6. Linear-Time Construction of Suffix Trees McCreight's Algorithm

Kena Alexander 26/05/2015

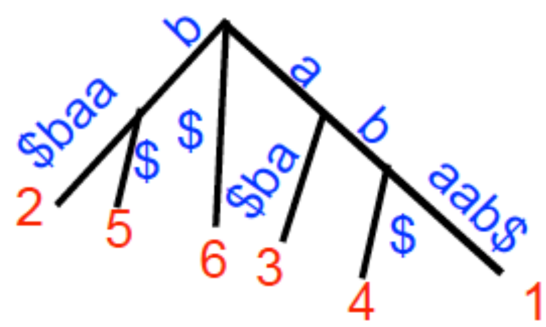
McCreight's Method

- **McCreight's method**
  - Linear-time construction algorithm
  - Space-saving improvement over Weiner's method.
  - Same space efficiency as Ukkonen's method

Suffix Trees

- **Definition**
  - A suffix tree of a sequence, x is a *compressed trie* of all suffixes of the sequence x$
  - $ is special end character

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



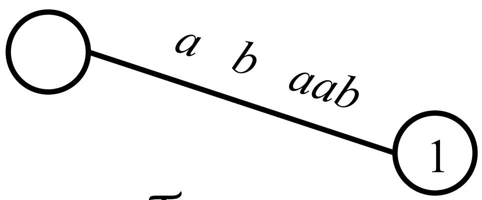
McCreight's Algorithm at a High Level

- **McCreight's algorithm**
  - Constructs the suffix tree T for m-length string S by
    - Inserting suffixes in order, one at a time starting from suffix one.
    - Iteratively for i=1,...,m+1 build tries, Ti,
    - where each in-between iteration is a Ti trie of sequences x$[1..$m$+1], x$[2..m+1], ..., x$[$i$..$m$+1]

McCreight's Algorithm

i	1	2	3	4	5	6
S	a	b	a	a	b	\$

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

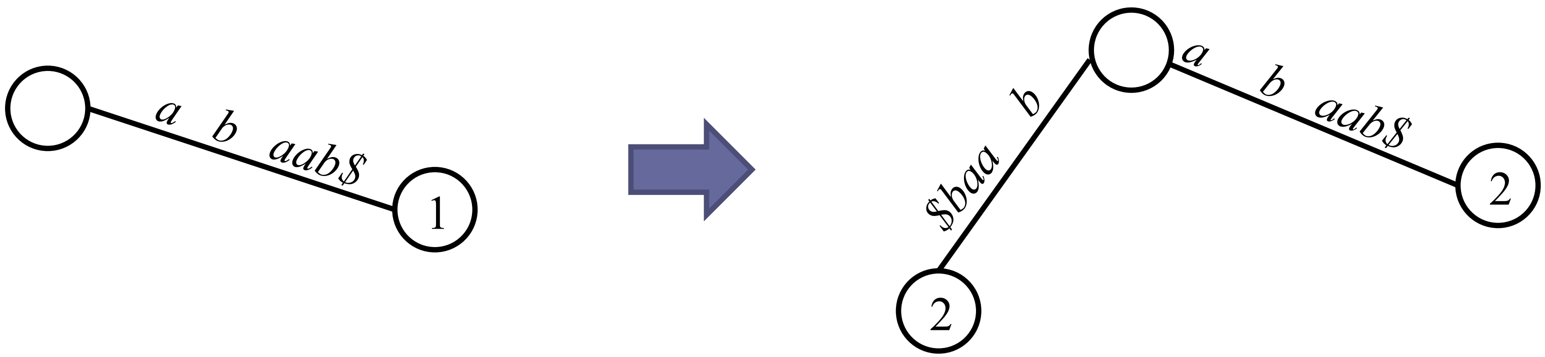


T_1

McCreight's Algorithm

- **Example**

i	1	2	3	4	5	6
S	a	b	a	a	b	\$

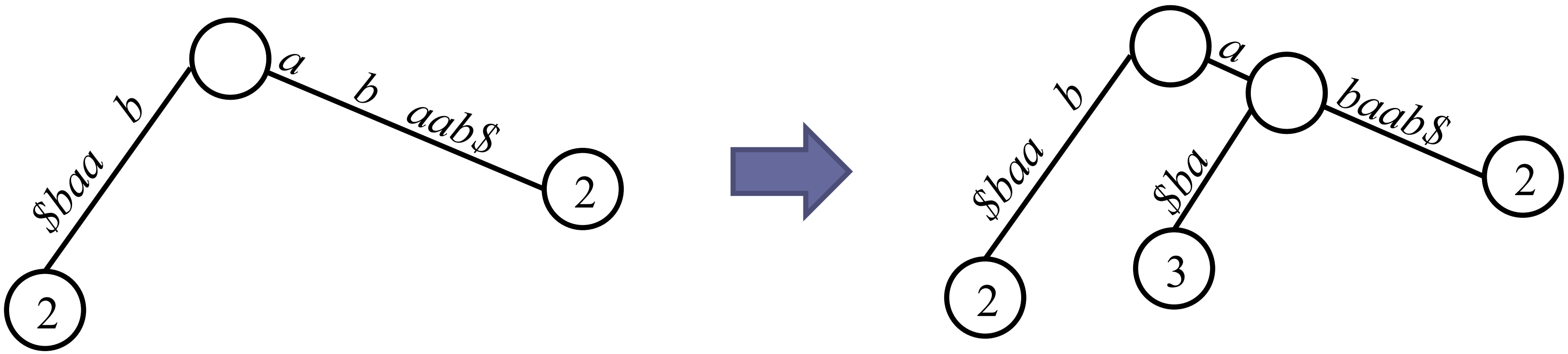


T_2

McCreight's Algorithm

- **Example**

i	1	2	3	4	5	6
S	a	b	a	a	b	\$

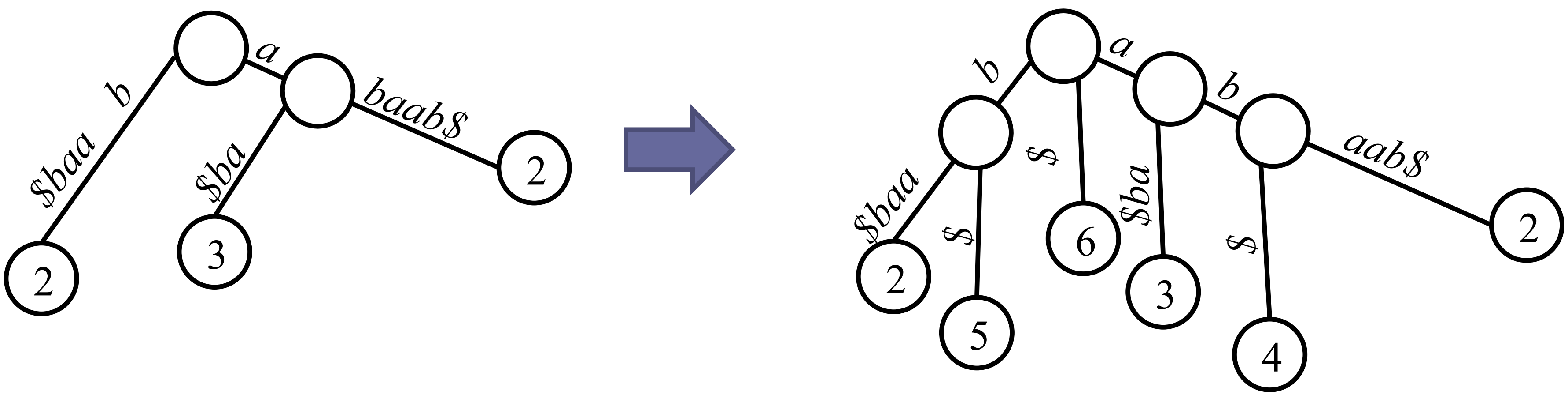


T_2 T_3

McCreight's Algorithm

i	1	2	3	4	5	6
S	a	b	a	a	b	\$

- **Example**



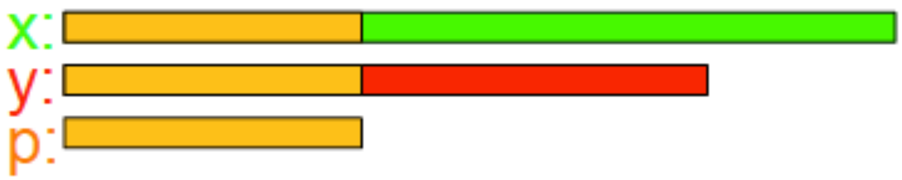
T_3 T_{m+1}

McCreight's Algorithm at a High Level

- **McCreight's algorithm**
  - Essential Trick
    - The essential trick is being clever in how we insert x[i..m]\$ into Ti so we don't spend O(m^2) time to build the tree.

Terminology

- **Common Prefix**
  - A common prefix of strings x and y is a string p that is a prefix of both.

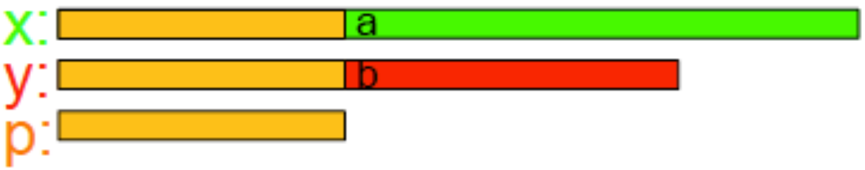


- **Example:**
  - x = abcabxabcd
  - y = abcdabxa
  - p = abc

Note: a, ab and abc are all common prefixes.

Terminology

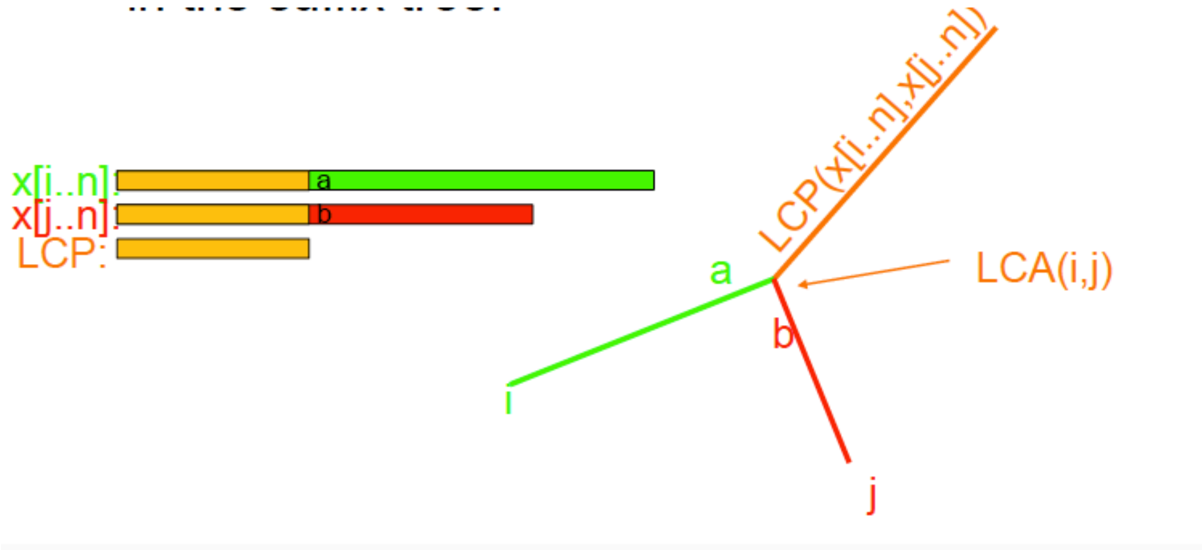
- **Longest Common Prefix**
  - The longest common prefix, p=LCP(x,y), is a prefix such that: $x[|p|+1] \neq y[|p|+1]$



- **Example**
  - x = abcabxabcd
  - y = abcdabxa
  - lcp = abc

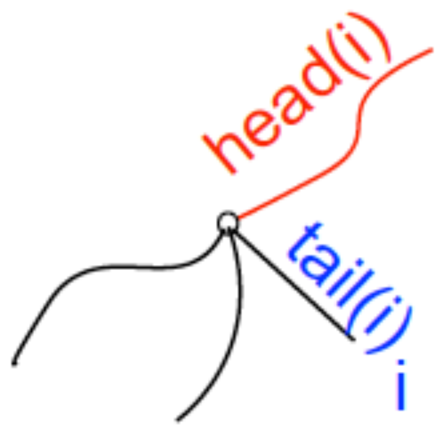
Terminology

- **Lowest Common Ancestor**
- For suffixes of x, x[i..n], x[j..n], their longest common prefix is their lowest common ancestor in the suffix tree:



Head and Tail

- **Head**
  - Let head(i) denote the longest LCP of x[i..n]$ and x[j..n]$ for all j < i
- **Tail**
  - Let tail(i) be the string such that x[i..n]$ = head(i)tail(i)

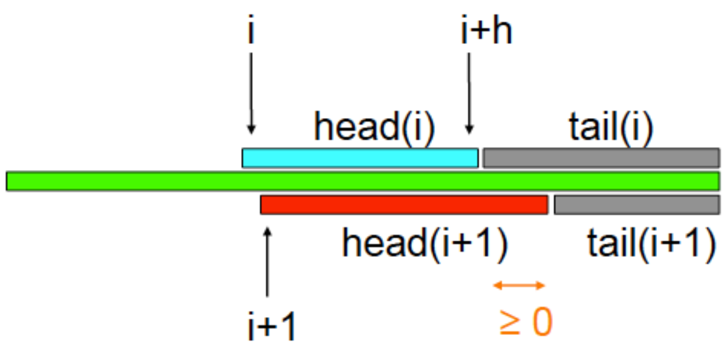


Head and Tail

- **Iteration**
  - Iteration *i* in McCreight's algorithm consist of
    - finding (or inserting) the node for head(i),
    - and appending tail(i))
  - The trick is a clever way of finding head(i)

Lemma

- **Lemma**
  - Let head(i) = x[*i..i+h*]. Then x[*i+1..i+h*] is a prefix of head(*i+1*)

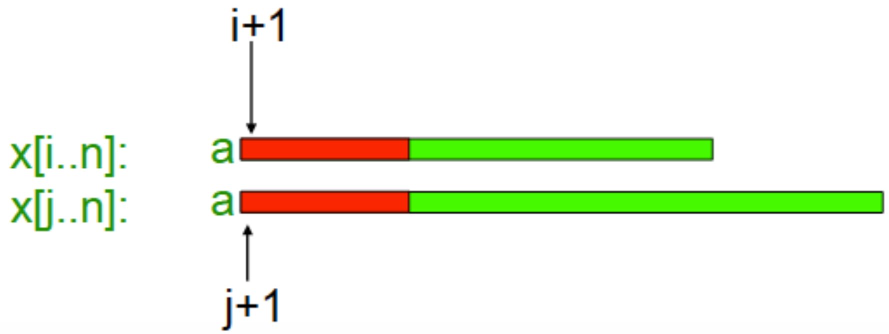


  - x = abcabxabcd
  - Head = x[1..3] = abc
  - Head+1 = bcd
  - x[2..3] = bc

Lemma

- **Proof**
  - Trivial for h=0 (head(i) empty), so assume h>0:
  - Let head(i) = ay: 

  - By def. exists j<i such that LCP(i,j)=ay
  - Thus suffix j+1 and i+1 share prefix y
  - Thus y is a prefix of LCP(i+1,j+1)
  - Thus y is a prefix of head(i+1)

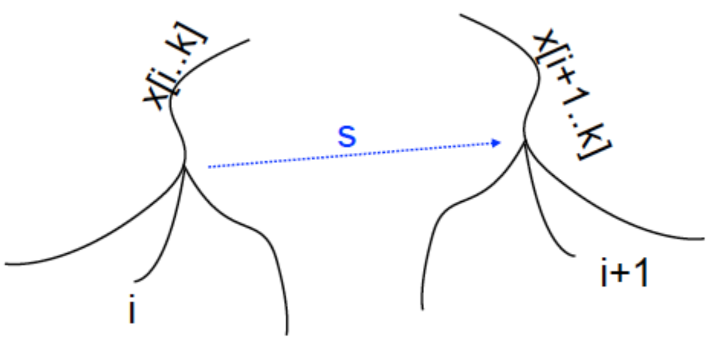


Suffix link

- **Definition of *suffix link***
  - Let u = $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring.

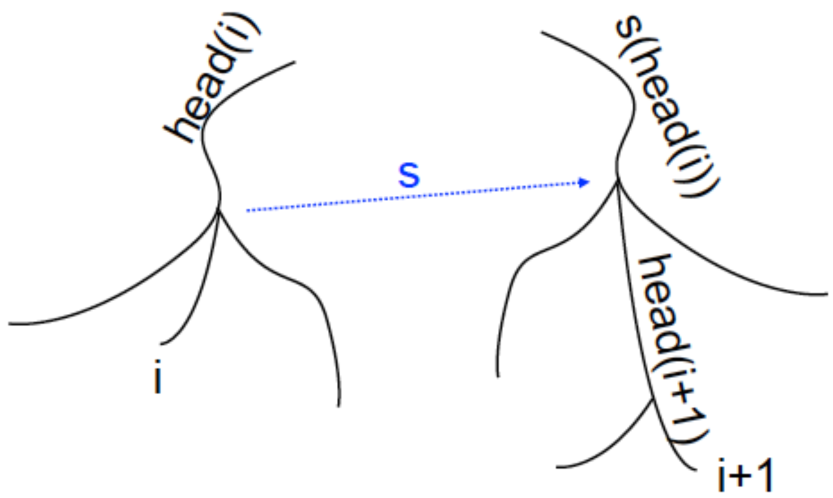
  - Then the suffix link s(u) = α for u = $x\alpha$ or
  - s(u) = “” (the root node) if u = “”

  - Suffix link is a pointer from a path labeled x[i..k] to a path labeled x[i+1..k]:



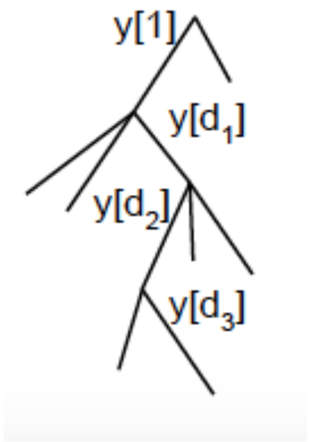
Corollary of Lemma

- s(head(i)) is a prefix of head($i+1$)
- Thus s(head(i)) is an ancestor of head($i+1$)



Slowscan and Fastscan

- Slowscan:
  - if we do not know if string y is in T*i*, we must search character by character
- Fastscan (Rescanning):
  - if we do know that y is in x, we can jump directly from node to node
    - At node u at (path-depth) d, follow the edge with label starting with y[d]
    - Continue until we reach the end of y.
    - We will either end on a
      - Node (if y is in Ti)
      - Edge (if y is a prefix of a string in Ti)
        - s(head(i)) is a prefix of head(i+1)
        - Thus s(head(i)) is an ancestor of head(i+1)



Sketch of McCreight's Algorithm

- Begin with the tree T1:
- For i=1,...,n, build tree Ti+1 satisfying:
  - Ti+1 is a compressed trie for x[j..n]$, $j \leq i+1$
  - All non-terminal nodes (with the possible exception of head(i)) have a s uffix link s(-)
- Each iteration must:
  - Add node i+1
  - Potentially add head(i+1)
  - Add tail(i+1)
  - Add suffix link head(i) → s(head(i))

iteration *i*

- Beginning of iteration
  - Let head(i)=uv
  - parent(head(i))=u
  - w=s(u)v=s(head(i))
- Rest of iteration
  - Move quickly to w, then search for head(i+1) starting there
    - By the invariant, s(parent(head(i))) and the suffix link exists;
    - by the lemma, w is an ancestor of head(i+1)

Observe: w is in T

- head(i) is a prefix of x[j..n] for some j < I
- Thus w is a prefix of x[j+1..n] for some j < I
  - i.e w is a prefix of some suffix $j \leq i$
  - i.e. w is in Ti
- Consequently: we can search for w from s(u) using fastscan!

w

- If w is a node
  - Update s(head(i)) := w
  - Then search for head(i+1) using slowscan

- If w is on an edge
  - If w is not a node, then all suffix j<i with prefix w agree on the next lett er
  - By definition of head(i) there is j<i such that suffix x[i..n] and x[j..n] di ffers after head(i)
    - x[i+1..n] must also disagree at that character
    - Thus head(i+1) must be w
- Add node w, update head(i):=w and set head(i+1)=w

McCreight's Algorithm

- Insert sufixes x[*1..m*] -> x[*m..m*]
- Step 1
  - Walk to the deepest node with a suffix link
- Step 2
  - Traverse down head(i+1) by looking at the first character of the edge
- Step 3
  - Traverse the rest of the suffix until we encounter an mismatch
  - Create a new node and using the remaining characters add an edge.

McCreight's Step 1

- Find the deepest node with a suffix link (lets call this head(i))
- head(i) can be divided into three substrings
  - $x\alpha\beta$
  - $x\alpha$ is the path label of the node y. if no such node exists $x\alpha$ = “”
  - If such a node exists then x is the first character and α is the rest of the characters in the string.

- If we follow the suffix link s($\boldsymbol{x\alpha}$) we get to w whose label is α
- The rest of the string will be β

McCreight's Step 2

- Rescanning phase {Fastscan}
- Rescan the characters of β from node w
  - (analyze only the first characters and jump from node to node until a mismatch or no nodes to jump)
- We know that β is part of the tree therefore
  - The number of steps required to rescan is proportaional to the intermedi ate nodes visited while traversing β
- If there is no node at this point, create a new node, d
- Update the suffix link s(head(i)) to point to d

McCreight's Step 3

- Scanning phase {Slowscan}
- Let the remaining part of head(i) = tail(i). since we do not kno w the length of this we have to analyze each character down th e subtree.
- If we get a mismatch then
  - Create a new node and let the remainder of characters be the leaf from t his node.

Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

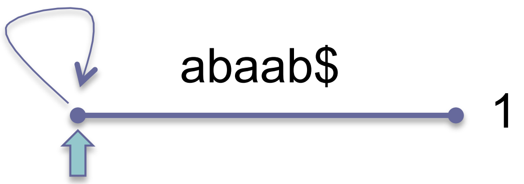
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

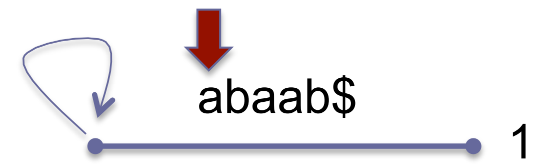
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

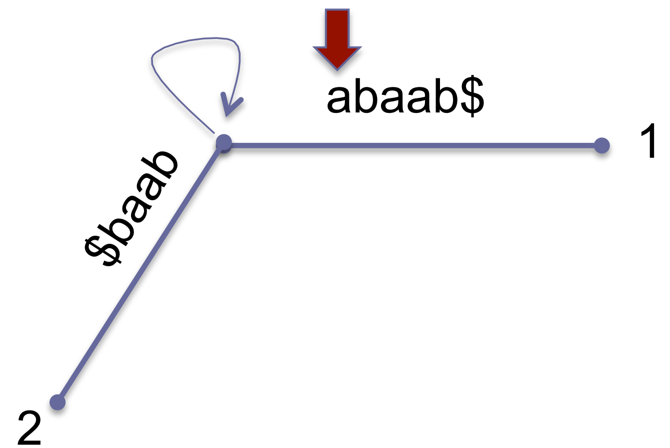
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

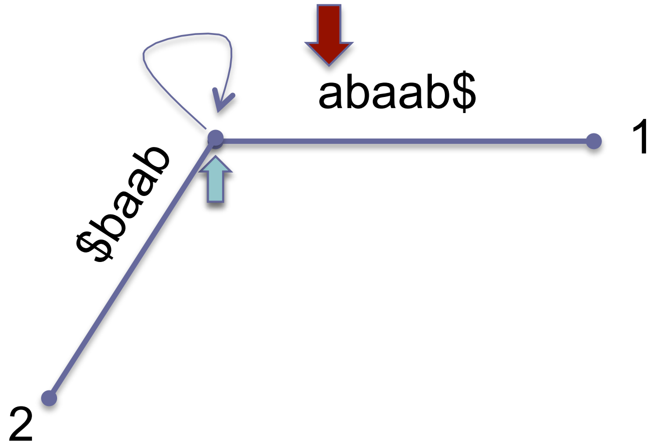
Algorithm McCreight;

Step1:
 Find the deepest node with a suffix link
 Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
 analyze the first characters of head(i) comparing to β
 jumping nodes if match.
 if there is no node then
  create a node = d
 update the suffix link suff(head(i)) → d

Step3: {Slowscan}
 let the remainder of head(i) be tail(i)
 compare each character of tail(i) to rest of characters of β
 until a mismatch.
 if we get a mismatch
  create a new node.
  let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

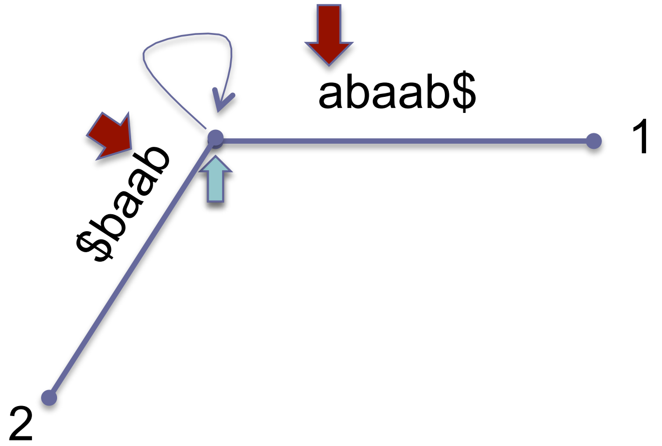
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

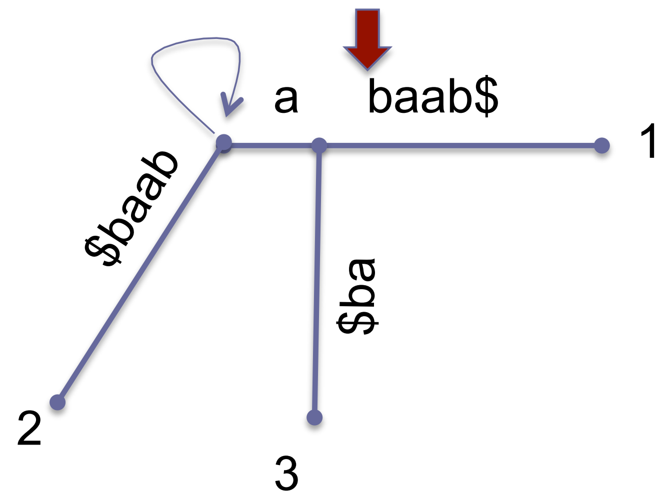
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

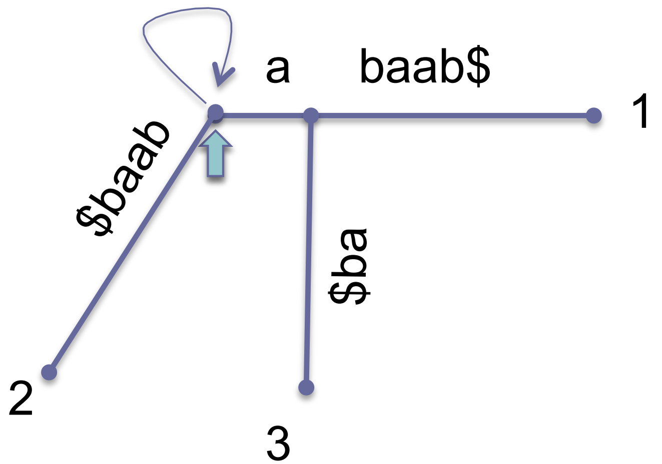
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

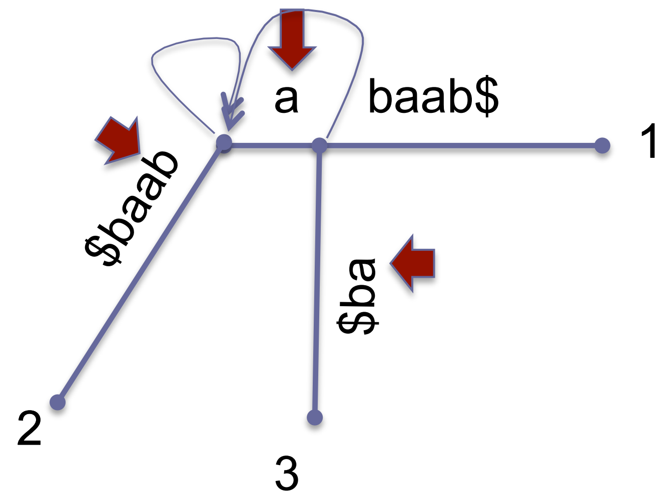
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

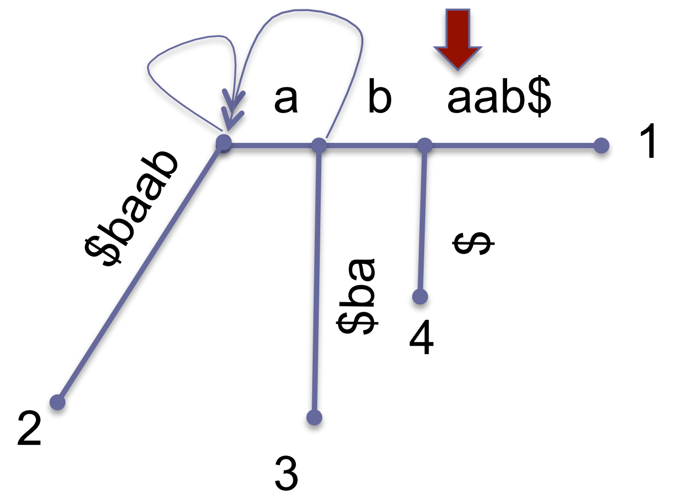
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

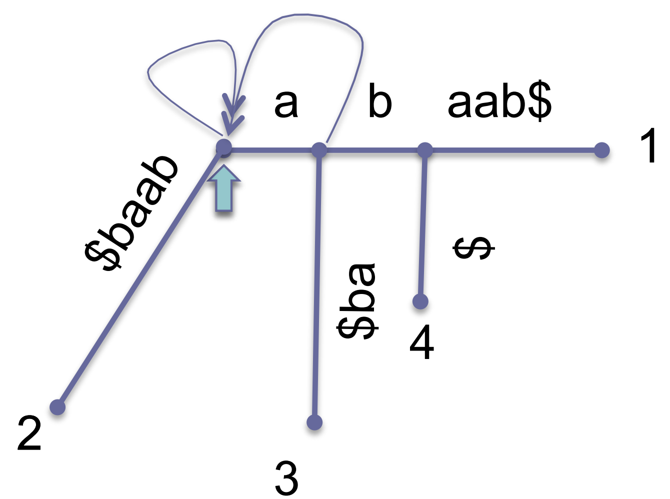
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

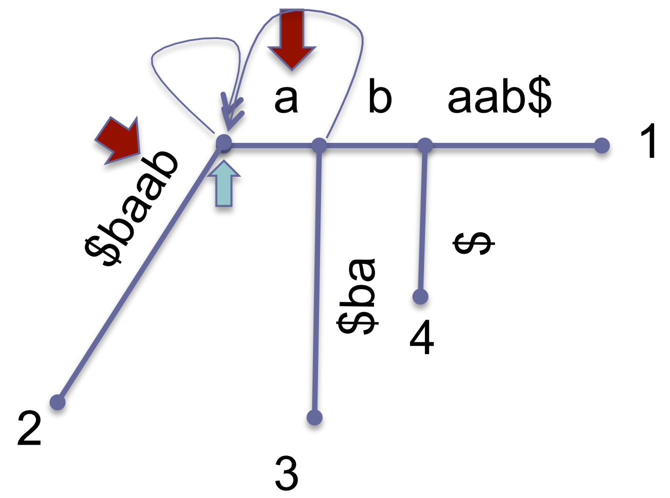
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

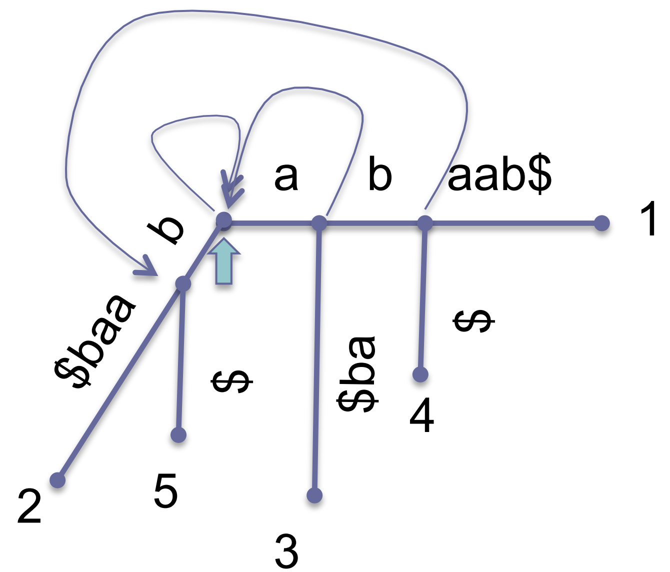
Algorithm McCreight;

Step1:
Find the deepest node with a suffix link
Follow the suffix link suff() to get to head(i) = αβ

Step2: {Fastscan}
analyze the first characters of head(i) comparing to β
jumping nodes if match.
if there is no node then
create a node = d
update the suffix link suff(head(i)) → d

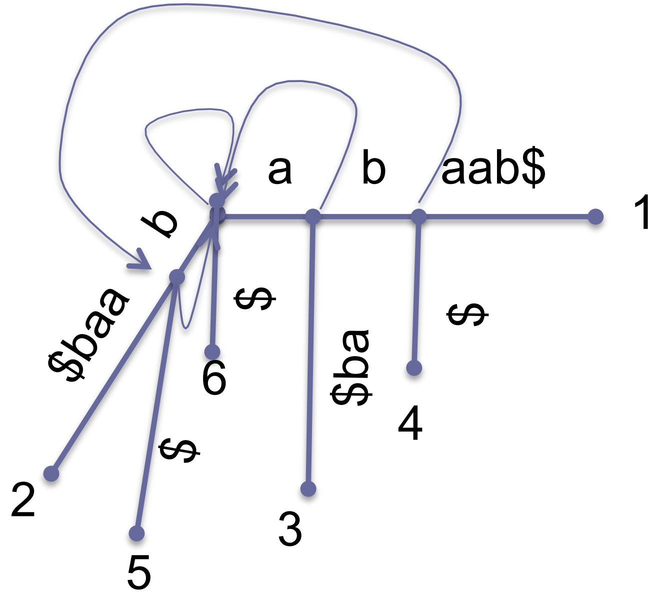
Step3: {Slowscan}
let the remainder of head(i) be tail(i)
compare each character of tail(i) to rest of characters of β
until a mismatch.
if we get a mismatch
create a new node.
let the remaining characters be the leaf.

end.



Example

i	1	2	3	4	5	6
S	a	b	a	a	b	$

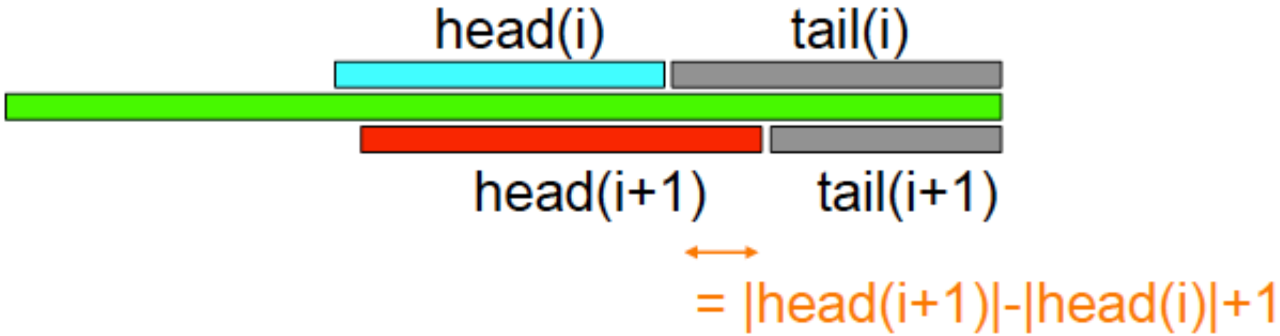


Time analysis

- Everything but searching is done in constant time per suffix,
- The running time is O(n + “slowscan” + “fastscan”).

Time analysis

- Slowscan
  - We use slowscan to find head(i+1) from w=s(head(i)),
  - The complexity of one run of *slowscan* at stage *i* is proportional to |head(i+1)|-|head(i)|+1
  -



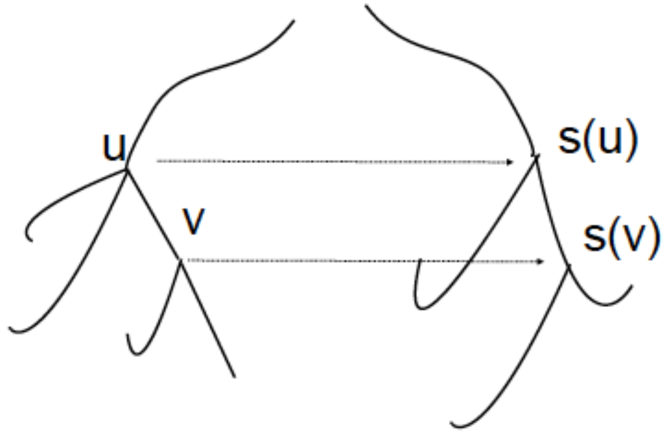
Time analysis

- Fastscan
  - Fastscan uses time proportional to the number of nodes it processes.
  - *If we define d(v) as the depth of node v*
    - Fastscan increases the node depth
    - Following parent and suffix pointers decreases the node depth

- Time usage of fastscan is bounded by the total depth-increase (amortized analysis)

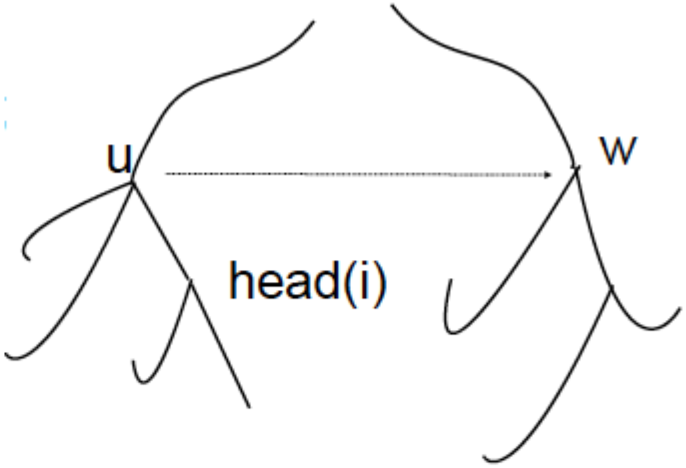
Time analysis

- Proposition
  - $d(v) \leq d(s(v))+1$
- Proof:
  - For any ancestor u of v, s(u) is an ancestor of s(v)
  - Except for the empty prefix and the single letter prefix of v, the s(u)'s are different



Time analysis

- Corollary
  - In each step, before calling fastscan, we decrease the depth by at most 2:
    - d(u) = d(head(i))-1;
    - d(w) $\geq$ d(u)-1
  - The total decrease is thus 2n



Time analysis

- The time usage of fastscan is bounded by n plus the total decrea se of depth,
  - i.e. the time usage of fastscan is O(n)

Summary

- We iteratively build tries of suffixes of x.
- Using suffix links and fastscan we can quickly find where to in sert the next suffix in our current trie.
- By amortized analysis, the total running time becomes linear.