Applied Databases

Lecture 15
Suffix Trees and Suffix Arrays

Sebastian Maneth

University of Edinburgh - March 13th, 2017

Horspool

Match **RIGHT-TO-LEFT**

T =	a	b	a	b	a	a	b	a	b	c	a	b	a	b	a	b	a	a	b	c
	a	b	a	b	c															

R(a) = 2

R(c) = 5
R(b) = 1

Horspool
If mismatch and P[m] aligned to z in T, shift pattern to the RIGHT by R(z).

Question → can you do Horspool on Unicode (e.g. UTF-8)??

variable length encoding

UTF-8

Maps a unicode character into **1, 2, 3, or 4 bytes**.

Unicode range	Byte sequence
U+000000 → U+00007F	0□□□□□□□
U+000080 → U+0007FF	110□□□□□ 10□□□□□□
U+000800 → U+00FFFF	1110□□□□ 10□□□□□□ 10□□□□□□
U+010000 → U+10FFFF	11110□□□ 10□□□□□□ 10□□□□□□ 10□□□□□□

Spare bits (□) are filled from right to left. Pad to the left with 0-bits.

E.g. U+00A9 in UTF-8 is 11000010 10101001
U+2260 in UTF-8 is 11100010 10001001 10100000

Question → can you do Horspool on Unicode (e.g. UTF-8)??

variable length encoding

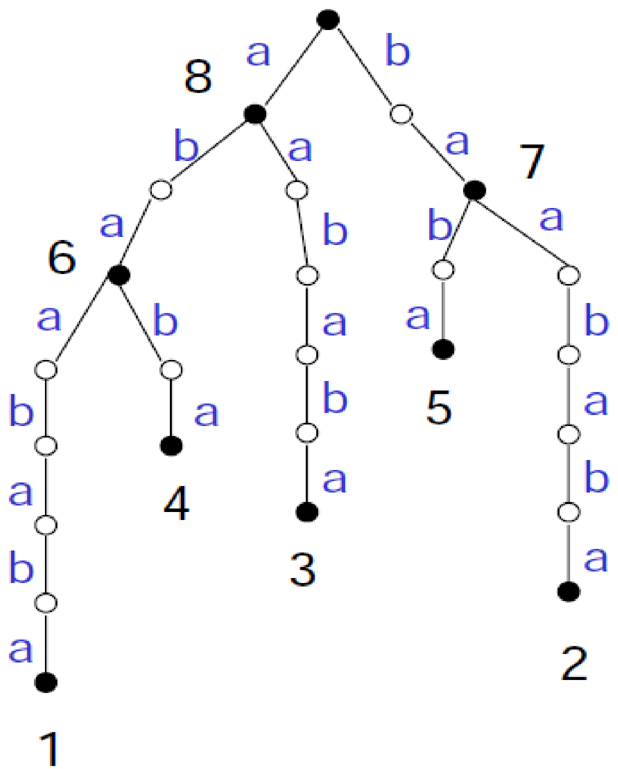
→ try to solve it yourself
→ possibly consult Patent US8819045

Outline

1. Suffix Tree
2. Suffix Tree Construction
3. Applications of Suffix Trees
4. Suffix Array

1. Suffix Tree

```
     12345678
T =  abaababa
```

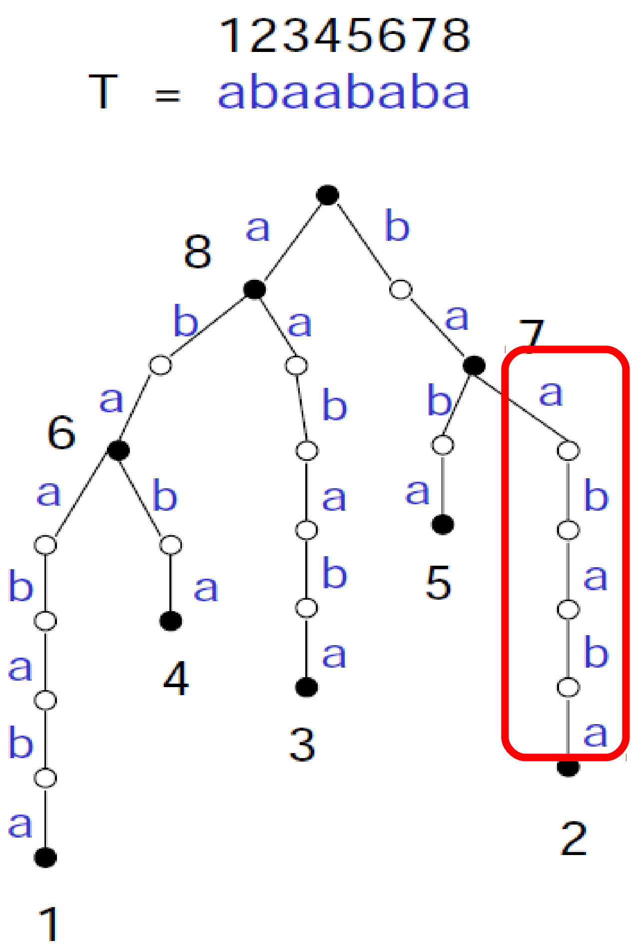


Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

New Idea

→ collapse paths of white nodes!

1. Suffix Tree



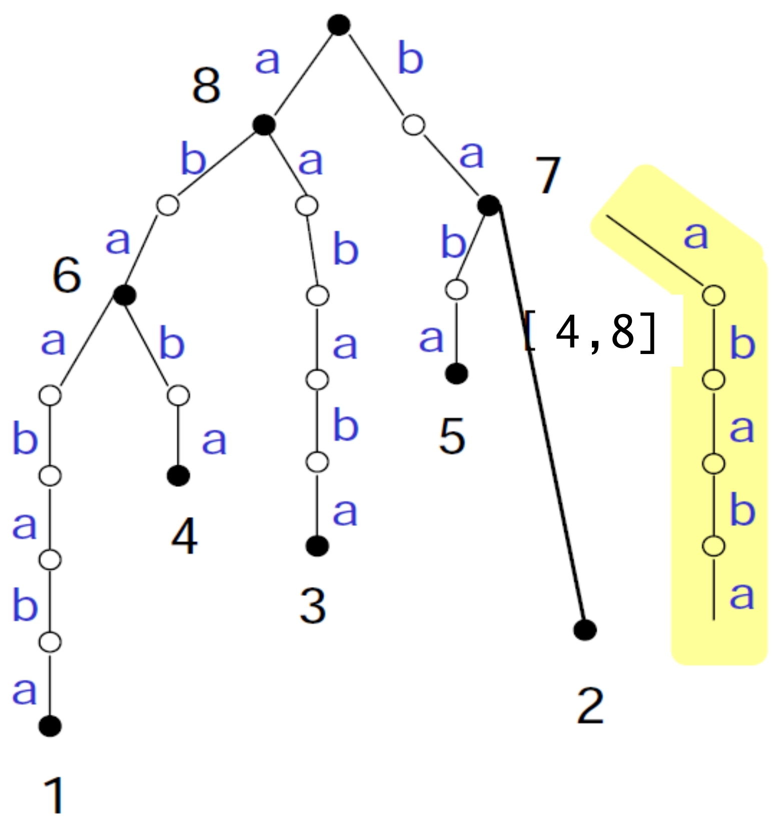
Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

New Idea

→ collapse paths of white nodes!

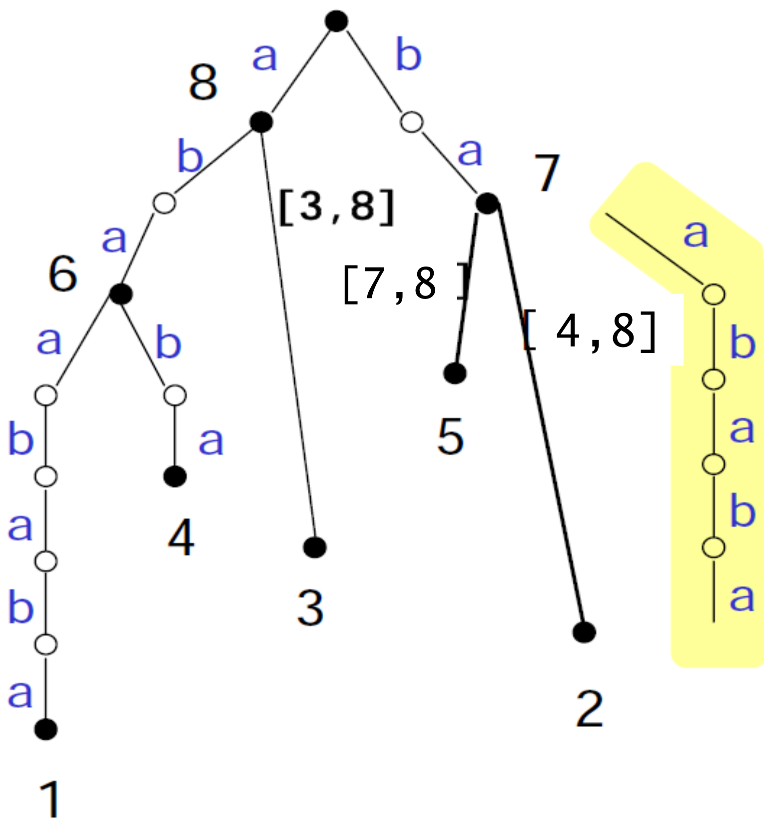
1. Suffix Tree

12345678
T = abaababa



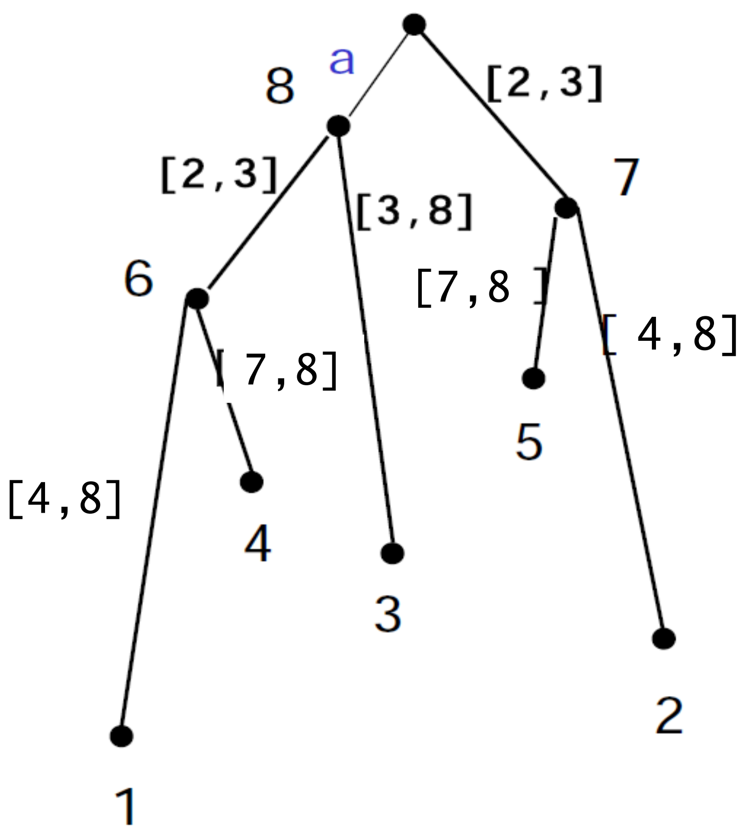
1. Suffix Tree

```
    12345678
T = abaababa
```



Suffix Tree

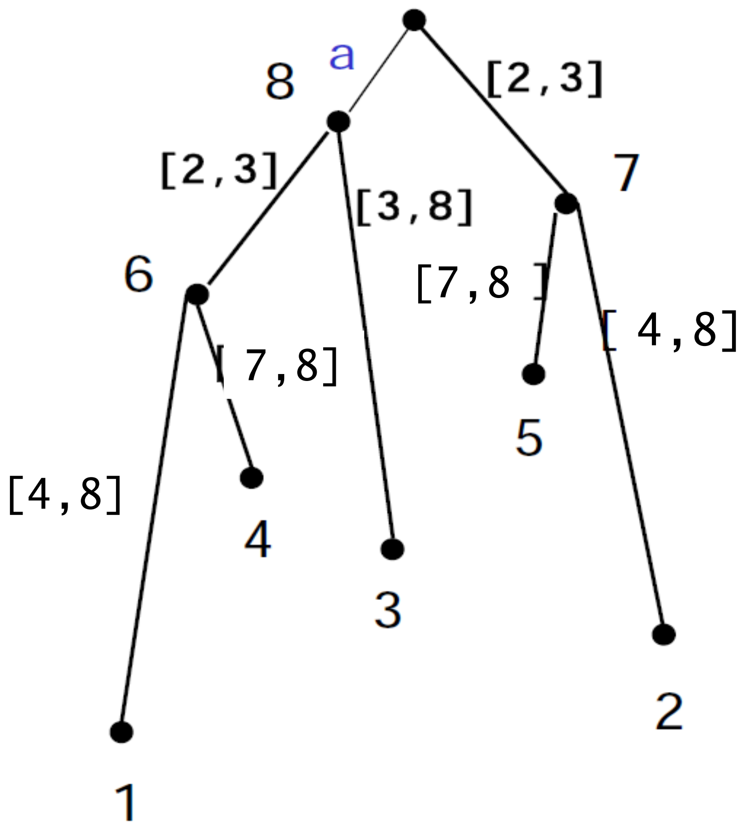
```
    12345678
T = abaababa
```



Suffix Tree of T

Suffix Tree

```
    12345678
T = abaababa
```

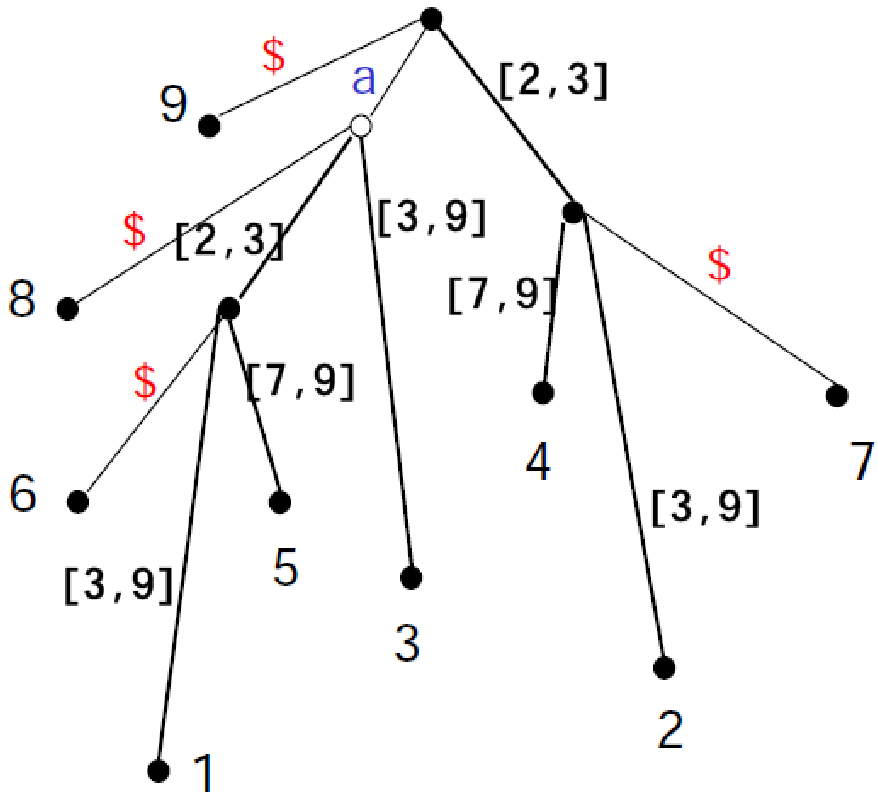


Suffix Tree of T

→ how many nodes (at most)
In the suffix tree of T?

Suffix Tree

```
   123456789
T = abaababa$
```



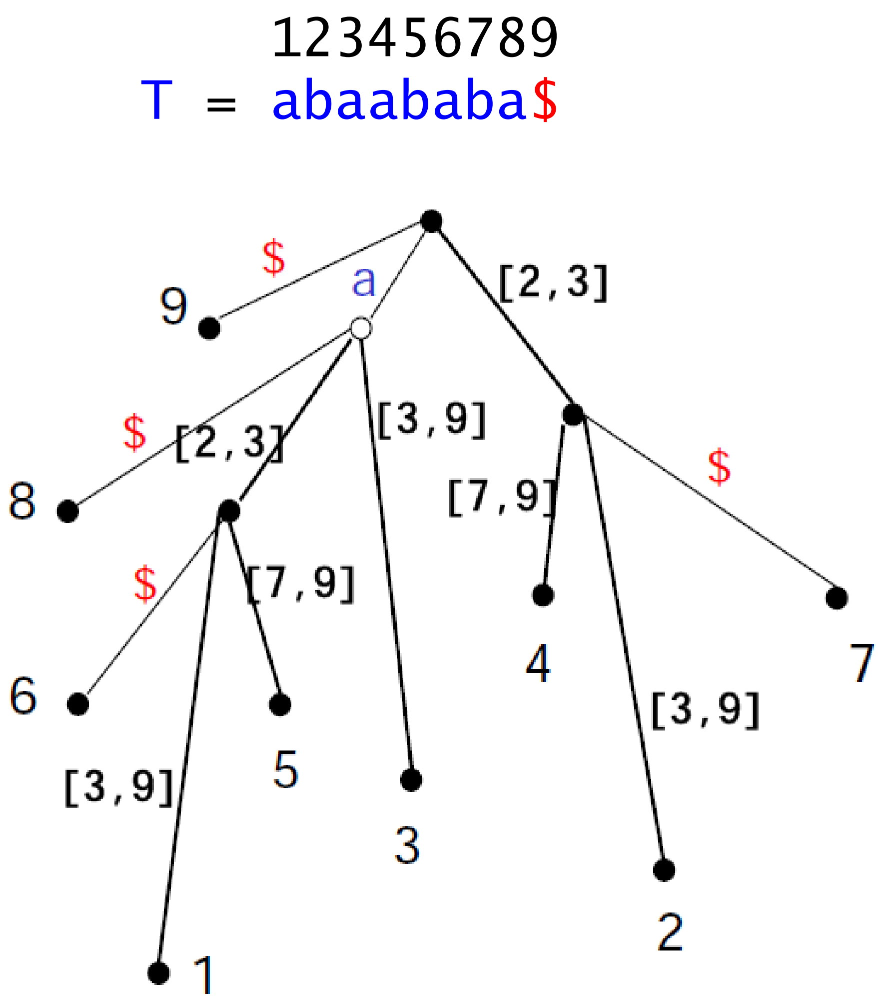
→ add end marker "$"

→ one-to-one correspondence of leaves to suffixes

→ a tree with n+1 leaves (and no nodes with only one child) has <= 2n+1 nodes!

> **Lemma**
> Size of suffix tree for "T$" is linear in n=|T|, i.e., in O(n).

Suffix Tree



→ add end marker "$"

→ one-to-one correspondence of leaves to suffixes

→ a tree with n+1 leaves (and no nodes with only one child) has <= 2n+1 nodes!

Lemma
Size of suffix tree for "T$" is linear in n=|T|, i.e., in O(n).

→ search time still O(|P|), as for suffix trie!
→ perfect data structure for our task!

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

But, rather complex construction algorithms

→ Weiner 1973 [Knuth: “Algorithm of the year 1973”]

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973 [Knuth: "Algorithm of the year 1973"]

→ McCreight 1976 Simplification of Weiner's algorithm

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973 [Knuth: "Algorithm of the year 1973"]

→ McCreight 1976 Simplification of Weiner's algorithm

→ Ukkonen 1995 ← first **online** algorithm!
→ T may come from a stream
→ build suffix tree for TT' from suffix tree for T
→ huge breakthrough!!

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

Linear time only for *constant-size alphabets*!
Otherwise, O(n log n)

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

Linear time only for *constant-size alphabets*!
Otherwise, O(n log n)

→ Farach 1997

Linear time for **any integer alphabet**,
drawn from a polynomial range

→ again a big breakthrough

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

Linear time only for *constant-size alphabets*!
Otherwise, O(n log n)

→ Farach 1997

→ Kurtz 1999

Practical algorithm
13–15n Bytes space requirement.

(→ e.g. McCreight: 28n Bytes)

2. Suffix Tree Construction

Good news:
Suffix tree *can* be constructed in linear time!

Complex construction algorithms

→ Weiner 1973

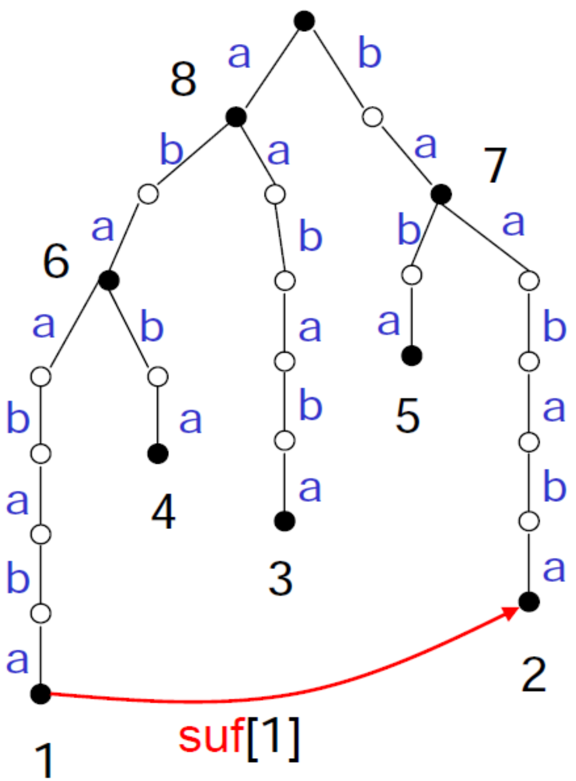
→ McCreight 1976

→ Ukkonen 1995

→ Farach 1997

Suffix Link

12345678
T = abaababa

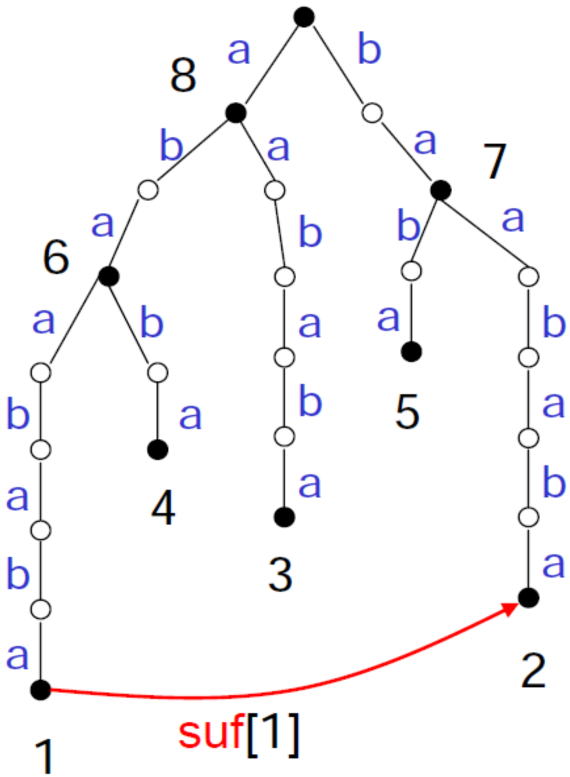


Definition

If x=ay is the string corresponding to a node u in the ST then the suffix link suf[u] is the node v corresponding to y in ST.

Suffix Link

12345678
T = abaababa

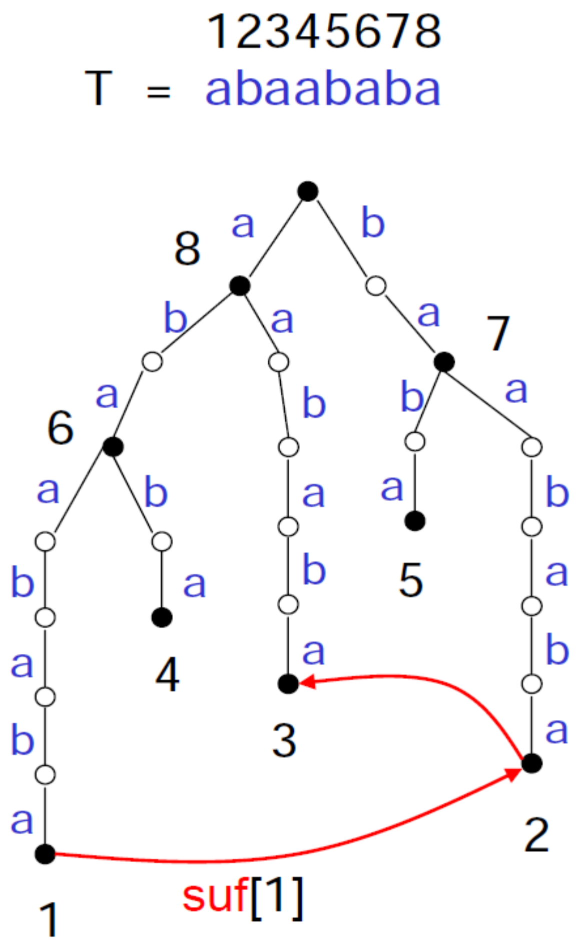


Definition

If x=ay is the string corresponding to a node u in the ST then the suffix link suf[u] is the node v corresponding to y in ST.

Where is the suffix link of node “2”?

Suffix Link



Definition

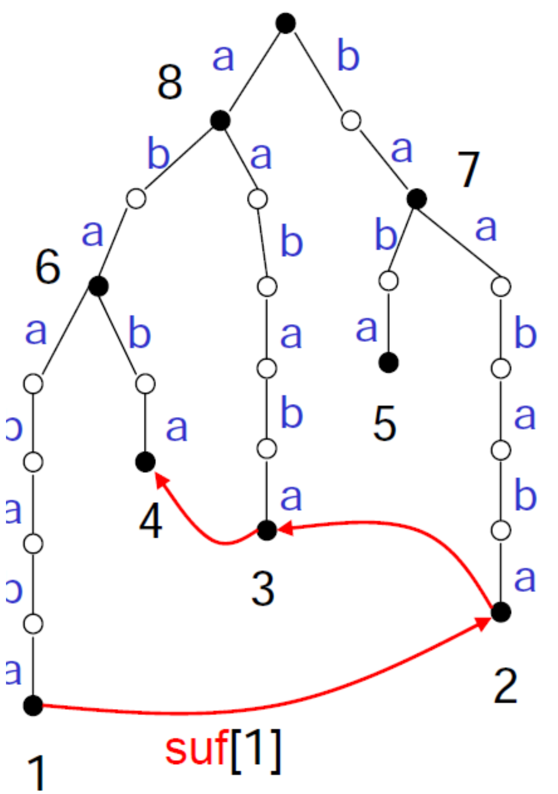
If x=ay is the string corresponding to a node u in the ST then the suffix link suf[u] is the node v corresponding to y in ST.

Where is the suffix link of node "2"?

- ● essential node
- ○ non-essential node

Suffix Link

    12345678
T = abaababa



● essential node

○ non-essential node

Definition

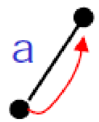
If x=ay is the string corresponding to a node u in the ST then the suffix link suf[u] is the node v corresponding to y in ST.

Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

b = T[current]

From v, follow (k times) suffix links (to u) until child(u, b) is defined.

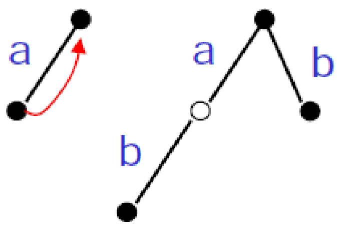
Create b-sons for v, suf[v], $suf^2[v]$, …, $suf^{k-1}[v]$

If there is no such u, create b-sons for all of them, up to k

Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

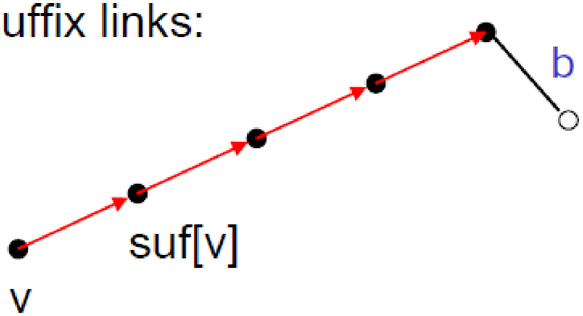
b = T[current]

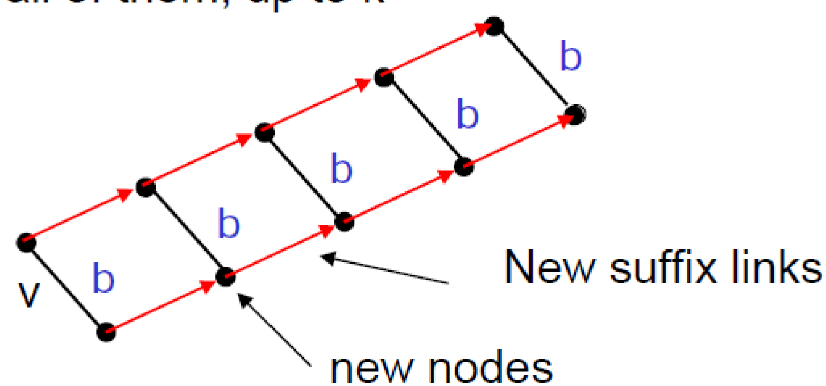
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

New suffix links:

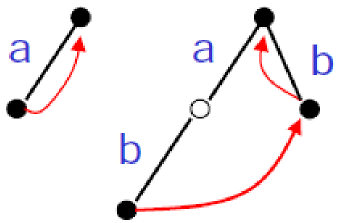




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

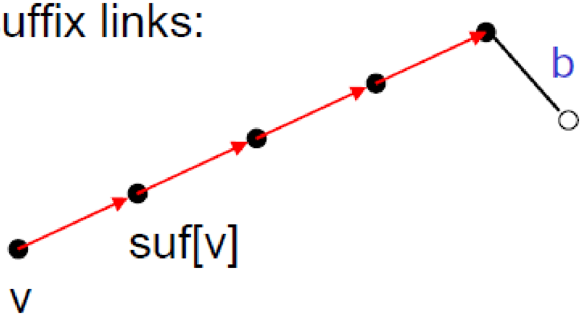
b = T[current]

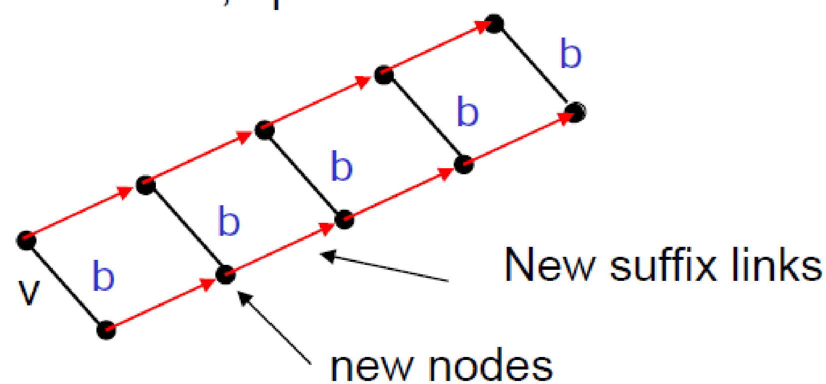
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

New suffix links:

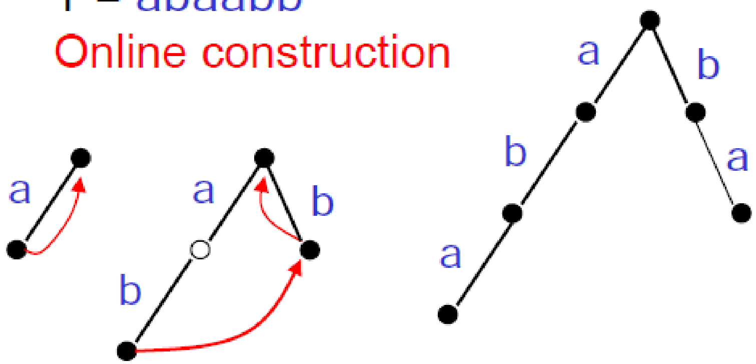




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

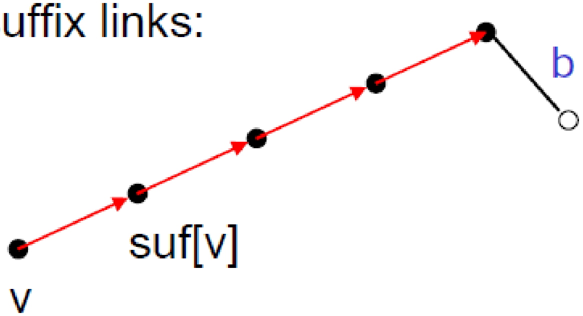
b = T[current]

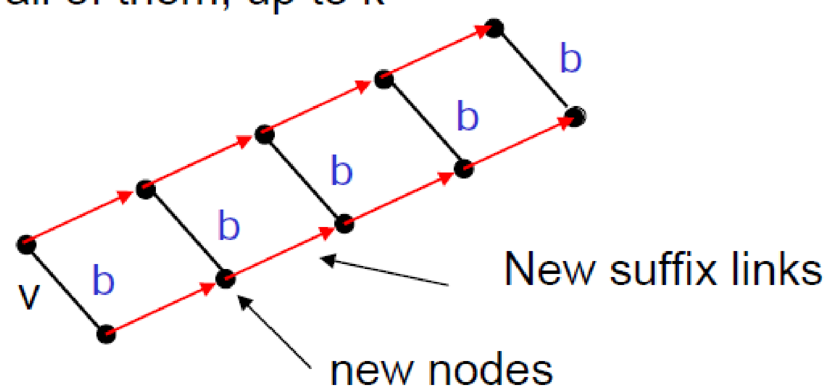
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], $suf^2[v]$, …, $suf^{k-1}[v]$

If there is no such u, create b-sons for all of them, up to k

New suffix links:

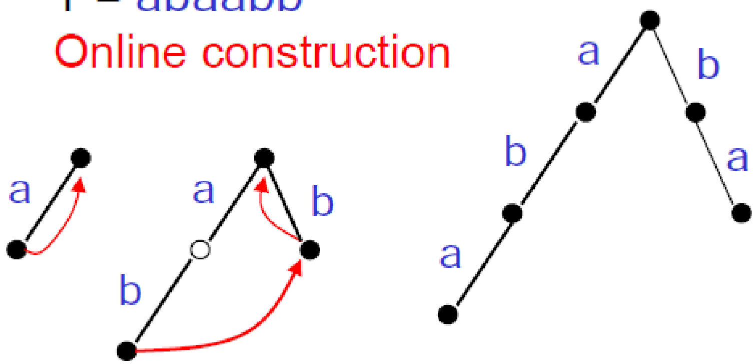




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



What are the new suffix links?

v = lowest leaf in tree

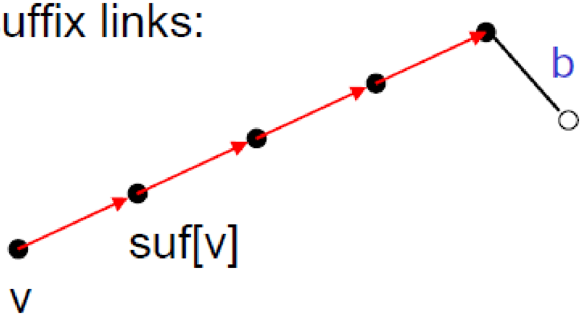
b = T[current]

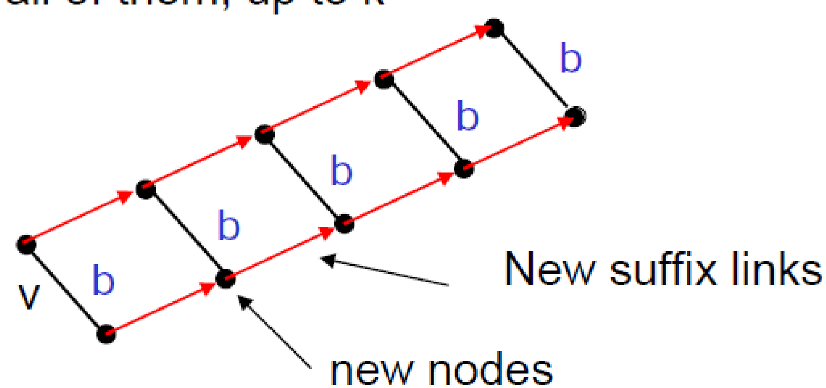
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

New suffix links:

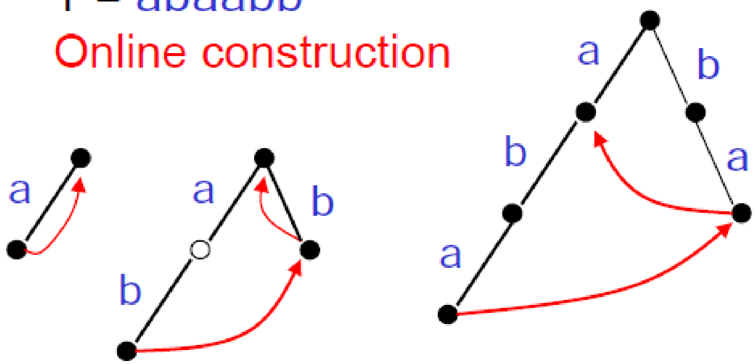




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

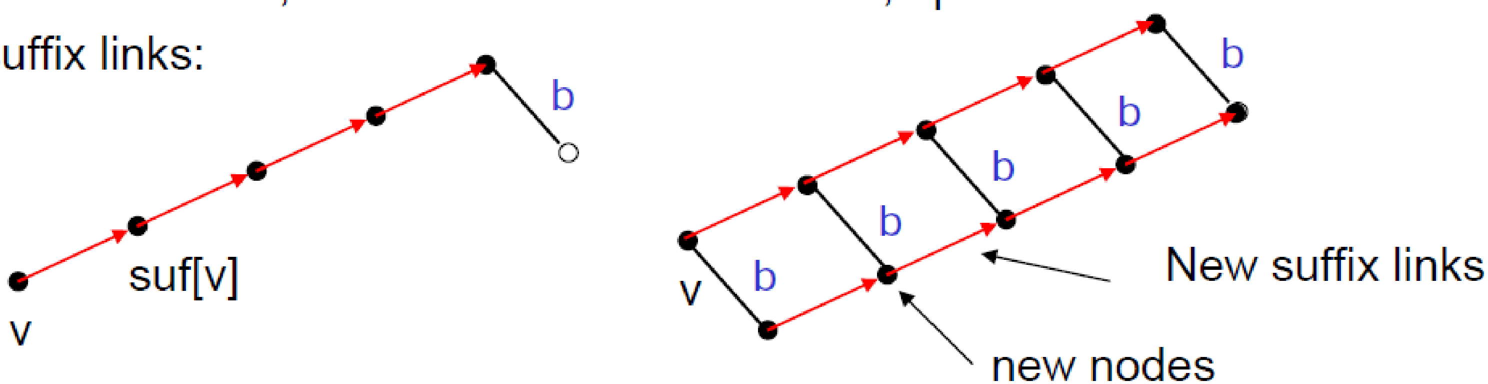
b = T[current]

From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

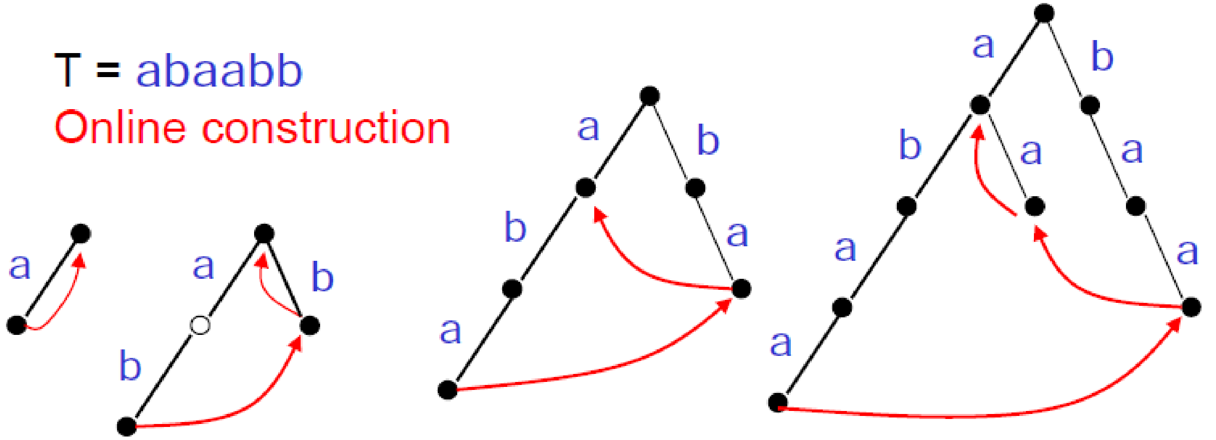
New suffix links:



Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

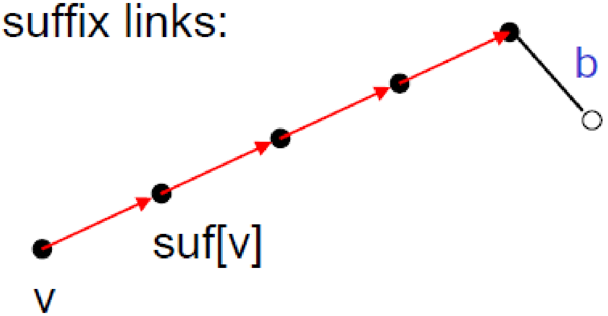
b = T[current]

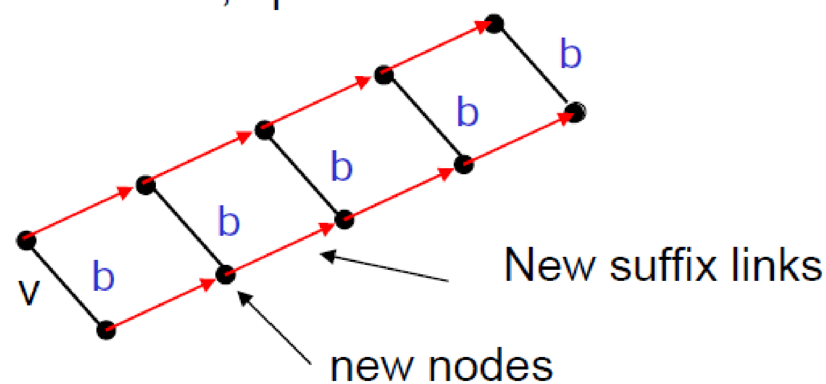
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

New suffix links:

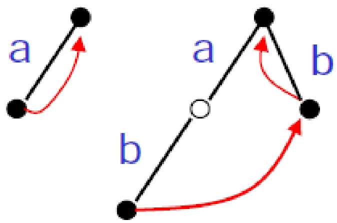
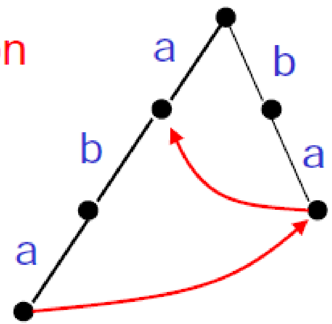
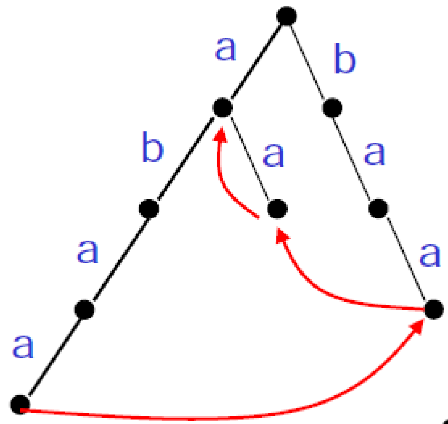
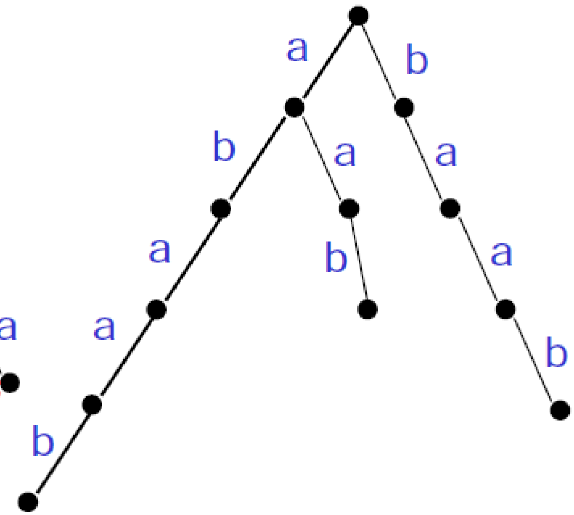




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction

v = lowest leaf in tree

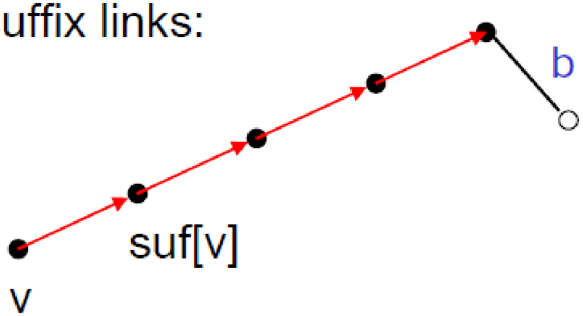
b = T[current]

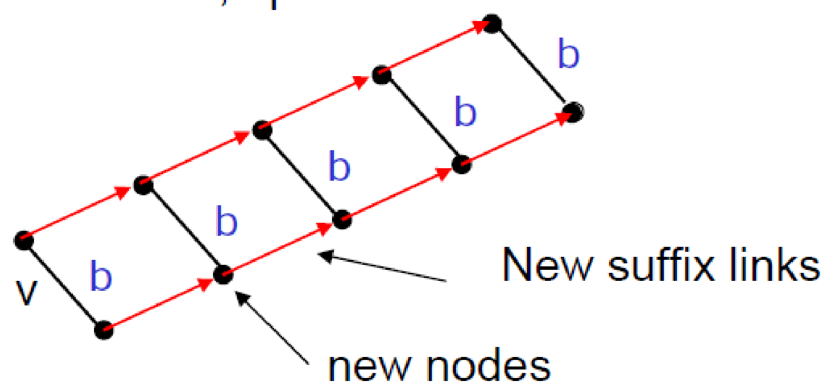
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], $suf^2[v]$, …, $suf^{k-1}[v]$

If there is no such u, create b-sons for all of them, up to k

New suffix links:

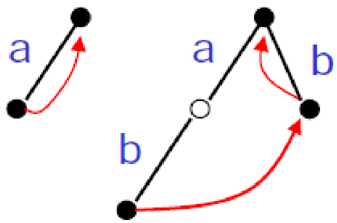


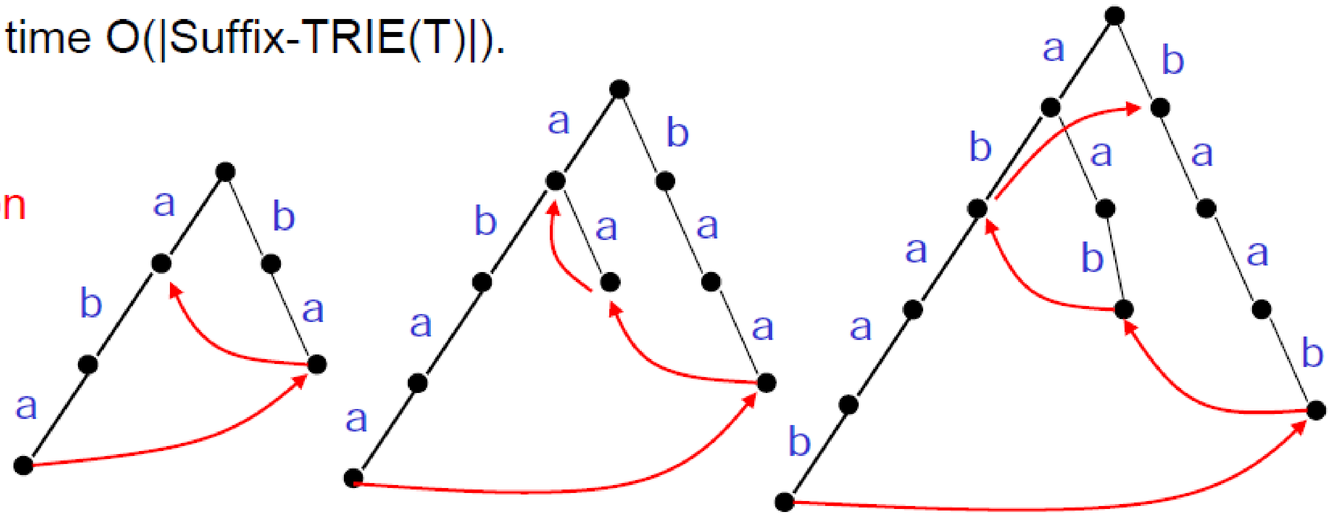


Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction





v = lowest leaf in tree

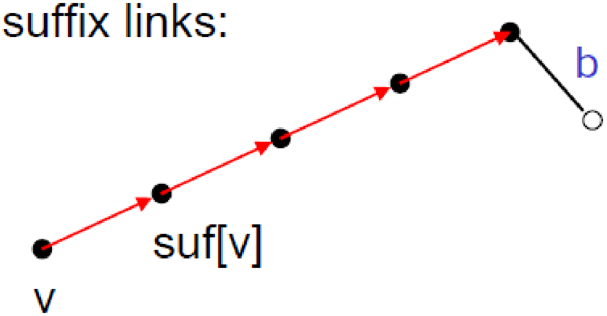
b = T[current]

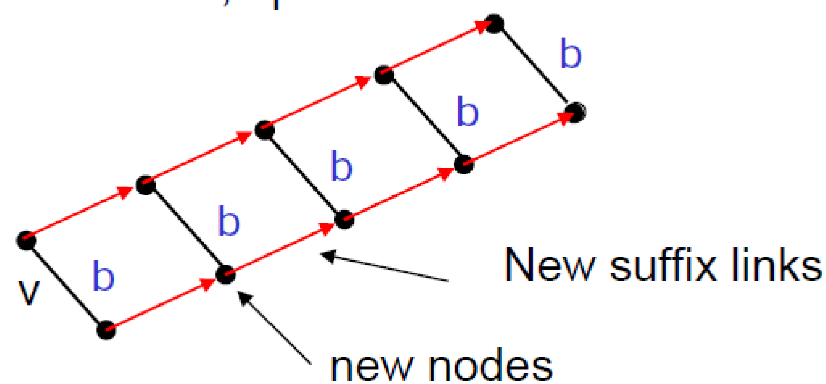
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

Create b-sons for v, suf[v], suf^2[v], …, suf^{k-1}[v]

If there is no such u, create b-sons for all of them, up to k

New suffix links:

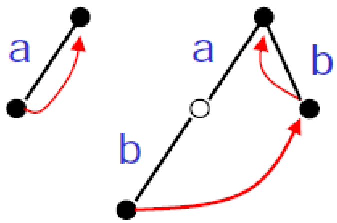


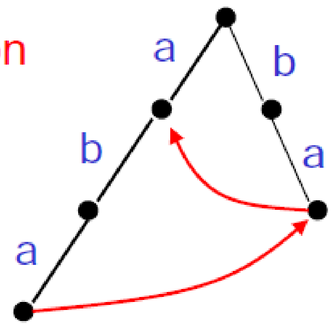


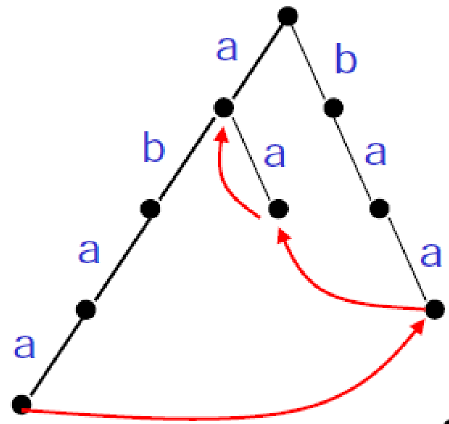
Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

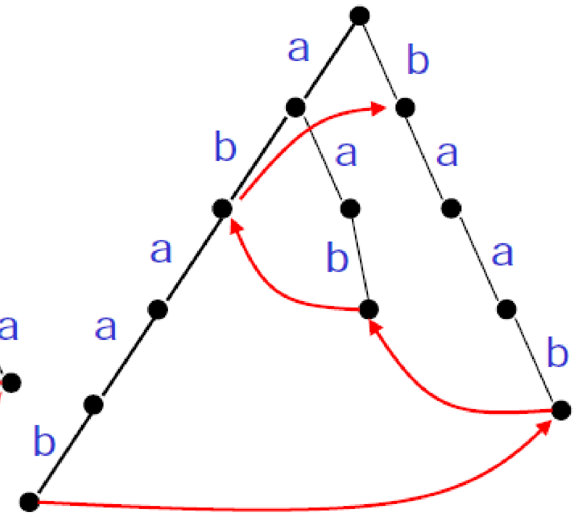
T = abaabb

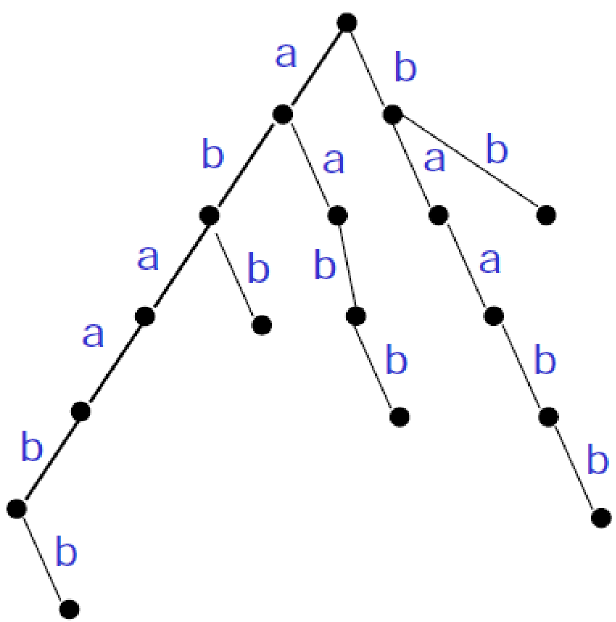
Online construction







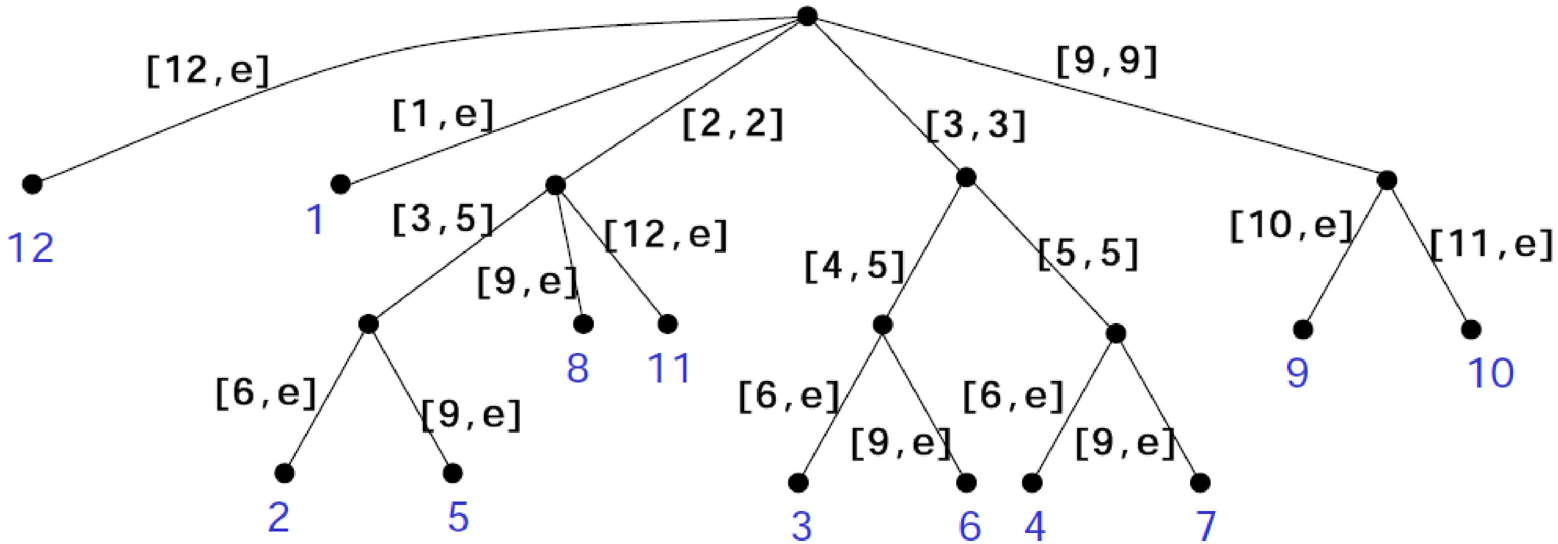




Ukkonen's on-line construction of suffix trees works in a similar way.

It maintains collapsed edges at all times.

T = mississippi$



3. Applications of Suffix Trees

Generalized Suffix tree for a SET S of strings:

$S = \{ S_1, S_2, S_3, \ldots, S_k \}$

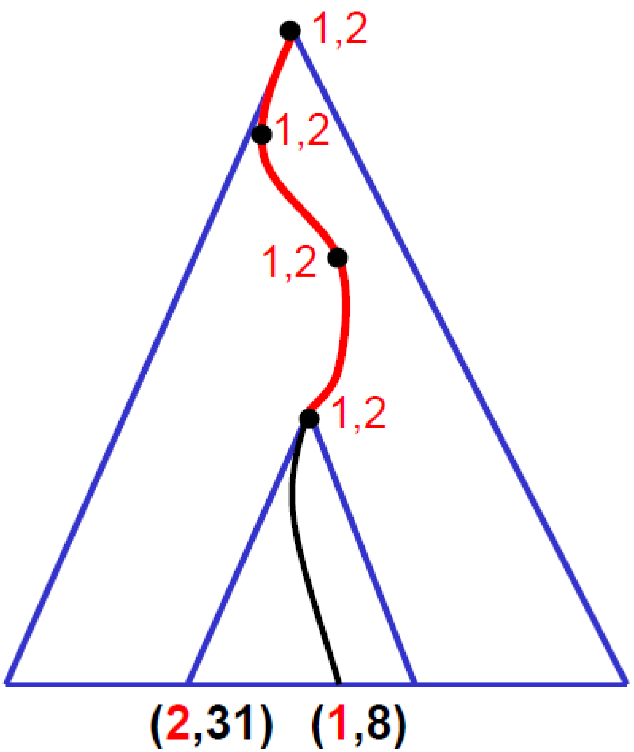
$T = S_1 \#_1 S_2 \#_2 S_3 \#_3 \ldots S_k \#_k$

Where $\#_1, \#_2, \ldots, \#_k$ are fresh new symbols.

(b) Longest Common Substring of two Strings

S_1 = superiorcalifornialives
S_2 = sealiver

LCS(S_1, S_2) = alive



→ Build generalized suffix tree of { S_1, S_2 }
→ Mark internal nodes with "1" or "2"
if subtree contains **(1,_)** pair or **(2, _)** pair.

LCS(S1, S2) =
maximal *string depth* of any
node marked "1,2"

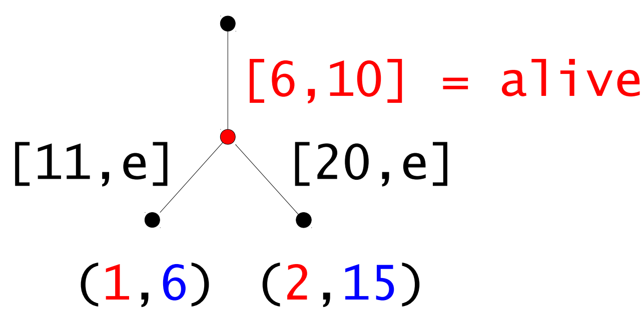
→ Can be determined by a simple
tree traversal

(b) Longest Common Substring of two Strings

S_1 = fornialives
S_2 = sealiver

$LCS(S_1, S_2)$ = alive

```
          11   1    2
12345678901    5    0
fornialives#sealiver
```



(b) Longest Common Substring of two Strings

Theorem
The *longest common substring* of two strings can be found in linear time, using a generalized suffix tree.

[Karp,Miller,Rosenberg1972] solved the problem in $O((m+n)\log(m+n))$ time where $m=|S_1|$ and $n=|S_2|$.

In 1970 Donald Knuth conjectured that it is *impossible* to solve the problem in linear time!

→ Linear time solution by [Weiner,1973]

First linear time suffix tree construction algorithm

(c) Matching Statistics

ms(k) = length L of longest substring T[k…k+L] that matches a substring in P.
p(k) = start position in P of a substring of length ms(k) matching T[k…k+ms(k)]

T = abcxabcdex . . . .
P = yabcwzqabcdw

ms(1) = 3
p(1) = 2

ms(5) = 4
p(4) = 8

Computation of ms and p

Build suffix tree of P (including suffix links).
At node v corresponding to ms(i),
compute ms(i+1) as follows:
(1) If v is internal, follow its suffix link.
(2) If v is leaf, walk to parent (label γ)

Current node is prefix of T[i+1…n].
Proceed downwards to longest match
(as in ordinary search)

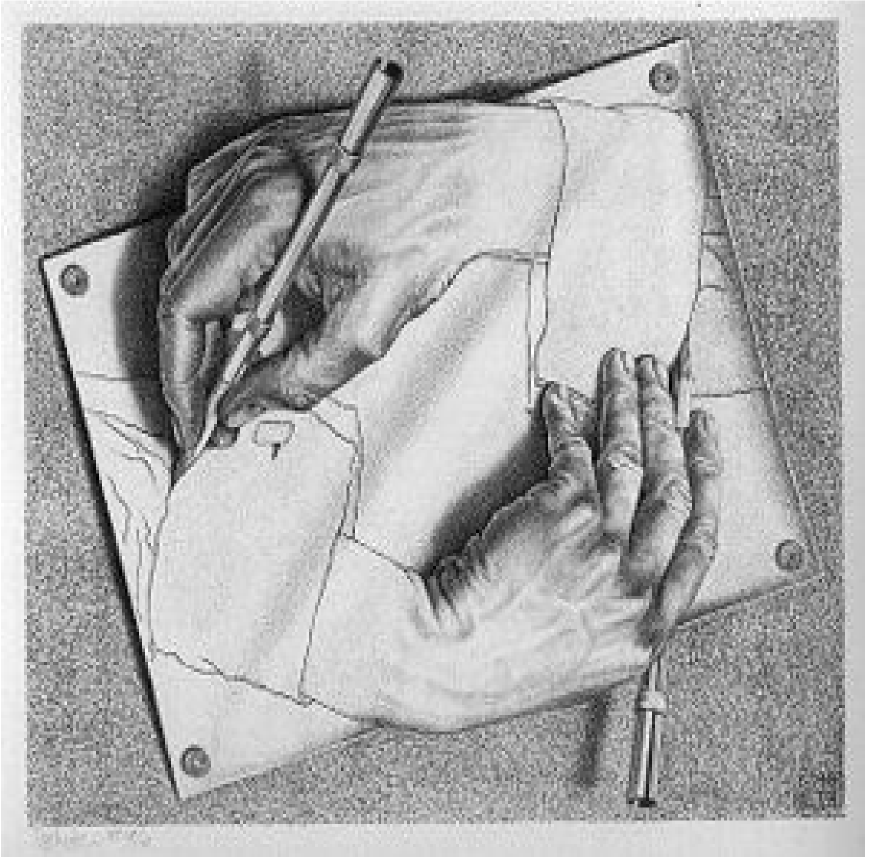
→Allows to find LCS(S_1,S_2) using only
one suffix tree (of the shorter string).

(d) Compression

→ E.g., infinite-window Lempel-Ziv like compression

a b a abaa aba baba ab b → a b a (1,4) (1,3) (9,4) (1,2) b

(position, length)



M. C. Escher (1948)

(d) Compression

Implemented in an open-source compression tool.
→ Very high compression ratios!

LZ-variant with infinite window

abaabaaabababaabb

a b a abaa aba baba ab b

longest string that has appeared before
coded as: (position, length)

a b a (1,4) (1,3) (9,4) (1,2) b

→ Build suffix tree of text T
→ Annotate internal nodes by smallest position number in their subtree

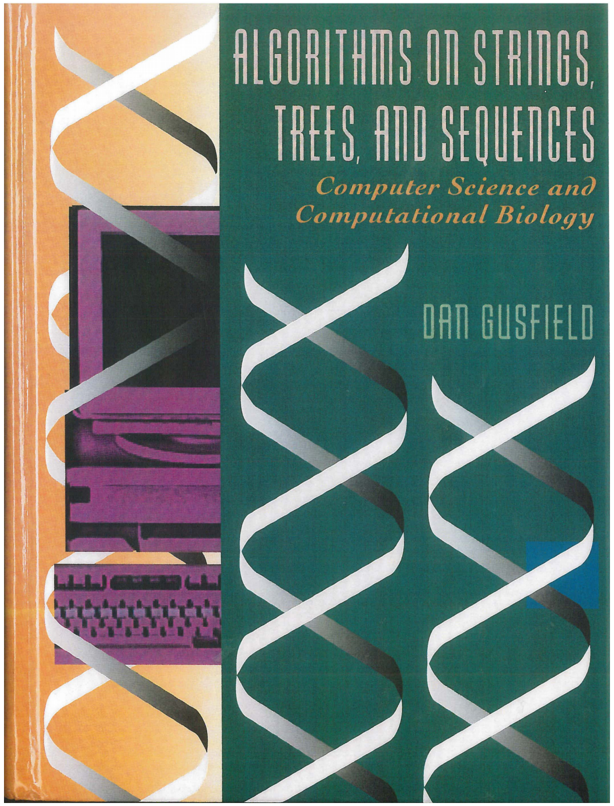
→ To find pair (x,y) at a position p in T, match T[x…] against suffix tree as long as minimal pos number is smaller than x.

 42

3. Applications of Suffix Trees

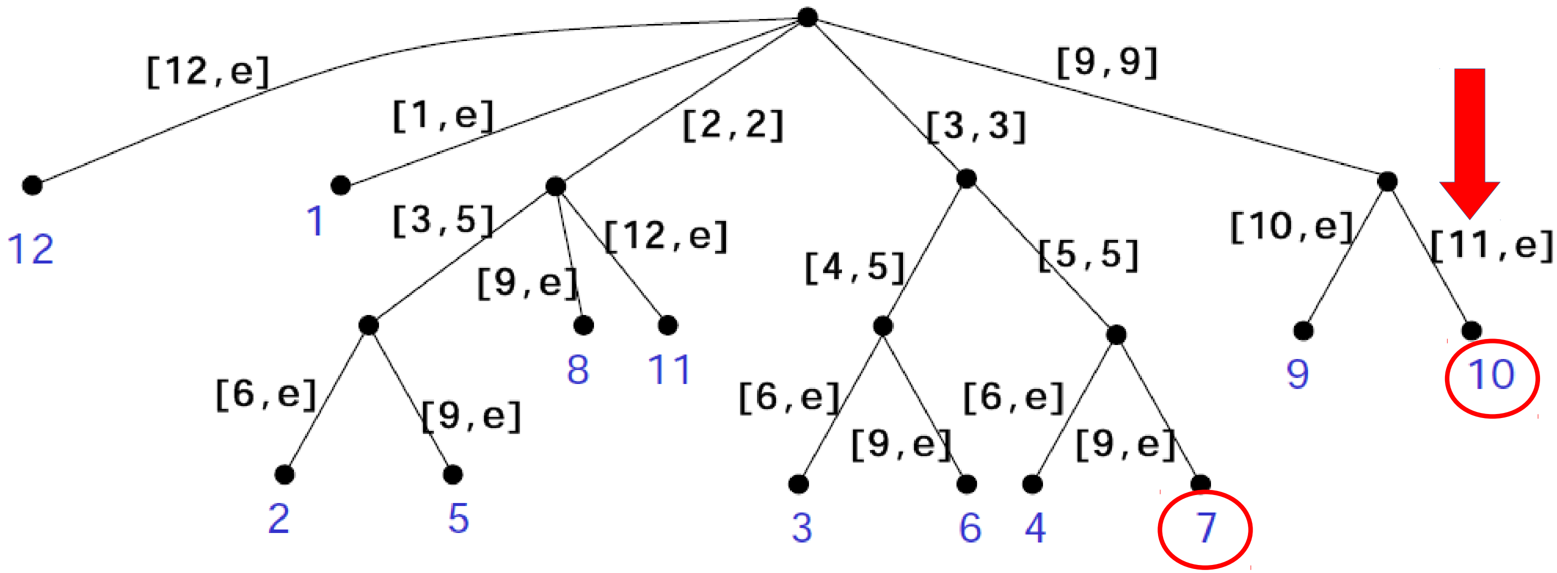
Suffix trees have *many* more applications
e.g. in computational biology see [Gusfield book].

→ Substring problem for a database of patterns
→ DNA contamination problem
→ Find complemented palindroms in DNA (e.g. AGCTCGCGAGCT)
→ Find all maximal repeats / maximal pairs
→ …



Space Consumption of Suffix Trees

T = mississippi$

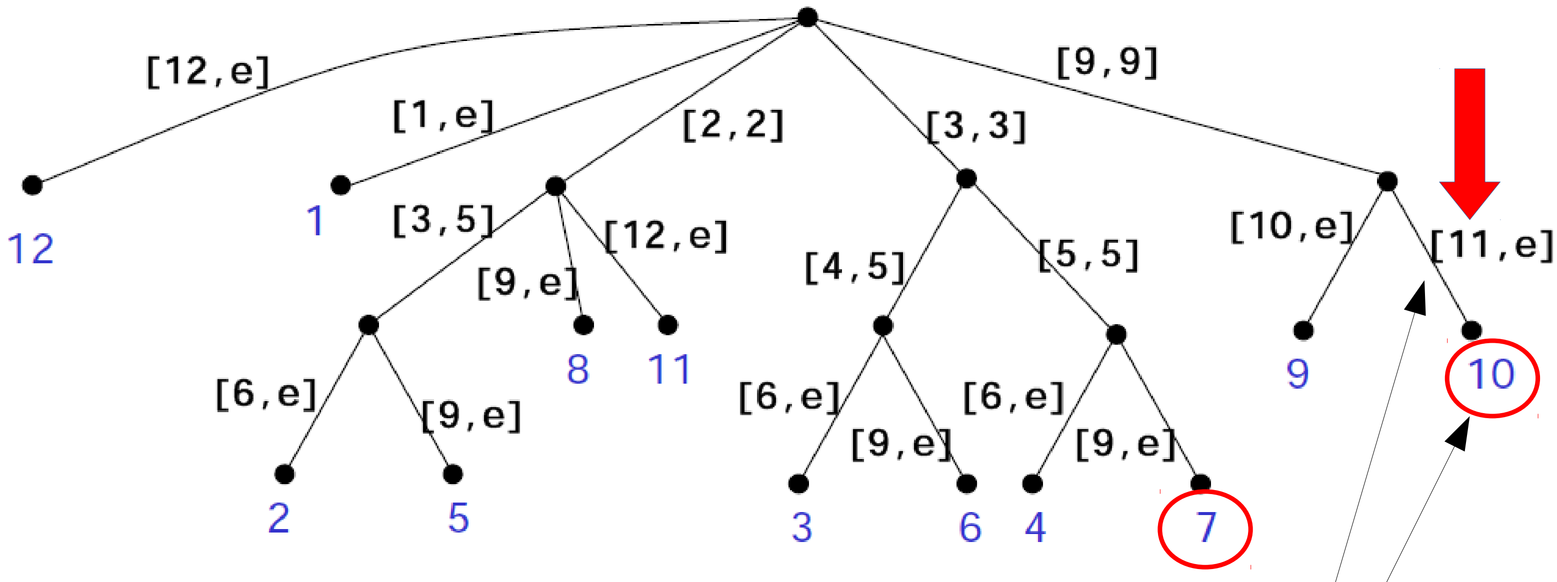


Questions

→ is the size of this tree really in O(**n**)?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??

T = mississippi$



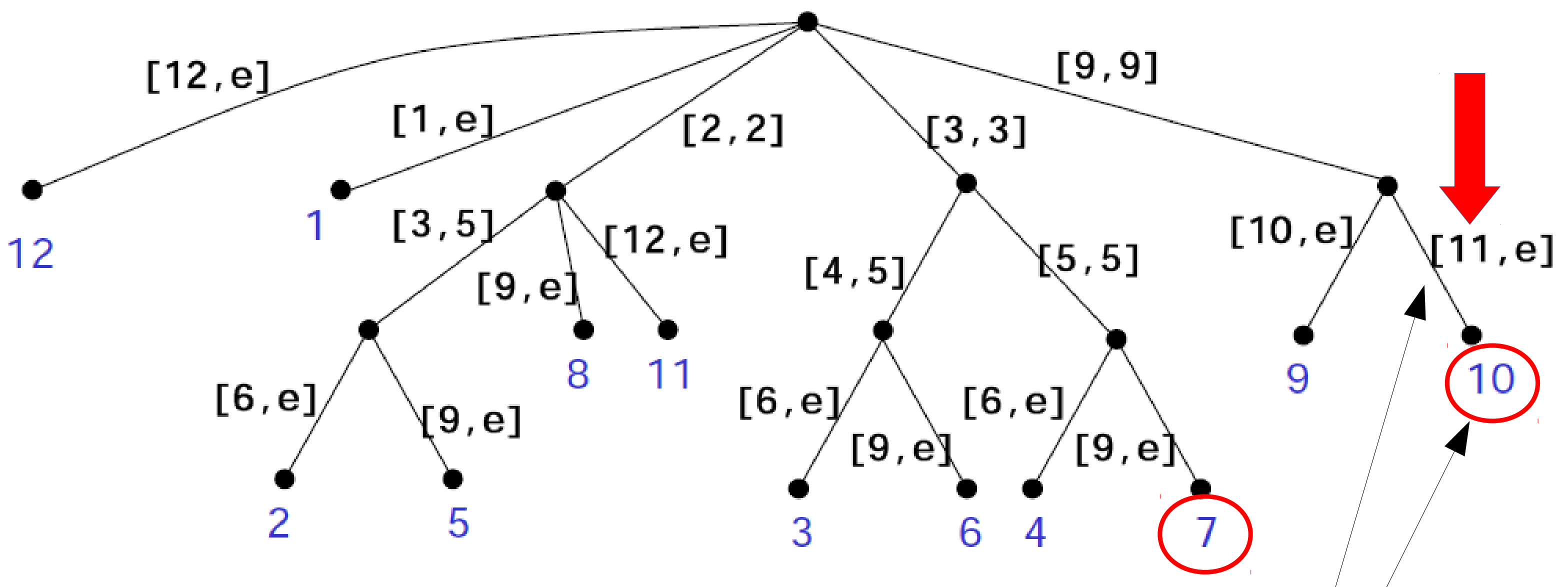
Questions

→ is the size of this tree really in O(**n**)?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??

each requires **log(n)** bits!?

T = mississippi$



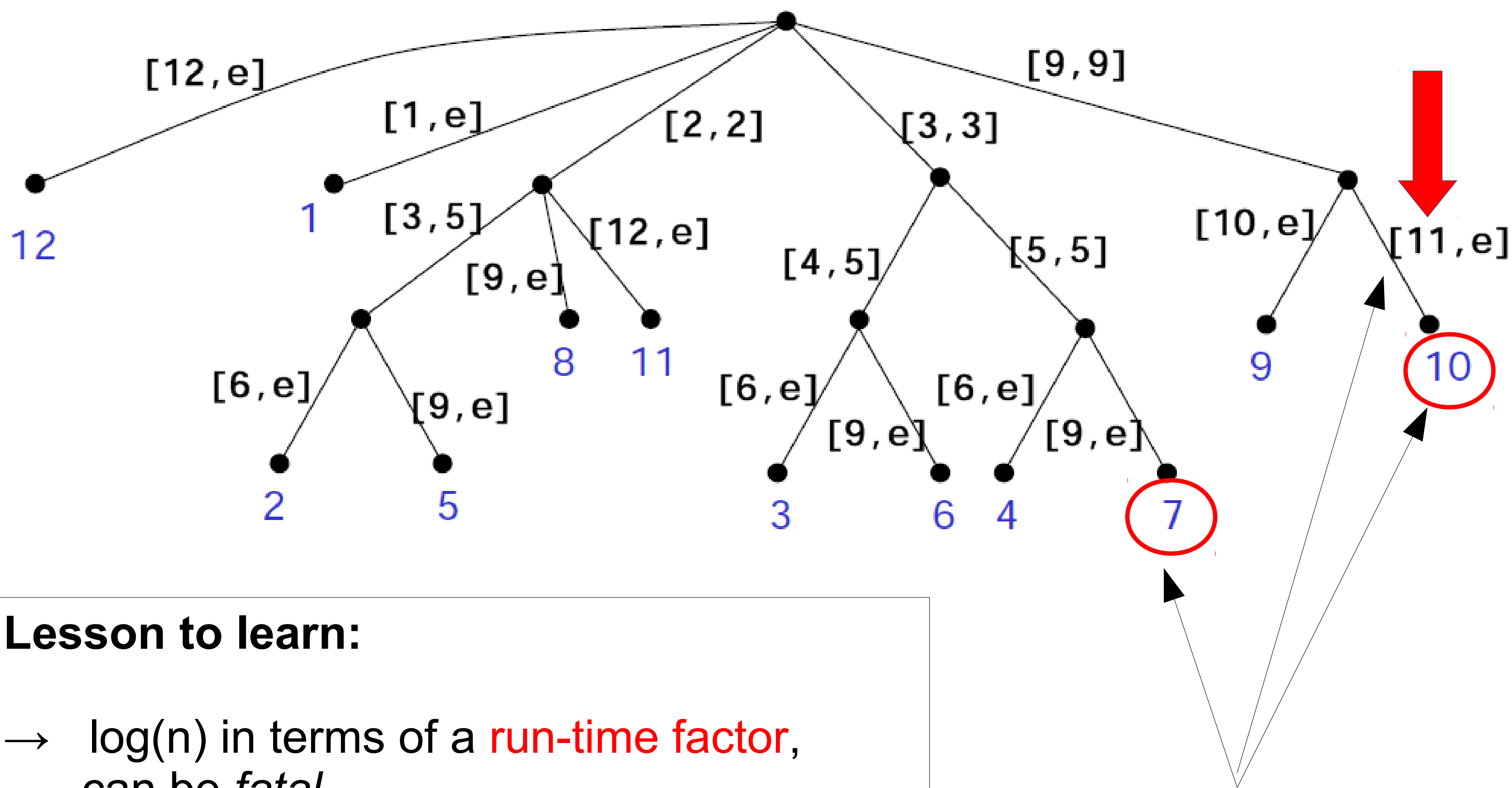
Yes, but:

→ log(**n**) is small, e.g., 64 bits

→ can be considered constant!

each requires **log(n)** bits!?

T = mississippi$



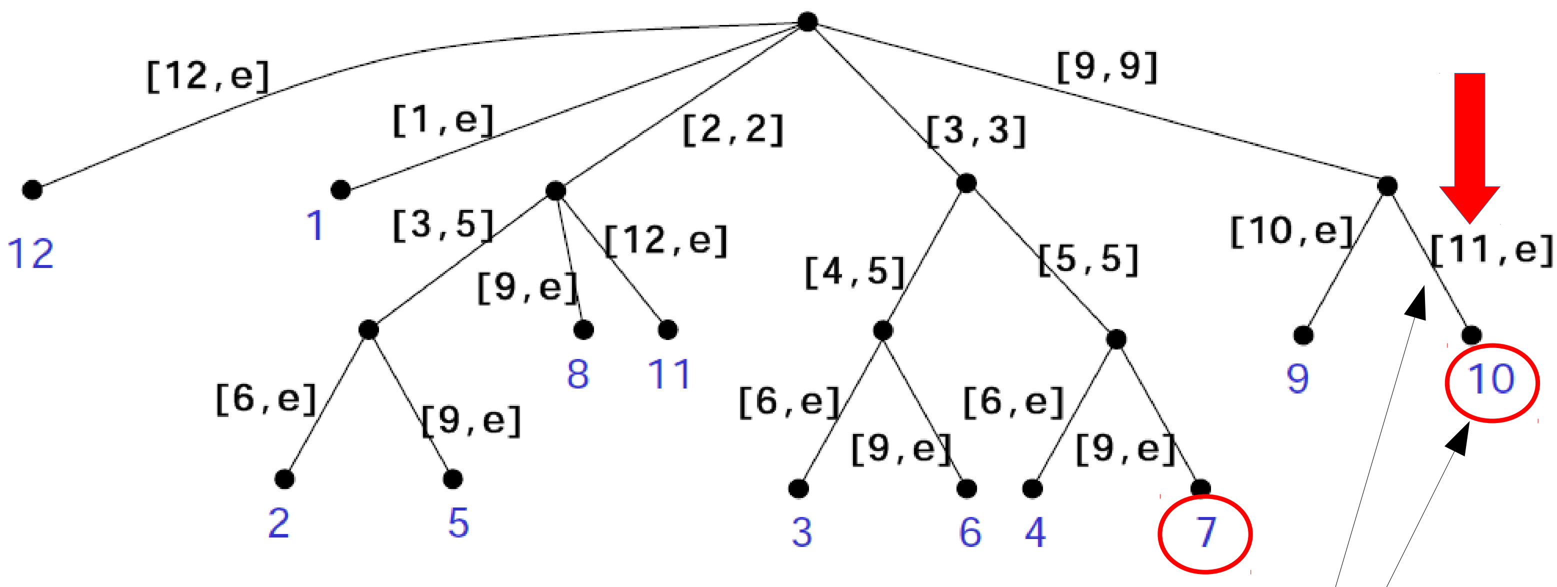
Lesson to learn:

→ log(n) in terms of a run-time factor, can be *fatal*

→ in terms of a space-factor, it is *fine*! how long will a text be? 2^64???

4 / 8 Bytes each is enough!

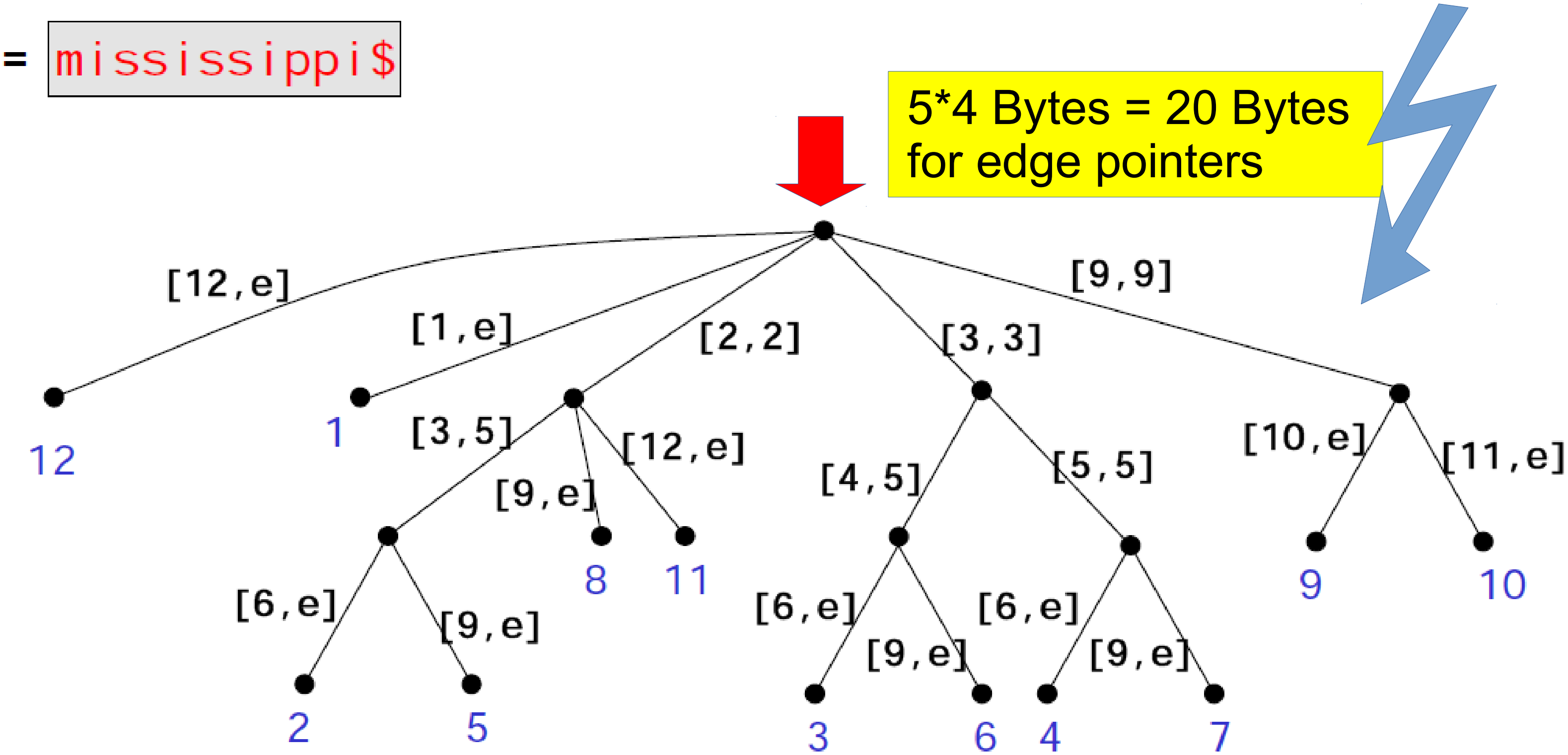
T = mississippi$



"For any text with fewer characters than #atoms in the universe, the label size for the suffix tree is a constant of **x bits**.. "

x Bits each is enough!

T = mississippi$



→ label size is not an issue

→ but, size of edge-pointers?

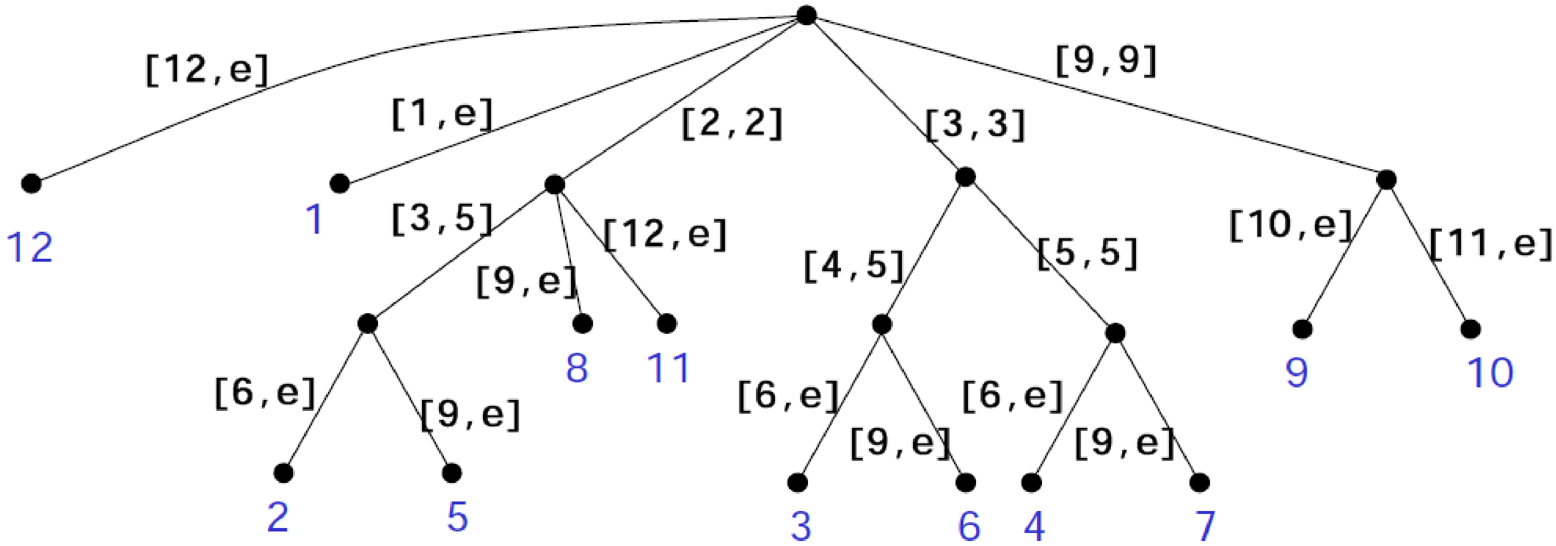
→ imagine each edge requires a 32-bit pointer!!

Actual Space of Suffix Trees

Space for edge-pointers is problematic:

→ actual space of suffix tree, ca. **20|T|**

→ on commodity hardware, texts of more than 1GB are not doable



→ how to avoid the huge space needed for edges?

4. Suffix Array

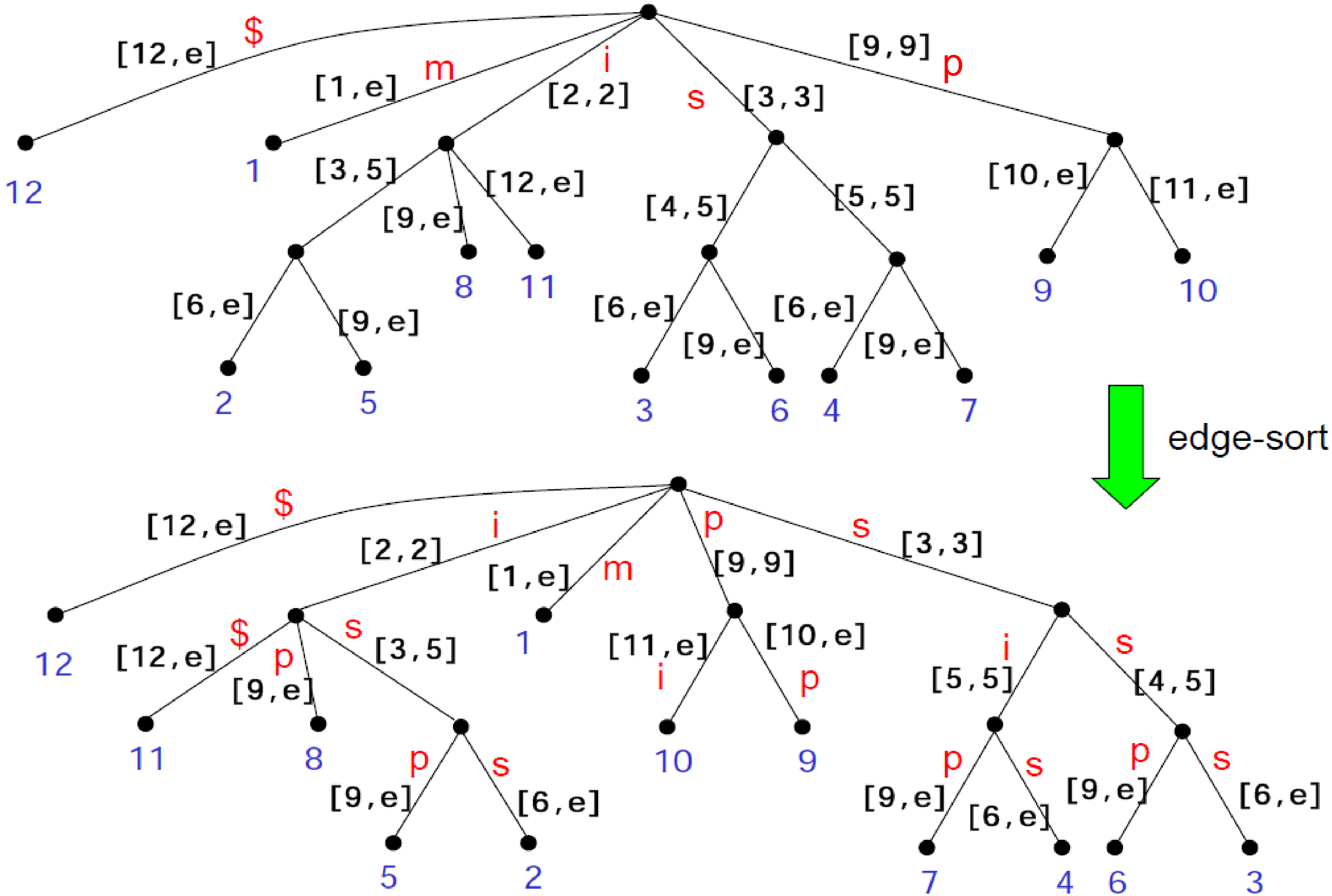
Definition

Given text T of length n. For i=1…n, SA[k]=i if suffix T[i…n] is at position k in the lexicographic order T's suffixes.

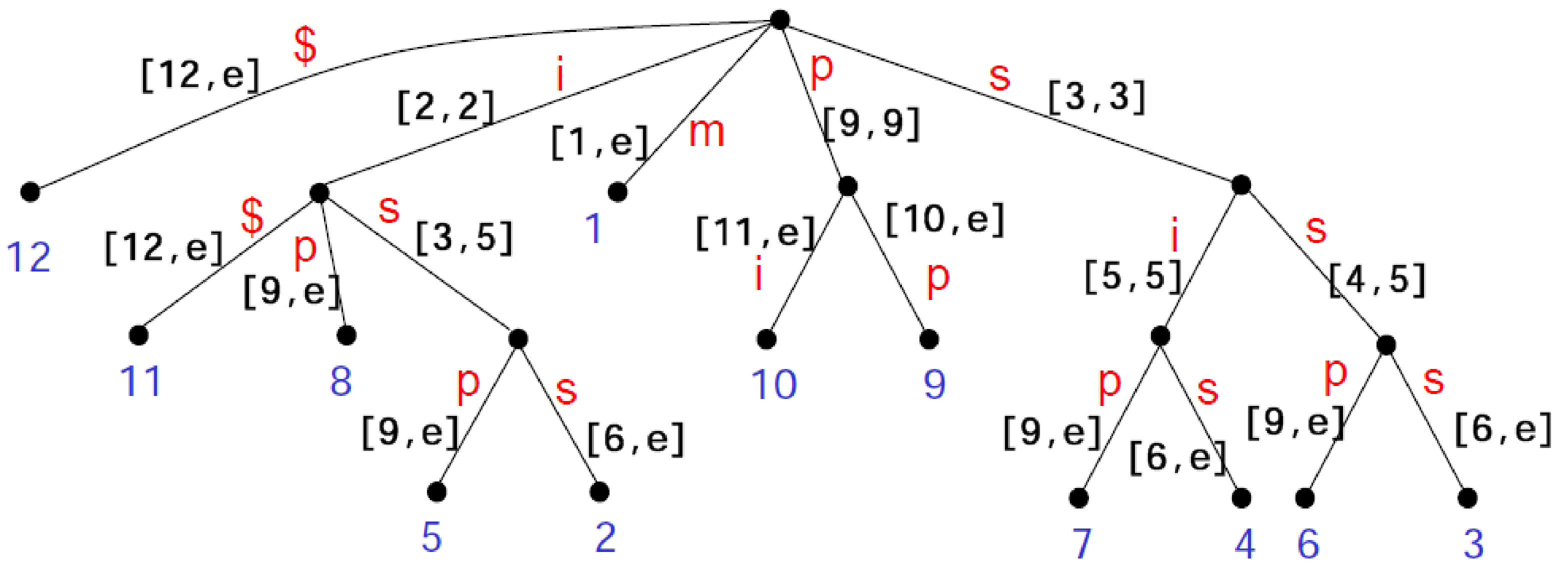
```
     1234567890
T  = mississippi$        Order  $ < i < m < p < s

    12  $
    11  i$
     8  ippi$            SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
     5  issippi$
     2  ississippi$
     1  mississippi$
    10  pi$
     9  ppi$
     7  sippi$
     4  sissippi$
     6  ssippi$
     3  ssissippi$
```

Suffix Array Construction



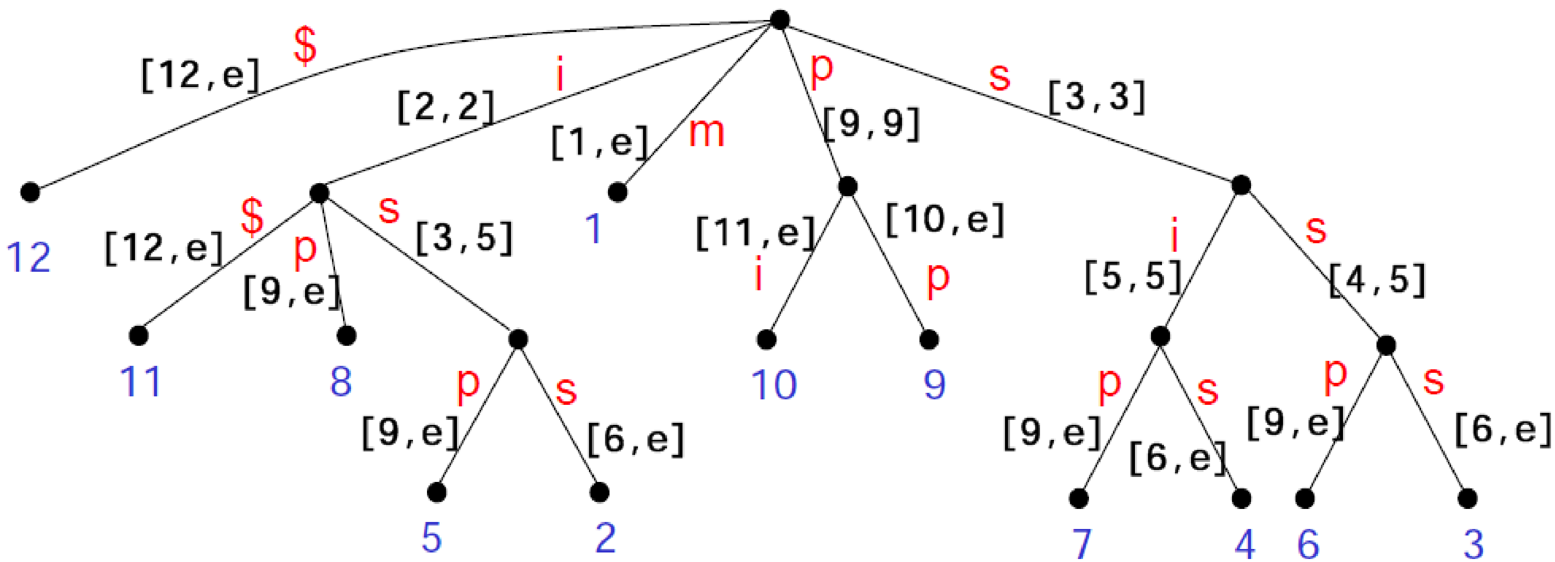
4. Suffix Array



→ read leaves from left-to-right!

SA(T) = [12,11,8,5,2,10,9,7,4,6,3]

4. Suffix Array



→ read leaves from left-to-right!

SA(T) = [12,11,8,5,2,10,9,7,4,6,3]

Theorem

The suffix array of T can be constructed in time O(|T|).

Search

Theorem
Using binary search on SA(T), all occurrences of P in T can be located in O(|P| * log|T|) time.

```
        1234567890
 T  = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

Search for P= issi

all occurren's consecutive in SA!

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

Search

```
     1234567890
T  = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
        ↑          ↑           ↑
        L          M           R
                issi ≤ miss?
```

12	\$
11	i\$
8	ippi\$
5	issippi\$
2	ississippi\$
1	mississippi\$
10	pi\$
9	ppi\$
7	sippi\$
4	sissippi\$
6	ssippi\$
3	ssissippi\$

Binary search for start-index:
L=1, R=|T|=n
Repeat
  $M = \lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

Search

```
     1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L

R

issi $\leq$ miss?

yes

issi $\leq$ ippi?

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

Search

1234567890
T = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L M R

issi $\leq$ issi

12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

Binary search for start-index:
L=1, R=|T|=n
Repeat
M = $\lceil (L+R-1)/2 \rceil$
If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

Search

```
   1234567890
T = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
```

L R

issi ≤ issi

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for start-index:
L=1, R=|T|=n
Repeat
  M = $\lceil (L+R-1)/2 \rceil$
  If P $\leq_{lex}$ T[M…M+|P|] then R:=M else L:=M
Until M does not change.

Search

```
      1234567890
 T = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
              ↑
              R
```

issi ≤ issi

M does not change.
➔ Return R

Binary search for start-index:
L=1, R=|T|=n
Repeat
  $M = \lceil (L+R-1)/2 \rceil$
  If $P \leq_{lex} T[M \ldots M+|P|]$ then R:=M else L:=M
Until M does not change.

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 missisippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Search

Start-index

```
     1234567890
T  = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]
```

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for end-index:

L=1, R=|T|=n

Repeat

  M = $\lceil (L+R-1)/2 \rceil$

  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

Search

Start-index

1234567890
T = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

L M R

not(issi < issi)
➔

12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

Binary search for end-index:
L=1, R=|T|=n
Repeat
  $M = \lceil (L+R-1)/2 \rceil$
  If $P <_{lex} T[M \ldots M+|P|]$ then R:=M else L:=M
Until M does not change.

Search

Start-index

```
     1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

M R

L

not(issi < issi)

➔ L:=M

Binary search for end-index:

L=1, R=|T|=n

Repeat

  M = $\lceil (L+R-1)/2 \rceil$

  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

12 $

11 i$

8 ippi$

5 issippi$

2 ississippi$

1 mississippi$

10 pi$

9 ppi$

7 sippi$

4 sissippi$

6 ssippi$

3 ssissippi$

Search

Start-index

1234567890
T = mississippi$

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

R

L

M does not change.

➔ Return L

12 $
11 i$
8 ippi$
5 issippi$
2 ississippi$
1 mississippi$
10 pi$
9 ppi$
7 sippi$
4 sissippi$
6 ssippi$
3 ssissippi$

Binary search for end-index:
L=1, R=|T|=n
Repeat
    $M = \lceil (L+R-1)/2 \rceil$
    If $P <_{lex} T[M \ldots M+|P|]$ then R:=M else L:=M
Until M does not change.

Search

Start-index

```
   1234567890
T = mississippi$
```

SA(T)=[12,11,8,5,2,1,10,9,7,4,6,3]

End-index

```
12 $
11 i$
 8 ippi$
 5 issippi$
 2 ississippi$
 1 mississippi$
10 pi$
 9 ppi$
 7 sippi$
 4 sissippi$
 6 ssippi$
 3 ssissippi$
```

Binary search for end-index:

L=1, R=|T|=n

Repeat

  $M = \lceil (L+R-1)/2 \rceil$

  If P $<_{lex}$ T[M…M+|P|] then R:=M else L:=M

Until M does not change.

Search

Theorem
Using binary search on SA(T), all occurrences of P in T can be located in O(|P| * log|T|) time.

Note
This is a pessimistic bound!
We *almost never* need O(|P|) time for one lexicographic comparison!

On random strings, this should run in O(|P| + log|T|) time.

Search

Theorem
Using binary search on SA(T), all occurrences of P in T can be located in O(|P| * log|T|) time.

Note
This is a pessimistic bound!
We *almost never* need O(|P|) time for one lexicographic comparison!

On random strings, this should run in O(|P| + log|T|) time.

→ O(|P| + log|T|) in practise, using a simple trick

→ O(|P| + log|T|) guaranteed, using LCP-array

LCP(k,j) = longest common prefix of T[SA[k]...]
and T[SA[j]...]

Suffix Arrays

→ much more space efficient than Suffix Tree
→ used in pratise (suffix tree more used in theory)

→ Suffix Array Construction, without Suffix Trees?

[Linear Work Suffix Array Construction,
J. Kärkkäinen, Sanders, Burkhardt,
Journal of the ACM, 2006]

→ See also (linked from course web page)

[A taxonomy of suffix array construction algorithms,
S. J. Puglisi, W. F. Smyth, A. Turpin,
ACM Computing Surveys 39, 2007]

PST

From p.7 of

PST

Table I. Performance Summary of the Construction Algorithms

Algorithm	Worst Case	Time	Memory
Prefix-Doubling			
MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive			
KA [Ko and Aluru 2003]	$O(n)$	2.5	7–$10n$
KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	10–$13n$
KSPP [Kim et al. 2003]	$O(n)$	—	—
HSS [Hon et al. 2003]	$O(n)$	—	—
KJP [Kim et al. 2004]	$O(n \log \log n)$	3.5	13–$16n$
N [Na 2005]	$O(n)$	—	—
Induced Copying			
IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	5–$6n$
MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	9–$10n$
BB [Baron and Bresler 2005]	$O(n\sqrt{\log n})$	2.1	$18n$
M [Maniscalco and Puglisi 2007]	$O(n^2 \log n)$	1.3	5–$6n$
MP [Maniscalco and Puglisi 2006]	$O(n^2 \log n)$	1	5–$6n$
Hybrid			
IT+KA	$O(n^2 \log n)$	4.8	$5n$
BK+IT+KA	$O(n \log n)$	2.3	5–$6n$
BK+S	$O(n \log n)$	2.8	5–$6n$
Suffix Tree			
K [Kurtz 1999]	$O(n \log \sigma)$	6.3	13–$15n$

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].

END
Lecture 15