ALGORITHMES POUR LA BIO-INFORMATIQUE ET LA VISUALISATION

COURS 3

Raluca Uricaru

Suffix trees, suffix arrays, BWT

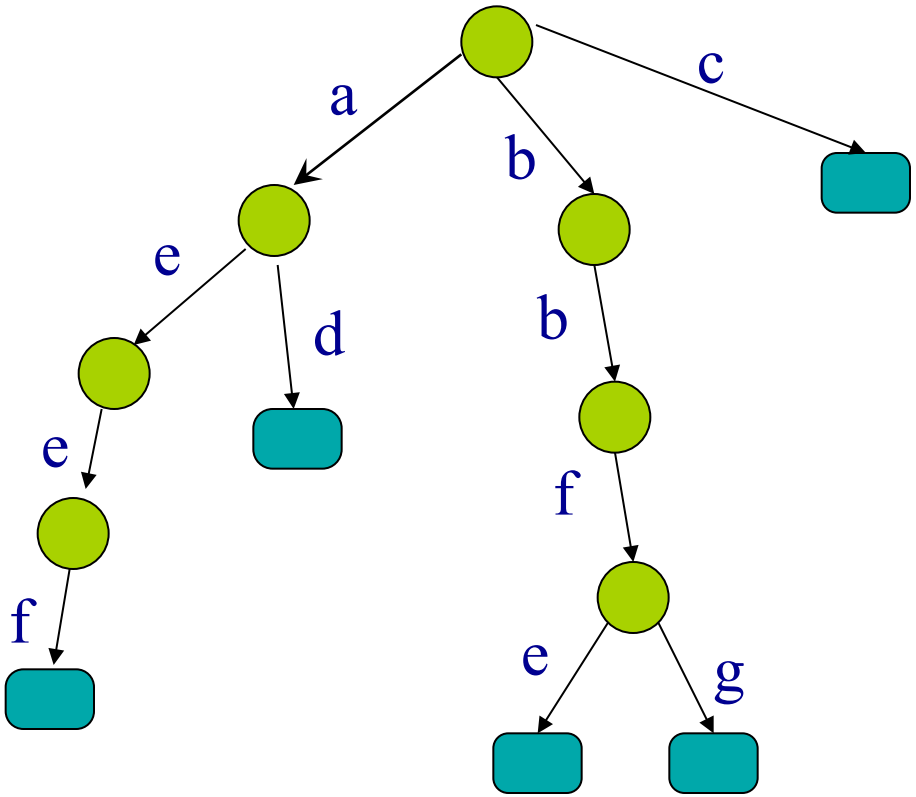
Based on:

Suffix trees and suffix arrays presentation by Haim Kaplan
Suffix trees course by Paco Gomez
Linear-Time Construction of Suffix Trees by Dan Gusfield
Introduction to the Burrows-Wheeler Transform and FM Index, Ben Langmead

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

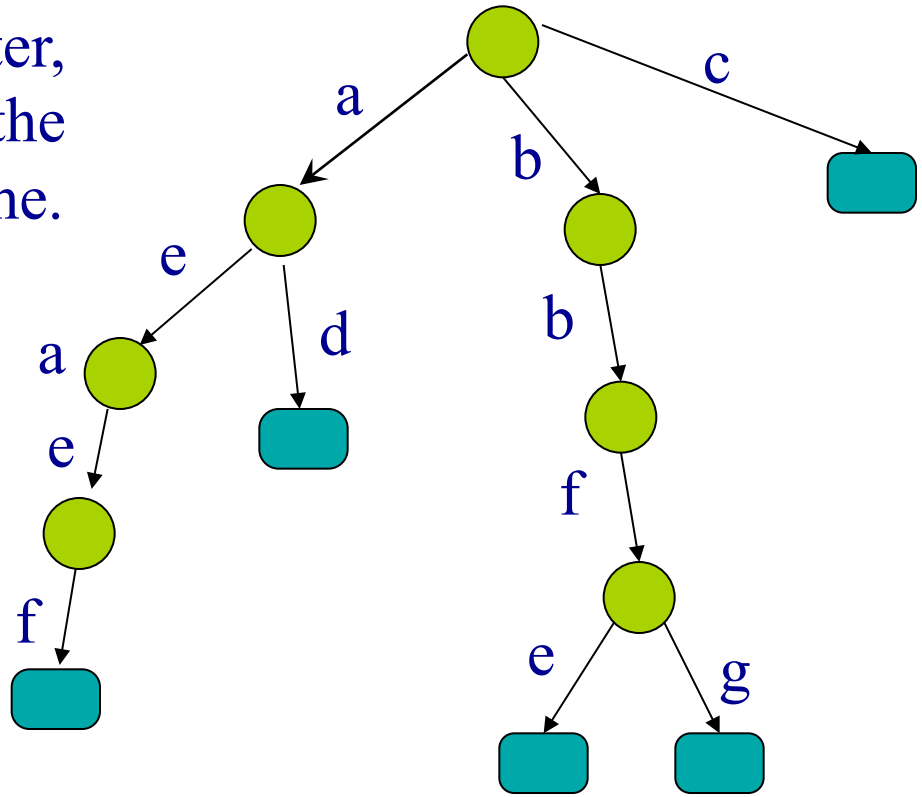


Trie

- Assume no string is a prefix of another

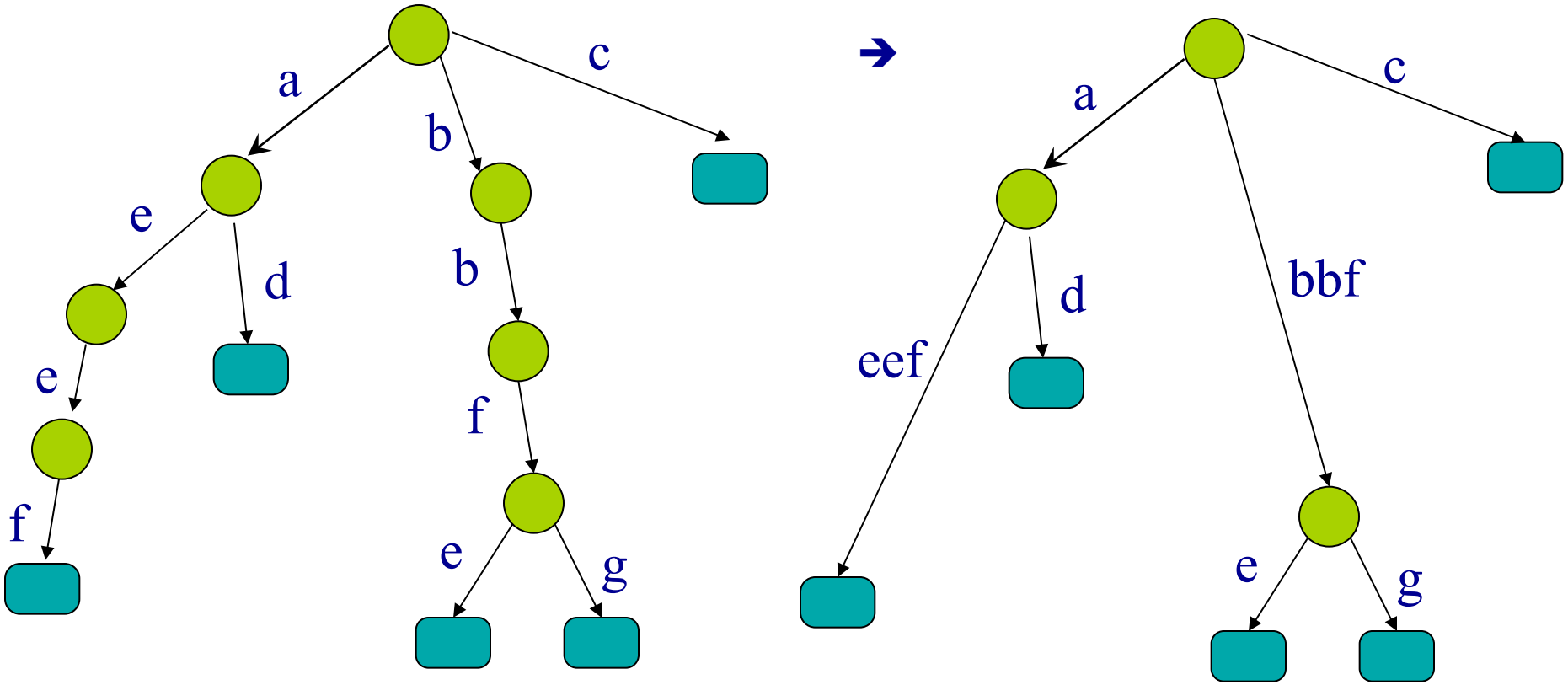
Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

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.

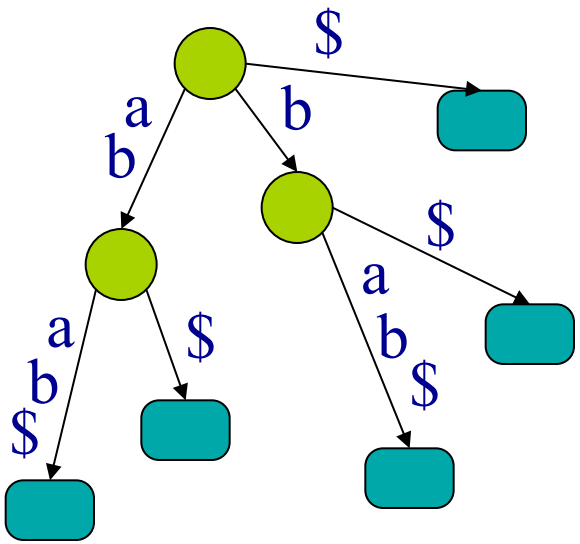
Observation:

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

Suffix tree (Example)

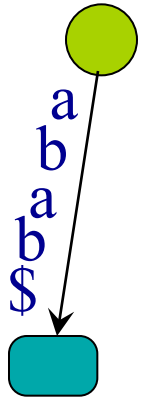
Let s=abab. A suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

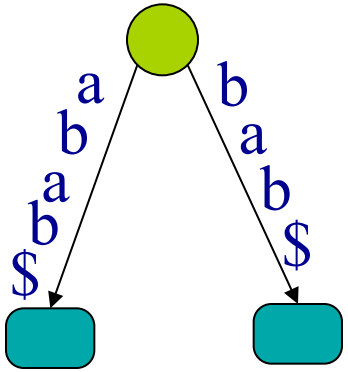


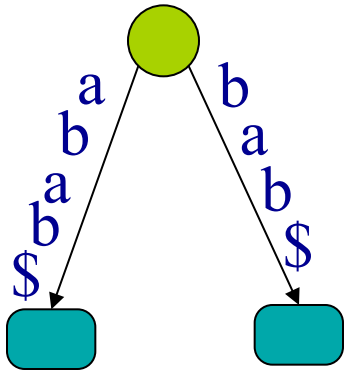
Trivial algorithm to build a Suffix tree

Put the largest suffix in

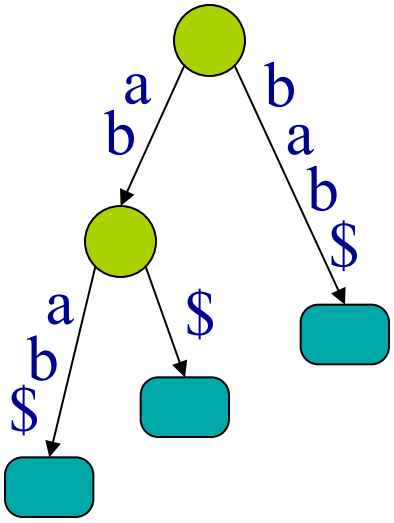


Put the suffix bab$ in

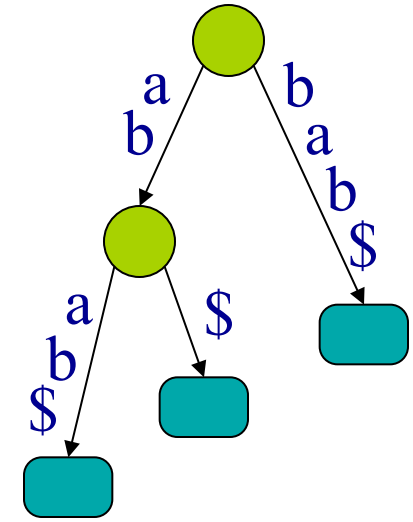


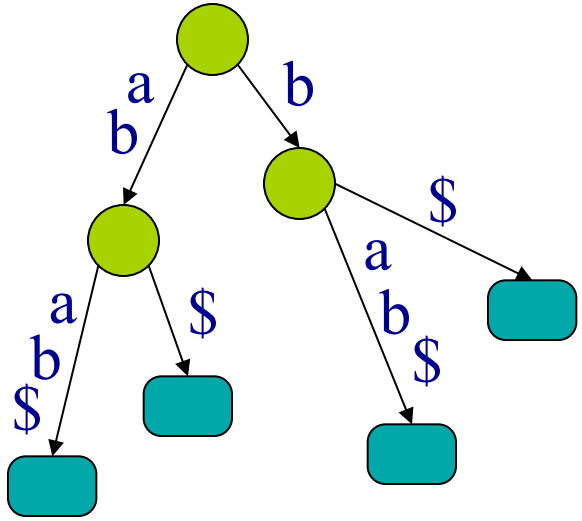


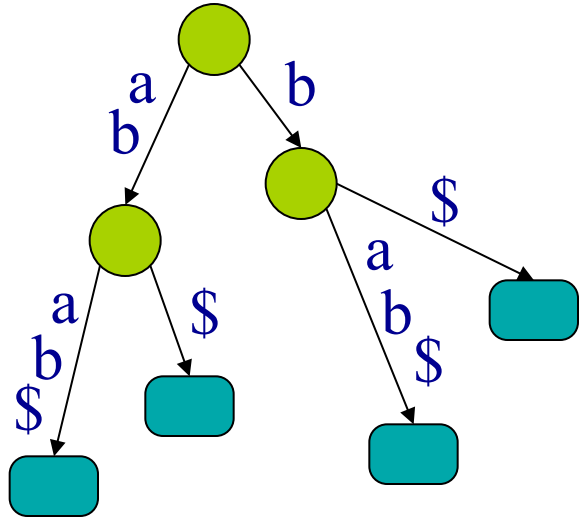
Put the suffix ab$ in



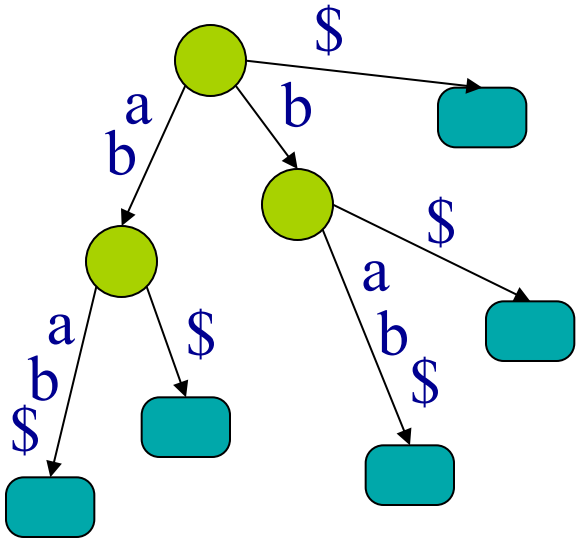
Put the suffix b$ in

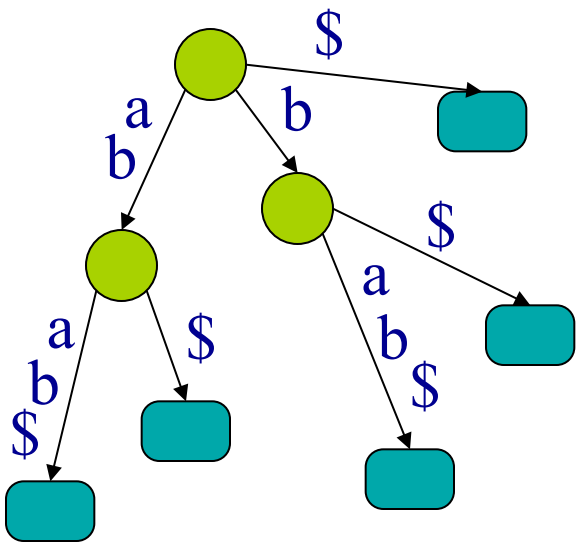




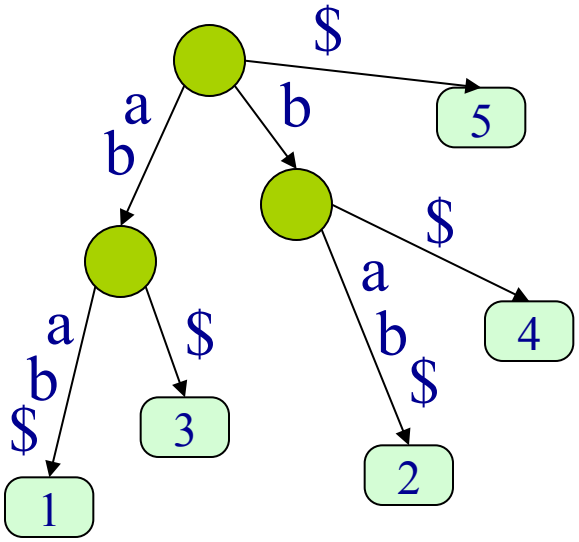


Put the suffix $ in





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



```
SIMPLE-SUFFIX-TREE-ALGORITHM(T)
  Create the root node, with empty string
  for i ← 1 to n do
    Traverse current tree from the root
    Match symbols in the edge label one-by-one with symbols in
        the current suffix, T_i
    if a mismatch occurs then
      Split the edge at the position of mismatch to create a new
        node, if need be
      Insert suffix T_i into the suffix tree at the position of mismatch
    end if
  end for
```

Analysis

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

But we can do it in O(n) time with Ukkonen algorithm

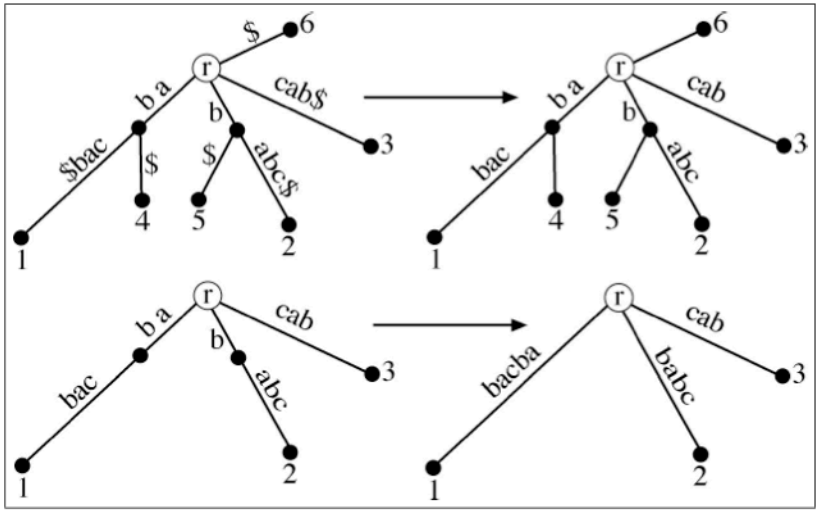
It needs : implicit suffix trees, suffix links …

Implicit suffix trees

1. Remove all the terminal symbols $
2. From the resulting tree, remove edges without label
3. Finally, from the resulting tree, remove nodes that do not have at least two children

T$={abcab$}

I(T) – implicit suffix tree



Suffix links

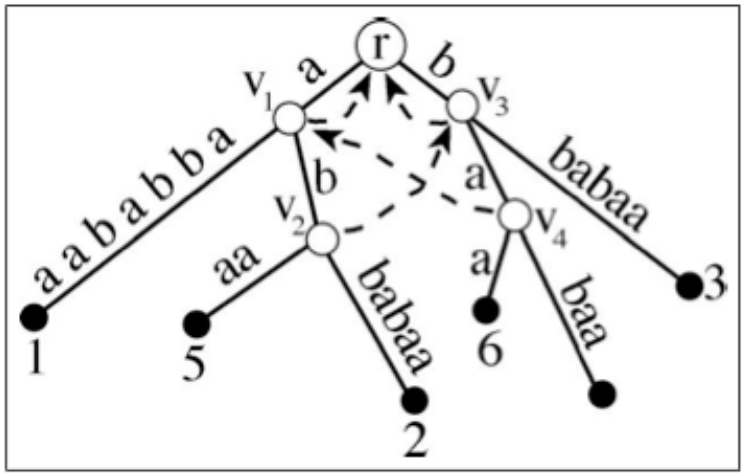
Let A be an arbitrary substring of T, including the possibility of being the empty string. Let z be a character of T.

Suppose there are two nodes v, w, the former with path-label zA and the latter with A.

A pointer from v to w is a ***suffix link****.*

Observation Every internal node has one suffix link.

Suffix links



Suffix links in the implicit suffix tree of string T = aabbabaa

What can we do with it ?

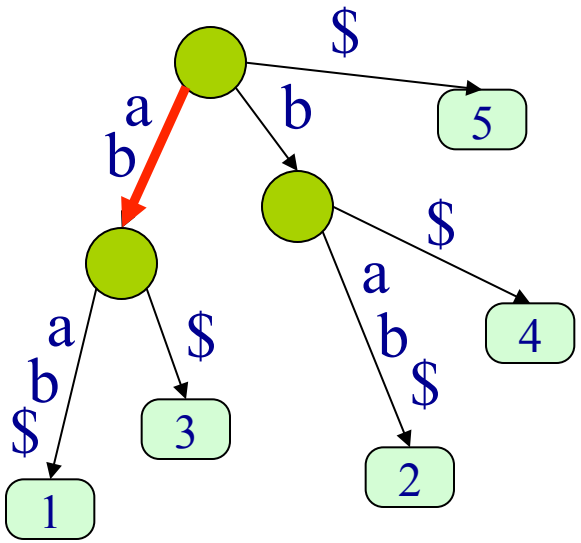
Exact string matching

Given a text T ($|T| = n$), preprocess it such that when a pattern P ($|P|=m$) arrives, you can quickly decide whether it occurs in T.

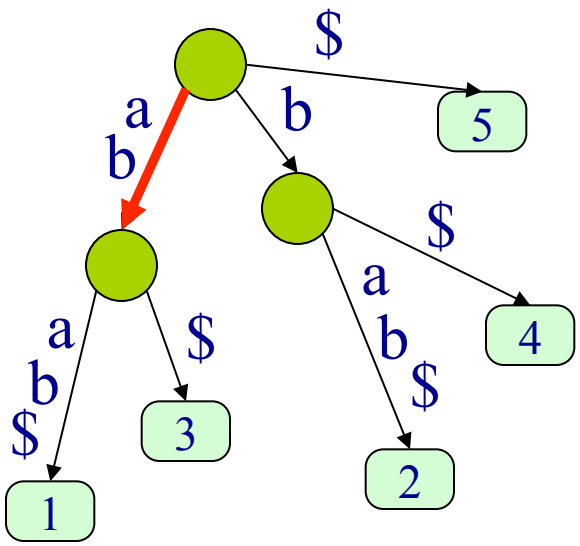
We may also want to find all occurrences of P in T.

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 tree 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

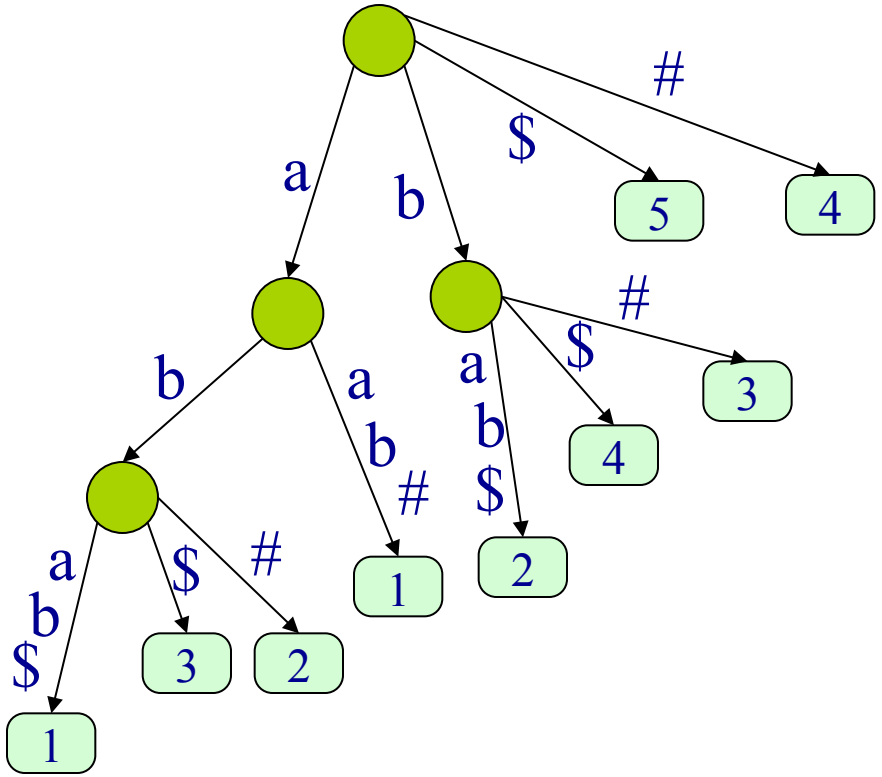
Generalized suffix tree (Example)

Let s_1=abab and s_2=aab, and a generalized suffix tree for s_1 and s_2

{

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

}

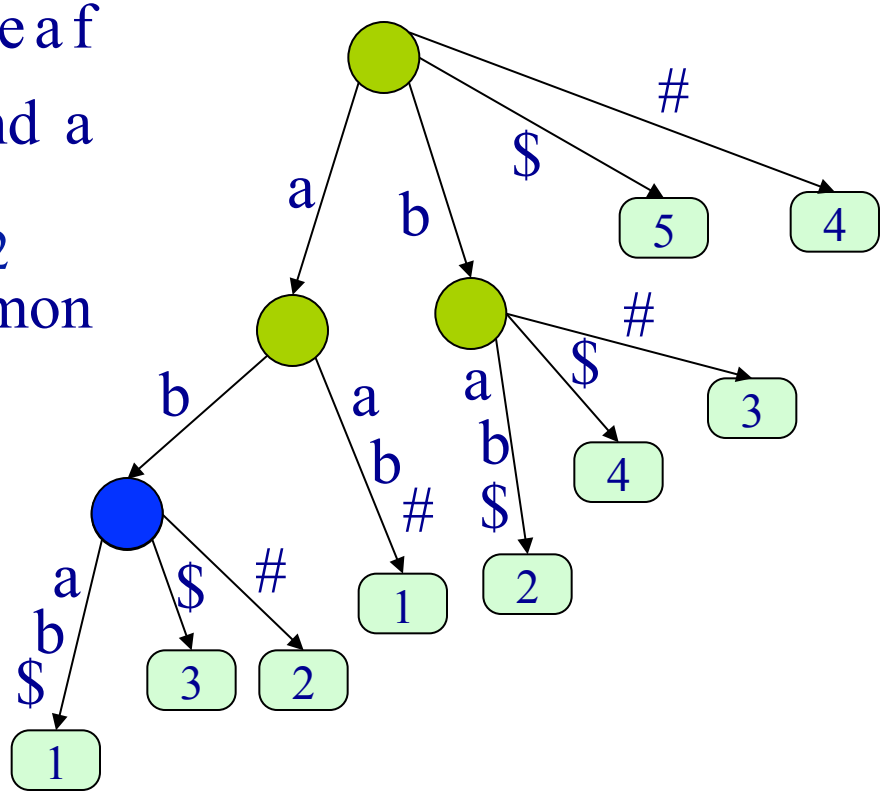


So what can we do with it ?

Match a pattern against a database of strings

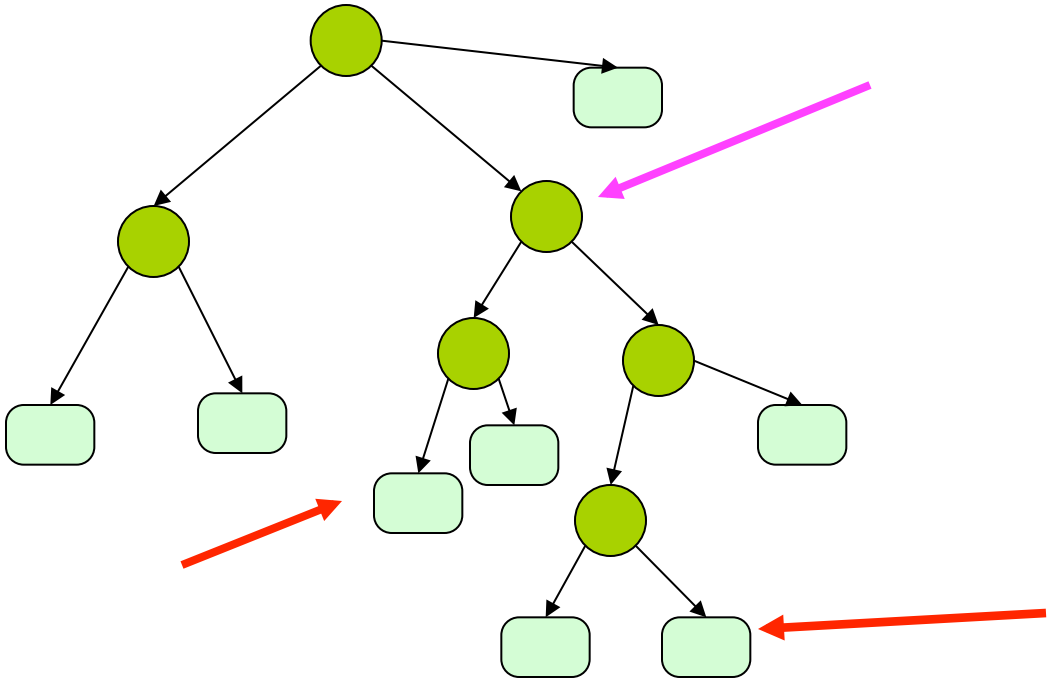
Longest common substring (of two strings)

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.



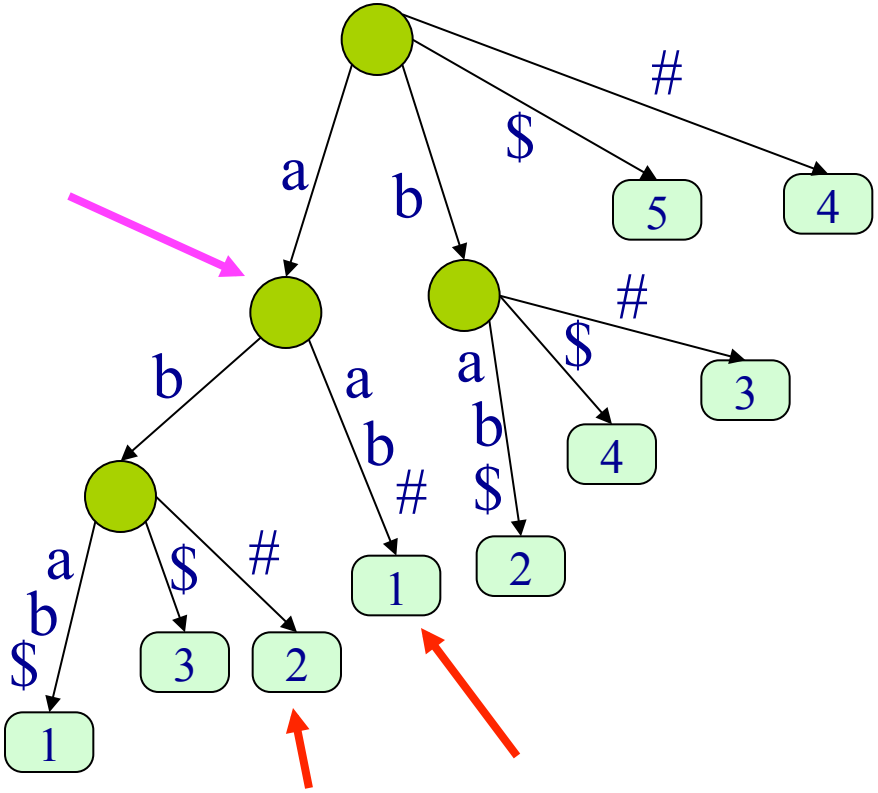
Lowest common ancestors

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



Why?

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



Finding maximal palindromes

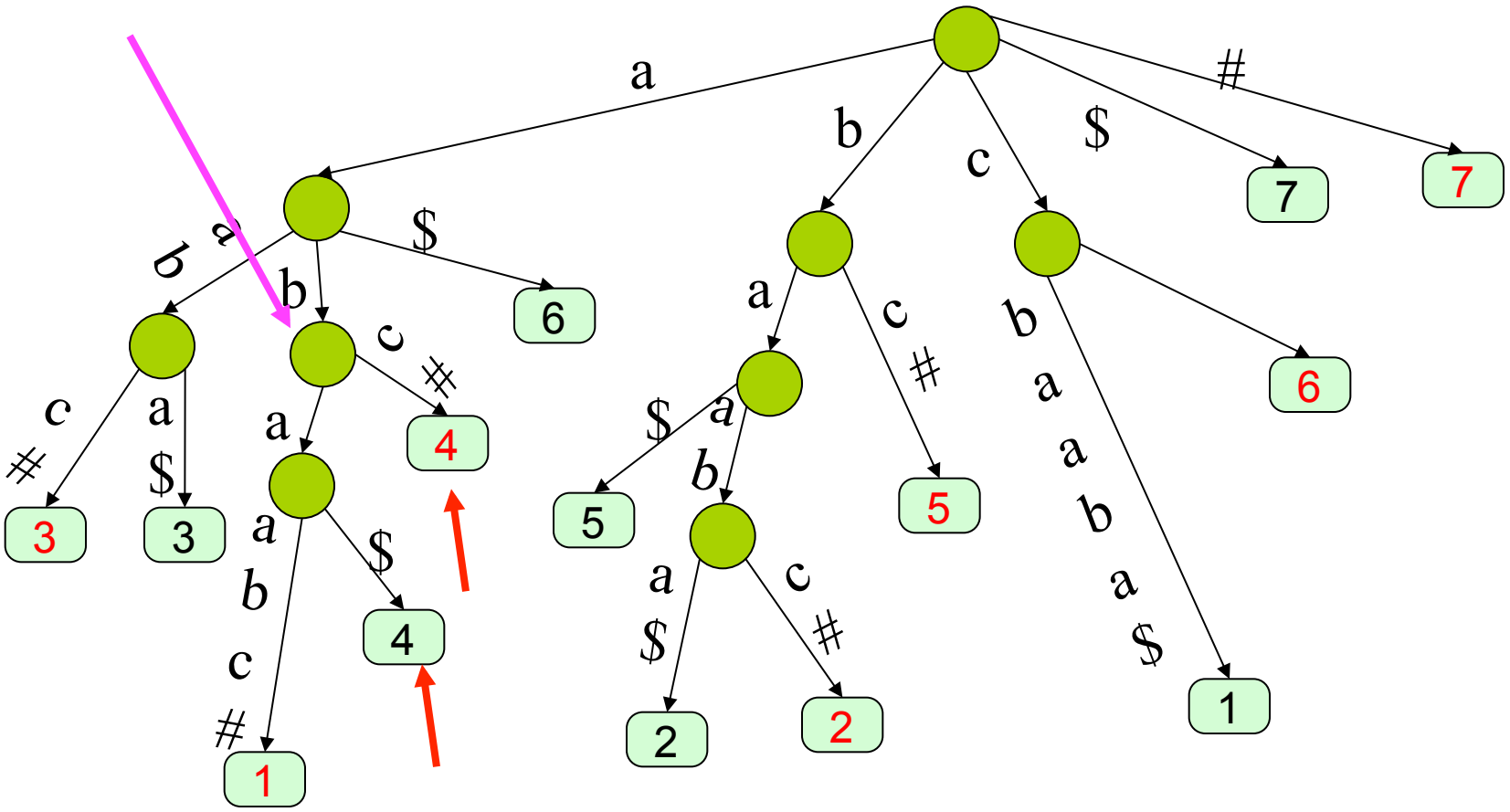
The maximal palindrome with center between i-1 and i in string s, is the LCP of the suffix at position i of s and the suffix at position m-i of s^r.

Let s = cbaaba$.

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#



Drawbacks

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

Suffix array (SA)

We loose some of the functionality but we save space.

Let s = abab

Sort the suffixes of s 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

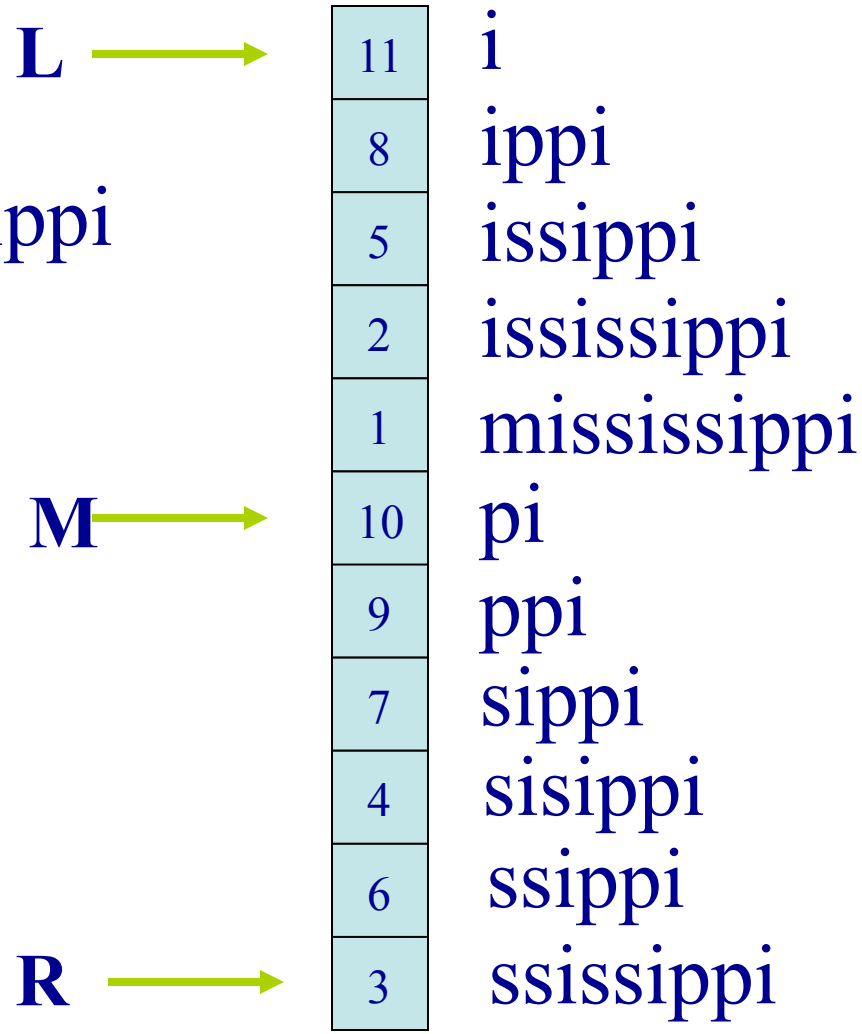
How do we search for a pattern ?

- 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(m logn) time

Example

Let S = mississippi

Let P = issa



Burrows-Wheeler Transform (BWT)

A way of permuting the characters of a string T into another string BWT (T)

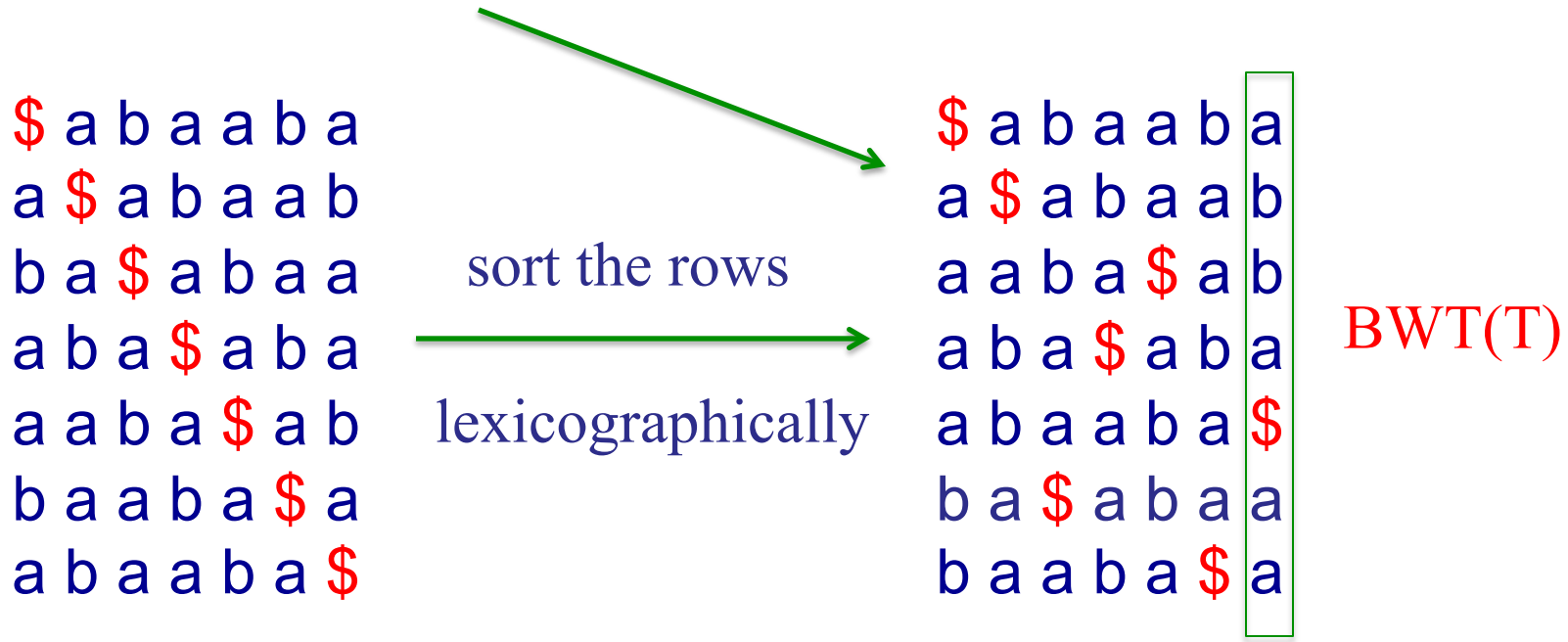
- Reversible permutation

- 2 main applications: compression and indexing

BWT via BWM

T = abaaba$

=> 6 x 6 matrix (BWM(T)) containing the rotations of T



BWT via suffix arrays (SA)

For BWM we sort T's rotations and for SA we sort T's suffixes. So, for i from 0 to |T|-1

SA[i]>0 ? BWT[i]=T[SA[i]-1] : $

BWM(T)	SA	Suffixes for SA
$ a b a a b a	6	$
a $ a b a a b	5	a $
a a b a $ a b	2	a a b a $
a b a $ a b a	3	a b a $
a b a a b a $	0	a b a a b a $
b a $ a b a a	4	b a $
b a a b a $ a	1	b a a b a $

LF Mapping

BWM(T) with ranks on T

F	L
$ a_0 b_0 a_1 a_2 b_1	a_3
a_3 $ a_0 b_0 a_1 a_2	b_1
a_1 a_2 b_1 a_3 $ a_0	b_0
a_2 b_1 a_3 $ a_0 b_0	a_1
a_0 b_0 a_1 a_2 b_1a_3	$
b_1 a_3 $ a_0 b_0 a_1	a_2
b_0 a_1 a_2 b_1 a_3 $	a_0

LF Mapping

the i^{th} occurrence of a character c in the last column has the same rank as the i^{th} occurrence of c in the first column.

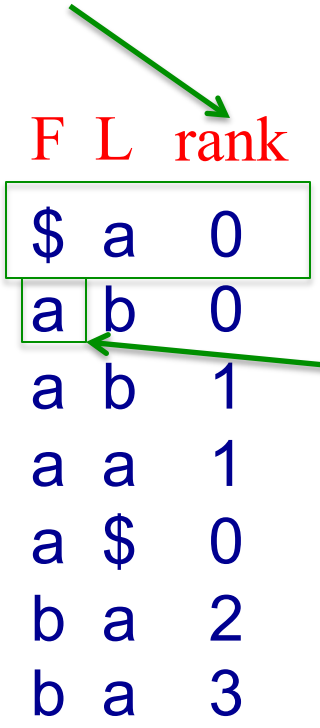
Example

a in the last column have ranks 3, 1, 2, 0

a in the first column have the same ranks

Reversing the BWT with LF Mapping

on the BWT(T)



F	L	rank
$	a	0
a	b	0
a	b	1
a	a	1
a	$	0
b	a	2
b	a	3

1. L[1] = a_0 is to the left of F[1]=$ in T

2. Find the char to the left of a_0 ⇔ find the row starting with a_0

Based on the LF Mapping, a_0 has rank 0 thus it corresponds to the first a in F

L[2] = b_0 is to the left of F[2]=a_0 in T
T = …ba$

… and so on for rows 6, 4, 3, 7, 5

Applications of the BWT

Finding all occurrences of P in T

- By applying LF Mapping repeatedly we find the range of rows prefixed by successively longer proper suffixes of P
- The size of the final range gives the number of times P occurs in T (if empty, P does not occur in T)

Compression

BWT("tomorrow and tomorrow and tomorrow$")

= wwwdd nnoooaatttmmmrrrrrrooo $ooo