# Lecture 10: Suffix trees, suffix arrays, and their applications

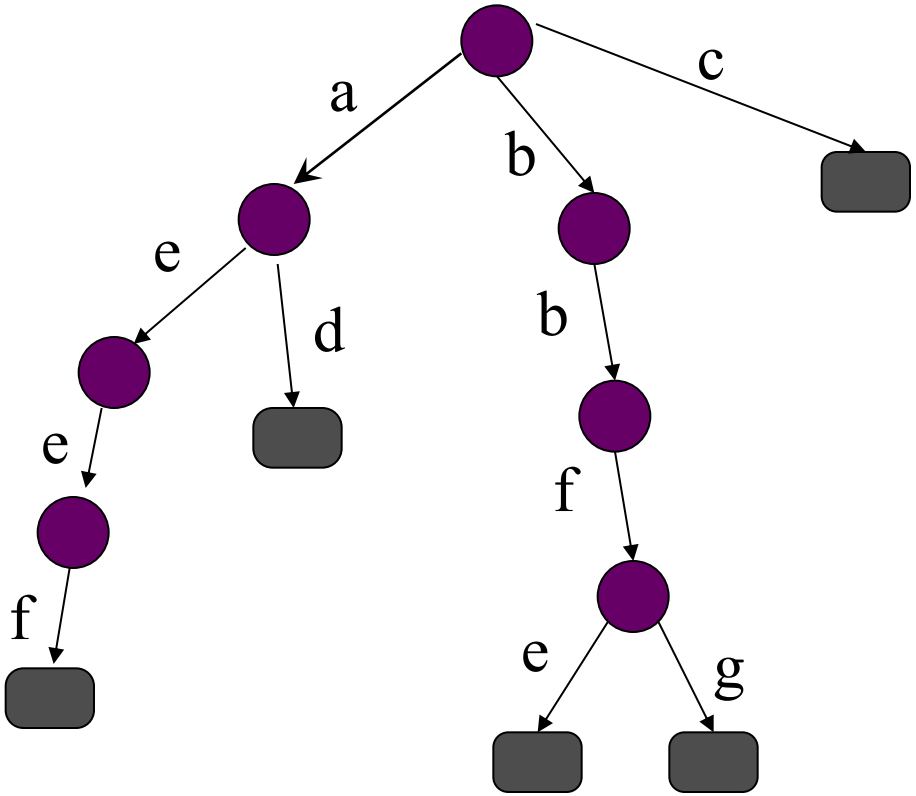
Spring 2020

March 12, 2020

# Trie

A tree representing a set of strings.

{aeef, ad, bbfe, bbfg, c}

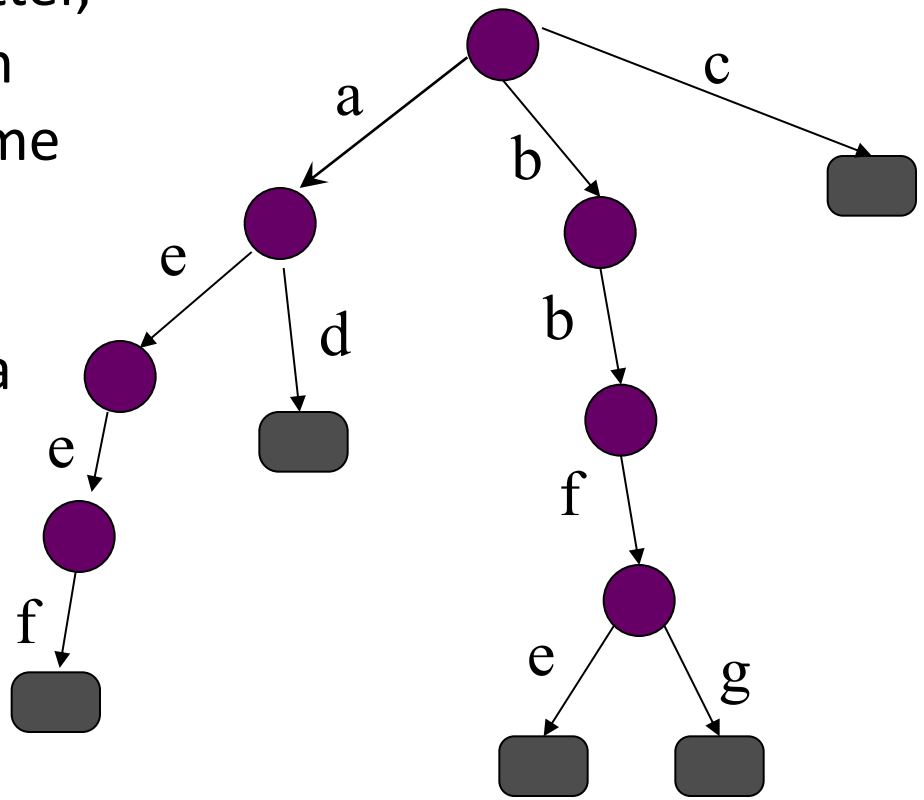


# Trie

Assume no string is a prefix of another

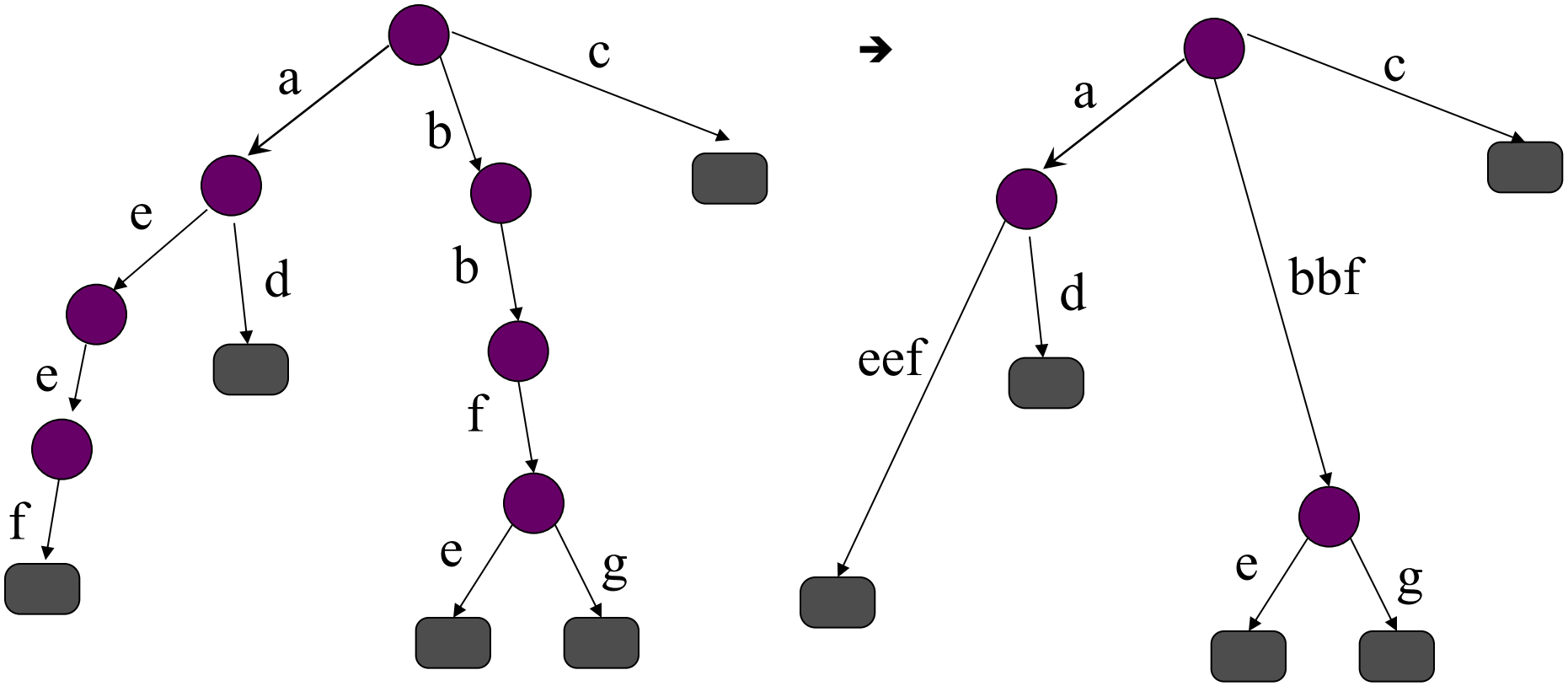
Each edge is labeled by a letter, no two edges outgoing from the same node have the same label.

Each string corresponds to a leaf.



# Compressed Trie

Compress nodes with single outgoing edge, label edges by strings



# Suffix tree

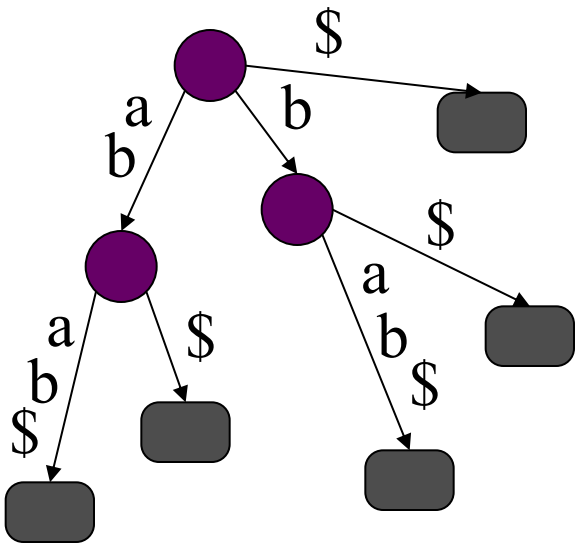
Given a string T, a suffix tree of T is a compressed trie of all suffixes of T

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

# Example

Suffix tree for the string T=abab$

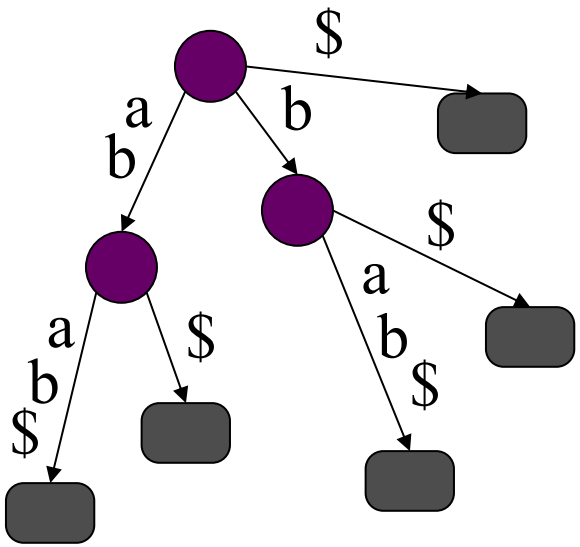
{$, b$, ab$,bab$, abab$}



# Suffix tree

## A suffix tree for an m-character string T:

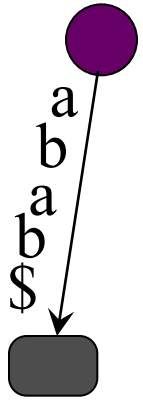
- A rooted directed tree with exactly m leaves numbered from 1 to m.
- Each internal node, other than root, has at least two children and each edge is labeled with a non-empty substring of T.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of T that starts at position i, that is, spells out T[i,…,m].



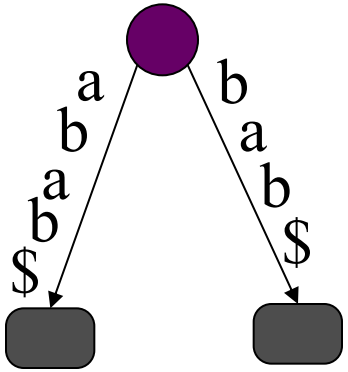
# Naïve algorithm to build suffix tree

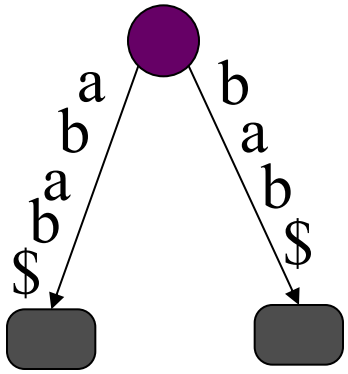
Constructing suffix tree for T=abab$

Insert the longest suffix

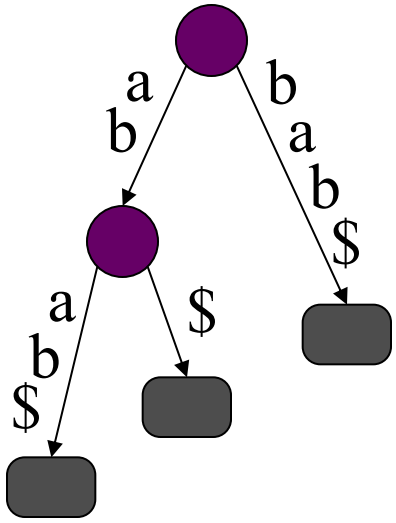


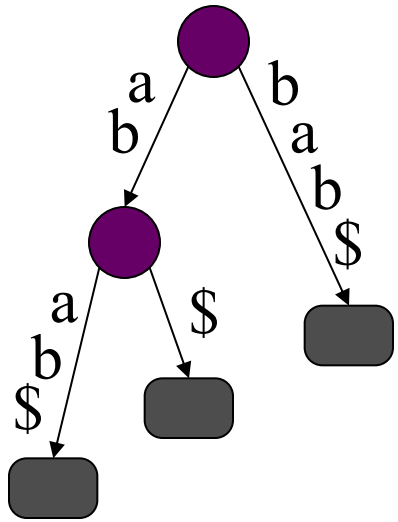

Next longest



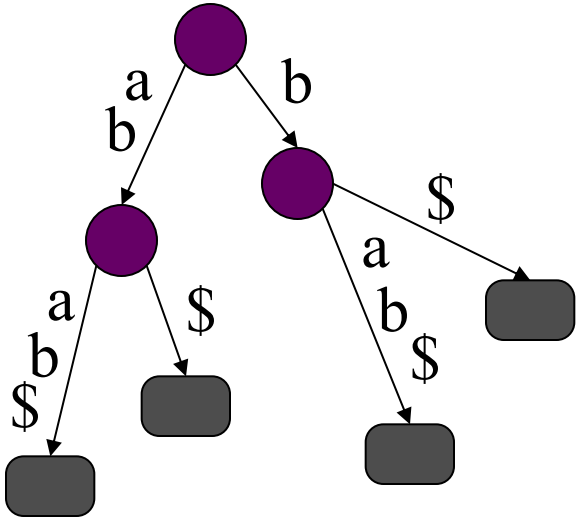


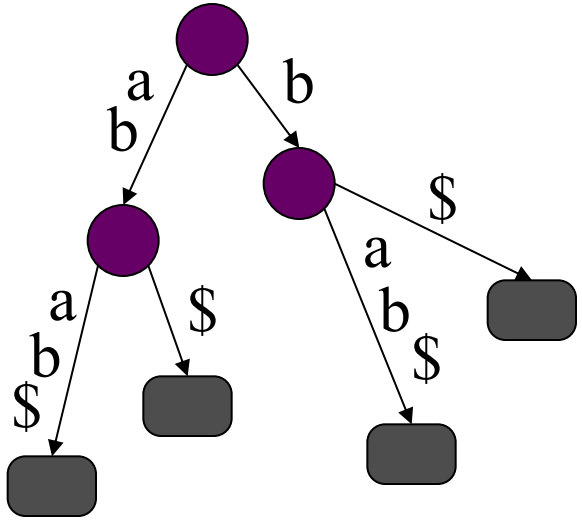

Insert the suffix ab$



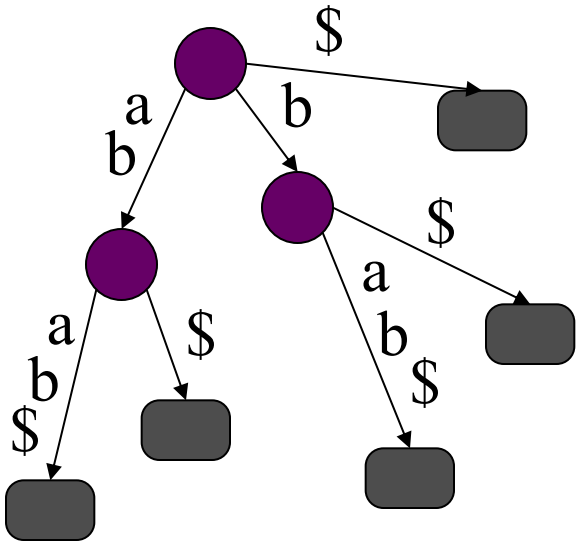


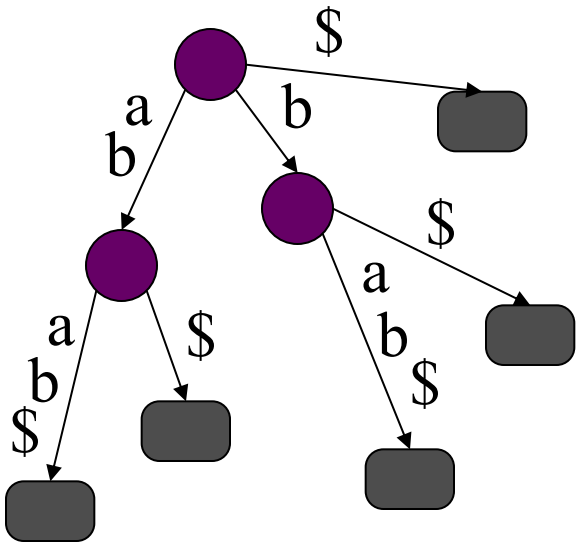

Insert the suffix b$



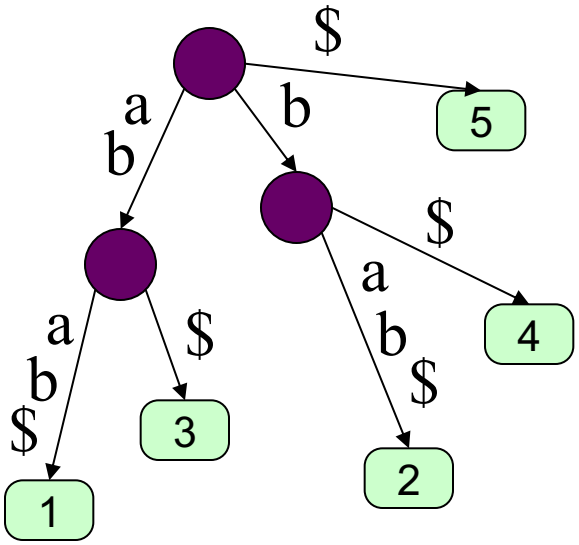



Insert the suffix $






label each leaf with the starting position of the corresponding suffix.



# Analysis

How long does it take to construct a suffix tree for a text of length m?

# Analysis

How long does it take to construct a suffix tree for a text of length m?

O(m) to insert a given suffix, so $O(m^2)$ overall.

There are algorithms that do it in O(m)! (Weiner 1973, McCreight 1976, Ukkonen 1975)

# What can we do with it?

## Exact string matching:

Given a Text T, |T| = m, and a string s, |s| = n, does s occur in T?

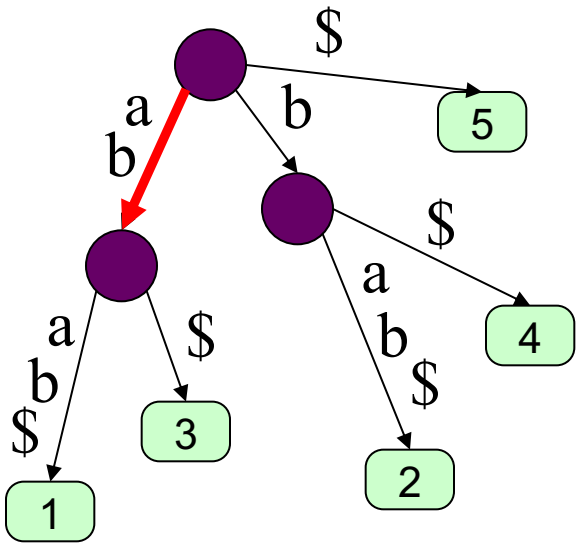
Naïve solution: search for occurrence of s at every position in T.

Running time?

Using suffix trees: Construct suffix tree for T.
How do we now check if s occurs in T?

# Exact string matching

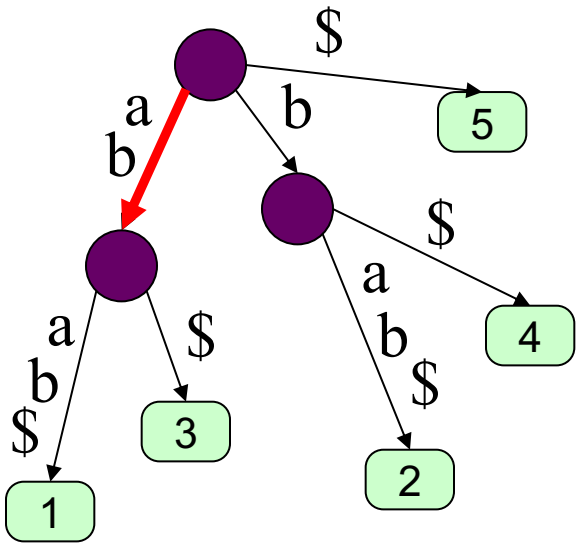
Is aba a substring of T=abab?

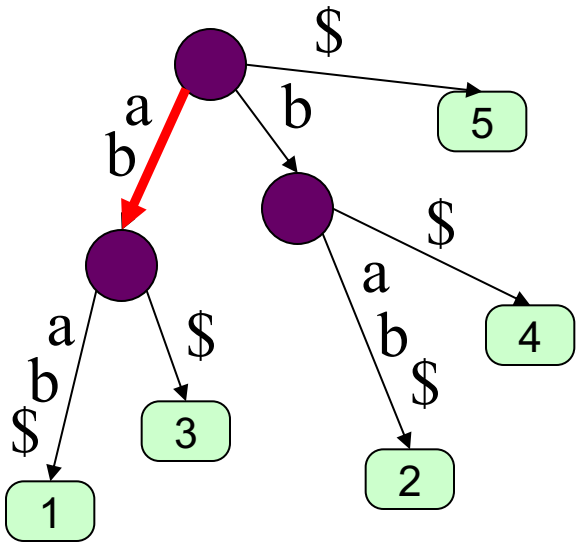


# Exact string matching

Is aba a substring of T=abab?

Traverse the tree using the string.






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

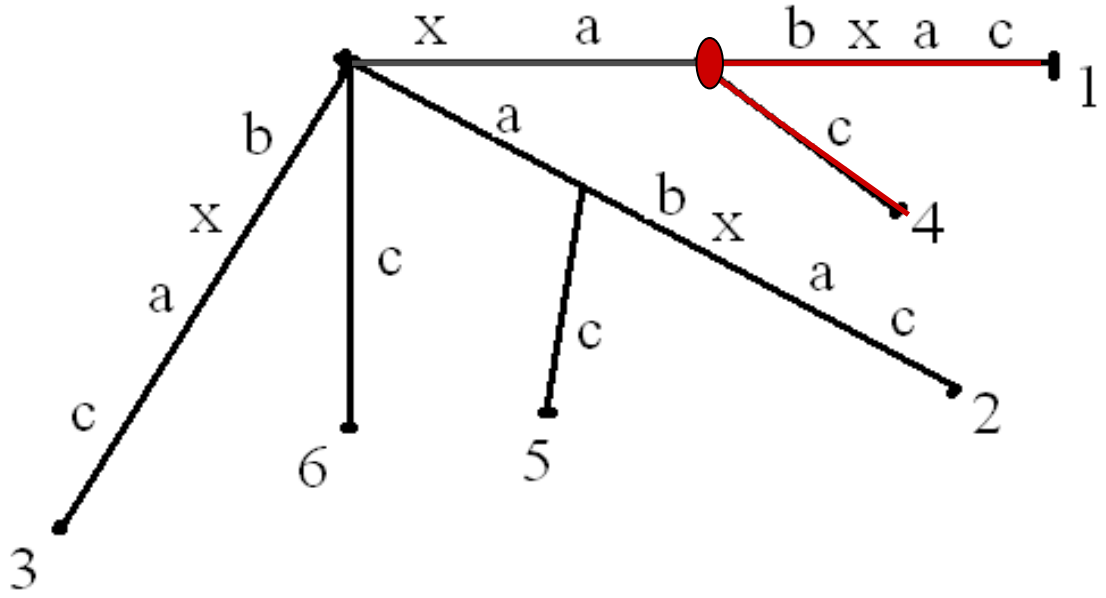
Each leaf in the subtree below the node we reach corresponds to an occurrence.

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

# Another example

T: xabxac

s: xa



# Running time

- Build the suffix tree: O(m)
- Match P to the unique path: O(n)
- Traverse the tree below the last matching point: O(k), where k is the number of occurrences, i.e., the number of leaves below the last matching point.
- Overall O(m+n+k).

# A collection of strings

Problem: Matching a string against a database of strings

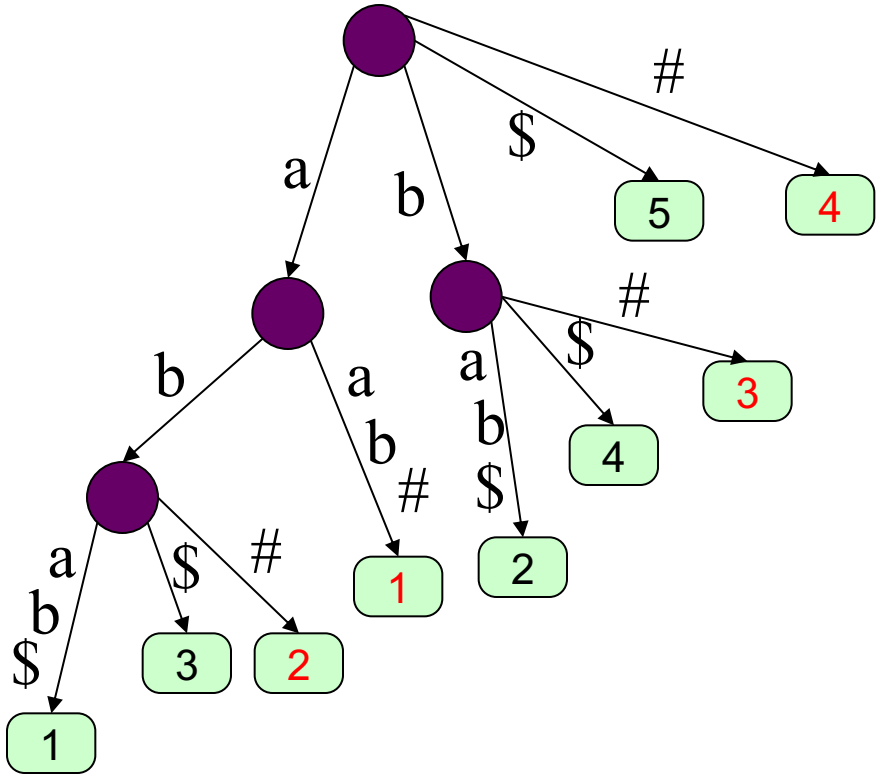
# Generalized suffix tree

A generalized suffix tree for $t_1$=abab and $t_2$=aab

{

| | |
|---|---|
| $ | # |
| b$ | b# |
| ab$ | ab# |
| bab$ | aab# |
| abab$ | |

}



# Generalized suffix tree

Given a **set** of strings T a generalized suffix tree of T is a compressed trie of all suffixes of $t \in T$.

To associate each suffix with a unique string in T add a different special character to each string in T.

# More application

- Inexact string matching; used in motif finding (e.g. the Weeder algorithm)
- Longest common substring
- Repeats/tandem repeats
- Maximal palindromes

# Drawbacks

Suffix trees consume a lot of space:

It is O(m) but the constant is quite big. Cannot fit a large mammalian genome in memory.

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. Otherwise, use a linked list, and traversal time depends on $|\Sigma|$.

# Suffix arrays

Use less space, but not as fast.

Let T = abab.

Sort the suffixes lexicographically:

ab, abab, b, bab

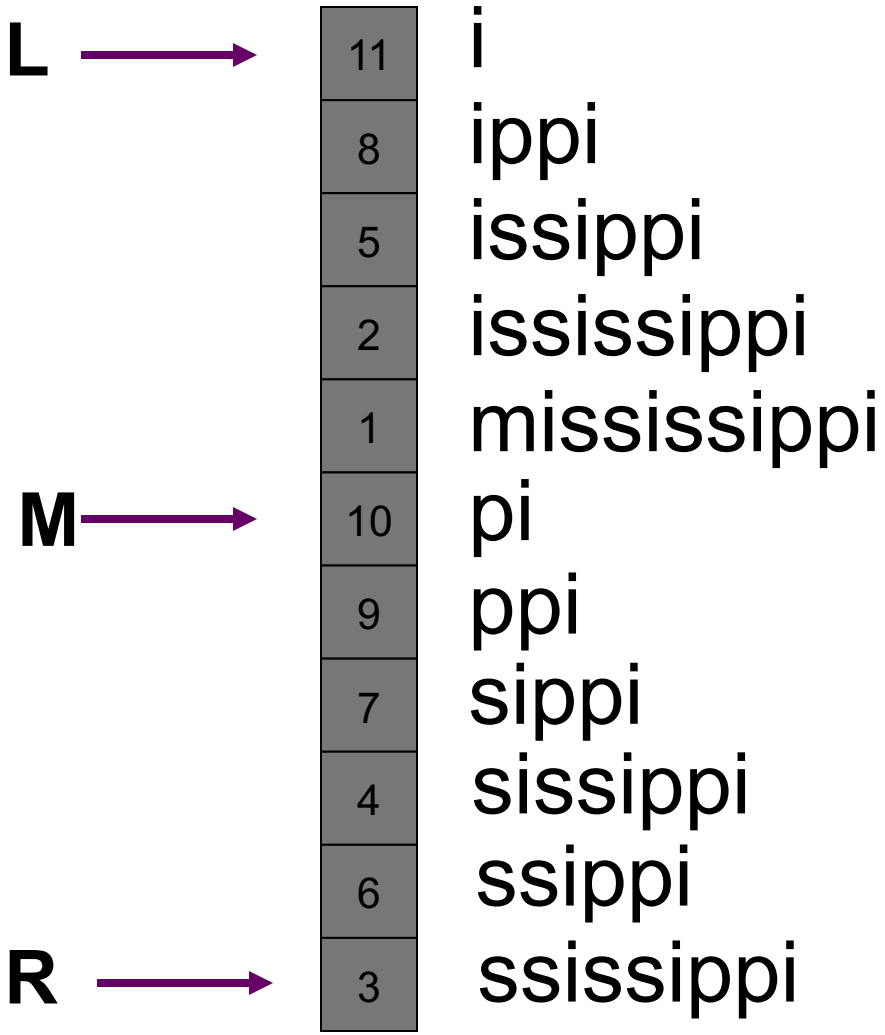
The suffix array gives the indices of the suffixes in sorted order.

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

# Example

T = mississippi

s = issa



# Searching a suffix array

If s occurs in T then all its occurrences are consecutive in the suffix array.

Do a binary search on the suffix array.

Takes O(n log m) time.

Note the connection with BWT. Essentially, BWT[i] is S[SA[i]-1 mod length] for sequence S and suffix array SA of S.

# Constructing a suffix array

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

But if our objective was to save space, do it directly: O(mr log m) time, r = longest repeat length.