6. Linear-Time Construction of Suffix Trees

Dam Quang Tuan

Linear-Time Construction of Suffix Trees

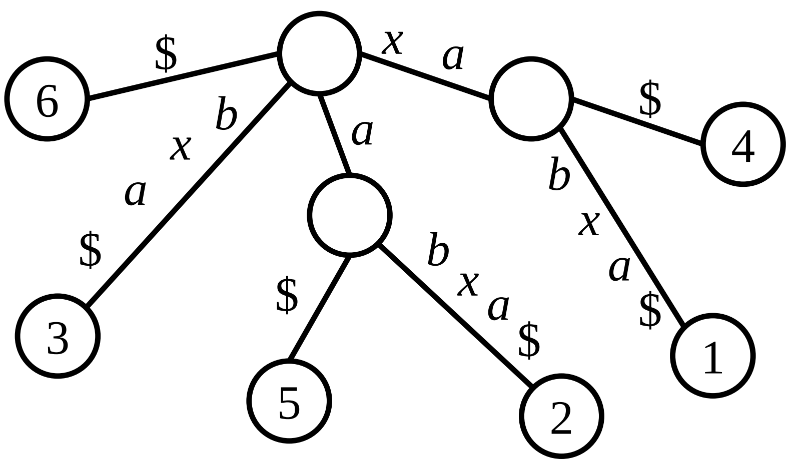
- **Two method**
  - Ukkonen's method
  - Weiner's method

Ukkonene's Method

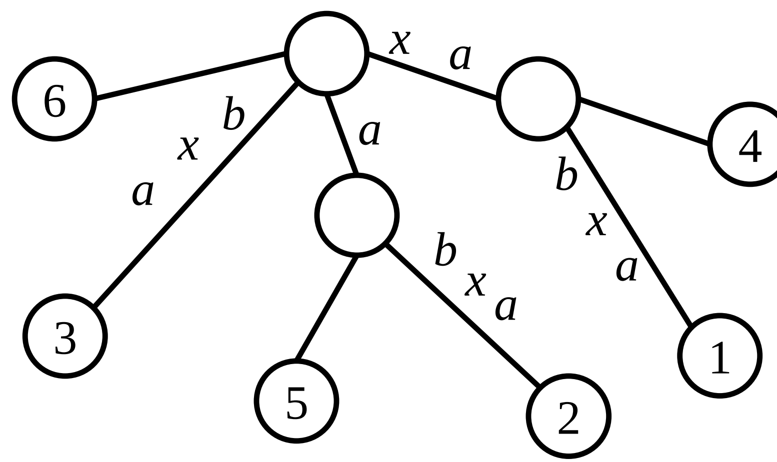
- **Ukkonen's method**
  - Linear-time construction algorithm
  - Space-saving improvement over Weiner's method
  - The simplicity of its description, proof and time analysis

Implicit Suffix Trees

- **Definition**
  - An *implicit suffix tree* for string S is a tree obtained from the suffix tree for $S\$$ by removing every copy of the terminal symbol \$ from the edge labels of the tree, then removing any edge that has no label and then removing any node that does not have at least two children

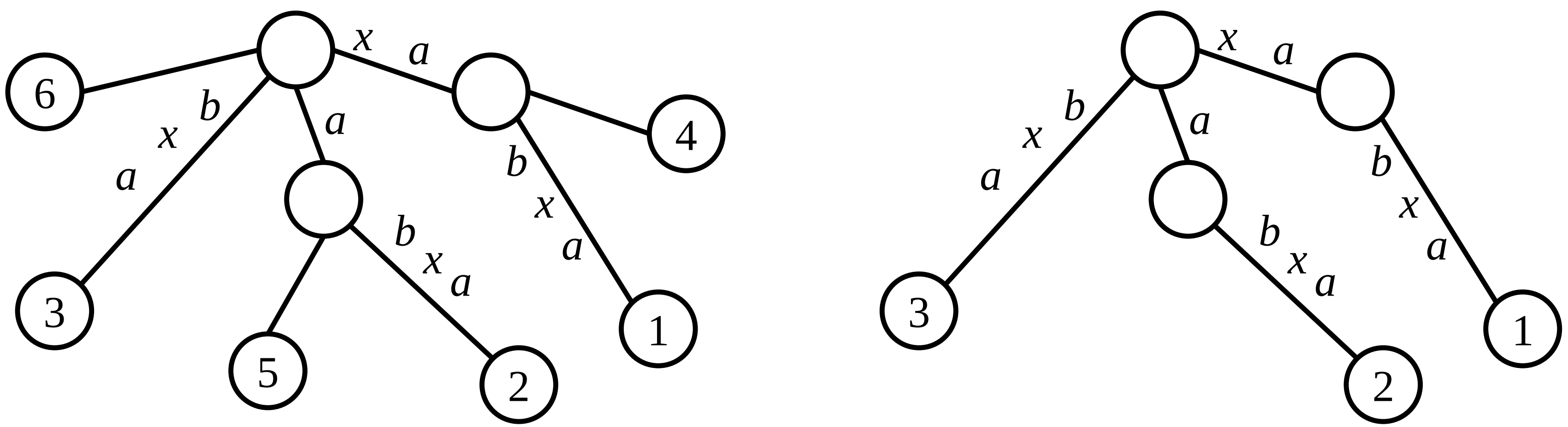


Suffix tree for string *xabxa*\$



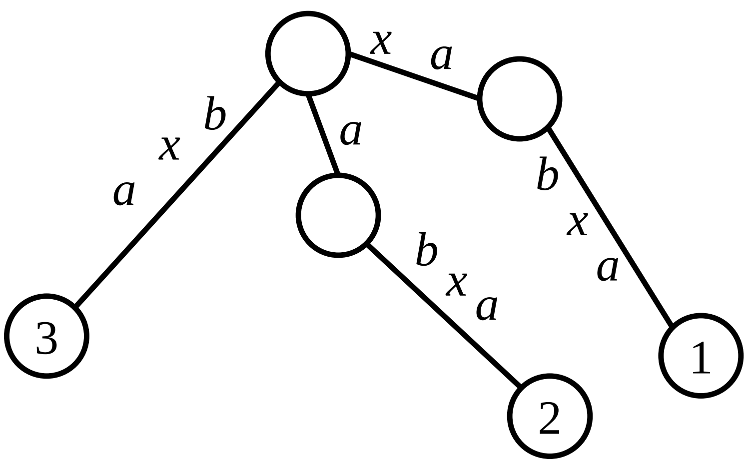
Implicit Suffix Trees

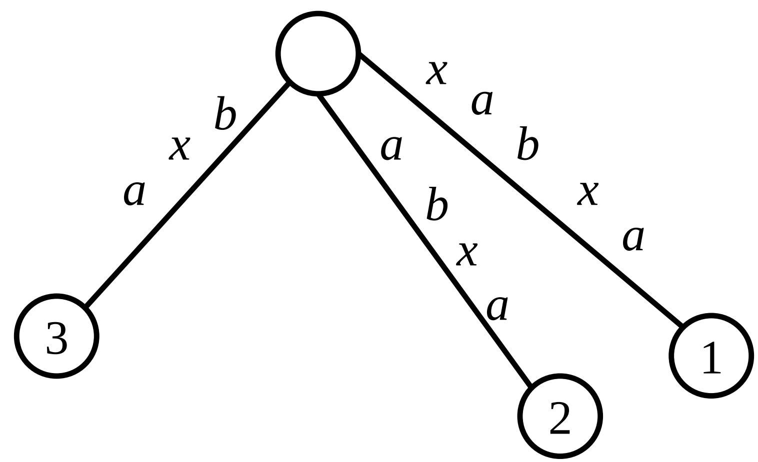
- **Definition**
  - An *implicit suffix tree* for string *S* is a tree obtained from the suffix tree for *S*$ by removing every copy of the terminal symbol $ from the edge labels of the tree, then removing any edge that has no label and then removing any node that does not have alt least two children



Implicit Suffix Trees

- **Definition**
  - An *implicit suffix tree* for string *S* is a tree obtained from the suffix tree for *S*$ by removing every copy of the terminal symbol $ from the edge labels of the tree, then removing any edge that has no label and then removing any node that does not have at least two children

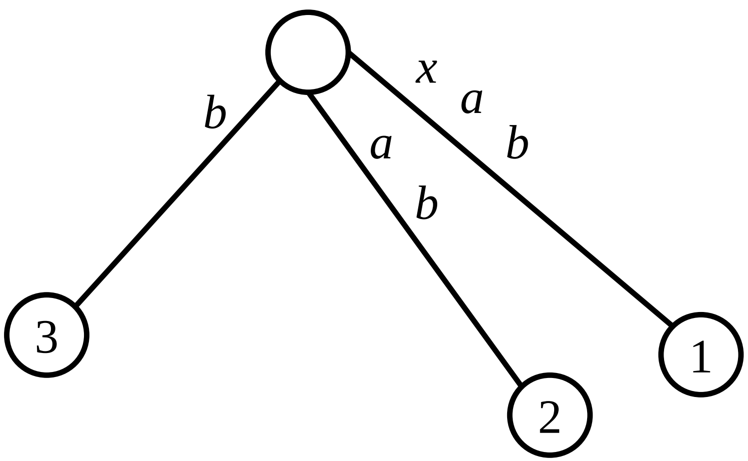




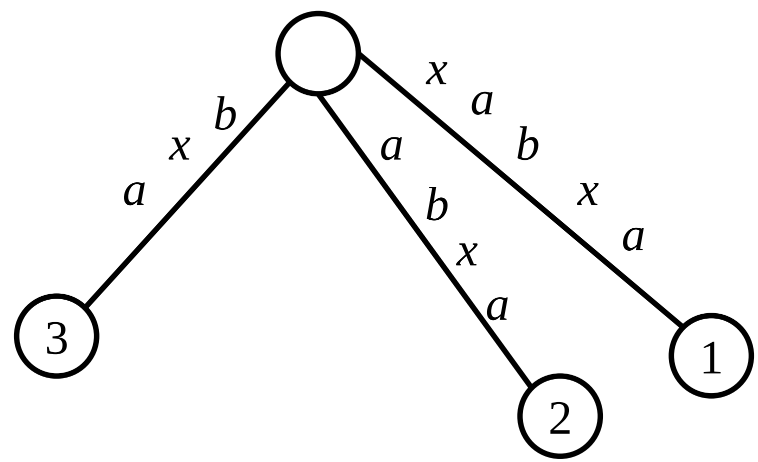
Implicit suffix tree for string *xabxa*

Implicit Suffix Trees

- **Definition**
  - We denote the implicit suffix tree of the string $S[1..i]$ by $\mathcal{T}_i$, for i from 1 to m
  - $S = xabxa$



$\mathcal{T}_3$



$\mathcal{T}_5$

Implicit Suffix Trees

- **Implicit suffix trees**
  - Somewhat less informative than true suffix trees
  - Will use them as a tool in Ukkonen's algorithm to finally obtain the true suffix tree for *S*

Ukkonen's Algorithm at a High Level

- **Ukkonen's algorithm**
  - Constructs an implicit suffix tree $\mathcal{T}_i$ for each prefix $S[1..i]$ of S
    - Starting from $\mathcal{T}_1$ and incrementing i by on until $\mathcal{T}_m$
  - The true suffix tree for S is constructed from $\mathcal{T}_m$
  - The time for the entire algorithm is $O(m)$

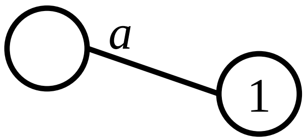
Ukkonen's Algorithm at a High Level

- **Ukkonen's algorithm**
  - First present an $O(m^3)$-time method
  - Divided into m phases
    - In phase $i + 1$, tree $\mathcal{T}_{i+1}$ is constructed from $\mathcal{T}_i$

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

- **Example**
  - Divided into m phases
    - In phase $i + 1$, tree T_{i+1} is constructed from T_i

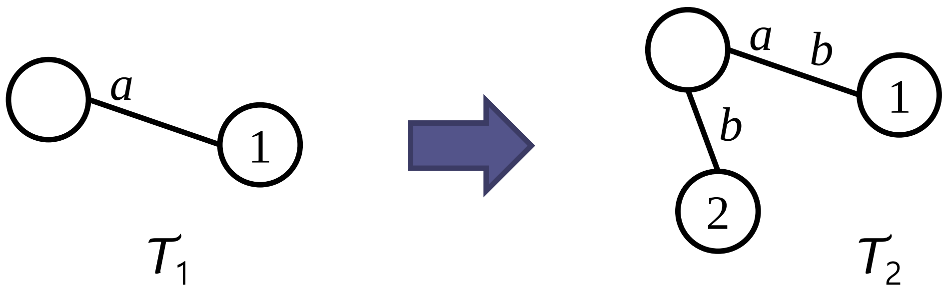


T_1

Ukkonen's Algorithm

- **Example**

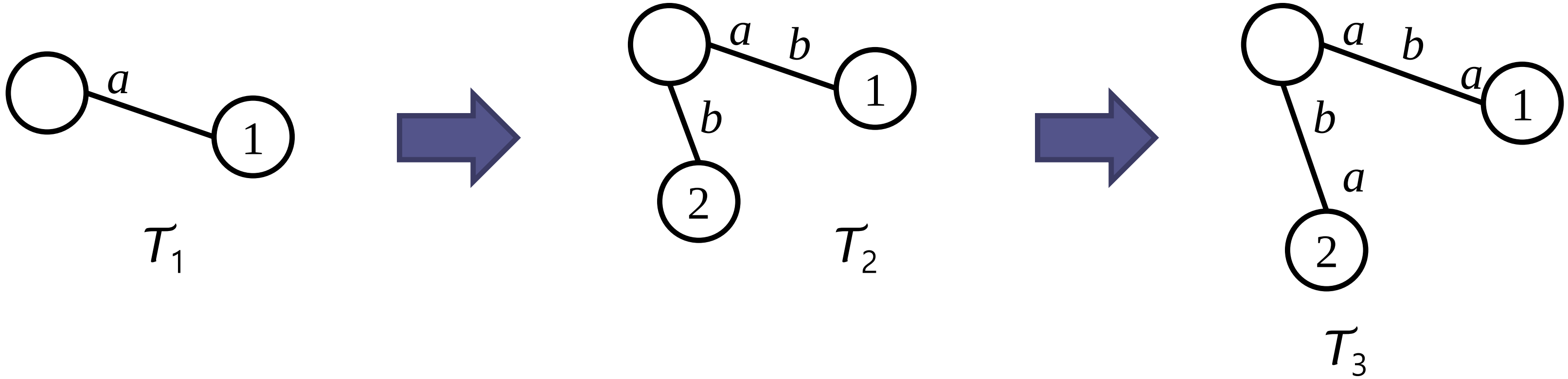
i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$



Ukkonen's Algorithm

- **Example**

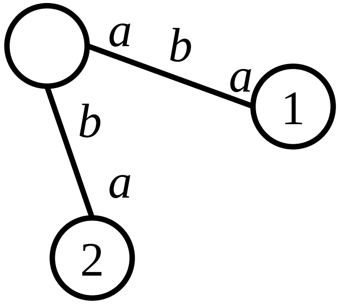
i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$



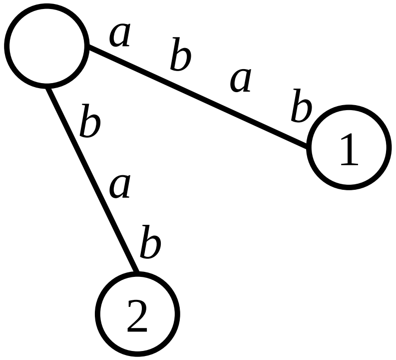
Ukkonen's Algorithm

- **Example**

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$



T_3

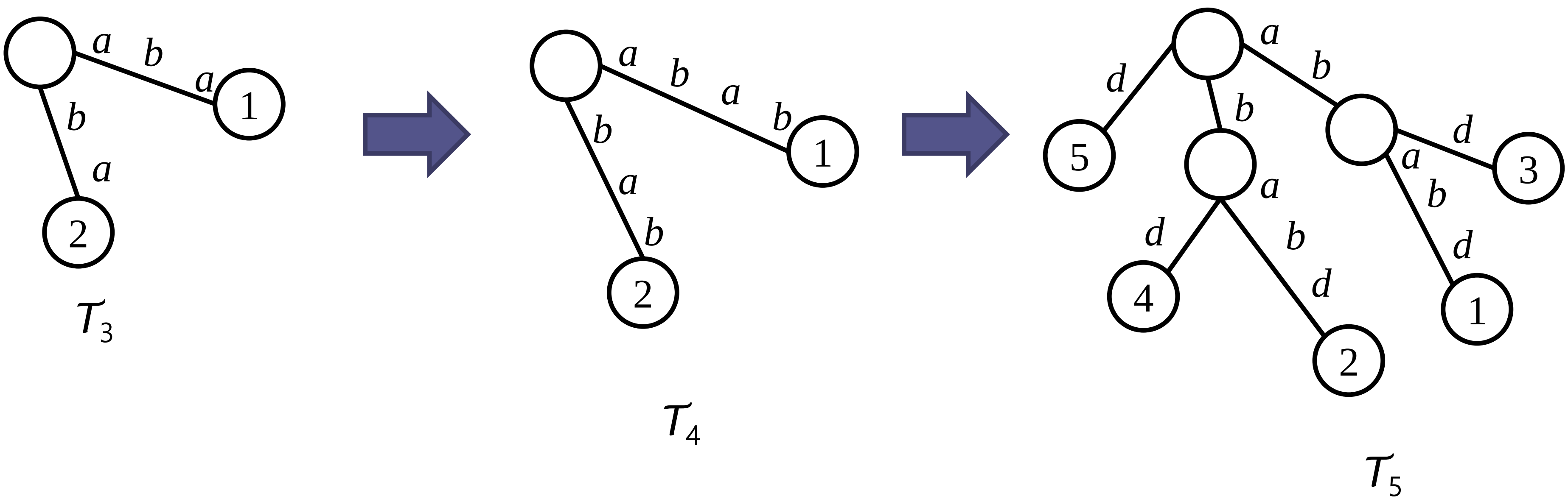


T_4

Ukkonen's Algorithm

- **Example**

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

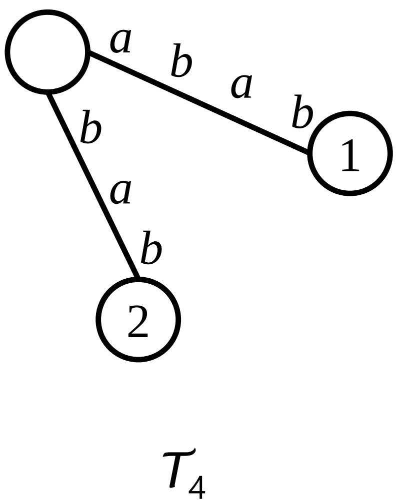


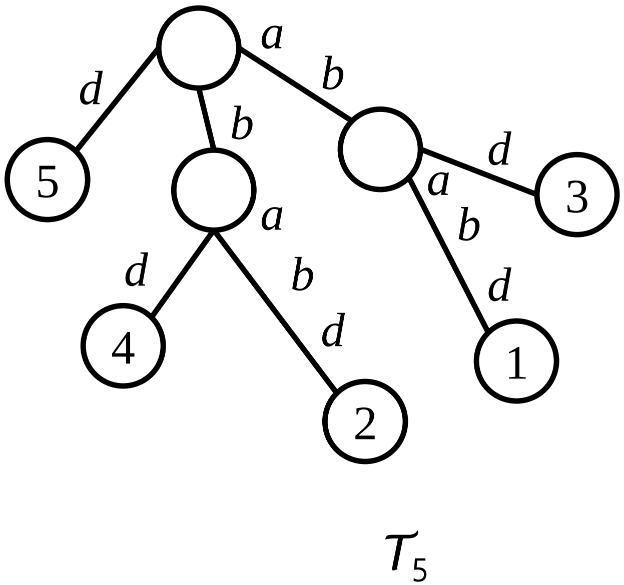
Ukkonen's Algorithm

- **Example**

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	$

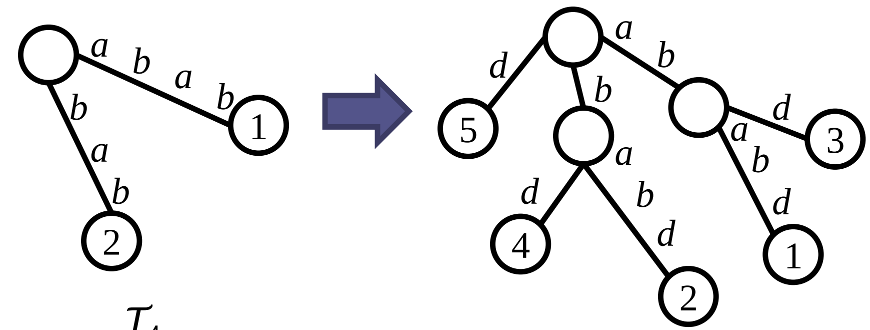
  - Each phase $i + 1$ is further divided into $i + 1$ extensions



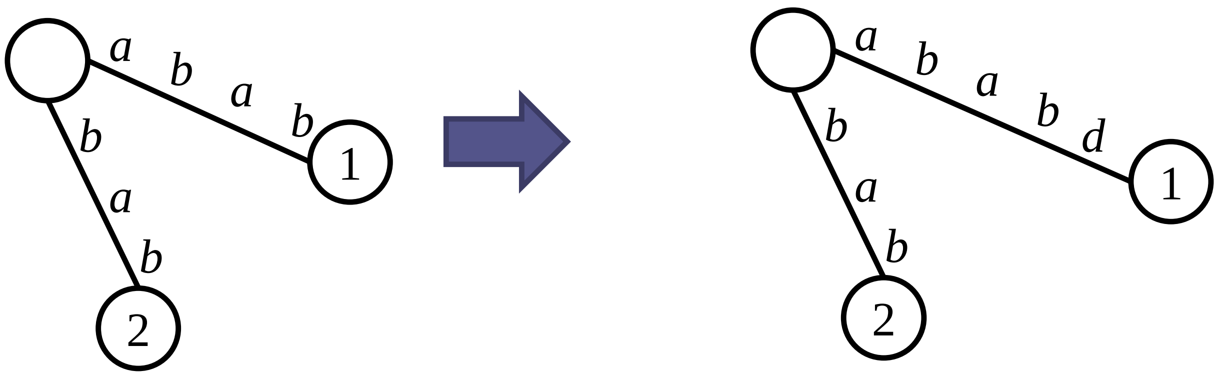


Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$



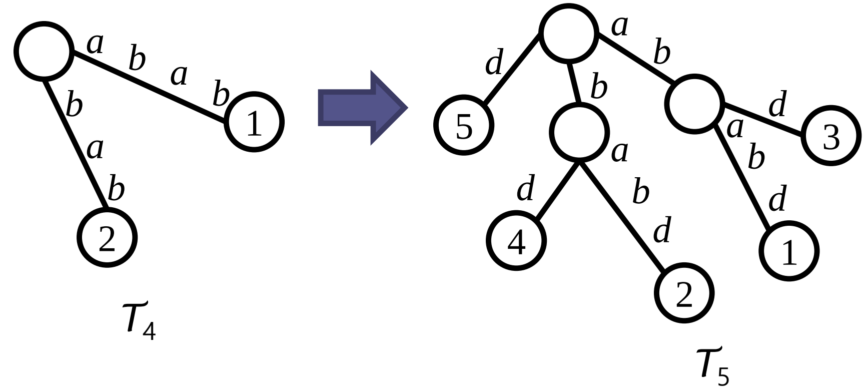
T_4 T_5

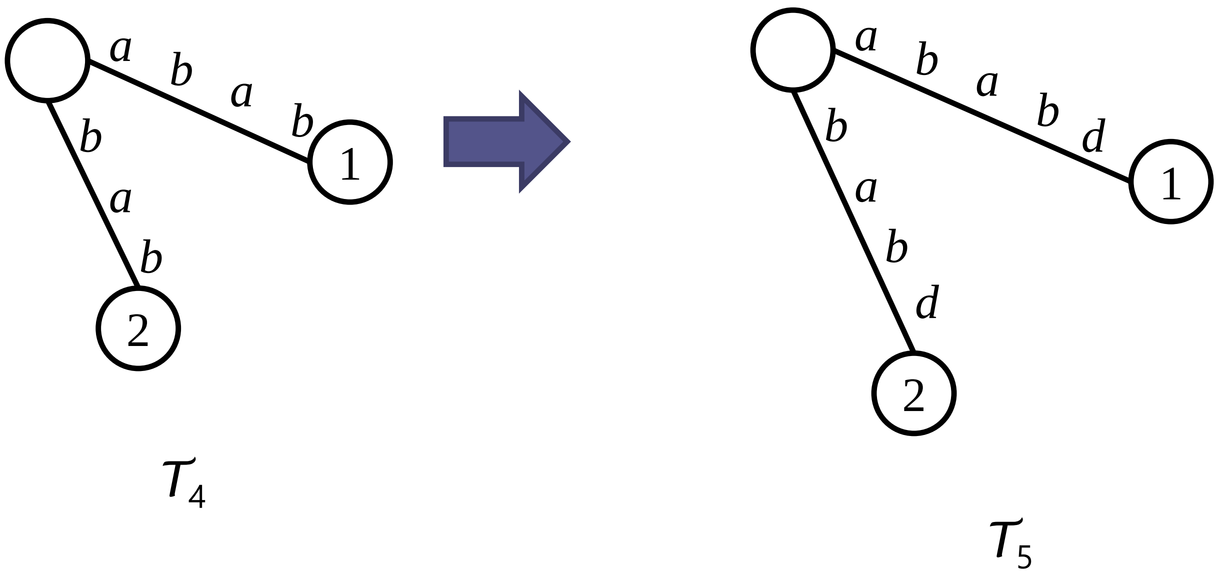


T_4 T_5

Ukkonen's Algorithm

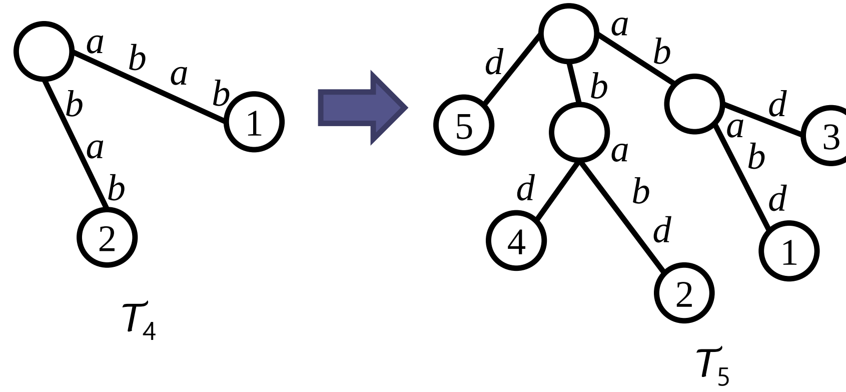
i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

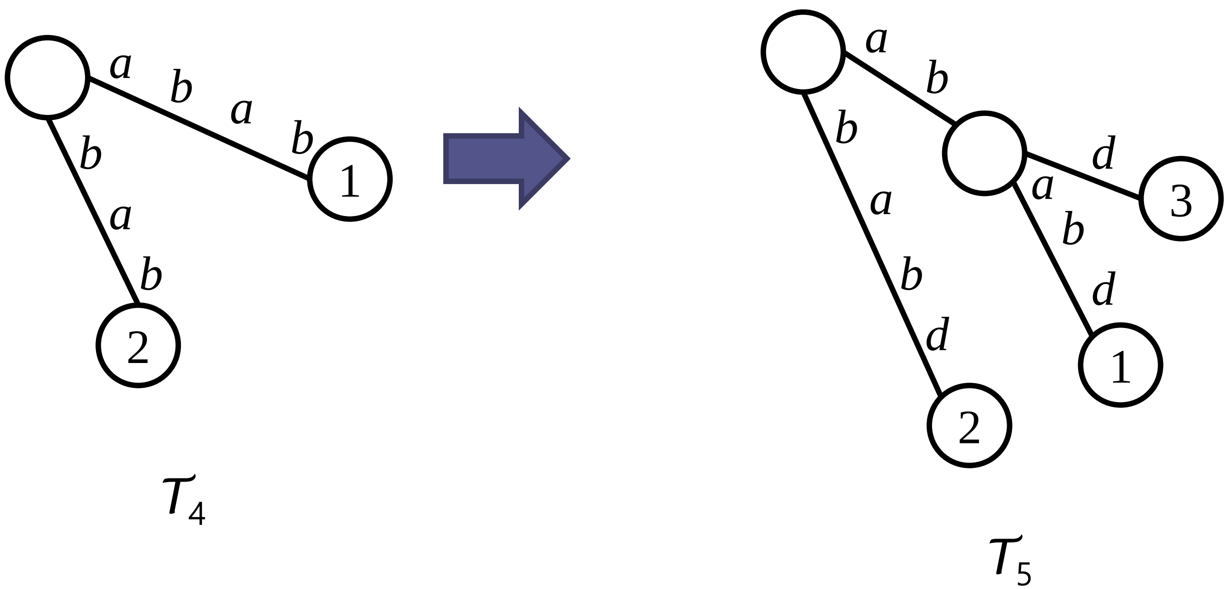




Ukkonen's Algorithm

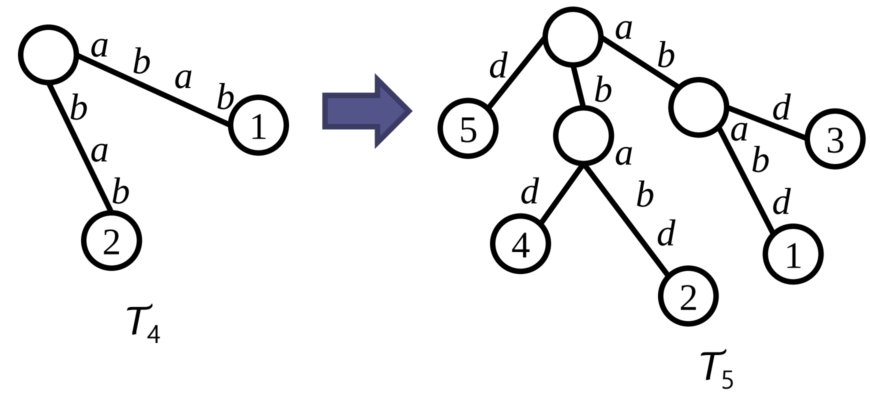
i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

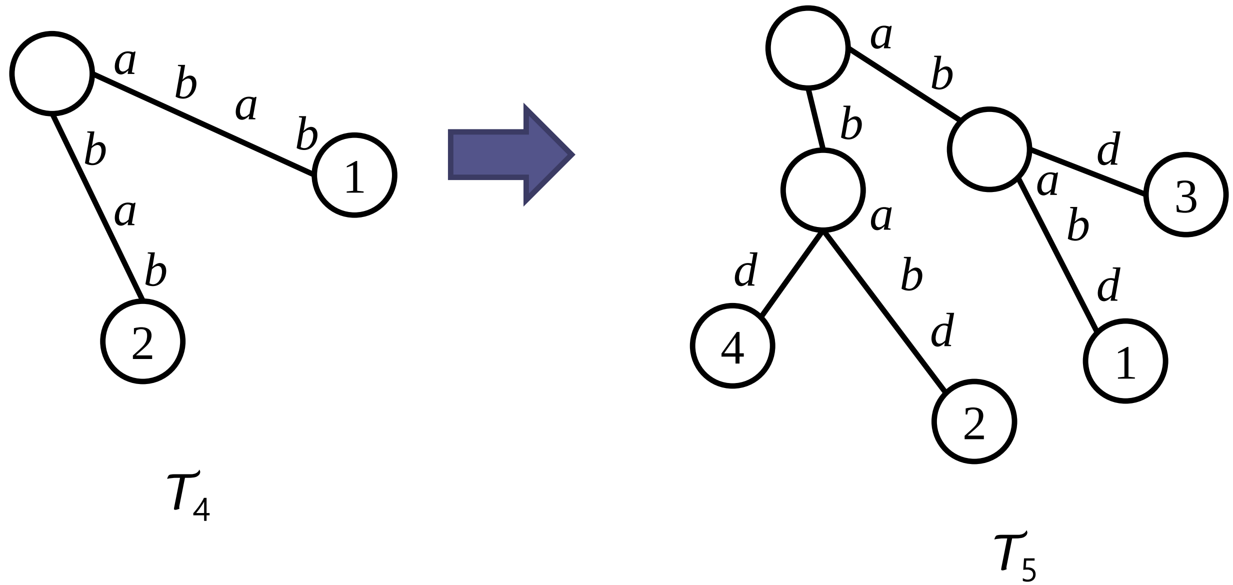




Ukkonen's Algorithm

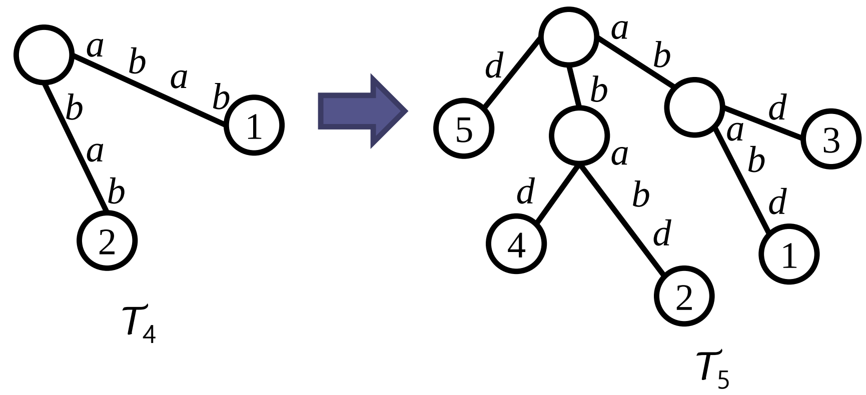
i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

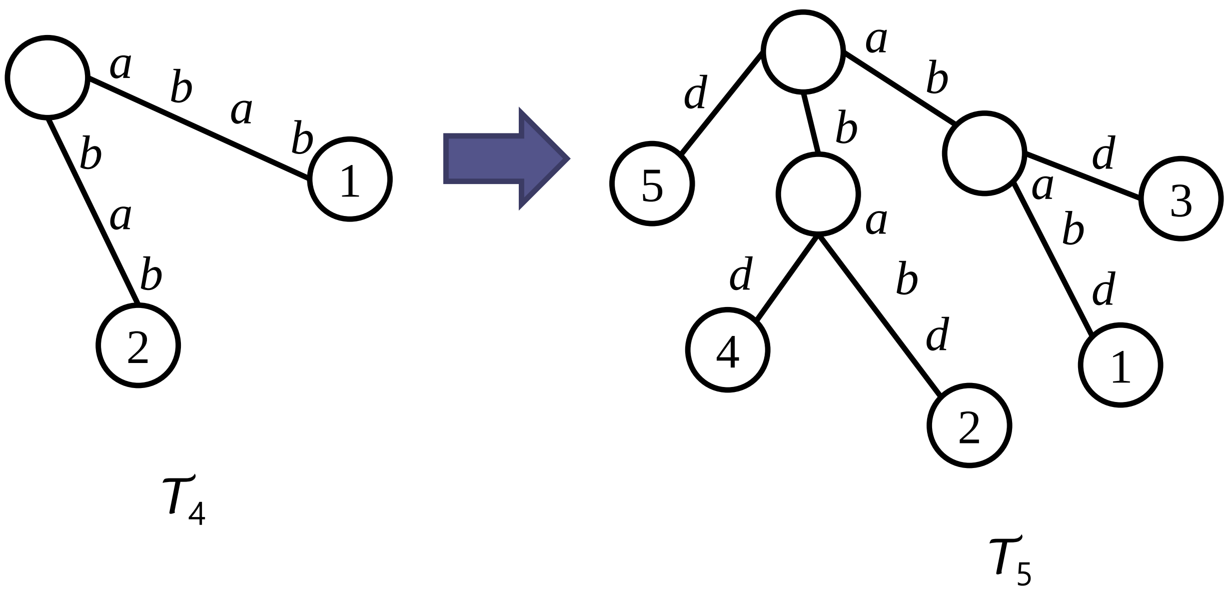




Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

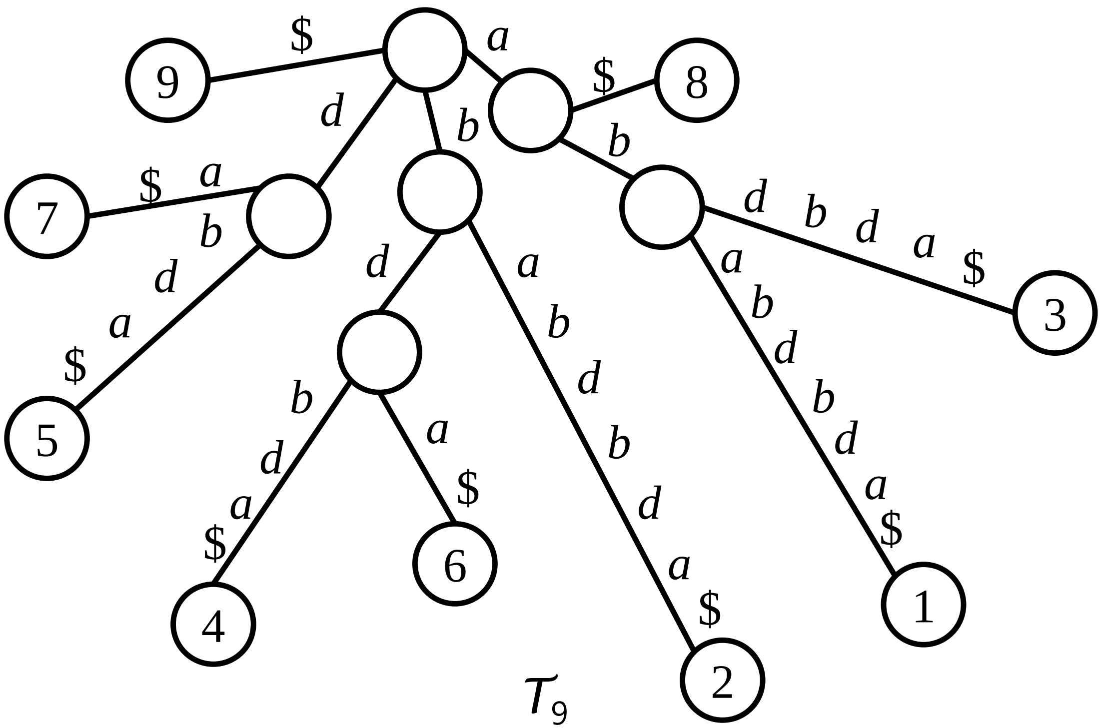




Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9
S	a	b	a	b	d	b	d	a	\$

- **Example**
  - Construct a suffix tree in $O(m^3)$ time
    - m phases, at most m extension in any phase, and at most m comparison in any extension



Suffix Extension Rules

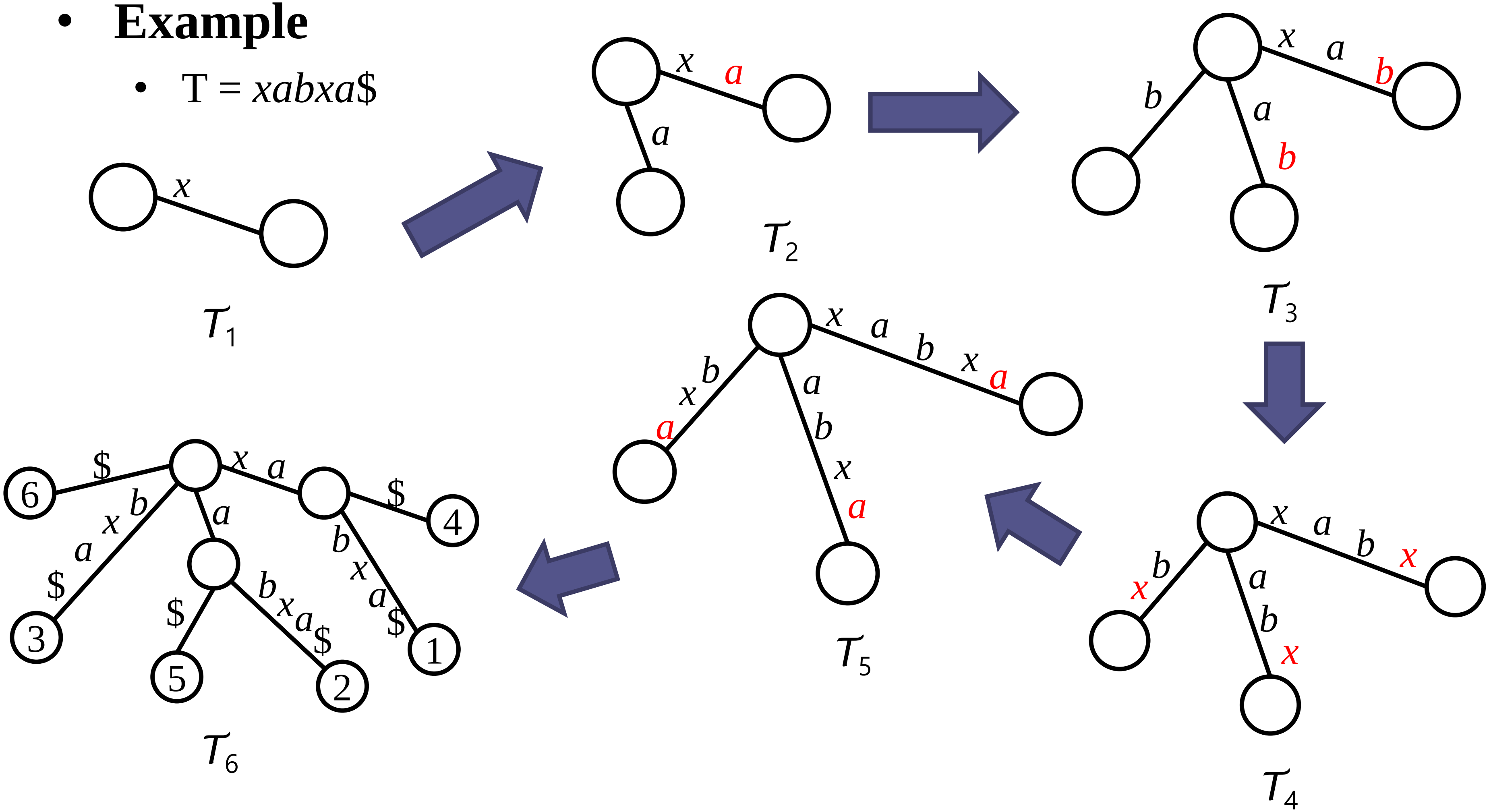
- **Suffix extension rules**
  - $S[j..i] = \beta$ is a suffix of $S[1..i]$
  - In extension j, when the algorithm finds the end of β in the current tree
  - It extends β to be sure the suffix $\beta S(i + 1)$ is in the tree
  - It does this according to one of the three rules

Suffix Extension Rules

- **Suffix extension rules**
  - Rule 1
    - In the current tree, path β ends at a leaf.
    - That is, the path from the root labeled β extends to the end of some leaf edge.
    - To update the tree, character $S(i + 1)$ is added to the end of the label on that leaf edge

Ukkonen's Algorithm at a High Level

- **Example**
  - T = *xabxa$*

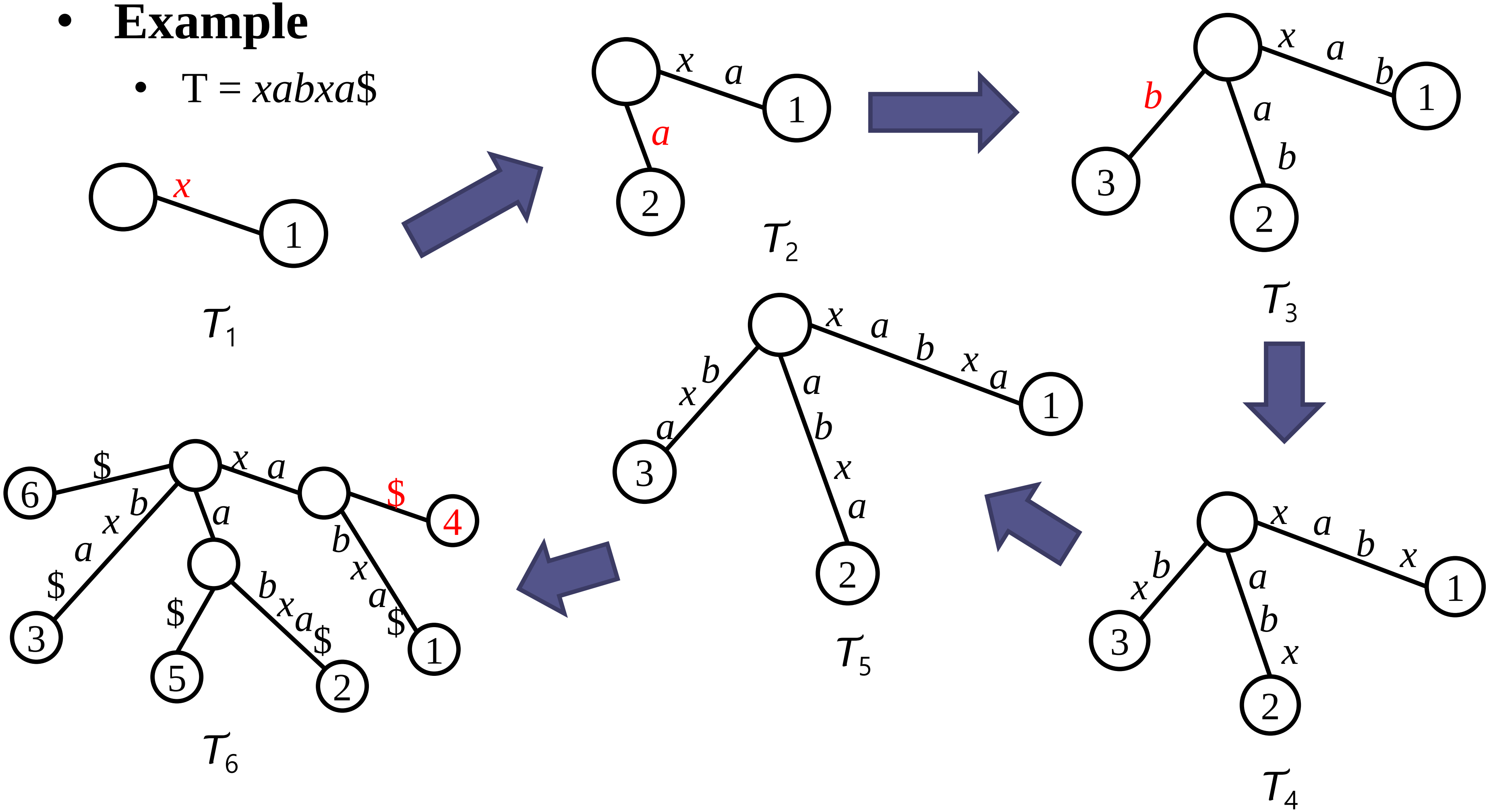


Suffix Extension Rules

- **Suffix extension rules**
  - Rule 2
    - No path from the end of string β starts with character $S(i + 1)$, but at least one labeled path continues from the end of β
    - In this case, a new leaf edge starting from the end of β must be created and labeled with character $S(i + 1)$.
    - A new node will also have to be created there if β ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

Ukkonen's Algorithm at a High Level

- **Example**
  - T = *xabxa$*

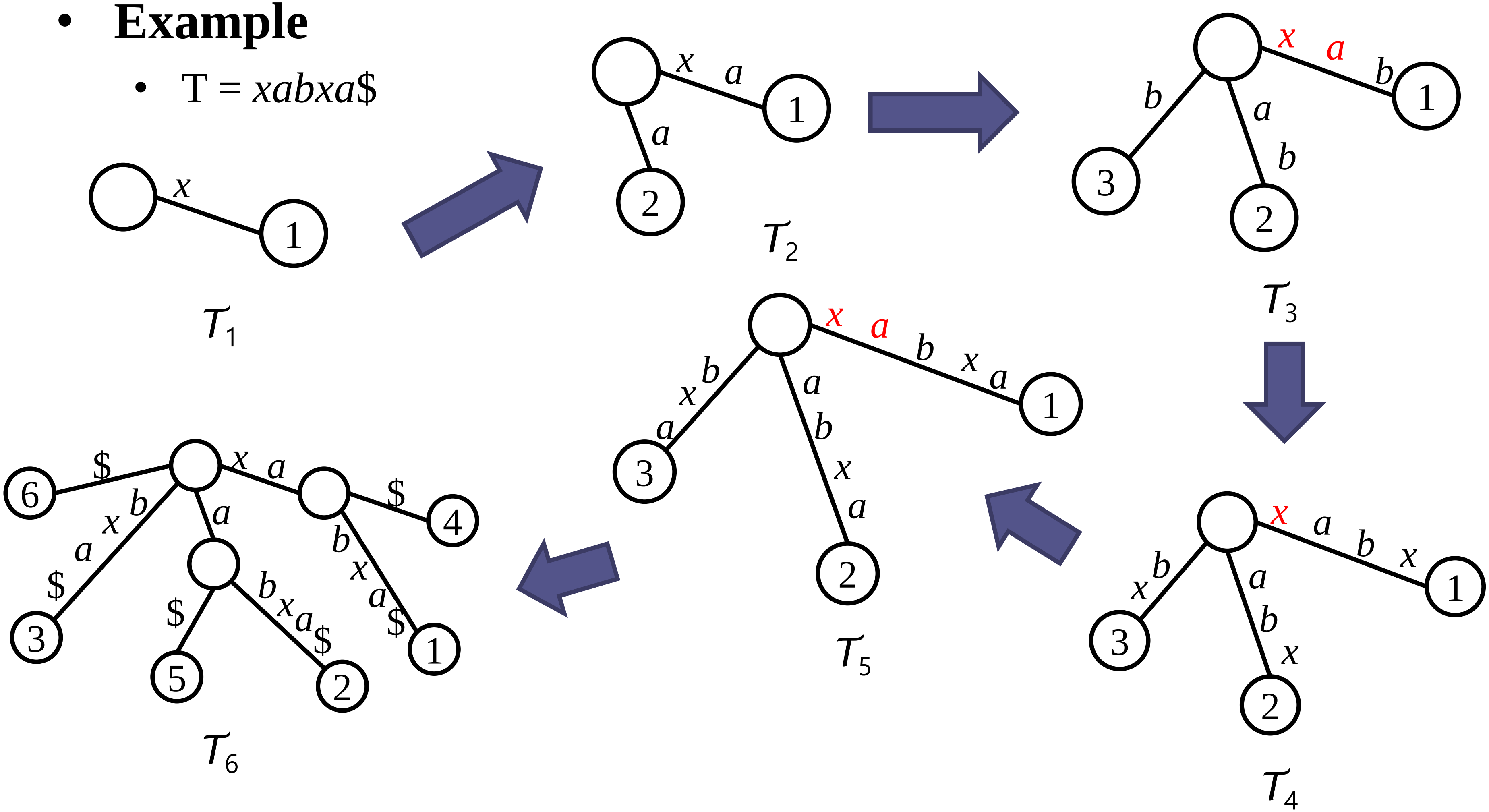


Suffix Extension Rules

- **Suffix extension rules**
  - Rule 3
    - Some path from the end of string β starts with character $S(i + 1)$.
    - In this case the string $\beta S(i + 1)$ is already in the current tree, so we do nothing.

Ukkonen's Algorithm at a High Level

- **Example**
  - T = *xabxa*$



Implementation and Speedup

- **Implementation and speedup**
  - Using the suffix extension rules
    - Rule 1, Rule2, and Rule3
  - Using a few observation and implementation trick
  - Reduce the $O(m^3)$ time to $O(m)$ time

Implementation and Speedup

- **Implementation and speedup**
  - Using the suffix extension rules
    - Rule 1, Rule2, and Rule3
  - Using a few observation and implementation trick
  - Reduce the $O(m^3)$ time to $O(m)$ time
  - The most important element of the acceleration is the use of *suffix links*

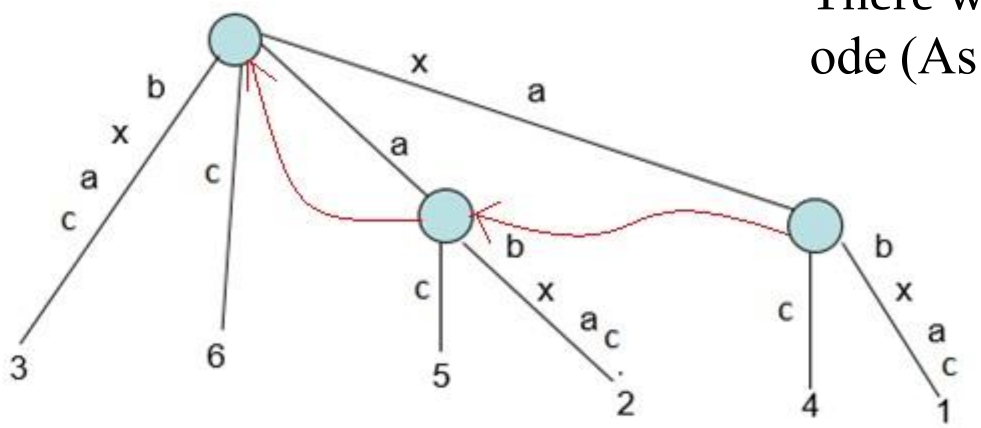
Suffix link

- **Definition of *suffix link***
  - Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possible empty) substring.
  - For an internal node v with path-label $x\alpha$, if there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a *suffix link*.

Suffix link

- **Definition of suffix link**

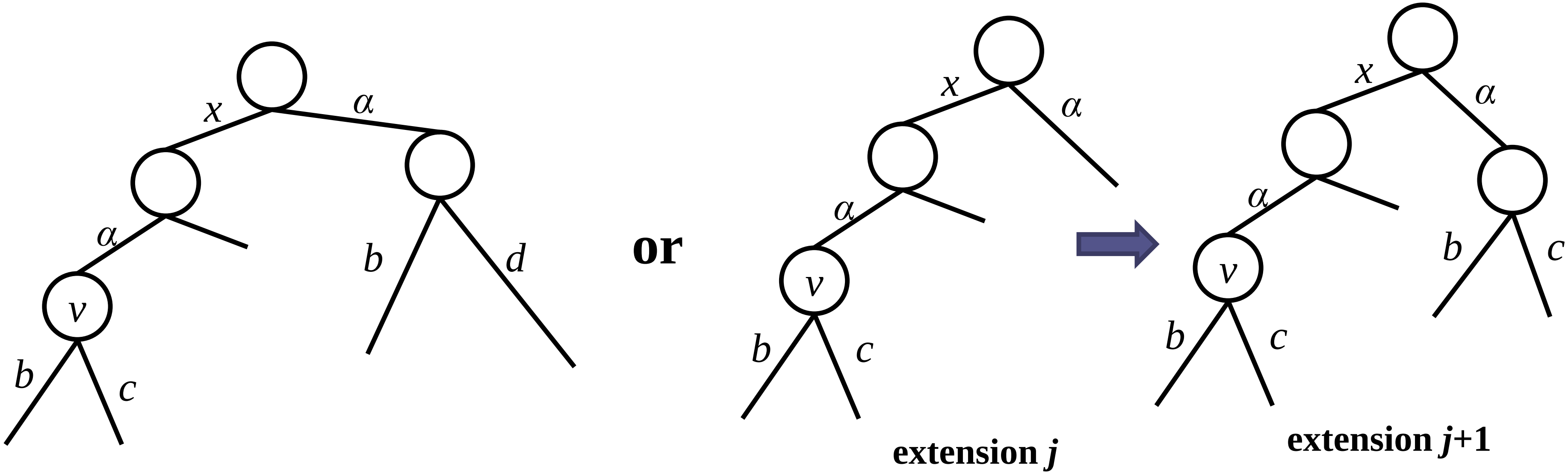
If α is empty string, suffix link from internal node will go to root node.
There will not be any suffix link from root n ode (As it's not considered as internal node).



Suffix Links in red arrows

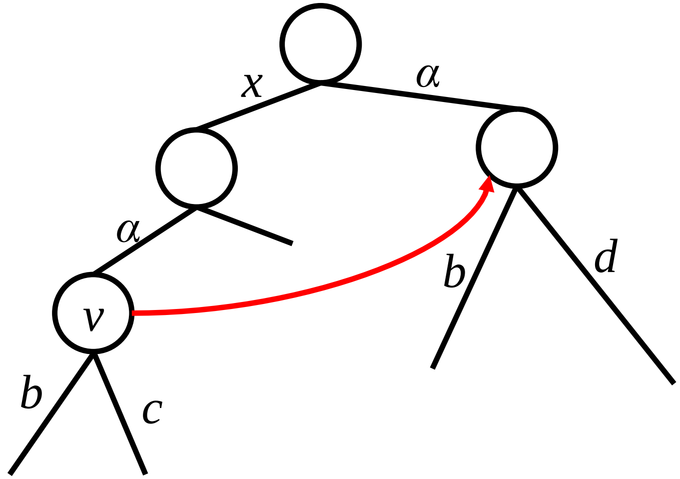
Suffix link

- **Lemma 6.1.1.**
  - In extension j of some phase i, if a new internal node v with path-label x α is added, then in extension j+1 in the same phase i:
    - Either the path labelled α already ends at an internal node (or root node if α is empty)
    - OR a new internal node at the end of string α will be created



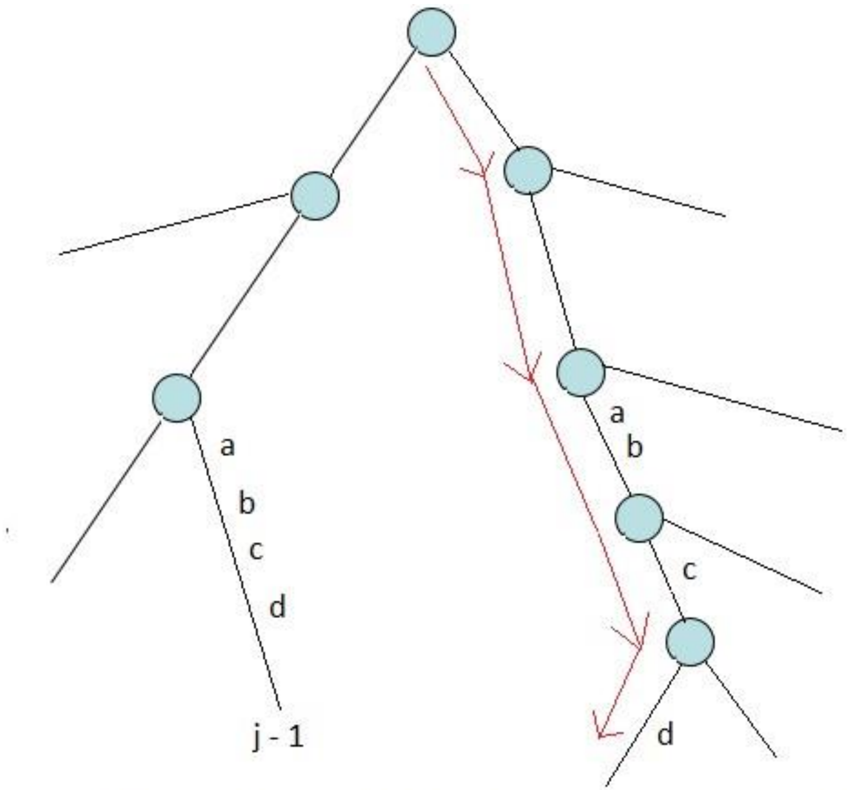
Suffix link

- **Corollary 6.1.1.**
  - In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.



Suffix link

- **How suffix links are used to speed up the implementation?**

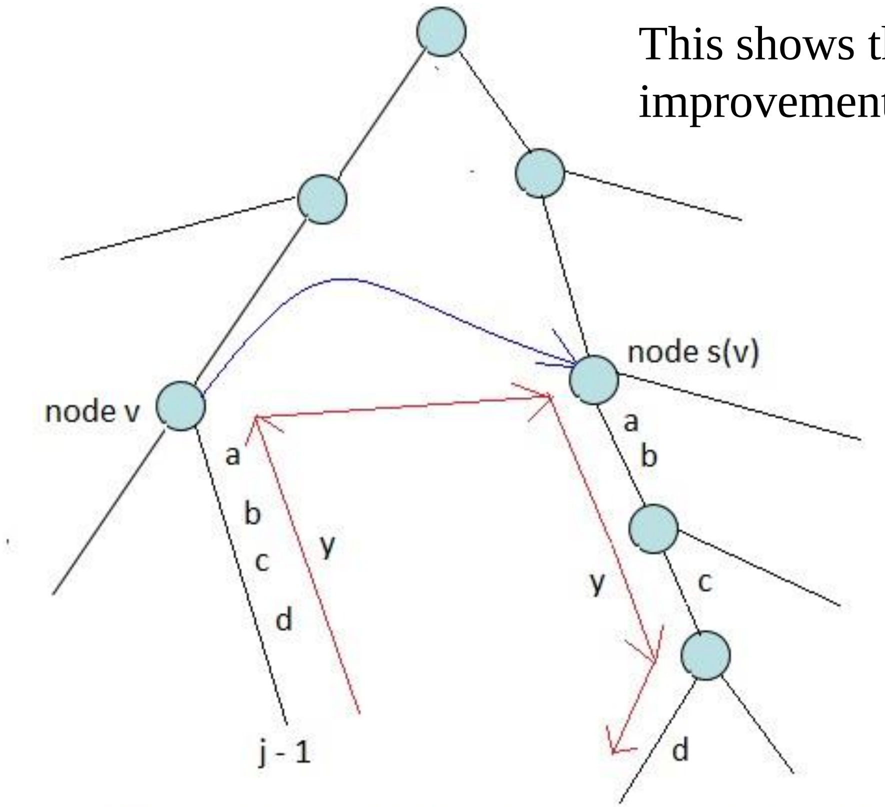


Traversal from root to leaf in extenstion j of phase i+1, to find end of S[j..i] , when suffix link is not used

In extension j of phase i+1, we need to find the end of the path from the root labelled S[j..i] in the current tree.
One way is start from root and traverse the edg es matching S[j..i] string. Suffix links provide a short cut to find end of the path.

Suffix link

- **How suffix links are used to speed up the implementation?**



Traversal from root to leaf in extenstion j of phase i+1, to find end of S[j..i] , when suffix link (blue arrow) is used

This shows the use of suffix link is an improvement over the process.

When there is a suffix link from node v to node s(v), then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node s(v) to a leaf. I n Figure there is a path label "abcd" from node v to a leaf, then there is a path will same label "abcd" from node s(v) to a leaf.

Single Extension Algorithm

- **Single extension algorithm**

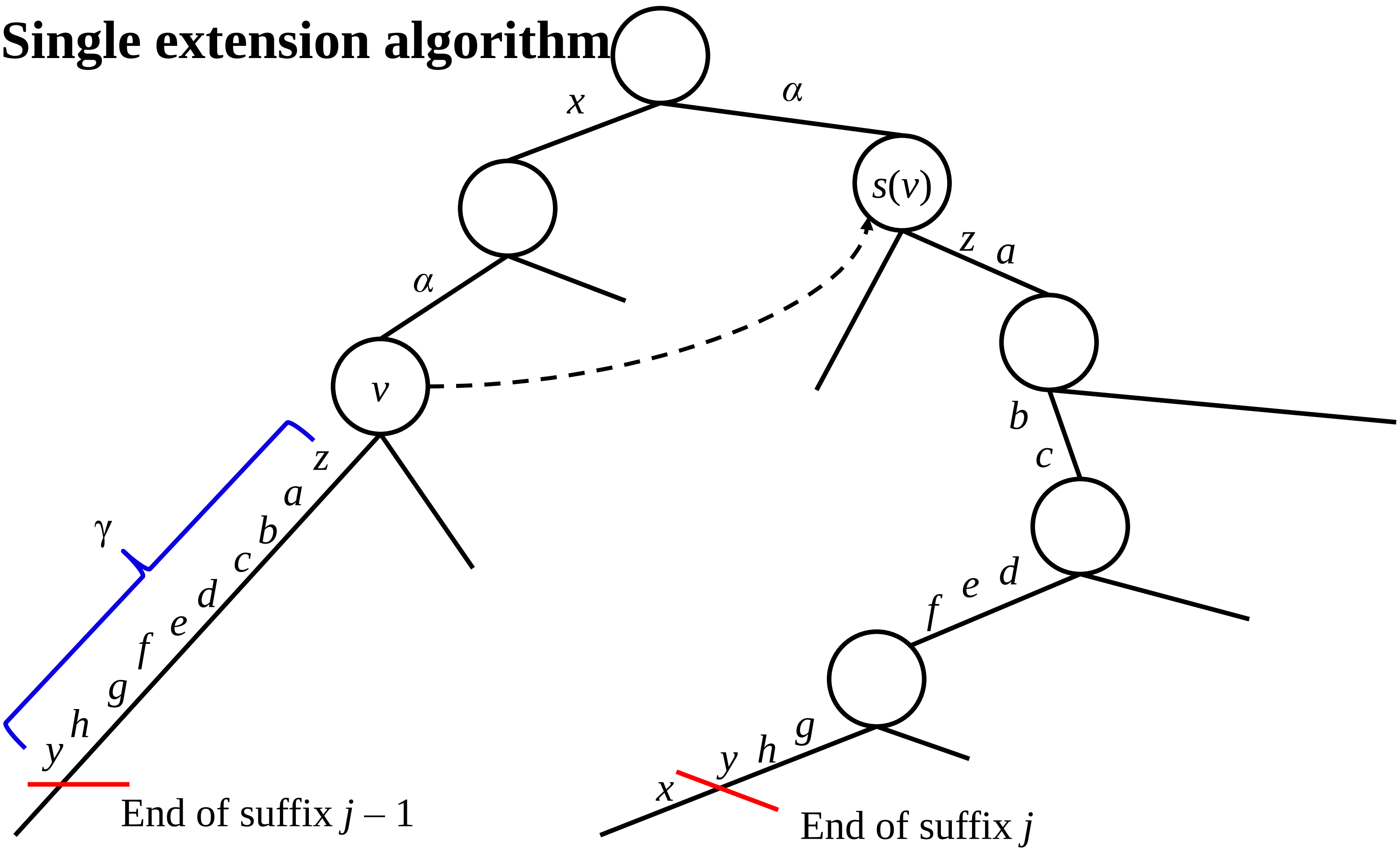
  - Find the first node v at or above the end of $S[j\text{-}1..i]$ that either has a suffix link from it or is the root.

  - This requires walking up at most on edge from the end of $S[j\text{-}1..i]$ in the current tree.

  - Let γ (possible empty) denote the string between v and the end of $S[j\text{-}1..i]$

Single Extension Algorithm

- **Single extension algorithm**



Single Extension Algorithm

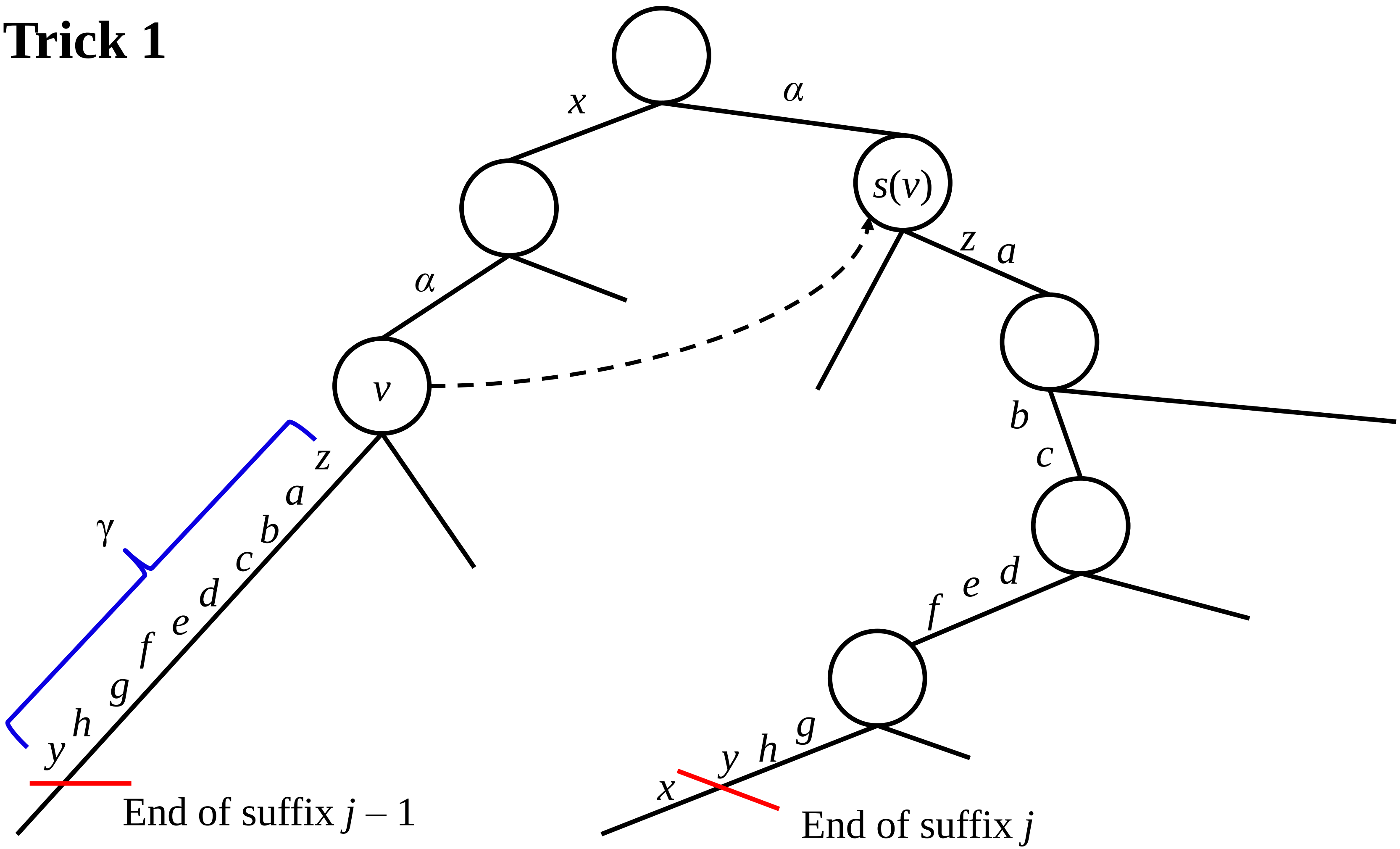
- **Single extension algorithm**
  - If v is not the root, traverse the suffix link from v to node $s(v)$ and then walk down from $s(v)$ following the path for string γ.
  - If v is the root, then follow the path for $S[j..i]$ from the root (as in the naïve algorithm).
  - Using the extension rules, ensure that the string $S[j..i]S(i+1)$ is in the tree

Ukkonen's Algorithm with Suffix Link

- **Ukkonen's Algorithm with Suffix Link**
  - The use of suffix links is clearly a practical improvement over walking from the root in each extension, as done in the naïve algorithm
  - But does their use improve the worst-case running time?
  - Introduce a trick that will reduce the worse-case time for the algorithm to $O(m^2)$

Trick number 1: skip/count trick

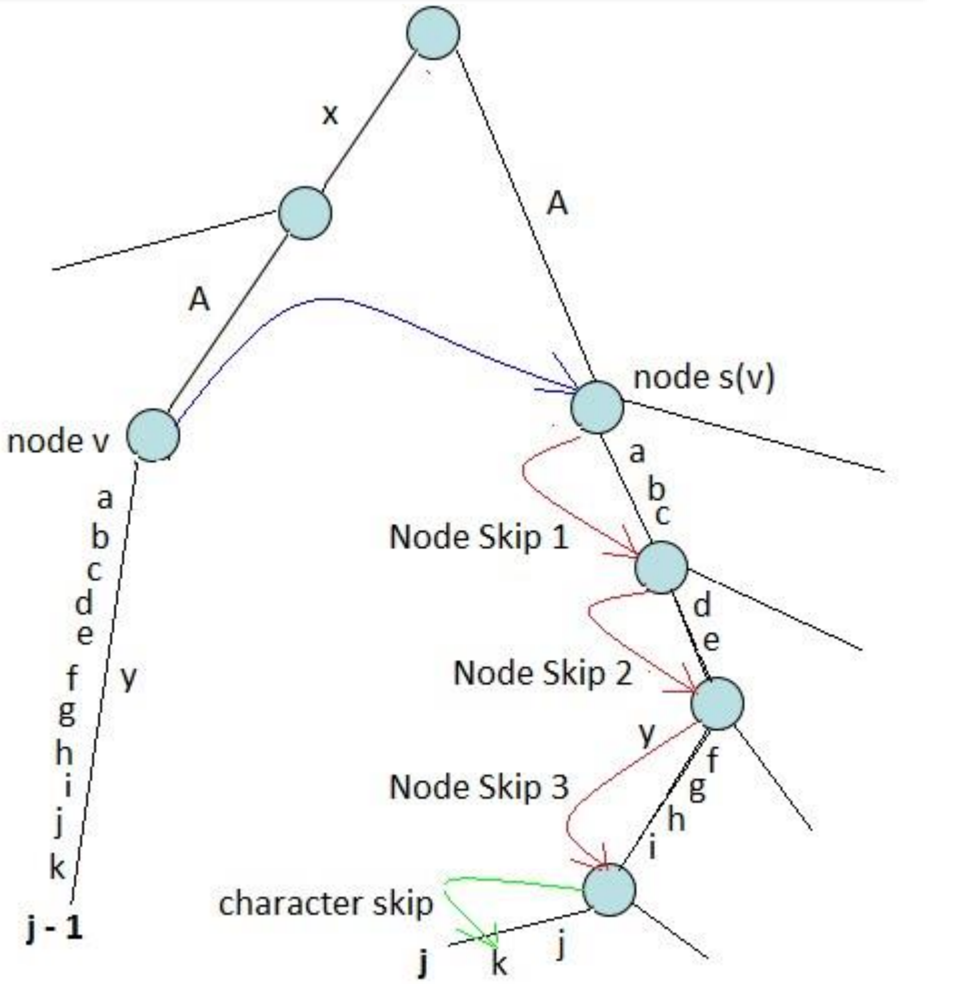
- **Trick 1**



Trick number 1: skip/count trick

- **Trick 1**

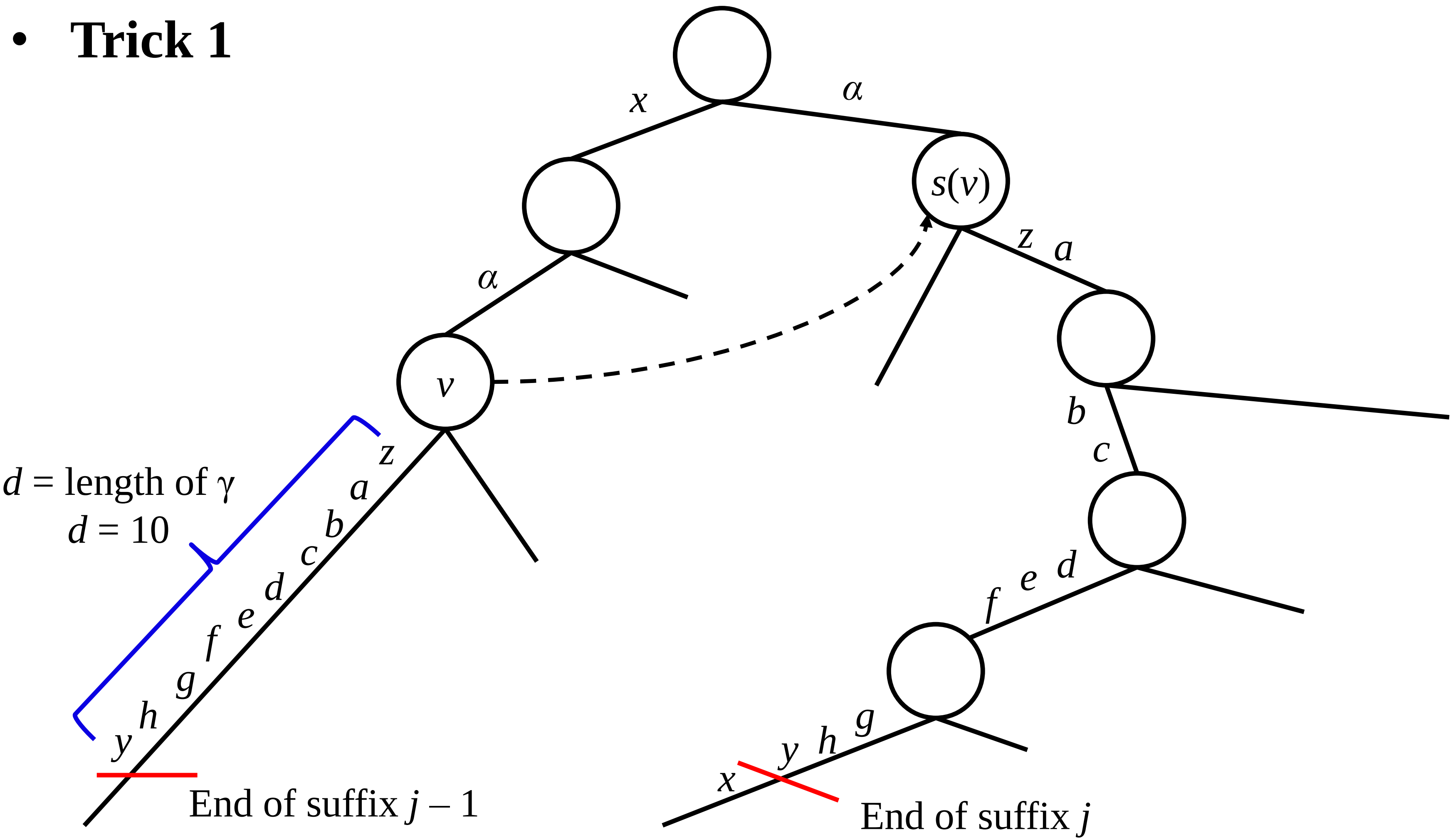
Instead of matching path character by char acter as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characte rs we need to travel.
If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge



skip/count trick: substring y from node v has length 11. substring y from node s(v) is two characters down the last node , after 3 node skips

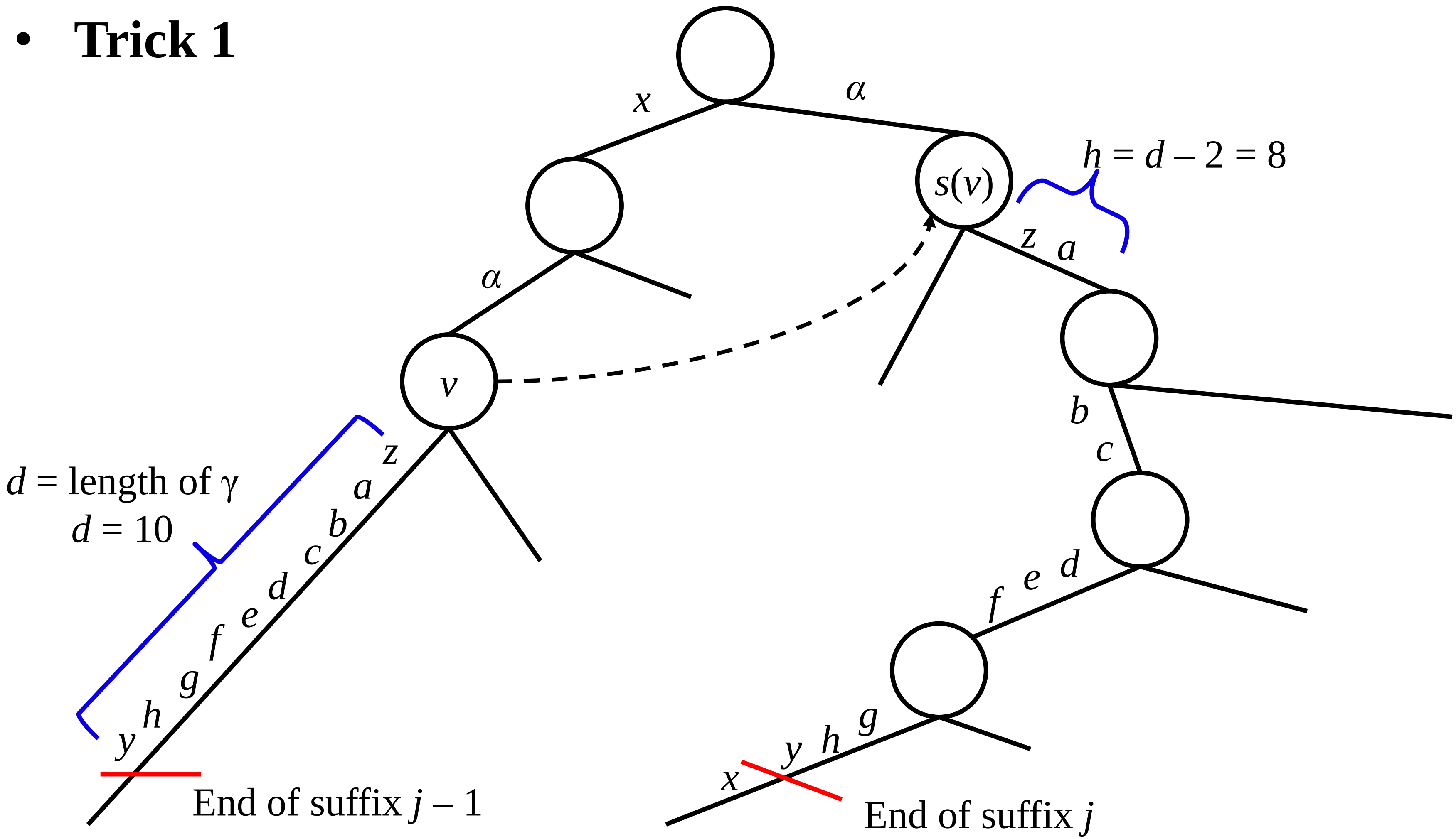
Trick number 1: skip/count trick

- **Trick 1**



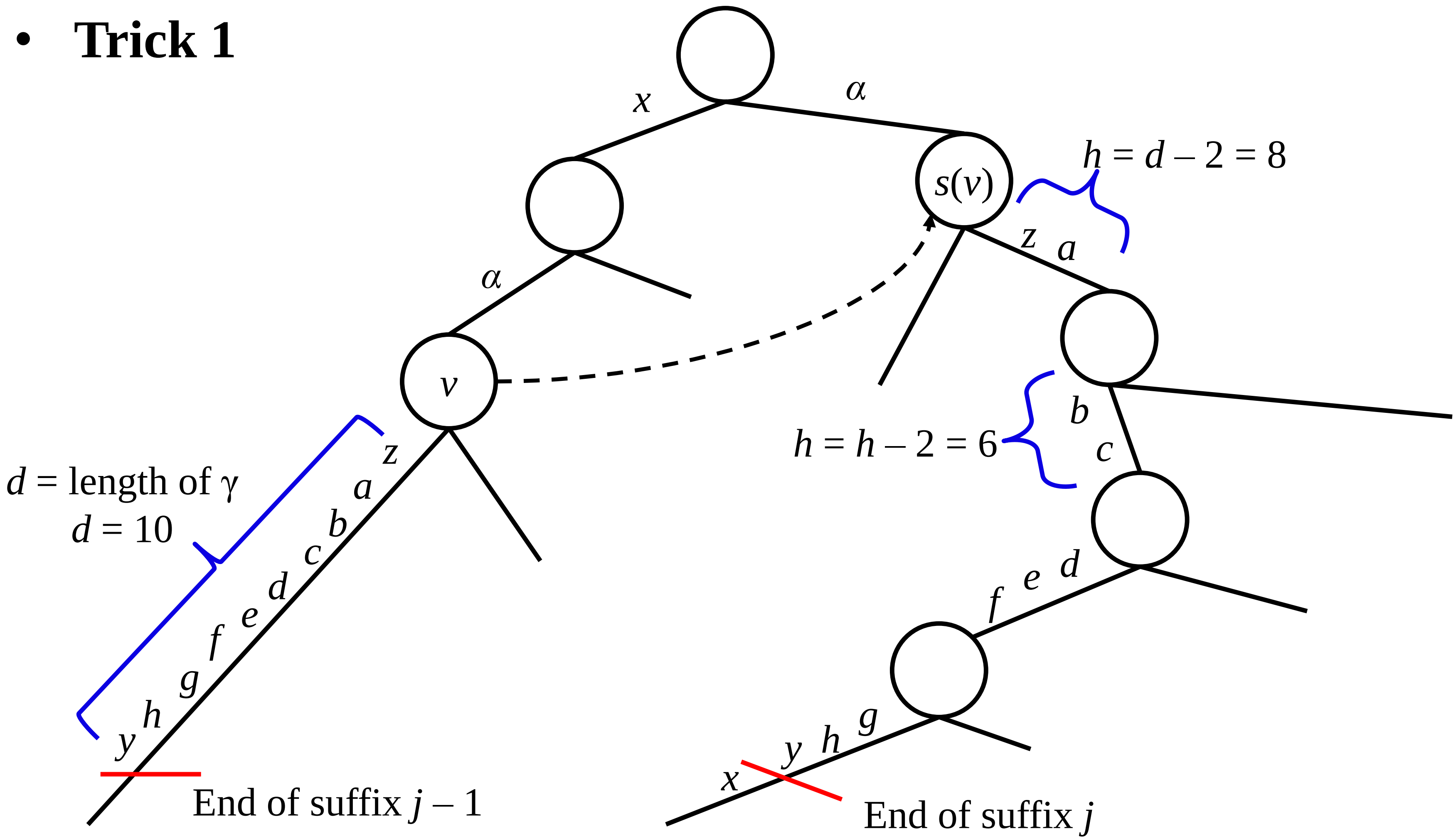
Trick number 1: skip/count trick

- **Trick 1**



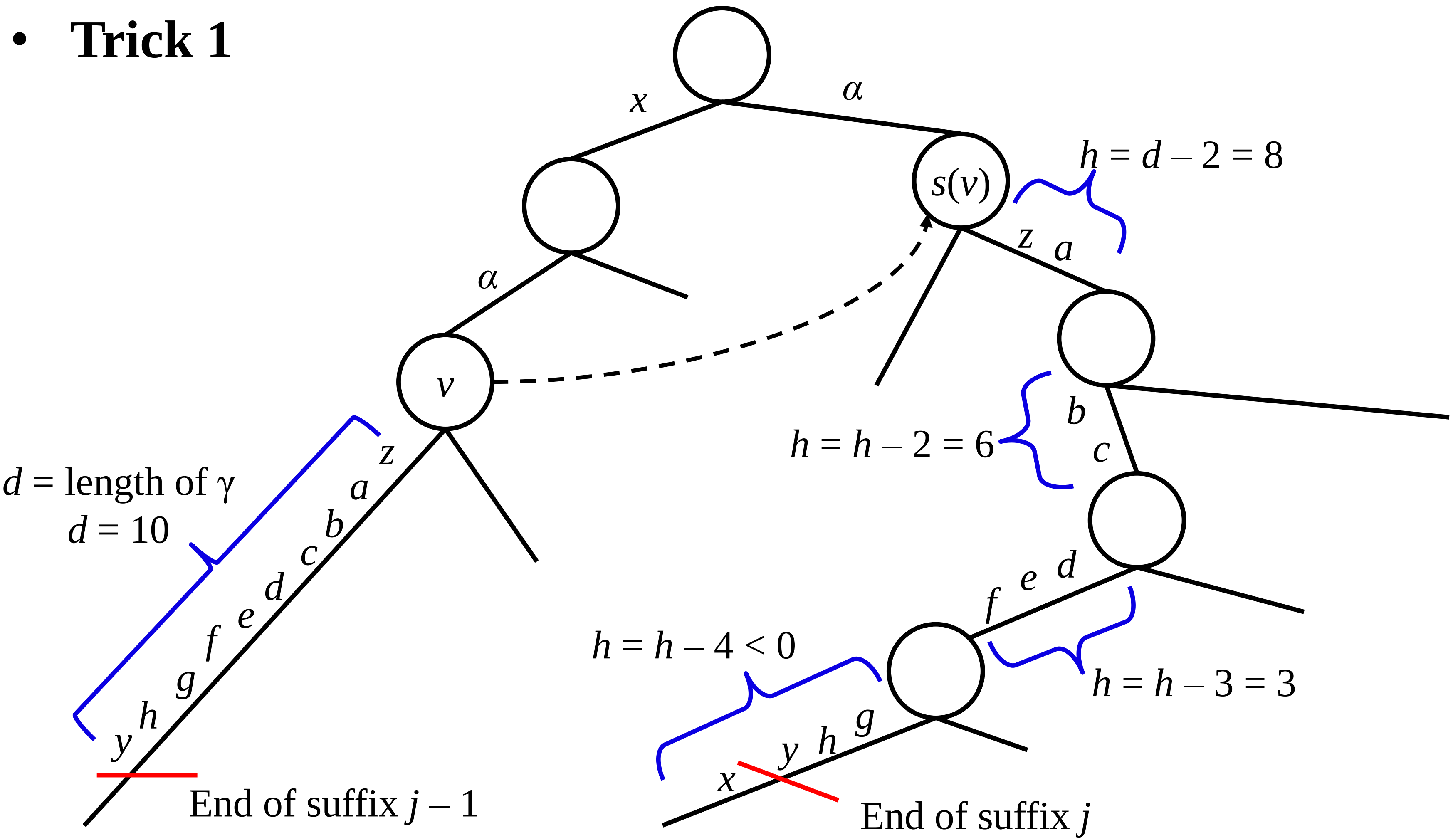
Trick number 1: skip/count trick

- **Trick 1**



Trick number 1: skip/count trick

- **Trick 1**



Trick number 1: skip/count trick

- **Trick 1**
  - The total time to traverse the path is proportional to the number of nodes
  - This is a useful heuristic, but what does it buy in terms of worst-case bound?
  - The next lemma leads immediately to the answer

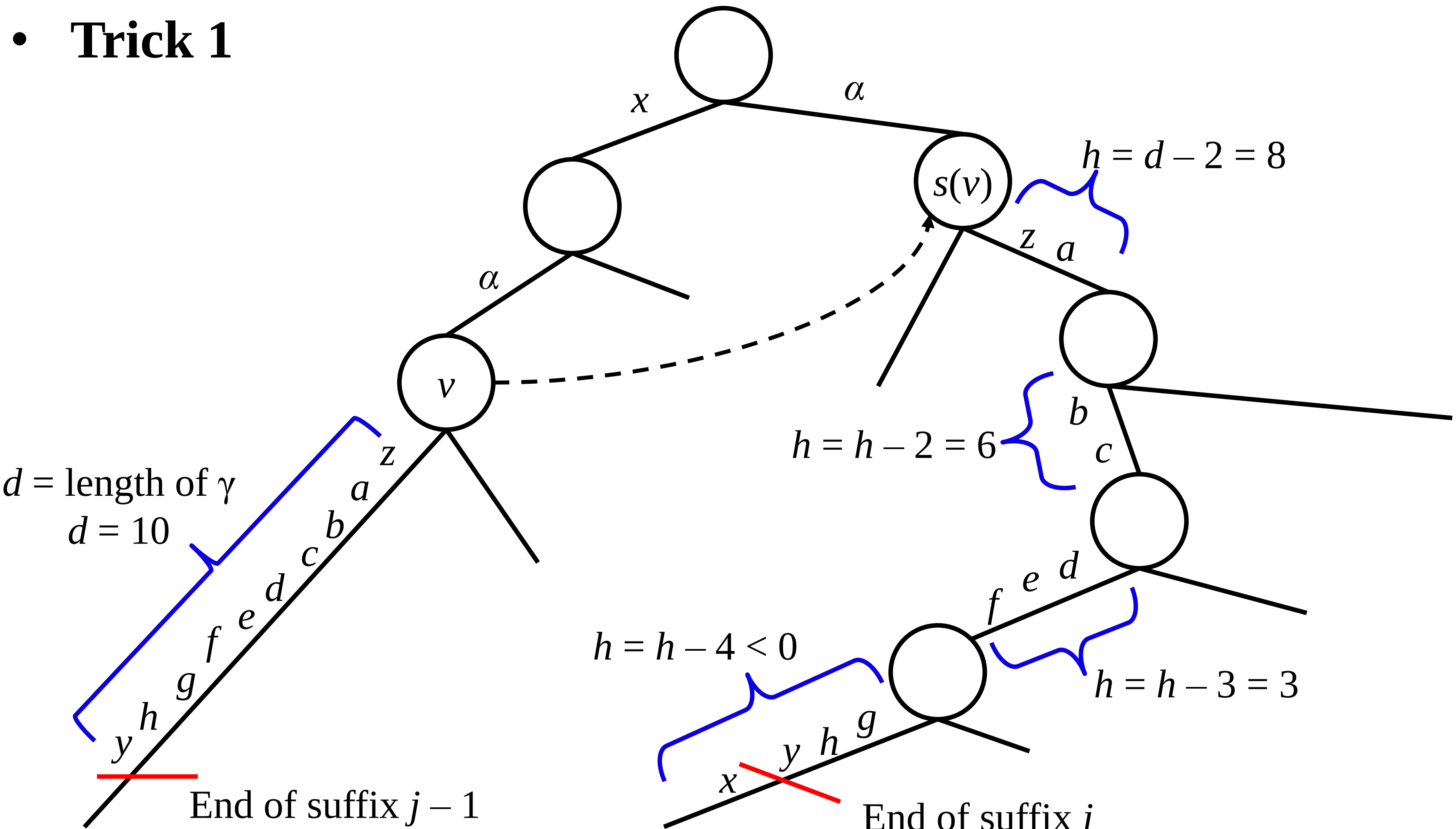
Trick number 1: skip/count trick

- **Lemma 6.1.2**
  - Let (v, s(v)) be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of v is at most one greater than the node depth of s(v)

    - The node-depth of a node u to be the number of nodes on the path from root to u

Trick number 1: skip/count trick

- **Trick 1**



Trick number 1: skip/count trick

- **Theorem 6.1.1**
  - Using the skip/count trick, any phase of Ukkonen's algorithm takes $O(m)$ time
- **Corollary 6.1.3**
  - Ukkonen's algorithm can be implemented with suffix links to run in $O(m^2)$ time

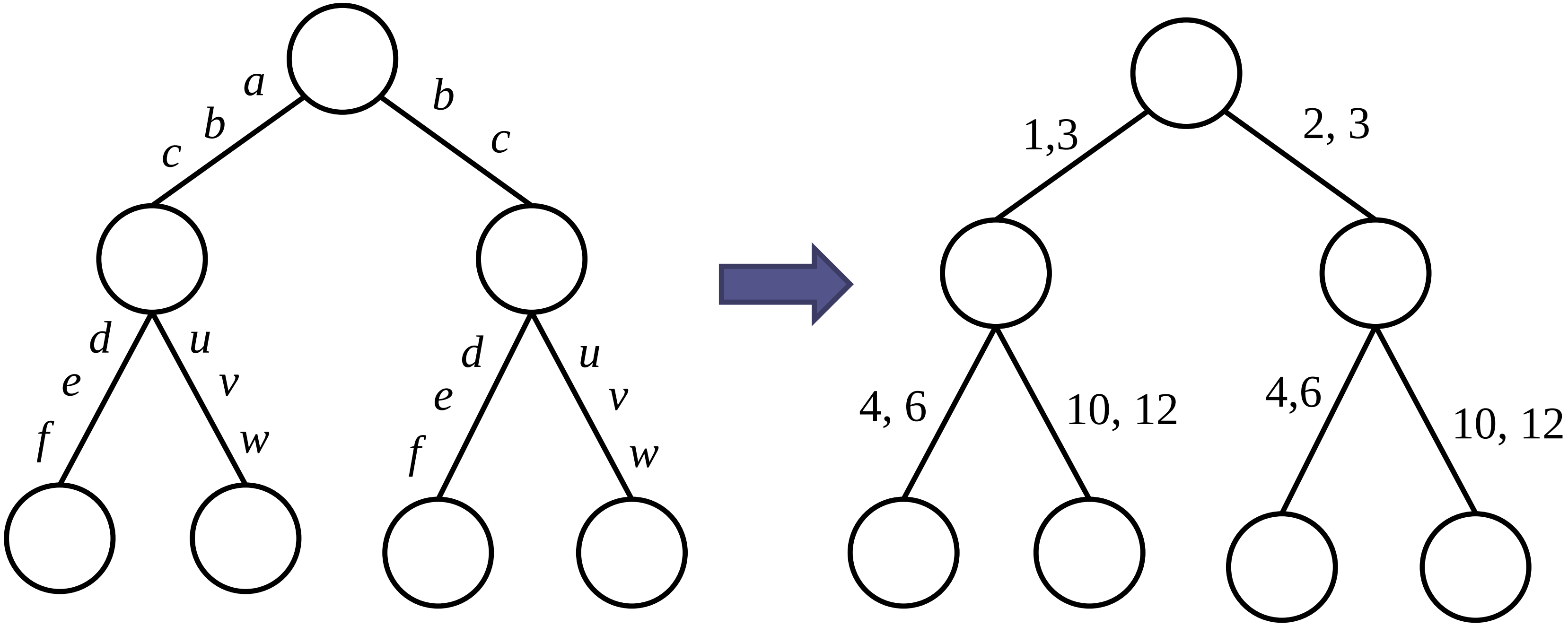
A Simple Implementation Detail

- **There is one immediate barrier to $O(m)$ time**
  - The suffix tree may require $\Theta(m^2)$ space
  - The edge-labels of a suffix tree might contain $\Theta(m)$ space since only 2 numbers are written on any edge and there are at most 2m – 1 edge.

Edge-label Compression

- **Edge-label compression**

i	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	c	d	e	f	a	b	c	u	v	w



Two More Little Tricks

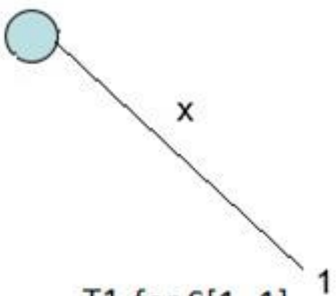
- **Two More Little Tricks**
  - Present two more implementation tricks that come from two observations
  - These tricks will lead immediately to the desired linear time bound

Two More Little Tricks

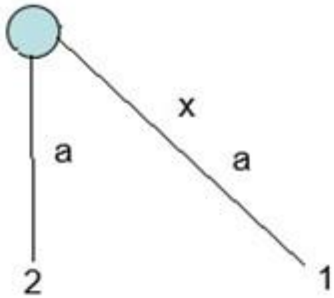
- **Observation 1: Rule 3 is show stopper**
  - In any phase, if suffix extension rule 3 applies in extension j, it will also apply in all further extensions (j+1 to i+1) until the end of the phase
  - When extension rule 3 applies, no work needs to be done since the suffix of interest is already in the tree

Observation 1: Rule 3 is show stopper

- **Example**
  - T = *xabxac*



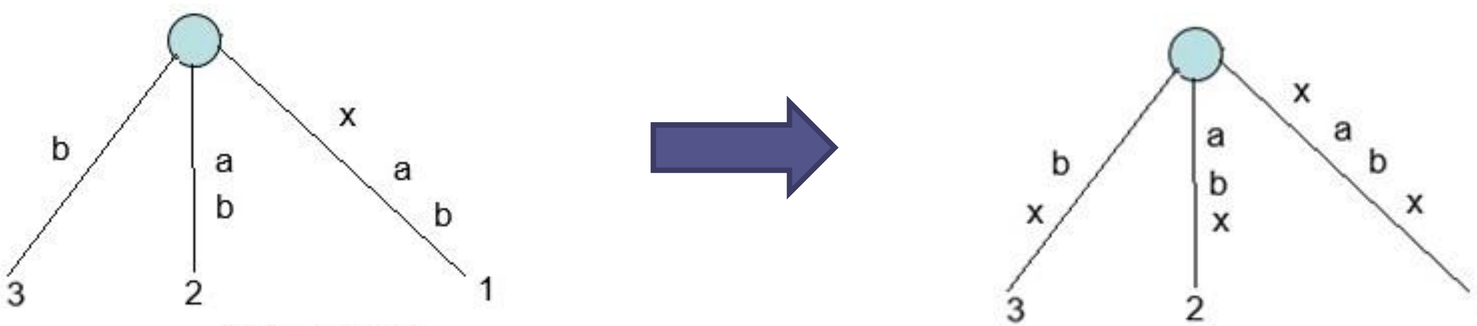
T1 for S[1..1]
Adding suffixes of x (x)
Rule 2 - A new leaf edge



T2 for S[1..2]
Adding suffixes of xa (xa and a)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - A new leaf edge

Observation 1: Rule 3 is show stopper

- **Example**
  - T = *xabxac*

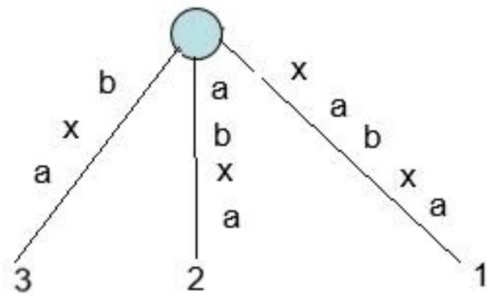


T3 for S[1..3]
Adding suffixes of xab (xab, ab and b)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - A new leaf edge

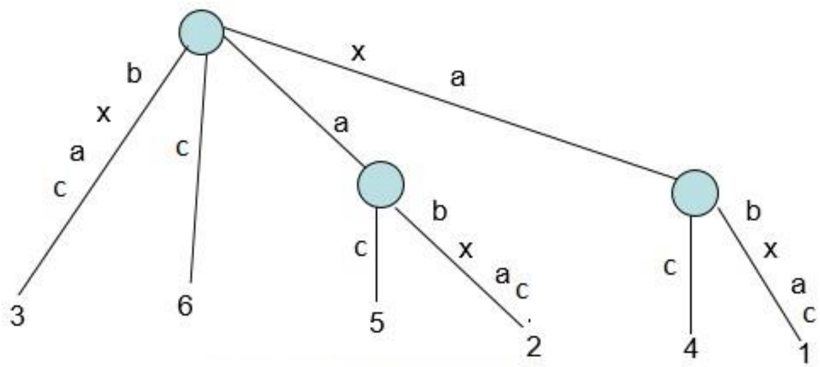
T4 for S[1..4]
Adding suffixes of xabx (xabx, abx, bx and x)
Rule 1 - Extending path label in existing leaf edge
Rule 3: Do nothing (path with label x already present)

Observation 1: Rule 3 is show stopper

- **Example**
  - T = *xabxac*



T5 for S[1..5]
Adding suffixes of xabxa (xabxa, abxa, bxa, xa and x)
Rule 1 - Extending path label in existing leaf edge
Rule 3: Do nothing (path with label xa and a already present)



T6 for S[1..6]
Adding suffixes of xabxac (xabxac, abxac, bxac, xac, ac, c)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - Three new leaf edges and two new internal nodes

Two More Little Tricks

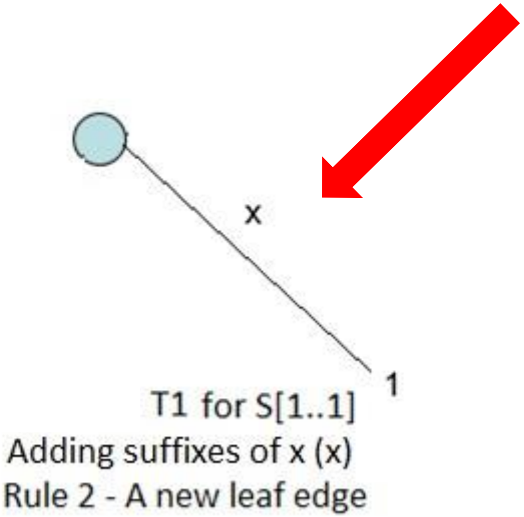
- **Trick 2**
  - End any phase i+1 the first time that extension rule 3 applies.
  - If this happens in extension j, then there is no need to explicitly find the end of any string $S[k..i]$ for $k > j$
  - The extensions in phase i+1 that are "done" after the first execution of rule 3 are said to be done *implicitly*.
  - Trick 2 is clearly a good heuristic to reduce work, but it's not clear if it leads to a better worst-case time bound.

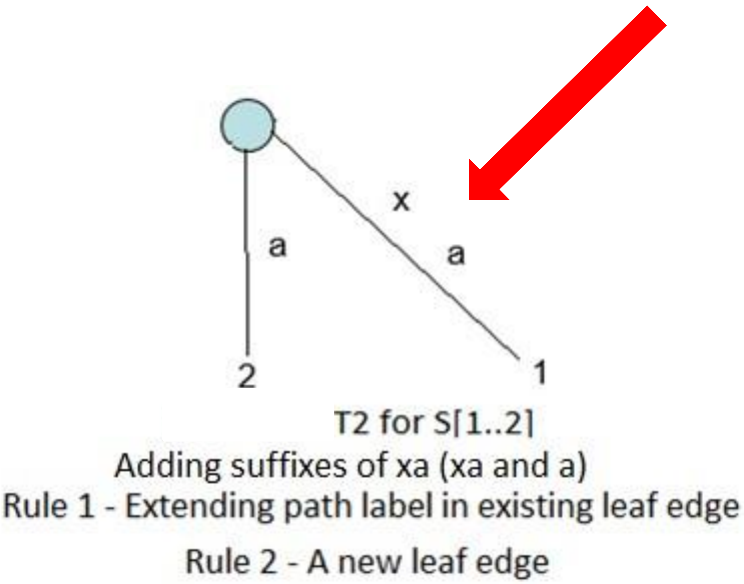
Two More Little Tricks

- **Observation 2: Once a leaf, always a leaf**
  - If at some point in Ukkonen's algorithm a leaf is created and labeled j, then that leaf will remain a leaf in all successive trees created during the algorithm
    - Because the algorithm has no mechanism for extending a leaf edge beyond its current leaf

Observation 2: Once a leaf, always a leaf

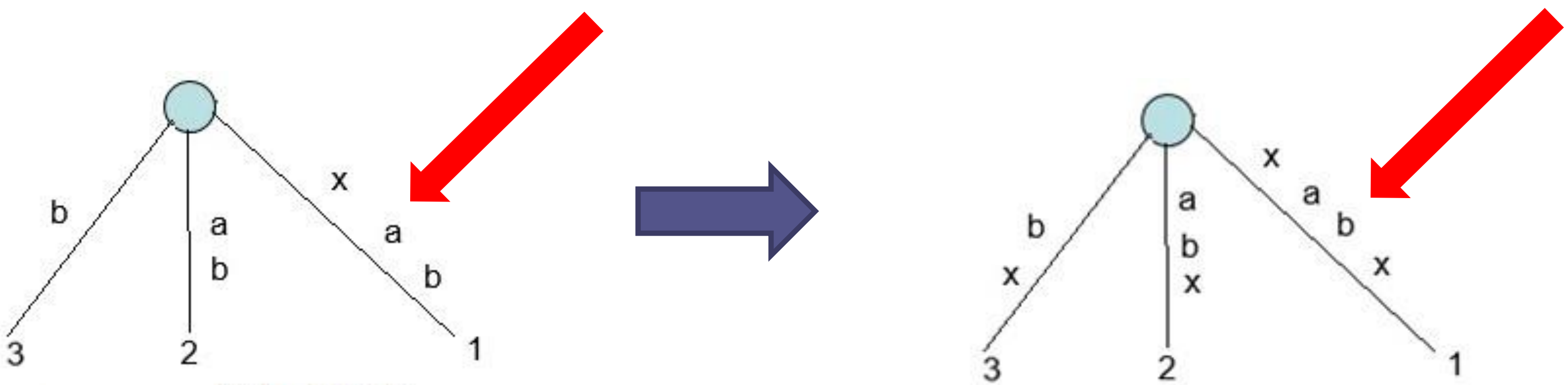
- **Example**
  - T = *xabxac*





Observation 2: Once a leaf, always a leaf

- **Example**
  - T = *xabxac*

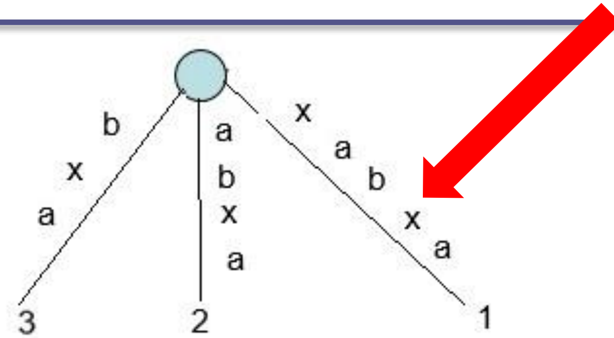


T3 for S[1..3]
Adding suffixes of xab (xab, ab and b)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - A new leaf edge

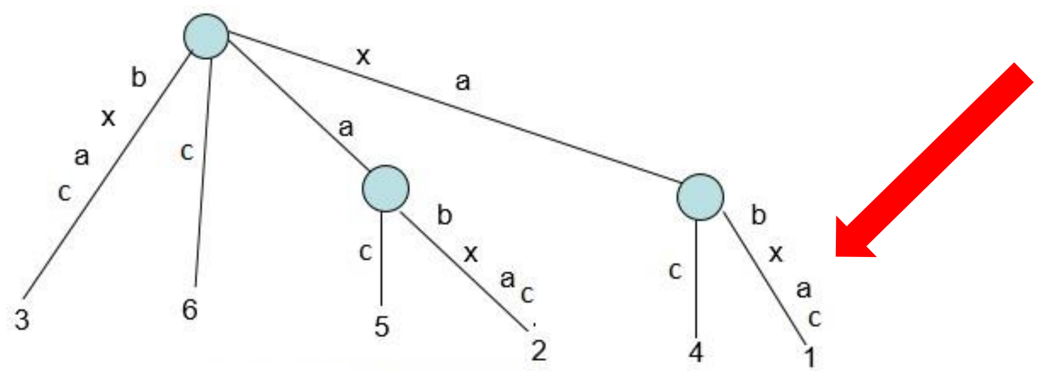
T4 for S[1..4]
Adding suffixes of xabx (xabx, abx, bx and x)
Rule 1 - Extending path label in existing leaf edge
Rule 3: Do nothing (path with label x already present)

Observation 2: Once a leaf, always a leaf

- **Example**
  - T = *xabxac*



T5 for S[1..5]
Adding suffixes of xabxa (xabxa, abxa, bxa, xa and x)
Rule 1 - Extending path label in existing leaf edge
Rule 3: Do nothing (path with label xa and a already present)



T6 for S[1..6]
Adding suffixes of xabxac (xabxac, abxac, bxac, xac, ac, c)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - Three new leaf edges and two new internal nodes

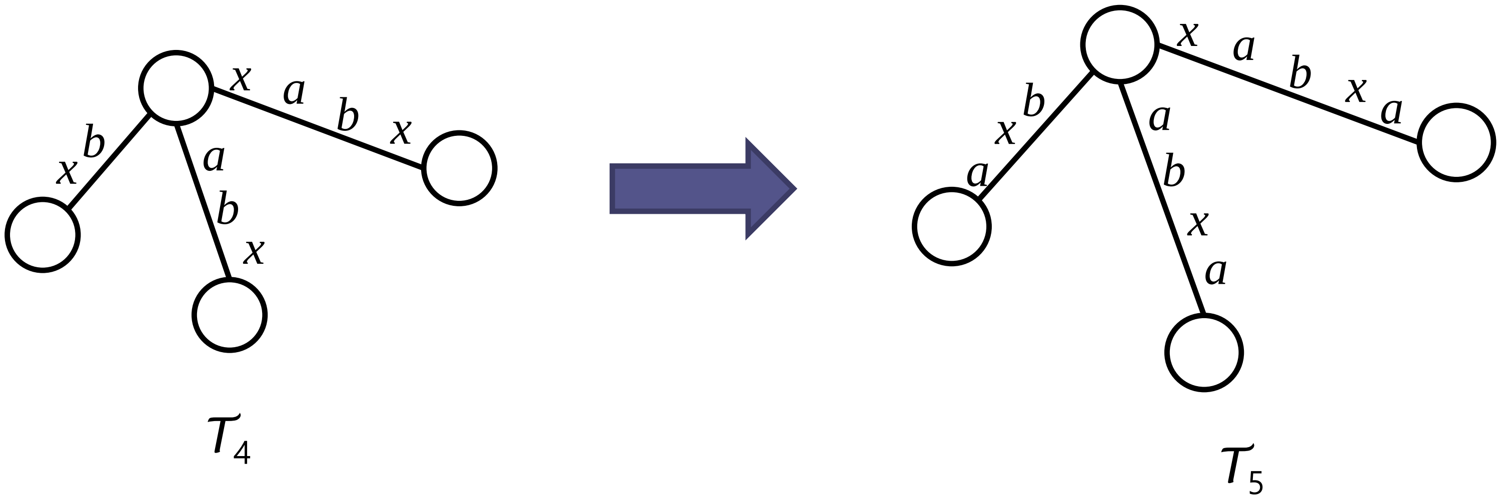
Two More Little Tricks

- **Trick 3**
  - In any phase i, leaf edges may look like (p, i), (q, i), (r, i), …. where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r, i+1),…. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number.
  - So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done.

Observation 2: Once a leaf, always a leaf

- **Example – Trick 3**
  - T = *xabxac*

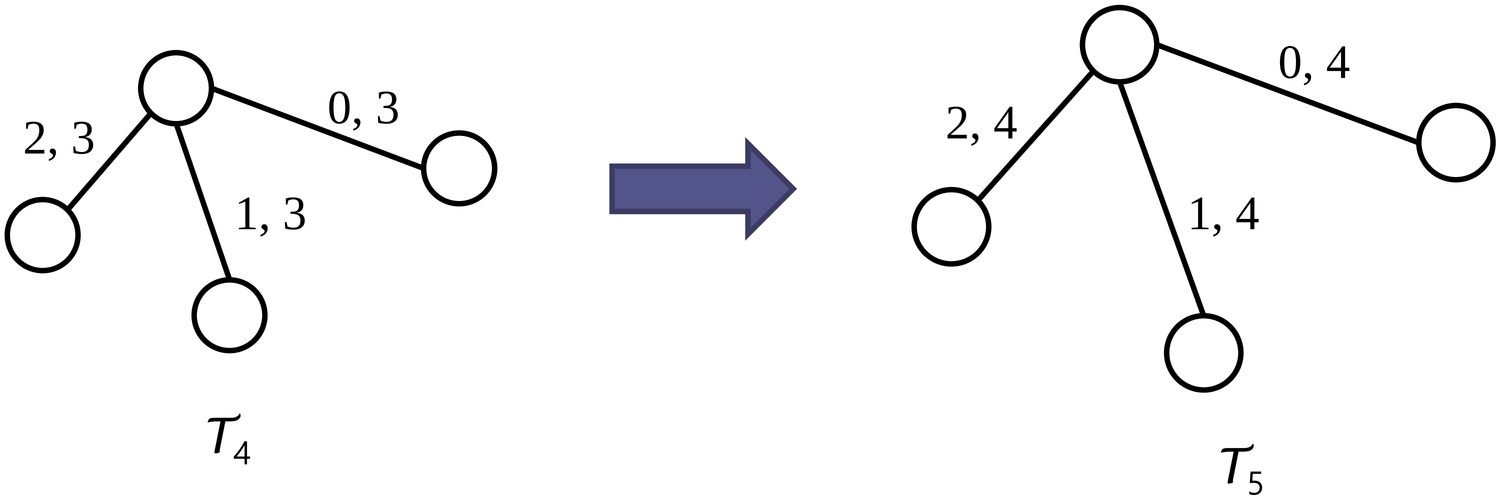
i	1	2	3	4	5	6
S	x	a	b	x	a	c



Observation 2: Once a leaf, always a leaf

- **Example – Trick 3**
  - T = *xabxac*

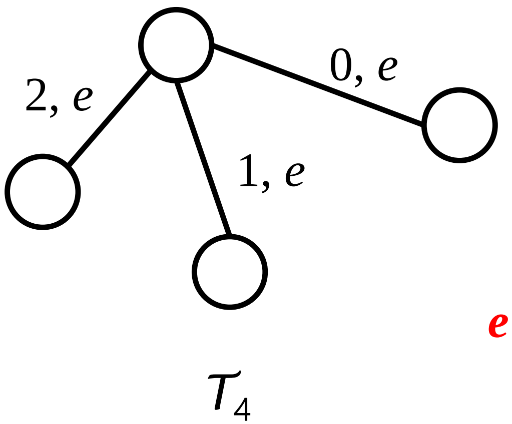
i	1	2	3	4	5	6
S	x	a	b	x	a	c



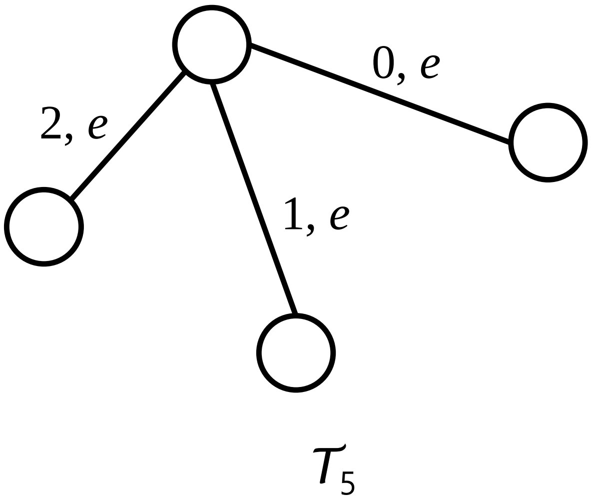
Observation 2: Once a leaf, always a leaf

- **Example – Trick 3**
  - T = *xabxac*

i	1	2	3	4	5	6
S	x	a	b	x	a	c

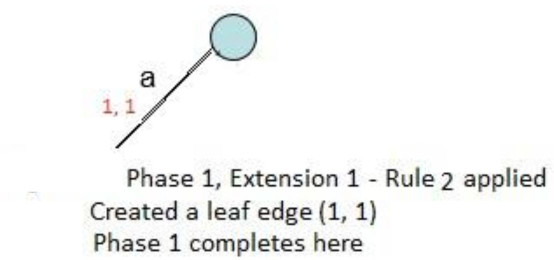


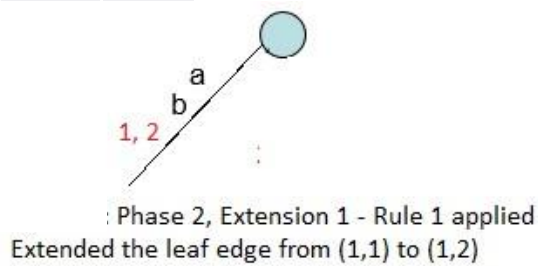
$e = 3 \rightarrow e = e + 1 = 4$

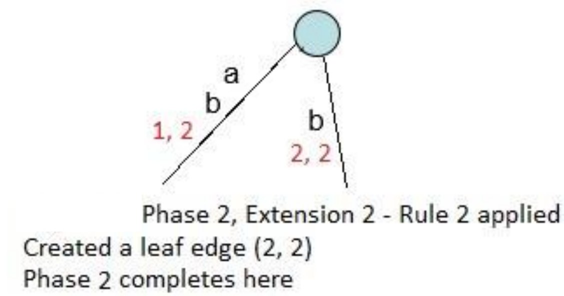


Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$







Set *e* to 1

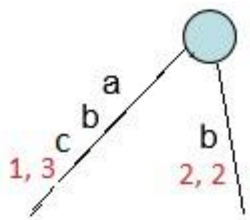
Set *e* to 2. This will do extensions 1. Trick 3

{a}

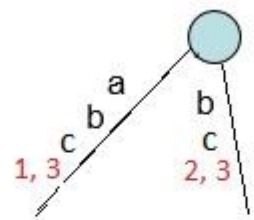
{ab, a}

Ukkonen's Algorithm

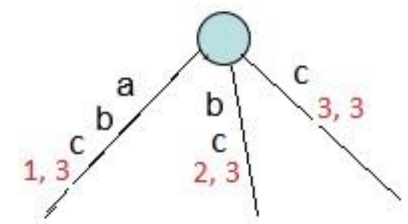
i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Phase 3, Extension 1 - Rule 1 applied
Extended the leaf edge from (1,2) to (1,3)



Phase 3, Extension 2 - Rule 1 applied
Extended the leaf edge from (2,2) to (2,3)



Phase 3, Extension 3 - Rule 2 applie
Created a leaf edge (3,3)
Phase 3 completes here

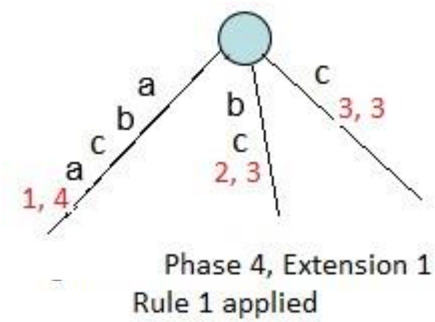
Set *e* to 3. This will do extensions 1 and 2. Trick 3

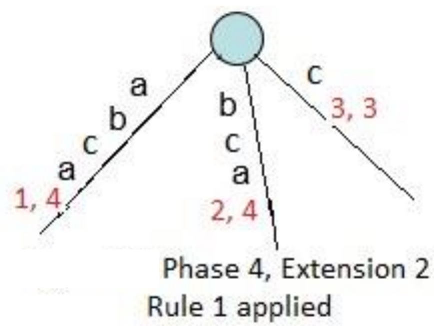
{abc, bc, c}

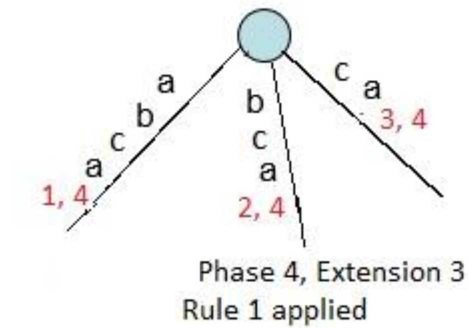
Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

Edge 'a' is present. Stop here as rule 3





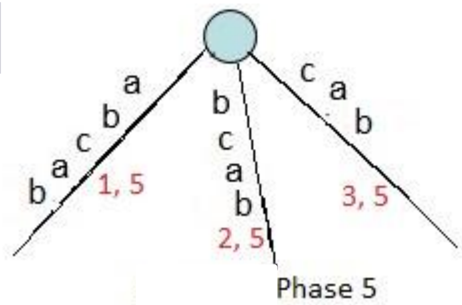


Set *e* to 4. This will do extensions 1, 2 and 3. Trick 3

{abca, bca, ca, a}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Set *e* to 5. This will do extensions 1, 2 and 3. Trick 3

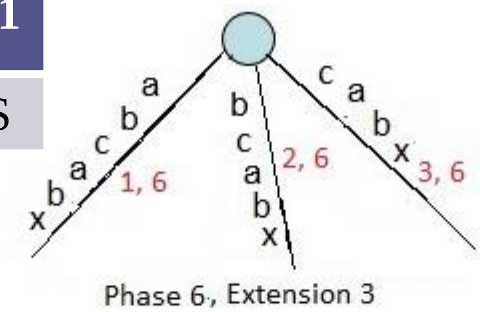
{abcab, bcab, cab, ab, b}

Using trick 1 start from root, stop at b, edge 'ab' is present. Stop here as rule 3. Trick 2.

{ab, b} are in tree implicitly

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

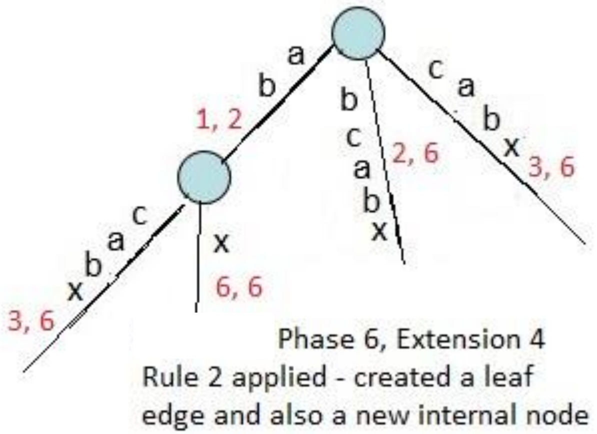


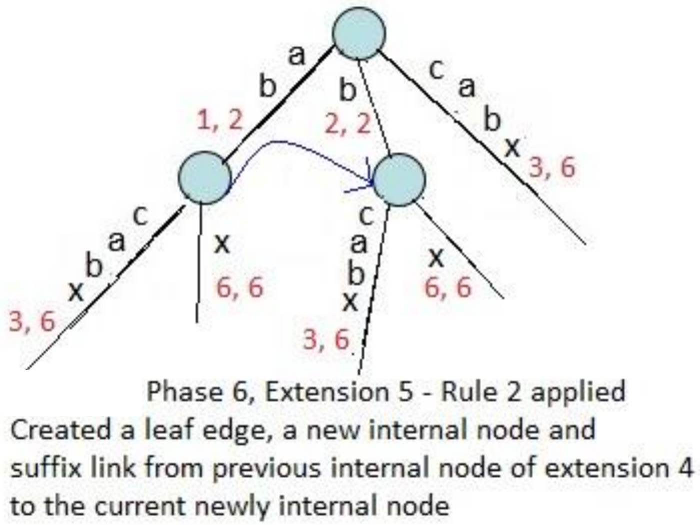
Set *e* to 6. This will do extensions 1, 2 and 3. Trick 3

{abcabx, bcabx, cabx, abx, bx, x}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$





{abx}

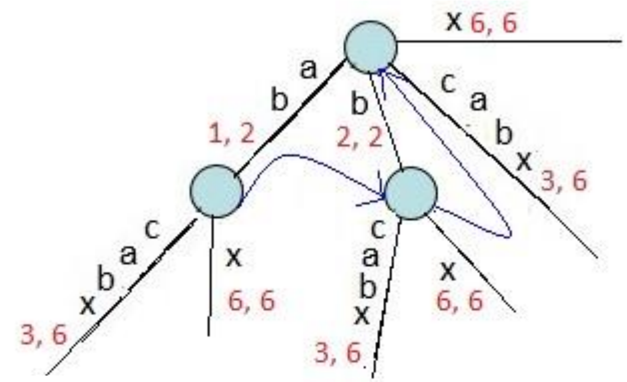
Using trick 1 start from root, stop at b (length of abx < length of edge) and create new leaf edge x.

{bx}

Using trick 1 start from root, stop at b and create new leaf edge x. Suffix link is also created from previous internal node (of extension 4) to the new internal nod e created in current extension 5.

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



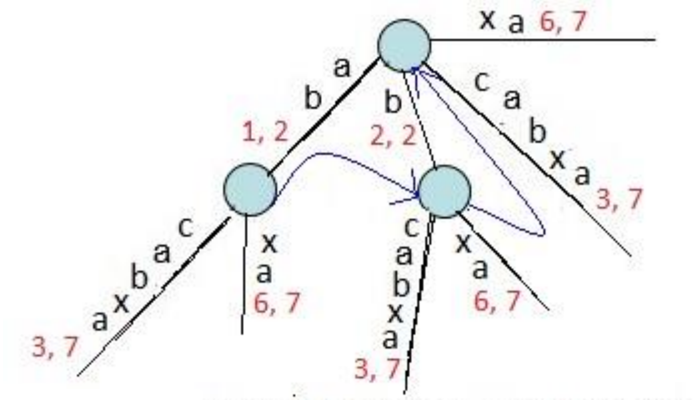
Phase 6, Extension 6 - Rule 2 applied
Created a leaf edge and suffix link from previous internal node of extension 5 to root node (as no new internal node created in extension 6, so suffix link goes to root)

{x}

Suffix link is also created from previous internal node (of extension 5) to the root node created in current extension 6.

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Phase 7, Extension 6 - Rule 1 applied

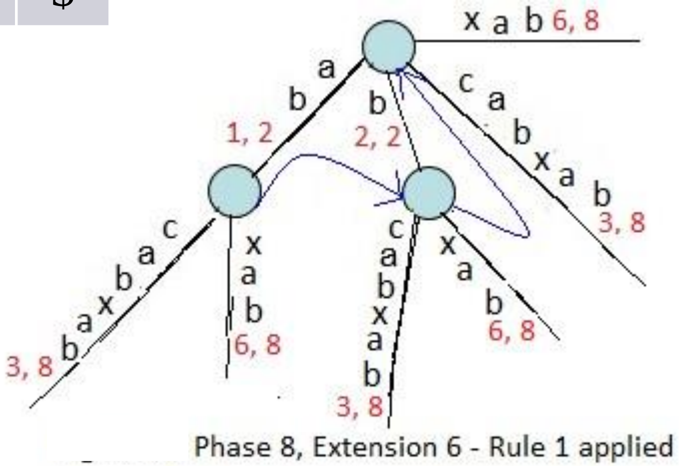
Set *e* to 7. This will do extensions 1, 2, 3, 4, 5 and 6. Trick 3.

{abcabxa, bcabxa, cabxa, abxa, bxa, xa, a}

{a} are in tree implicitly

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



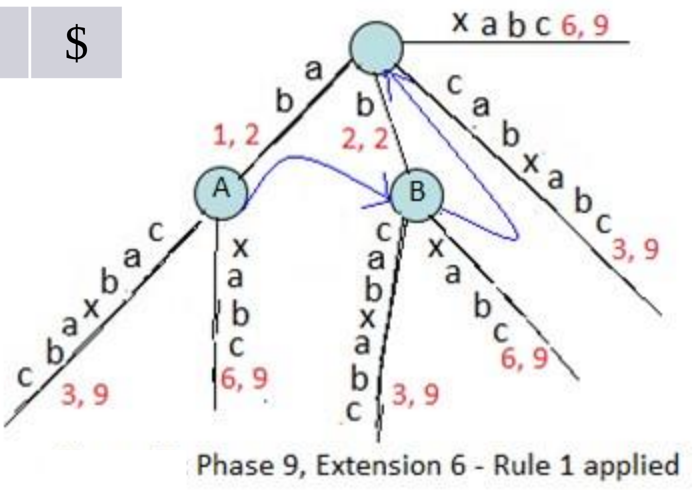
Set *e* to 8. This will do extensions 1, 2, 3, 4, 5 and 6. Trick 1

{abcabxab, bcabxab, cabxab, abxab, bxab, xab, ab, b}

Using trick 1 start from root, stop at b.
Edge 'ab' is present. Stop here as rule 3, trick 2.

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Set *e* to 9. This will do extensions 1, 2, 3, 4, 5 and 6. Trick 3

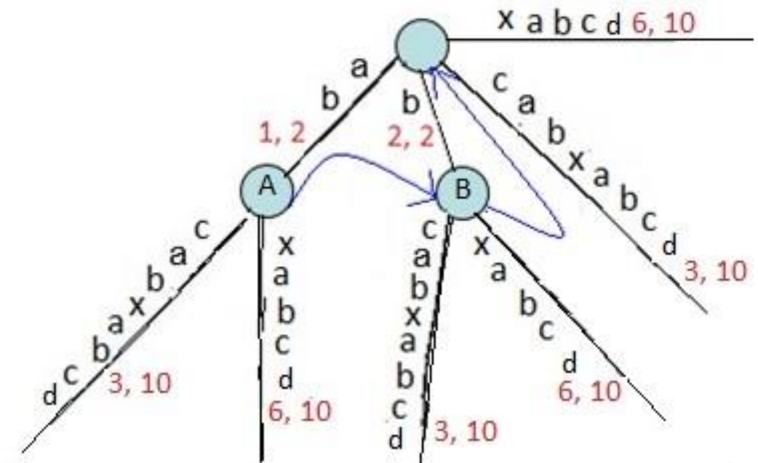
{abcabxabc, bcabxabc, cabxabc, abxabc, bxabc, xabc, abc, bc, c}

Using trick 1 start from root, skip to A, stop at c.
Edge 'abc' is present. Stop here as rule 3, trick 2.

{abc, bc, c} are in tree implicitly

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

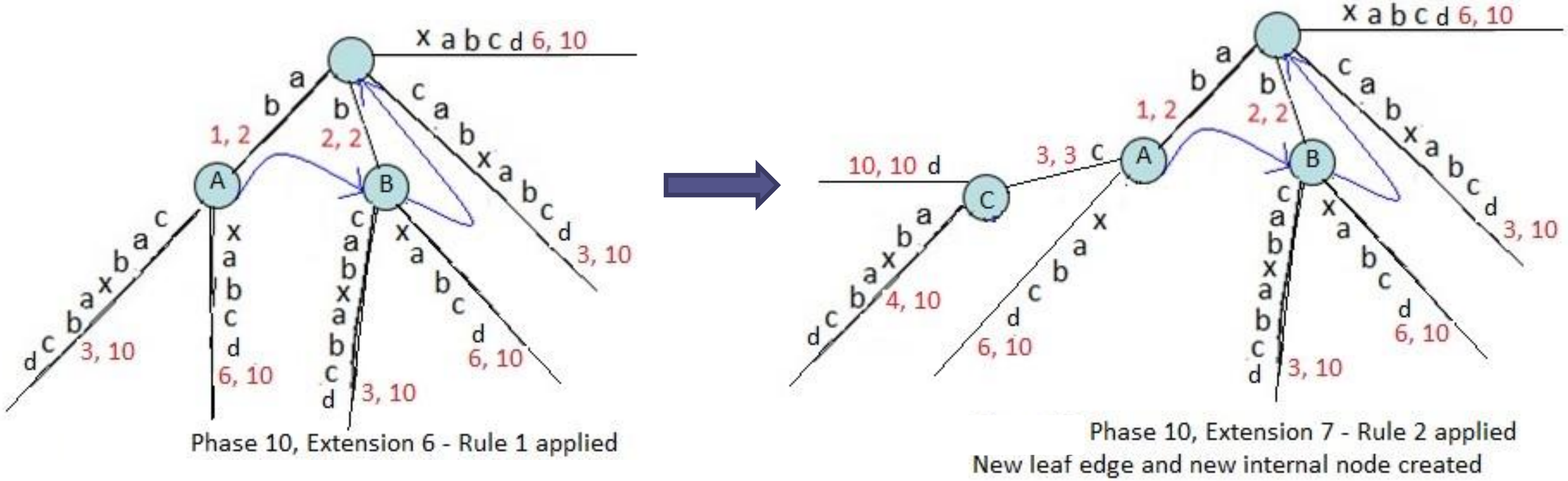


Set *e* to 10. This will do extensions 1, 2, 3, 4, 5 and 6. Trick 3

{abcabxabcd, bcabxabcd, cabxabcd, abxabcd, bxabcd, xabcd, abcd, bcd, cd, d}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



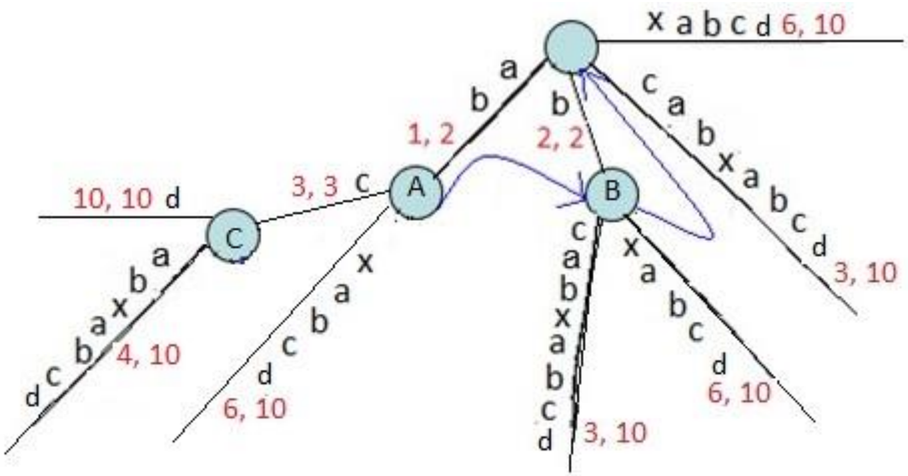
Using trick 1 start from root, skip to A, stop at c.
Create new leaf edge 'd'.

{abcd}

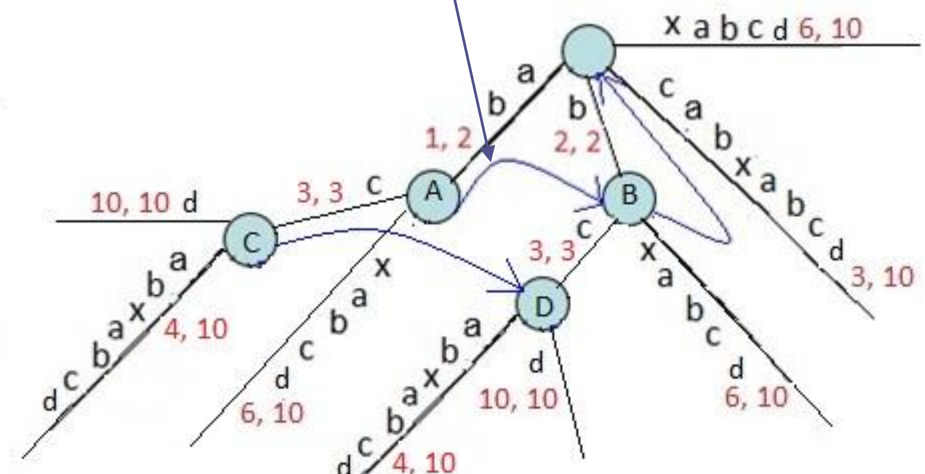
Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

Follow suffix link. Trick 1



Phase 10, Extension 7 - Rule 2 applied
New leaf edge and new internal node created



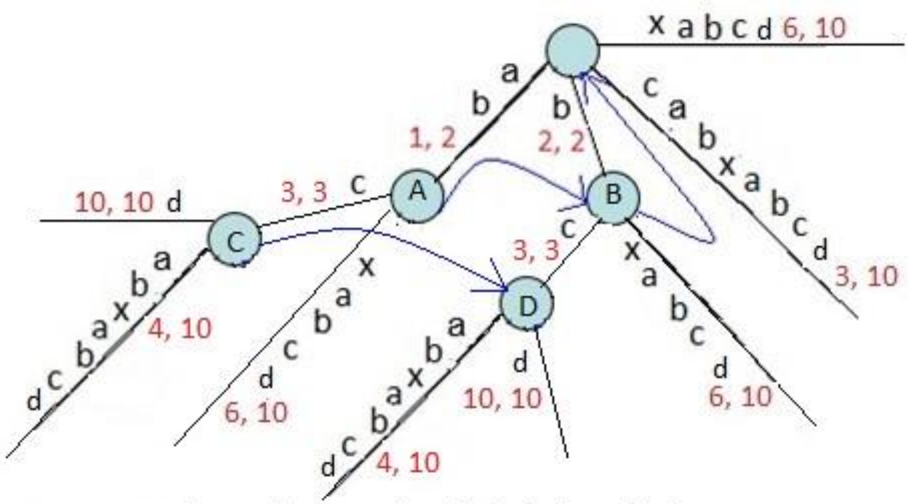
Phase 10, Extension 8 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node C created in previous extension 7, pointing to the internal node D via siffix link

{abcd} {bcd}

Ukkonen's Algorithm

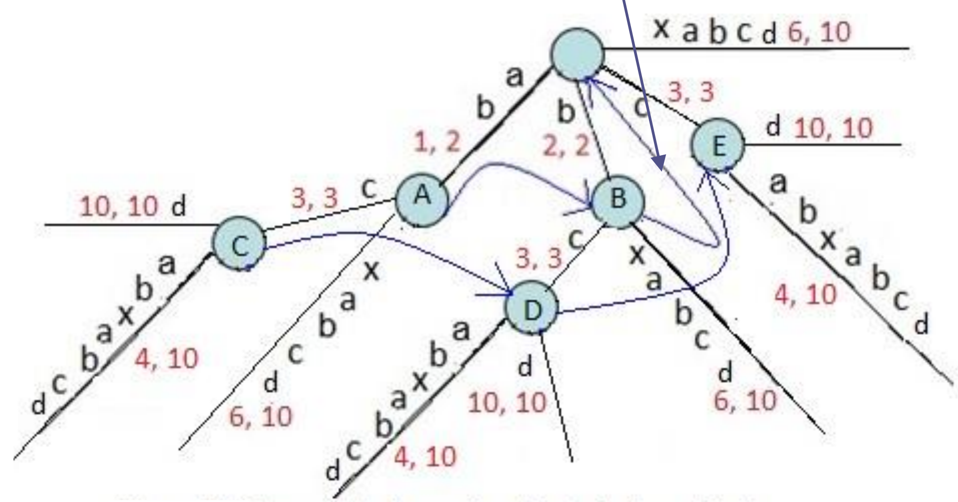
i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

Follow suffix link. Trick 1



Phase 10, Extension 8 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node C created in previous extension 7, pointing to the internal node D via siffix link

{bcd}

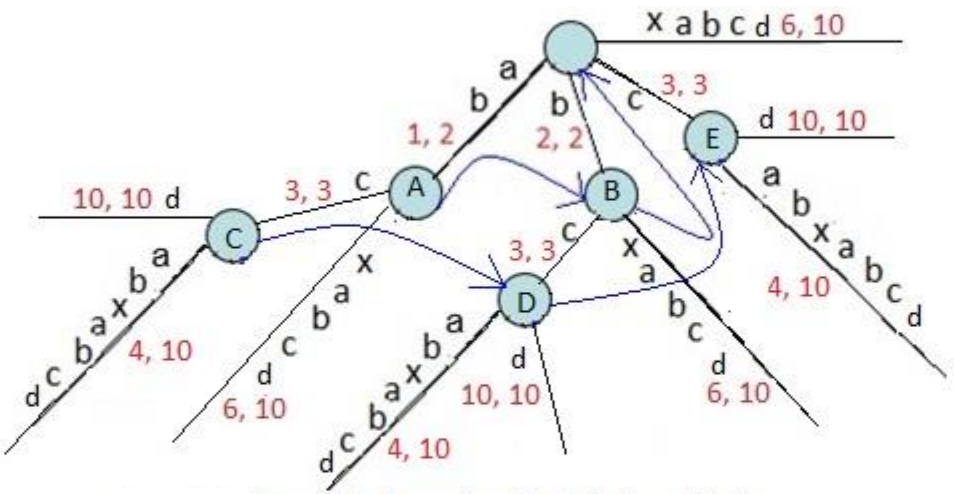


Phase 10, Extension 9 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node D created in previous extension 8, pointing to the internal node E via suffix link

{cd}

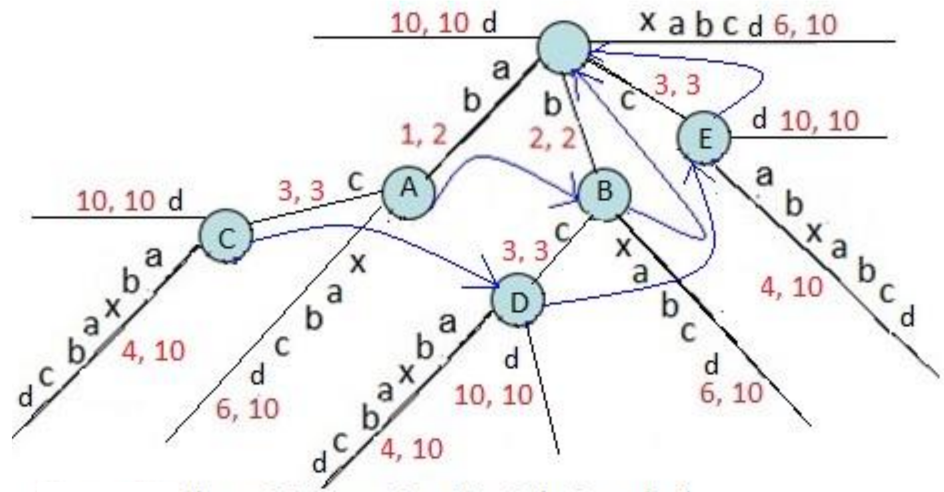
Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Phase 10, Extension 9 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node D created in previous extension 8, pointing to the internal node E via suffix link

{cd}

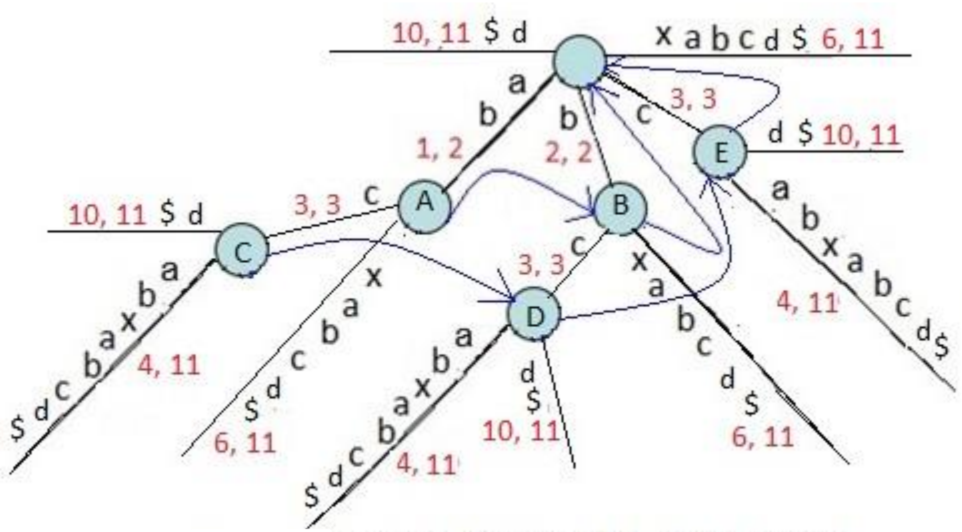


Phase 10, Extension 10 - Rule 2 applied
New leaf edge created. Also the internal node E created in previous extension 9, pointing to root node via suffix link

{d}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



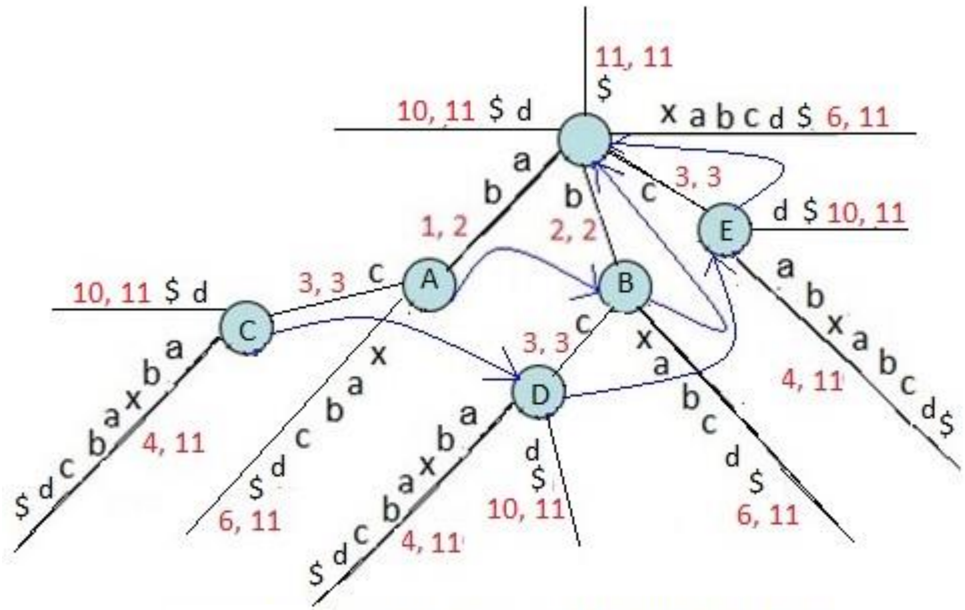
Phase 11, Extension 10 - Rule 1 applied

Set *e* to 11. This will do extensions 1 to 10. Trick 3

{abcabxabcd$, bcabxabcd$, cabxabcd$, abxabcd$, bxabcd$, xabcd$, abcd$, bcd$, cd$, d$, $}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$

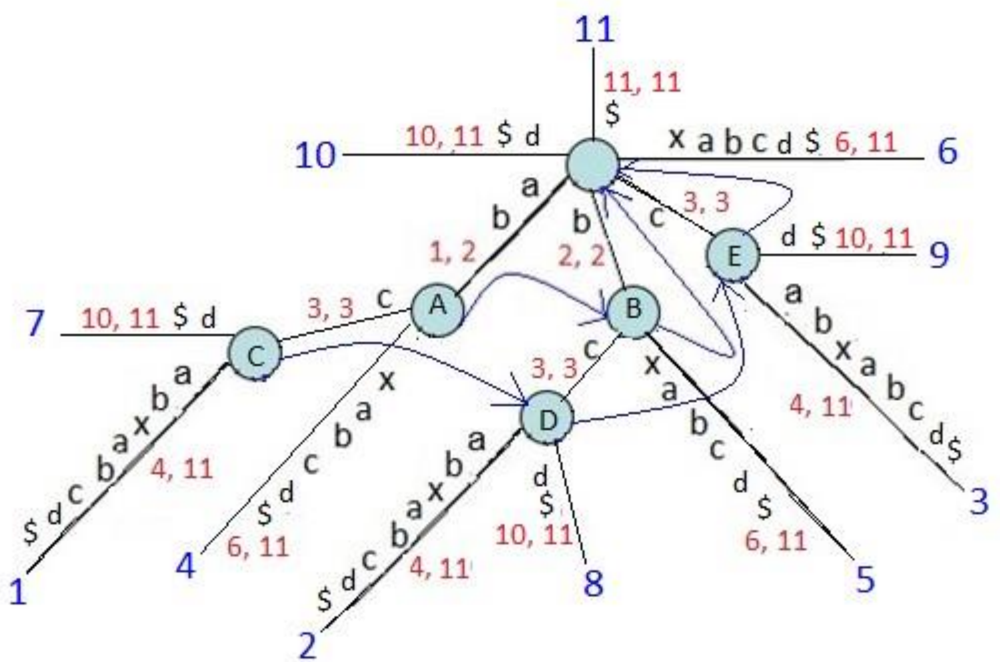


Phase 11, Etension 11 - Rule 2 applied

{$}

Ukkonen's Algorithm

i	1	2	3	4	5	6	7	8	9	10	11
S	a	b	c	a	b	x	a	b	c	d	$



Final Suffix Tree for String S = abcabxabcd$

Ukkonen's Algorithm

- **Theorem 6.1.2**
  - Using suffix links and implementation tricks 1, 2, and 3, Ukkonen's algorithm builds implicit suffix trees $\mathcal{T}_1$ through $\mathcal{T}_m$ in $O(m)$ total time.