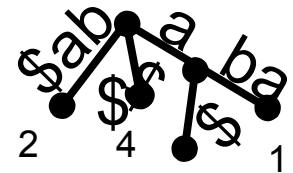
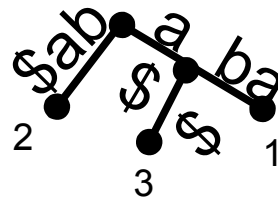
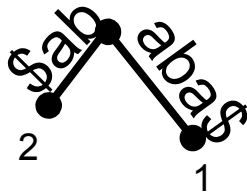
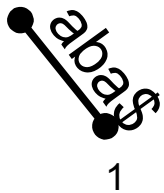


Ukkonen's suffix tree construction algorithm



Motivation

Yet another suffix tree construction algorithm...

Why?

Motivation

Yet another suffix tree construction algorithm...

Why?

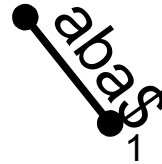
An online algorithm, i.e. one we can update with new sequences

Sketch of algorithm

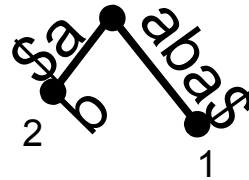
- Recall McCreight's approach:
 - For $i = 1 \dots n+1$, build compressed trie of $\{x[j..n] \$ \mid j \leq i\}$
- Ukkonen's approach:
 - For $i = 1 \dots n+1$, build compressed trie of $\{x[j..i] \$ \mid j \leq i\}$
 - Compressed trie of all suffixes of prefix $x[1..i] \$$ of $x \$$
 - A suffix tree except for “leaf” property

McCreight's algorithm – x=aba

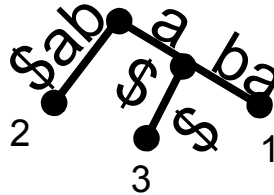
$T_1 : \{x\$_{[j..n]} \mid j \leq 1\}$



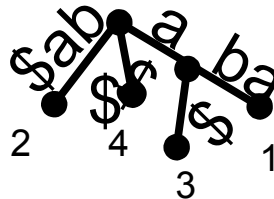
$T_2 : \{x\$_{[j..n]} \mid j \leq 2\}$



$T_3 : \{x\$_{[j..n]} \mid j \leq 3\}$



$T_4 : \{x\$_{[j..n]} \mid j \leq 4\}$

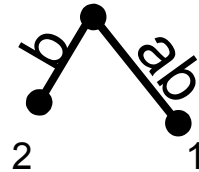


Ukkonen's algorithm – $x=aba$

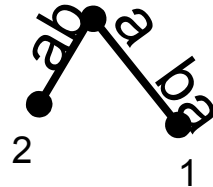
$T_1 : \{x\$_{[j..1]}\}$



$T_2 : \{x\$_{[j..2]}\}$

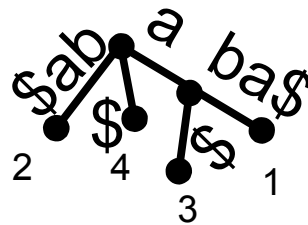


$T_3 : \{x\$_{[j..3]}\}$



Note: no node for $x[3..3] = \text{"a"}$

$T_4 : \{x\$_{[j..n]}\}$

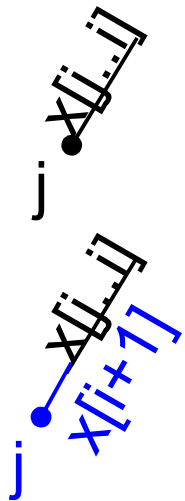


Tasks in iteration i

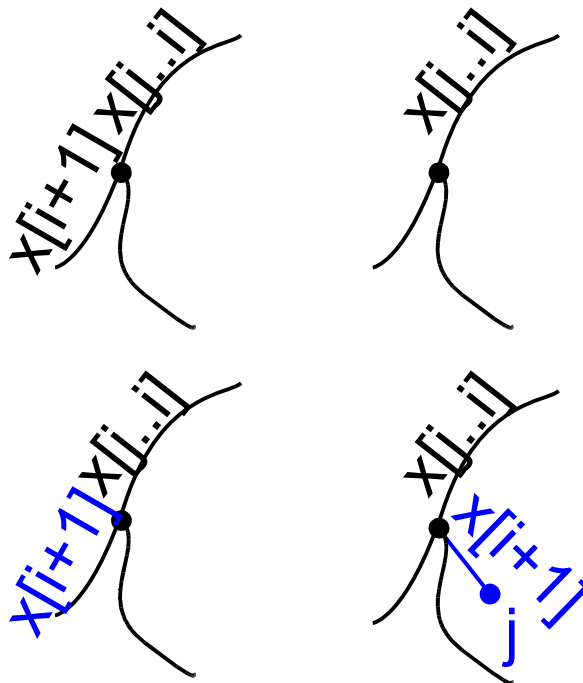
In iteration i we must

- Update each $x[j..i]$ to $x[j..i+1]$
- Add string $x[i+1]$ (special case of above)

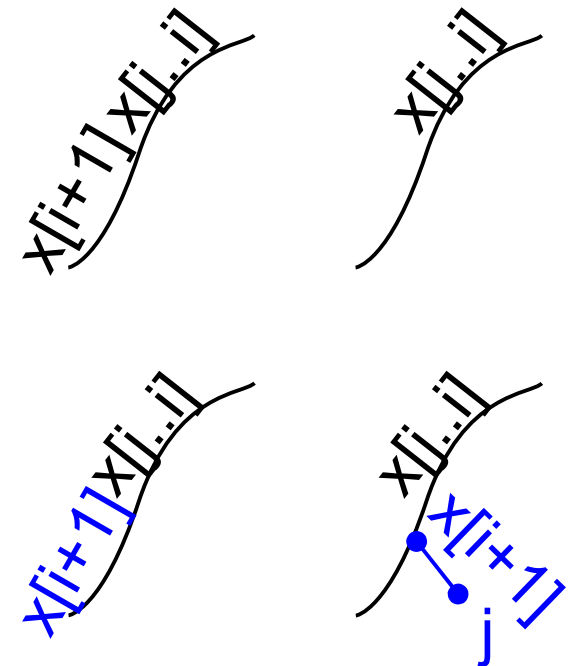
Leaf:



Inner node:



Edge:



First attempt...

“Obvious” algorithm:

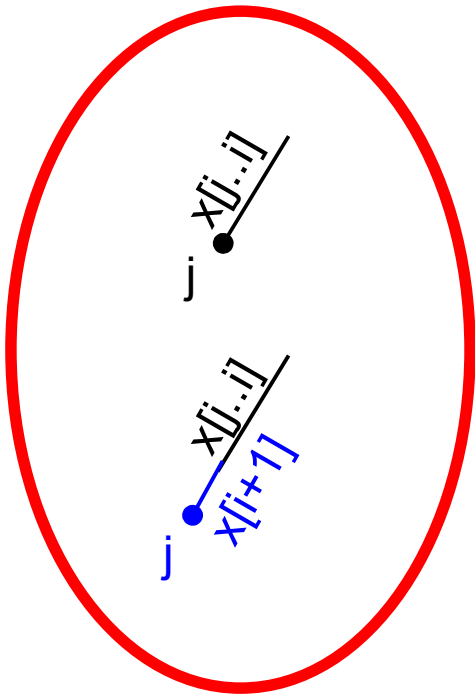
```
For i=1,...,n+1:  
  for j=1,...,i:  
    find x[j..i]  
    append x[i+1]
```

- Running time $O(n^3)$
- Need lots of tricks to get $O(n)$!

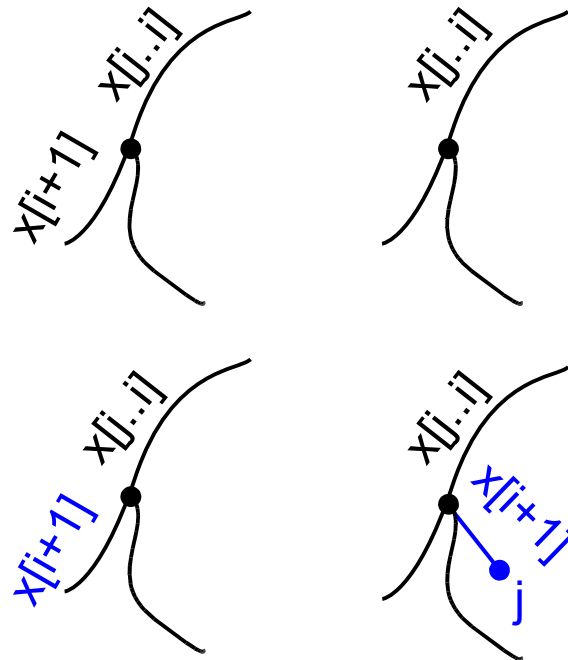
Updating leaves for free

If we label leaves with (k, ∞) – denoting “k to the current i”, updating a leaf is automatic

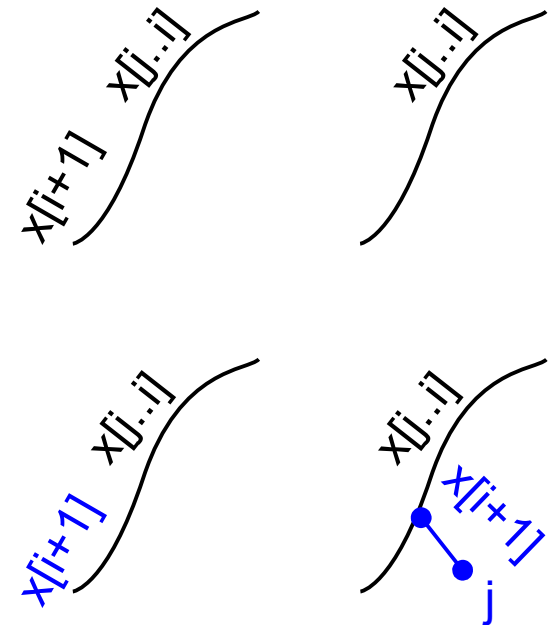
Leaf:



Inner node:



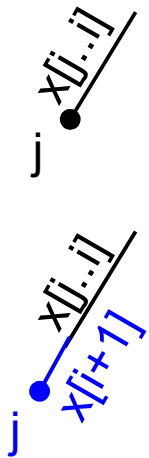
Edge:



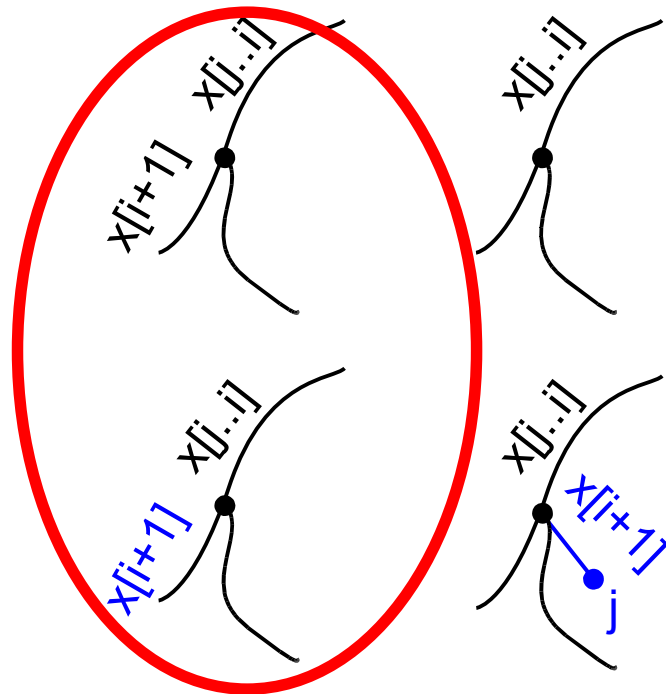
Updating existing strings is free

If $x[j..i+1]$ is already in the tree, the update is automatic

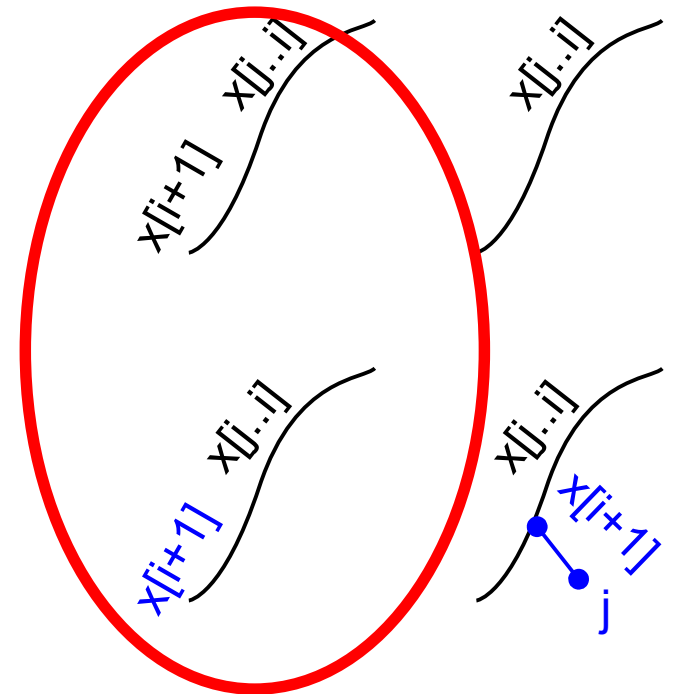
Leaf:



Inner node:



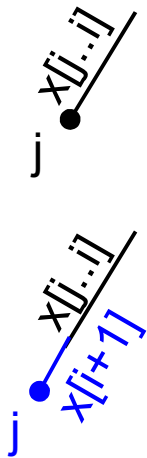
Edge:



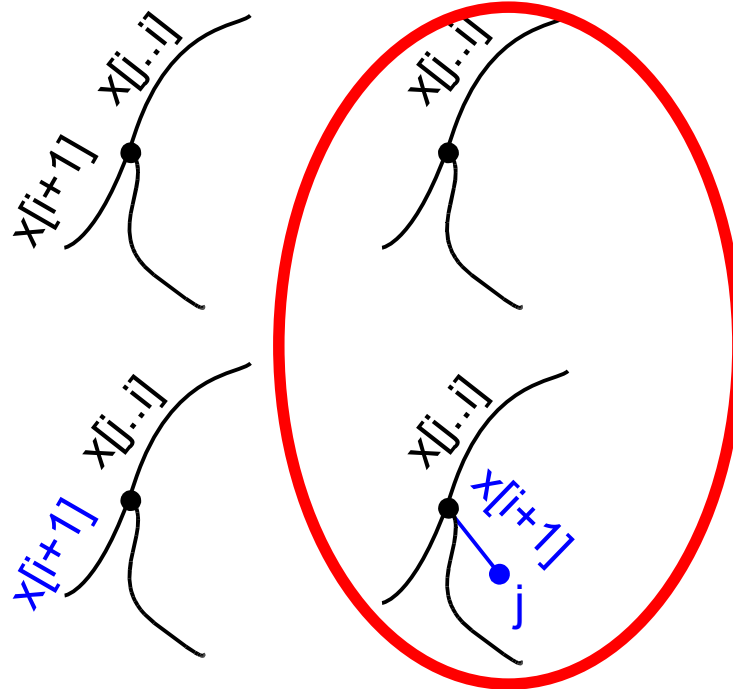
“Real” operations

If we can recognize the free operations, we need only deal with the remaining

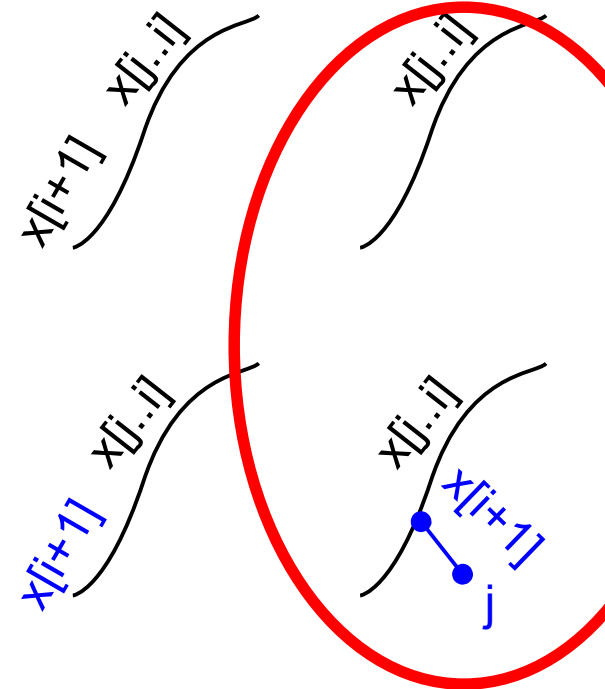
Leaf:



Inner node:



Edge:



Lemma 5.2.4

Let j denote suffix $x[j..i]$ of $x[1..i]$

- a) If $j > 1$ is a leaf node in T_i , then so is $j-1$
- b) If, from $j < i$, there is a path in T_i that begins with “a”, then there is a path in T_i from $j+1$ beginning with “a”

Lemma 5.2.4

Let j denote suffix $x[j..i]$ of $x[1..i]$

- a) If $j > 1$ is a leaf node in T_i , then so is $j-1$
- b) “Once an edge, always an edge”

Lemma 5.2.4

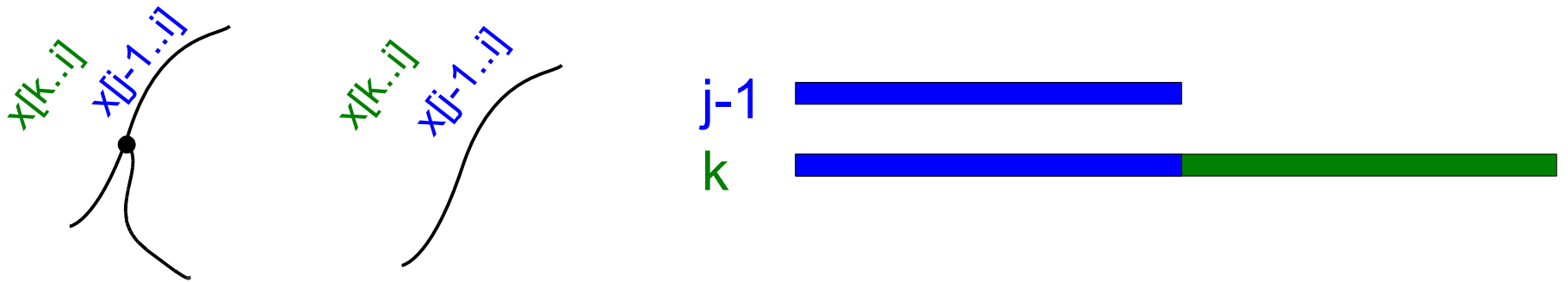
Let j denote suffix $x[j..i]$ of $x[1..i]$

- a) “Once a leaf, always *been* a leaf”
- b) If, from $j < i$, there is a path in T_i that begins with “a”, then there is a path in T_i from $j+1$ beginning with “a”

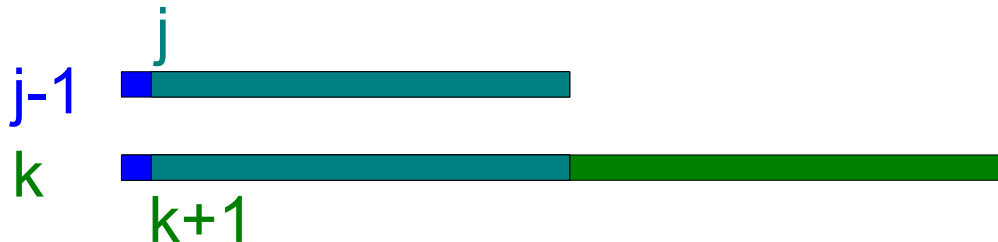
Proof of lemma 5.2.4 a)

If $j > 1$ is a leaf node in T_i , then so is $j-1$

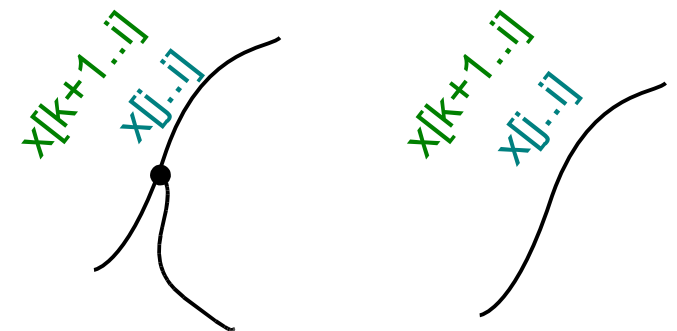
Assume $j-1$ is not a leaf. Then there exists $k < j-1$ such that:



Then



thus:



Proof of lemma 5.2.4 b)

If, from $j < i$, there is a path in T_i that begins with “a”, then there is a path in T_i from $j+1$ beginning with “a”

Assume j is followed by “a”, then there exists $k < j$ such that:

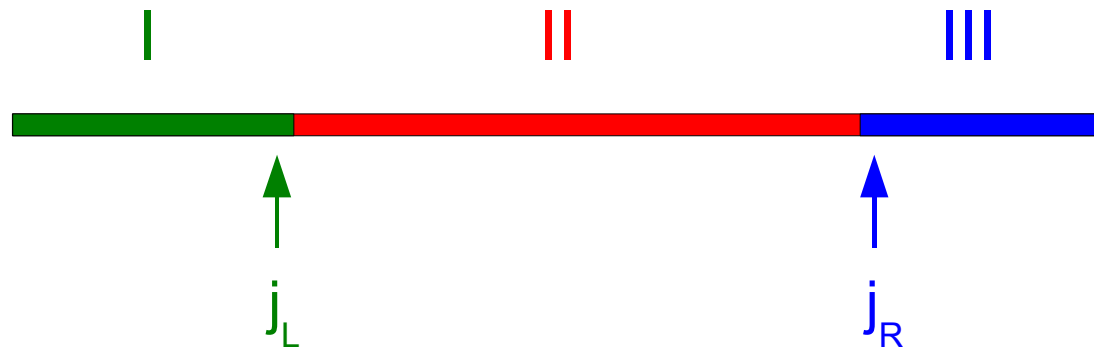


Hence $j+1$ is followed by “a”.

Corollary

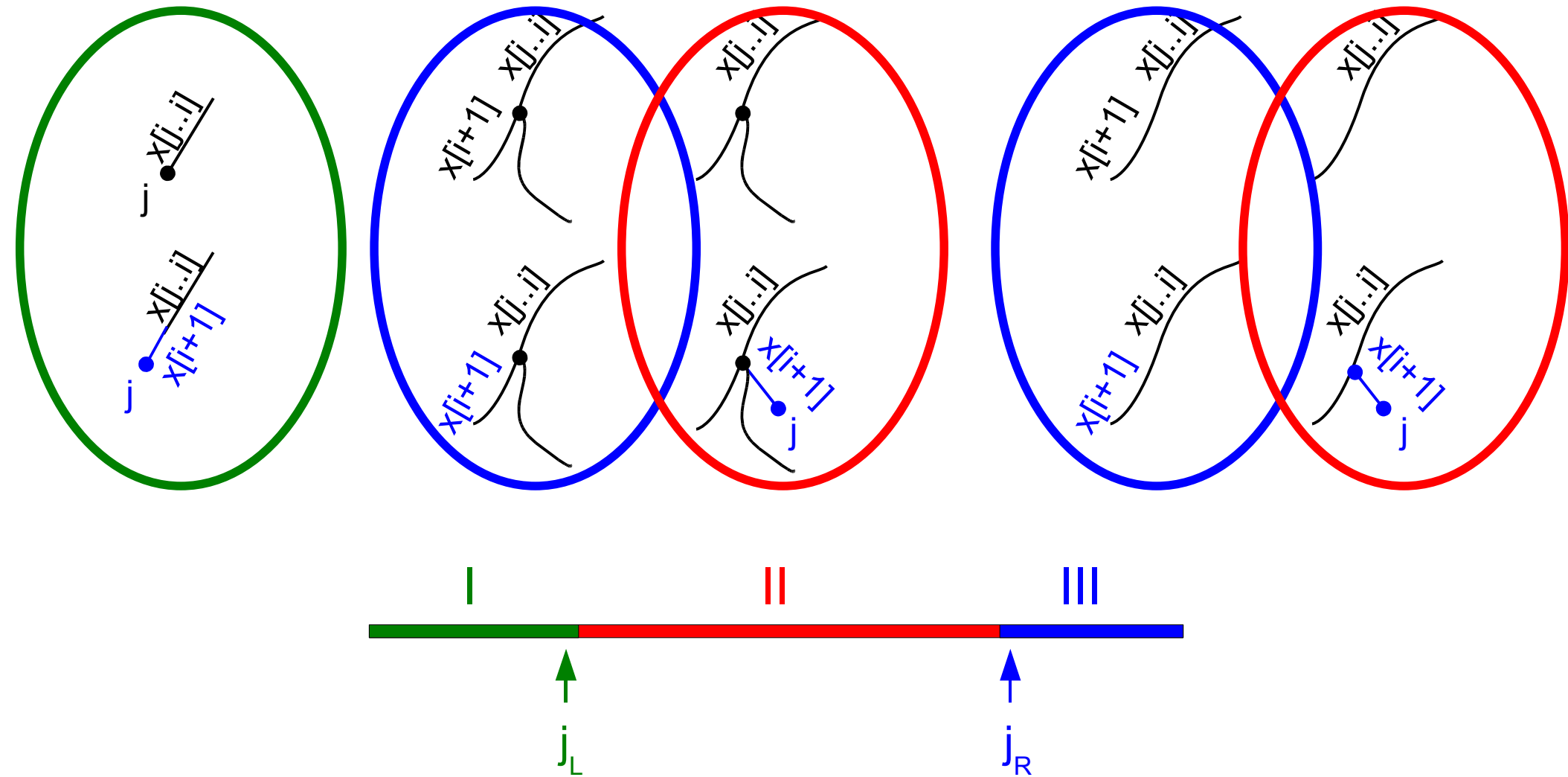
In iteration i , there exist indices j_L and j_R such that:

- All suffixes $j \leq j_L$ are leaves
- All suffixes $j \geq j_R$ are already in the trie



Consequence of corollary

I and III are free operations



Updated algorithm

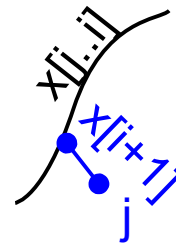
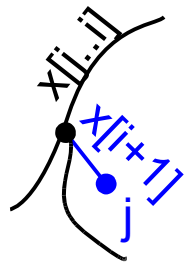
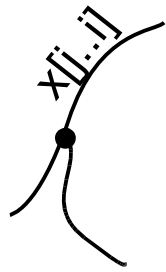
Implicitly handling “free” operations:

```
For  $i=1, \dots, n+1$ :  
  for  $j=j_L, \dots, j_R$ :  
    find  $x[j..i]$   
    append  $x[i+1]$ 
```

- j_L in iteration i is the last leaf inserted in iteration $i-1$ (all smaller indices are already leaves)
- j_R in iteration i is the first index where $x[j..i+1]$ is already in the trie (all larger indices are already in the trie)

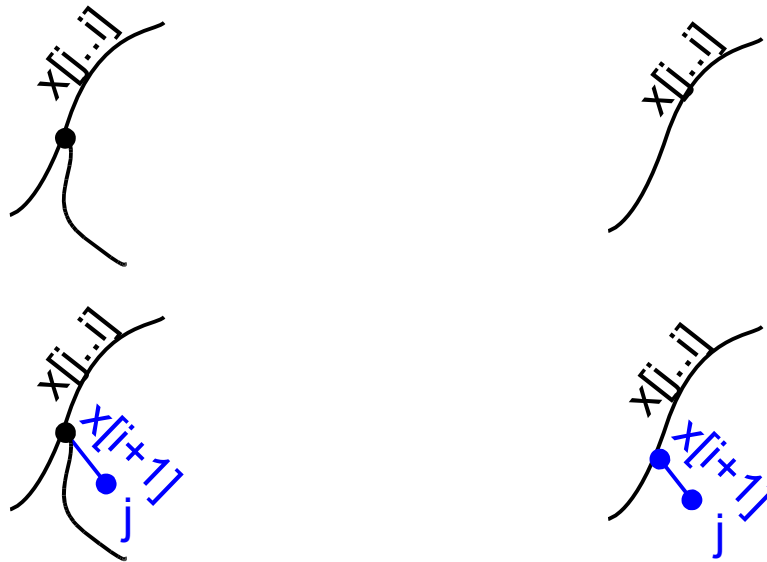
Suffixes in Π are made into leaves

Whenever $j_L < j < j_R$, j is made a leaf:



Suffixes in II are made into leaves

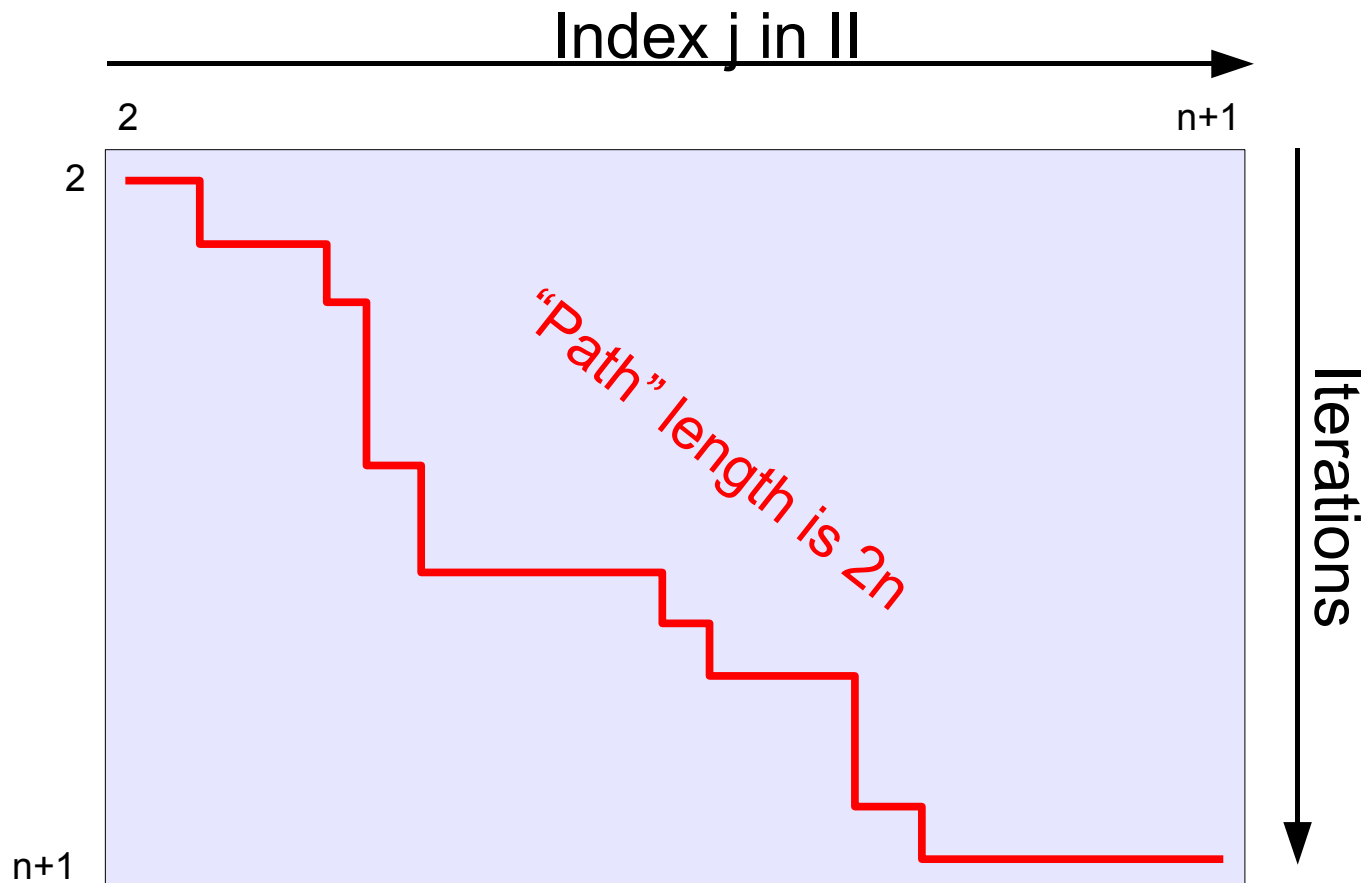
Whenever $j_L < j < j_R$, j is made a leaf:



Once j is a leaf, it will be in I and never in II again

Time to go from II to I

We handle j in II or implicitly in III time $2n$:



Updated running time

Only $2n$
of these:

For $i=1, \dots, n+1$:

for $j=j_L, \dots, j_R$:

find $x[j..i]$

append $x[i+1]$

Constant time

Running time is $2n * T(\text{find } x[j..i])$

Updated running time

Only $2n$
of these:

For $i=1, \dots, n+1$:

for $j=j_L, \dots, j_R$:

find $x[j..i]$

append $x[i+1]$

Constant time?

Constant time

Running time is $2n * T(\text{find } x[j..i])$

- We just have to deal with $T(\text{find } x[j..i])$ in $O(1)$
- No worries!

Using *fastscan* and *s(-)*

- When searching for $x[j..i]$, it is already in the trie
 - We can use **fastscan** for the search
 - $T(\text{find } x[j..i])$ in $O(d)$ where d is the (node-)depth of $x[j..i]$
- If we keep suffix links, $s(-)$, in the tree we can use these as shortcuts

Suffix links

Invariant: All inner nodes have suffix links

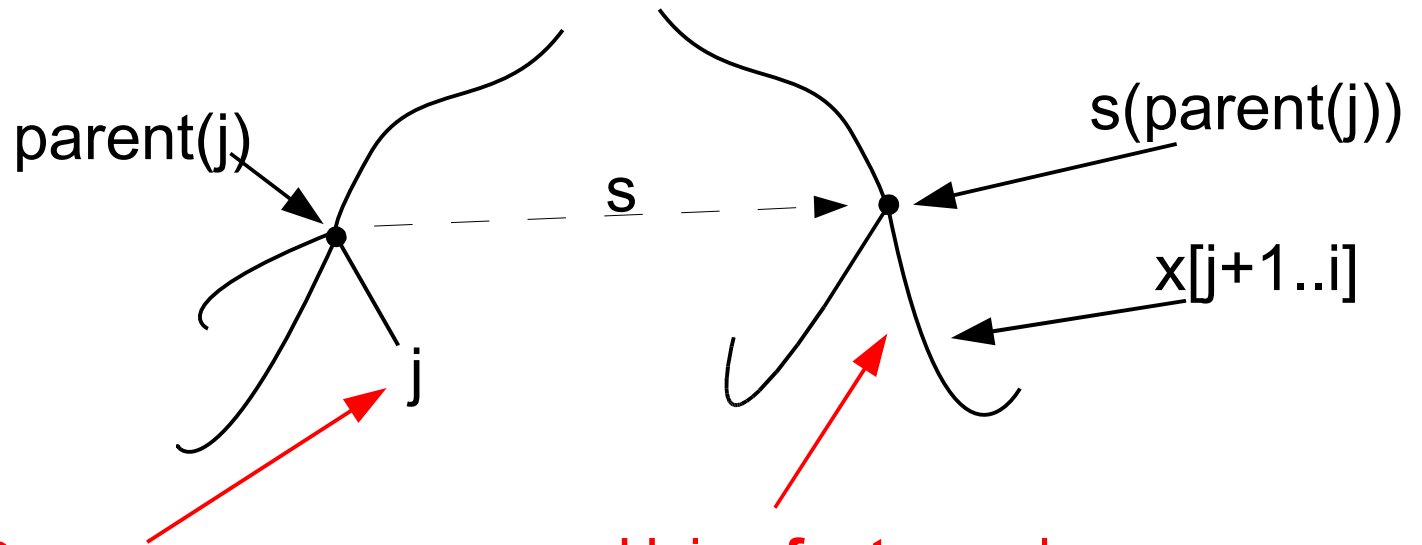
Suffix links

Invariant: All inner nodes have suffix links

Ensuring the invariant:

- We only insert inner nodes $x[j..i]$ when adding leaves j
- Whenever we insert a new node, $x[j..i]$ for some $j < i$, we also find or insert $x[j+1..i]$, and can update $s(x[j..i]) := x[j+1..i]$
- If we insert $x[i..i]$, then $s(x[i..i]) := \varepsilon$

Finding $x[j+1..i]$ from $x[j..i]$



Starting from here
(initial j is j_L and we can keep a
pointer to that node between
iterations)

Using **fastscan** here

Bound on *fastscan*

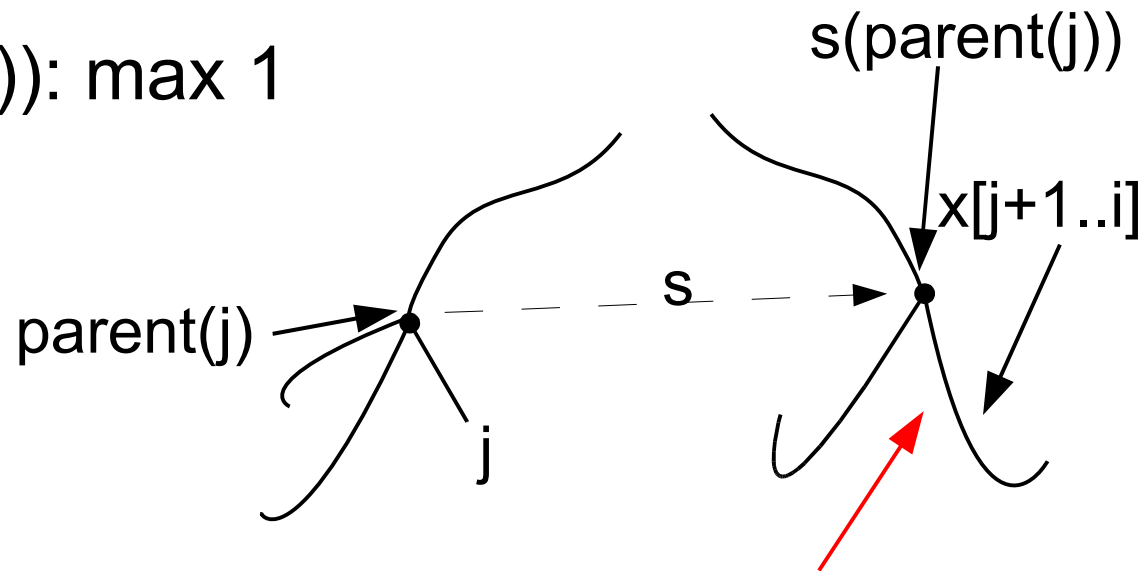
Time usage by **fastscan** is bounded by n – for the maximal (node-)depth in the trie – plus total decrease of (node-)depth

- Decrease in depth:
 - Moving to $\text{parent}(j)$: 1
 - Moving to $s(\text{parent}(j))$: max 1
 - “Restarting” at j_L : ?

Bound on *fastscan*

Time usage by **fastscan** is bounded by n – for the maximal (node-)depth in the trie – plus total decrease of (node-)depth

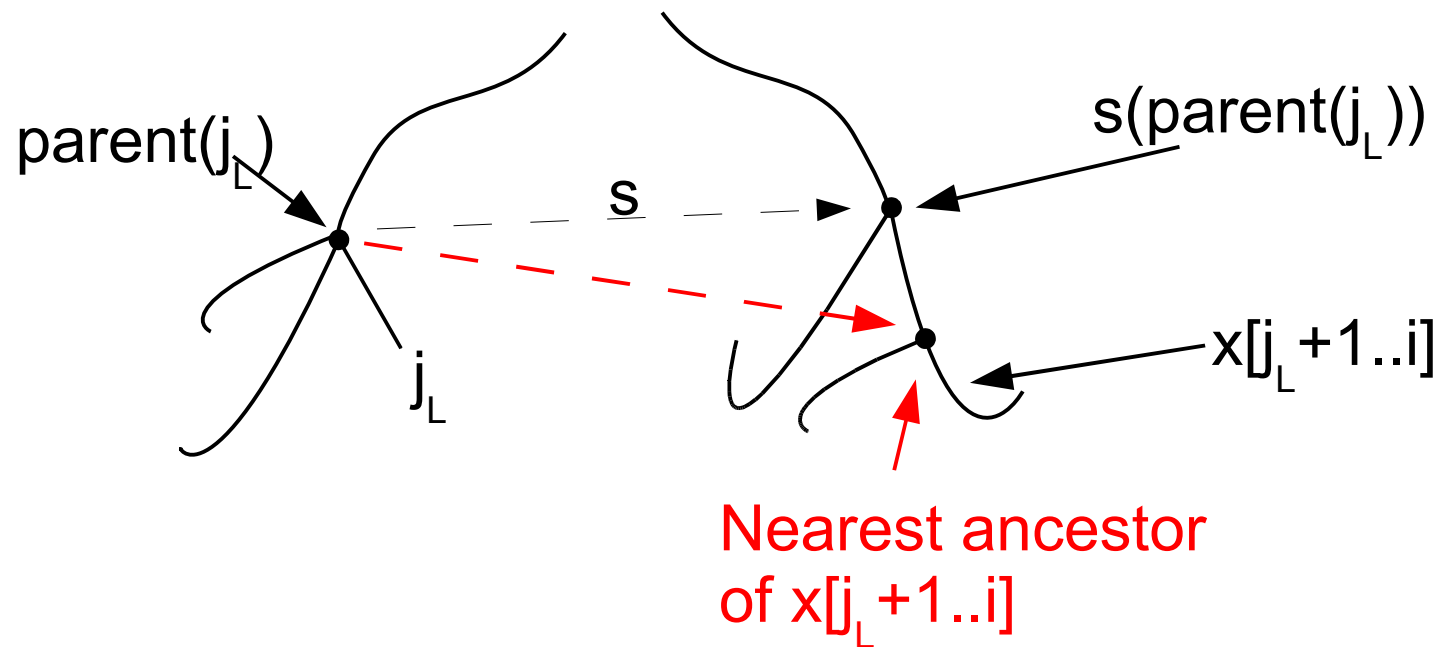
- Decrease in depth:
 - Moving to $\text{parent}(j)$: 1
 - Moving to $s(\text{parent}(j))$: max 1
 - “Restarting” at j_L : ?



Using **fastscan** here

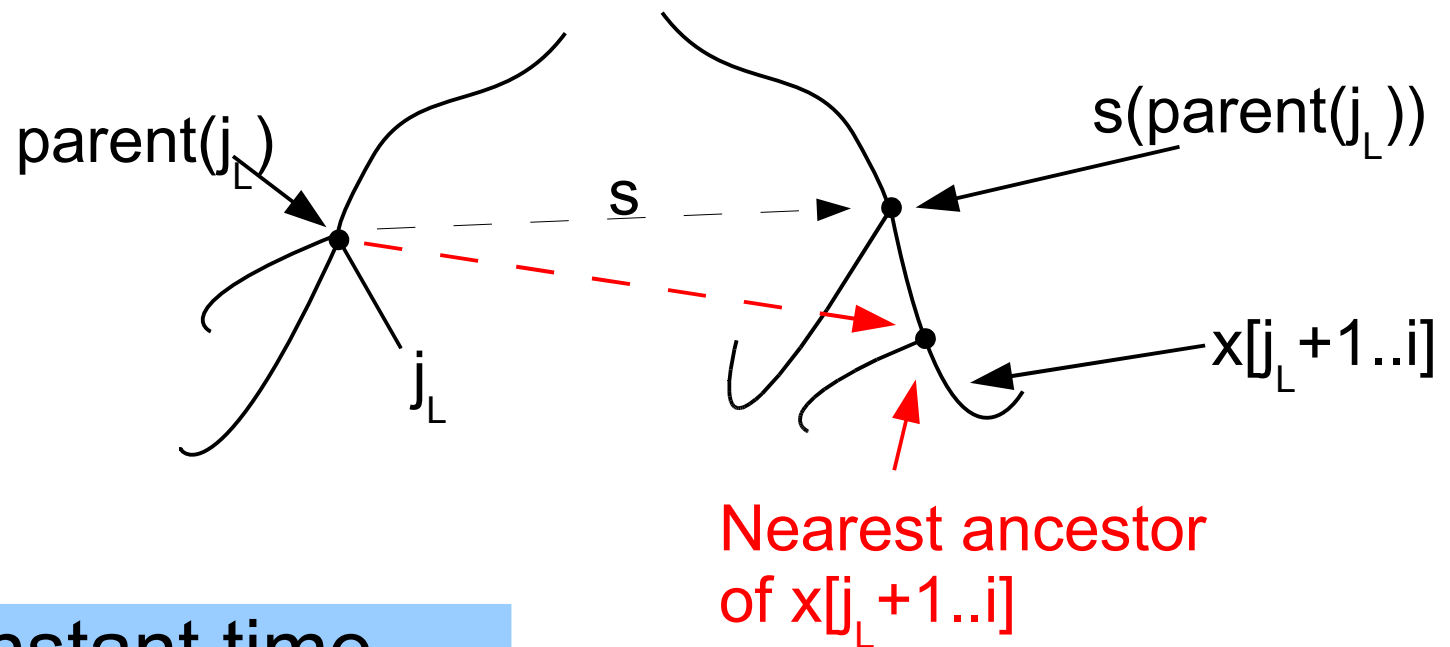
Hacking the suffix link

When searching for $x[j_L + 1..i]$, update $s(x[j_L..i])$ to point to the nearest ancestor of $x[j_L + 1..i]$



Hacking the suffix link

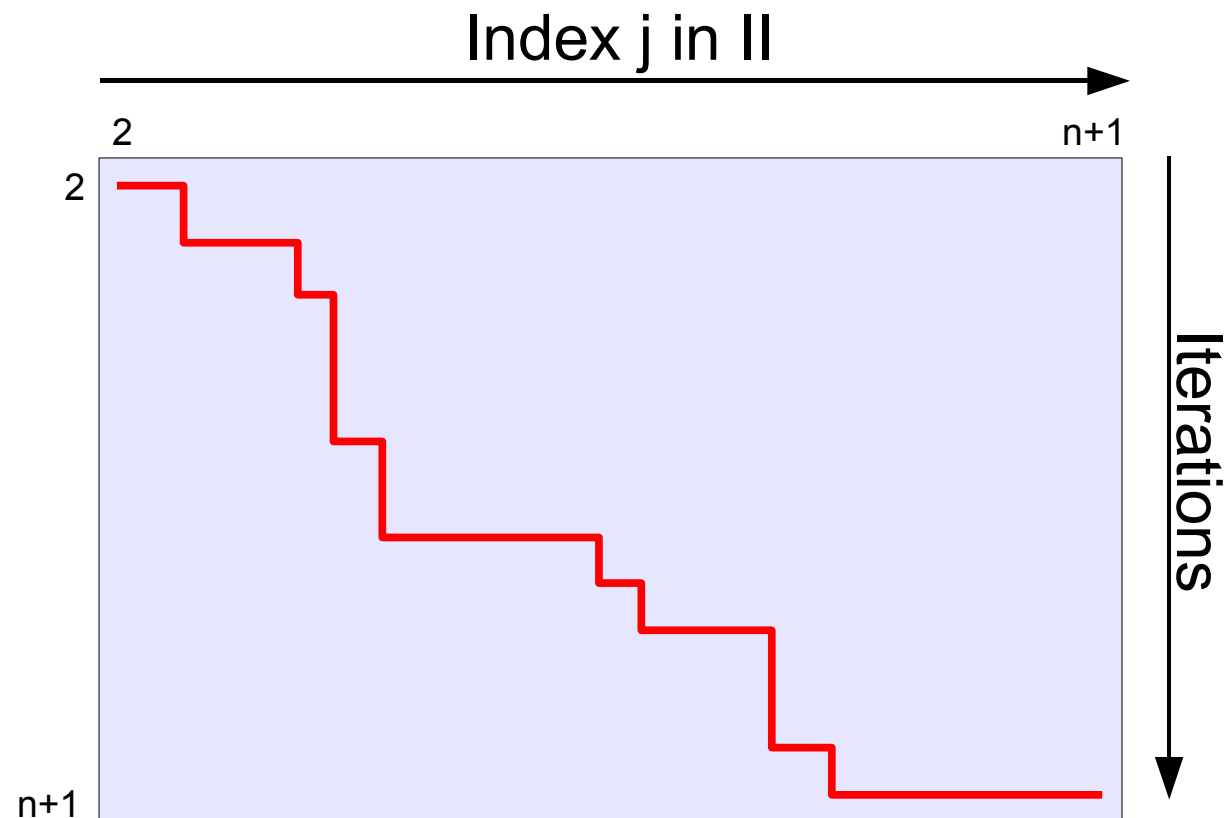
When searching for $x[j_L + 1..i]$, update $s(x[j_L..i])$ to point to the nearest ancestor of $x[j_L + 1..i]$



“Restart” in constant time

Searching time

- Vertical steps are paid for by the previous horizontal step (free restarting)
- Horizontal steps are total **fastscan** bounded by $O(n)$
- **Runtime $O(n)$**



Summary

- We have seen a new suffix tree construction algorithm
- Ukkonen's algorithm is an “online” algorithm:
 - As long as no suffix is a prefix of another, the intermediate trees are suffix trees
 - Generalized suffix trees can be built one string at a time