

## 06 – Suffix Trees (1)



# Text search

---

Various scenarios:

## Static texts

- Literature databases
- Library systems
- Gene databases
- World Wide Web

## Dynamic texts

- Text editors
- Symbol manipulators

Algorithms by Knuth, Morris & Pratt and Boyer & Moore

# Properties of suffix trees

---

## Search index

for a **text**  $\sigma$  in order to search for several **patterns**  $\alpha$ .

## Properties:

1. **Substring searching** in time  $O(|\alpha|)$ .
2. **Queries to  $\sigma$  itself**, e.g.:  
Longest substring of  $\sigma$  that occurs at least twice.
3. **Prefix search**: all positions in  $\sigma$  with prefix  $\alpha$ .

# Properties of suffix trees

---

4. **Range search:** all locations (substrings) in  $\sigma$  belonging to an interval  $[\alpha, \beta]$  with  $\alpha \leq_{\text{lex}} \beta$ , e.g.

abrakadabra, acacia  $\in$  [abc, acc],  
abacus  $\notin$  [abc, acc] .

5. **Linear complexity:**

Space requirement and construction time in  $O(|\sigma|)$ .

# Tries

---

Alphabet  $\Sigma$ , set  $S$  of keys,  $S \subset \Sigma^*$

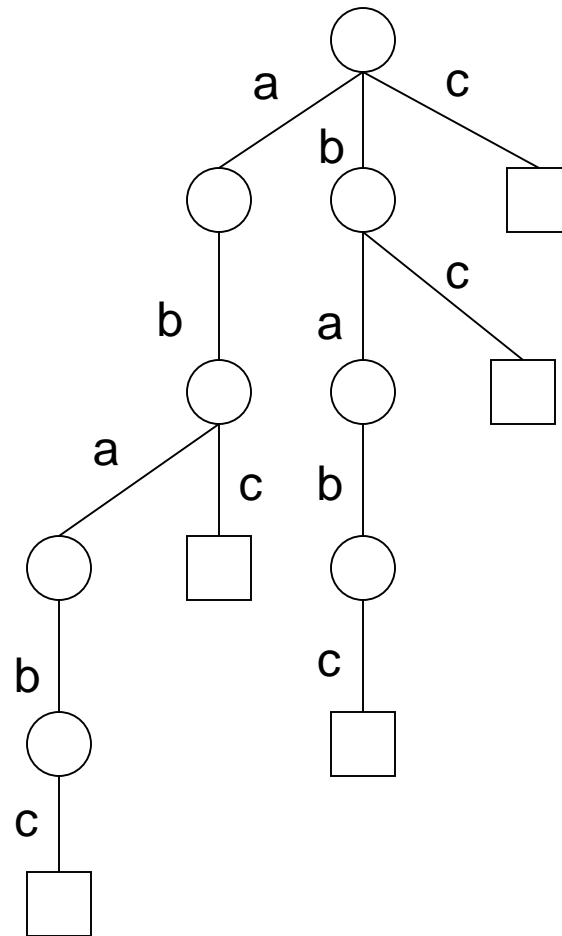
**Key:** string in  $\Sigma^*$

**Trie:** A tree representing a set of keys.

**Edge** of a trie  $T$ : labeled with a **single** character of  $\Sigma$

**Neighboring edges** (edges that lead to different children of a node):  
labeled with **different** characters

**Example:**



A **leaf** represents a key:

The corresponding key is the **string** consisting of the edge labels along the path from the root to the leaf.

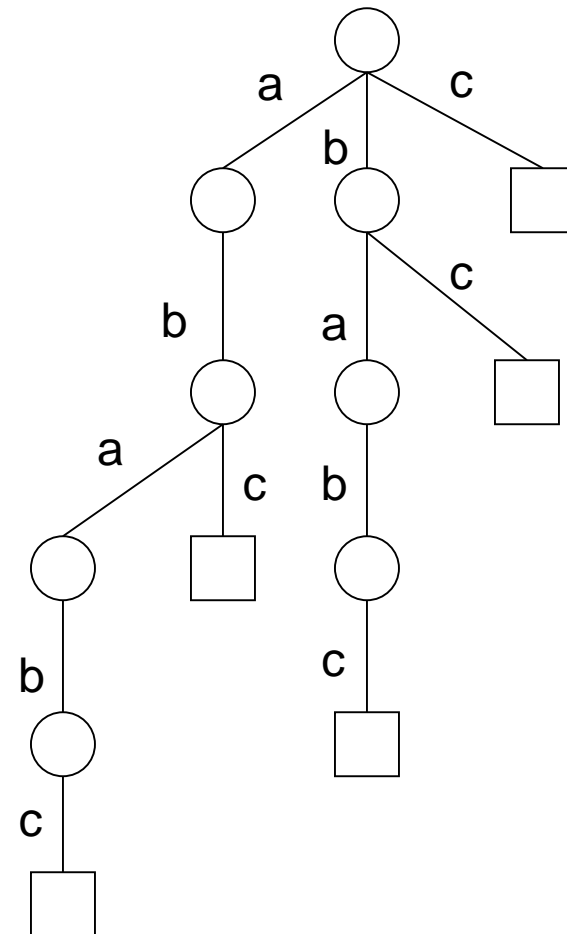
**Keys are not stored in nodes!**

# Suffix tries

Trie representing all suffixes of a string

**Example:**  $\sigma = ababc$

suffixes:  $ababc = \text{suf}_1$   
 $babc = \text{suf}_2$   
 $abc = \text{suf}_3$   
 $bc = \text{suf}_4$   
 $c = \text{suf}_5$





# Suffix tries

---

Nodes of a suffix trie  $\triangleq$  substrings of  $\sigma$

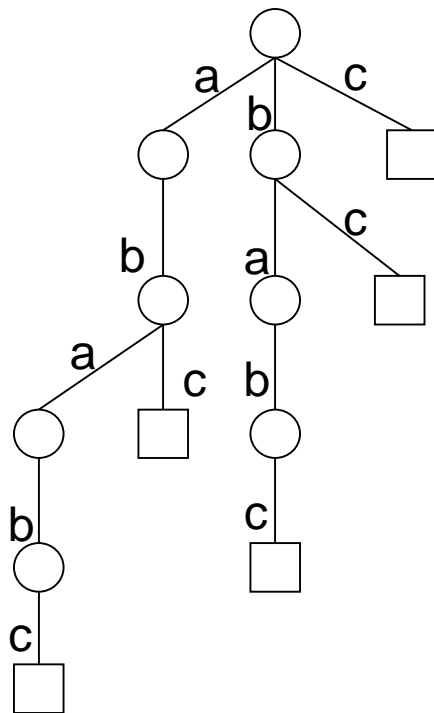
Each substring of  $\sigma$  is represented by a node.

Let  $\sigma = a^n b^n$ . Then there are  $n^2 + 2n + 1$  different substrings (or internal nodes).

$\Rightarrow$  space requirement is  $O(n^2)$

# Suffix tries

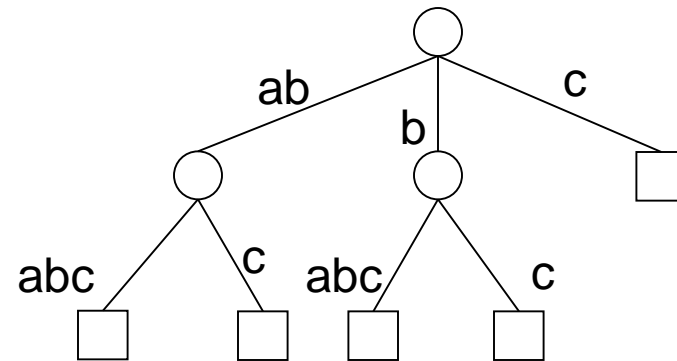
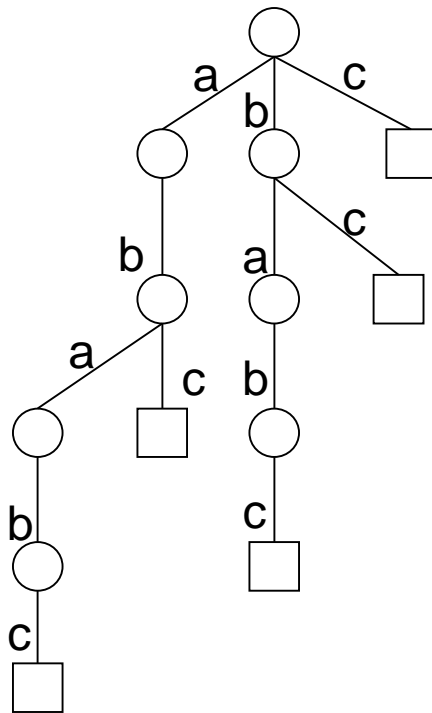
A suffix trie  $T$  satisfies some of the desired properties:



1. **String matching for  $\alpha$** : Following the path with edge labels  $\alpha$  takes  $O(|\alpha|)$  time.  
leaves of the subtree  $\hat{=}$  occurrences of  $\alpha$
2. **Longest substring** occurring at least twice:  
internal node with maximum depth having at least two children
3. **Prefix search**: All occurrences of strings with prefix  $\alpha$  are represented by the nodes of the subtree rooted at the internal node corresponding to  $\alpha$ .

# Suffix trees

A suffix tree is obtained from a suffix trie by **contracting unary nodes**.



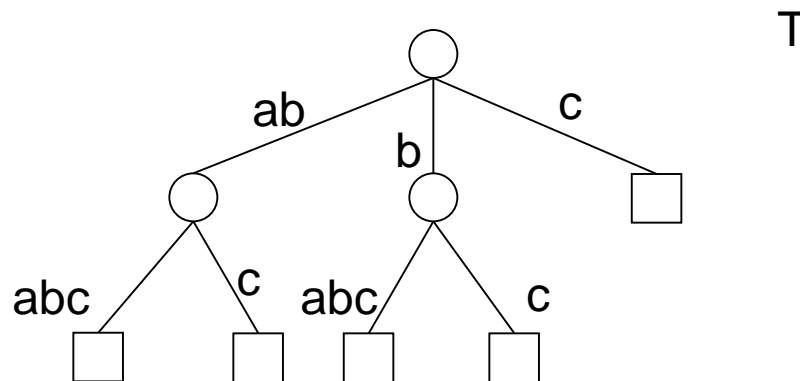
suffix tree = contracted suffix trie

# Internal representation of suffix trees

## Child-sibling representation

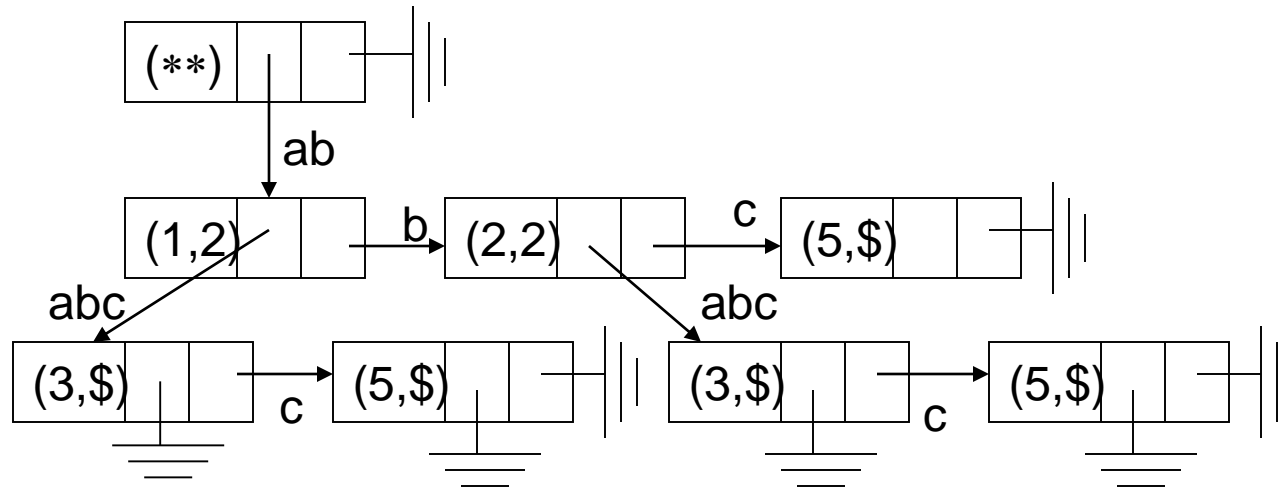
substring: pair of numbers  $(i,j)$

Example:  $\sigma = ababc$



# Internal representation of suffix trees

Example:  $\sigma = ababc$



node  $v = (v.l, v.u, v.c, v.s)$

Further pointers (suffix links) are added later.

# Properties of suffix trees

---

- (S1) No suffix of  $\sigma$  is prefix of another suffix.  
This holds if the last character of  $\sigma$  is  $\$ \notin \Sigma$ .

## Search:

- (T1) edge  $\triangleq$  non-empty substring of  $\sigma$ .
- (T2) neighboring edges:  
corresponding substrings start with different characters

# Properties of suffix trees

## Size

(T3) each internal node ( $\neq$  root) has at least two children

(T4) leaf  $\triangleq$  (non-empty) suffix of  $\sigma$ .

Let  $n = |\sigma| > 1$ .

(T4)  
 $\Rightarrow$  number of leaves =  $n$

(T3)  
 $\Rightarrow$  number of internal nodes  $\leq n - 1$

$\Rightarrow$  space requirement in  $O(n)$

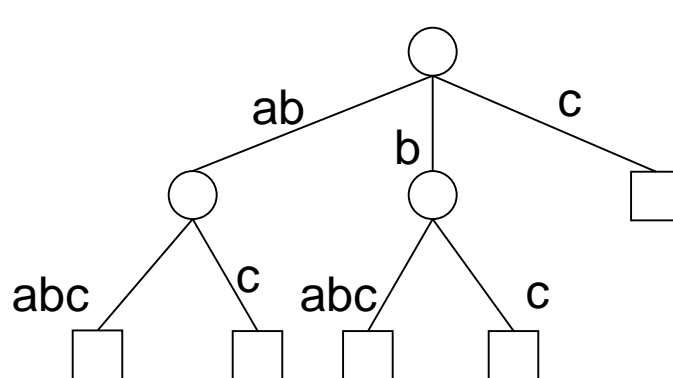
# Construction of suffix trees

## Definitions

**Partial path:** Path from the root to a node in  $T$ .

**Path:** A partial path ending at a leaf.

**Location** of a string  $\alpha$ : Node where the partial path corresponding to  $\alpha$  ends (if it exists).



$T$

$\alpha = bab$  (has no location)

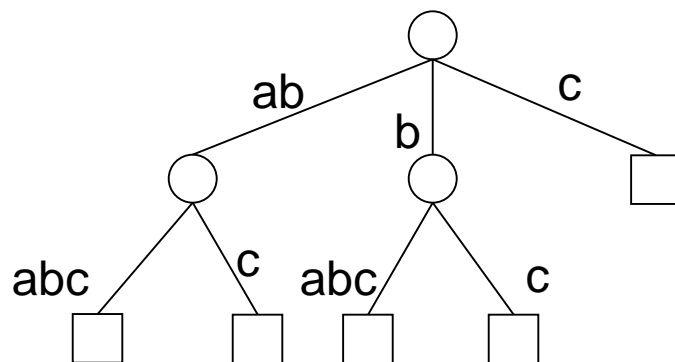


# Construction of suffix trees

**Extension** of a string  $\alpha$ : string with prefix  $\alpha$

**Extended location** of a string  $\alpha$ : location of the **shortest extension** of  $\alpha$  whose location is defined

**Contracted location** of a string  $\alpha$ : location of the **longest prefix** of  $\alpha$  whose location is defined



T

$\alpha = \text{bab}$  (has no location)

# Construction of suffix trees

---

## Definitions:

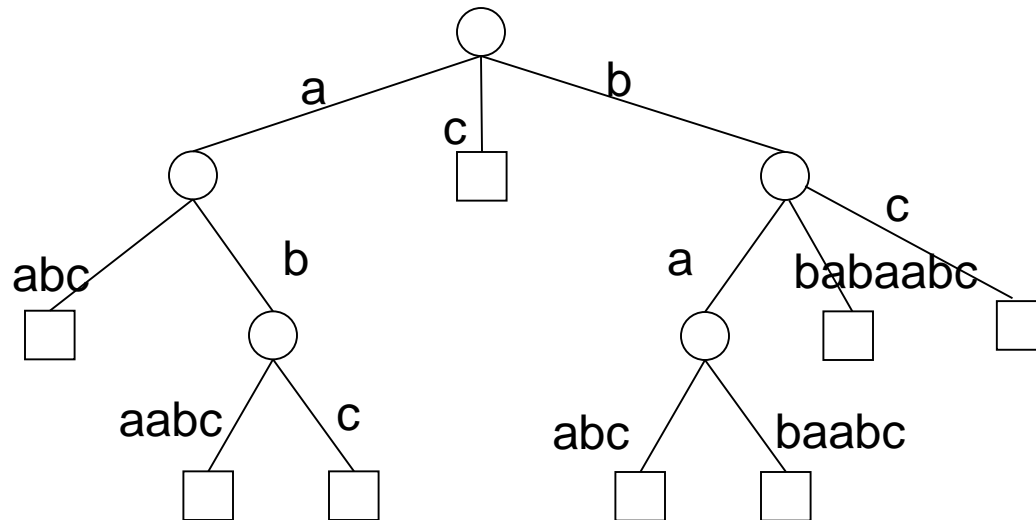
*suf<sub>i</sub>*: suffix of  $\sigma$  beginning at position  $i$ , e.g.  $suf_1 = \sigma$ ,  $suf_n = \$$ .

*head<sub>i</sub>*: longest prefix of  $suf_i$  that is also a prefix of  $suf_j$  for some  $j < i$ .

Example:             $\sigma = \text{bbabaabc}$   
                       $suf_4 = \text{baabc}$   
                       $head_4 = \text{ba}$

# Construction of suffix trees

$\sigma = \text{bbabaabc}$



# Naive suffix tree construction

Start with the empty tree  $T_0$ .

The tree  $T_{i+1}$  is constructed from  $T_i$  by inserting the suffix  $\text{ suf}_{i+1}$ .

**Algorithm** *suffix-tree*

**Input:** string  $\sigma$

**Output:** suffix tree  $T$  for  $\sigma$

1  $n := |\sigma|$ ;  $T_0 := \emptyset$ ;

2 **for**  $i := 0$  **to**  $n - 1$  **do**

3     insert  $\text{ suf}_{i+1}$  into  $T_i$ , store the result in  $T_{i+1}$  ;

4 **endfor**;

# Naive suffix tree construction

---

All suffixes  $suf_j$  with  $j \leq i$  have a location in  $T_i$ .

→  $head_{i+1}$  = longest prefix of  $suf_{i+1}$  that is a prefix of  $suf_j$ , with  $j < i+1$

## Definition:

$tail_{i+1} := suf_{i+1} - head_{i+1}$  i.e.  $suf_{i+1} = head_{i+1} tail_{i+1}$ .

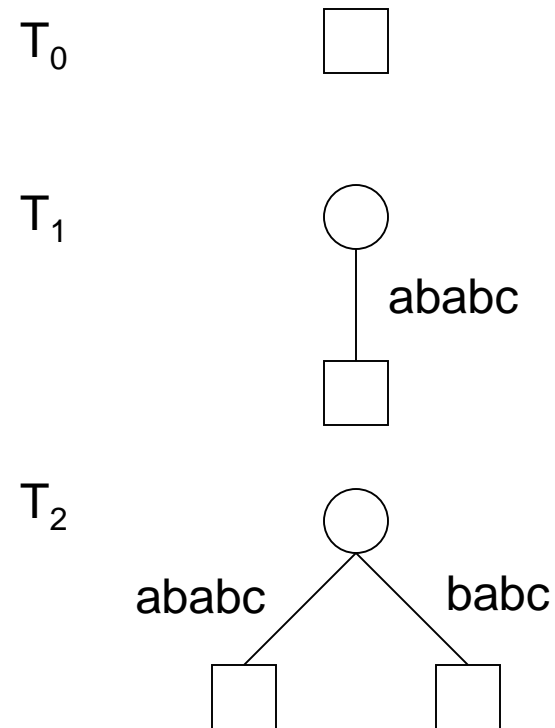
(S1)

⇒  $tail_{i+1} \neq \varepsilon$

# Naive suffix tree construction

Example:  $\sigma = ababc$

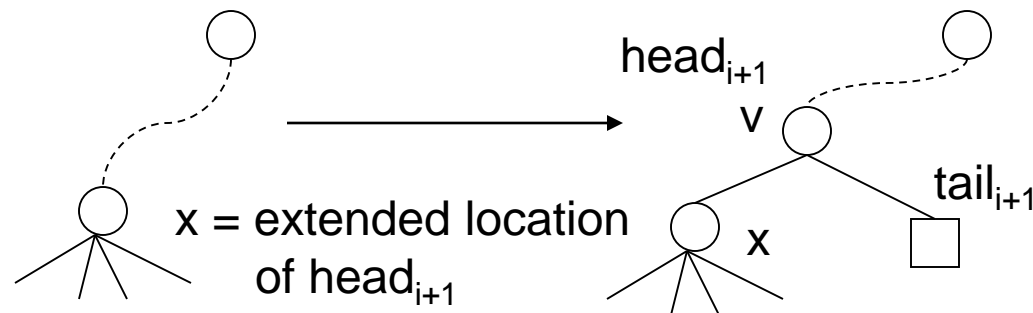
$\text{suf}_3 = abc$   
 $\text{head}_3 = ab$   
 $\text{tail}_3 = c$



# Naive suffix tree construction

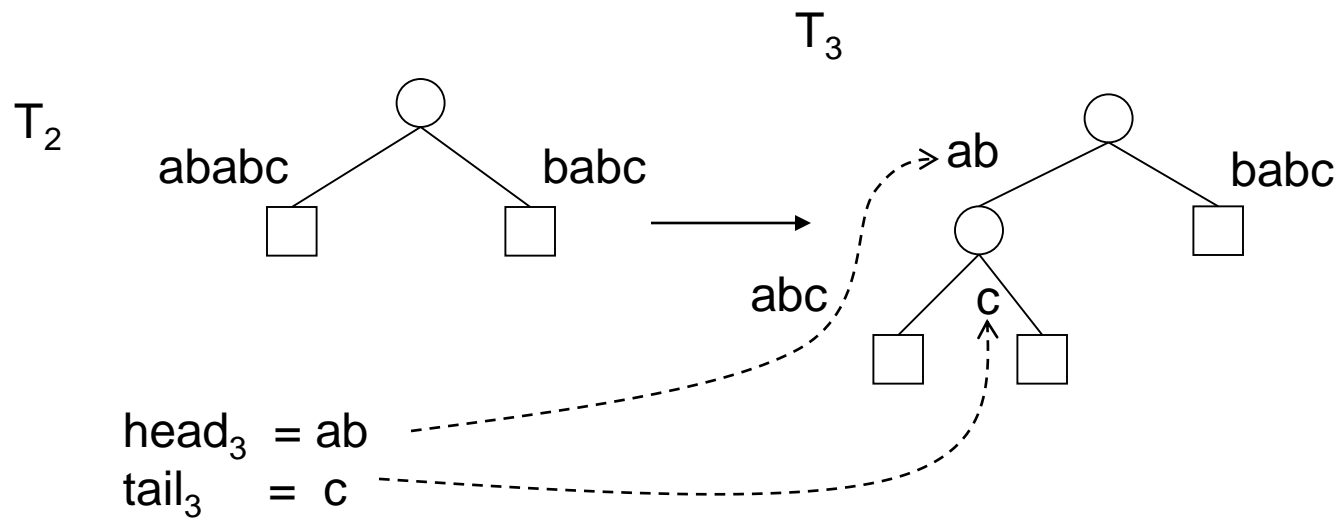
$T_{i+1}$  can be constructed from  $T_i$  as follows:

1. Determine the **extended location** of  $head_{i+1}$  in  $T_i$  and split the **last edge** leading to this location into two new edges by inserting a new node.
2. Insert a new leaf as location for  $suf_{i+1}$ .



# Naive suffix tree construction

Example:  $\sigma = ababc$





# Naive suffix tree construction

**Algorithm** *suffix-insertion*

**Input:** tree  $T_i$  and suffix  $suf_{i+1}$

**Output:** tree  $T_{i+1}$

```
1  $v :=$  root of  $T_i$ ;  
2  $j := i$ ;  
3 repeat  
4   find child  $w$  of  $v$  with  $\sigma_{w.l} = \sigma_{j+1}$ ;  
5   if  $w \neq \text{nil}$  then  
6      $k := w.l$ ;  $j := j + 1$ ;  
7     while  $k < w.u$  and  $\sigma_{k+1} = \sigma_{j+1}$  do  
8        $k := k + 1$ ;  $j := j + 1$ ;  
9     endwhile;  
10    if  $k = w.u$  then  $v := w$  endif;
```

# Naive suffix tree construction

---

```
11  endif  
12  until  $k < w.u$  or  $w = \text{nil}$ ;    /*  $v$  is the contracted location of  $\text{head}_{i+1}$  */  
13  insert the location of  $\text{head}_{i+1}$  and  $\text{tail}_{i+1}$  below  $v$  into  $T_i$ ;
```

Running time of *suffix-insertion* :  $O(n-i)$

Total time required for the naive construction:  $O(n^2)$

# The algorithm MCC

---

(McCreight, 1976)

Idea: Extended location of  $head_{i+1}$  in  $T_i$  is determined in **constant amortized** time. (Additional information required!)

When the extended location of  $head_{i+1}$  in  $T_i$  has been found:  
Creating a new node and splitting an edge takes  $O(1)$  time.

## Theorem 1

**Algorithm MCC** constructs a suffix tree for  $\sigma$  with  $|\sigma|$  leaves and at most  $|\sigma| - 1$  internal nodes in time  $O(|\sigma|)$ .

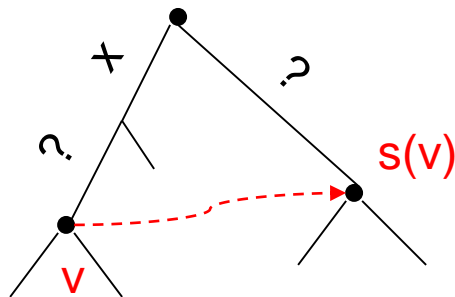
# Suffix links

## Definition:

Let  $x?$  be an arbitrary string where  $x$  is a **single character** and  $?$  some (possibly empty) substring.

For an internal node  $v$  with edge labels  $x?$  the following holds:

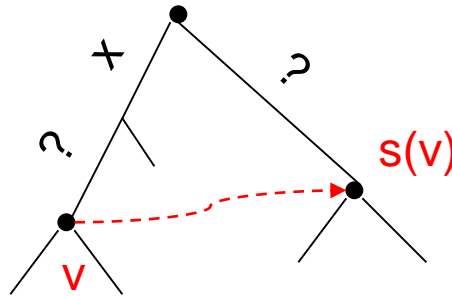
If there exists a node  $s(v)$  with edge label  $?$ , then there is a pointer from  $v$  to  $s(v)$  that is called a **suffix link**.



# Suffix links

The idea is as follows:

By following the suffix links, we **do not have** to start each **search** for a splitting point at the **root** node. Instead, we can use the suffix links in order to determine these nodes more efficiently, i.e. in **constant amortized time**.



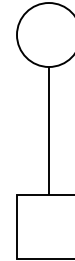
# Suffix tree: example

$T_0$



$\text{suf}_1 = \text{bbabaabc}$

$T_1$

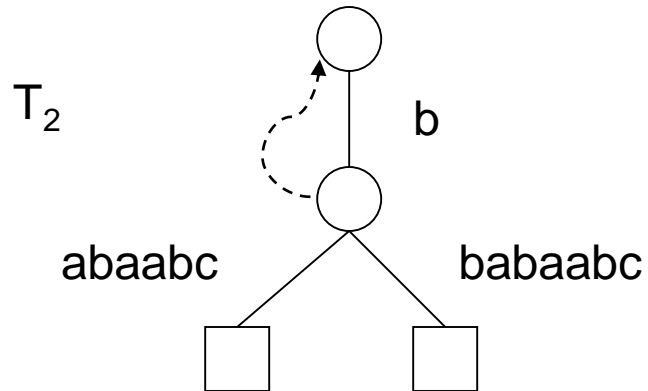


bbabaabc

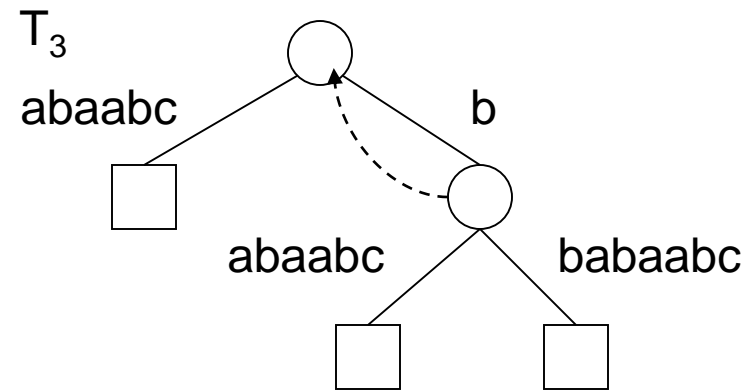
$\text{suf}_2 = \text{babaabc}$

$\text{head}_2 = \text{b}$

# Suffix tree: example

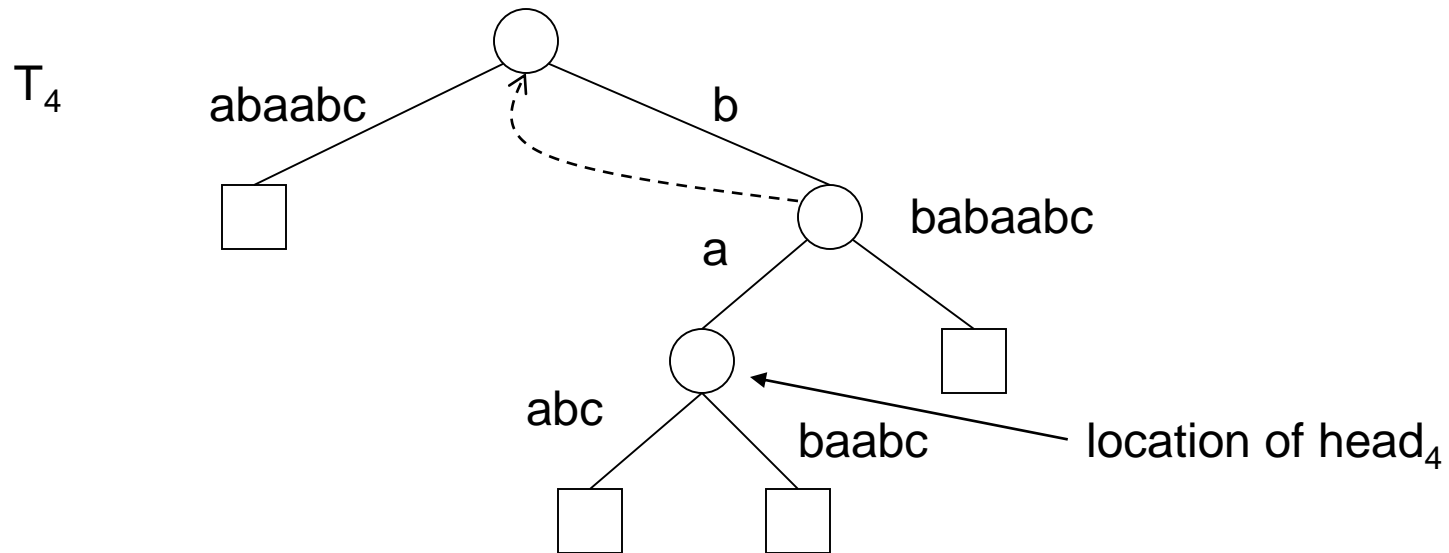


$\text{suf}_3 = \text{abaabc}$   
 $\text{head}_3 = \varepsilon$



$\text{suf}_4 = \text{baabc}$   
 $\text{head}_4 = \text{ba}$

# Suffix tree: example

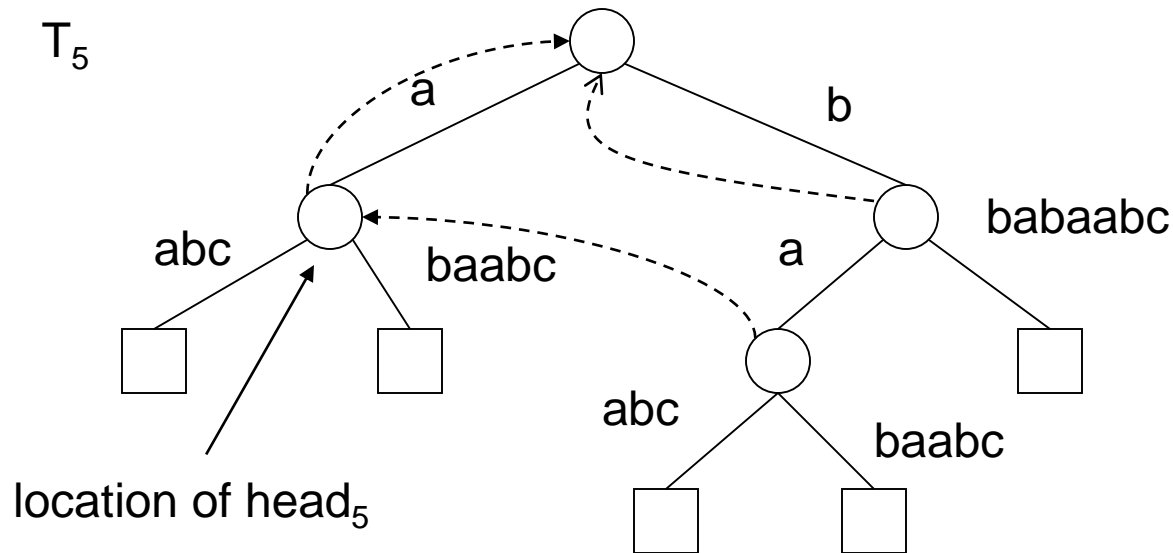


$\text{suf}_5 = \text{aabc}$

$\text{head}_5 = \text{a}$

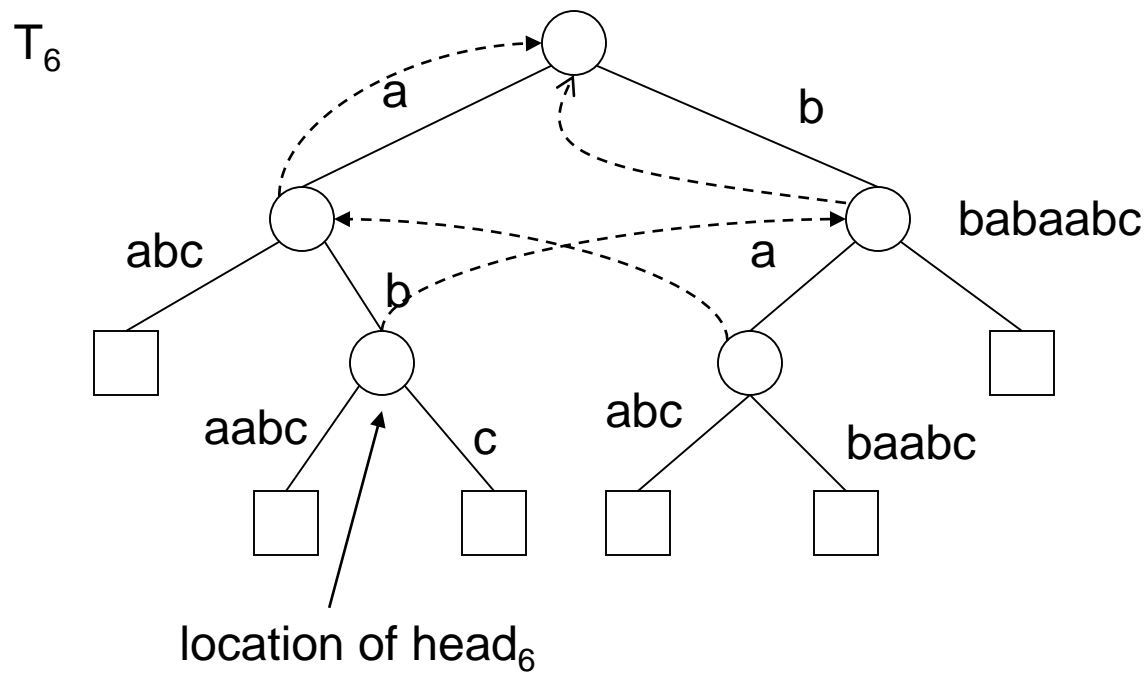


# Suffix tree: example



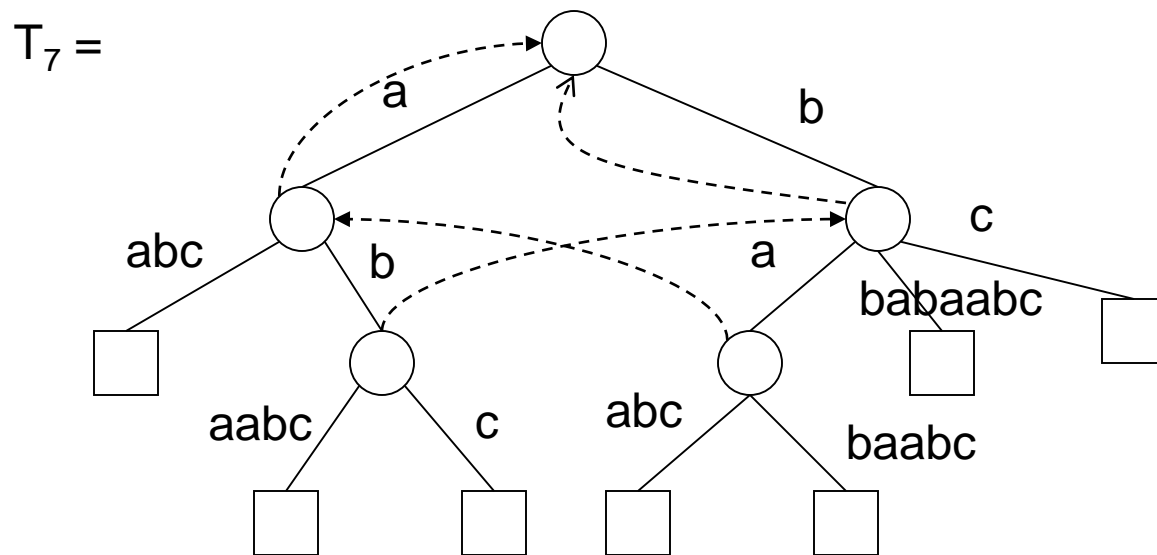
$\text{suf}_6 = \text{abc}$   
 $\text{head}_6 = \text{ab}$

# Suffix tree: example



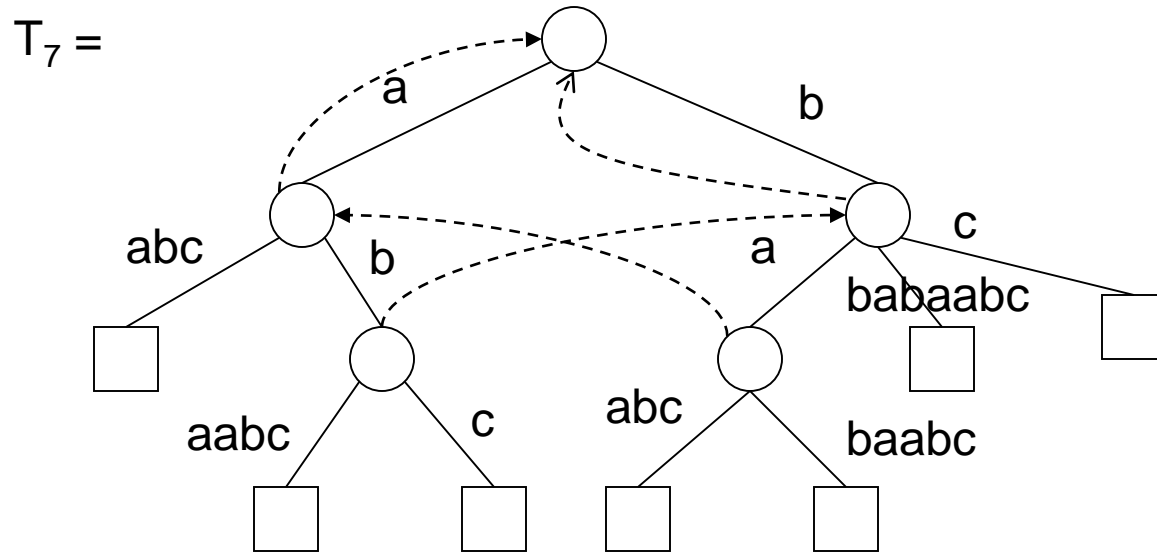
suf<sub>7</sub> = bc  
 head<sub>7</sub> = b

# Suffix tree: example



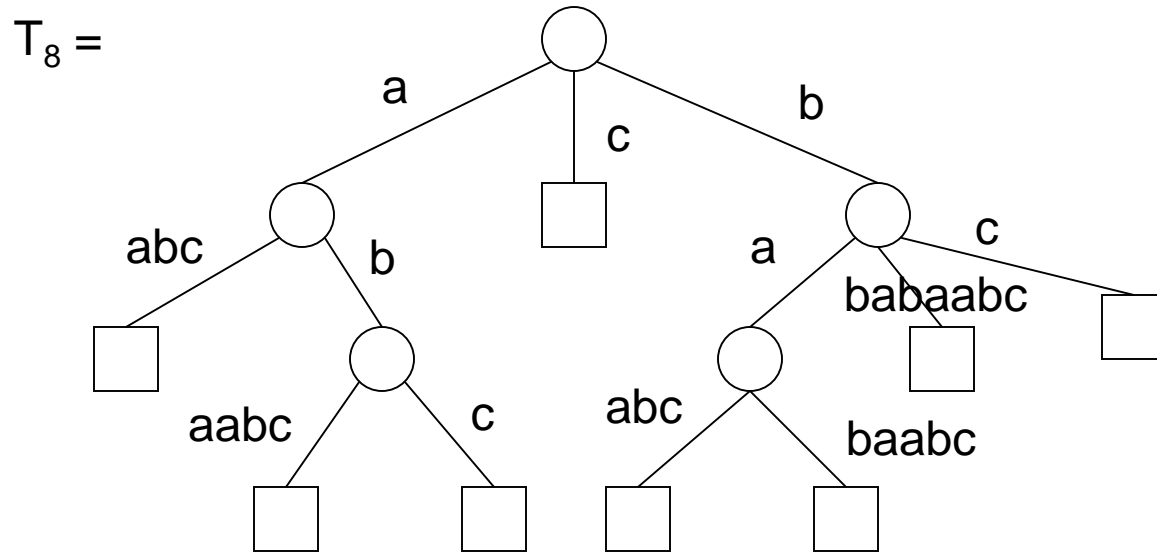
$\text{suf}_8 = c$

# Suffix tree: example



$\text{suf}_8 = c$

# Suffix tree: example



# The algorithm MCC

Iteration  $i + 1$ : Given  $T_i$ , construct  $T_{i+1}$ :

**Invariant:** In  $T_i$  all internal nodes have a suffix link, except for the internal node possibly inserted into  $T_i$  in iteration  $i$ .

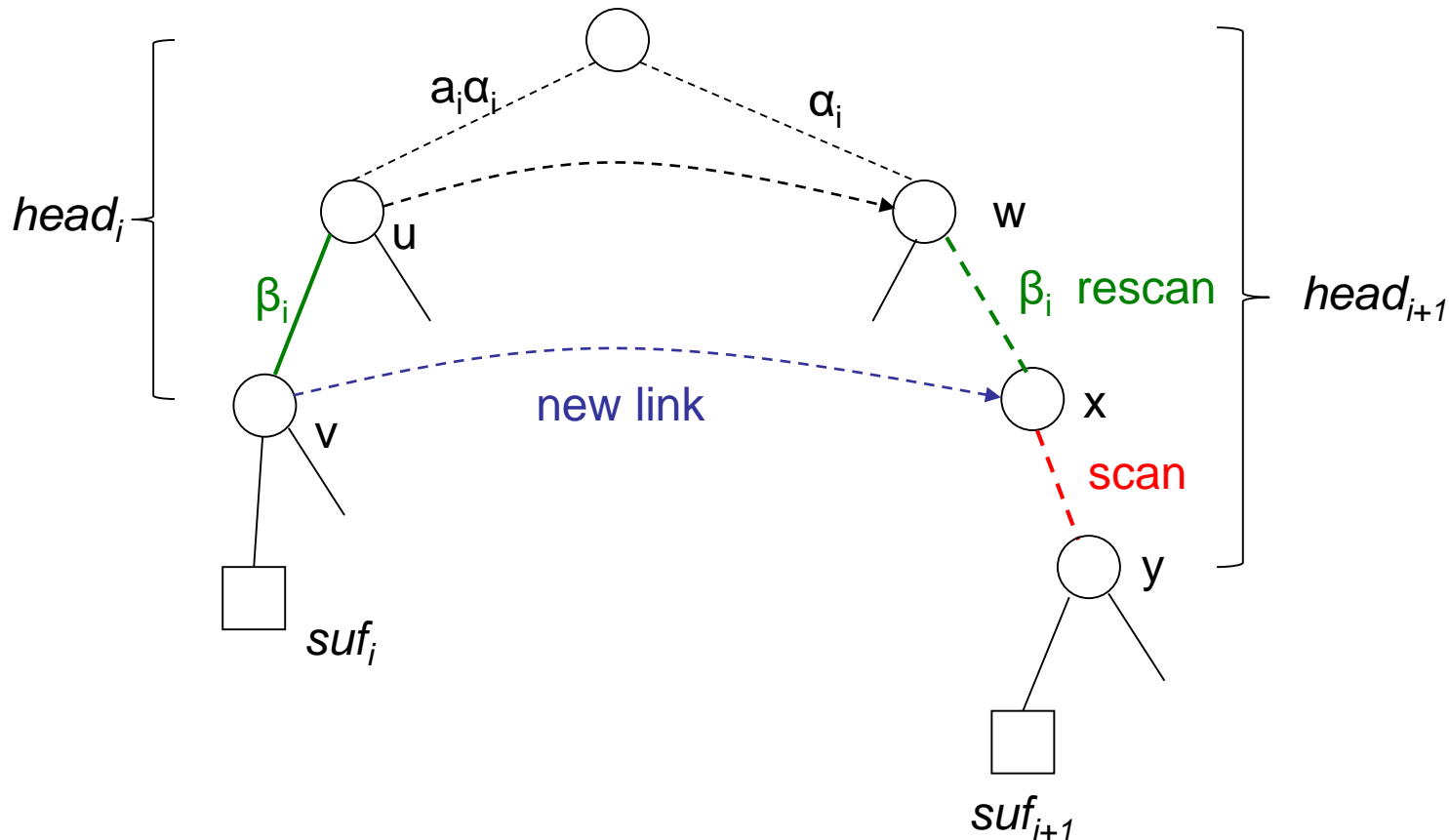
**Lemma:** If  $ay$  has a location in  $T_i$ , so does  $y$  in  $T_{i+1}$ .

**Proof:** Note that a string  $\alpha$  has a location in  $T_i$  if and only if there exist two suffixes  $suf_j$  and  $suf_k$ , where  $1 \leq j \neq k \leq i$ , such that  $\alpha$  is the longest common prefix of  $suf_j$  and  $suf_k$ .

Thus if  $ay$  is the longest common prefix of  $suf_j$  and  $suf_k$ , with  $1 \leq j \neq k \leq i$ , then  $y$  is the longest common prefix of  $suf_{j+1}$  and  $suf_{k+1}$ , where  $1 \leq j+1 \leq i+1$  and  $1 \leq k+1 \leq i+1$ .

Hence  $y$  has a location in  $T_{i+1}$ .

# Iteration $i + 1$



MCC traverses the suffix link of the nearest ancestor of  $suf_i$  having such a link. Then it identifies  $head_{i+1}$ , using rescan and scan operations, and sets a new suffix link if required.

# Analysis

Iteration  $i+1$ :

$\gamma_i$  = longest prefix of  $\text{suf}_i$  having a location with suffix link in  $T_i$

**Cost rescan:** It is not necessary to scan all the characters of  $\beta_i$ . Since  $\beta_i$  is in the tree, starting at node  $w$ , it suffices to traverse the respective edges by inspecting the indices of the edge labels. Thus the cost is proportional to number of edges traversed. Whenever an edge is fully traversed, the edge label adds to  $\gamma_{i+1}$ . The rescan starts at a string of length  $|\gamma_i|-1$ . Therefore the cost is a constant factor times  $|\gamma_{i+1}| - (|\gamma_i|-1) + 1$ .

**Cost scan:** Proportional to number of character comparisons. The scan starts at a string of length  $|\text{head}_i|-1$ . Thus the cost is a constant factor times  $|\text{head}_{i+1}| - (|\text{head}_i|-1) + 1$ .



## Summation over all iterations

$$\sum_{0 \leq i \leq n-1} (|Y_{i+1}| - (|Y_i| - 1) + 1) = |Y_n| - |Y_0| + 2n \leq 3n$$

$$\sum_{0 \leq i \leq n-1} (|\mathit{head}_{i+1}| - (|\mathit{head}_i| - 1) + 1) = |\mathit{head}_n| - |\mathit{head}_0| + 2n \leq 3n$$

# Suffix tree: application

---

Usage of a suffix tree  $T$ :

- 1 **Search for a string  $\alpha$ :**  
Follow the path with edge labels  $\alpha$  (takes  $O(|\alpha|)$  time).  
leaves of the subtree  $\hat{=}$  occurrences of  $\alpha$
- 2 **Search for the longest substring occurring at least twice:**  
Find the location of a substring with maximum weighted depth that is an internal node.
- 3 **Prefix search:**  
All occurrences of strings with prefix  $\alpha$  are represented by the nodes of the subtree rooted at the extended location of  $\alpha$  in  $T$ .

# Suffix tree: application

## 4 Range search for $[\alpha, \beta]$ :

