ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Algorithm Theory

12 Suffix Trees

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik

Wintersemester 2007/08

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Text Search

- **Scenarios**
- **Static texts**
  - Literature databases
  - Library systems
  - Gene databases
  - World Wide Web
- **Dynamic texts**
  - Text editors
  - Symbol manipulators

Properties of Suffix Trees

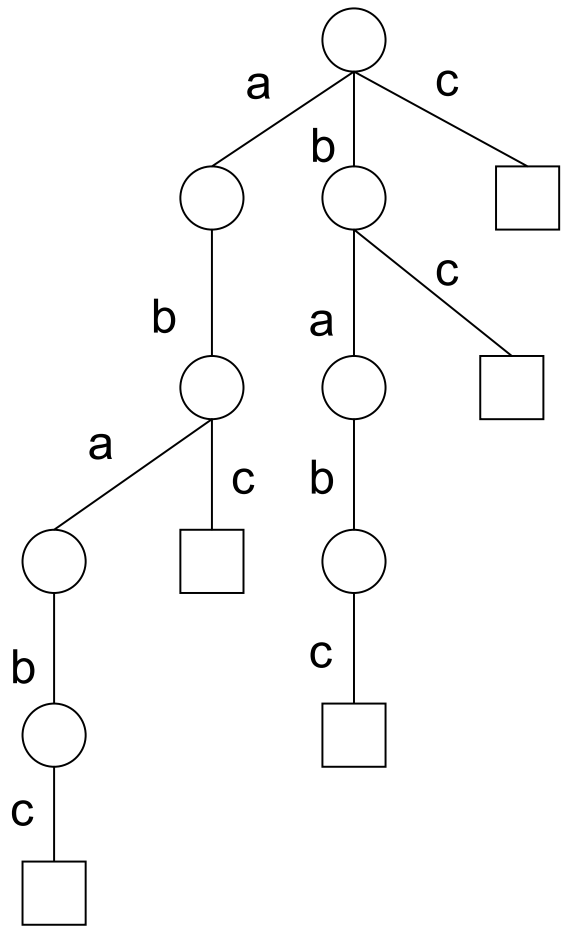
- **Search index**
  - for a text σ in order to search for several patterns α
- **Properties**
  - **Substring searching** in time O(|α|)
  - **Queries to σ itself**, e.g.:
    - Longest substring of σ occuring at least twice
  - **Prefix search**:
    - all positions in σ with prefix α

Properties of Suffix Trees

- **Range search**:
  - all locations (substrings) in σ belonging to an interval [α, β] with $\alpha \leq_{lex} \beta$, e.g.
    - abrakadabra, acacia ∈ [abc, acc],
    - abacus ∉ [abc, acc]
- **Linear complexity:**
  - Space requirement and construction time in O(|σ|)

Trie

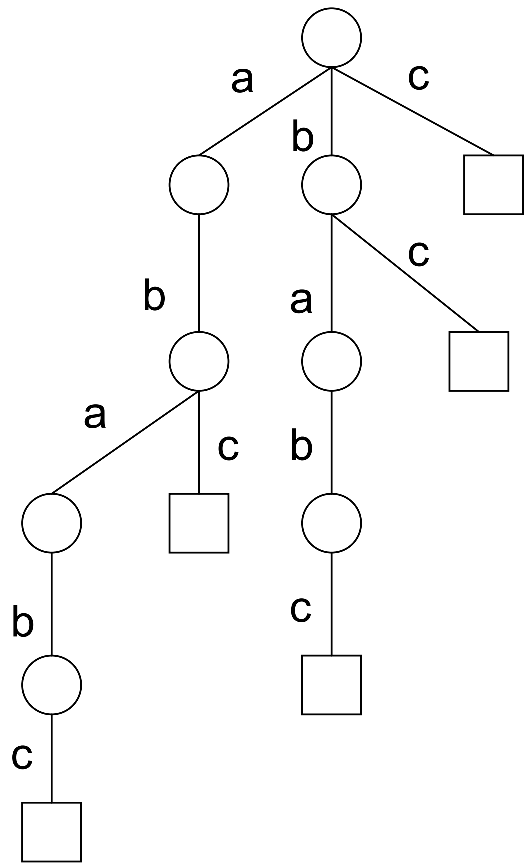
- **Trie:**
  - A tree representing a set of keys.
  - for alphabet Σ, set S of keys, S⊂Σ*
  - Key: string in Σ*
- **Edge of a trie T**
  - labeled with a single character of Σ
- **Neighboring edges**
  - edges that lead to different children of a node
  - labeled with different characters
- **A leaf 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:** σ = ababc

suffixes: ababc = suf_1
babc = suf_2
abc = suf_3
bc = suf_4
c = suf_5

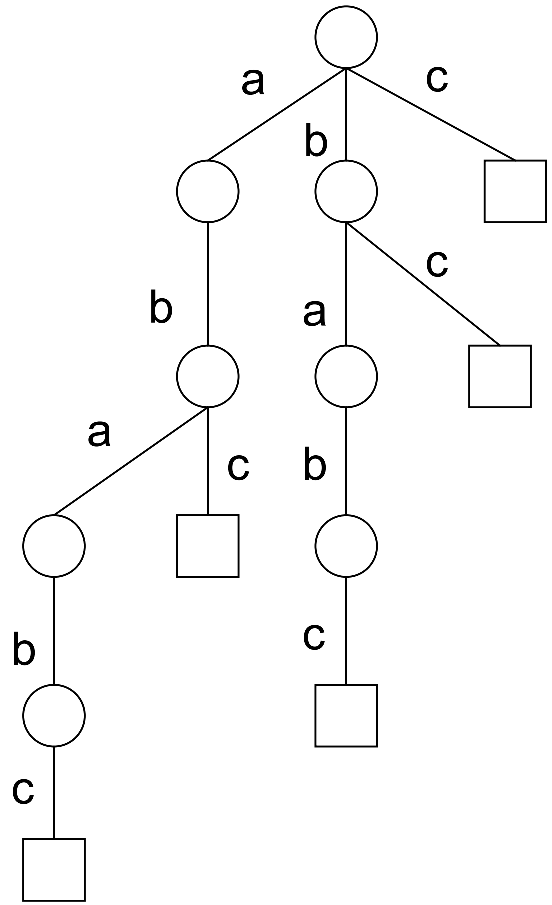


Suffix Trie

- **Internal nodes of a suffix trie correspond to substrings of σ**
- **Each proper substring of σ is represented by an internal node.**
- **Let $\sigma = a^n b^n$. Then, there are $(n+1)^2$ different substrings (or internal nodes).**
  ⇒ space requirement $O(n^2)$

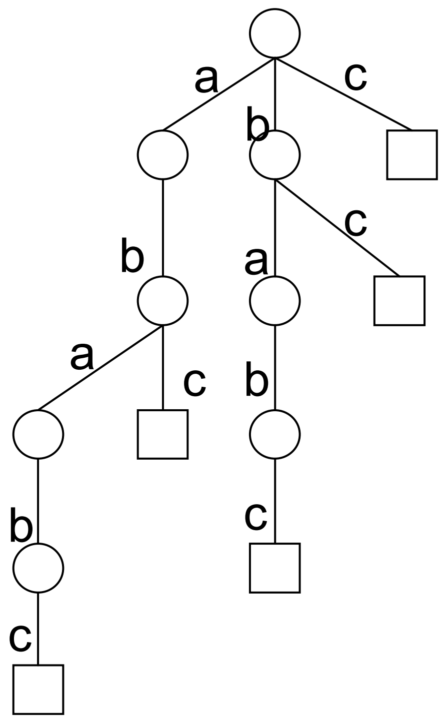
Suffix Tries

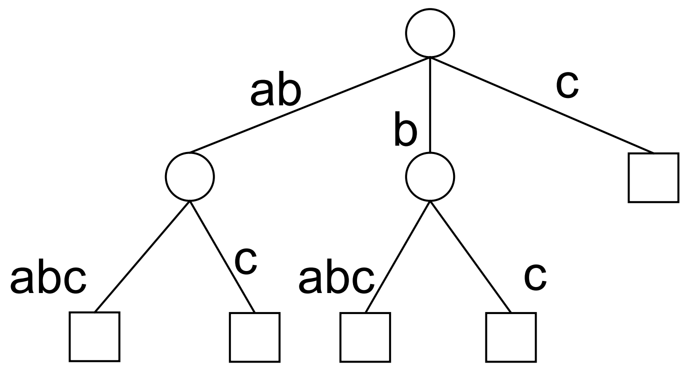
- **A suffix trie T satisfies some of the desired properties:**
  - **String matching** for α:
    - Following the path with edge labels α takes O(|α|) time.
    - Leaves of the subtree = occurrences of α
  - **Longest substring occurring at least twice:**
    - internal node with maximum depth having at least two children
  - **Prefix search**
    - All occurrences of strings with prefix α are represented by the nodes of the subtree rooted at the internal node corresponding to α.



Suffix Trees

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

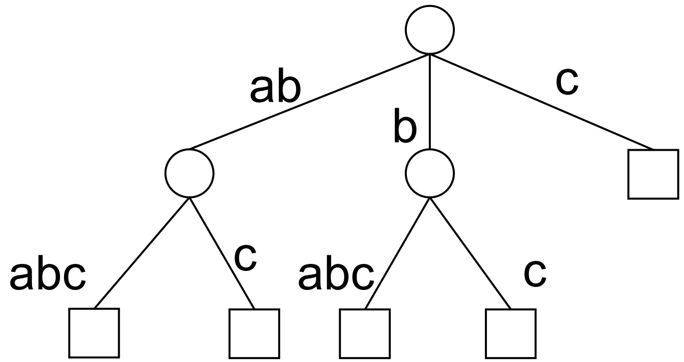


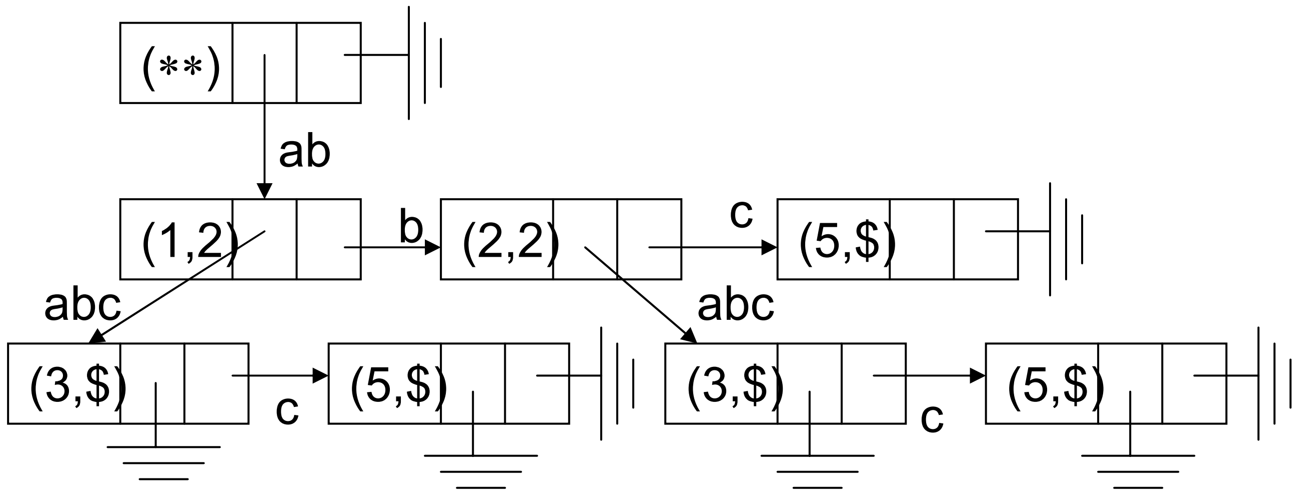


suffix tree = contracted suffix trie

Internal Representation of Suffix Trees

- **Child-sibling representation**
  - substring: pair of numbers(i,j)
- **Example: σ= ababc**
  - node v = (v.l.,v.u,v.c,.v.s)
- **Further pointers (suffix links are added later)**





Properties of Suffix Trees

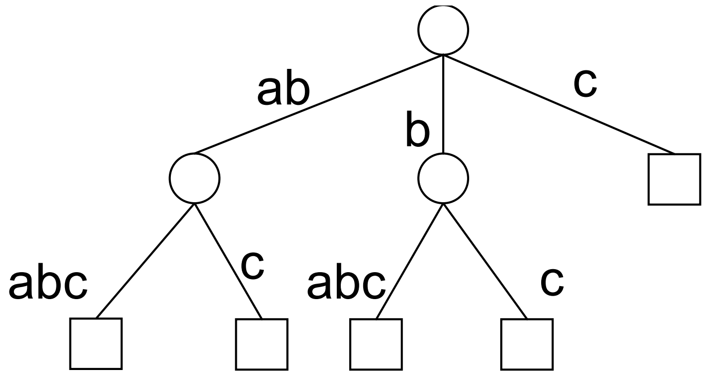
- **(S1) No suffix is prefix of another suffix.**
  - This holds if the last character of σ is $\$ \notin \Sigma$.
- **Search:**
  - (T1) edge = non-empty substring of σ.
  - (T2) neighboring edges:
    corresponding substrings start with different characters

Properties of Suffix Trees

- **Size**
  - (T3) each internal node (≠root) has at least two children
  - (T4) leaf = (non-empty) suffix of σ.
- **Let n= |σ| ≠1.**
  - (by T4) then the number of leaves is n
  - (by T3) number of intervals ≤ n-1
  - implies space requirement O(n)

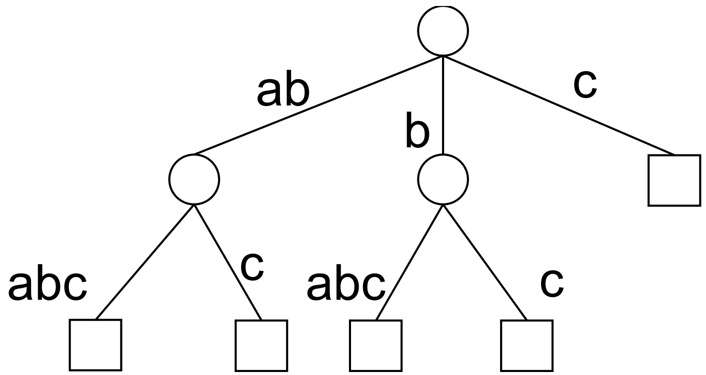
Construction of Suffix Trees

- **Definitions:**
  - **Partial path**: Path from the root to a node in T.
  - **Path**: A partial path ending at a leaf.
  - **Location** of a string α: Node where the partial path corresponding to α ends (if it exists).



Construction of Suffix Trees

- **Extension of a string α:**
  - string with prefix α
- **Extended location of a string α:**
  - location of the shortest extension of α whose location is defined
- **Contracted location of a string α:**
  - location of the longest prefix of α whose location is defined



Construction of Suffix Trees

- **Definitions**
  - suf_i : suffix of σ beginning at position i,
    - e.g. $suf_1 = \sigma$, $suf_n = \$$.
  - $head_i$:
    - longest prefix of suf_i which is also a prefix of suf_j for some $j < i$.
- **Example:**

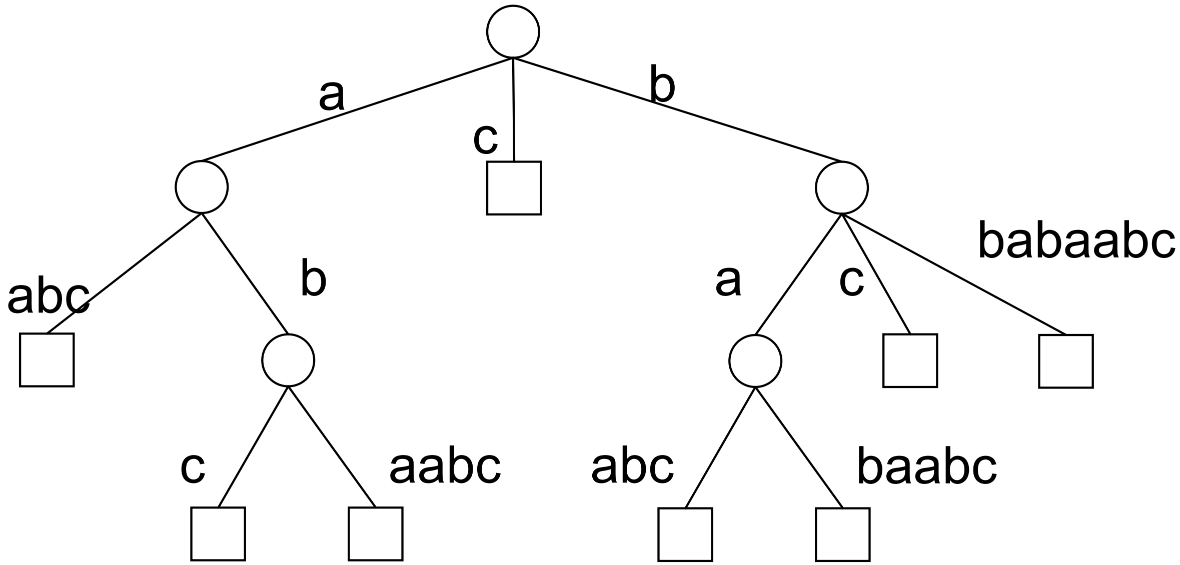
σ = bbabaabc α= baa (has no location)

suf_4 = baabc

$head_4$ = ba

Construction of Suffix Trees

σ = 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 suf_{i+1}**
- **Algorithm suffix-tree**
  - Input: string σ
  - Output: suffix tree T for σ

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

2 for i := 0 **to** n – 1 **do**

3 insert suf_{i+1} into T_i, store the result in T_{i+1};

4 end for

Naive Suffix Tree Construction

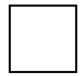
- **All suffixes suf_j with $j \leq i$ have a location in T_i**
  - $head_{i+1}$= longest prefix of suf_{i+1} whose extended location exists in T_i
- **Definition:**
  - $tail_{i+1}$:= suf_{i+1}–$head_{i+1}$ i.e. suf_{i+1}= $head_{i+1}$ $tail_{i+1}$
  - ⇒ (by S1) $tail_{i+1} \neq \varepsilon$.

Naive Suffix Tree Construction

- **Example:** σ= ababc

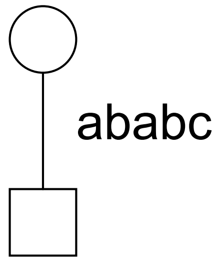
suf_3 = abc
$head_3$ = ab
$tail_3$ = c

T_0 =



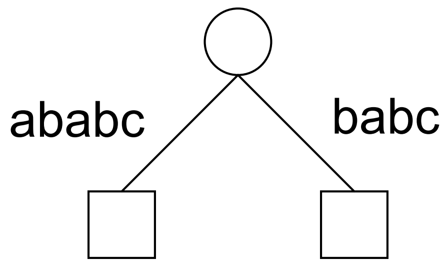
suf_1 = ababc

T_1 =



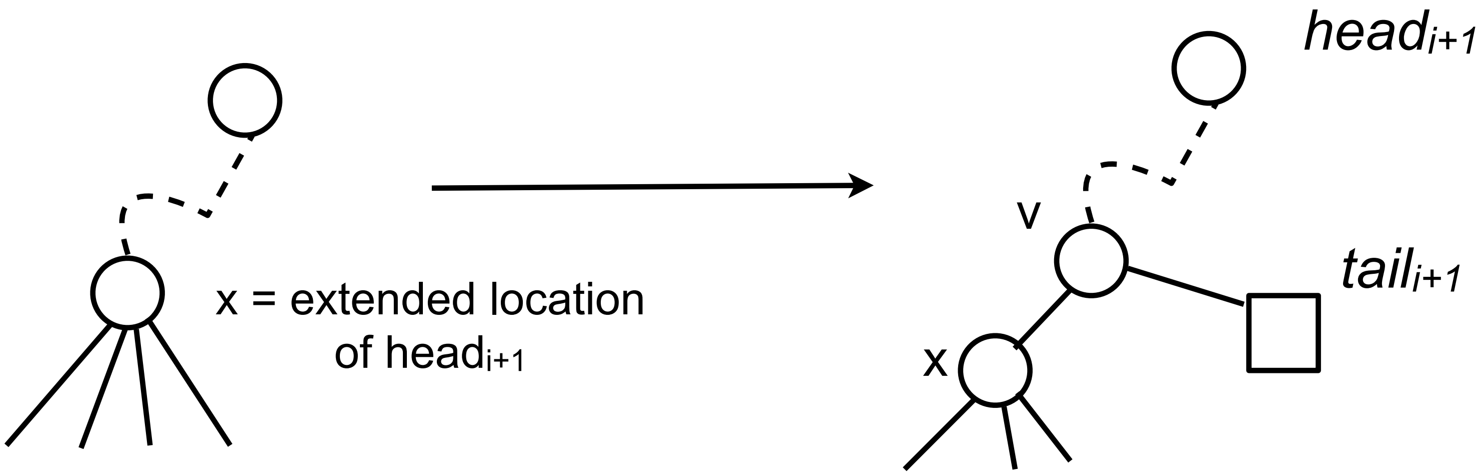
suf_2 = babc

T_2 =



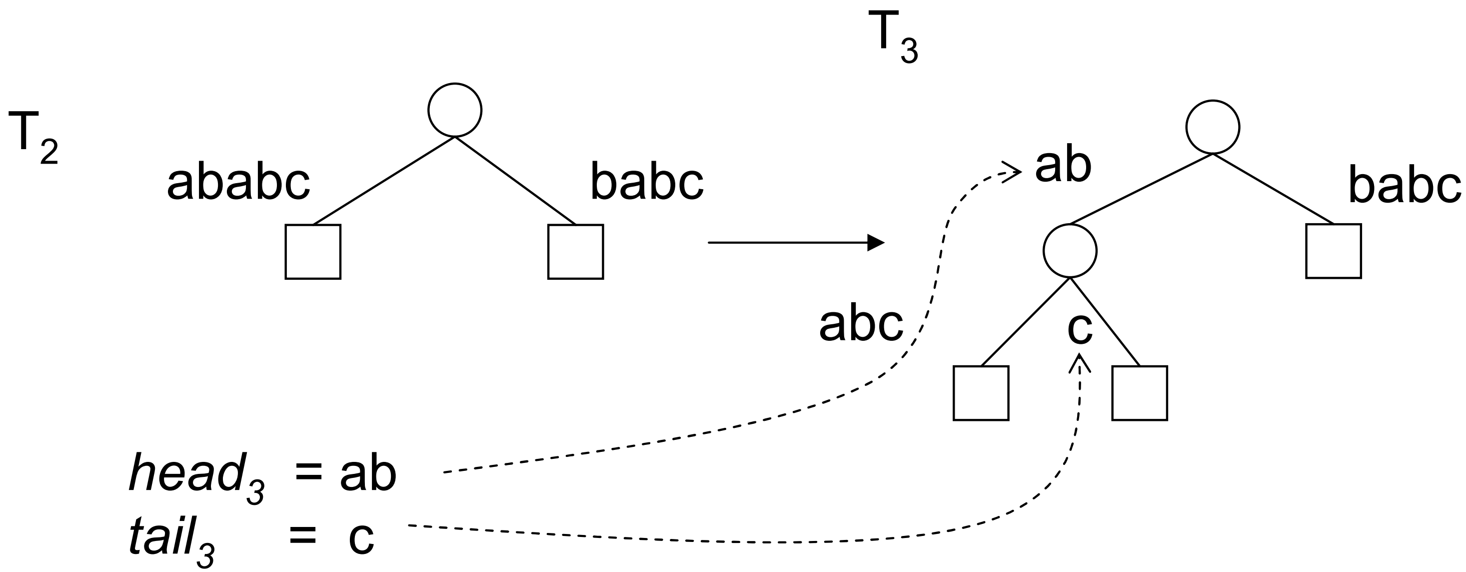
Naive Suffix Tree Construction

- **Ti+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: σ = ababc



Naive Suffix Tree Construction

Algorithm *suffix-insertion*

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

Output: tree T_{i+1}

v := root of T_i
j := i
repeat
    find child w of v with $\sigma_{w.l} = \sigma_{j+1}$
    k:= w.l –1;
    while k < w.u and $\sigma_{k+1} = \sigma_{j+1}$ **do**
        k:= k+1; j := j + 1
    end while
    if k= w.u **then** v:= w
until k <w.u or w= nil
/* v is the contracted location of $head_{i+1}$*/
insert the location of $head_{i+1}$ and $tail_{i+1}$ below v into T_i

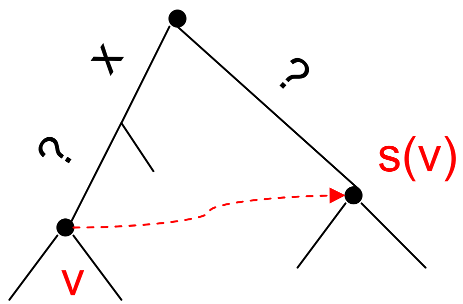
Running time of suffix-insertion: O(n-i)
Total time required for the naive construction: $O(n^2)$

Mc Creight´s Algorithm

- **Idea:**
  - Extended location of $head_{i+1}$ in T_i is determined in constant amortized time.
- 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 M constructs a suffix tree σ with $|\sigma|$ leaves and at most $|\sigma|-1$ internal nodes in time $O(|\sigma|)$

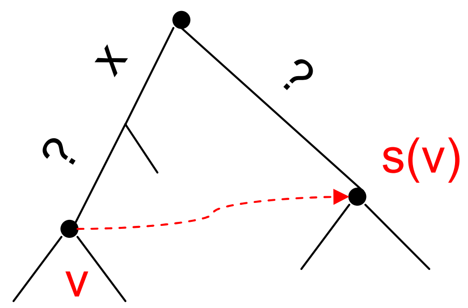
Suffix Links

- **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) which is called a suffix link.



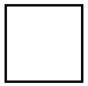
Suffix Links

- **Idea:**
  - 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.

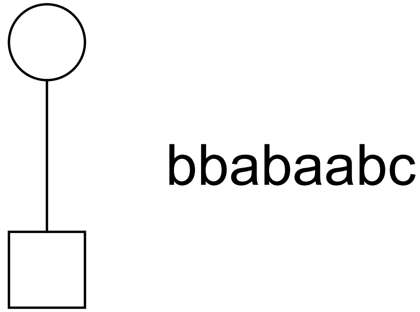


Suffix Tree: Example

T_0 =



T_1 =

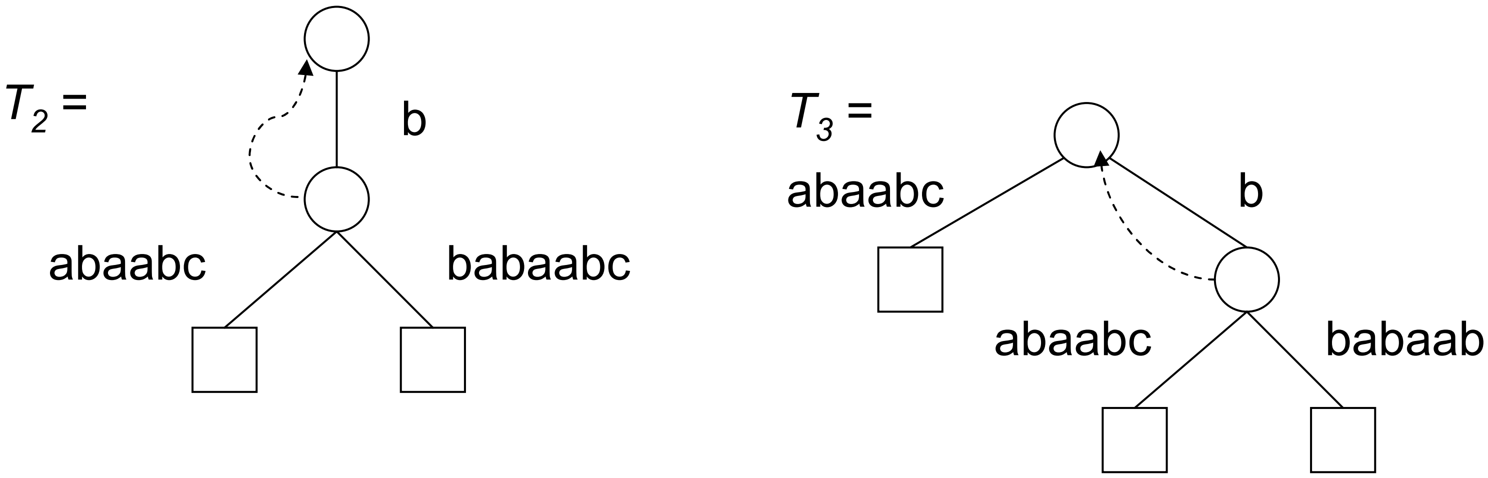


suf_1 = bbabaabc

suf_2 = babaabc
$head_2$ = b

Suffix Tree: Example

x=b
?=ε

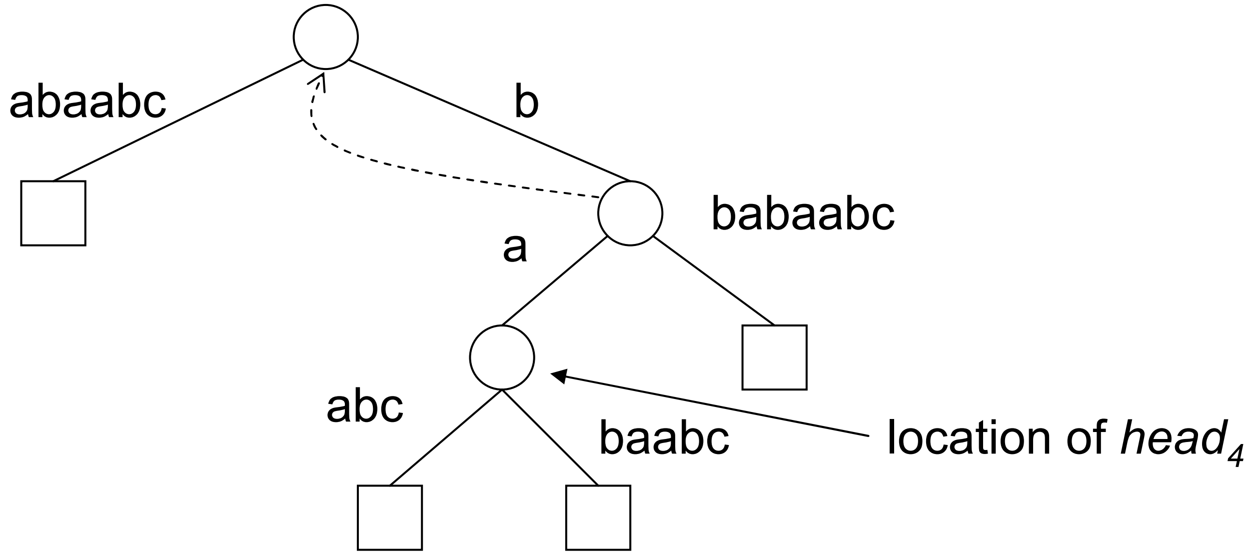


suf_3 = abaabc
$head_3 = \varepsilon$

suf_4 = baabc
$head_4$ = ba

Suffix Tree: Example

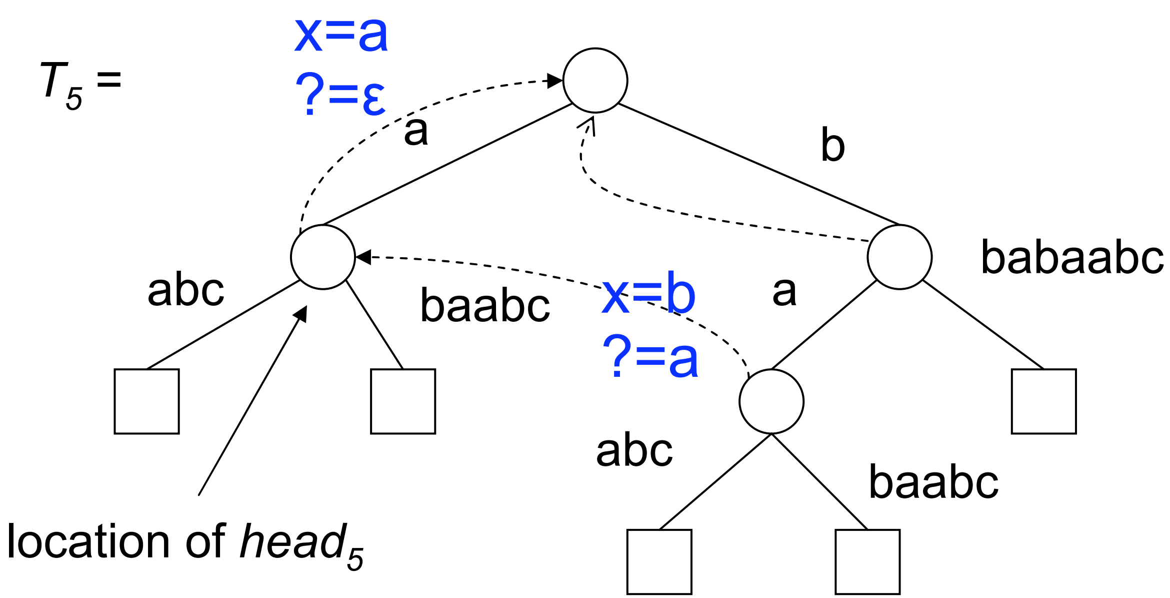
T_4 =



suf_5 = aabc
$head_5$ = a

Suffix Tree: Example

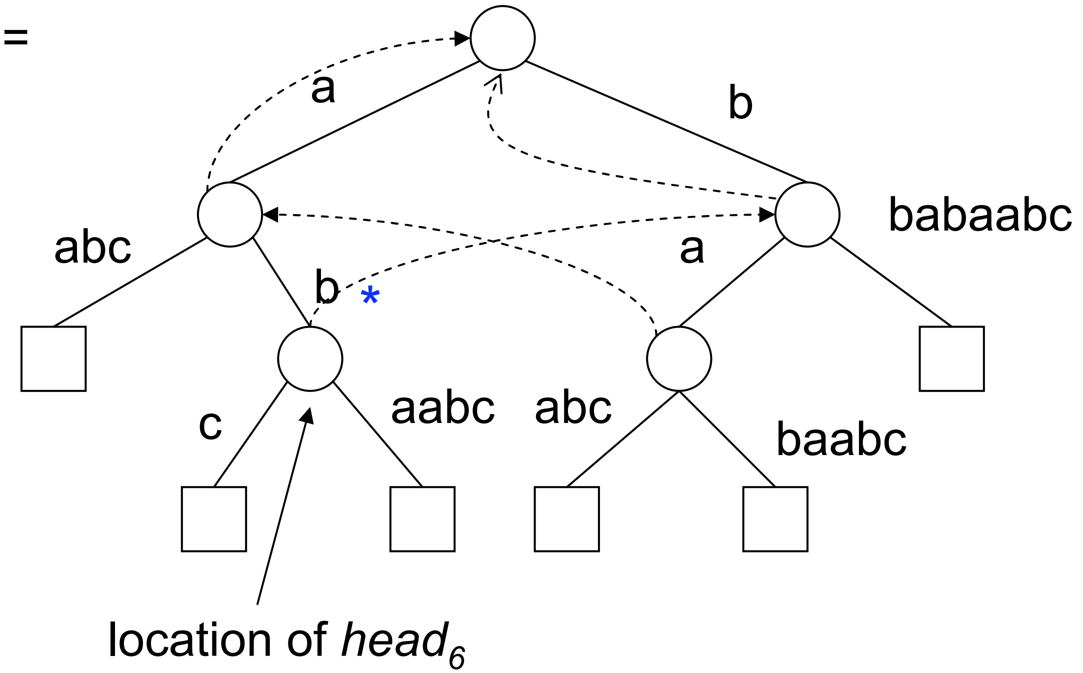
T_5 =



suf_6 = abc
$head_6$ = ab

Suffix Tree: Example

T_6 =

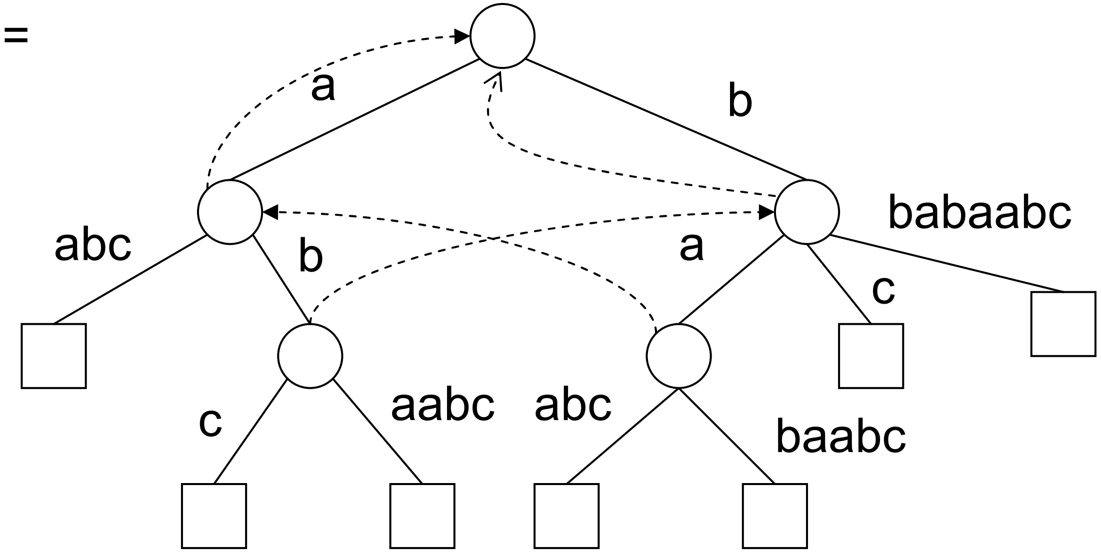


* x=a
?=b

suf_7 = bc
$head_7$ = b

Suffix Tree: Example

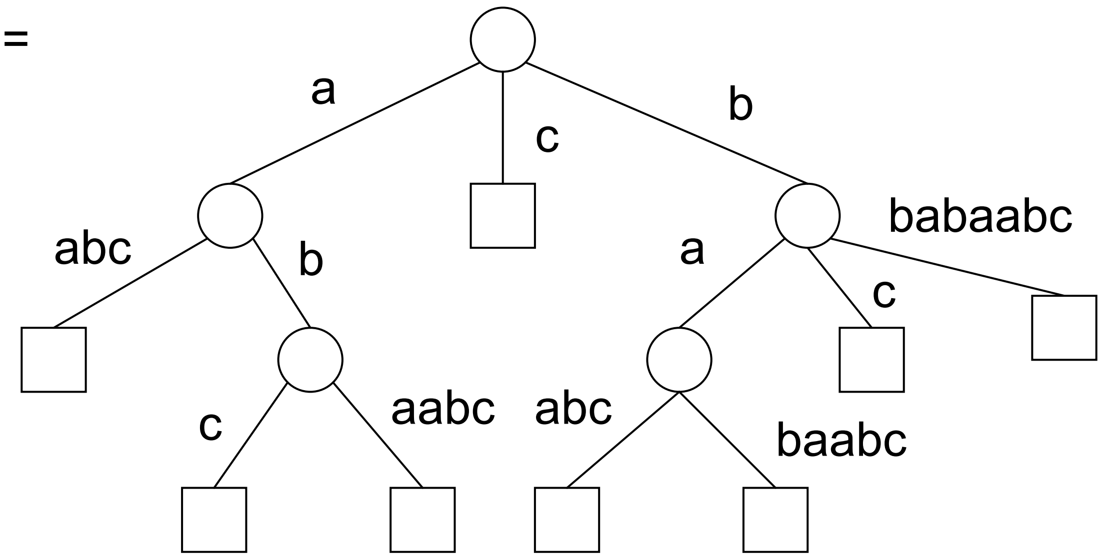
T_7 =



suf_8 = c

Suffix Tree: Example

T_8 =



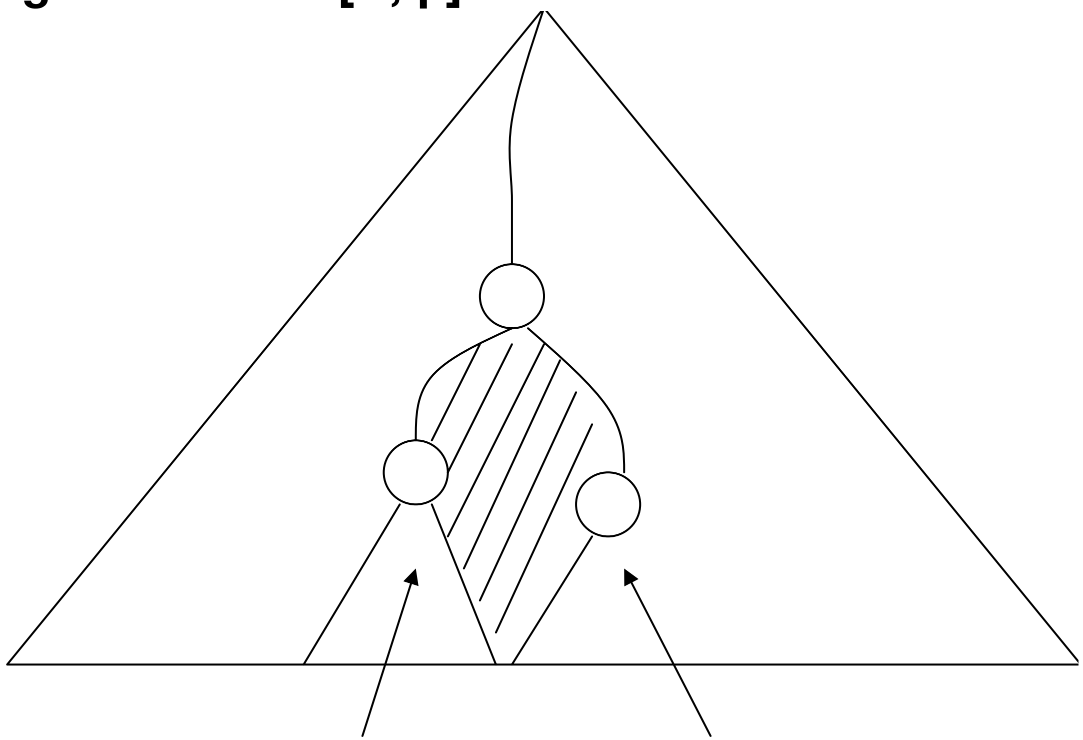
Suffix Tree: Application

- **Usage of a suffix tree T:**

1. **Search for a string α:**
   Follow the path with edge labels α (takes O(|α|) time).
   leaves of the subtree = occurrences of α
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 α are represented by the nodes of the subtree rooted the location of α in T.

Suffix Tree: Application

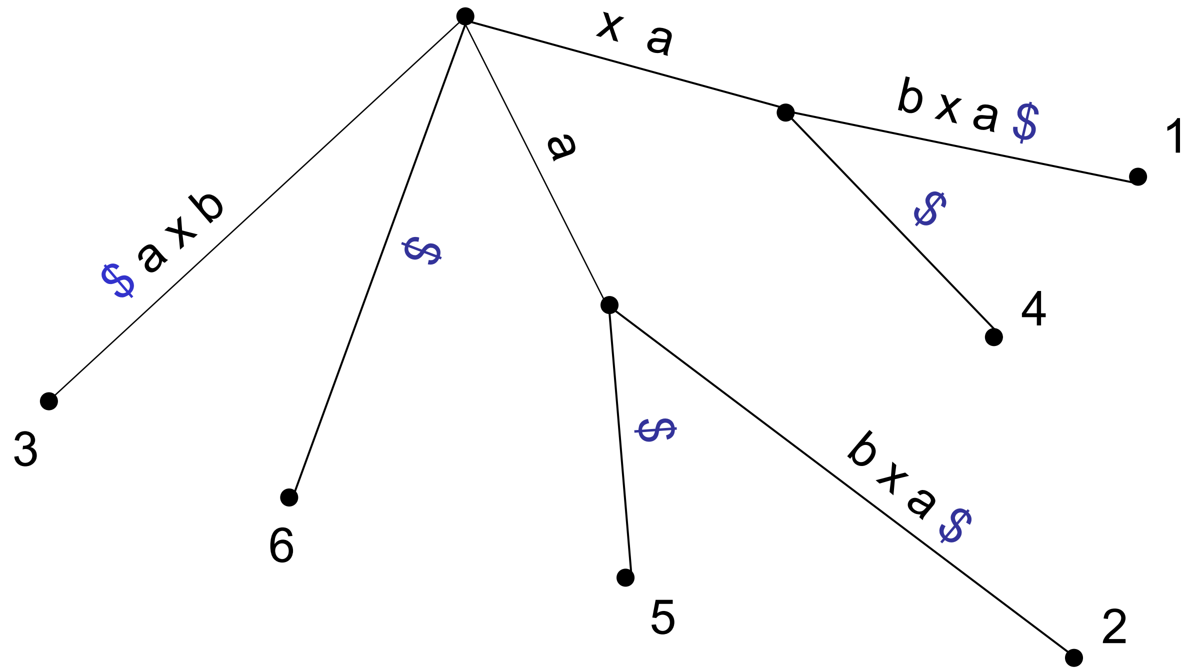
4. **Range search for [α, β]**



range boundaries

Suffix Tree

t = x a b x a $
1 2 3 4 5 6

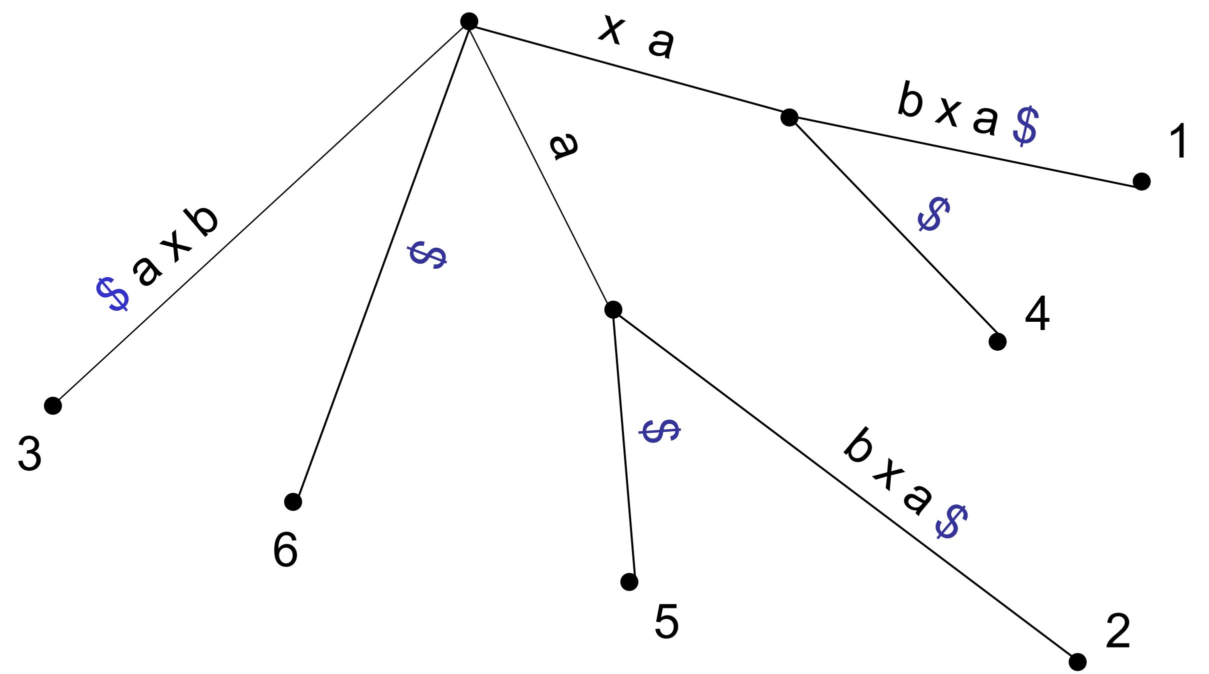


Ukkonen´s Algorithm: Implicit Suffix Trees

- **Definition:**
  - An implicit suffix tree is a tree obtained from the suffix tree for t$ by
    (1) deleting every copy of $ from the edge labels,
    (2) deleting edges that have no label,
    (3) deleting unary nodes.

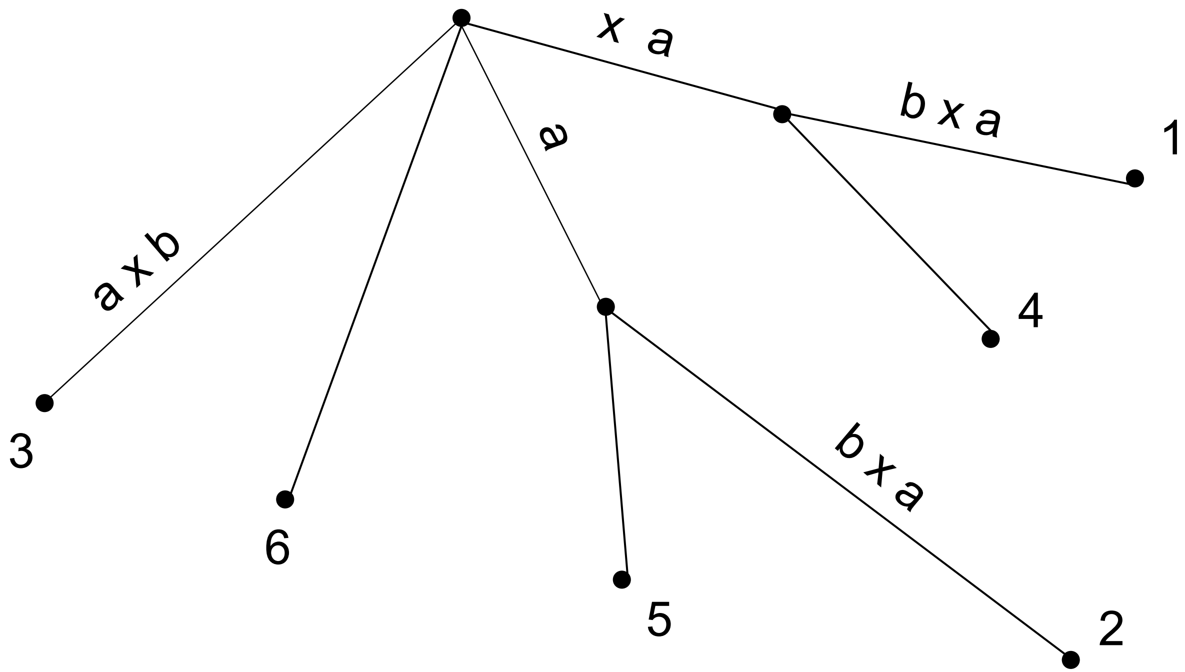
Ukkonen´s Algorithm: Implicit Suffix Trees

t = x a b x a $
1 2 3 4 5 6



Ukkonen´s Algorithm: Implicit Suffix Trees

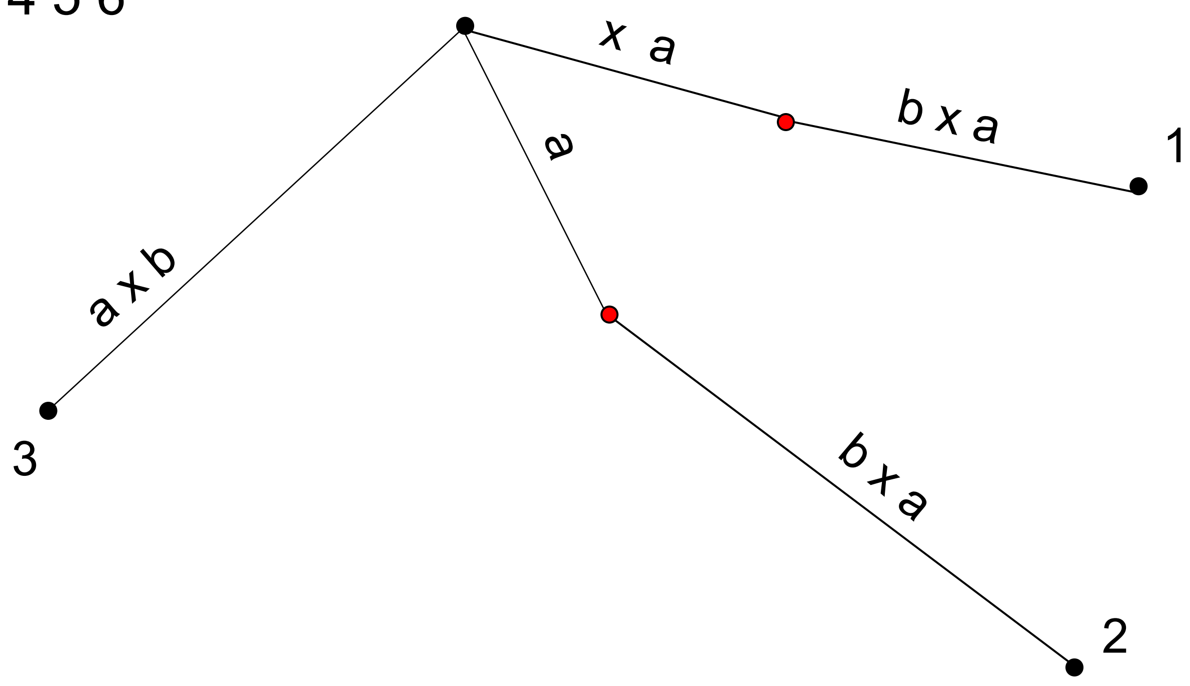
(1) deleting $ from the edge labels



Ukkonen´s Algorithm: Implicit Suffix Trees

(2) deleting edges that have no label

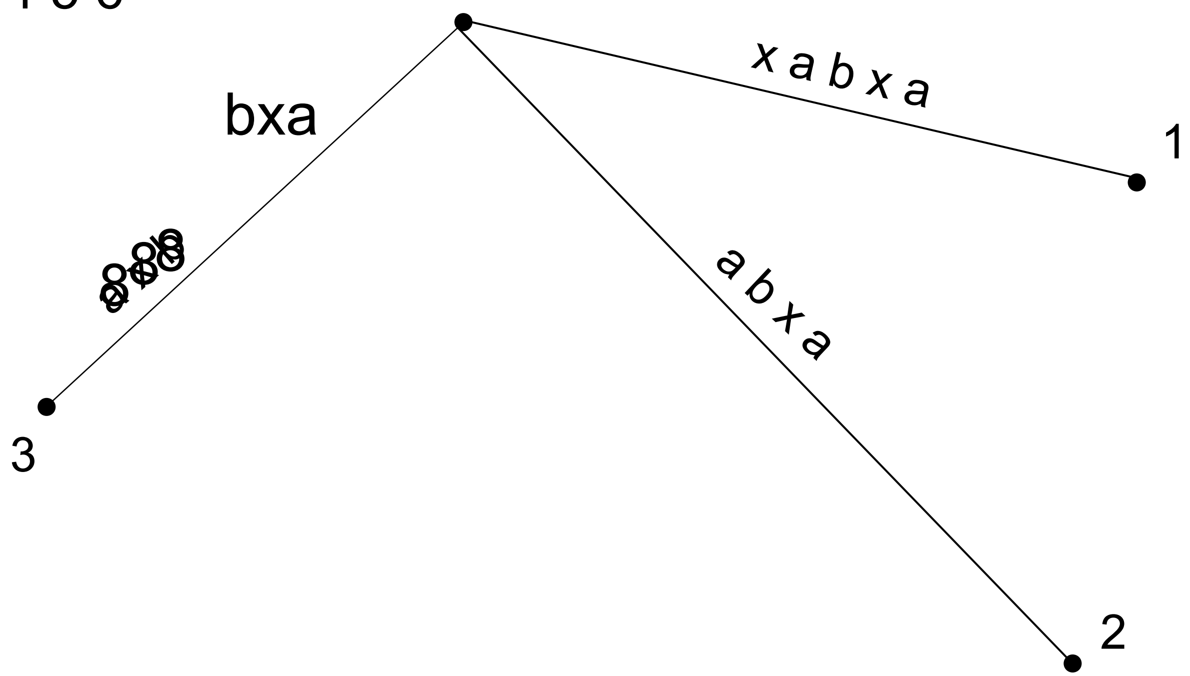
t = x a b x a $
1 2 3 4 5 6



Ukkonen´s Algorithm: Implicit Suffix Trees

(3) deleting unary nodes

t = x a b x a $
1 2 3 4 5 6



Ukkonen´s Algorithm

- Let $t = t_1 t_2 t_3 \ldots t_m$.
- Ukk is an **online** algorithm: The suffix tree ST(t) is constructed step by step by constructing a sequence of implicit suffix trees for the prefixes of t:
  - $ST(\varepsilon)$, $ST(t_1)$,$ST(t_1 t_2)$, ...,$ST(t_1 t_2 \ldots t_m)$
- $ST(\varepsilon)$ is the empty implicit suffix tree, consisting of the root only.
- $ST(t_1 t_2 \ldots t_i)$ is the implicit suffix tree containing all suffixes of $t_1 t_2 \ldots t_i$

Ukkonen´s Algorithm

- This is an online approach in the sense that in each step, the implicit suffix tree for a prefix of t is created without knowledge of the rest of the input string t.
- Since the algorithm reads the input string character by character from left to right, it works incrementally.

Ukkonen´s Algorithm

- **Incremental construction of an implicit suffix tree:**
- **Induction basis:** ST(ε) consists of the root only.
- **Induction step:** $ST(t_1 .... t_i)$ is extended to $ST(t_1 ... t_i t_{i+1})$ for all $i < m$.
- Let T_i be the implicit suffix tree for t[1...i].
  - At first, we construct T_1: This tree has a single edge labeled with character t_1.
  - In phase i+1, we construct tree T_{i+1} from T_i.
  - We iterate for i = 1 …m–1.

Ukkonen´s Algorithm

Pseudo code for Ukk:

Construct tree T_1

for i = 1 **to** m–1 **do**

begin{phase i+1}

    for j = 1 **to** i +1 **do**

    begin{extension j}

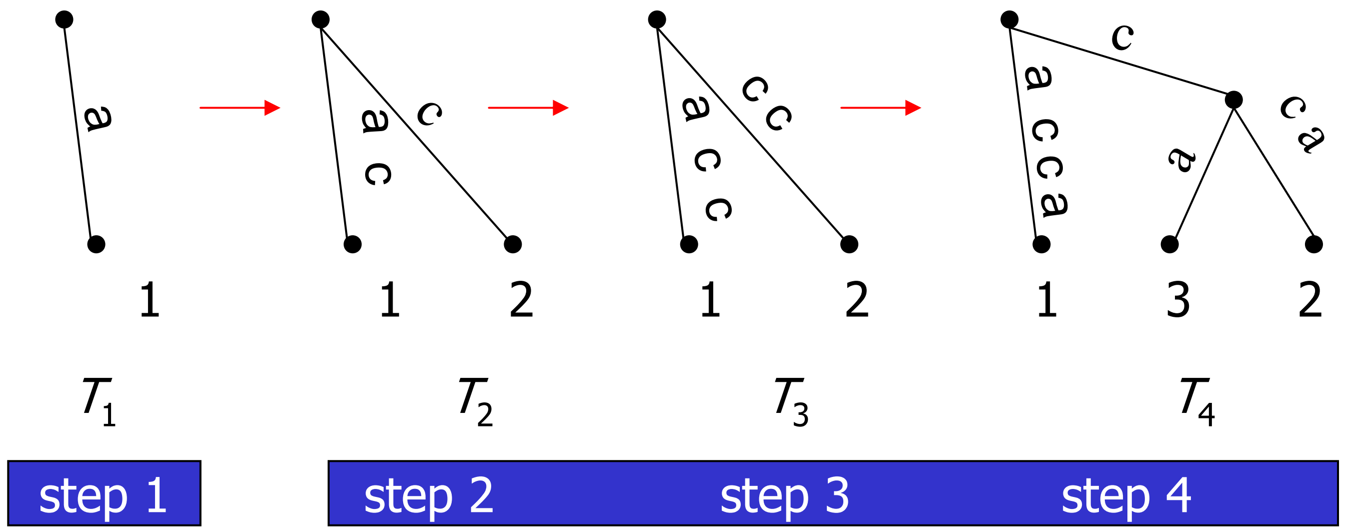
        In the current tree find the end of the path from the root labeled t[j ... i]. If necessary, extend that path by adding character t[i+1], thus ensuring that string t[j...i+1] is in the tree.

    end;

end;

Ukkonen´s Algorithm

t = a c c a $



Ukkonen´s Algorithm

- In extension j of phase i+1, the end of the path from the root labeled with substring t[j...i] is determined. Then, this substring is extended by adding the character t[i+1] to its end (unless t[i+1] already appears there).
- In phase i+1, string t[1...i+1] is first inserted into the tree, followed by strings t[2...i+1] , t[3...i+1] ,.... (in extensions 1,2,3,...., respectively).
- Extension i+1 of phase i+1 inserts the single character string t[i+1] into the tree (unless it is already there).

Ukk: Suffix Extension Rules

- Extension j (in phase i+1) results from applying one of the following rules:
  - **Rule 1:**
    - If the path t[j...i] ends at a **leaf**, character t[i+1] is added to the end of the label on that leaf edge.
  - **Rule 2:**
    - If no path from the end of string t[j...i] starts with character t[i+1], then a new leaf edge labeled with character t[i+1] is created. A new internal node will also be created there if t[j...i] ends inside an edge.
      - This is the only extension that increases the number of leaves! The new leaf represents the suffix starting at position j.
  - **Rule 3:**
    - If some path from the end of string t[j ...i] starts with character t[i+1], then string t[j…i+1] is already in the current tree, so we do nothing.

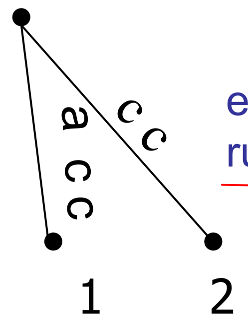
Ukkonen´s Algorithm

t = a c c a $

t [1...3] = acc

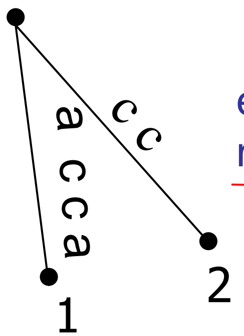
t [1...4] = acca

T_3



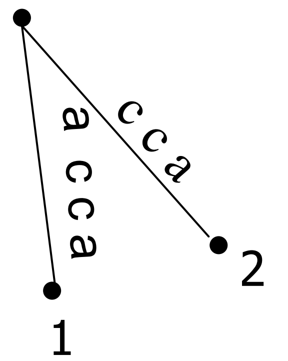
t [1..4] = acca

extend suffix 1
rule 1



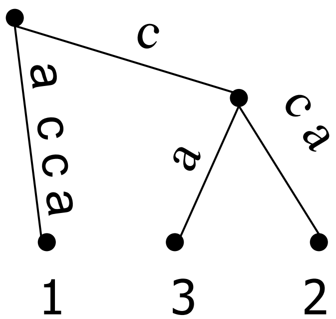
t [2..4] = cca

extend suffix 2
rule 1



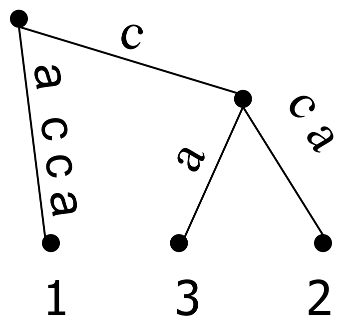
t [3..4] = ca

extend suffix 3
rule 2



t [4..4] = a

a is already in
the tree
rule 3



T_4

Ukkonen´s Algorithm

- During phase i+1 (when T_{i+1} is constructed from T_i) the following holds:

(1) If rule 3 applies in extension j, then the path labeled $t[j...i]$ in T_i must continue with character $t[i+1]$. So, any path labeled $t[j'... i]$ for $j' \geq j$ also continues with character $t[i+1]$.

- Therefore, rule 3 again applies in extensions $j' = j+1,...,i+1$.
- Once rule 3 applies in an extension of phase i+1, this phase may be ended.
  - Why? If in extension j rule 3 applies. Then,
    - t[j,...,i+1] is prefix of some t[k,..i] with k<j
    - for any m>1: t[j+m...i+1] is prefix of some t[k+m, ...,i]
    - t[k+m,..,i] is already in T_i

Ukkonen´s Algorithm

(2) If a leaf is created in T_i, then it will remain a leaf in all successive trees $T_{i'}$ for $j' \geq j$ (once a leaf, always a leaf!).

- Reason: A leaf edge is never extended beyond its current leaf.
- t = a c c a b a a c b a

Ukkonen´s Algorithm

- Implication:
  - Leaf 1 is created in phase 1. In each phase i+1 there is an initial sequence of successive extensions (starting with extension 1) where rule 1 or 2 applies
    - The first time rule 3 applies the phase is terminated
  - Let j_i denote the last extension in this sequence of phase i where rule 1 or rule 2 is applied
  - Then $j_i \leq j_{i+1}$
    - Phase i: extension j with $j \leq j_i$
      - If rule 1 is applied then in the next phase also rule 1 is applied in extension j of the phase i+1
        - then at the very same leaf t[i+1] is added
      - If rule 2 is applied then in the next phase rule 1 in extension j will be applied, since then t[i+1] can be added to the leaf

Ukkonen´s Algorithm

- **Extensions according to rule 1 may be performed implicitly!**

Ukkonen´s Algorithm

- **Improving the algorithm:**
- In phase i+1, rule 1 applies in all extensions j for $j \in [1, j_i]$.
  - Only constant time is required to do those extensions implicitly.
- If $j \in [j_i + 1, i+1]$, then find the end of the path labeled t[j...i] and extend it by character t[i+1] according to rules 2 or 3.
- If rule 3 applies, set $j_{i+1} = j - 1$ and end phase i+1.

Ukkonen´s Algorithm

- **Example:**

phase 1:	compute extensions	$1 \ldots j_1$
phase 2:	compute extensions	$j_1+1 \ldots j_2$
phase 3:	compute extensions	$j_2+1 \ldots j_3$
	...	
phase i-1:	compute extensions	$j_{i-2}+1 \ldots j_{i-1}$
phase i:	compute extensions	$j_{i-1}+1 \ldots j_i$

Ukkonen´s Algorithm

- As long as explicit extensions are performed, keep track of the index j* of the current explicit extension
- During the execution of the algorithm, j* never decreases.
- As there are only m phases (where $m = |t|$) and j* is bounded by m, the algorithm performs only m explicit extensions.

Ukkonen´s Algorithm

- Extended pseudo code for Ukk:

Construct tree T_1; j_1= 1;

for i = 1 **to** m–1 **do**

  begin{phase i+1}

  Do all implicit extensions.

  for j = j_i +1 **to** i +1 **do**

  begin{extension j}

    In the current tree find the end of the path from the root labeled t[j ... i]. If necessary, extend that path by adding character t[i+1], thus ensuring that string t[j...i+1] is in the tree.

    j_{i+1}:= j;

    if rule 3 was applied **then** j_{i+1}:= j –1 and phase i+1 ends;

  end;

end;

Ukkonen´s Algorithm

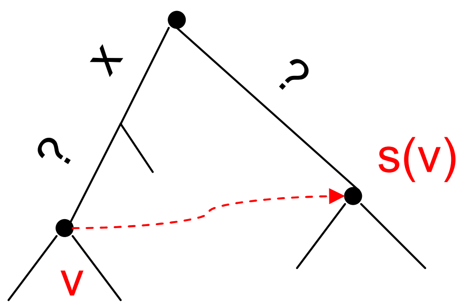
t = pucupcupu

i: 0	1	2	3	4	5	6	7	8	9
ε	*p	pu	puc	pucu	pucup	pucupc	pucupcu	pucupcup	pucupcupu
		*u	uc	ucu	ucup	ucupc	ucupcu	ucupcup	ucupcupu
			*c	cu	cup	cupc	cupcu	cupcup	cupcupu
				u	*up	upc	upcu	upcup	upcupu
					p	*pc	pcu	pcup	pcupu
						c	cu	cup	*cupu
							u	up	*upu
								p	pu
									u

- Suffixes that cause an extension according to rule 2 are marked with *.
- Underlined suffixes indicate the last extension where rule 2 applies.
- Suffixes that end a phase (the first time rule 3 applies) are colored blue.

Ukkonen´s Algorithm

- **Idea:**
  - By following the suffix links, we do not have to start each search for a split 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.



Ukkonen´s Algorithm

- Using suffix links, extensions rules 2 and 3 can be applied more efficiently.
- An explicit extension takes amortized O(1) time (not shown here).
- Since there are only *m* explicit extensions, the total running time of Ukkonen‘s algorithm is O(m) (where m=|t|)

Ukkonen´s Algorithm

- **The true suffix tree:**
- The final implicit suffix tree T_m can be converted to a true suffix tree in O(*m*) time.
  (1) Add a terminal symbol $ to the end of t.
  (2) Let Ukkonen‘s algorithm continue with this character
- The resulting tree is the true suffix tree where no suffix is prefix of another suffix and where each suffix ends at a leaf.

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

12 Text Search

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik

Wintersemester 2007/08

CoNe Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG