Applied Databases

Lecture 15
Indexed String Search, Suffix Trees

Sebastian Maneth

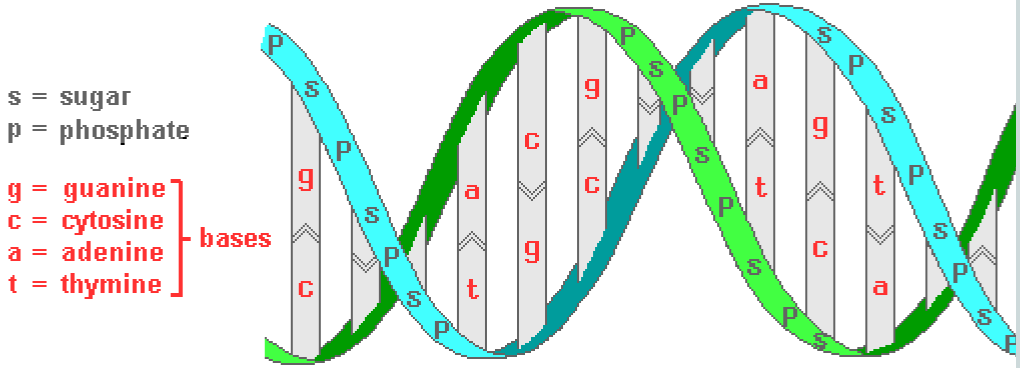
University of Edinburgh - March 7th, 2016

Outline

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

String Search

→ search over DNA sequences
→ huge sequence over C, T, G A (ca. 3.2 billion)
→ no spaces, no tokens....



String Search

→ search over DNA sequences
→ huge sequence over C, T, G A (ca. 3.2 billion)
→ no spaces, no tokens....

Given
– a long string T (text)
– a short string P (pattern)

Problem 1: find all occurrences of P in T
Problem 2: count #occurrence of P in T

String Search

→ search over DNA sequences
→ huge sequence over C, T, G A (ca. 3.2 billion)
→ no spaces, no tokens....

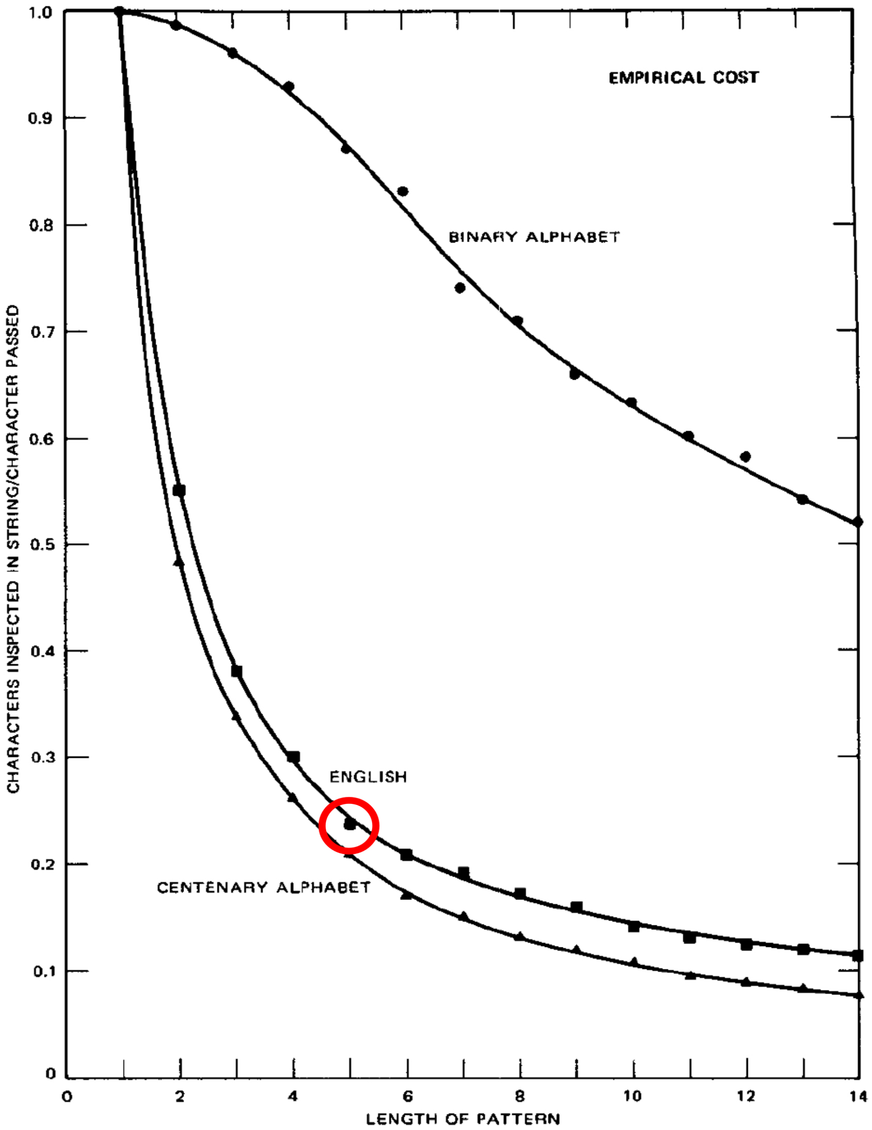
Given
– a long string T (text) of length n
– a short string P (pattern) of length m

Problem 1: find all occurrences of P in T
Problem 2: count #occurrence of P in T

Online Search O(|T|) time with O(|P|) preprocessing
E.g., using *automaton* or *KMP*

→ **sublinear time** using *Horspool* / *Boyer-Moore*
→ average time limit: $O(n (\log m)/m)$

BM – Average Case



To find first occurrence i of an arbitrary 5-letter word in an English text Inspects on average

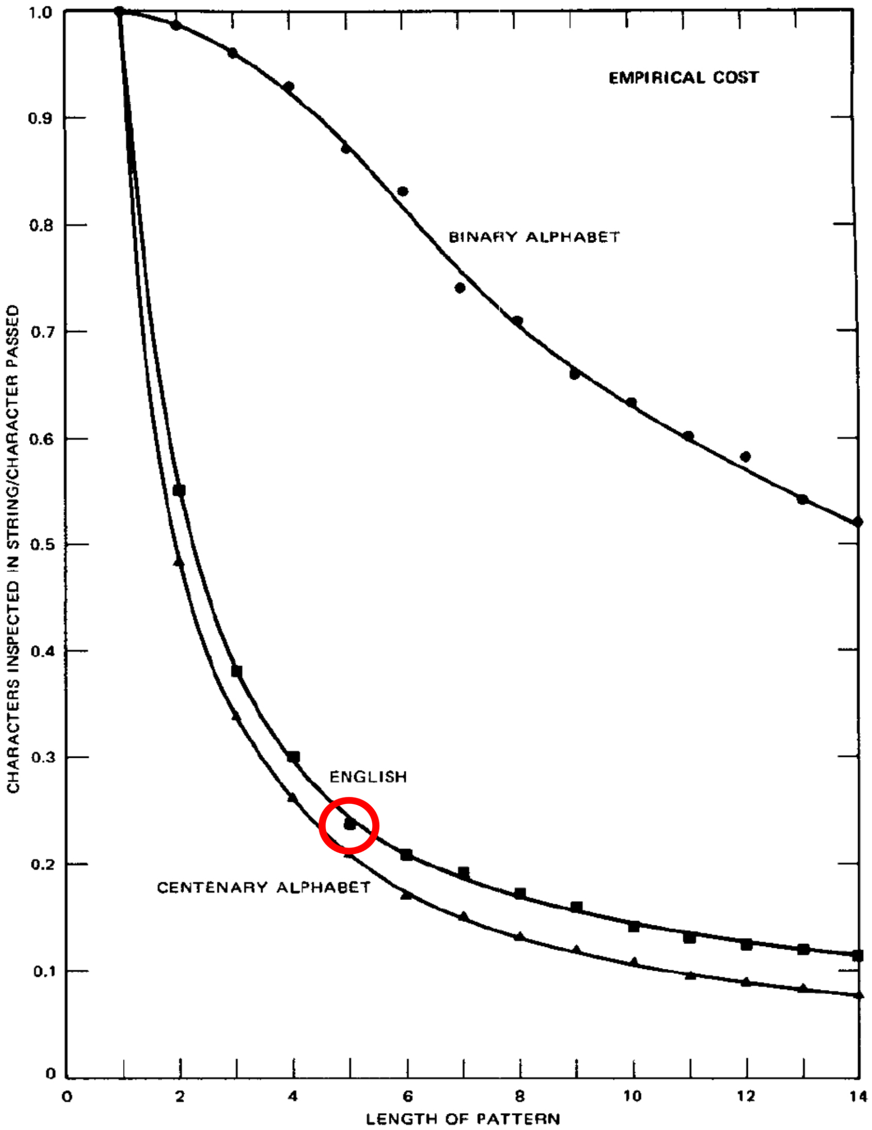
(0.25 * i)

text symbols.

→ **sublinear time** using *Horspool* / *Boyer-Moore*

→ average time limit: O(T (log m)/m)

BM – Average Case



To find first occurrence i of an arbitrary 5-letter word in an English text Inspects on average

(0.25 * i)

text symbols.

→ for DNA, 40% of 3.2 billion is still huge (linear scan of >1TB)

Indexed String Search

Given
– a long string T (text)
– a short string P (pattern) $m = |P|$

Problem 1 find all occurrences of P in T
Problem 2 count #occurrence of P in T

Offline Search = Indexed Search
= (linear time) preprocessing of T

Highlights → O(m) time for Problem 1
→ O(m + #occ) time for Problem 2

Indexed String Search

Given
– a long string T (text)
– a short string P (pattern) $m = |P|$

Problem 1 find all occurrences of P in T
Problem 2 count #occurrence of P in T

Offline Search = Indexed Search
= (linear time) preprocessing of T

Highlights → O(m) time for Problem 1
→ O(m + #occ) time for Problem 2

Independent of size of text T!!!

Indexed String Search

Count / Find all occurrences of P in T

Preprocessing ("indexing") of T is permitted

<u>Naive Solution</u>

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct "next letter" (tree structure)

a

(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
(c

c

Indexed String Search

Count / Find all occurrences of P in T

Preprocessing ("indexing") of T is permitted

<u>Naive Solution</u>

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct "next letter" (tree structure)

a
(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
c
.
.
(c

Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with
next letter P[2]
Etc.
after m jumps, reach (or not) matching substring
with its occurrence list

Indexed String Search

Count / Find all occurrences of P in T

Preprocessing ("indexing") of T is permitted

Search Time
→ O(m) [good!]

Indexing Time
→ ????

Naive Solution

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct "next letter" (tree structure)

a
c

(a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
(c

Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with next letter P[2]
Etc.
after m jumps, reach (or not) matching substring with its occurrence list

Indexed String Search

Count / Find all occurrences of P in T

Preprocessing ("indexing") of T is permitted

Search Time
→ O(m) [good!]

Indexing Time
→ exceeds O(n^2)
(sort n^2 substrings)

Naive Solution

1. List all substrings of T, together with their occurrence lists
   (string1, [3,7,21]), (string2, [3,21]), …

2. Lexicographically sort the substrings

3. Record the beginnings of each distinct "next letter" (tree structure)

a → (a, [3,4,6,7,10,...])
(ab, [7,10, ..])
(ad, ..
(b, …
.
.
c → (c

Search occurrences of P:

→ jump to substrings starting with letter P[1]
→ from there, jump to substrings with
next letter P[2]
Etc.
after m jumps, reach (or not) matching substring
with its occurrence list

1. Suffix Trie

→ Idea: consider all suffixes of text T
i.e., suffix starting at position 1 (= T)
suffix starting at position 2
suffix starting at position 3
Etc.

→ arrange suffixes in a "prefix tree" (trie),
with longest common prefixes shared

1. Suffix Trie

→ Idea: consider all suffixes of text T
i.e., suffix starting at position 1 (= T)
suffix starting at position 2
suffix starting at position 3
Etc.

→ arrange suffixes in a “prefix tree” (trie),
with longest common prefixes shared

→ trie datastructure: 1959 by de la Briandais

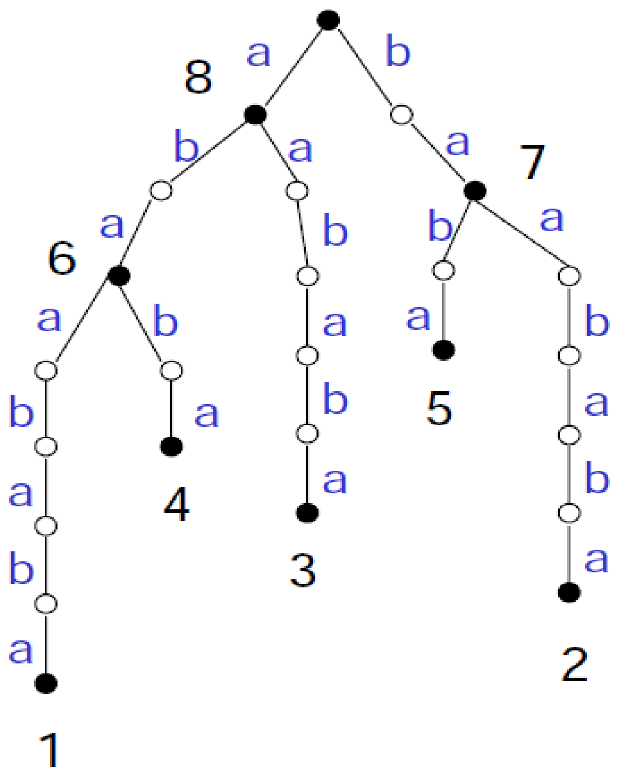
→ “trie” (Fredkin, 1961), pronounced /ˈtriː/ (as "tree")

RETR**IE**VAL

→ to distinguish from “tree” many authors
say /ˈtraɪ/ (as "try")

1. Suffix Trie

```
     12345678
T =  abaababa
```



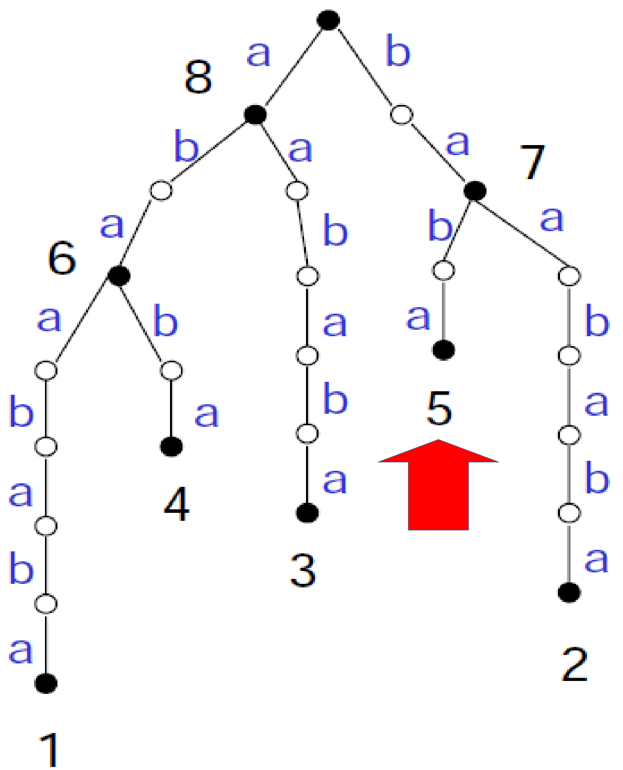
Suffixes

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

Trie of all suffixes of T=abaababa.

1. Suffix Trie

```
    12345678
T = abaababa
```



Trie of all suffixes of T=abaababa.

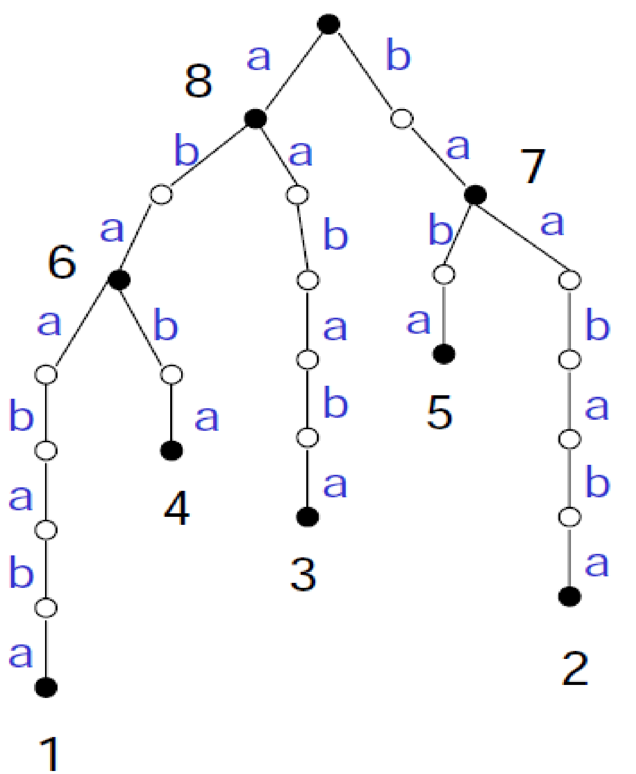
```
Suffixes
1  abaababa
2  baababa
3  aababa
4  ababa
5  baba
6  aba
7  ba
8  a
```

→ black nodes represent suffixes

→ are labeled by the corresponding number of the suffix

1. Suffix Trie

```
   12345678
T = abaababa
```



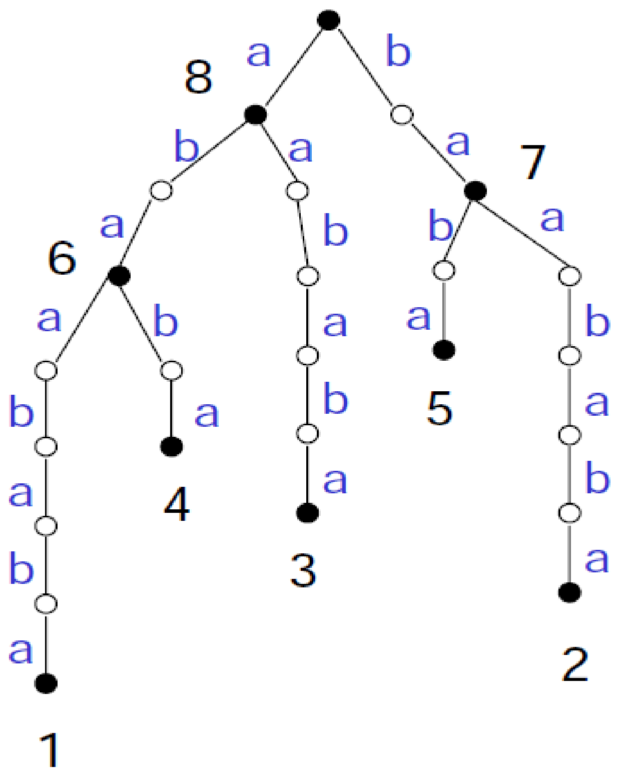
Trie of all suffixes of T=abaababa.

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

→ how to search for all occurrences of a pattern P?

1. Suffix Trie

```
   12345678
T = abaababa
```



Trie of all suffixes of T=abaababa.

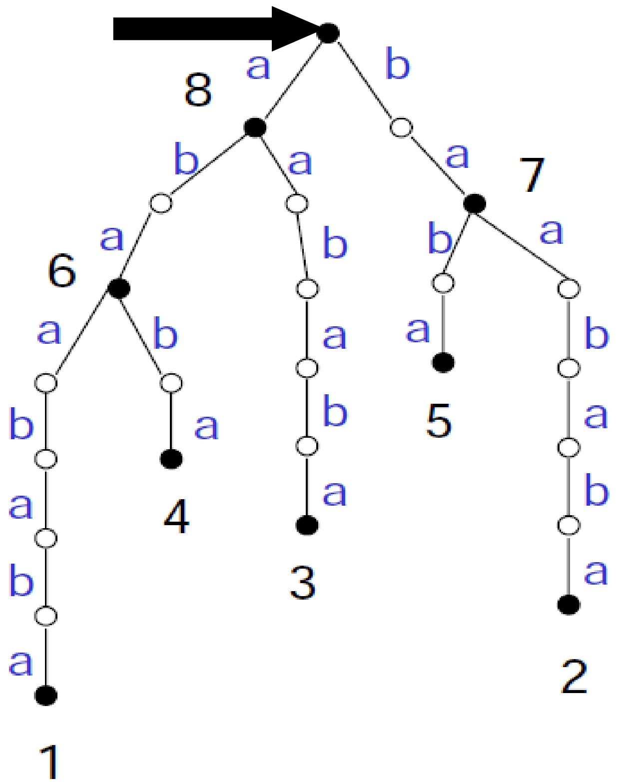
```
Suffixes
1  abaababa
2  baababa
3  aababa
4  ababa
5  baba
6  aba
7  ba
8  a
```

→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

1. Suffix Trie

```
      12345678
T  =  abaababa
```



Trie of all suffixes of T=abaababa.

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

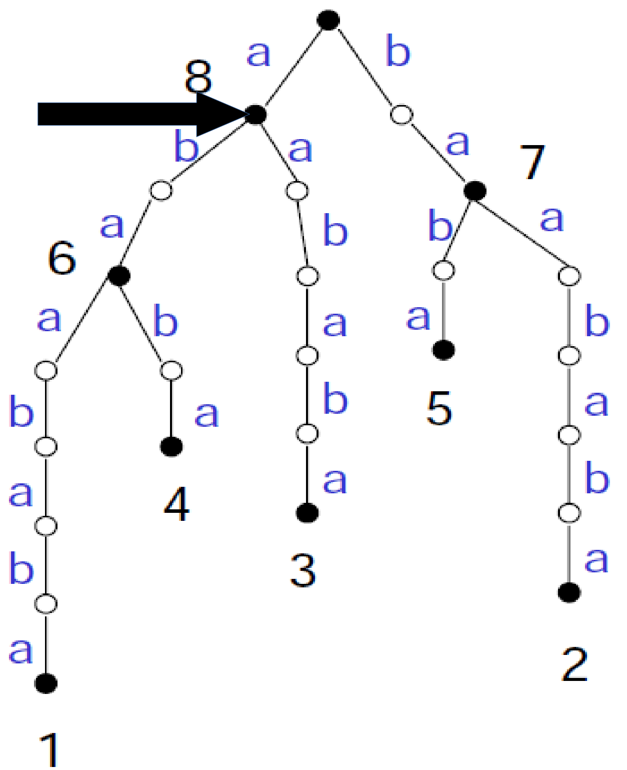
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

1. Suffix Trie

```
     12345678
T =  abaababa
```



Trie of all suffixes of T=abaababa.

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

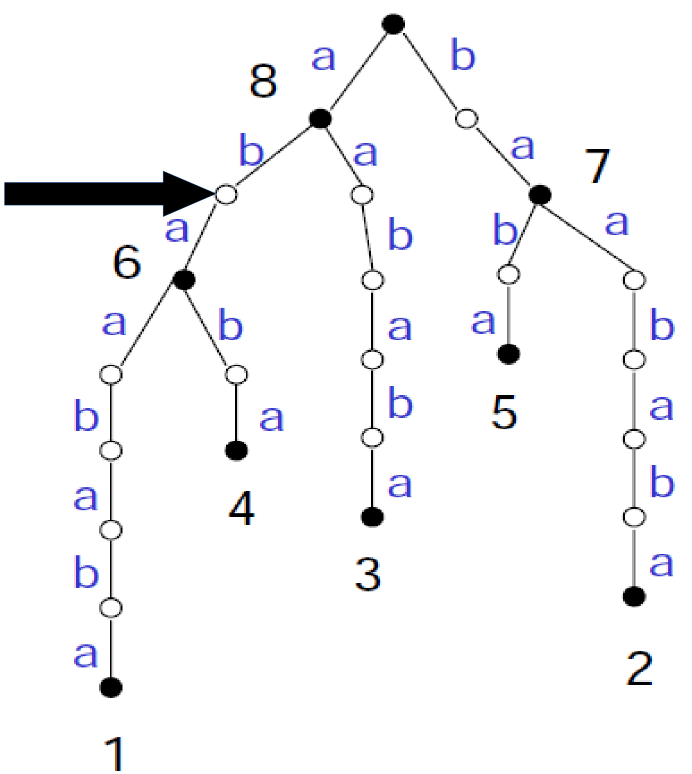
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = a**b**a

1. Suffix Trie

12345678
T = abaababa



Trie of all suffixes of T=abaababa.

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

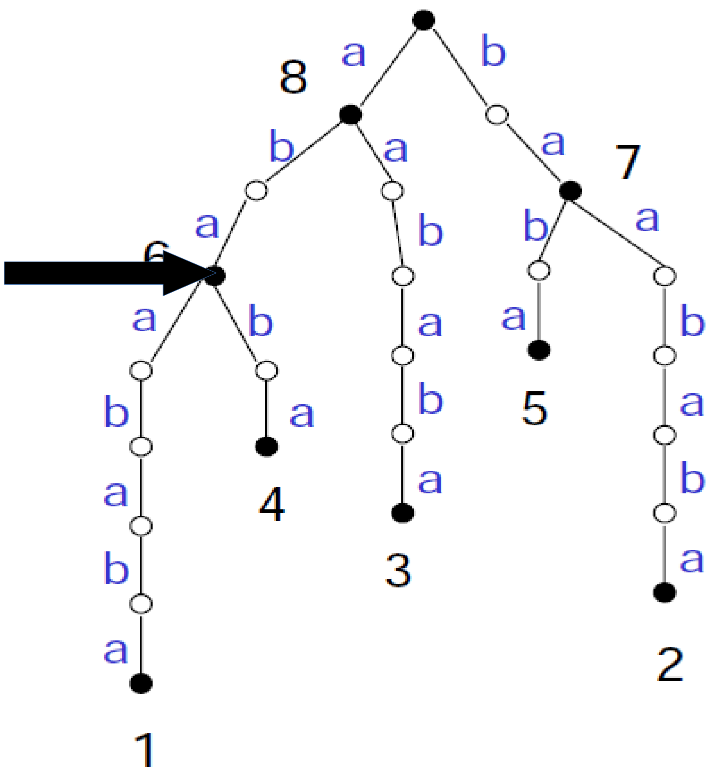
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = ab**a**

1. Suffix Trie

12345678
T = abaababa



Trie of all suffixes of T=abaababa.

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

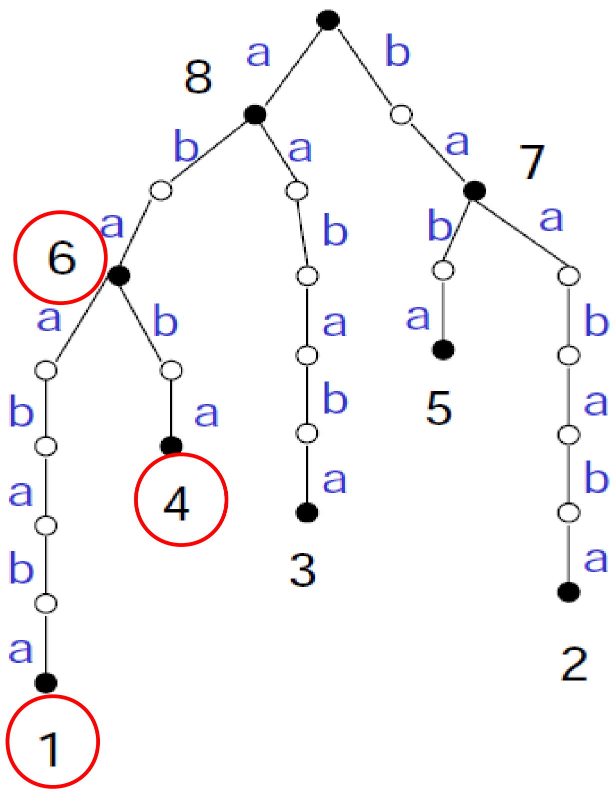
→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

1. Suffix Trie

```
   12345678
T = abaababa
```



3 matches of P = “aba”

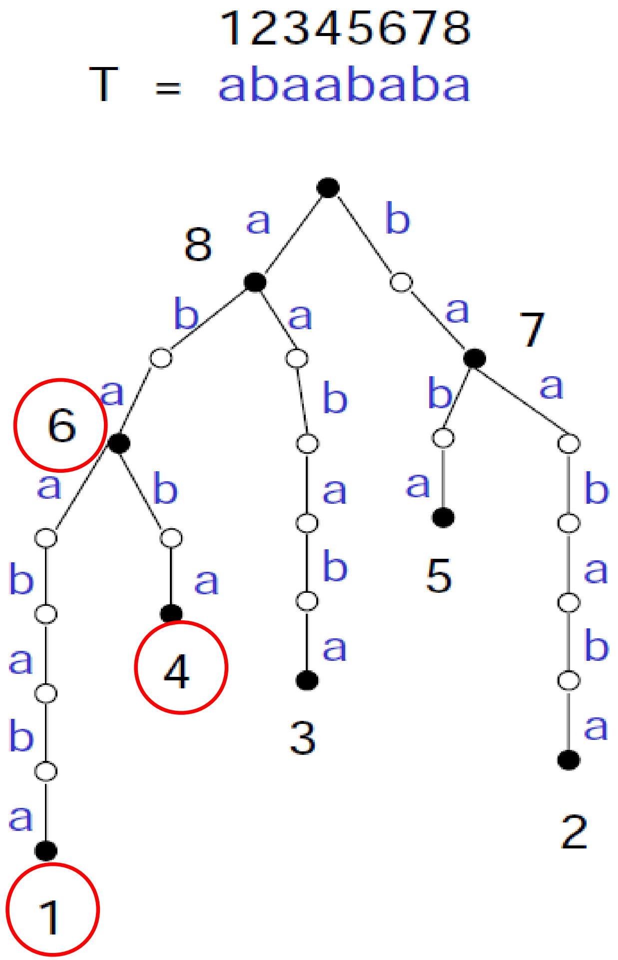
Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

→ how to search for all occurrences of a pattern P?

→ starting at the root node follow letter-by-letter wrt P the unique edges in the trie!

P = aba

1. Suffix Trie



3 matches of P = "aba"

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

→ O(m) **count time**

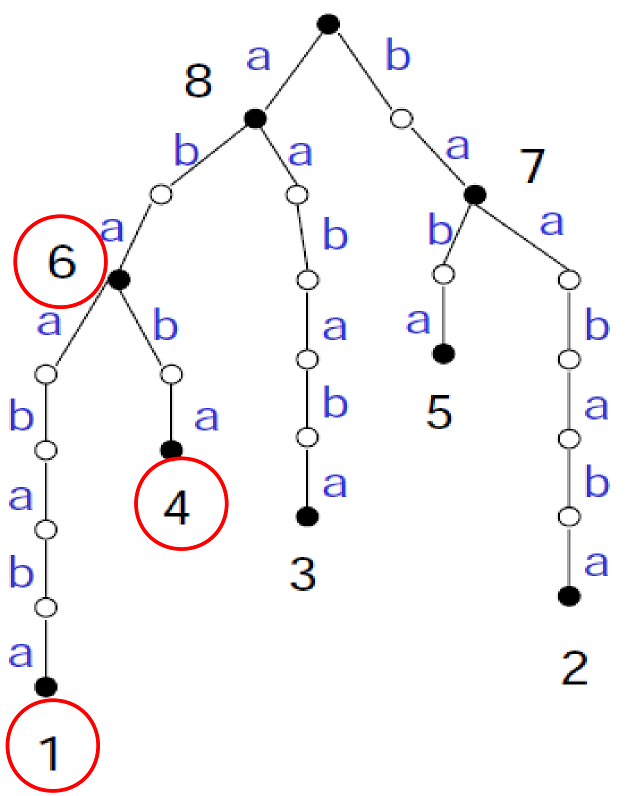
If we can count #black nodes of a subtree in constant time.

→ O(m + #occ) **retrieval time**

If we can iterate leaves of a subtree with constant delay

1. Suffix Trie

```
    12345678
T = abaababa
```



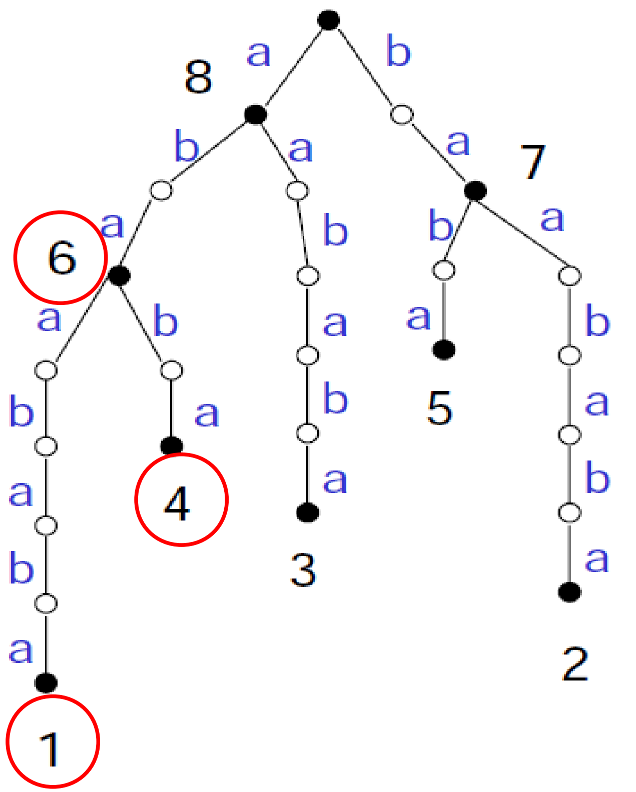
3 matches of P = “aba”

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

→ **Indexing time?**

1. Suffix Trie

```
     12345678
T  = abaababa
```



3 matches of P = “aba”

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

→ **Indexing time?**

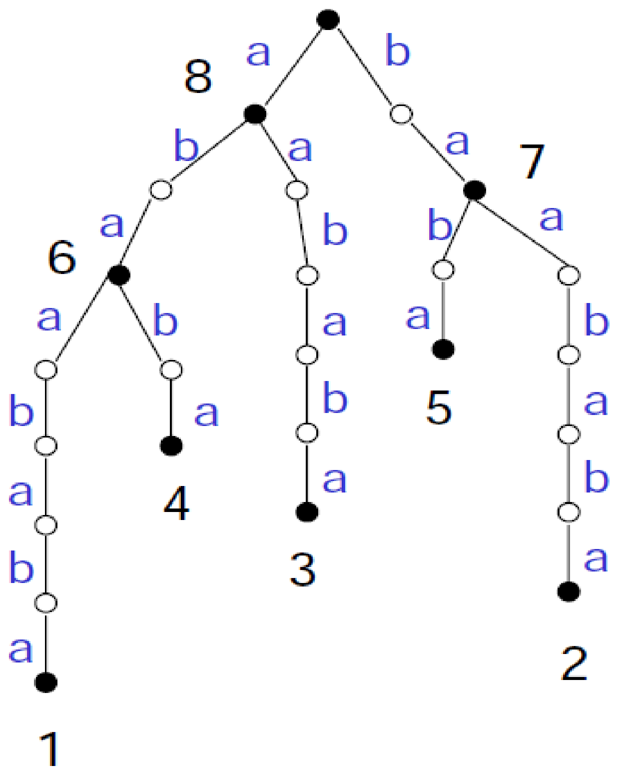
No sorting, but

→ still quadratic in m, i.e., O(m^2) :-(

→ e.g. $T = a^n b^n a^n b^n d$

2. Suffix Tree

```
     12345678
T =  abaababa
```



Suffixes

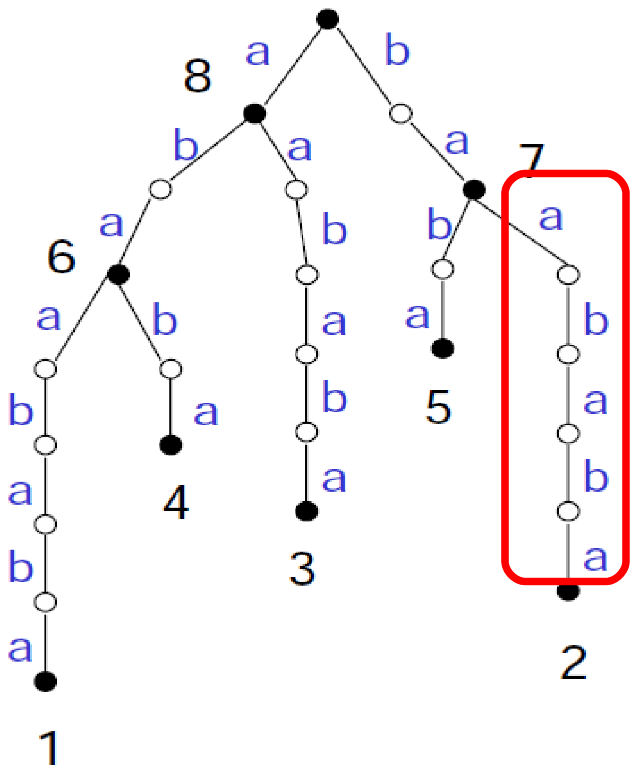
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

New Idea

→ collapse paths of white nodes!

2. Suffix Tree

12345678
T = abaababa



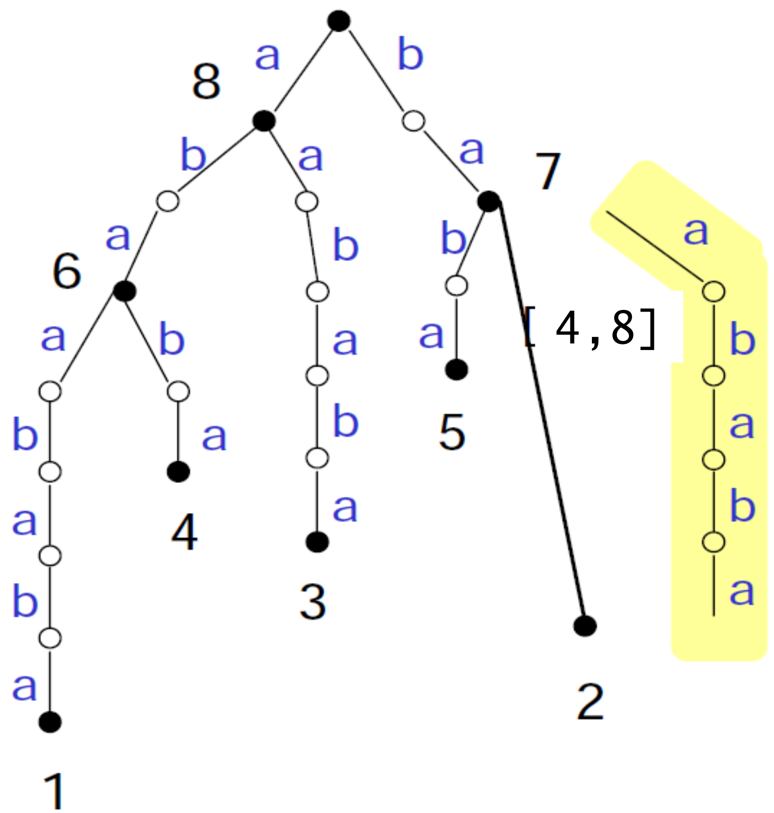
Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

New Idea

→ collapse paths of white nodes!

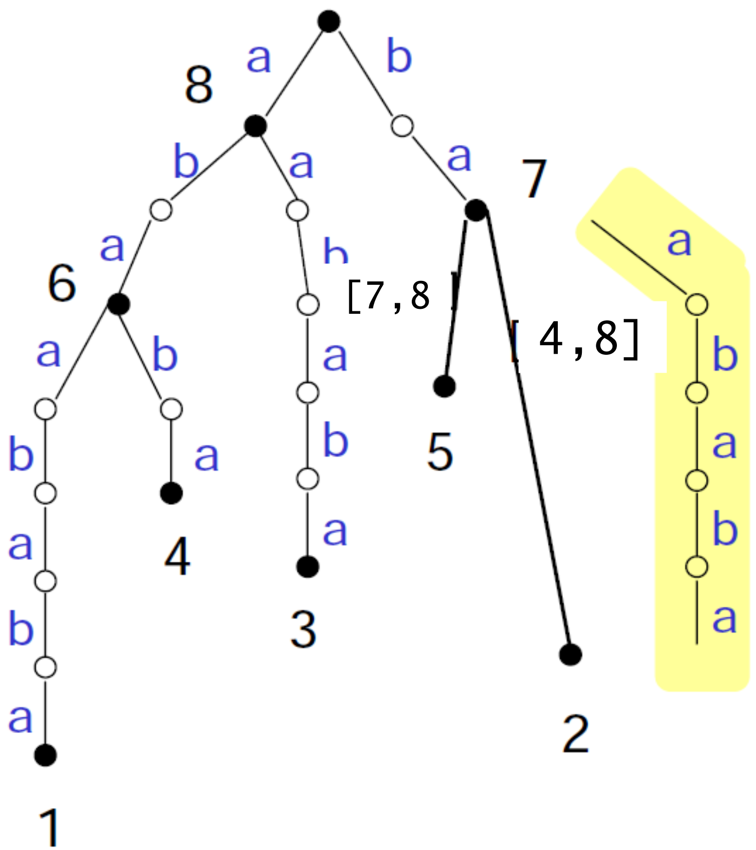
2. Suffix Tree

```
     12345678
T  = abaababa
```



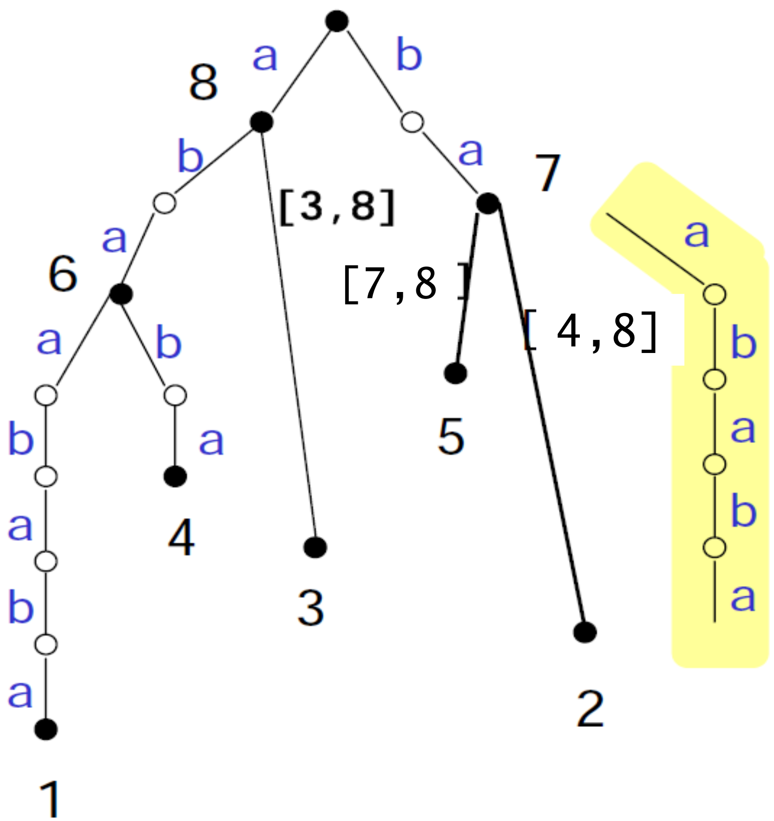
2. Suffix Tree

```
    12345678
T = abaababa
```



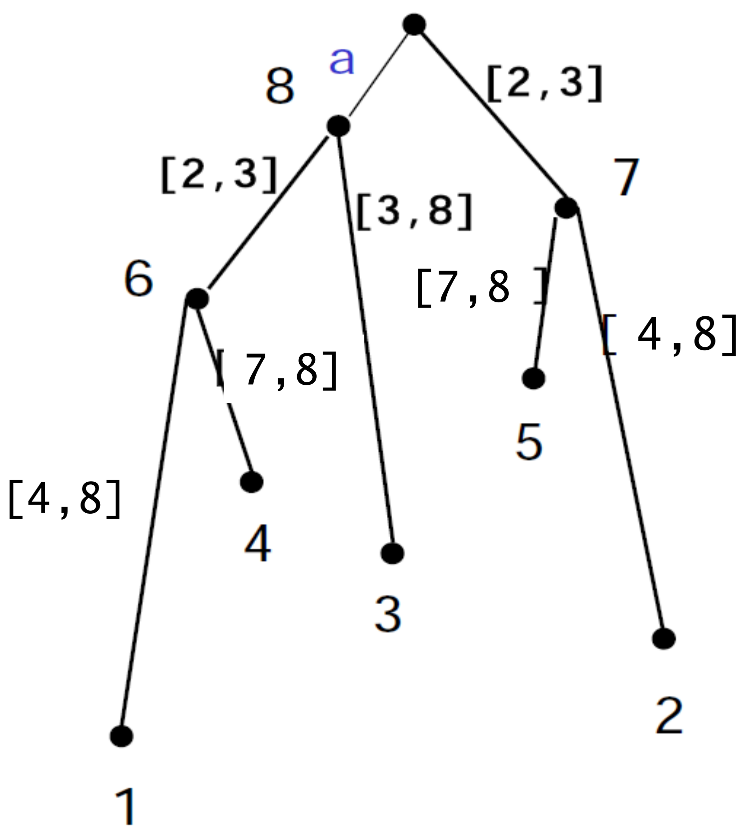
2. Suffix Tree

```
    12345678
T = abaababa
```



2. Suffix Tree

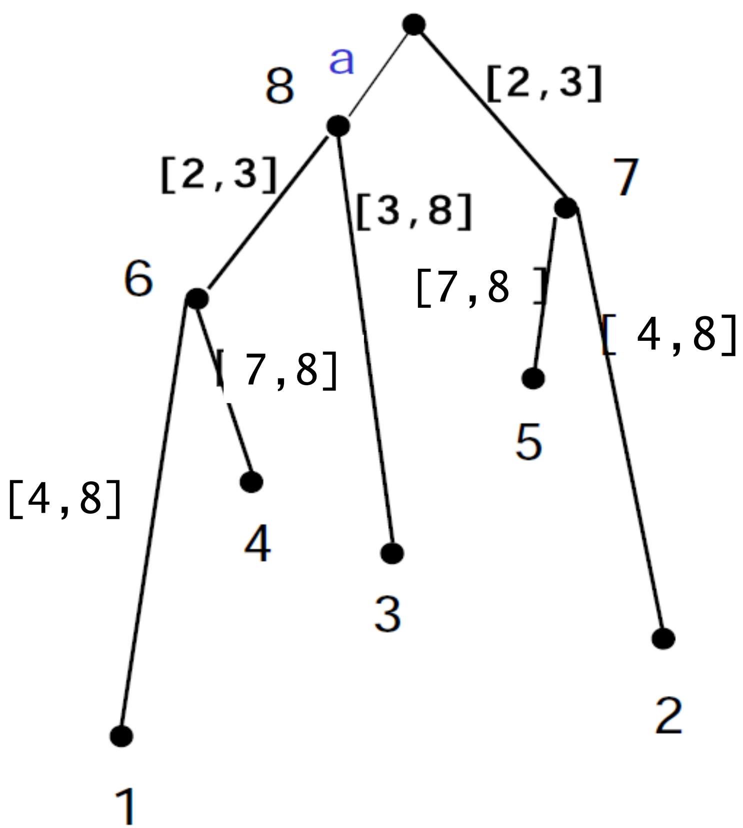
```
    12345678
T = abaababa
```



Suffix Tree of T

2. Suffix Tree

12345678
T = abaababa

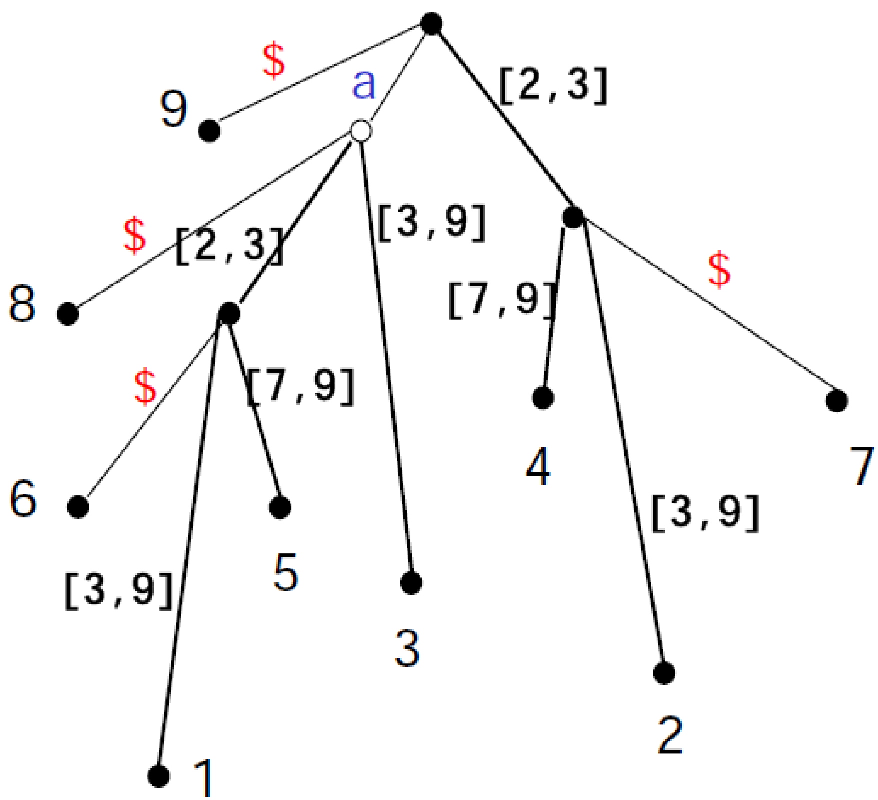


Suffix Tree of T

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

2. Suffix Tree

```
     123456789
T =  abaababa$
```



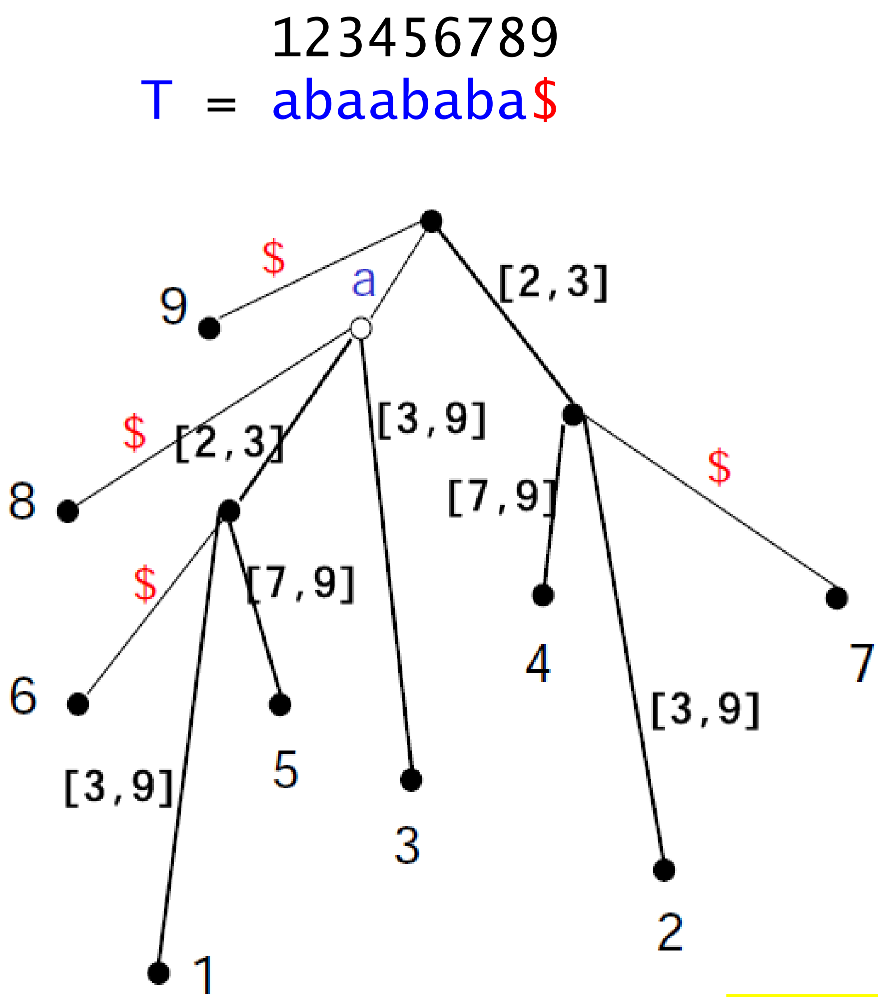
→ add end marker “$”

→ one-to-one correspondence of leaves to suffixes

→ a tree with m+1 leaves has <= 2m+1 nodes!

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

2. Suffix Tree



→ add end marker "$"

→ one-to-one correspondence of leaves to suffixes

→ a tree with m+1 leaves has <= 2m+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!

3. 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"]

3. 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

3. 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!!

3. 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)

3. 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

3. Suffix Tree Construction

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

Complex construction algorithms

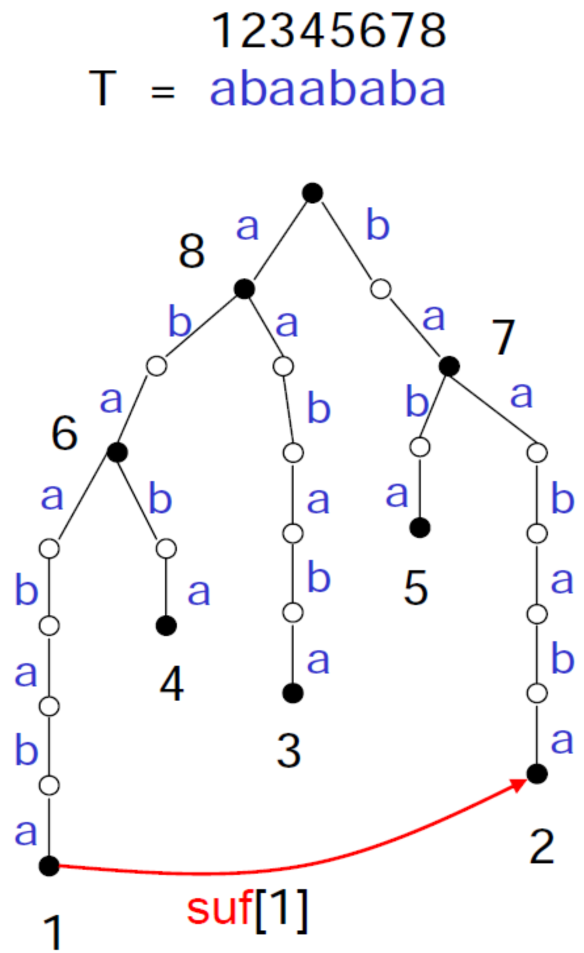
→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

→ Farach 1997

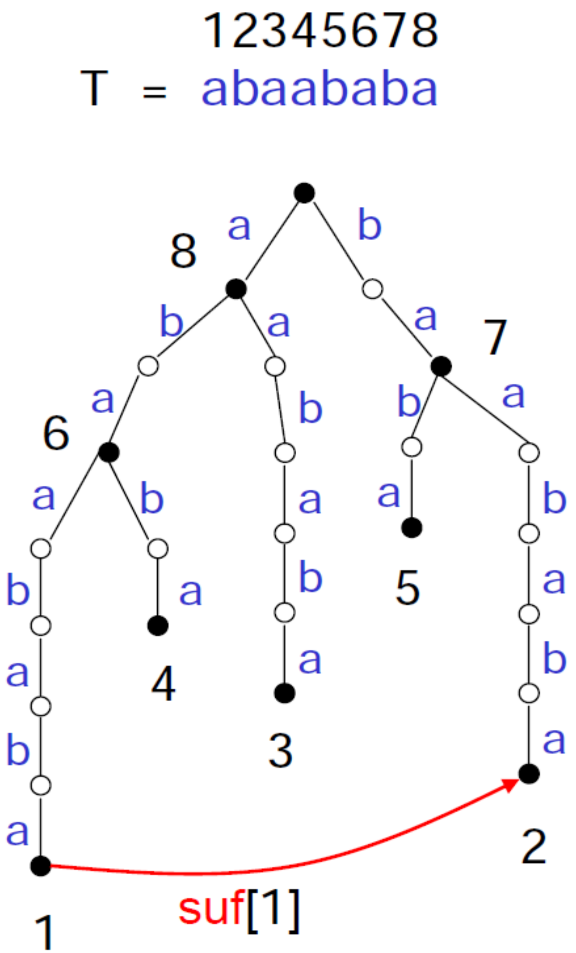
Suffix Link



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

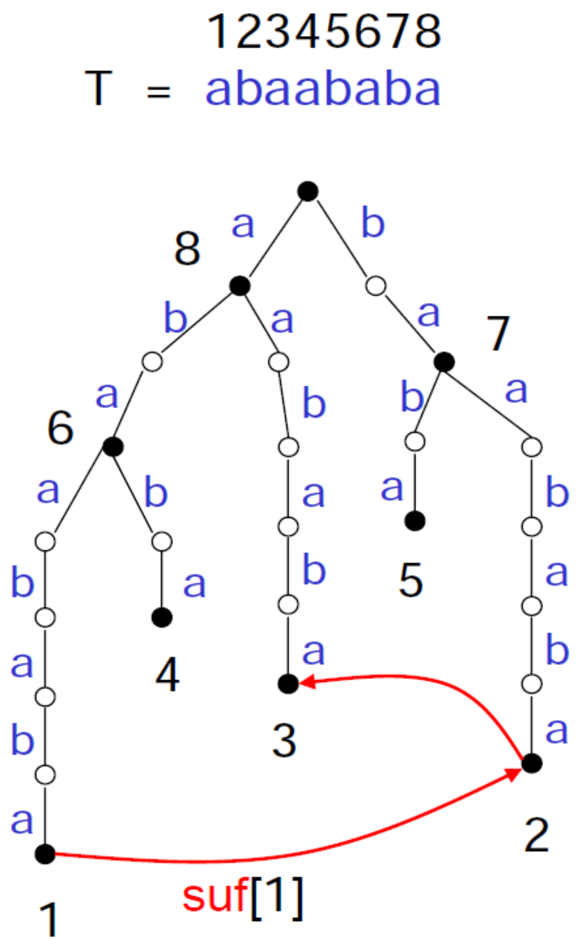


Definition

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



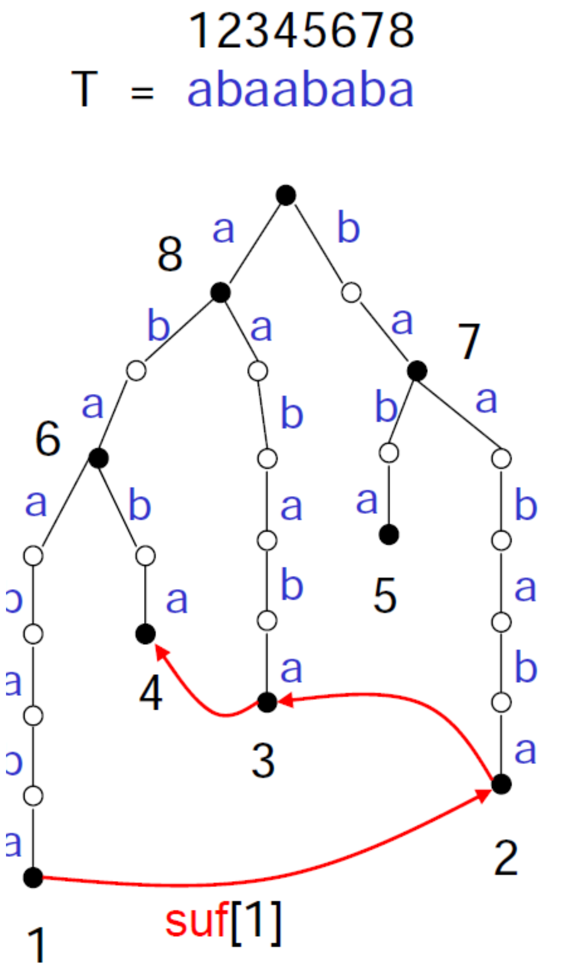
Definition

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



• essential node
○ non-essential node

Definition

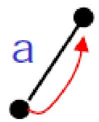
If x=ay is the string corresponding to a node u in the ST then the suffix link suf[u] is the node v corresponding to y in ST.

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.

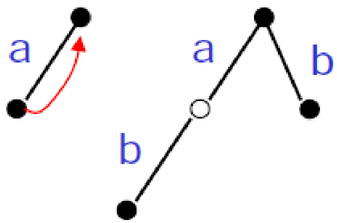
Create b-sons for v, suf[v], $suf^2[v]$, …, $suf^{k-1}[v]$

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

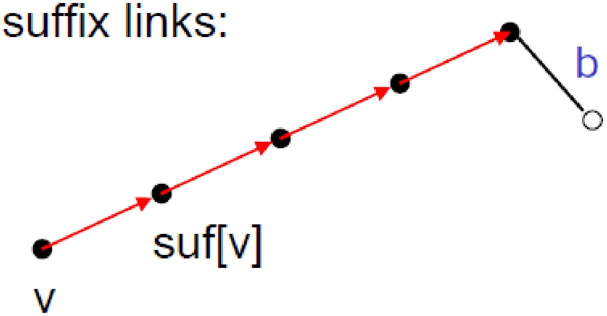
b = T[current]

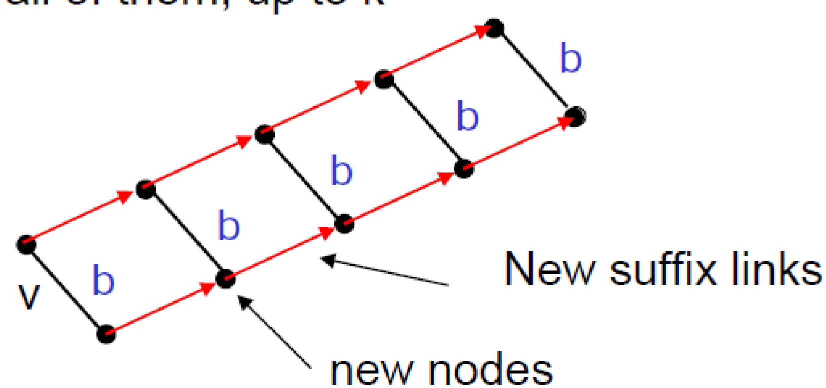
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

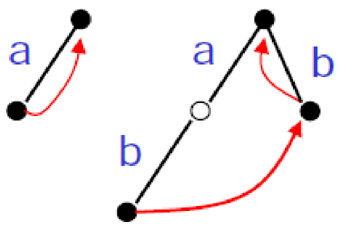




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

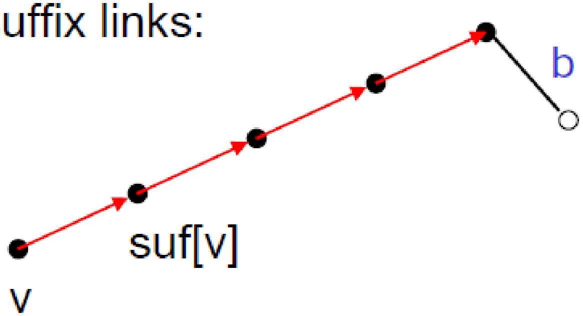
b = T[current]

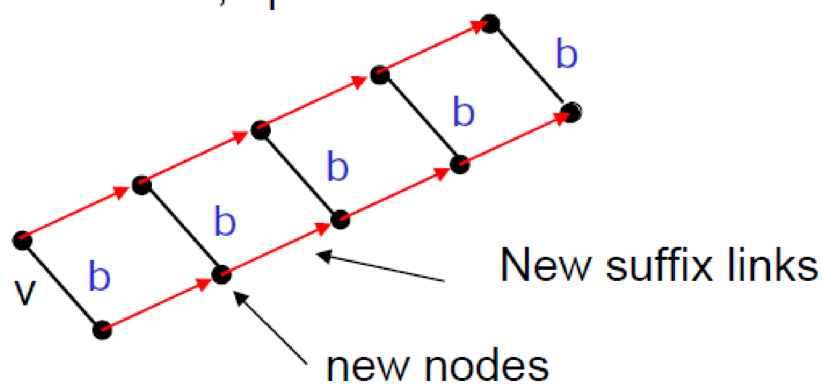
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

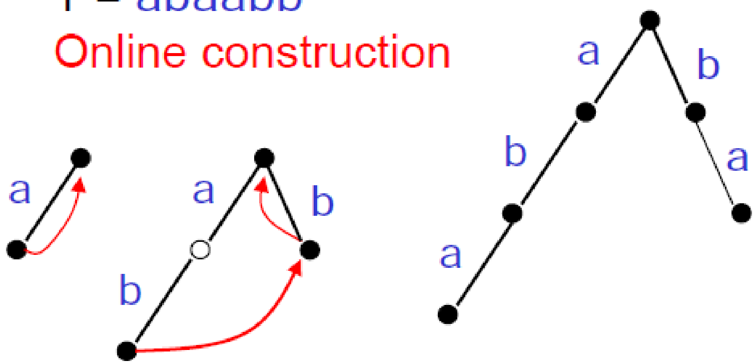




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

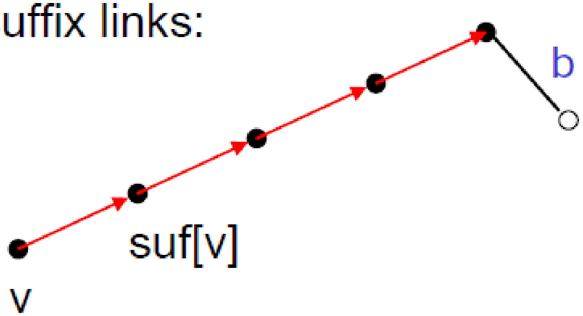
b = T[current]

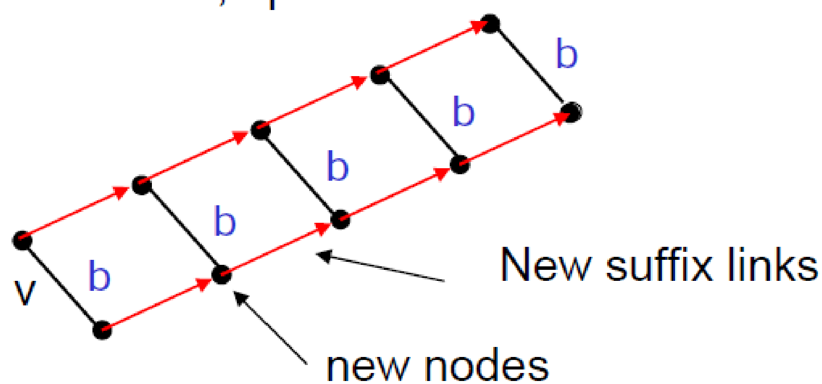
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

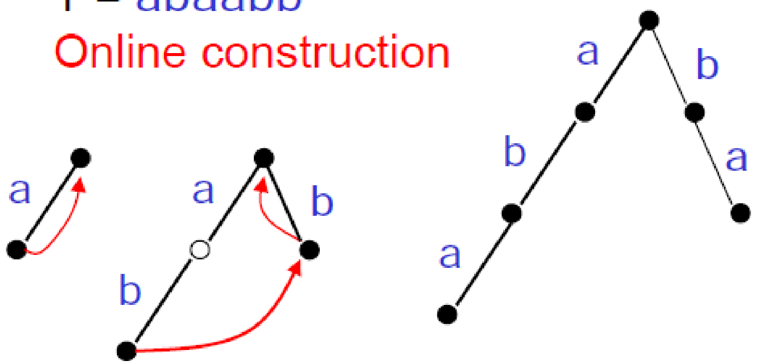




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



What are the new suffix links?

v = lowest leaf in tree

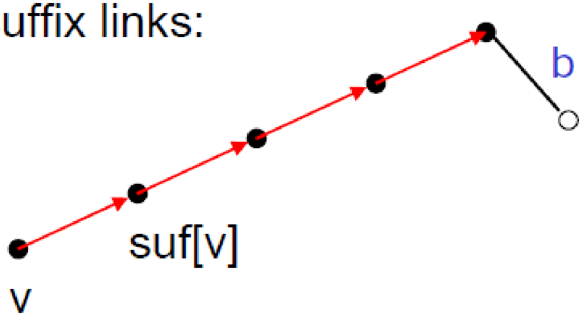
b = T[current]

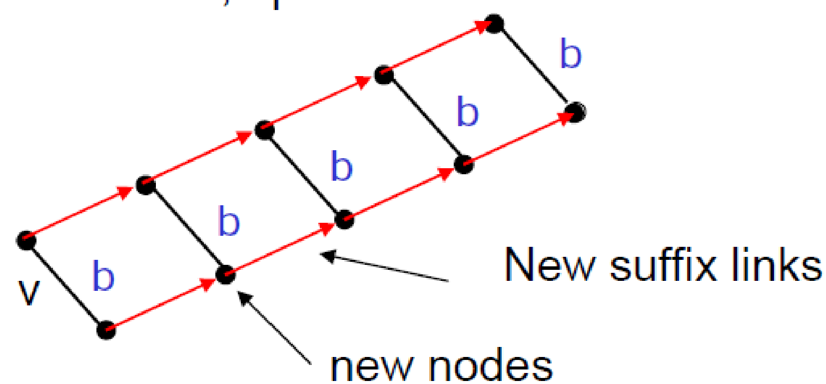
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

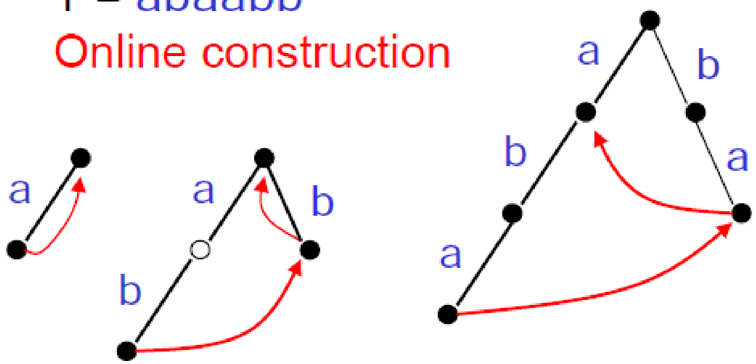




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

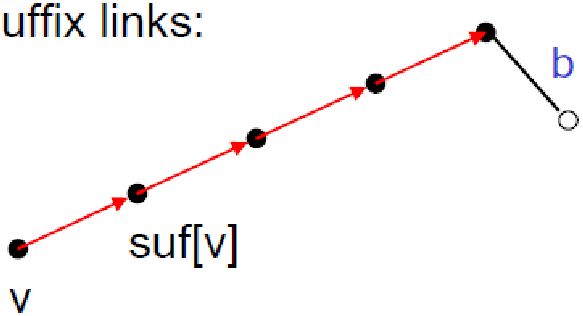
b = T[current]

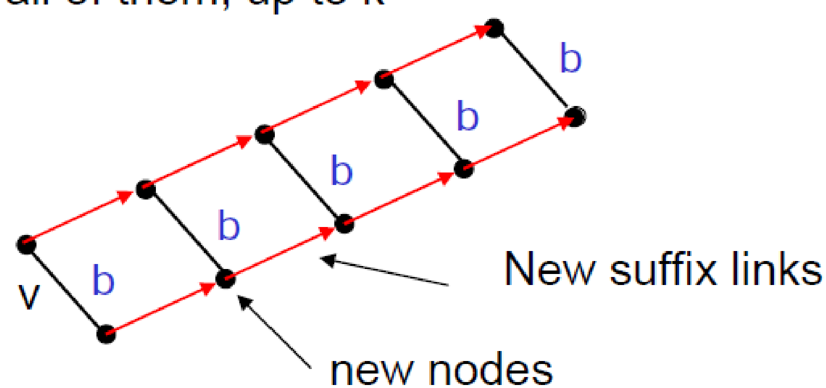
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

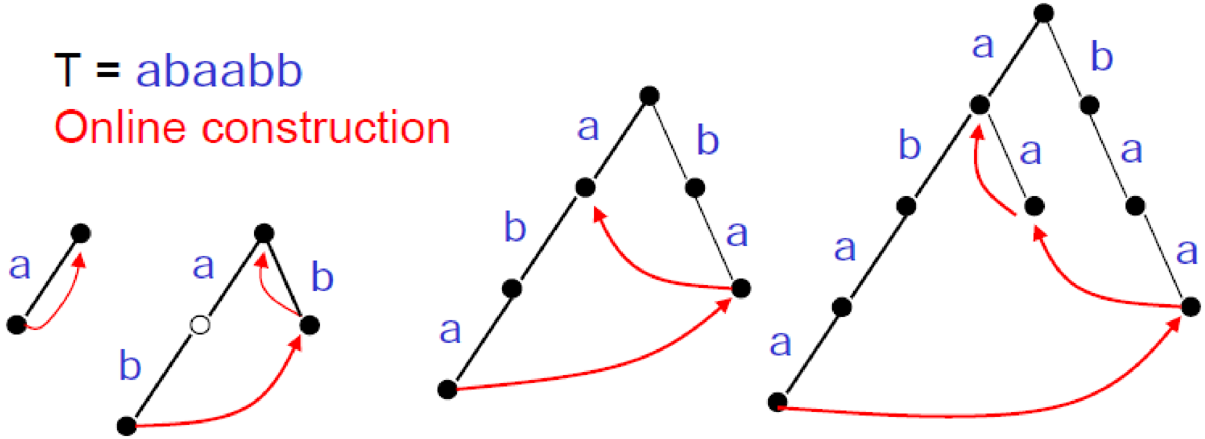




Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction



v = lowest leaf in tree

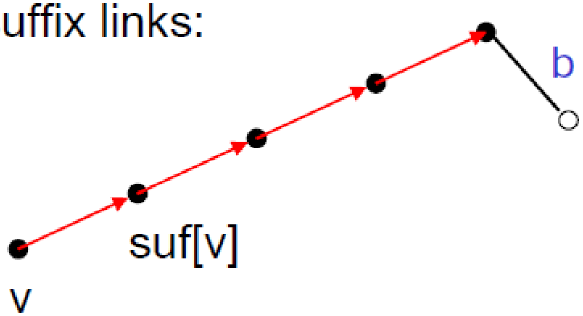
b = T[current]

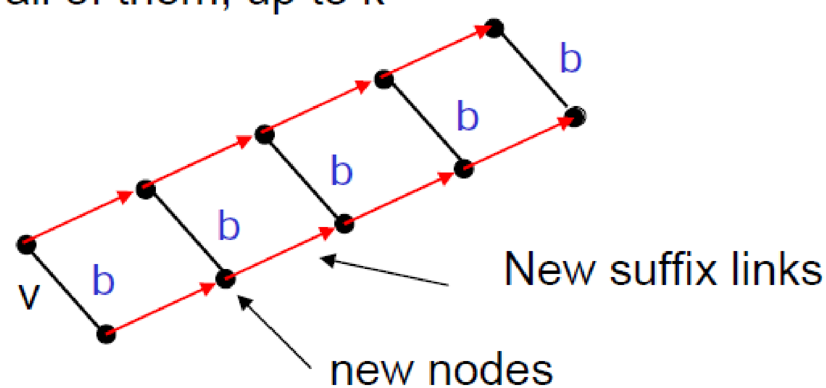
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

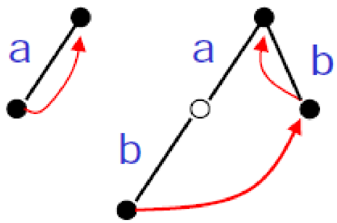


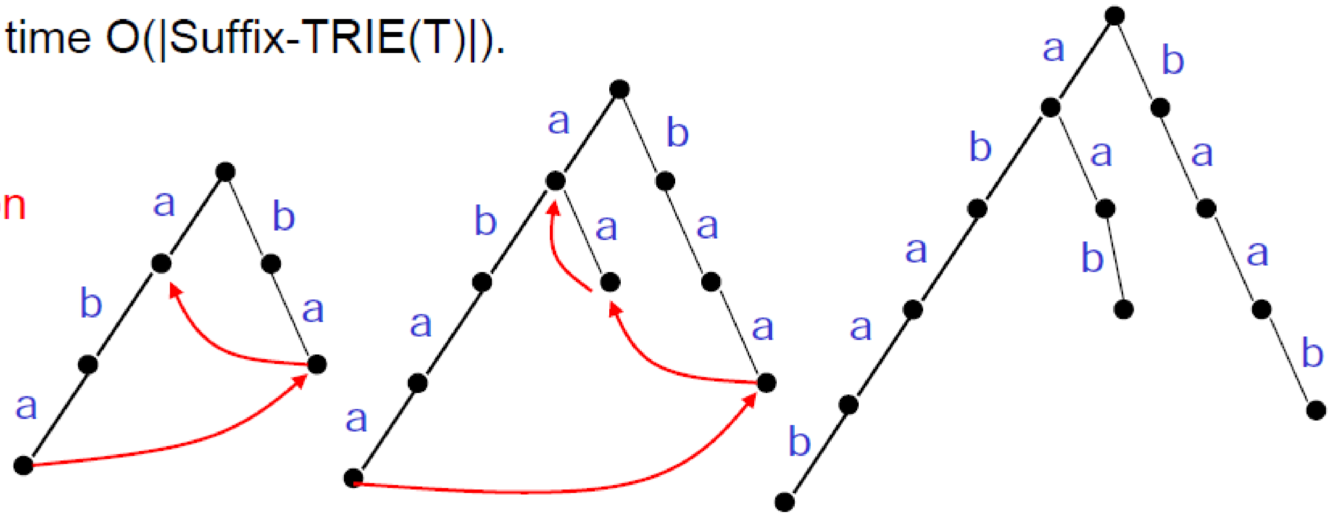


Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction





v = lowest leaf in tree

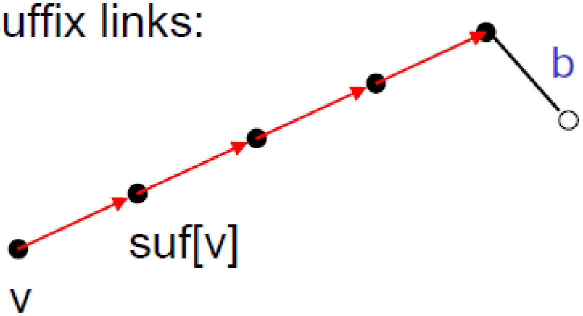
b = T[current]

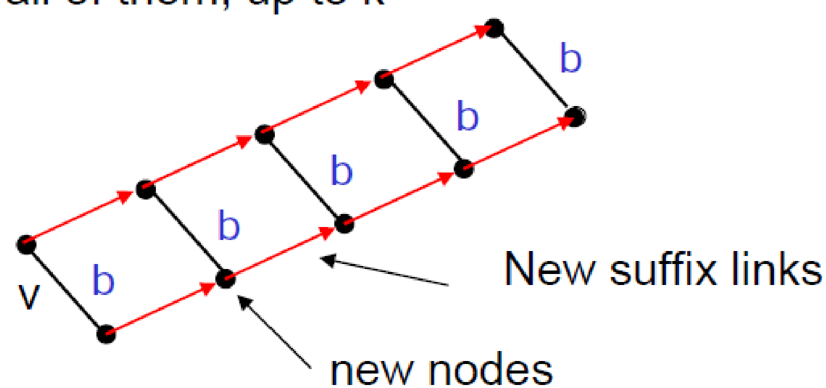
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

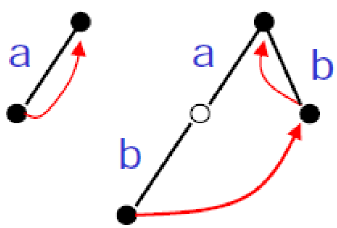


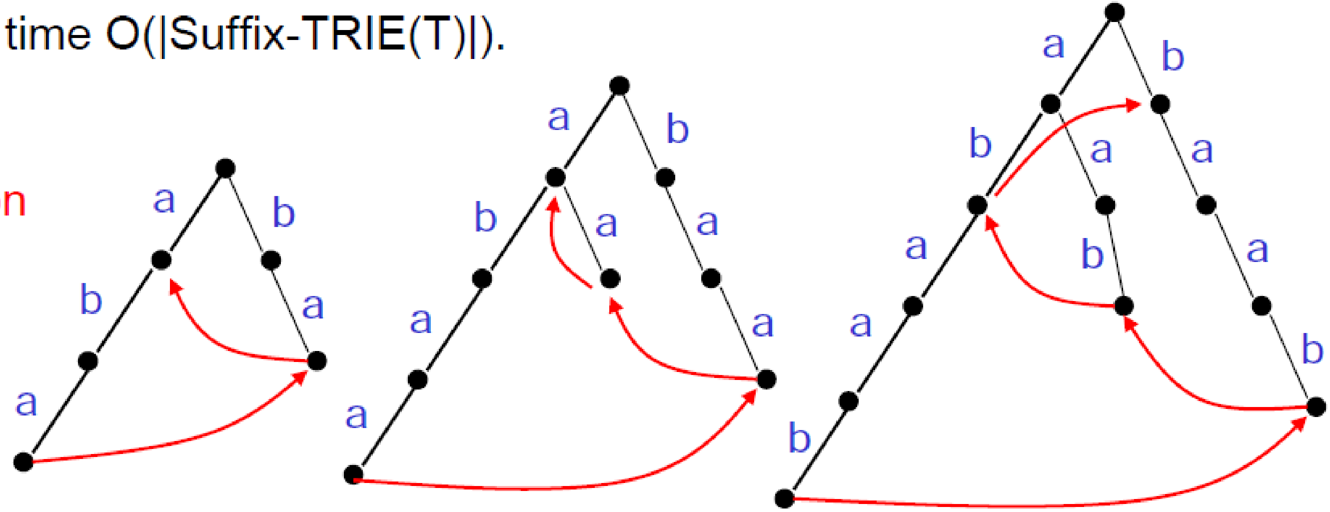


Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction





v = lowest leaf in tree

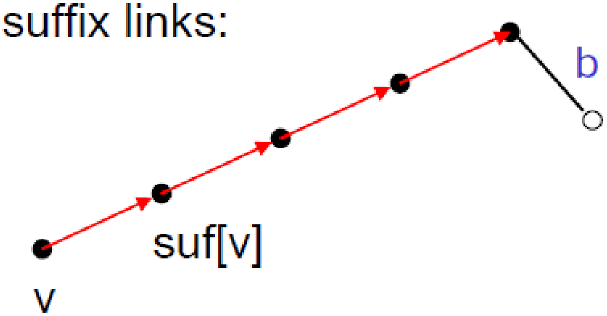
b = T[current]

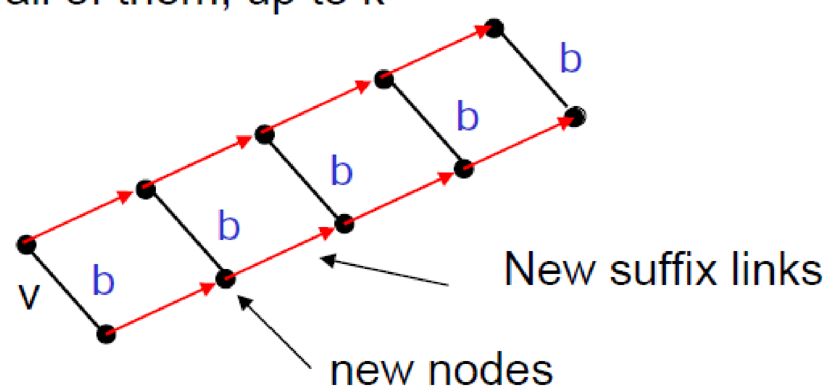
From v, follow (k times) suffix links (to u) until child(u, b) is defined.

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:

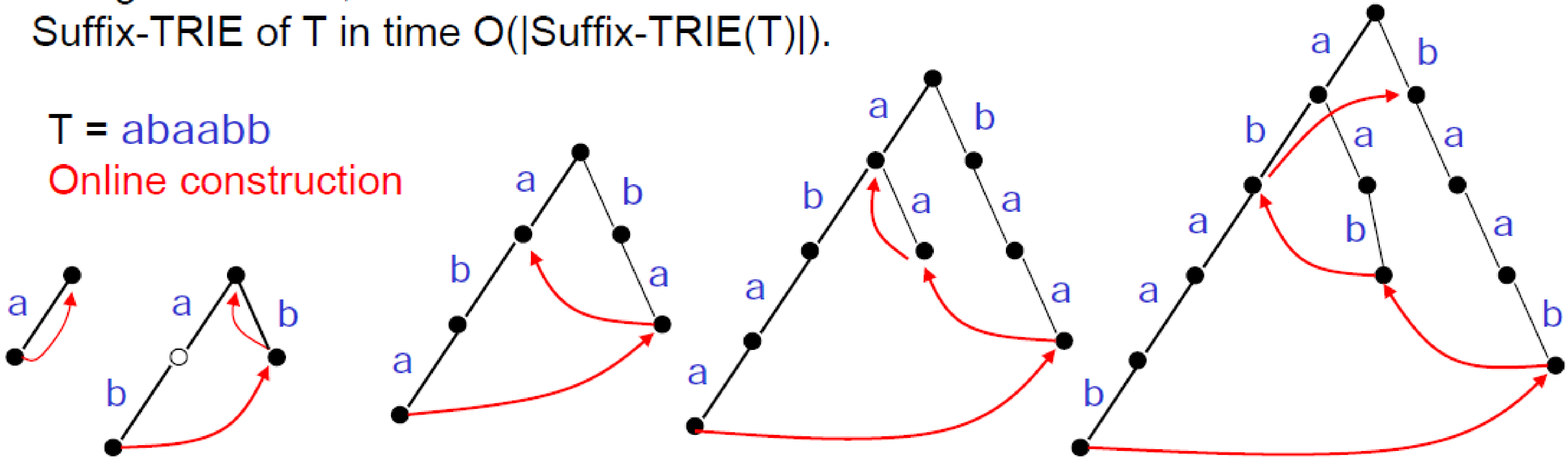


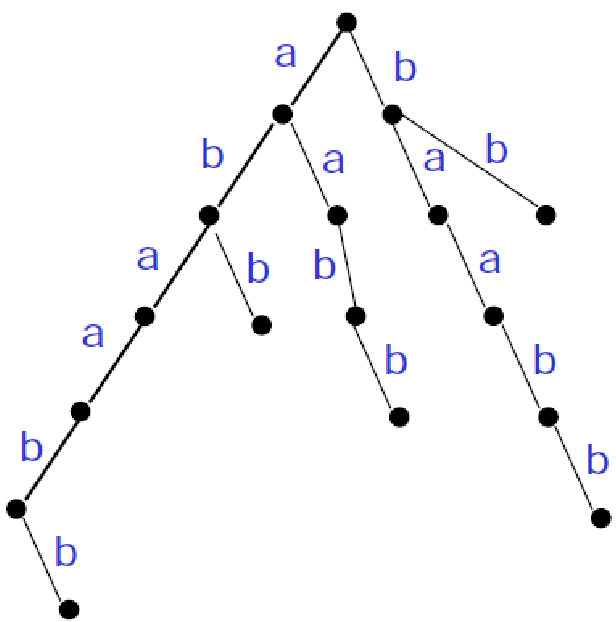


Using suffix links, we can *on-line* build the Suffix-TRIE of T in time O(|Suffix-TRIE(T)|).

T = abaabb

Online construction

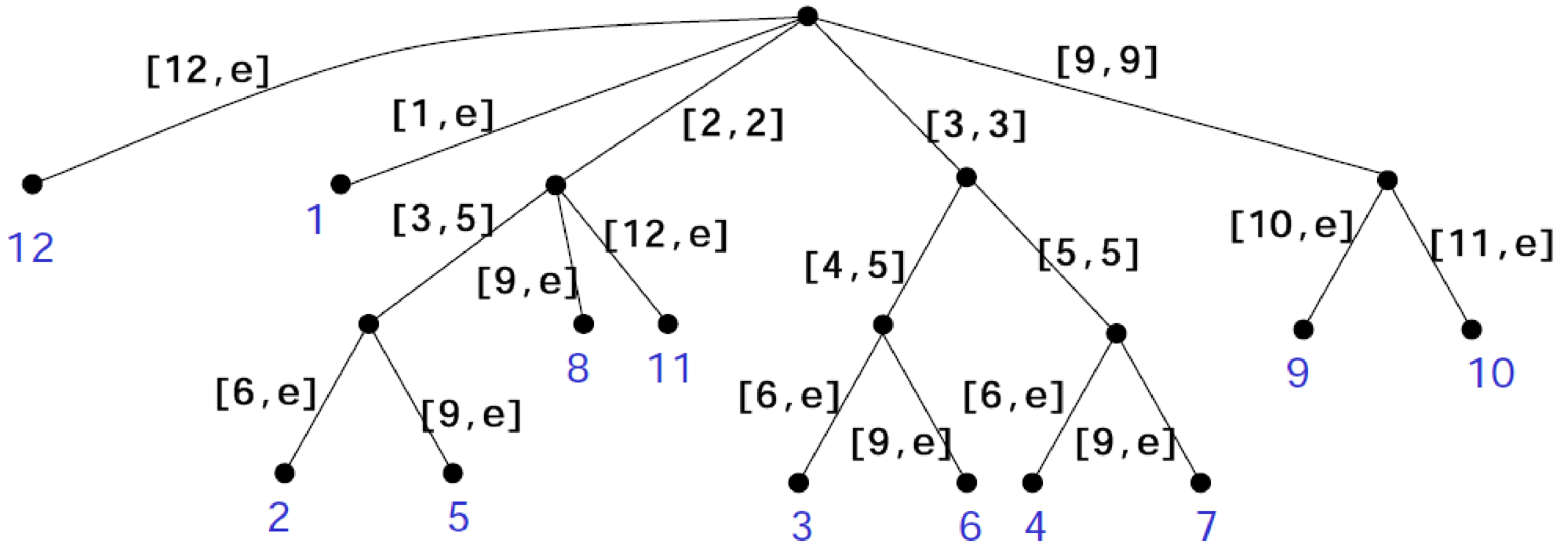




Ukkonen's on-line construction of suffix trees works in a similar way.

It maintains collapsed edges at all times.

T = mississippi$



4. Applications of Suffix Trees

Generalized Suffix tree for a SET S of strings:

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

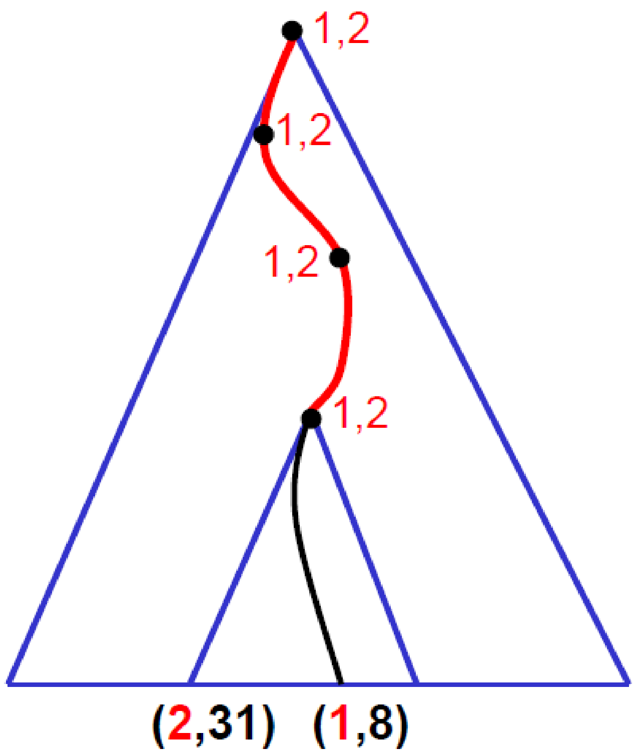
$T = S_1 \#_1 S_2 \#_2 S_3 \#_3 \ldots. S_k \#_k$

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

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

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

LZ-variant with infinite window

abaabaaabababaabb

a b a $\underbrace{\text{abaa}}$ 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

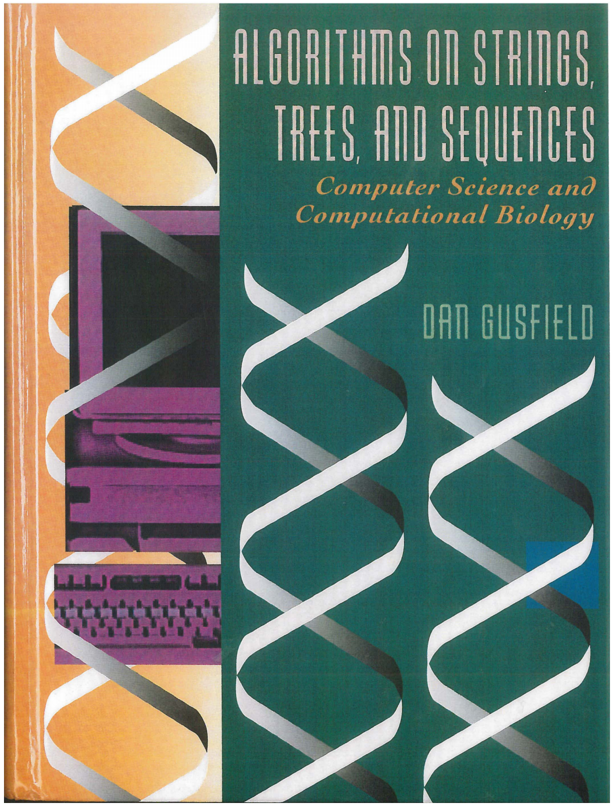
→ To find pair (x,y) at a position p in T, match T[x…] against suffix tree as long as minimal pos number is smaller than x.

 63

4. 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
→ …



7 First Applications of Suffix Trees 122

7.1 APL1: Exact string matching 122
7.2 APL2: Suffix trees and the exact set matching problem 123
7.3 APL3: The substring problem for a database of patterns 124
7.4 APL4: Longest common substring of two strings 125
7.5 APL5: Recognizing DNA contamination 125
7.6 APL6: Common substrings of more than two strings 127
7.7 APL7: Building a smaller directed graph for exact matching 129
7.8 APL8: A reverse role for suffix trees, and major space reduction 132
7.9 APL9: Space-efficient longest common substring algorithm 135
7.10 APL10: All-pairs suffix-prefix matching 135
7.11 Introduction to repetitive structures in molecular strings 138
7.12 APL11: Finding all maximal repetitive structures in linear time 143
7.13 APL12: Circular string linearization 148
7.14 APL13: Suffix arrays – more space reduction 149
7.15 APL14: Suffix trees in genome-scale projects 156
7.16 APL15: A Boyer–Moore approach to exact set matching 157
7.17 APL16: Ziv–Lempel data compression 164
7.18 APL17: Minimum length encoding of DNA 167
7.19 Additional applications 168
7.20 Exercises 168

END

Lecture 15