Weiner’s linear-time suffix tree algorithm

2015. 05. 26

BIS Lab
Kyungjung Song

Weiner's algorithm

- **Weiner's algorithm**
  - starts with the entire string *S* (unlike Ukkonen's algorithm)
  - enters one suffix at a time into a growing tree(like Ukkonen's algorithm)
    - although in a very different order

- It first enters string $S(n)\$$
- $S[n\text{-}1\ldots n]\$$
- $S[n\text{-}2\ldots n]\$$
- …
- $S[1\ldots n]\$$
  - = Entire string

Ex) *T* = xabxac

- xabxac$
- xabxac$
- xabxac$
- …
- xabxac$
  - = Entire string

Weiner's algorithm

- **Definition**
  - Suff_i: suffix $S[i..n]$
    - Suff_n: the single character $S(n)$
    - Suff_1: the entire string S

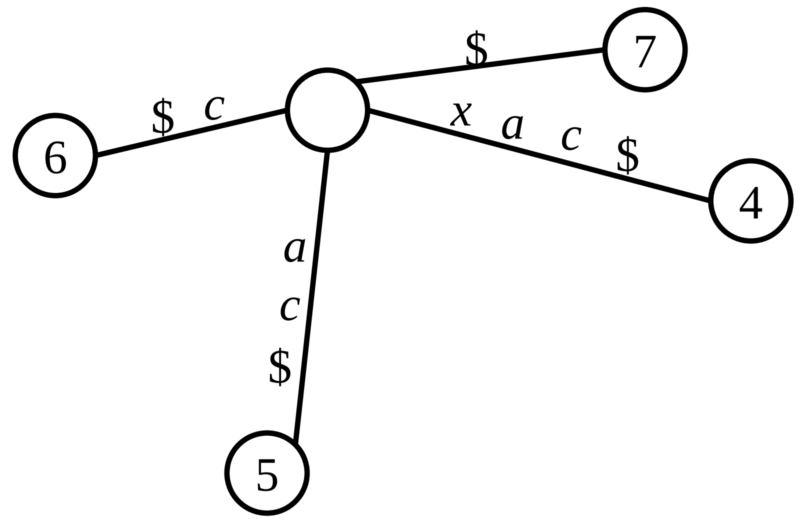
Ex) T = xabxac

- Suff_6: c
- Suff_4: xac
- Suff_1: xabxac

Weiner's algorithm

- **Definition**
  - $\mathcal{T}_i$: a suffix tree of string $S[i..n]\$$
    - n-i+2 leaves numbered i through n+1
      - The path from root to any leaf $j(i \le j \le n+1)$ has label $\mathrm{Suff}_j\$$

Ex) T = xabxac

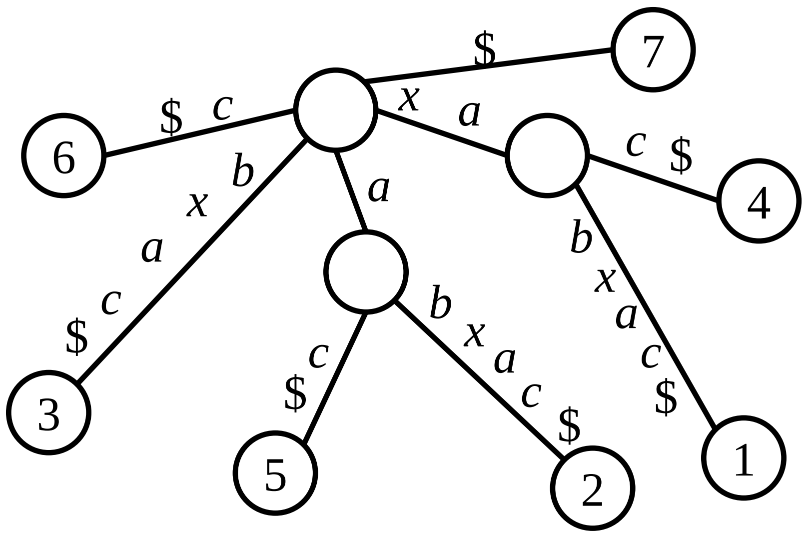


$\mathcal{T}_4$

Weiner's algorithm

- **Definition**
  - $\mathcal{T}_i$: a suffix tree of string $S[i..n]\$$
    - n-i+2 leaves numbered i through n+1
      - The path from root to any leaf $j(i \le j \le n+1)$ has label $\text{Suff}_j\$$

Ex) T = xabxac



$\mathcal{T}_1$

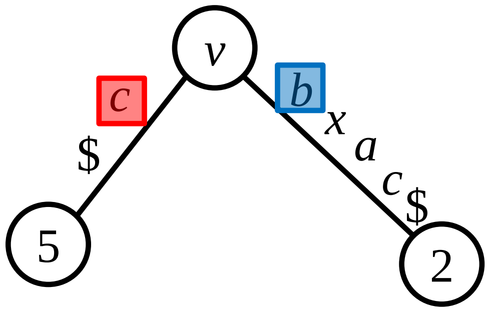
Weiner's algorithm

- **Weiner's algorithm constructs trees**
  - From $\mathcal{T}_{n+1}$ down to $\mathcal{T}_1$

  - First, we will implement the method in a straightforward inefficient way
  - Then we will speed up the straightforward construction
    - to obtain Weiner's linear-time algorithm

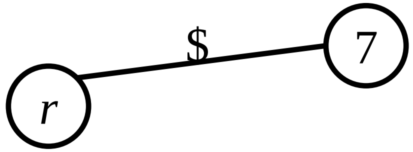
A straightforward construction

- **The idea of the method**
  - Essentially the same as the idea for constructing keyword trees (Section 3.4)
  - Construct each tree $\mathcal{T}_i$
    - from $\mathcal{T}_{i+1}$ and character $S(i)$
    - for each i from n down to 1
  - For any node v in $\mathcal{T}_i$, no two edges out of v have edge-labels beginning with the same character



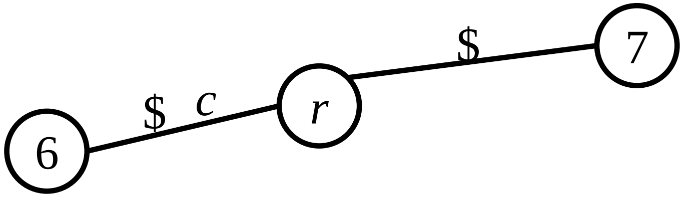
A straightforward construction

- **Ex)** ***T*** **= xabxac**
  - Start at $i = n+1 = 7$



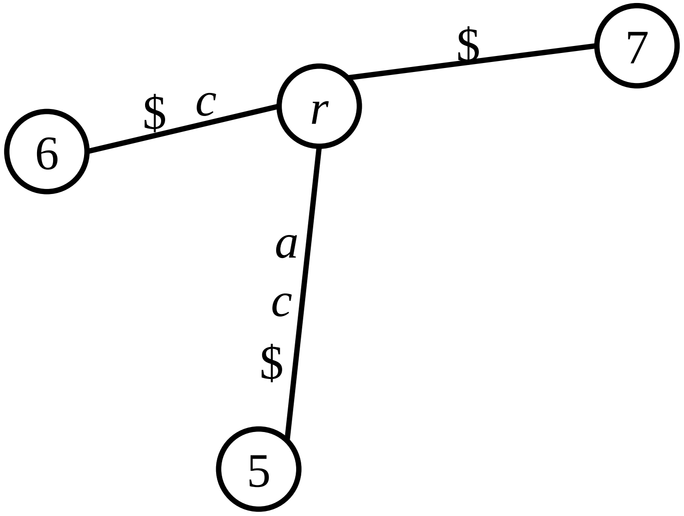
A straightforward construction

- **Ex) *T* = xabxac**
  - $i = 6$



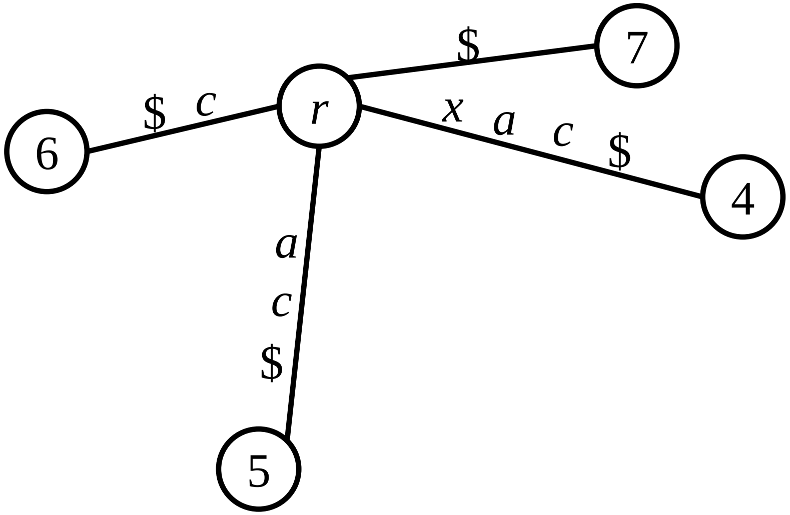
A straightforward construction

- **Ex)** ***T*** **= xabxac**
  - $i = 5$



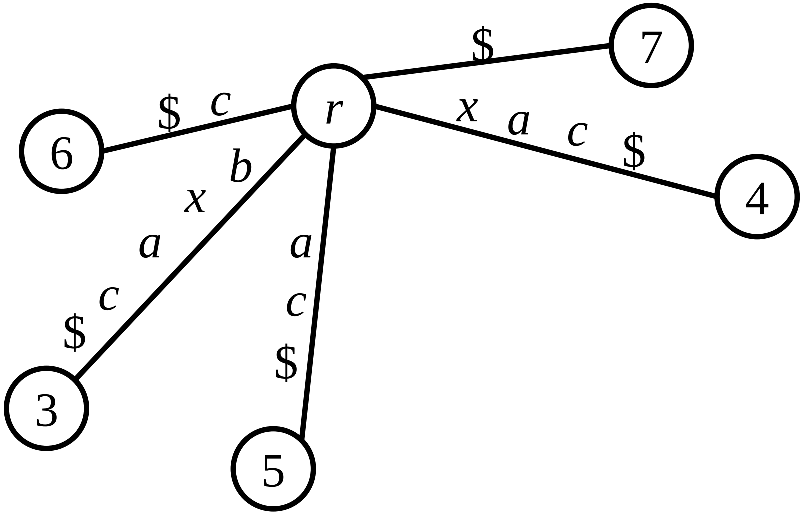
A straightforward construction

- **Ex) *T* = xabxac**
  - $i = 4$



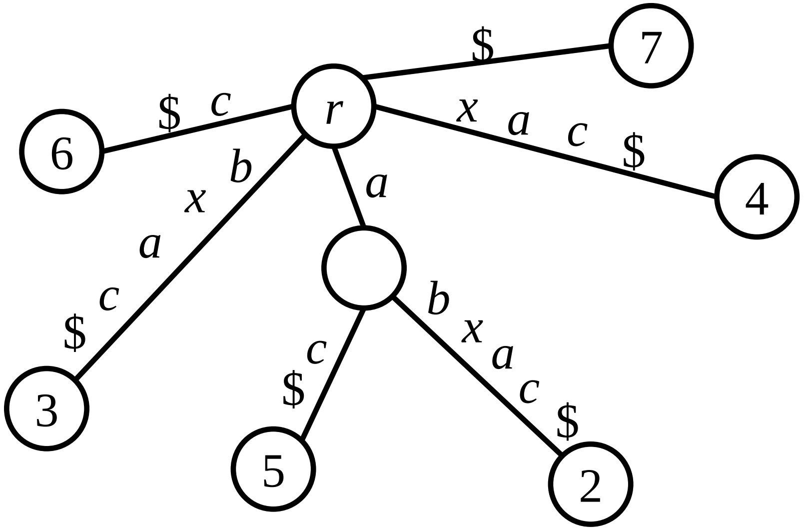
A straightforward construction

- **Ex)** ***T*** **= xa****bxac**
  - $i = 3$



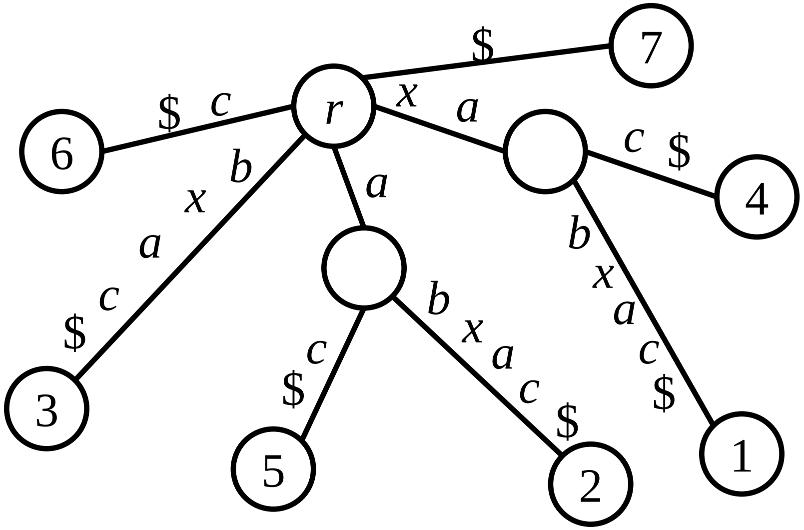
A straightforward construction

- **Ex)** ***T*** **= xabxac**
  - $i = 2$



A straightforward construction

- **Ex)** ***T*** **=** **xabxac**
  - $i = 1$



A straightforward construction

- **Definition**
  - *Head*(i): the longest prefix of $S[i..n]$ that matches
    a substring of $S[i+1..n]\$$

Ex) T = xabxac

- *Head*(2): the longest prefix of ‘**a**bxac’ that matches
  a substring of ‘bx**a**c$’
- *Head*(2) = a

A straightforward construction

- **Definition**
  - *Head*(i): the longest prefix of $S[i..n]$ that matches
    a substring of $S[i+1..n]\$$

Ex) T = xabxac

- *Head*(1): the longest prefix of '**xa**bxac' that matches
  a substring of 'ab**xa**c$'
- *Head*(1) = xa

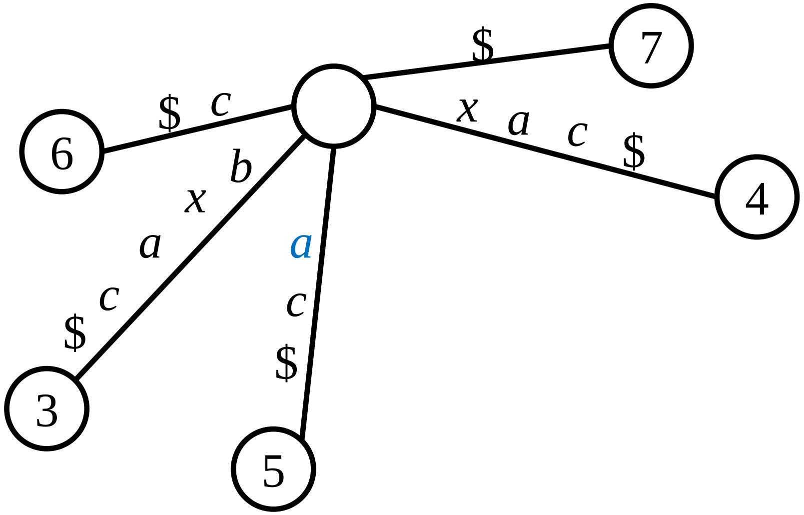
A straightforward construction

- **Naïve weiner algorithm**

A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$

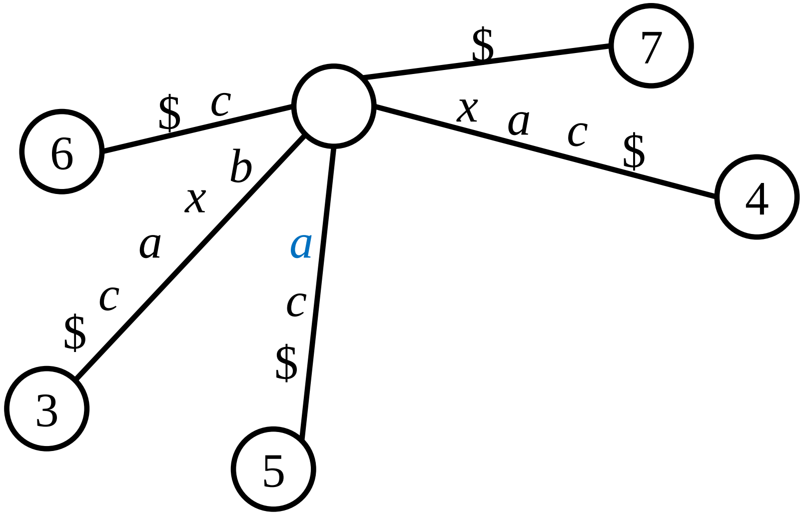
Ex) T = xabxac, i=2
Suff$_2$\$ = abxac\$, *Head*(2) = a



A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node

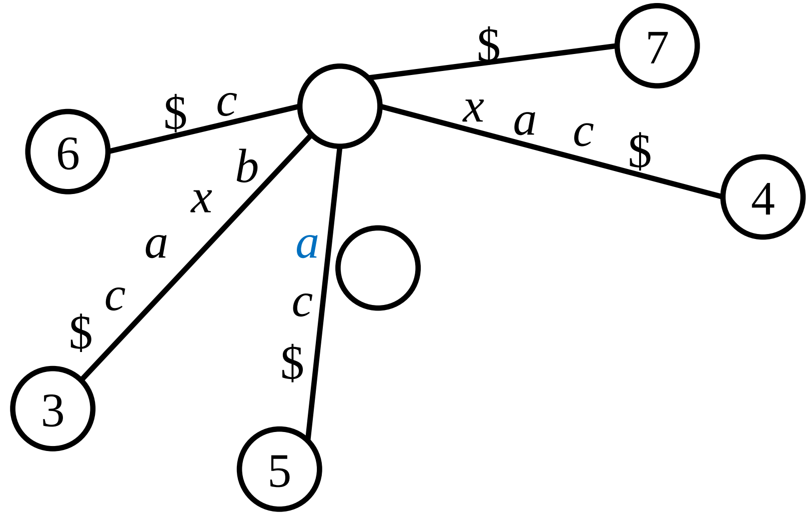
Ex) T = xabxac, i=2
Suff$_2$\$ = abxac\$, *Head*(2) = a



A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node

Ex) T = xabxac, i=2
Suff$_2$\$ = abxac\$, *Head*(2) = a

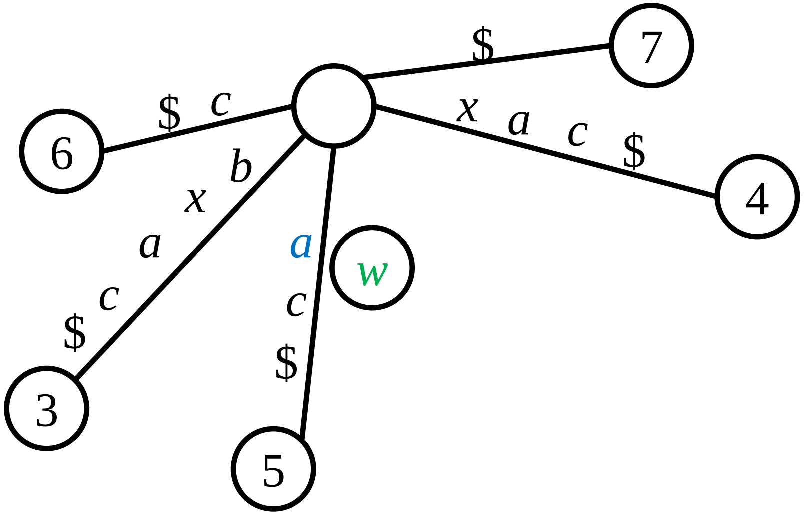


A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node
  3. Let w denote the node at the end of *Head*(i)
     - Created or not

Ex) T = xabxac, i=2

Suff$_2$\$ = abxac\$, *Head*(2) = a

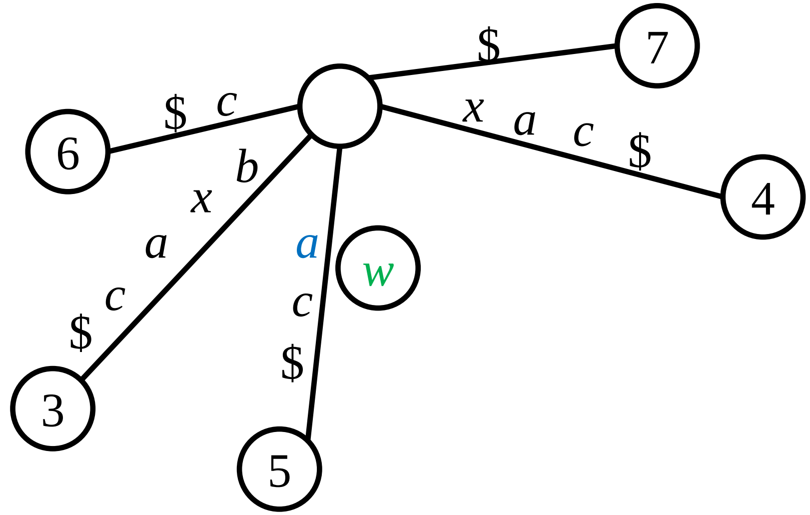


A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node
  3. Let w denote the node at the end of *Head*(i)
     - Created or not
  4. Splitting an existing edge and its existing edge-label
     - So that w has node-label *Head*(i)

Ex) T = xabxac, i=2

Suff$_2$\$ = abxac\$, *Head*(2) = a

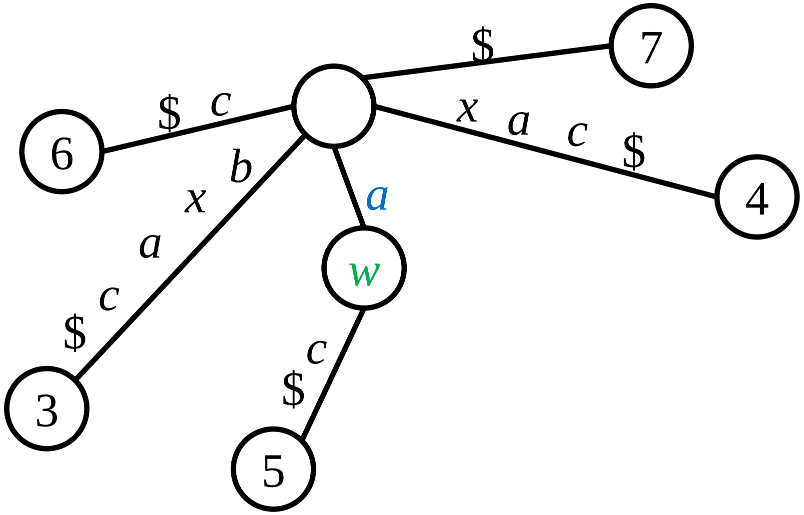


A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node
  3. Let w denote the node at the end of *Head*(i)
     - Created or not
  4. Splitting an existing edge and its existing edge-label
     - So that w has node-label *Head*(i)

Ex) T = xabxac, i=2

Suff$_2$\$ = abxac\$, *Head*(2) = a



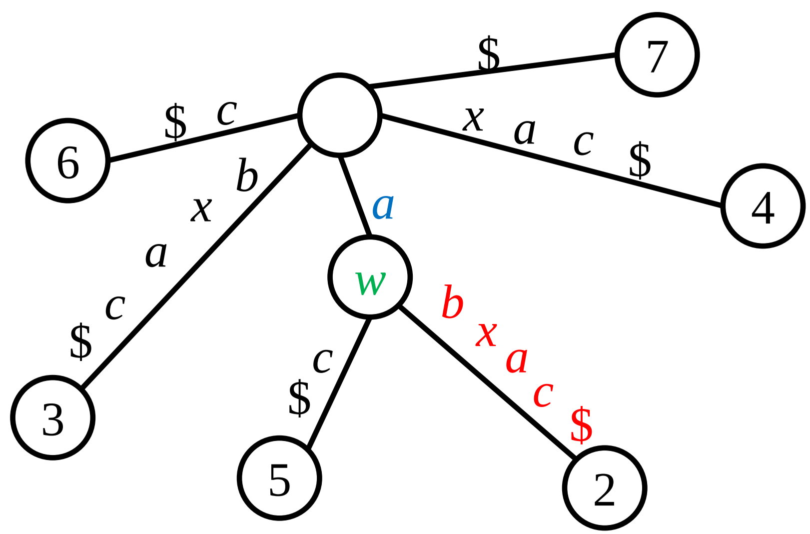
A straightforward construction

- **Naïve weiner algorithm**
  1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$
  2. If there is no node at the end of *Head*(i)
     - Create a node
  3. Let w denote the node at the end of *Head*(i)
     - Created or not
  4. Splitting an existing edge and its existing edge-label
     - So that w has node-label *Head*(i)
  5. Create a new leaf numbered i and a new edge (w,i) labeled with the remaining characters of $\text{Suff}_i\$$

$O(1)$

Ex) T = xabxac, i=2

$\text{Suff}_2\$$ = abxac$, *Head*(2) = a



A straightforward construction

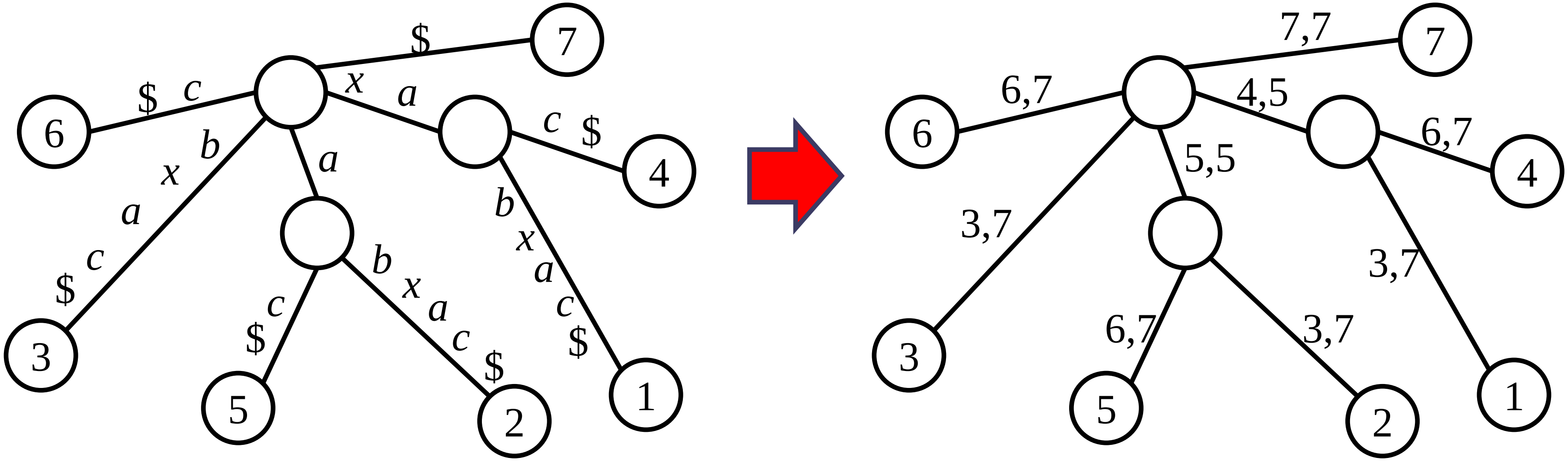
- **The final suffix tree $\mathcal{T} = \mathcal{T}_1$**
  - Constructed in $O(n^2)$ time
  - The difficult part of the algorithm is **finding *Head*(*i*)**
  - So, to speed up the algorithm
    - Need a more efficient way to find $Head(i)$

Toward a more efficient implementation

- **Edge-labeling**
    - As in the discussion of Ukkonen’s algorithm
        - If edge-labels are explicitly written on the tree
        - a linear time bound is not possible

Toward a more efficient implementation

- **Each edge-label is represented by two indices**
  - Indicating the start and end positions of the labeling substring



T = xabxac$

Finding *Head*(*i*) efficiently

- **The key to Weiner's algorithm**
  - Two vectors kept at each nonleaf node (including the root)
    1. Indicator vector *I*
       - A bit vector(0 or 1)
    2. Link vector *L*
       - The reverse of the suffix link in Ukkonen's algorithm

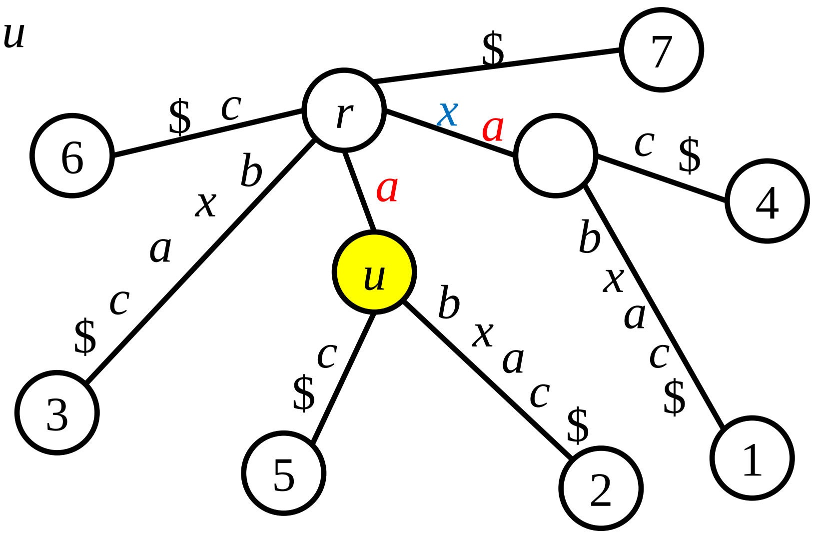
  - Length of vector: the size of the alphabet
  - Indexed by the characters of the alphabet

Ex) node *v*

	a	…	*x*	y	*z*
I	0	…	1	1	1
L	null	…	*v*''	null	*w*

Finding *Head*(*i*) efficiently

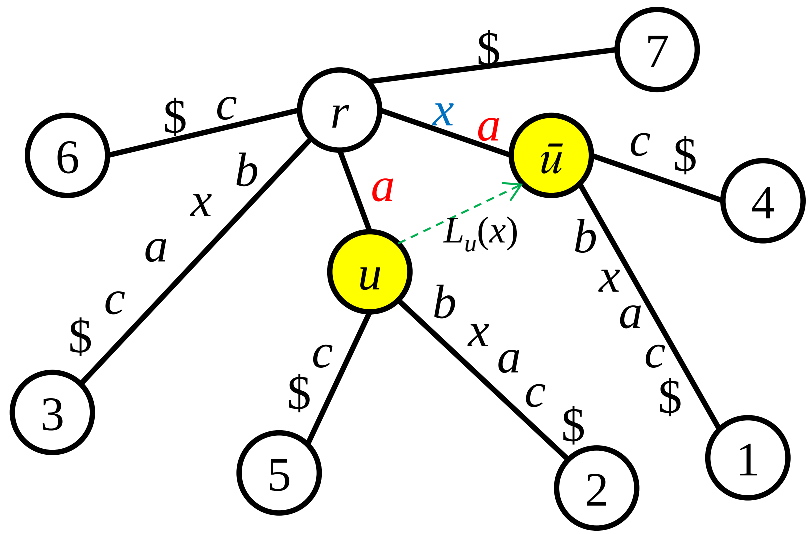
- $I_u(x) = 1$
  - if and only if there is a path from the root labeled $x\alpha$
    - where α is the path-label of node u



	a	…	*x*	y	z
I	0	…	1	0	0

Finding *Head*(i) efficiently

- **$L_u(x)$ points to (internal) node $\bar{u}$**
  - if and only if $\bar{u}$ has path-label $x\alpha$
    - where u has path-label α
  - Otherwise $L_u(x)$ = null

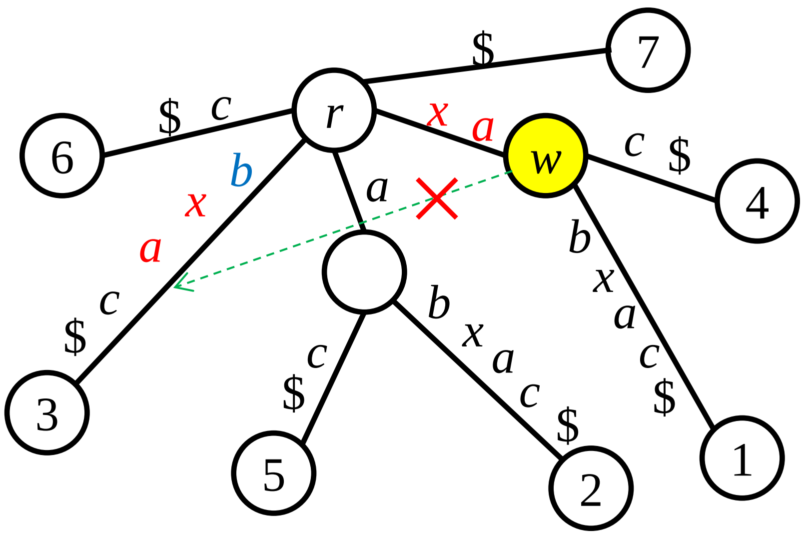


	a	…	*x*	y	*z*
L	null	…	$\bar{u}$	null	null

Finding *Head*(*i*) efficiently

- **$L_u(x)$ is nonnull only if $I_u(x)$ =1**
  - But the converse is not true

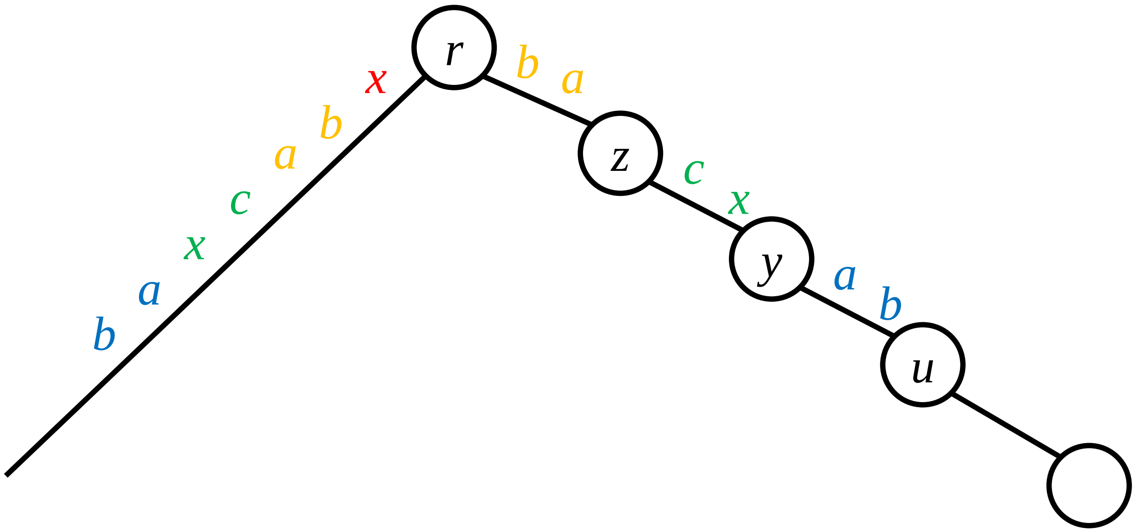
- T = xabxac
  - $I_w(b) = 1$
  - $L_w(b) =$ null



	a	b	…	y	z
I	0	1	…	0	0
L	null	null	…	null	null

Finding *Head*(*i*) efficiently

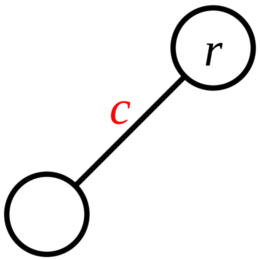
- **If $I_u(x) = 1$ then $I_v(x) = 1$**
  - v: every ancestor node of u



Finding *Head*(*i*) efficiently

- **The root *r*, only one nonleaf node**
  - $I_r(S(n)) = 1$, $I_r(x) = 0$ for every other character x
  - $L_r(x)$ = null, for every character x

  - Ex) T = xabxac



	a	b	c	…	z
I	0	0	1	…	0
L	null	null	null	…	null

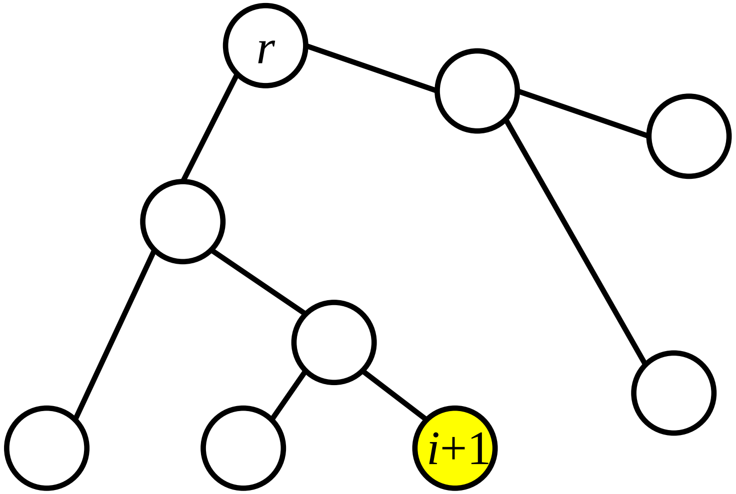
  - The algorithm will maintain the vectors as the tree changes

The basic idea of Weiner's algorithm

- **Using indicator and link vectors**
  - to find *Head*(i)
  - to construct $\mathcal{T}_i$ more efficiently

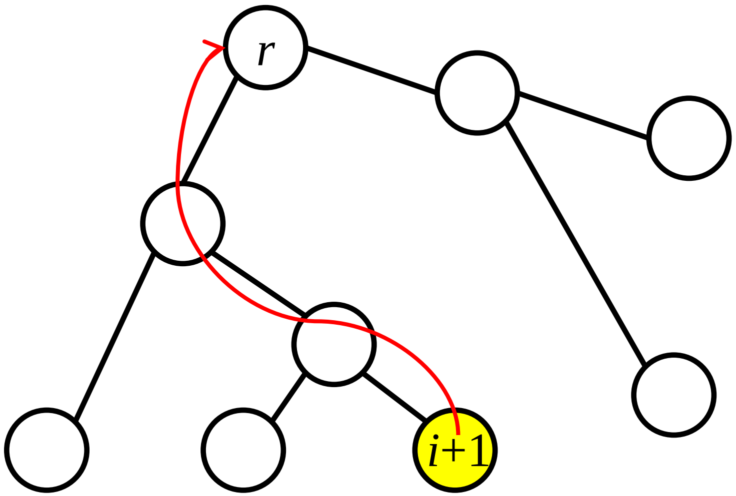
The basic idea of Weiner's algorithm

- **The algorithm**
  1. Start at leaf i+1 of T_{i+1}



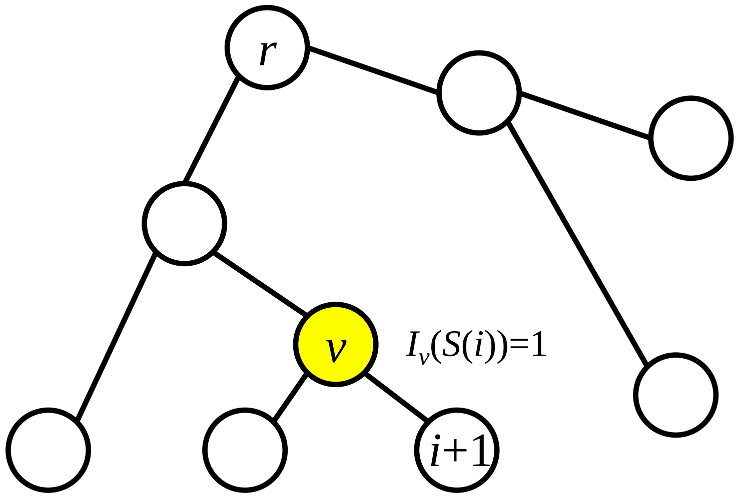
The basic idea of Weiner's algorithm

- **The algorithm**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$
  2. Walk toward the root



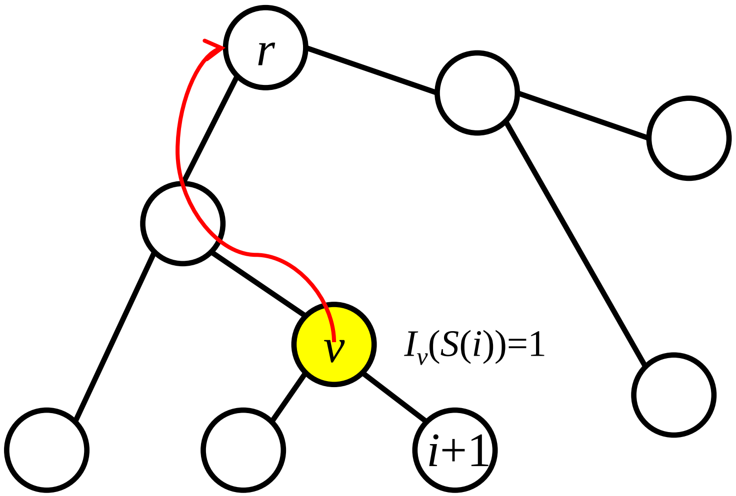
The basic idea of Weiner's algorithm

- **The algorithm**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$
  2. Walk toward the root
     - looking for the first node v such that $I_v(S(i)) = 1$ (if it exists)



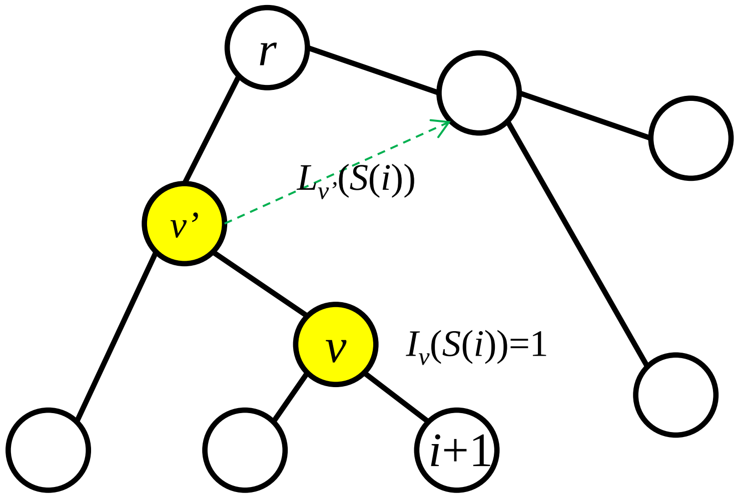
The basic idea of Weiner's algorithm

- **The algorithm**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$
  2. Walk toward the root
     - looking for the first node v such that $I_v(S(i)) = 1$ (if it exists)
  3. Then continues from v to the root



The basic idea of Weiner’s algorithm

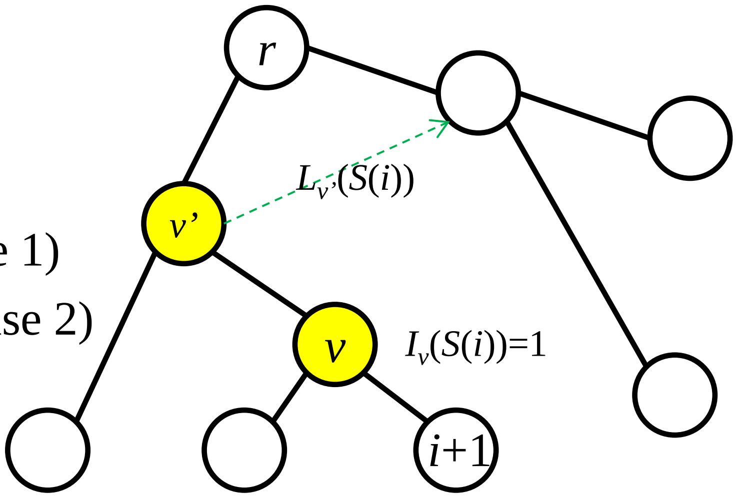
- **The algorithm**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$
  2. Walk toward the root
     - looking for the first node v such that $I_v(S(i)) = 1$ (if it exists)
  3. Then continues from v to the root
     - Searching for the first node v’
     - it encounters (possibly v) where $L_{v'}(S(i))$ is nonnull



The basic idea of Weiner's algorithm

- **The algorithm**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$
  2. Walk toward the root
     - looking for the first node v such that $I_v(S(i)) = 1$ (if it exists)
  3. Then continues from v to the root
     - Searching for the first node v'
     - it encounters (possibly v) where $L_{v'}(S(i))$ is nonnull

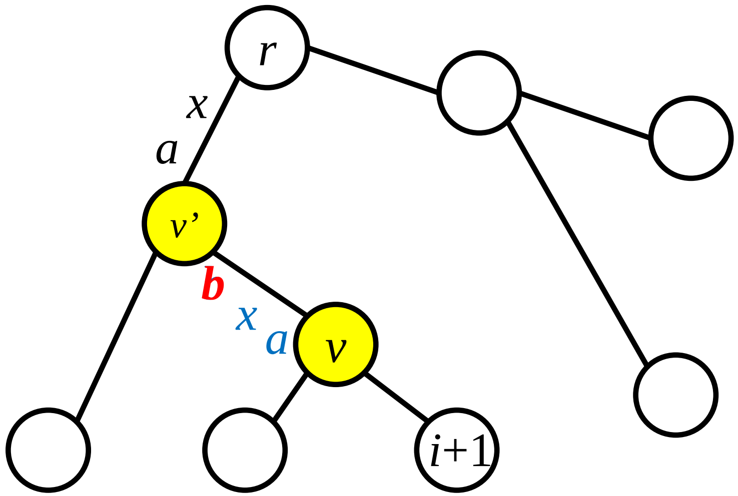
- **Three cases**
  A. Both v and v' exist(good case)
  B. Neither v nor v' exist(degenerate case 1)
  C. v exists but v' does not(degenerate case 2)



A. The algorithm in the good case

- ***l_i***
  - the number of characters on the path between v' and v
  - If $l_i = 0$, then $v' = v$

- ***c***
  - The first character of these l_i characters (if $l_i > 0$)

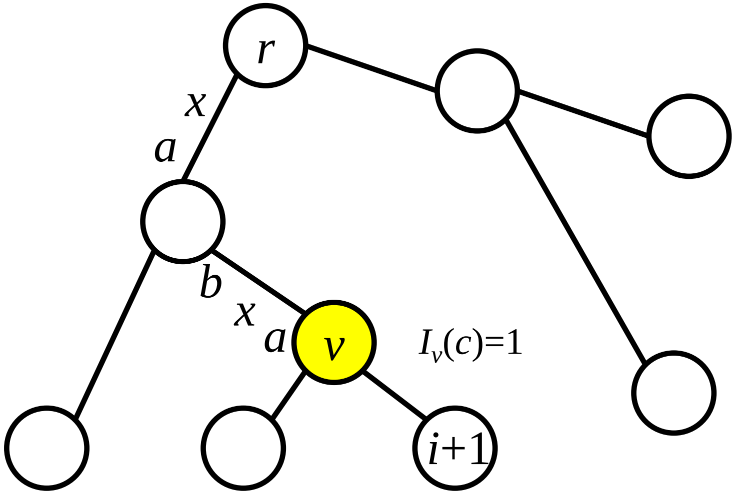


A. The algorithm in the good case

- **Theorem 6.2.1**
  - Assume that node *v* has been found by the algorithm and that it has path-label α. Then the string *Head*(*i*) is exactly *S*(*i*)α.

  - Ex) i = 4, $S(i)$ = c

    => *Head*(4) = cxabxa



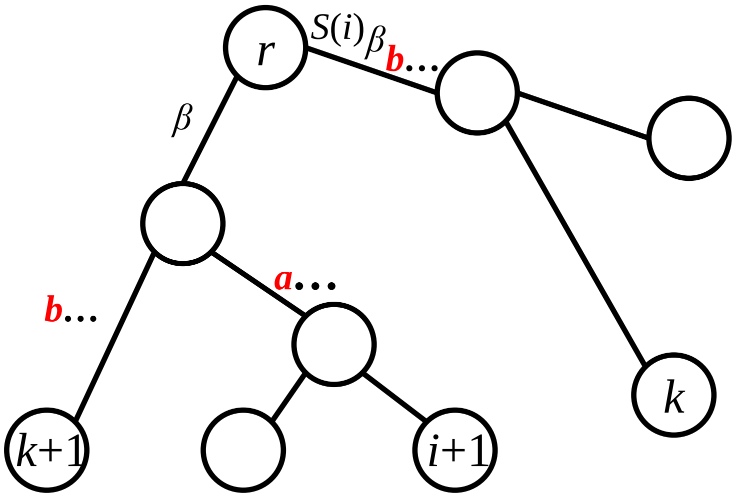
A. The algorithm in the good case

- **Proof**
  - *Head*(i)
    - the longest prefix of Suff_i that is also a prefix of Suff_k for some $k > i$
  - $I_v(S(i)) = 1$
    - there is a path that begins with $S(i)$
    - So *Head*(i) is at least one character long.
  - Therefore, we can express *Head*(i) as $\boldsymbol{S(i)\beta}$,

    for some (possibly empty) string β.

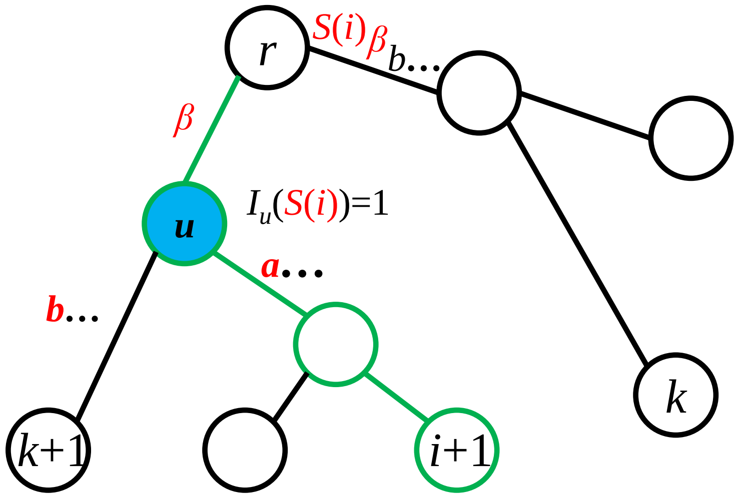
A. The algorithm in the good case

- **Proof**
  - Suff_i and Suff_k
    - both begin with string $Head(i) = S(i)\beta$
    - and differ after that.
  - Suff_i begins $S(i)\beta\boldsymbol{a}$ and Suff_k begins $S(i)\beta\boldsymbol{b}$
    - then Suff_{i+1} begins $\beta\boldsymbol{a}$ and Suff_{k+1} begins $\beta\boldsymbol{b}$.
  - Therefore, there must be a path
    - from the root labeled β
    - that extends in two ways with ***a*** and ***b***



A. The algorithm in the good case

- **Proof**
  - Hence there is a node ***u*** with path-label β, and $I_u(S(i)) = 1$
  - Further, node ***u*** must be on the path to leaf $i + 1$
    - since β is a prefix of suff_{i+1}



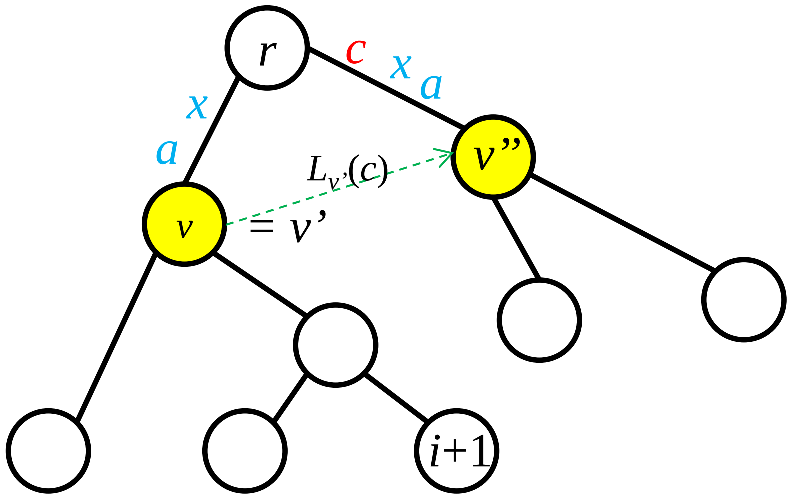
A. The algorithm in the good case

- **Proof**
  - $I_v(S(i)) = 1$ and v has path-label α
    - So $Head(i)$ must begins with $S(i)\alpha$
    - That means that $\boldsymbol{\alpha}$ **is a prefix of** $\boldsymbol{\beta}$
    - so node $\boldsymbol{u}$ must either be $\boldsymbol{v}$ **or below** $\boldsymbol{v}$ on the path to leaf $i + 1$
  - If $u \neq v$ then
    - u would be a node below v on the path to leaf $i + 1$ and $I_v(S(i)) = 1$

      ~> **contradict to choice of node** $\boldsymbol{v}$
  - So $v = u$, $\alpha = \beta$
  - That is, ***head*(*i*) is exactly the string *S*(*i*)*α***

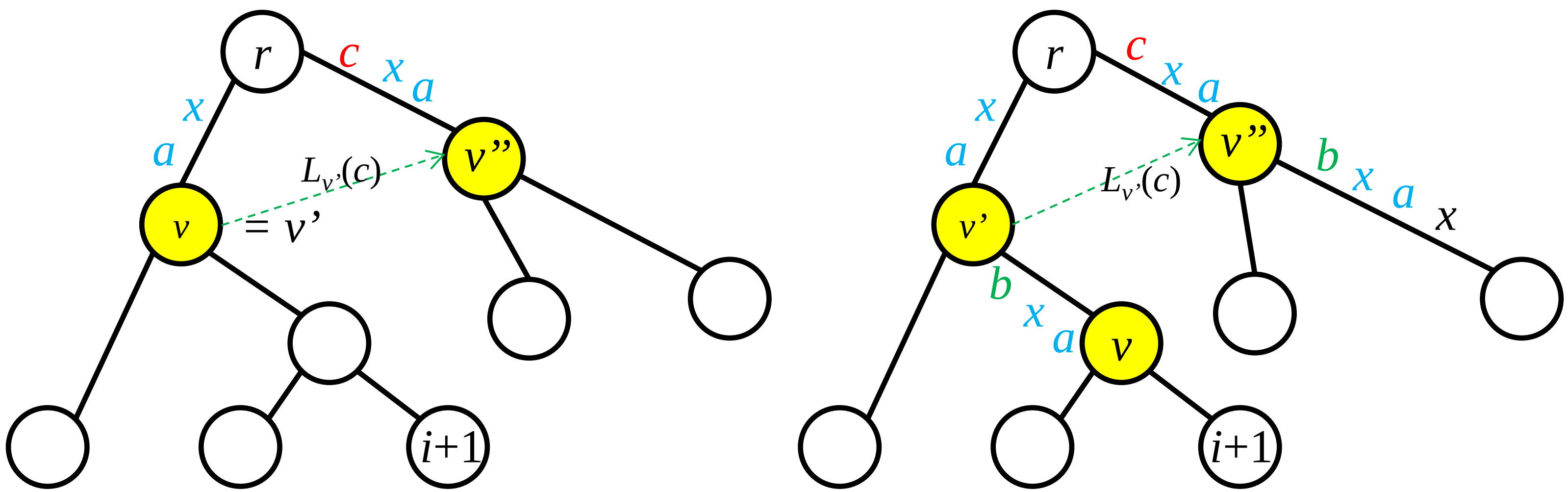
A. The algorithm in the good case

- **Theorem 6.2.2**
  - Assume both v and v' have been found and $L_v(S(i))$ points to node v"
    - If l_i=0 then *Head*(i) ends at v"
    - Otherwise it ends after exactly l_i characters on a single edge out of v" that starts with c.



A. The algorithm in the good case

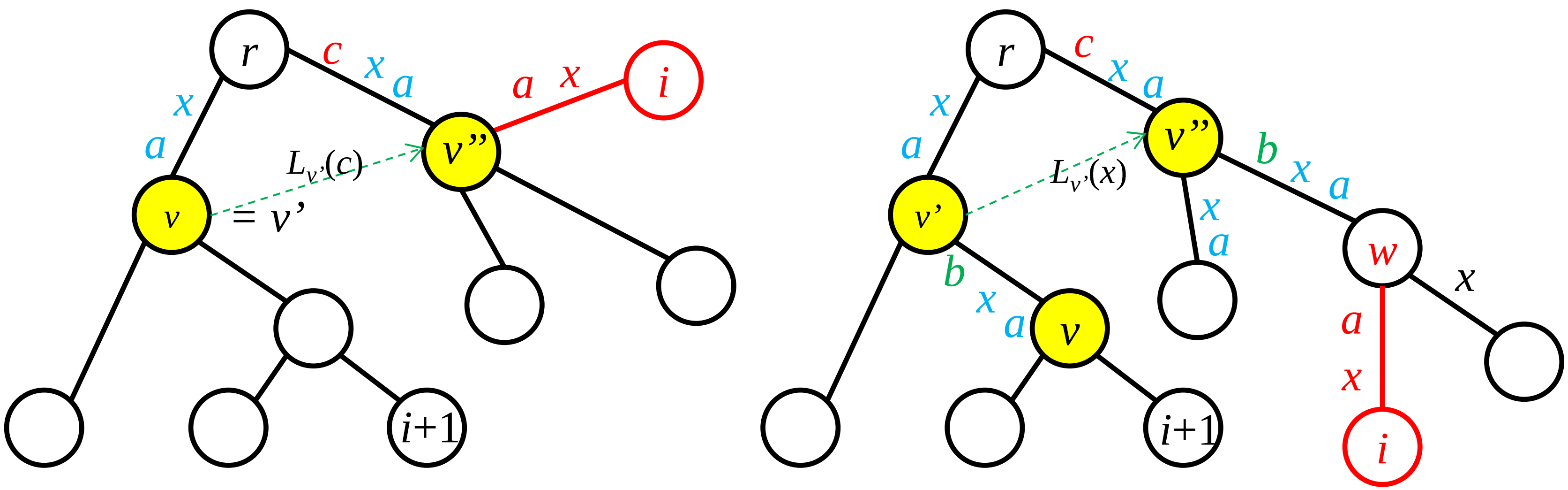
- **Proof**
  - Since *v'* is on the path to leaf *i*+1 and $L_{v'}(S(i))$ Points to node *v''*
    - The path from the root labeled *Head*(*i*) must include *v''*
  - By theorem 6.2.1, *Head*(*i*) = *S*(*i*)*α*, so *Head*(*i*) must end exactly l_i characters below *v''*



A. The algorithm in the good case

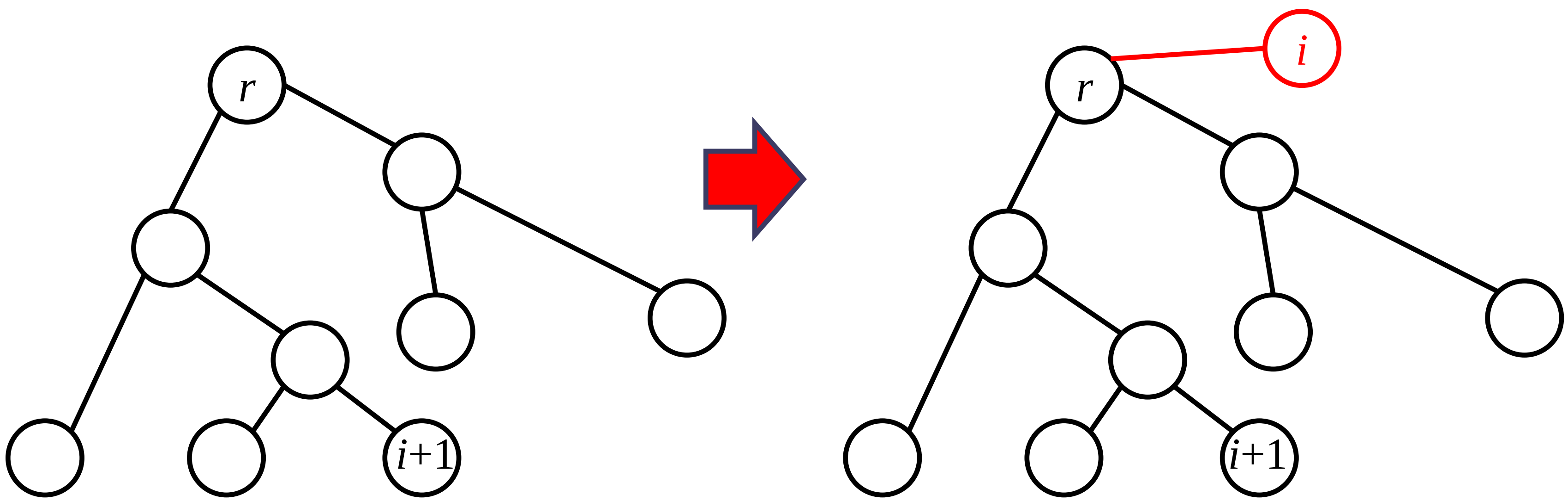
- **Tree T_i is then constructed by**
  - subdividing edge *e*
  - creating a node *w* at this point
  - adding a new edge from *w* to leaf *i* labeled with the remainder of Suff_i

$O(1)$



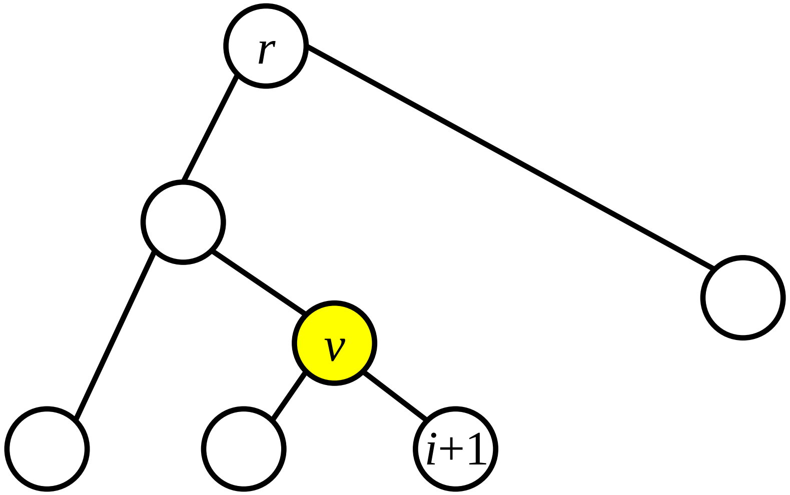
B. Degenerate case 1

- **Degenerate case 1: Neither *v* nor *v*' exist**
  - $I_r(S(i)) = 0$
  - So, Head(i) is the empty string and ends at the root



C. Degenerate case 2

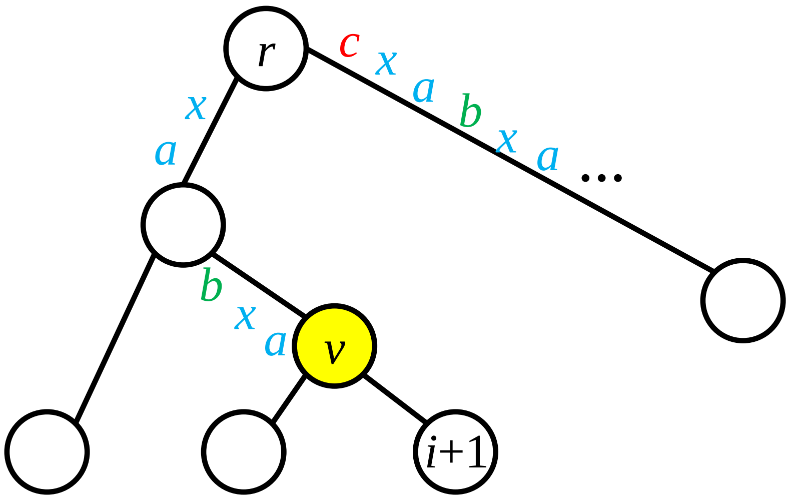
- **Degenerate case 2: *v* exists but *v*' does not**
  - $I_v(S(i)) = 1$ for some v (possibly the root), but v' does not exist
  - The walk ends at the root with $L_r(S(i))$ = null



C. Degenerate case 2

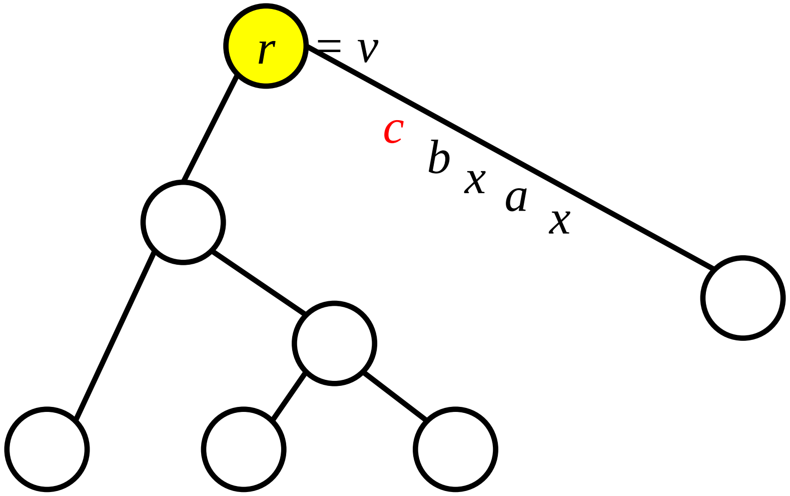
- **Degenerate case 2: *v* exists but *v'* does not**
  - Let t_i be the number of characters from the root to v
    - *a.* $t_i = 0$ (when v is the root node)
    - *b.* $t_i > 0$ (else)

  - From Theorem 6.2.1, *Head*(i) ends exactly t_i+1 characters from the root



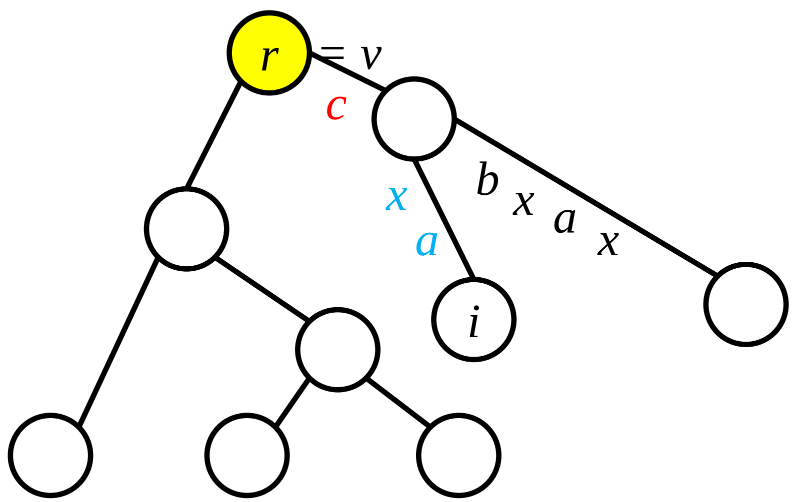
C. Degenerate case 2

- **Degenerate case 2: *v* exists but *v'* does not**
  - *Head*(*i*) ends exactly t_i+1 characters from the root
    - *a.* If t_i=0
      - *Head*(*i*) ends after the first character, *S*(*i*) on edge *e* which start at root



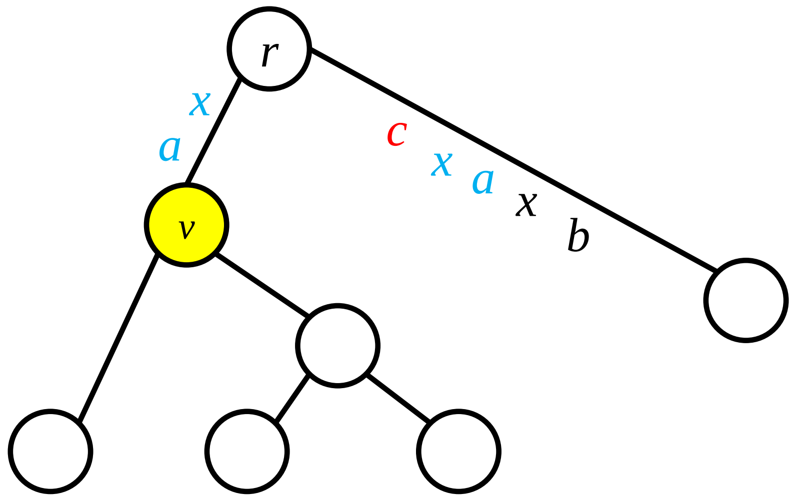
C. Degenerate case 2

- **Degenerate case 2: *v* exists but *v'* does not**
  - *Head*(i) ends exactly t_i+1 characters from the root
    - *a.* If t_i=0
      - *Head*(i) ends after the first character, *S*(i) on edge *e* which start at root



C. Degenerate case 2

- **Degenerate case 2: *v* exists but *v'* does not**
  - *Head*(*i*) ends exactly t_i+1 characters from the root
    - *b.* If t_i>1
      - *Head*(*i*) ends exactly t_i+1 character from the root

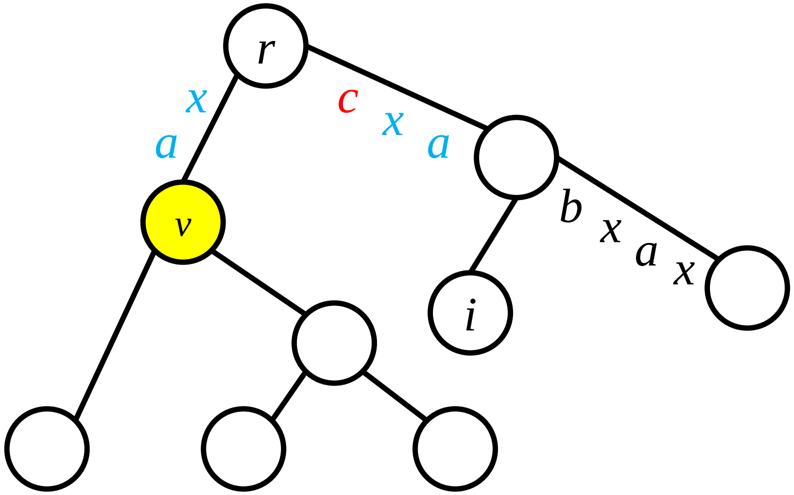


C. Degenerate case 2

- **Degenerate case 2: *v* exists but *v'* does not**
  - *Head*(i) ends exactly t_i+1 characters from the root

    b. If t_i>1

    - *Head*(i) ends exactly t_i+1 character from the root



The two degenerate cases

- **In either of these degenerate cases**
  - *Head*(i) is found in constant time after the walk reaches the root

The full algorithm for creating $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$

- **Weiner's Tree extension**
  1. Start at leaf i+1 of $\mathcal{T}_{i+1}$ and walk toward the root searching for the first node v on the walk such that $I_v(S(i)) = 1$
  2. If the root is reached and $I_r(S(i)) = 0$ (that is, degenerate case 1),
     - create a new node and new edge from root
  3. Let v be the node found(possibly the root) such that $I_v(S(i)) = 1$
     - Then continue walking upward searching for the first node v'(possibly v itself) such that $L_{v'}(S(i))$ is nonnull

     3a. If the root is reached and $L_r(S(i))$ is null(that is, degenerate case 2)

     3b. If v' was found such that $L_{v'}(S(i))$ is v''(that is, the good case)

Correctness

- **The algorithm correctly creates tree $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$**
  - from Theorems 6.2.1, 6.2.2 and the discussion of the degenerate cases
  - although before it can create $\mathcal{T}_{i-1}$, it must update the *I* and *L* vectors

How to update the vectors

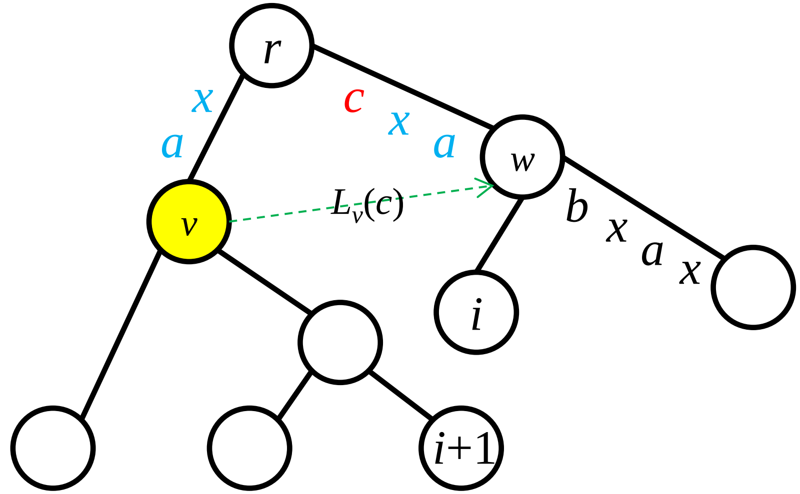
- **After finding (or creating) node *w***
  - We must update the I and L vectors
    - so that they are correct for tree $\mathcal{T}_i$

  - Update L vectors

  - Update I vectors

How to update the vectors

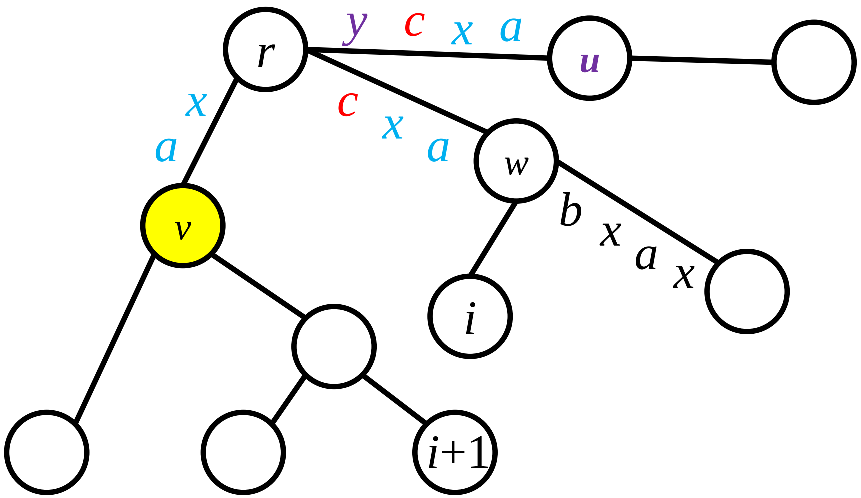
- **Update L vectors**
  - If node v was found(the good case and degenerate case 2)
    - Node w has path-label $S(i)\alpha$ in $\mathcal{T}_i$
    - In this case, $L_v(S(i))$ should be set to point to w in $\mathcal{T}_i$
    - If node w is newly created, all its link entries should be null



	a	…	x	y	z
L	null	…	null	null	null

How to update the vectors

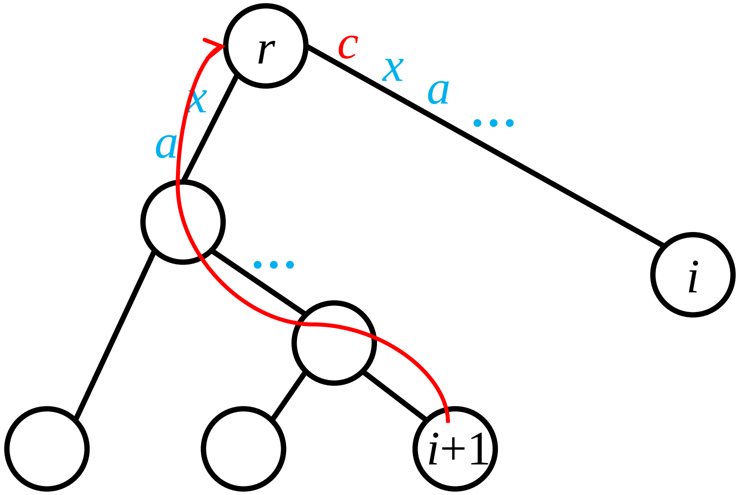
- **Update L vectors**
  - If node w is newly created, all its link entries should be null
    - Proof
      - Suppose there is a node ***u*** in $\mathcal{T}_i$ with path-label $xHead(i)$
      - But then there must have been a node in $\mathcal{T}_{i+1}$ with path-label $Head(i)$



	a	…	x	y	z
L	null	…	null	null	null

How to update the vectors

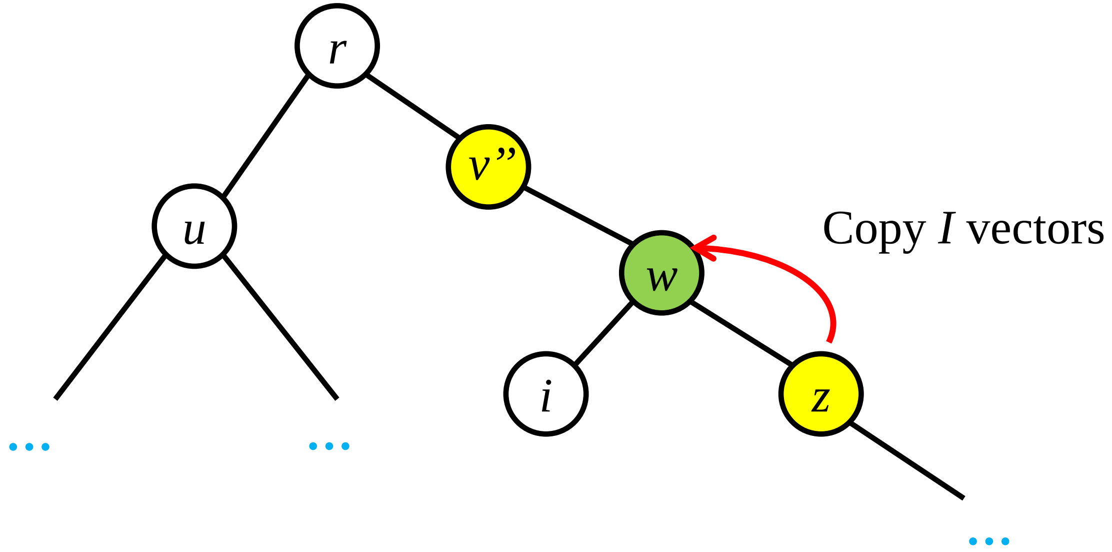
- **Update *I* vector**
  - For every node ***u*** on the path from the root to leaf i+1
    - $I_u(S(i))$ must be set to 1 in $\mathcal{T}_i$
      - Since there is now a path for sting Suff_i in $\mathcal{T}_i$



	…	c	…
I	…	1	…

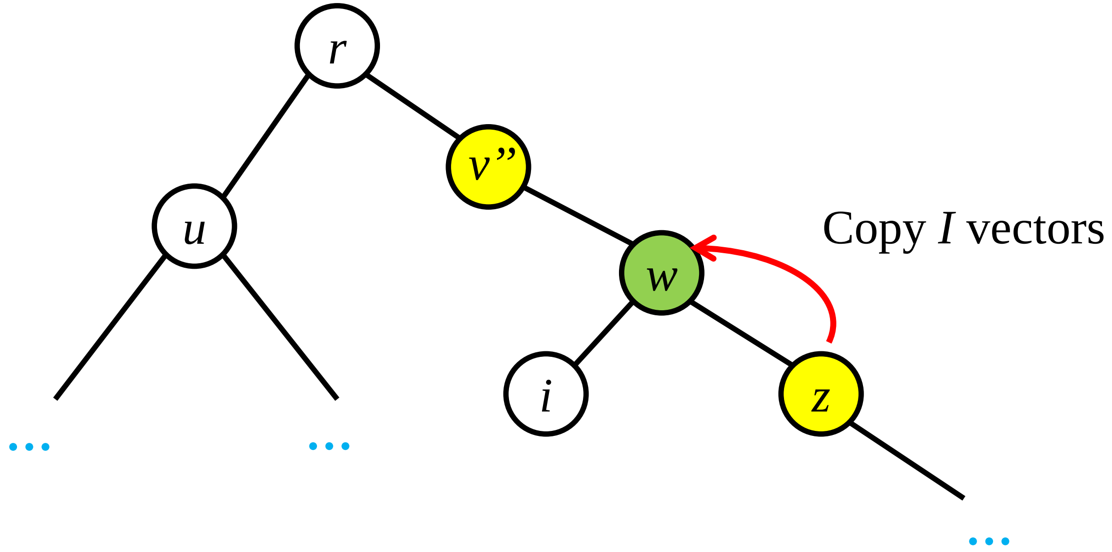
How to update the vectors

- **Update *I* vector**
  - **Theorem 6.2.3**
    - When a new node ***w*** is created in the interior of an edge (*v*”, *z*)
      - *I* vector for *w* should be copied from the *I* vector for *z*



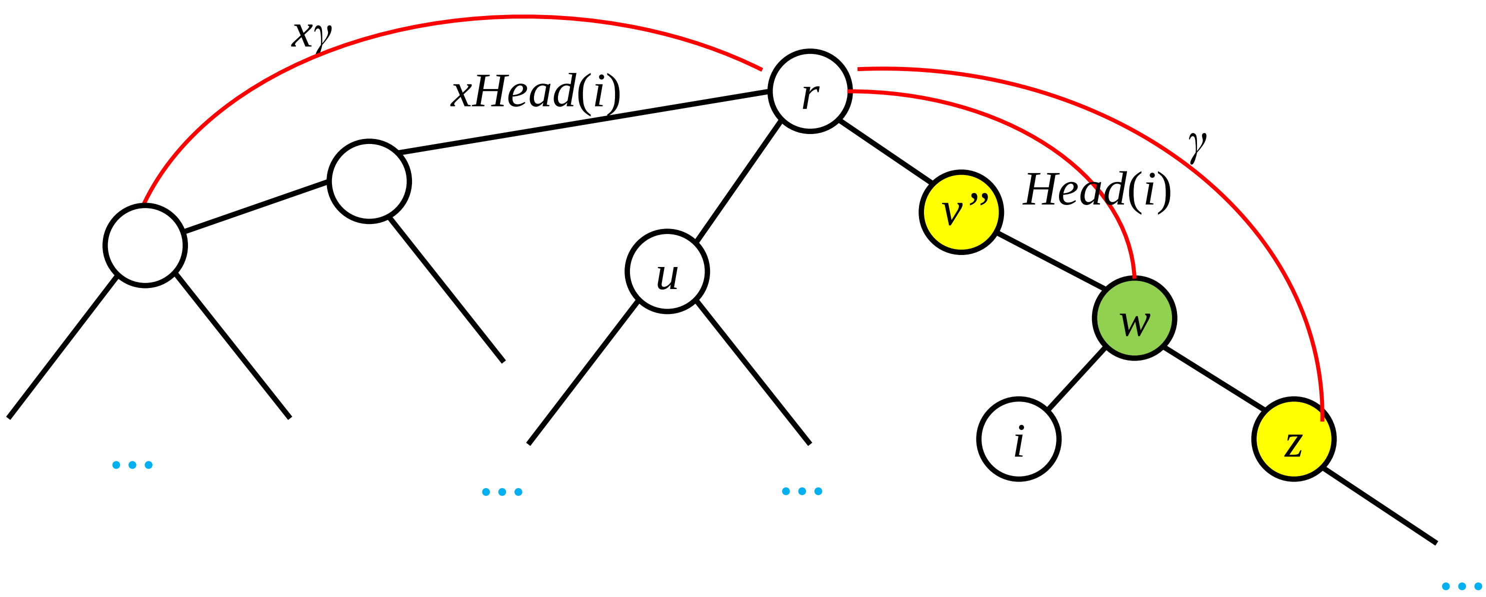
How to update the vectors

- **Update *I* vector**
  - **Proof**
    - It is immediate that if $I_z(x) = 1$ then $I_w(x)$ must also be 1
    - Can it happen that $I_w(x) = 1$ and $I_z(x) = 0$ at the moment that w is created? ~> **It cannot**



How to update the vectors

- **Update *I* vector**
  - **Proof**
    - It is immediate that if $I_z(x) = 1$ then $I_w(x)$ must also be 1
    - Can it happen that $I_w(x) = 1$ and $I_z(x) = 0$ at the moment that w is created? ~> **It cannot**



Time analysis of Weiner's algorithm

- **The time to construct $\mathcal{T}_i$**
  - ≈ the time needed during the walk from leaf i+1 ending either at v' or the root
    - Move to one node(constant time)
    - Follow a L link pointer(constant time)
    - Add a node and edge(constant time)

  - So, ≈ the number of nodes encountered on the walk from leaf i+1
    = **The node-depth**(the number of nodes from the root to node v)

Time analysis of Weiner's algorithm

- **The time to construct $\mathcal{T}_i$**
  - When the algorithm walks up a path from a leaf
    - The current node-depth can decrease by one each time
  - A new node is created
    - The current node-depth can increase by one each time

Time analysis of Weiner's algorithm

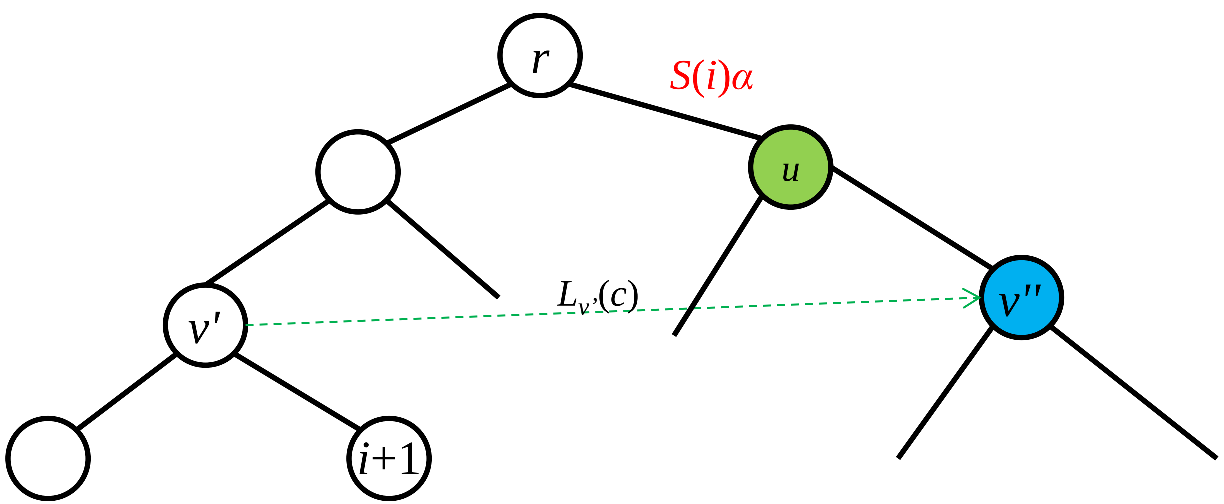
- **The time to construct T_i**
  - A link pointer is traversed

Lemma 6.2.1

When the algorithm traverses a link pointer from a node *v'* to a node *v''*, the current node-depth increases by **at most one**

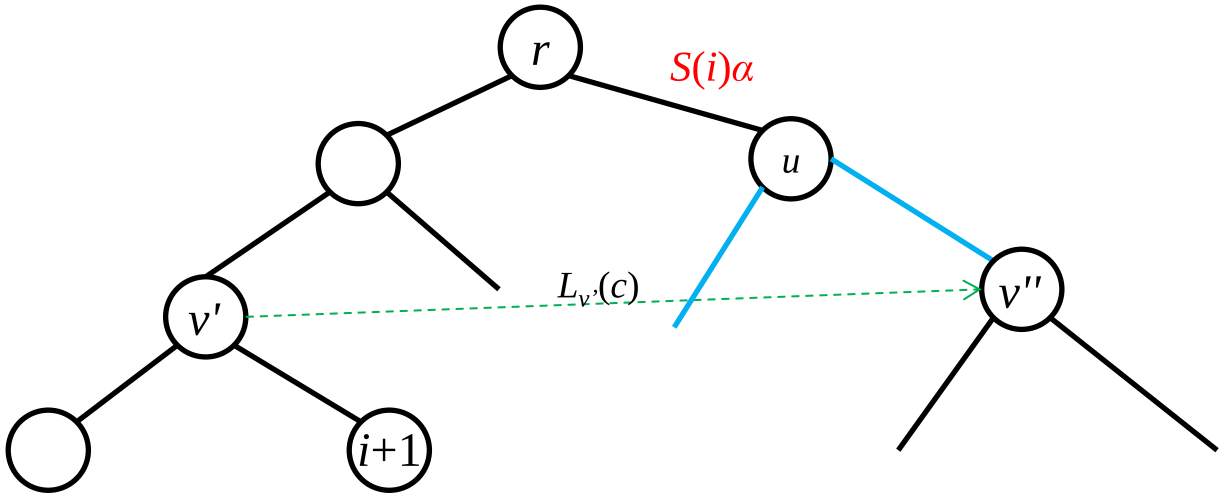
Time analysis of Weiner's algorithm

- **The time to construct T_i**
  - Proof
    - Let ***u*** be a nonroot node on the path from the root to v'', and suppose ***u*** has path-label $S(i)\alpha$ for some nonempty string α.
    - All nodes on the root-to-v'' path are of this type
    - except for the single node (if it exists) with path-label $S(i)$.



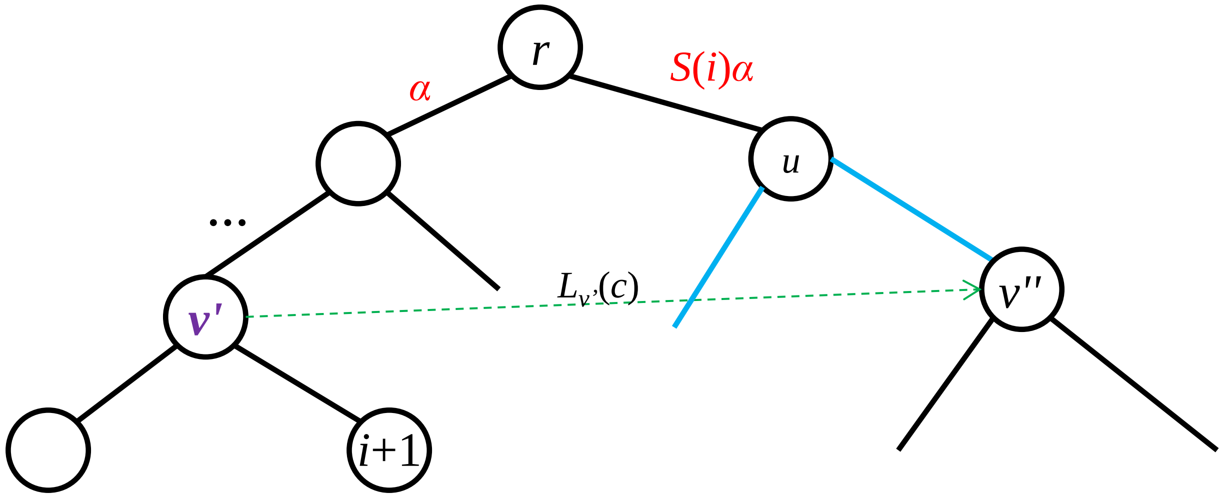
Time analysis of Weiner's algorithm

- **The time to construct $\mathcal{T}_i$**
  - Proof
    - $S(i)\alpha$ is the prefix of Suff_i and of Suff_k for some $k > i$
    - and this string extends differently in the two cases



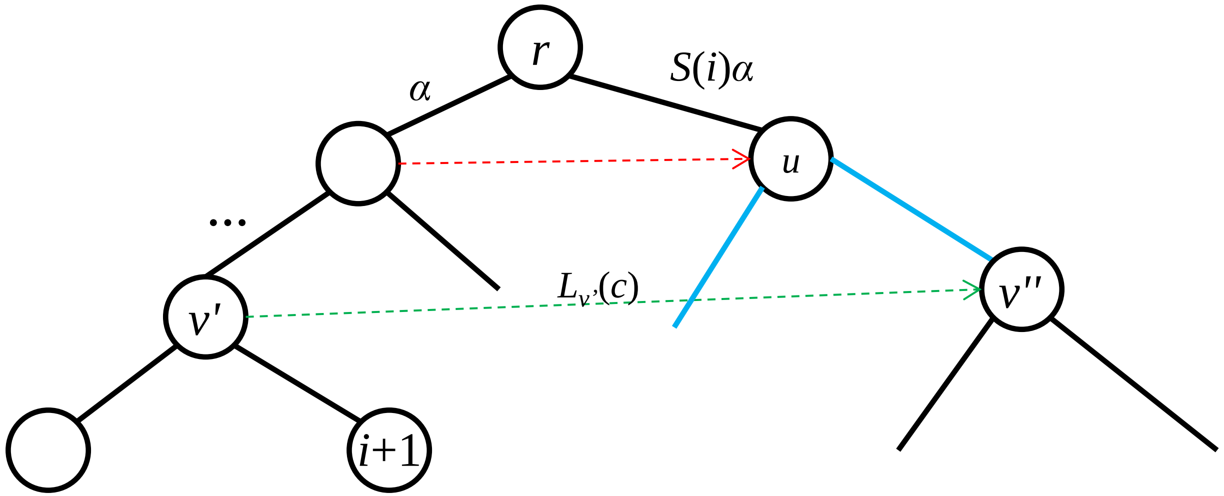
Time analysis of Weiner's algorithm

- **The time to construct $\mathcal{T}_i$**
  - Proof
    - Since ***v'*** is on the path from the root to leaf i+1,
    - α is a prefix of $suff_{i+1}$
    - and there must be a node with path-label α (possibly the root) on the path to v'



Time analysis of Weiner's algorithm

- **The time to construct T_i**
  - Proof
    - Hence the path to *v*' has a node corresponding to every node on the path to *v*''
    - except the node (if it exists) with path-label $S(i)$
    - Hence the depth of *v*'' is **at most one** more than the depth of *v*', although it could be less



Time analysis of Weiner's algorithm

- **Theorem 6.2.4**
  - Assuming a finite alphabet, Weiner's algorithm constructs the suffix tree for a string of length n in $O(n)$ time
  - Proof
    - the total number of increased in the current node-depth is at most $2n$
    - the current node-depth can also only decrease at most $2n$ times
    - So the total number of nodes visited during all the upward walks is
      - At most $2n$

Last comments about Weiner's algorithm

- **Theorem 6.2.5**
  - If *v* is a node in the suffix tree labeled by the string $x\boldsymbol{\alpha}$
    - where x is a single character
  - then there is a node in the tree labeled with the string $\boldsymbol{\alpha}$