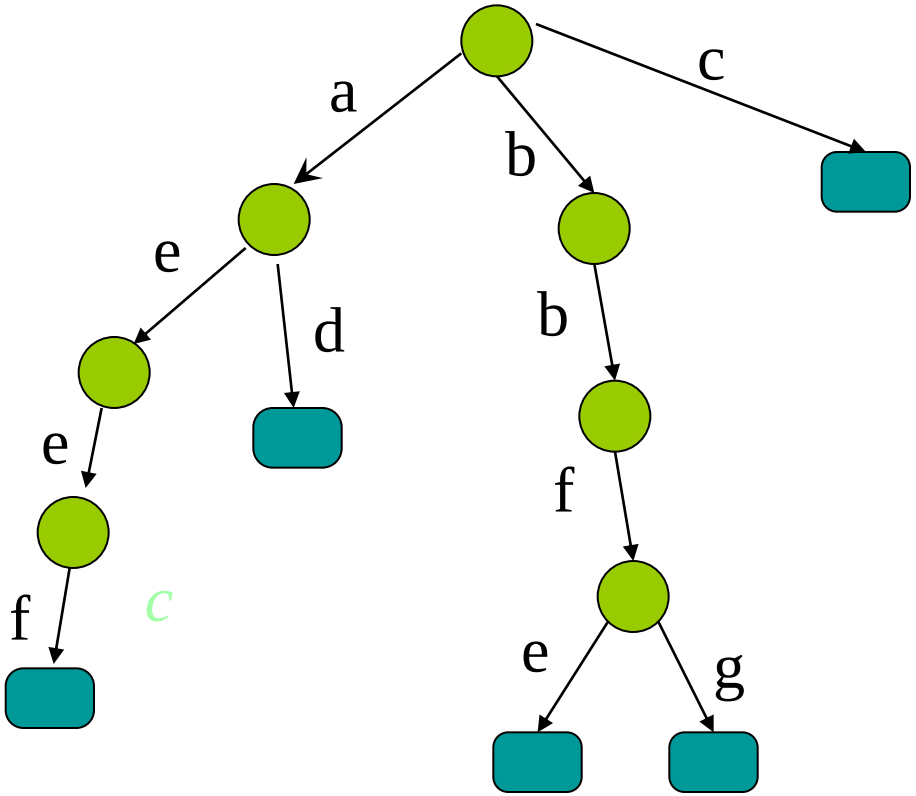
Suffix trees and suffix arrays

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

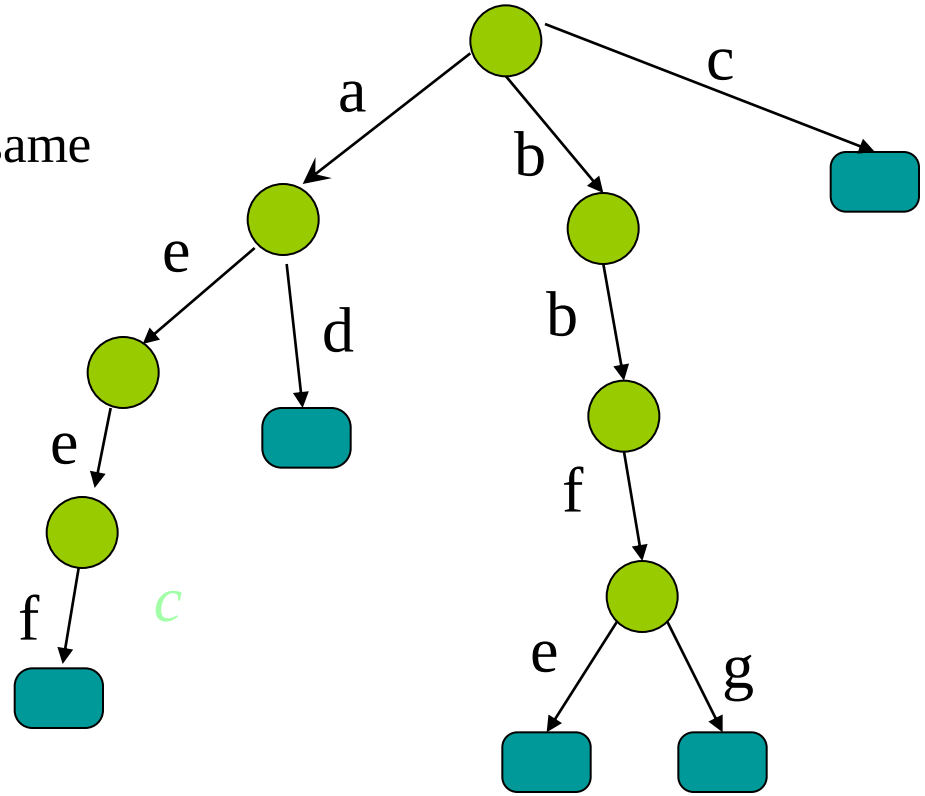


Trie (Cont)

- Assume no string is a prefix of another

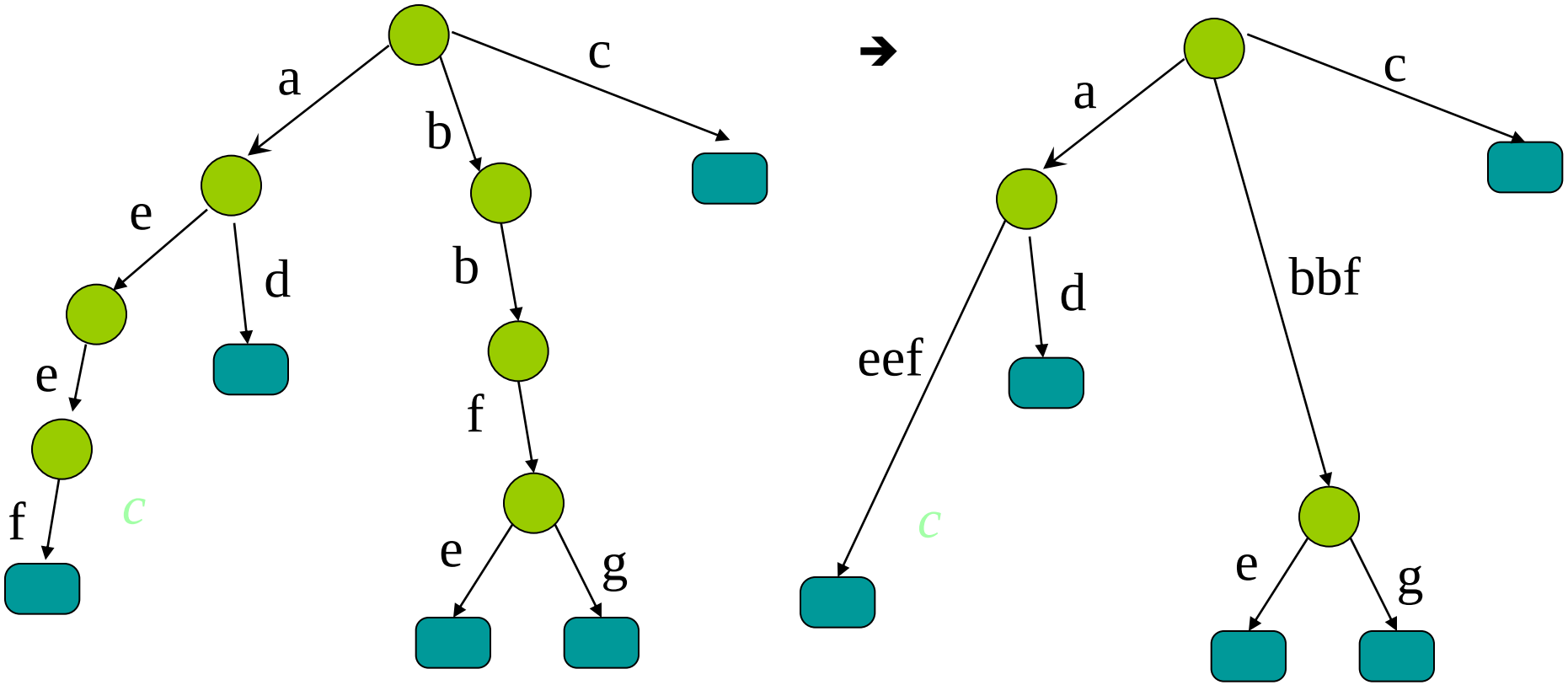
Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.



Compressed Trie

- Compress unary nodes, label edges by strings



Suffix tree

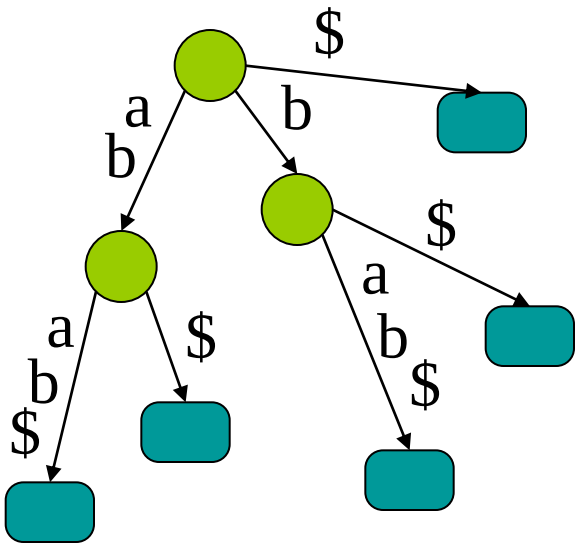
Given a string s a suffix tree of s is a compressed trie of all suffixes of s

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

Suffix tree (Example)

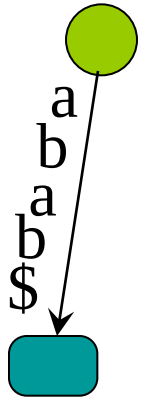
Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
  $
  b$
  ab$
  bab$
  abab$
}

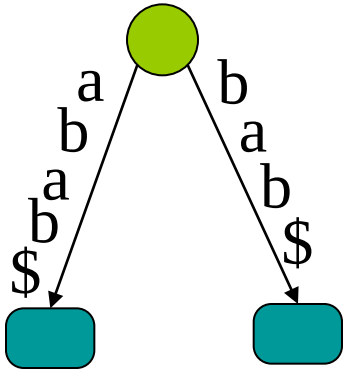


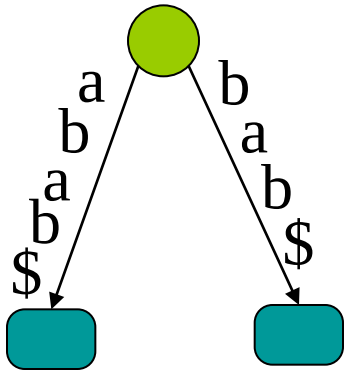
Trivial algorithm to build a Suffix tree

Put the largest suffix in

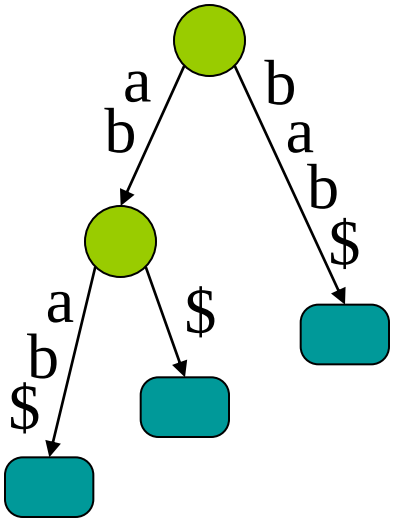


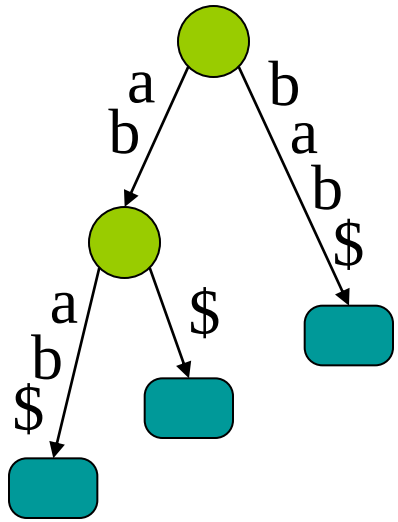
Put the suffix bab$ in



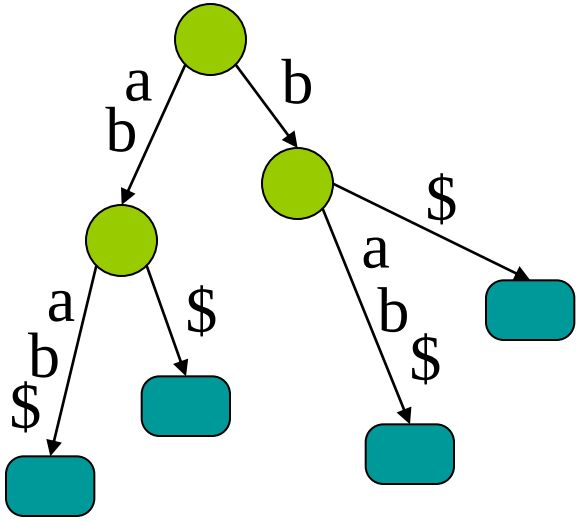


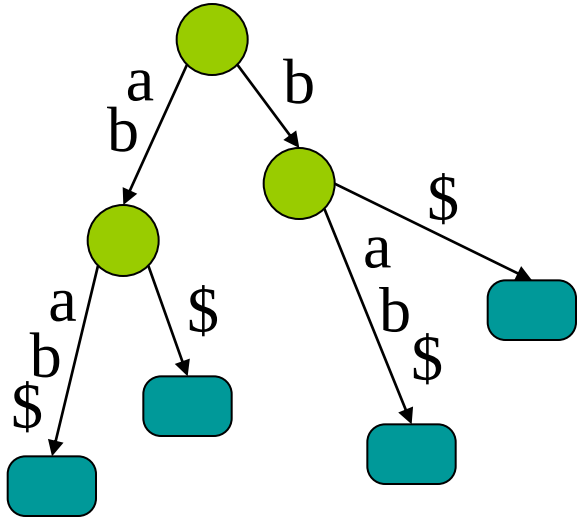
Put the suffix ab$ in



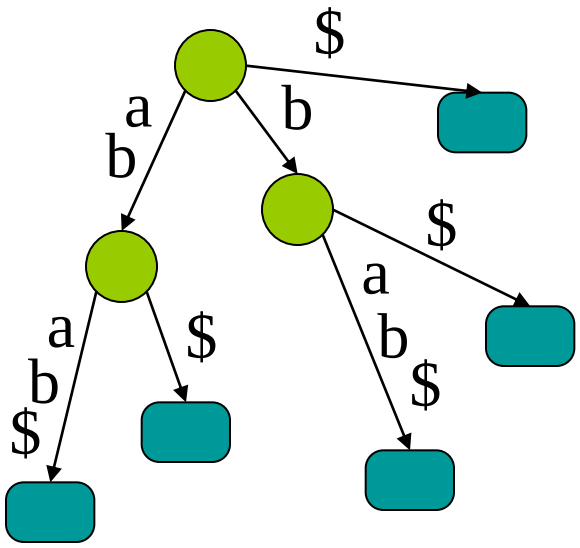


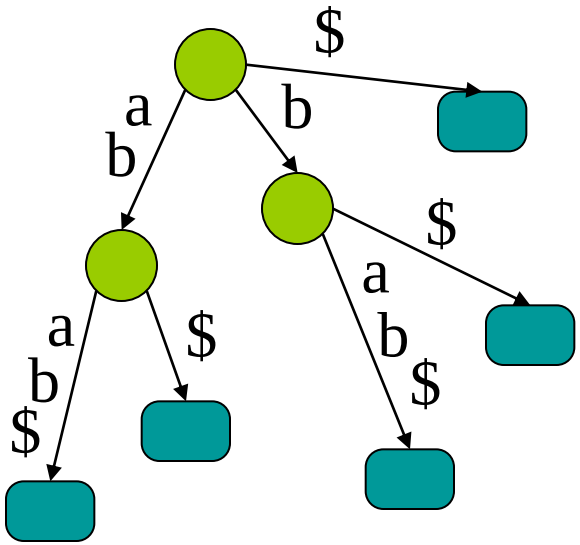
Put the suffix b$ in



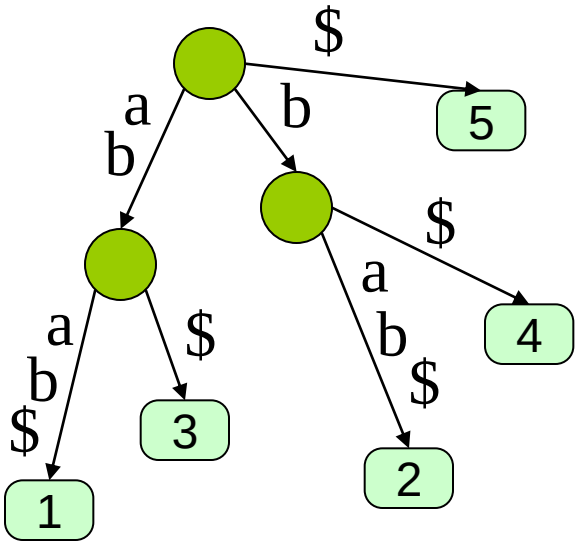


Put the suffix $ in





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



Analysis

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

We will see how to do it in O(n) time

What can we do with it ?

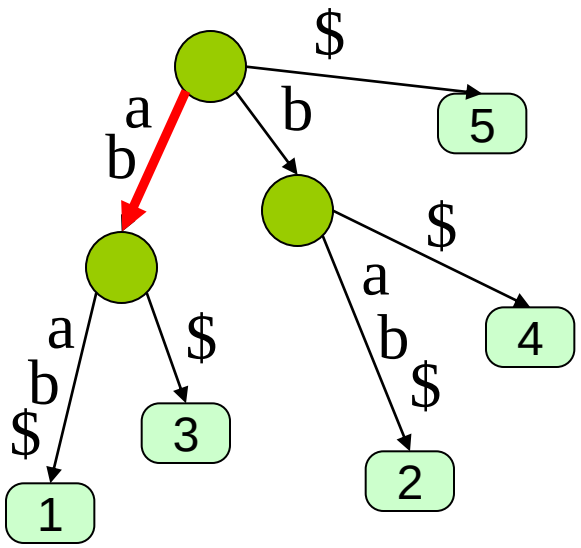
Exact string matching:

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

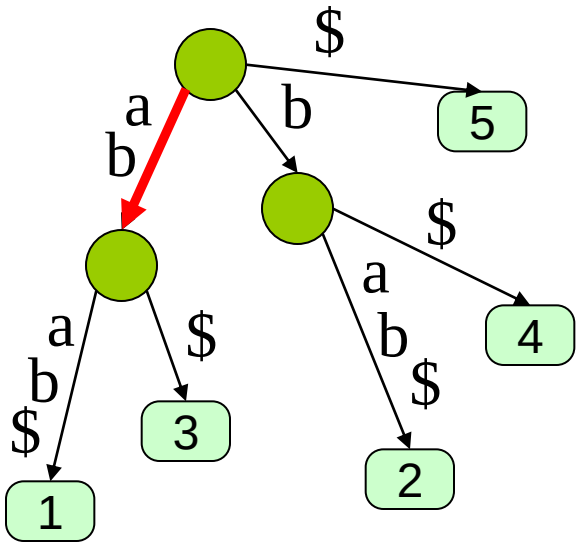
We may also want to find all occurrences of P in T

Exact string matching

In preprocessing we just build a suffix tree in O(n) 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 to each s

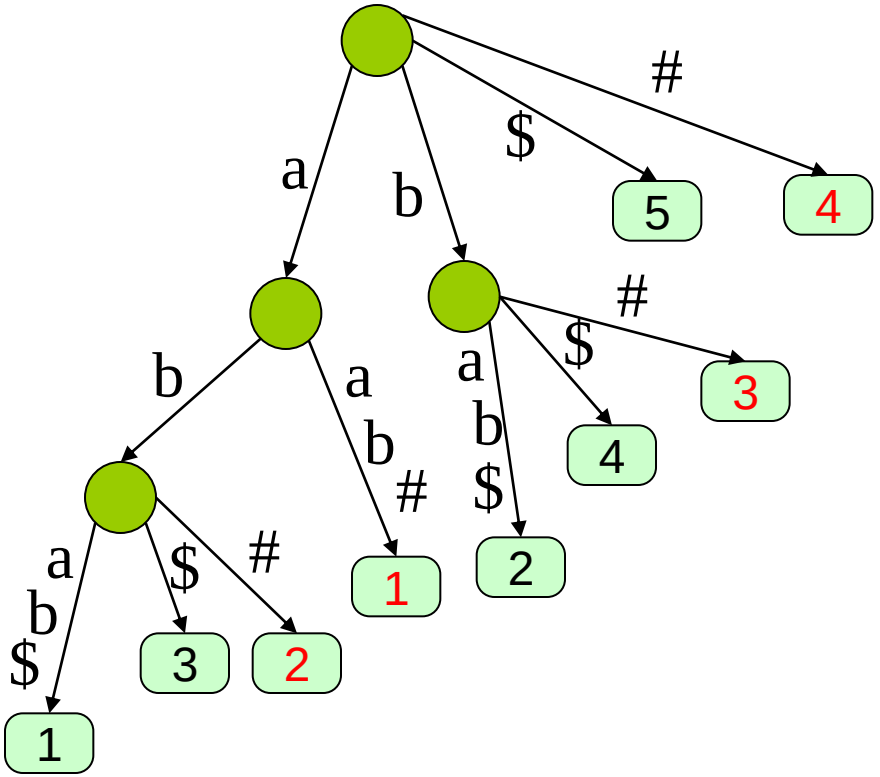
Generalized suffix tree (Example)

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$	

}



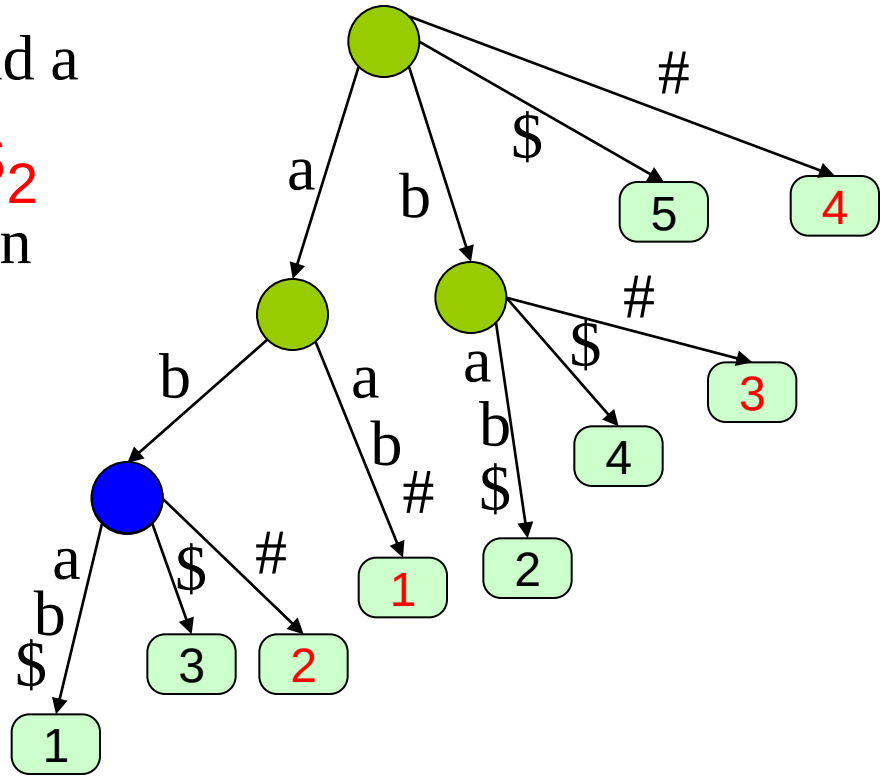
So what can we do with it ?

Matching a pattern against a database of strings

Longest common substring (of two strings)

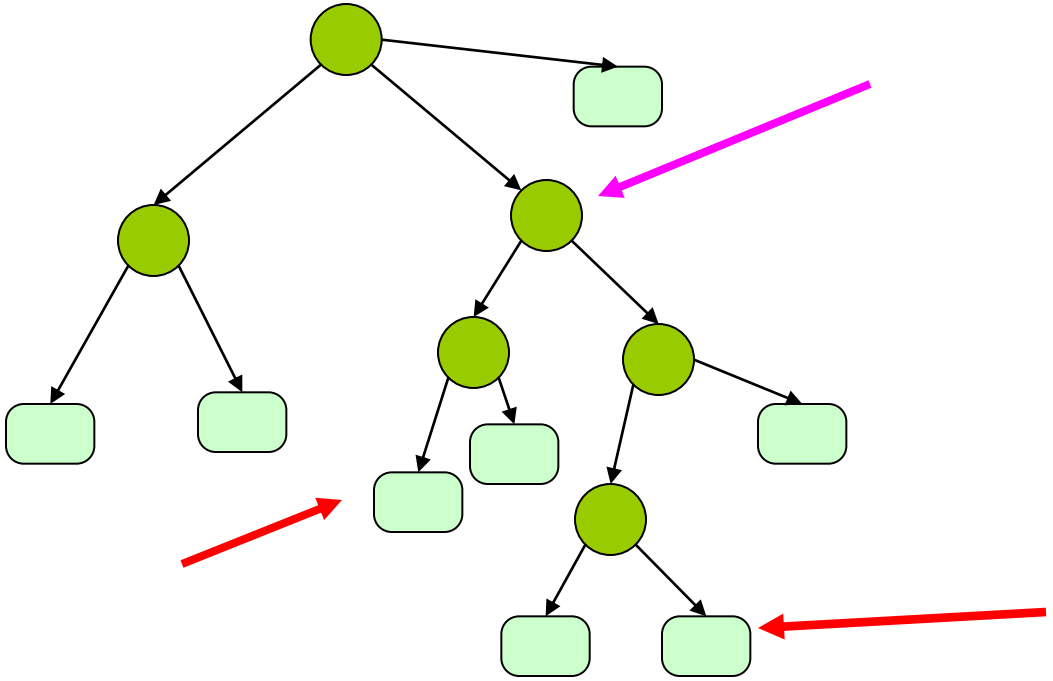
Every node with a leaf descendant from string S_1 and a leaf descendant from string S_2 represents a maximal common substring and vice versa.

Find such node with largest “string depth”



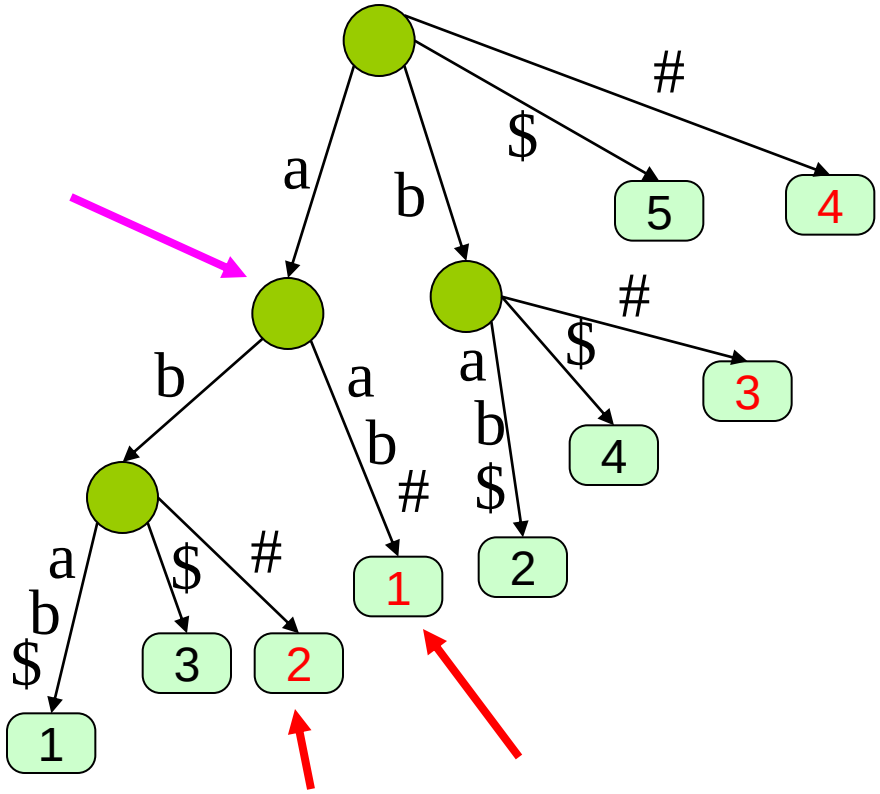
Lowest common ancetors

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



Why?

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



Finding maximal palindromes

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

Let s = cbaaba

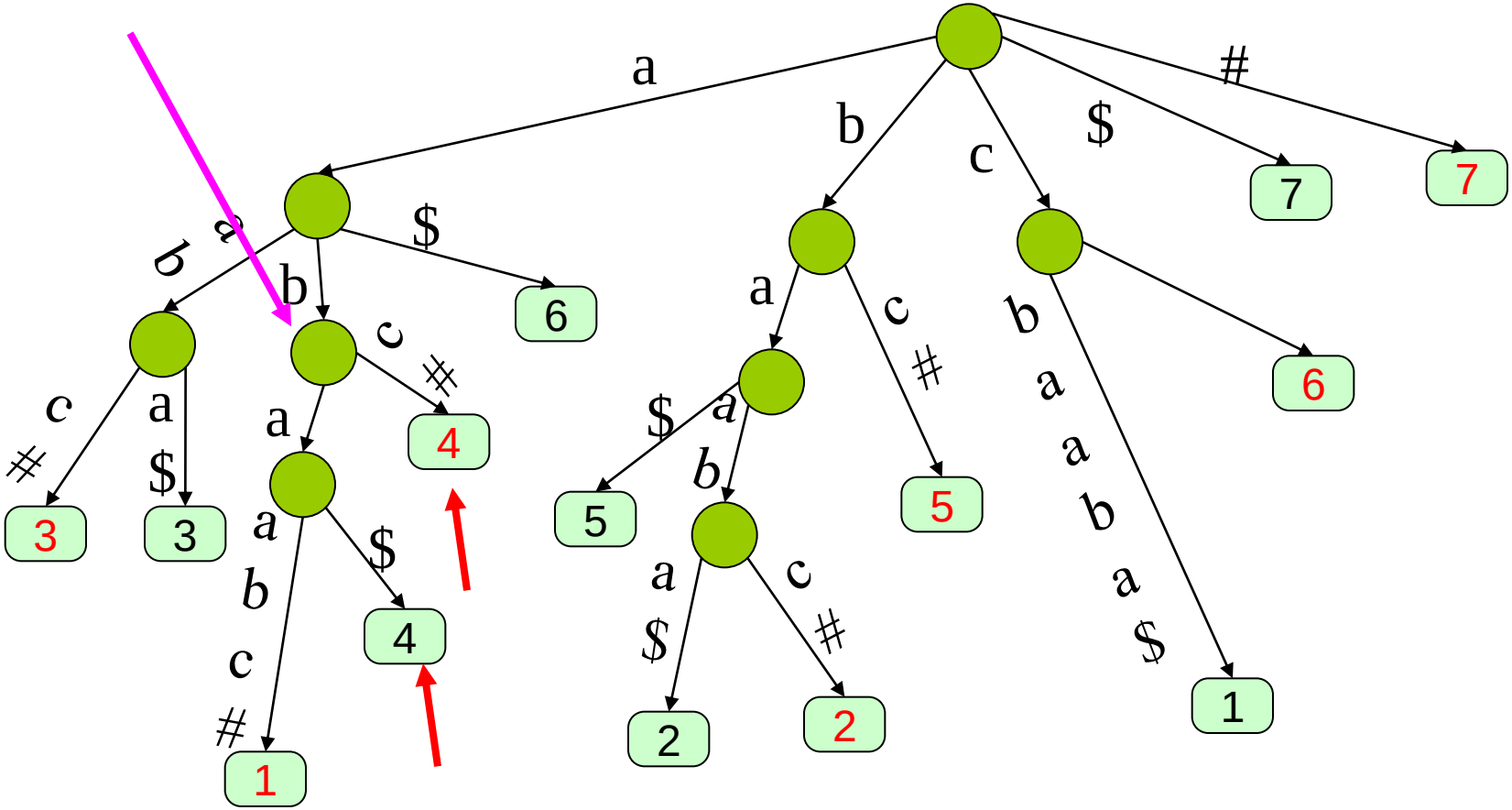
The maximal palindrome with center between i-1 and i is the LCP of the suffix at position i of s and the suffix at position m-i+1 of s^r

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(n) time to identify all palindromes

Drawbacks

- Suffix trees consume a lot of space
- It is O(n) but the constant is quite big
- Notice that if we indeed want to traverse an edge in O(1) time then we need an array of ptrs. of size |Σ| in each node

Suffix array

- We loose 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(n) time

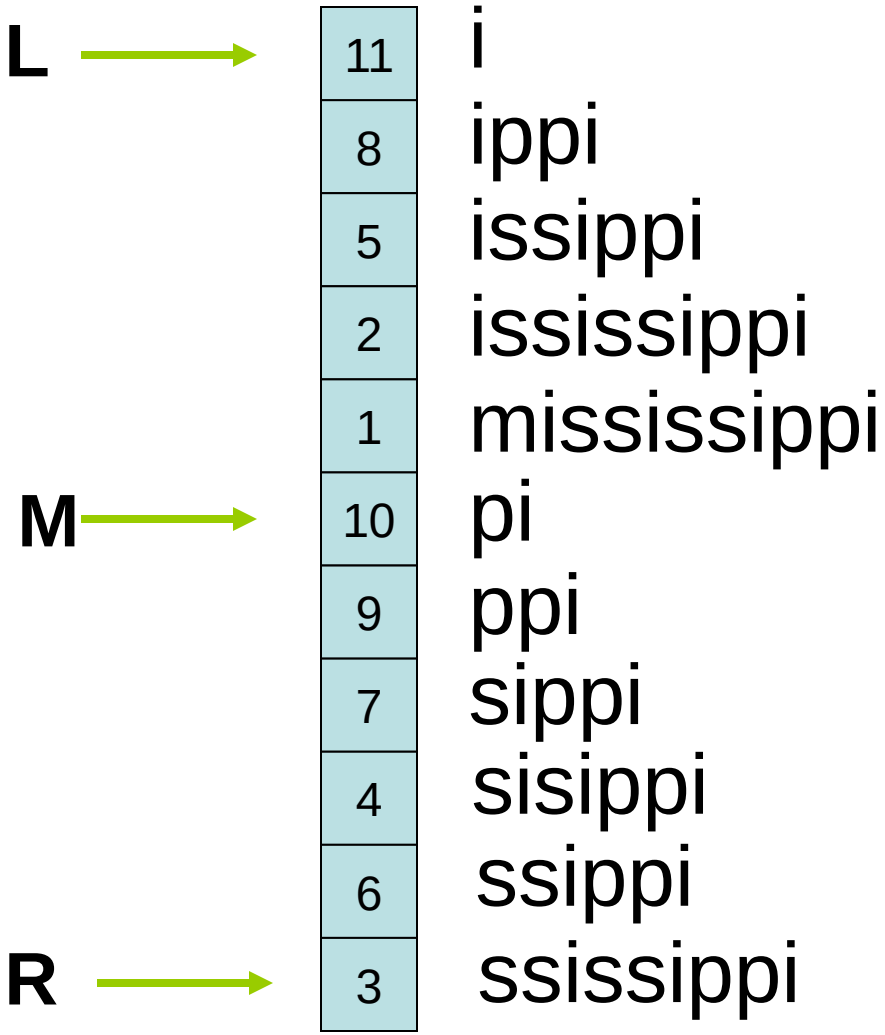
How do we search for a pattern ?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes O(mlogn) time

Example

Let S = mississippi

Let P = issa

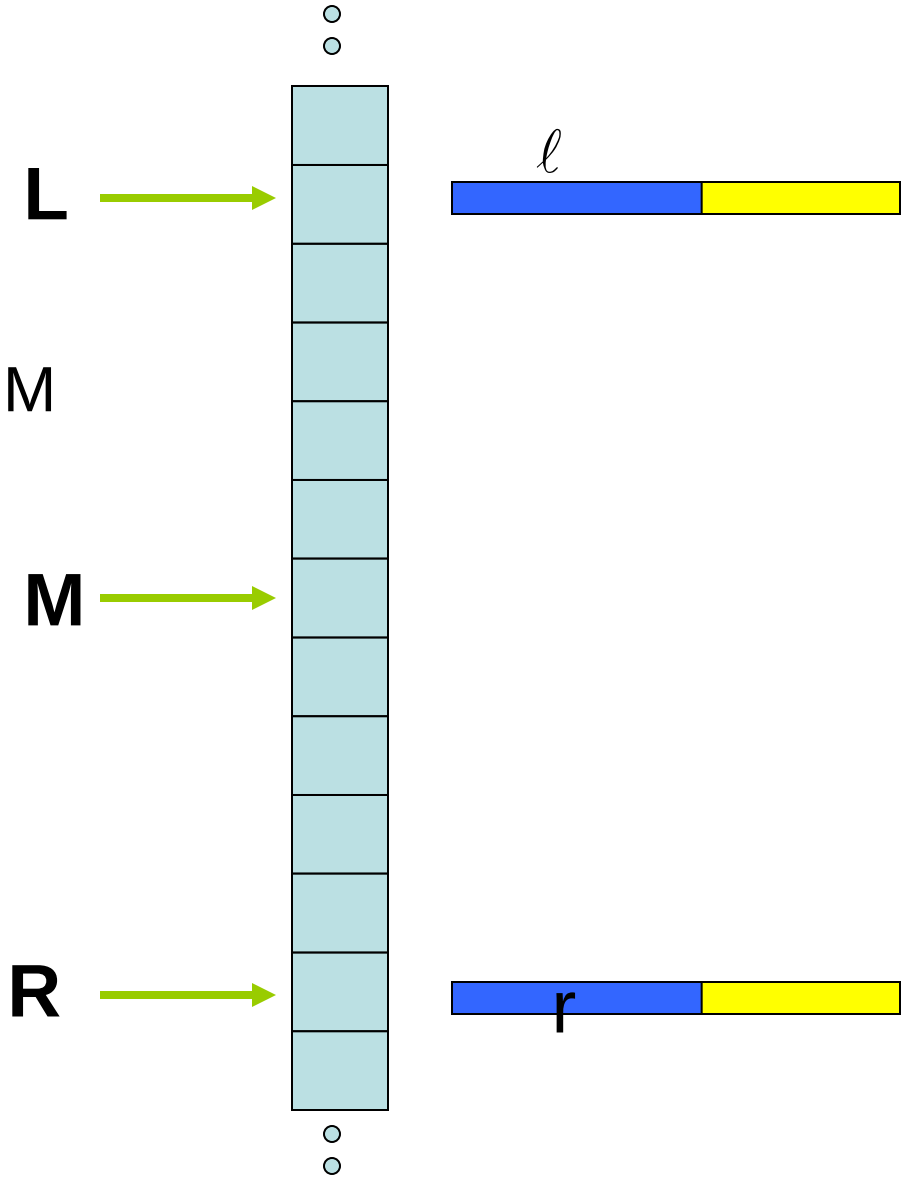


How do we accelerate the search ?

Maintain ℓ = LCP(P,L)

Maintain r = LCP(P,R)

If ℓ = r then start comparing M to P at ℓ + 1



How do we accelerate the search ?

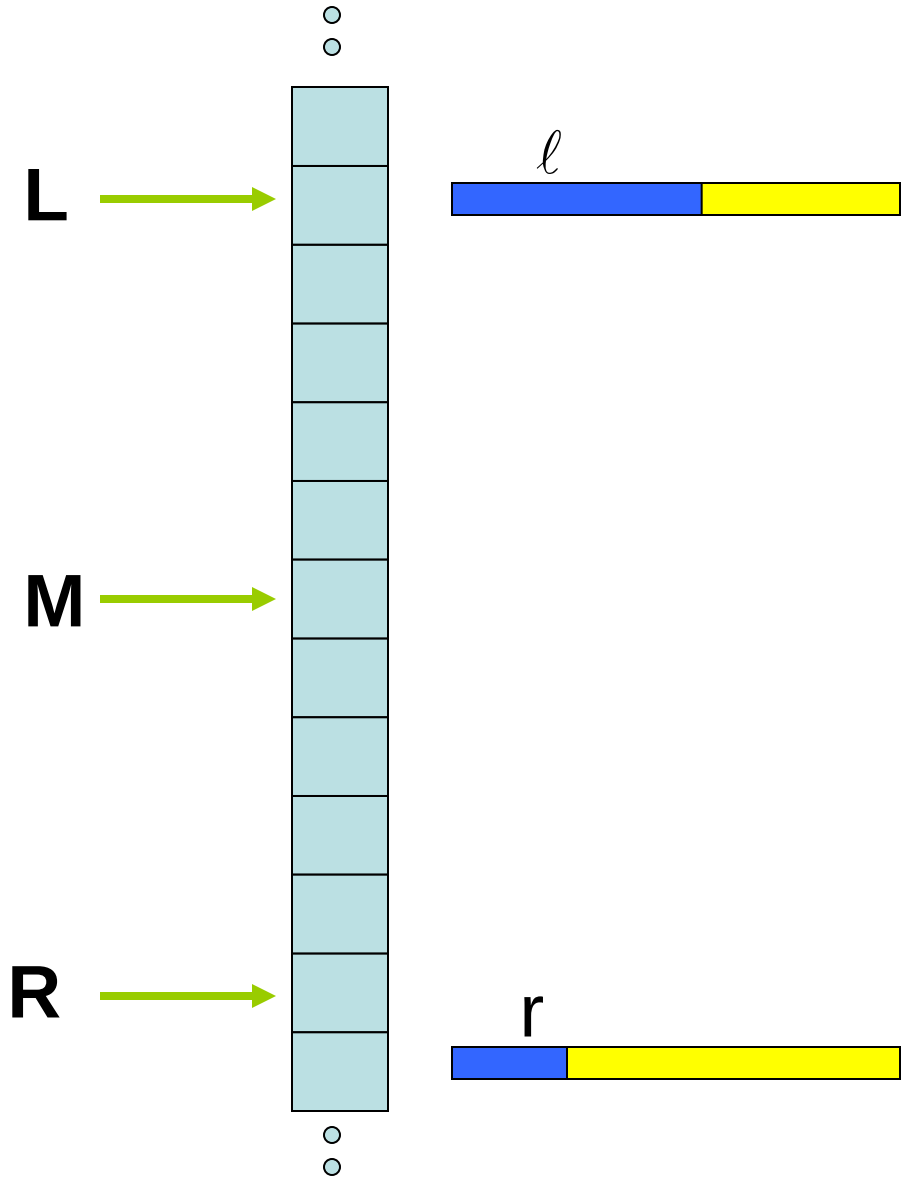
If $\ell > r$ then

Suppose we know LCP(L,M)

If LCP(L,M) < ℓ we go left

If LCP(L,M) > ℓ we go right

If LCP(L,M) = ℓ we start comparing at $\ell + 1$



Analysis of the acceleration

If we do more than a single comparison in an iteration then max(ℓ, r) grows by 1 for each comparison ➔ O(logn + m) time