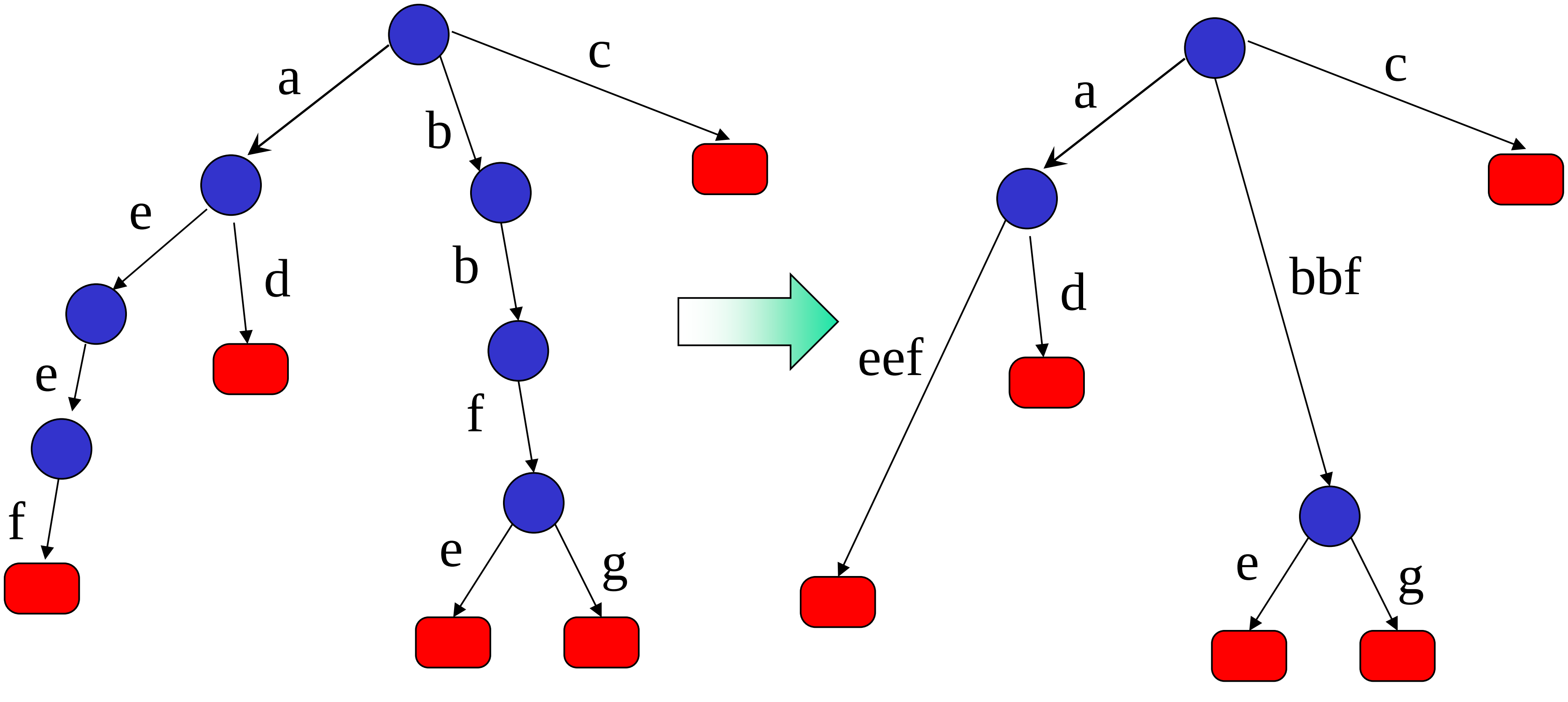
2) No two edges outgoing from the same node are labeled the same.

3) Each string corresponds to a leaf.



Compressed Trie

- Compress unary nodes, label edges by strings



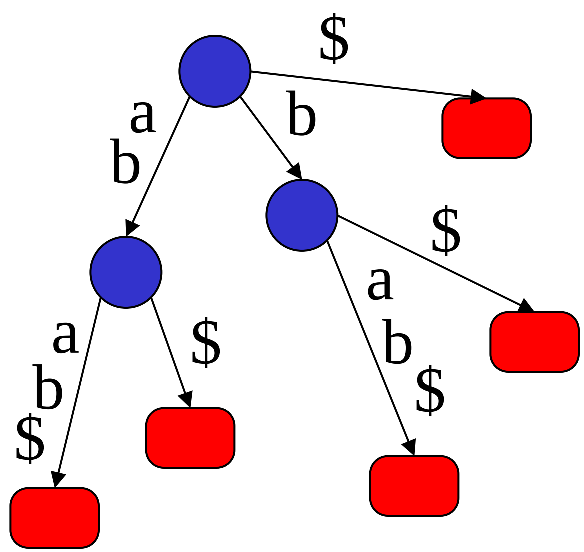
Suffix Tree

- Given a string *s*, a suffix tree of *s* is a compressed trie of all suffixes of *s*.
- To make these suffixes prefix-free we add a special character, say \$, at the end of *s*.

Suffix Tree

- For example, let *s* = abab, a suffix tree of *s* is a compressed trie of all suffixes of abab$.

{
$
b$
ab$
bab$
abab$
}

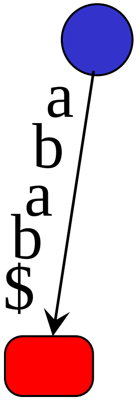


Note that a suffix tree has O(*n*) nodes *n* = |s|. Why?

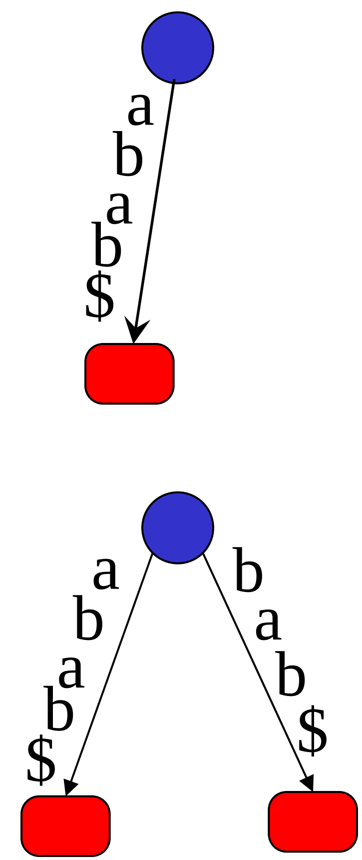
Suffix Tree Construction

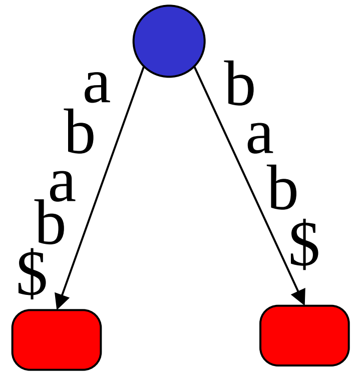
- The trivial algorithm

Put the largest suffix in

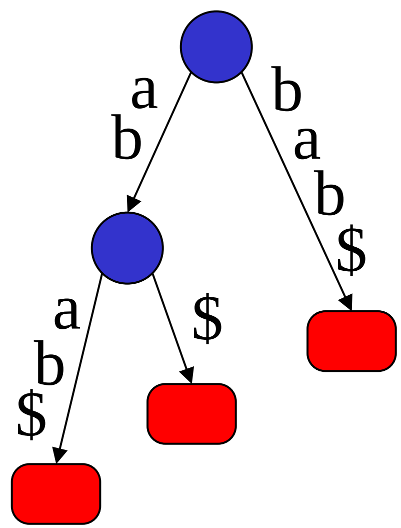


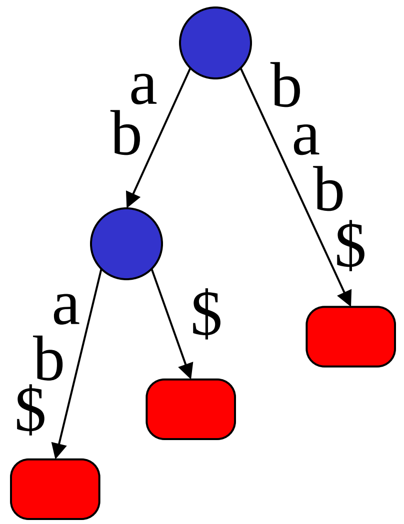
Put the suffix bab$ in



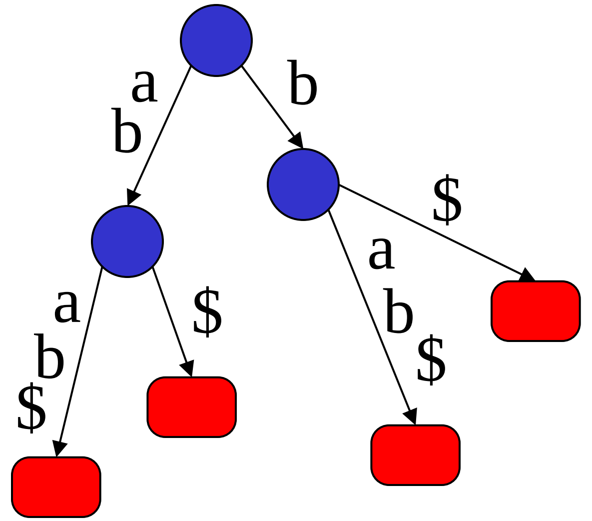


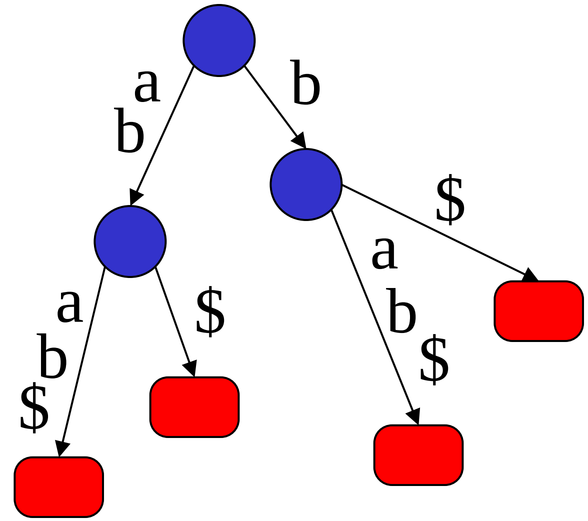
Put the suffix ab$ in



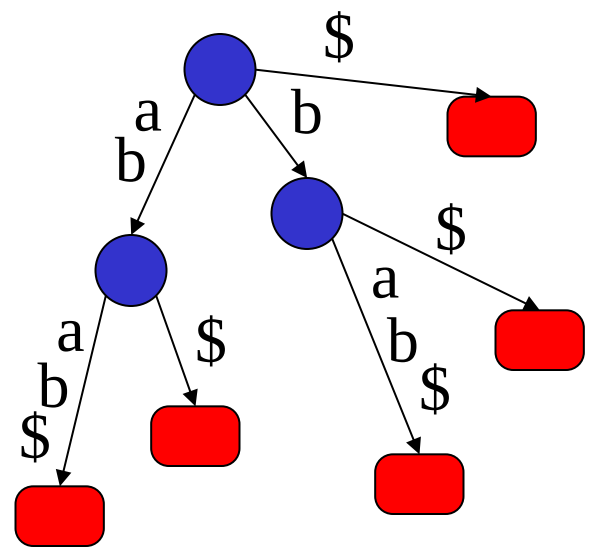


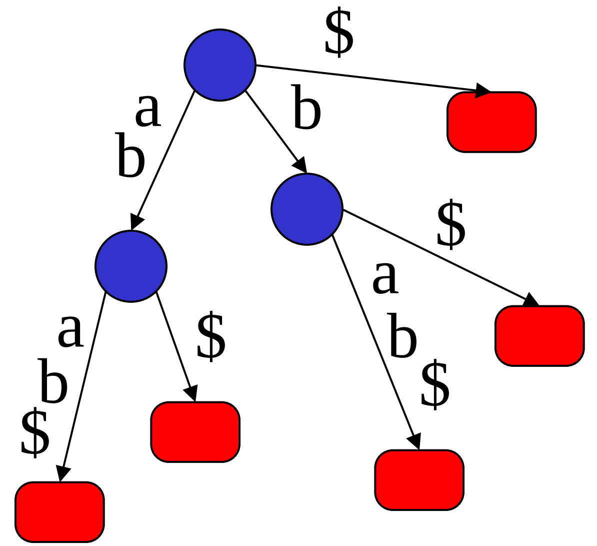
Put the suffix b$ in



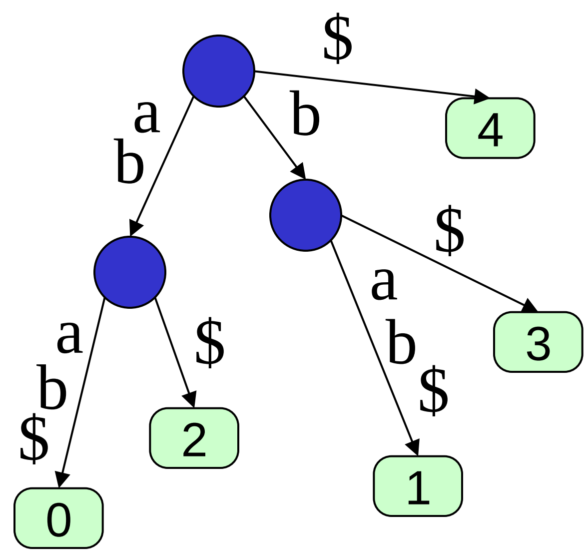


Put the suffix $ in





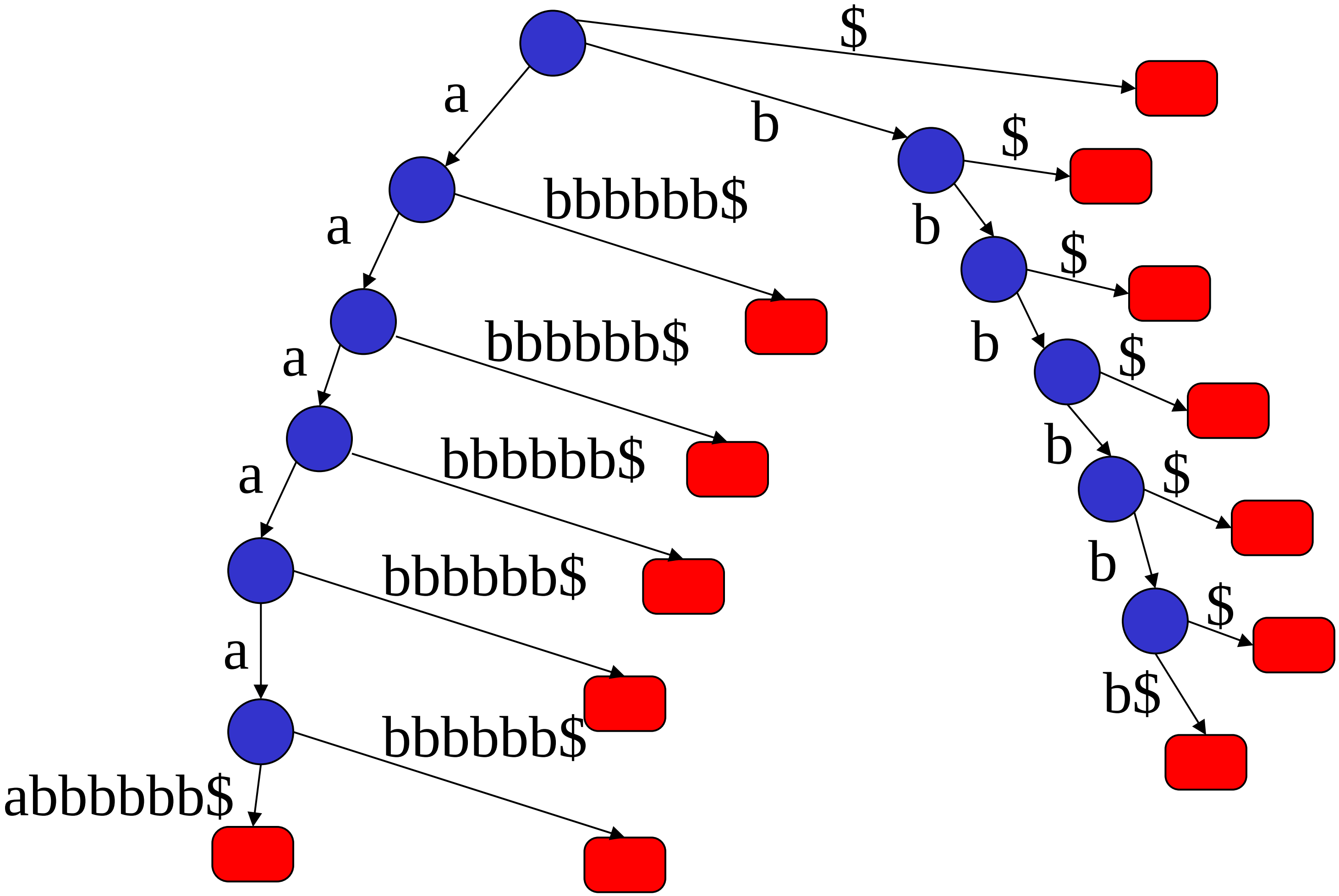
We will also label each leaf with the starting point of the corresponding suffix



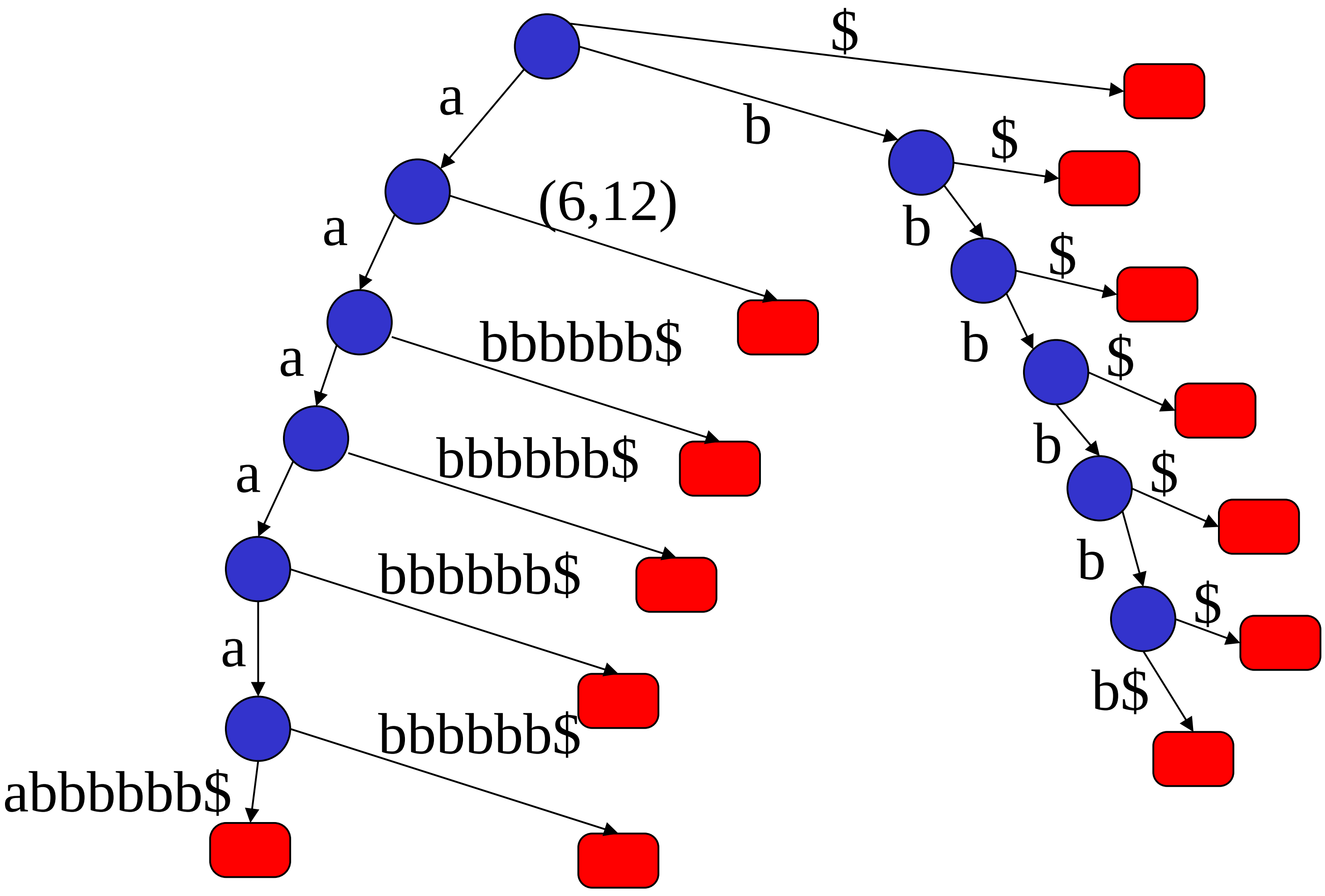
Suffix Tree Construction

- The trivial algorithm takes $O(n^2)$ time.
- It is possible to build a suffix tree in $O(n)$ time using ***Ukkonen's algorithm***.
  - But, how come? Does it take $O(n)$ space?
  - To use only $O(n)$ space, encode the edge-labels as (beginning-position, end-position) .

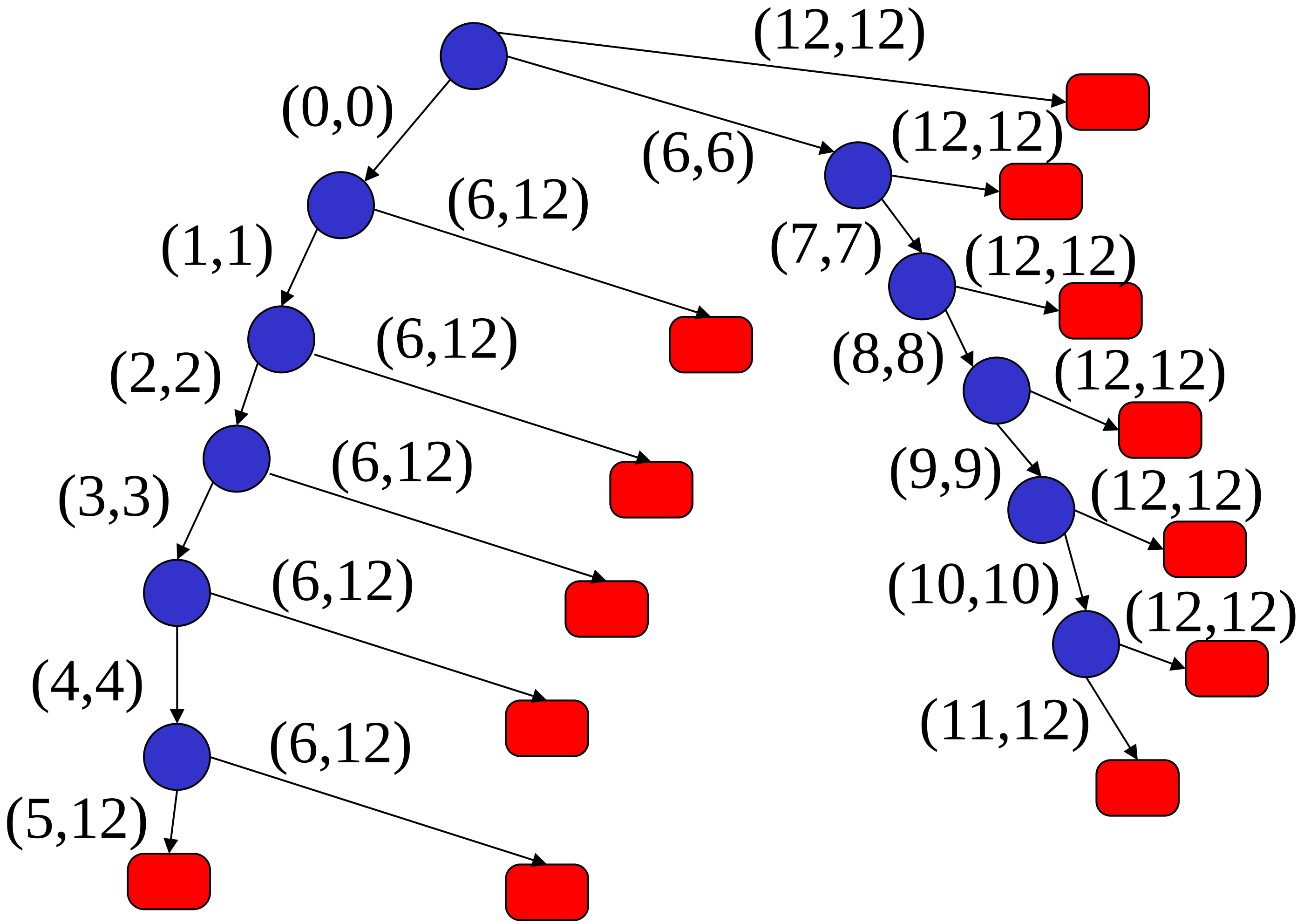
Consider the string aaaaaabbbbbb$



Consider the string aaaaaabbbbbb$



Consider the string aaaaaabbbbbb$



Suffix Tree Applications

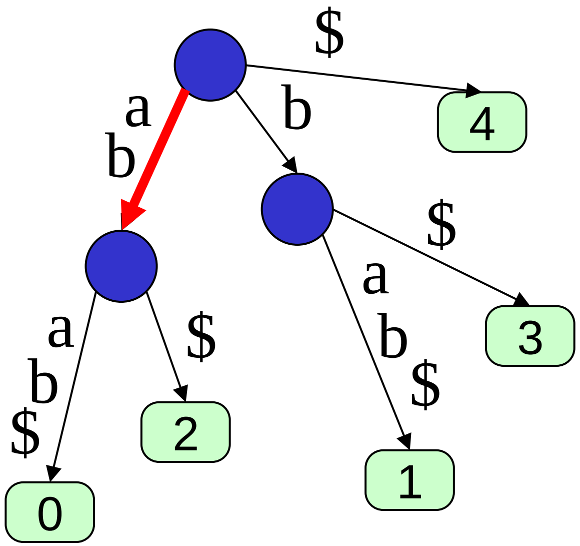
- What Can We Do with It?
  - Exact String Matching
  - Exact Set Matching
  - The Substring Problem for a Database of Patterns
  - Longest Common Substring of Two Strings
  - Recognising DNA Contamination
  - Common Substring of More Than Two Strings
  - ……

Exact String Matching

- Given text T ($|T| = n$), pre-process it such that when a pattern P ($|P| = m$) arrives you can quickly decide when it occurs in T.
- We may also want to find all occurrences of P in T.

Exact String Matching

- In pre-processing, we just build a suffix tree in O(n) time



Exact String Matching

- Given a pattern P = ab we traverse the tree according to the pattern.
- If we do not get stuck traversing the pattern then the pattern occurs in the text, otherwise it does not.
- 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.

Exact String Matching

- How to match a pattern (query) against a database of strings (documents)?

Generalized Suffix Tree

- Given a set of strings S, the generalized suffix tree of S is a compressed trie of all suffixes of each $s \in$ 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 to each s.
- Each leaf node needs to be labelled by the document id together with the suffix position.

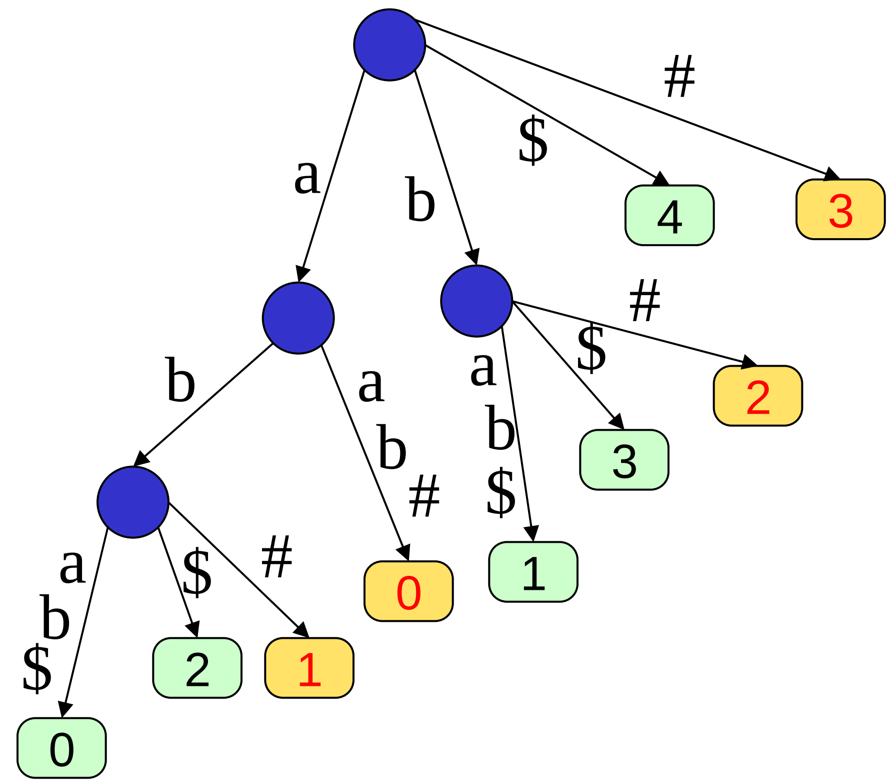
Generalized Suffix Tree

- For example, Let s_1 = abab and s_2 = aab, here is a generalized suffix tree for s_1 and s_2.

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

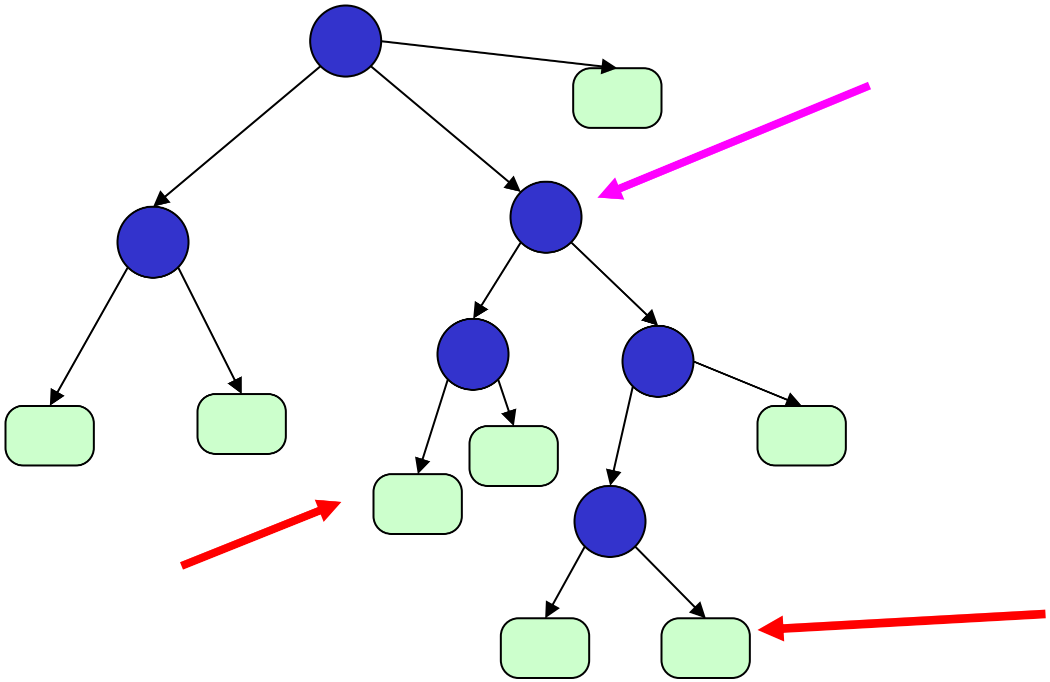


Longest Common Substring

- Given two strings s_1 and s_2, we build their generalized suffix tree.
- 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 “string depth”.

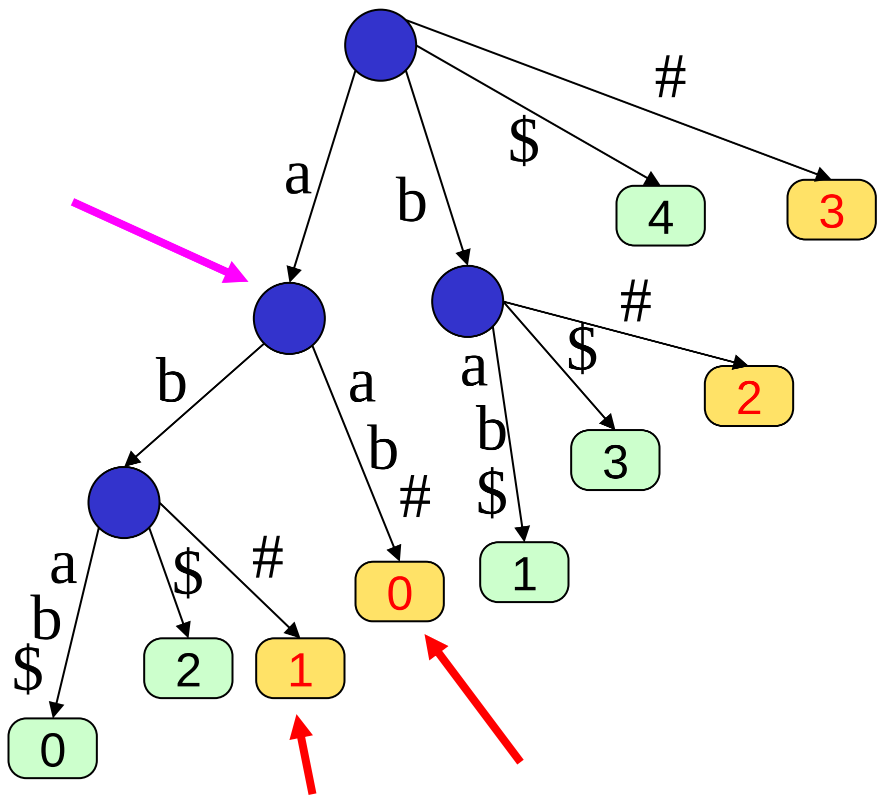
Lowest Common Ancestor

- A lot more can be gained from the suffix tree, if we pre-process it so that we can answer LCA queries on it in constant time.



Lowest Common Ancestor

- Why? The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes



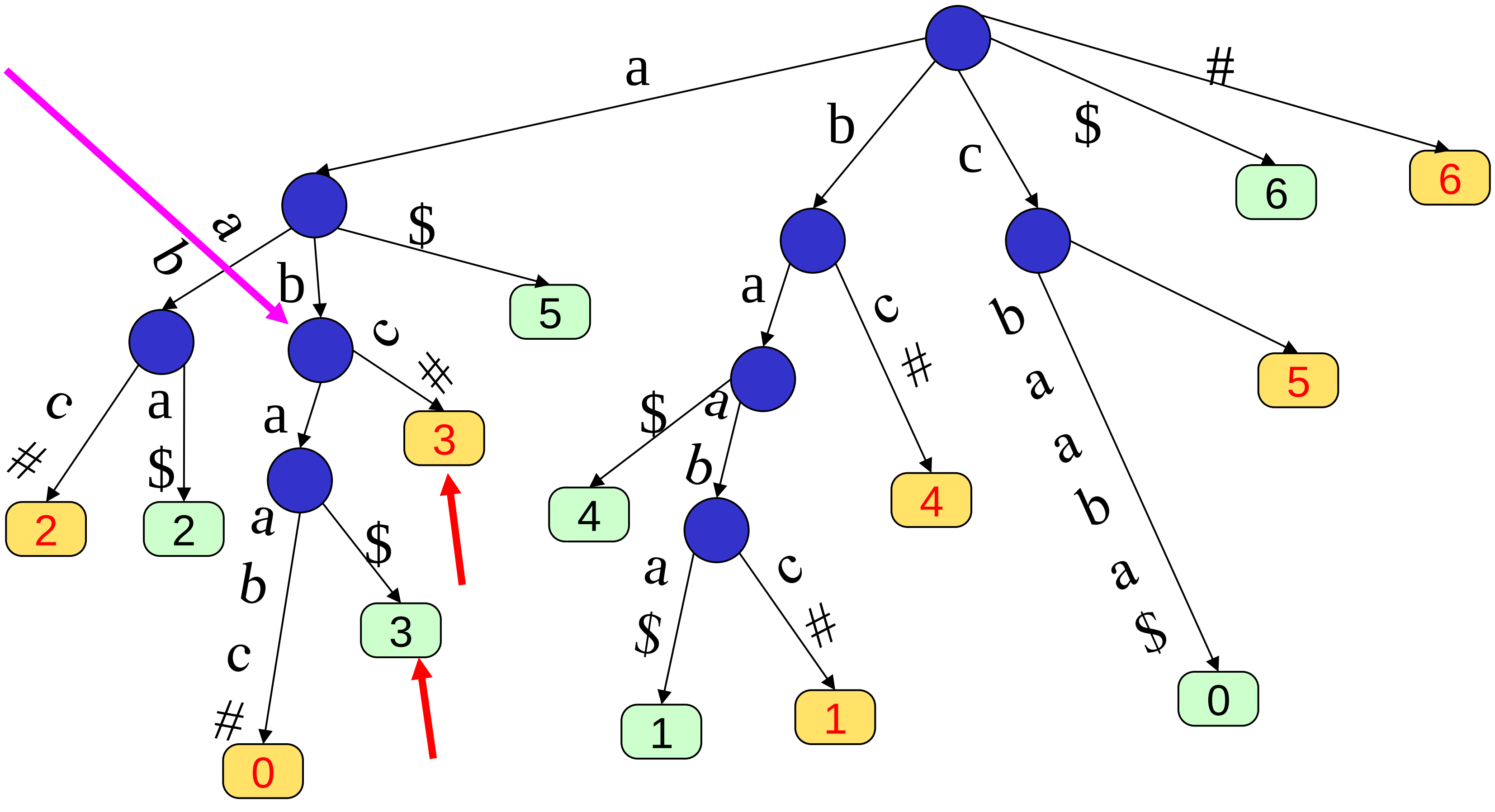
Finding Maximal Palindromes

- A palindrome: cbaabc, caabaac, …
- To find all palindromes in a string *s* (of length m), we build a generalized suffix tree for the string *s* and the reversed string s^r.
- The palindrome with centre between i-1 and i is the LCP of the suffix at position i of *s* and the suffix at position m-i of s^r.

Finding Maximal Palindromes

- For example, consider the string cbaaba.
- Prepare a generalized suffix tree for
  s = cbaaba$ and s^r = abaabc#
- For every i find the LCA of
  the suffix i of s and the suffix m-i of s^r.
- All palindromes can be identified in linear time.

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



Suffix Tree Drawbacks

- It is $O(n)$ but the constant is quite big.
- It consume a lot of space.
  - Notice that if we indeed want to traverse an edge in $O(1)$ time then we need an array (of pointers) of size $|\Sigma|$ in each node, where Σ is the alphabet.

Suffix Array

- It is much simpler and easier to implement.
- Compared with suffix trees, we lose some functionality, but we save space.

Suffix Array

- For example, let s = abab
  - Sort the suffixes lexicographically: ab, abab, b, bab
  - The suffix array gives the indices of the suffixes in sorted order

2	**0**	**3**	**1**

Suffix Array Construction

- The trivial algorithm
  - Quicksort
- The linear time algorithm
  - Build a suffix tree in $O(n)$ time first, and then traverse the tree in in-order, lexicographically picking edges outgoing from each node, and fill the suffix array.
  - It can also be built in $O(n)$ time directly.

Exact String Matching

- How do we search for a pattern *P* in the text *T*, using the suffix array of *T*?
- If *P* occurs in *T*, then all its occurrences are consecutive in the suffix array.
- So we can do two binary searches on the suffix array: the first search locates the starting position of the interval, and the second one determines the end position.
- It takes $O(m \log(n))$ time, as a single suffix comparison needs to compare up to m characters.

Exact String Matching

- It is also possible to do it in $O(m+\log(n))$ with an additional array of LCP.
  - Manber & Myers (1990)

T = mississippi

P = issa