CS481: Bioinformatics Algorithms

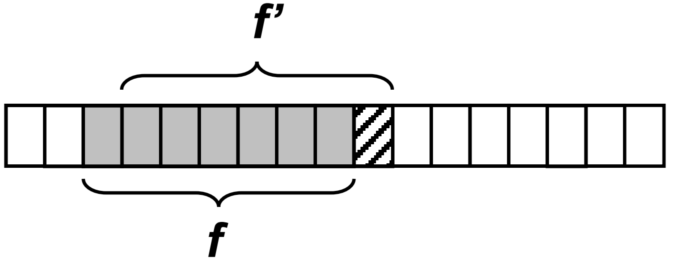
Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/

EXACT STRING MATCHING

Fingerprint idea

- Assume:
  - We can compute a fingerprint f(P) of P in O(m) time.
  - If f(P) $\neq$ f(T[s .. s+m–1]), then P $\neq$ T[s .. s+m–1]
  - We can compare fingerprints in O(1)
  - We can compute f' = f(T[s+1.. s+m]) from f(T[s .. s+m–1]), in O(1)

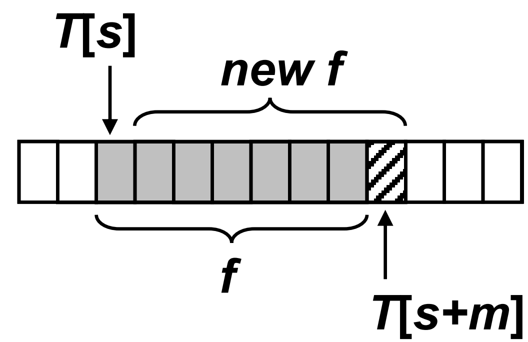


Algorithm with Fingerprints

- Let the alphabet Σ ={0,1,2,3,4,5,6,7,8,9}
- Let fingerprint to be just a decimal number, i.e., f(“1045”) = 1*103 + 0*102 + 4*101 + 5 = 1045

- **Fingerprint-Search(T,P)**

```
fp ← compute f(P)
f ← compute f(T[0..m-1])
for s ← 0 to n – m do
    if fp = f return s
    f ← (f – T[s]*10^(m-1))*10 + T[s+m]
return -1
```



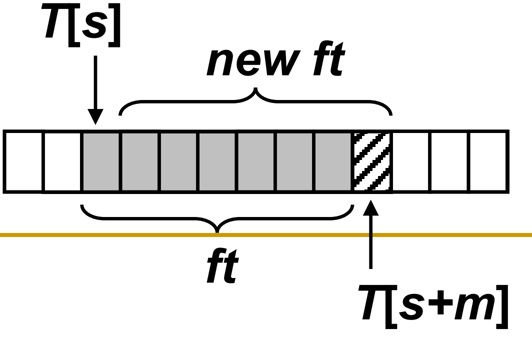
- **Running time 2*O*(*m*) + *O*(*n*–*m*) = *O*(*n*)**

Using a Hash Function

- Problem:
  - we can not assume we can do arithmetics with m-digits-long numbers in O(1) time
- Solution: Use a hash function h = f mod q
  - For example, if q = 7, h("52") = 52 mod 7 = 3
  - $h(S1) \neq h(S2) \Rightarrow S1 \neq S2$
  - But h(S1) = h(S2) does not imply S1=S2
    - For example, if q = 7, h("73") = 3, but "73" $\neq$ "52"
- Basic "mod q" arithmetics:
  - (a+b) mod q = (a mod q + b mod q) mod q
  - (a*b) mod q = (a mod q)*(b mod q) mod q

Preprocessing and Stepping

- Preprocessing:
  - fp = P[m-1] + 10*(P[m-2] + 10*(P[m-3]+ … + 10*(P[1] + 10*P[0])…)) mod q
  - In the same way compute ft from T[0..m-1]
  - Example: P = “2531”, q = 7, fp = ?
- Stepping:
  - ft = (ft – T[s]*10^{m-1}mod q)*10 + T[s+m]) mod q
  - 10^{m-1} mod q can be computed once in the preprocessing
  - Example: Let T[…] = “5319”, q = 7, what is the corresponding ft?



Stepping

- T = 25319446766…, m = 4, q=7
- T_0 = "2531"
  - ft = 2531 mod 7 = 4
- T_1 = "5319"
  - ft = ((ft – T[s]*(10^{m-1} mod q))*10 + T[s+m]) mod q
  - ft = ((ft – T[0]*(10^3 mod 7))*10 + T[0+4]) mod 7
    - = ((4 – (2*1000 mod 7)) * 10 + T[4]) mod 7
    - = ((4-(2*6))*10+6) mod 7 = (-8*10+ 9) mod 7
    - = -71 mod 7 = 6
  - 5319 mod 7 = 6

Rabin-Karp Algorithm

```
Rabin-Karp-Search(T,P)
q ← a prime larger than m
c ← 10^(m-1) mod q  // run a loop multiplying by 10 mod q
fp ← 0; ft ← 0
for i ← 0 to m-1  // preprocessing
   fp ← (10*fp + P[i]) mod q
   ft ← (10*ft + T[i]) mod q
for s ← 0 to n – m  // matching
   if fp = ft then   // run a loop to compare strings
      if P[0..m-1] = T[s..s+m-1] return s
   ft ← ((ft – T[s]*c)*10 + T[s+m]) mod q
return -1
```

Analysis

- If q is a prime, the hash function distributes m-digit strings evenly among the q values
  - Thus, only every q^{th} value of shift s will result in matching fingerprints (which will require comparing strings with O(m) comparisons)
- Expected running time (if q > m):
  - Preprocessing: O(m)
  - Outer loop: O(n-m)
  - All inner loops: $\frac{n -}{q} m = \qquad -$
  - Total time: O(n-m)
- Worst-case running time: O(nm)

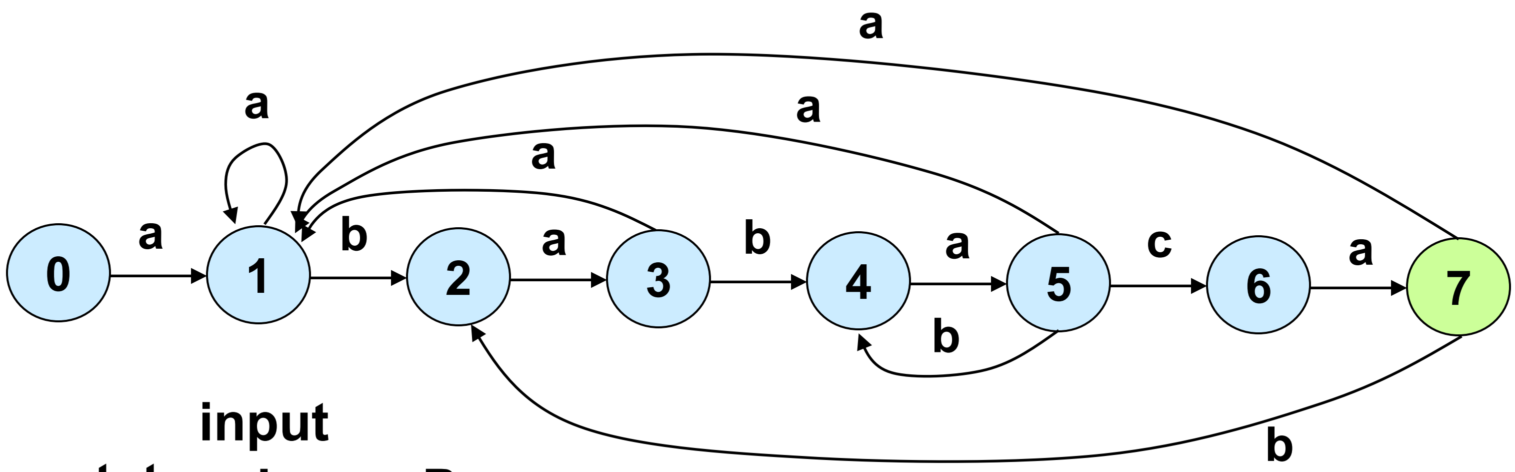
Rabin-Karp in Practice

- If the alphabet has d characters, interpret characters as radix-d digits (replace 10 with d in the algorithm).
- Choosing prime q > m can be done with randomized algorithms in O(m), or q can be fixed to be the largest prime so that 10*q fits in a computer word.

Searching in n comparisons

- The goal: each character of the text is compared only once!
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - T = "Tweedledee and Tweedledum" and P = "Tweedledum"
    - T = "pappappappar" and P = "pappar"

Finite automaton search



input

state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i --		1	2	3	4	5	6	7	8	9	10	11
T[i] --		a	b	a	b	a	b	a	c	a	b	a
state $\phi(i)$	0	1	2	3	4	5	4	5	6	7	2	3

Processing time takes $\Theta(n)$.
But have to first construct FA.
Main Issue: How to construct FA?

Need some Notation …

$\phi(w)$ = state FA ends up in after processing w.

Example: $\phi(abab) = 4$.

$\sigma(x) = \max\{k: P_k \underline{suf}\ x\}$. Called the suffix function.

Examples: Let P = ab.

$\sigma(\varepsilon) = 0$

$\sigma(ccaca) = 1$

$\sigma(ccab) = 2$

Note: If $|P| = m$, then $\sigma(x) = m$ indicates a match.

T: a b a b b a b b a c …

States: 0 1……m………m……….

match match

FA Construction

Given: P[1..m]   Let Q = states = {0, 1, …, m}.

initial   final

Define transition function δ as follows:

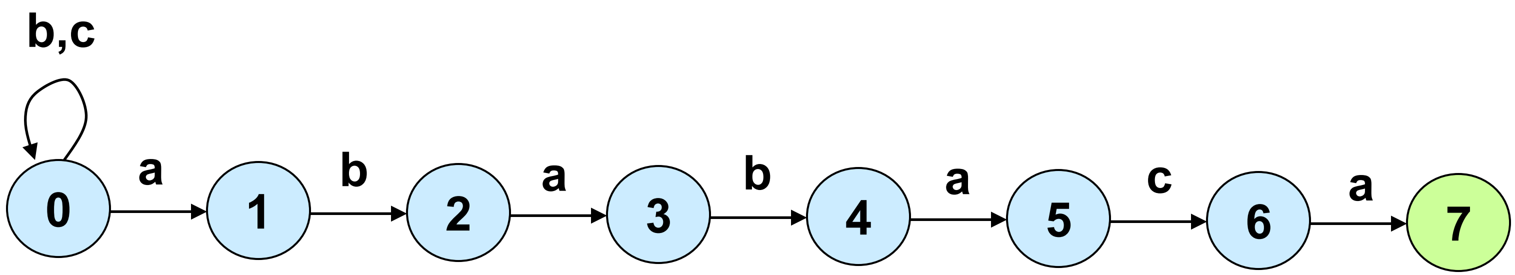
$\delta(q, a) = \sigma(P_q a)$ for each q and a.

Example: P = ababaca

$$\delta(5, b) = \sigma(P_5 b)$$
$$= \sigma(ababab)$$
$$= 4$$

Intuition: Encountering a 'b' in state 5 means the current substring doesn't match. But, you know this substring ends with "abab" -- and this is the longest suffix that matches the beginning of P. Thus, we go to state 4 and continue processing "abab…" .

P=ababaca

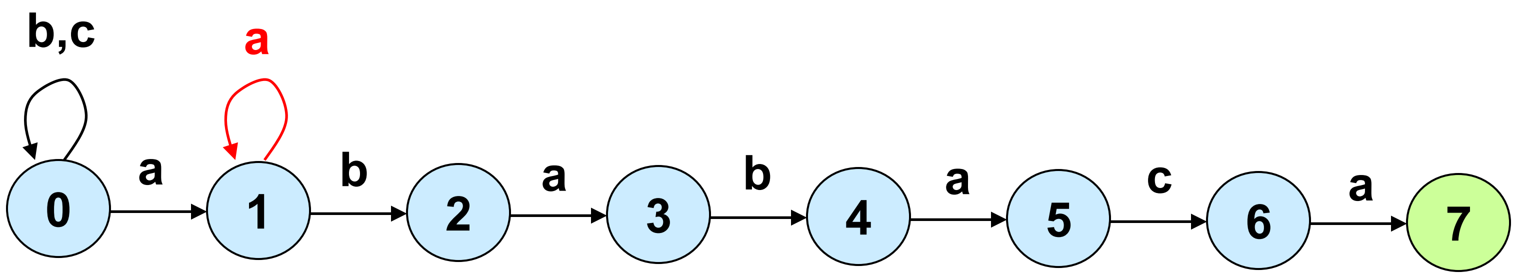


m=7; Q={0,1,2,3,4,5,6,7)

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

P=ababaca

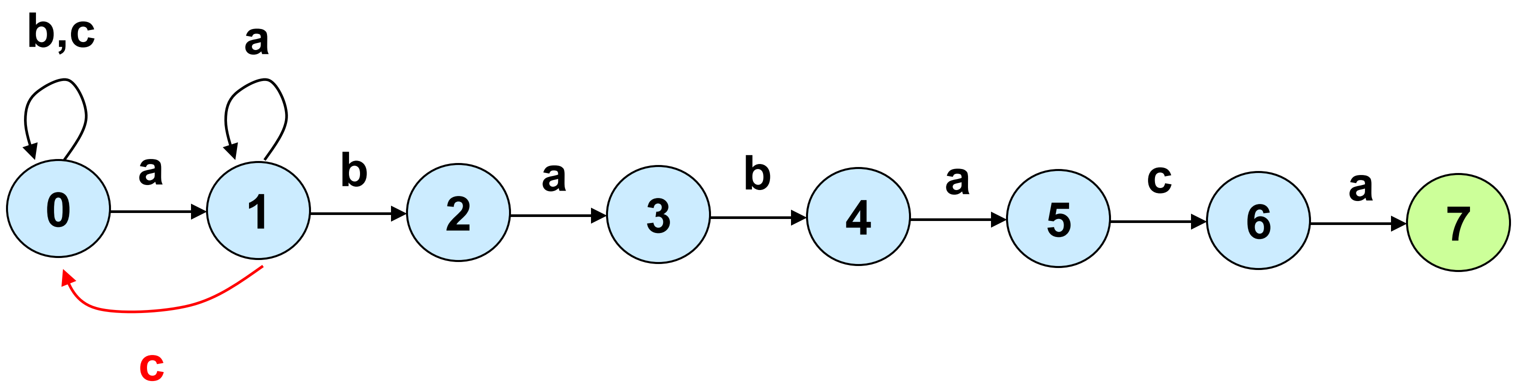


$\delta(1, a) = \sigma(P_1 a) = \sigma(aa) = \sigma(a) = 1$

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

P=ababaca

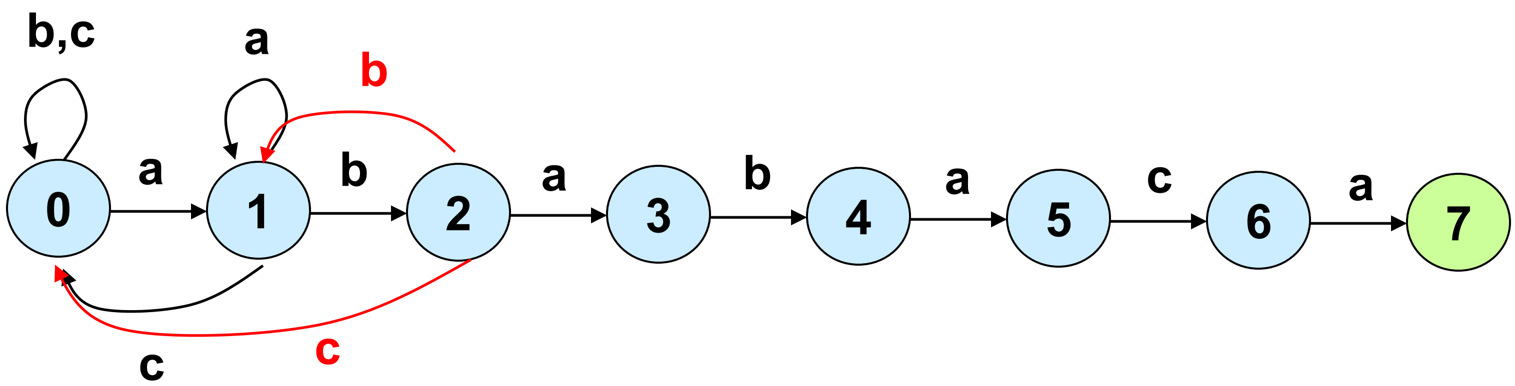


$\delta(1, a) = \sigma(P_1 a) = \sigma(aa) = \sigma(a) = 1$
$\delta(1, c) = \sigma(P_1 c) = \sigma(ac) = 0$

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

P=ababaca

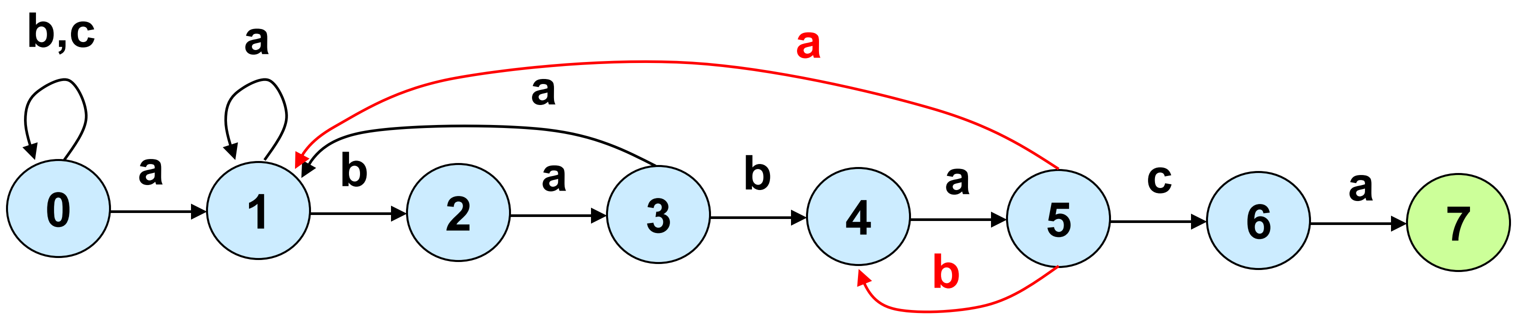


$\delta(2, b) = \sigma(P_2 b) = \sigma(aba) = \sigma(a) = 1$
$\delta(2, c) = \sigma(P_2 c) = \sigma(abc) = 0$

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

P=ababaca (fast forward & simplified)

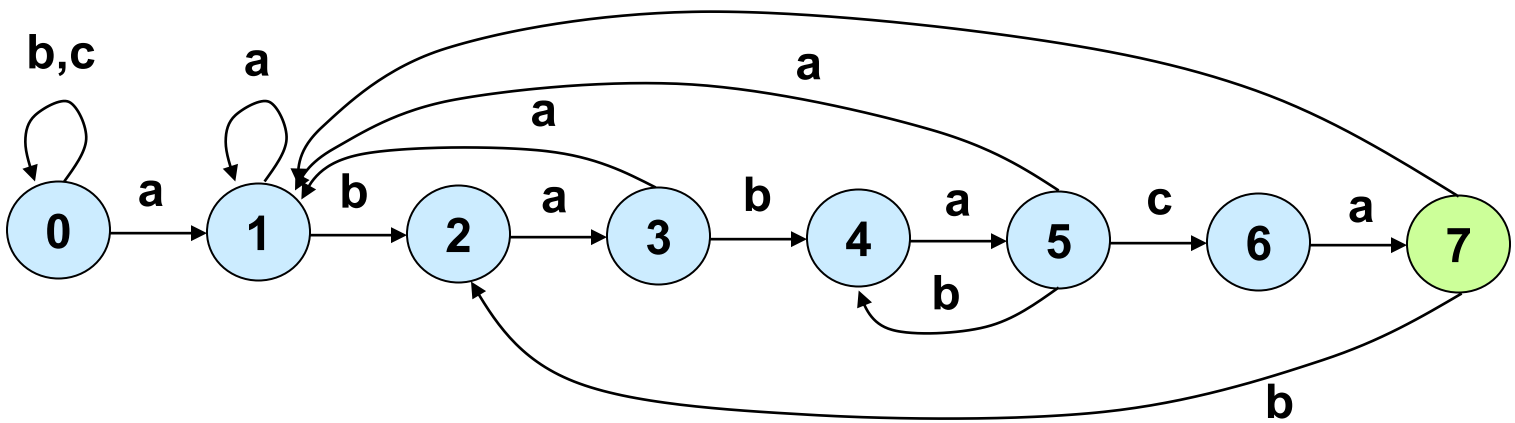


$\delta(5, a) = \sigma(P_5 a) = \sigma(ababaa) = \sigma(a) = 1$
$\delta(5, b) = \sigma(P_5 b) = \sigma(ababab) = \sigma(abab) = 4$

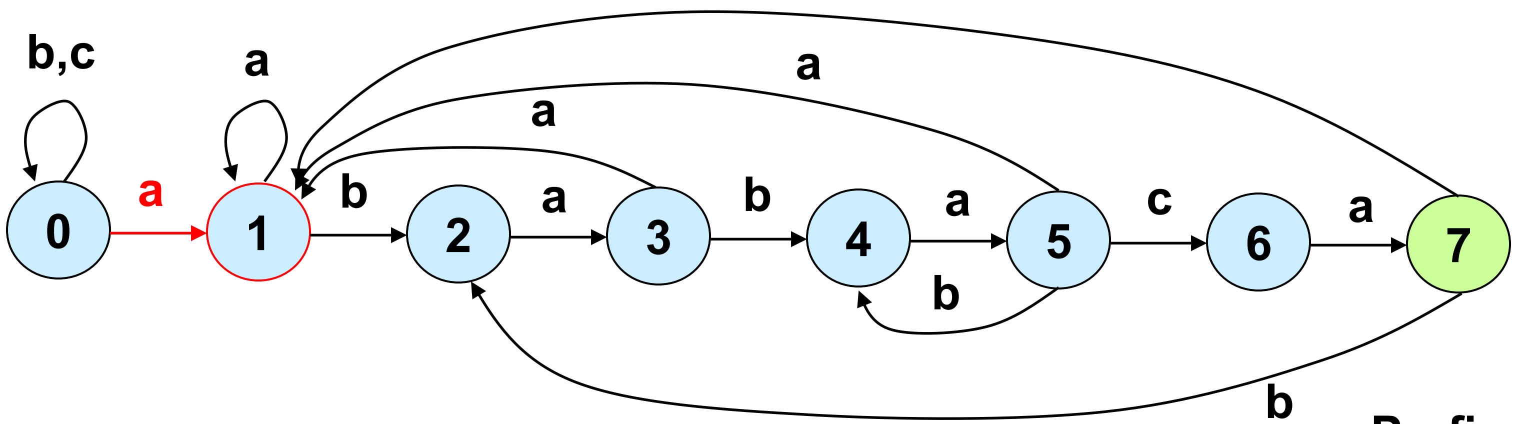
Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

P=ababaca (final, simplified)



Search



T= ababababacaba

Prefixes

a

ab

aba

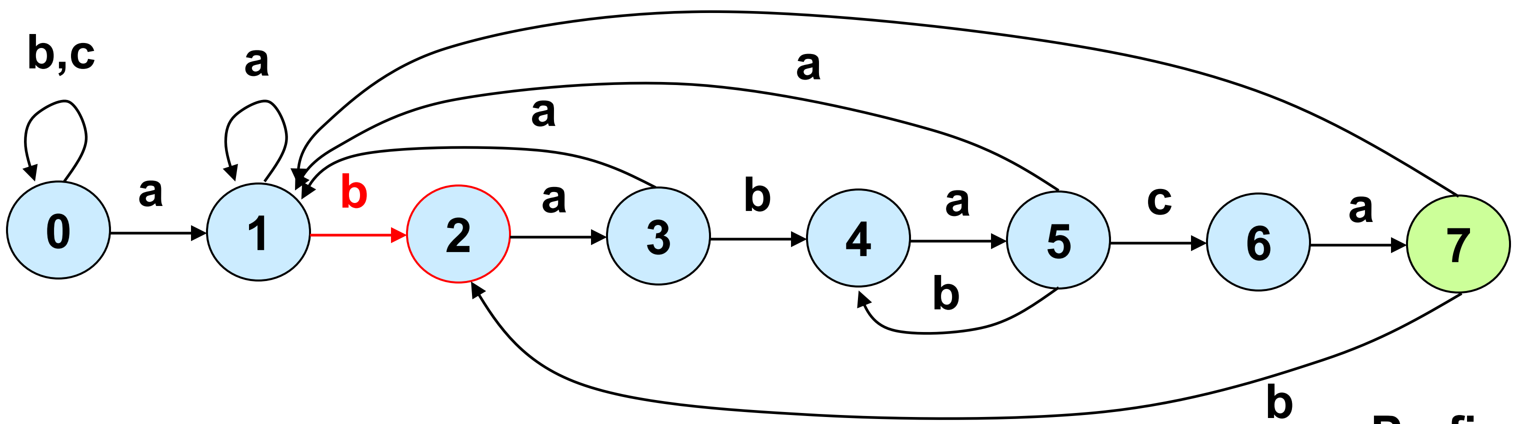
abab

ababa

ababac

ababaca

Search

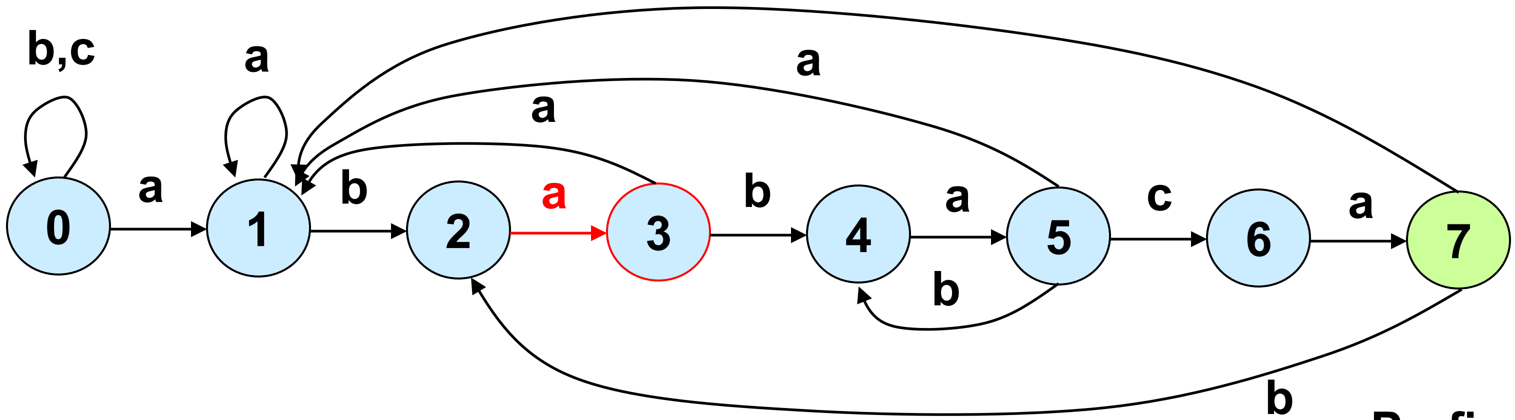


T= ababababacaba

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

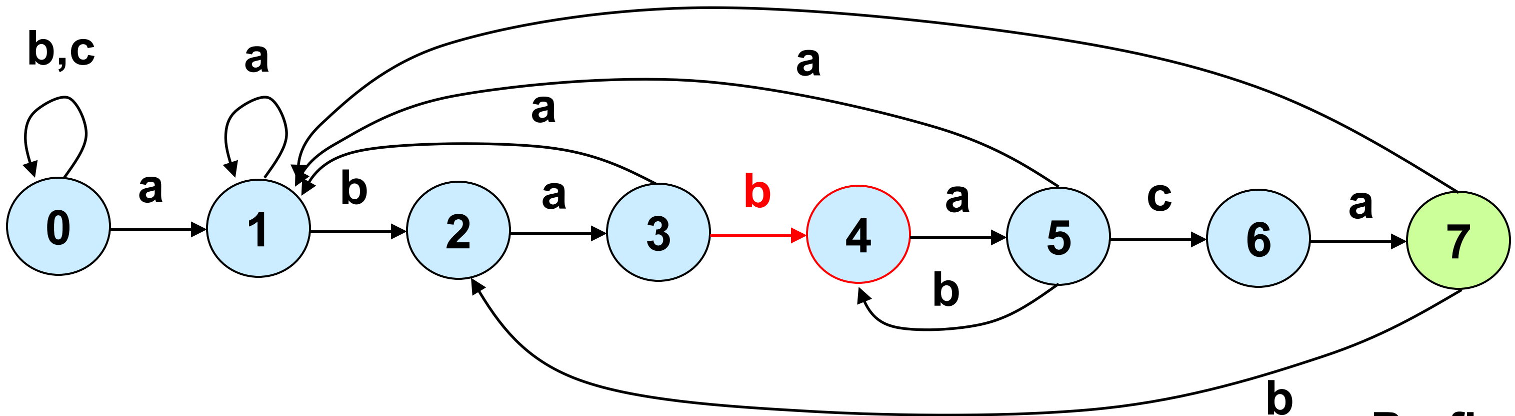
Search



T= ababababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca

Search



T= ababababacaba

<u>Prefixes</u>

a

ab

aba

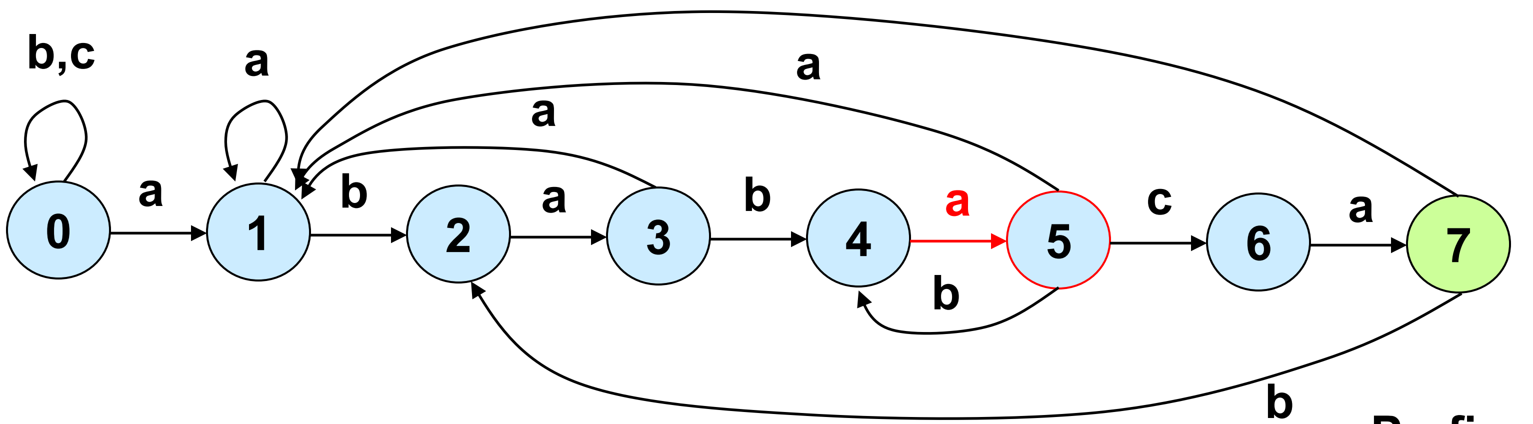
abab

ababa

ababac

ababaca

Search

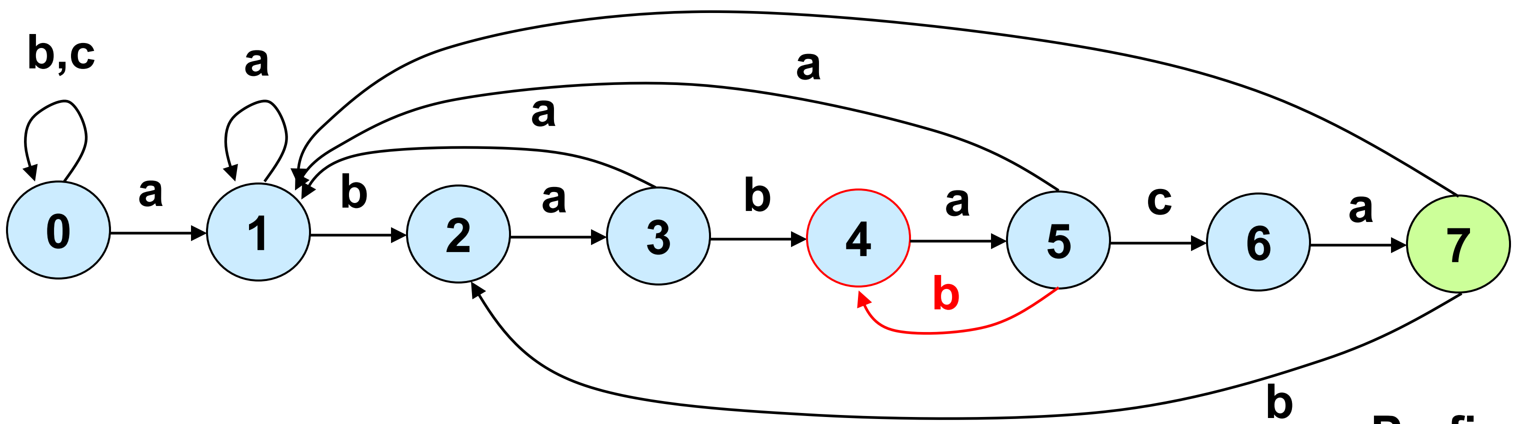


T= ababababacaba

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

Search

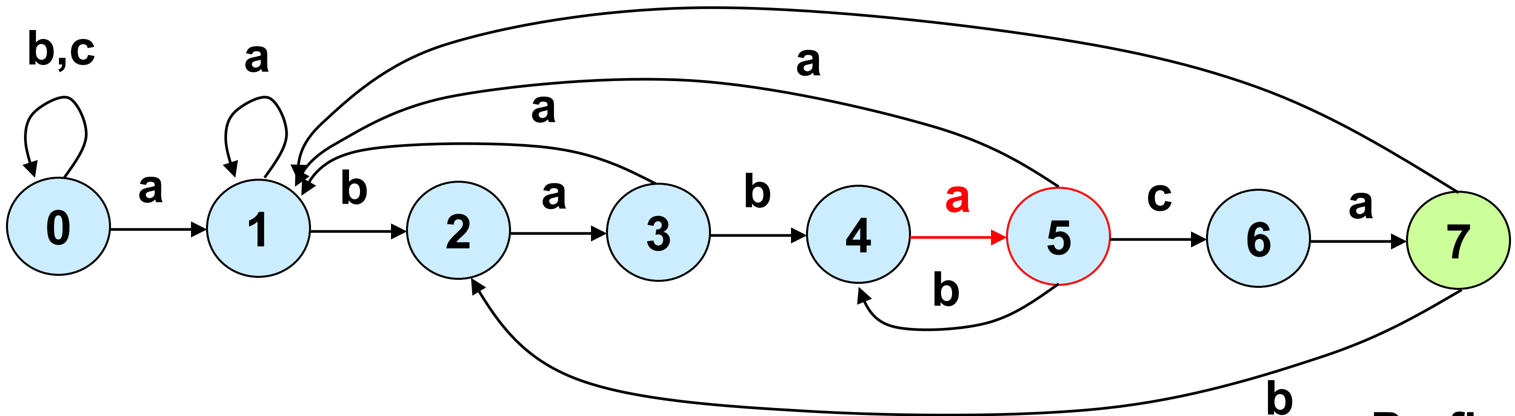


T= ababab̲acaba

Prefixes

a
ab
aba
abab
ababa
ababac
ababaca

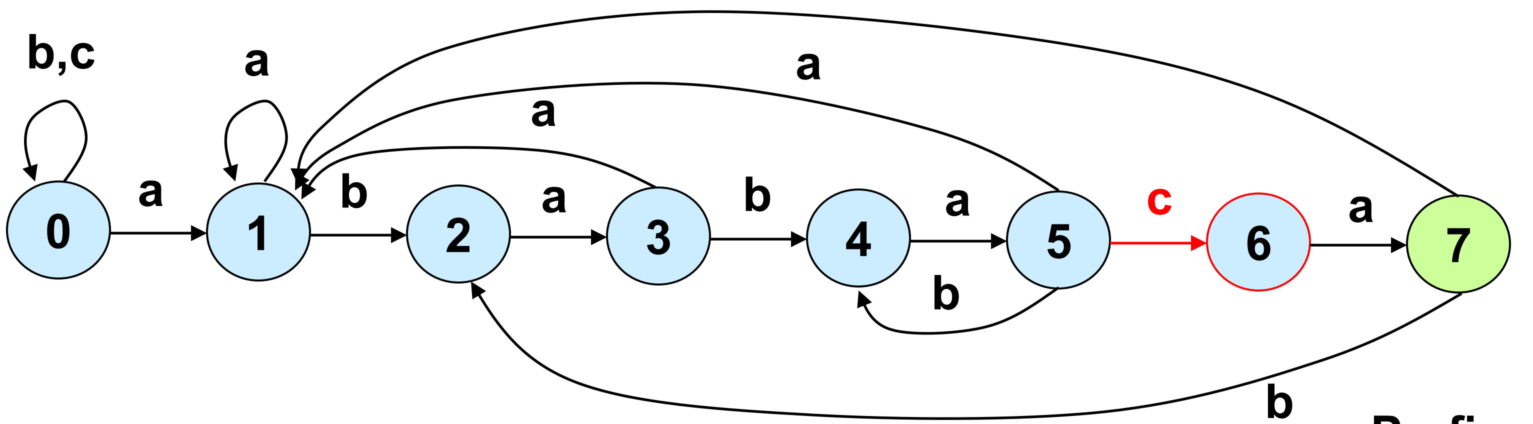
Search



T= ababab**a**caba

<u>Prefixes</u>
a
ab
aba
abab
ababa
ababac
ababaca

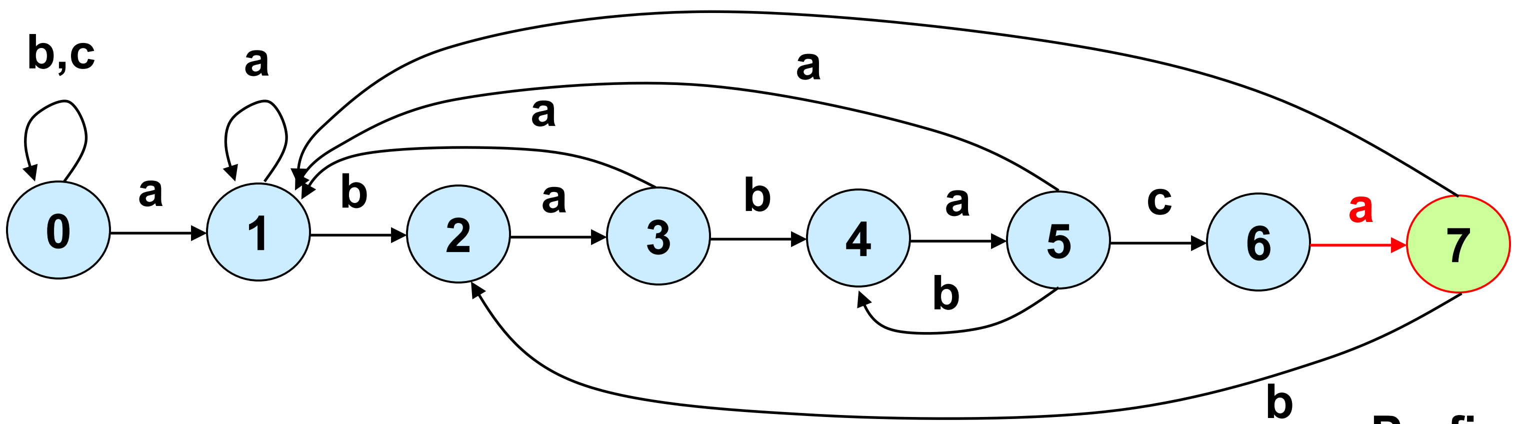
Search



T= abababacaba

<u>Prefixes</u>
a
ab
aba
abab
ababa
ababac
ababaca

Search



Prefixes
a
ab
aba
abab
ababa
ababac
ababaca

T= abababacaba

Accept state, we are done

Analysis of FA

- Searching: $O(n)$ → good
- Preprocessing: $O(m|\Sigma|)$ → bad
- Memory: $O(m|\Sigma|)$ → bad

COMBINATORIAL PATTERN MATCHING

Genomic Repeats

- Example of repeats:
  - AT<u>GGTC</u>TA<u>GGTC</u>CTAGT<u>GGTC</u>
- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats

Genomic Repeats

- The problem is often more difficult:
  - AT<u>GGTC</u>TA<u>GGAC</u>CTAGT<u>GTTC</u>
- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats

ℓ-mer Repeats

- Long repeats are difficult to find
- Short repeats are easy to find (e.g., hashing)

- Simple approach to finding long repeats:
  - Find exact repeats of short ℓ-mers (ℓ is usually 10 to 13)
  - Use ℓ-mer repeats to potentially extend into longer, *maximal* repeats

ℓ-mer Repeats (cont'd)

- There are typically many locations where an ℓ-mer is repeated:

GCTTACAGATTCAGTCTTACAGATGGT

- The 4-mer TTAC starts at locations 3 and 17

Extending l-mer Repeats

GCTTACAGATTCAGTCTTACAGATGGT

- Extend these 4-mer matches:

GC<u>TTACAGAT</u>TCAGTC<u>TTACAGAT</u>GGT

- Maximal repeat: TTACAGAT

Maximal Repeats

- To find maximal repeats in this way, we need ALL start locations of all ℓ-mers in the genome
- **Hashing** lets us find repeats quickly in this manner

Hashing DNA sequences

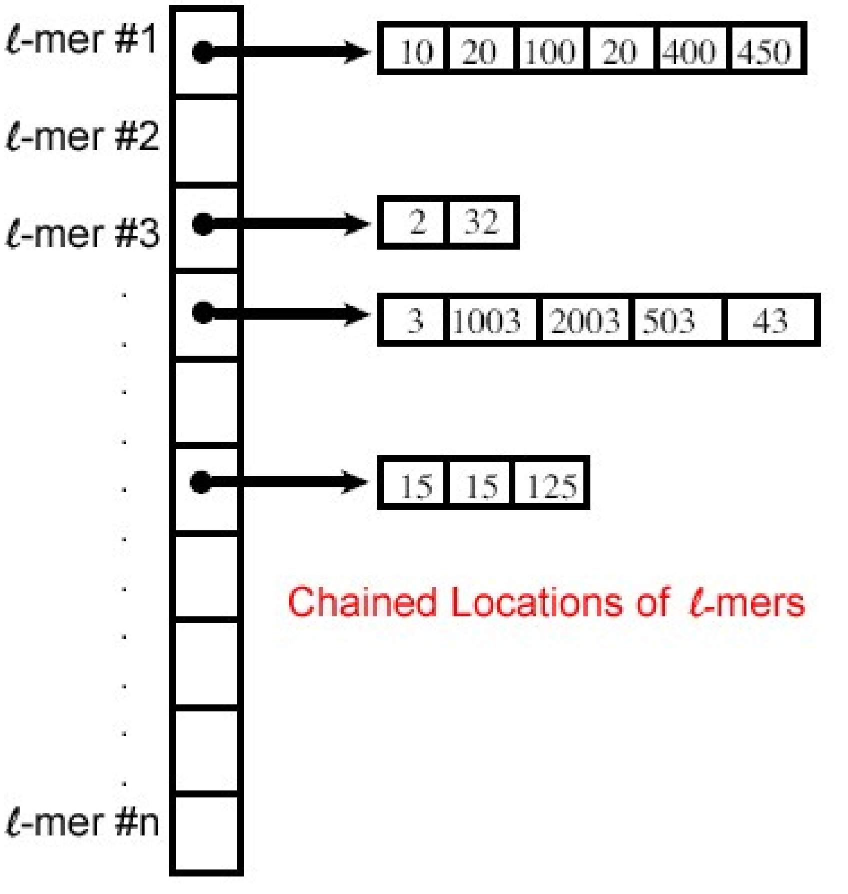
- Each ℓ-mer can be translated into a binary string (**A**, **T**, **C**, **G** can be represented as **00**, **01**, **10**, **11)**
- After assigning a unique integer per ℓ-mer it is easy to get all start locations of each ℓ-mer in a genome

Hashing: Maximal Repeats

- To find repeats in a genome:
  - For all ℓ-mers in the genome, note the start position and the sequence
  - Generate a hash table index for each unique ℓ-mer sequence
  - In each index of the hash table, store all genome start locations of the ℓ-mer which generated that index
  - Extend ℓ-mer repeats to maximal repeats

Hashing: Collisions

- Dealing with collisions:
  - "Chain" all start locations of ℓ-mers (linked list)



Hashing: Summary

- When finding genomic repeats from ℓ-mers:
  - Generate a hash table index for each ℓ-mer sequence
  - In each index, store all genome start locations of the ℓ-mer which generated that index
  - Extend ℓ-mer repeats to maximal repeats

Pattern Matching

- What if, instead of finding repeats in a genome, we want to find all sequences in a database that contain a given pattern?
- This leads us to a different problem, the ***Pattern Matching Problem***

Pattern Matching Problem

- Goal: *Find all occurrences of a pattern in a text*
- Input: Pattern $\boldsymbol{p} = p_1 \ldots p_n$ and text $\boldsymbol{t} = t_1 \ldots t_m$
- Output: All positions $1 \leq i \leq (m - n + 1)$ such that the n-letter substring of $\boldsymbol{t}$ starting at i matches $\boldsymbol{p}$
- **Motivation**: Searching database for a known pattern

Exact Pattern Matching: A Brute-Force Algorithm

PatternMatching(**p**,**t**)

$m \leftarrow$ length of pattern **p**

$n \leftarrow$ length of text **t**

for $i \leftarrow 1$ to $(n - m + 1)$

    if $t_i \ldots t_{i+m-1} = \mathbf{p}$

        output i

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
  - Pattern GCAT
  - Text CGCATC

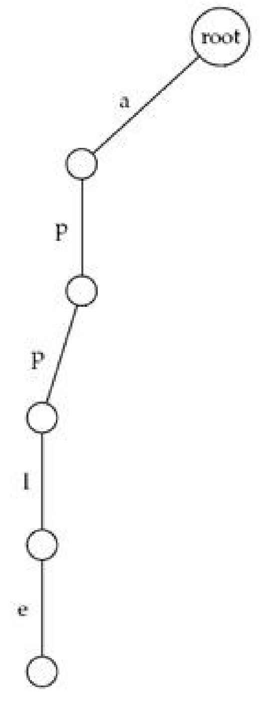
```
GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC
```

Exact Pattern Matching: Running Time

- *PatternMatching* runtime: O(*nm*)
  - KMP or BM: O(n+m)
- Multiply by *k* if looking for *k* different patterns
- Better solution: suffix trees
  - Can solve problem in O(*n*) time
  - Conceptually related to keyword trees

Keyword Trees: Example

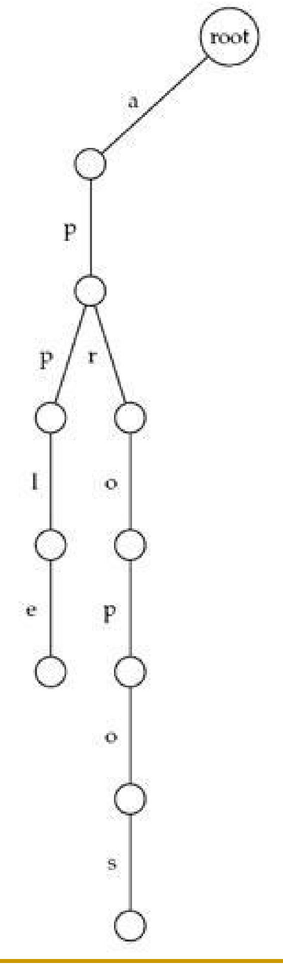
- ***Keyword tree***:
  - Apple



Also known as "trie"

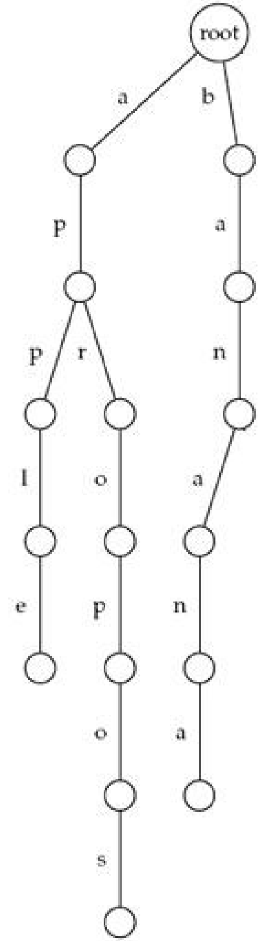
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos



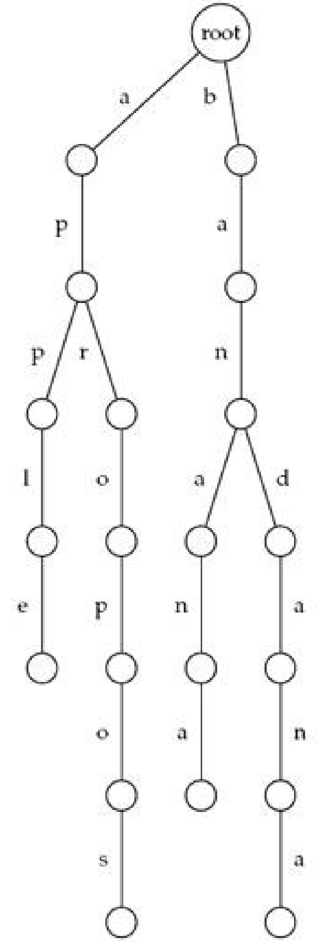
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana



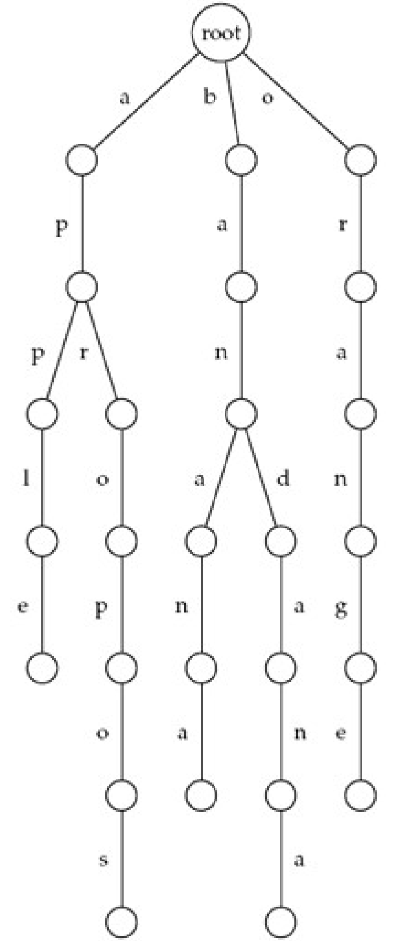
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana



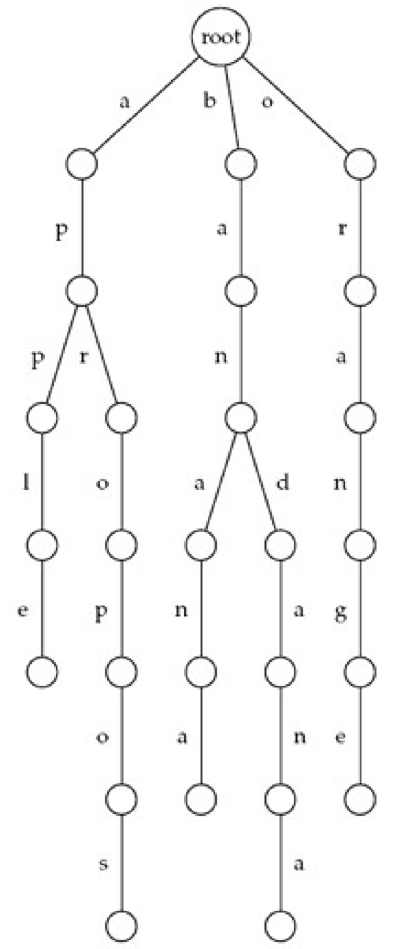
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana
  - Orange



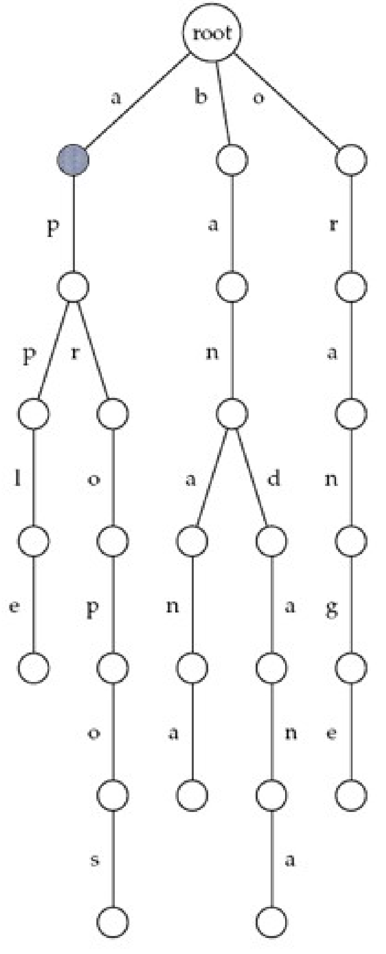
Keyword Trees: Properties

- Stores a set of keywords in a rooted labeled tree
- Each edge labeled with a letter from an alphabet
- Any two edges coming out of the same vertex have distinct labels
- Every keyword stored can be spelled on a path from root to some leaf



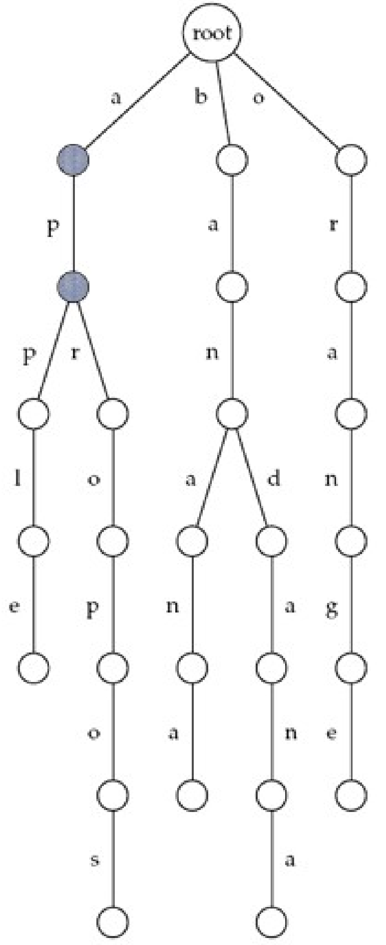
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - <u>a</u>ppeal



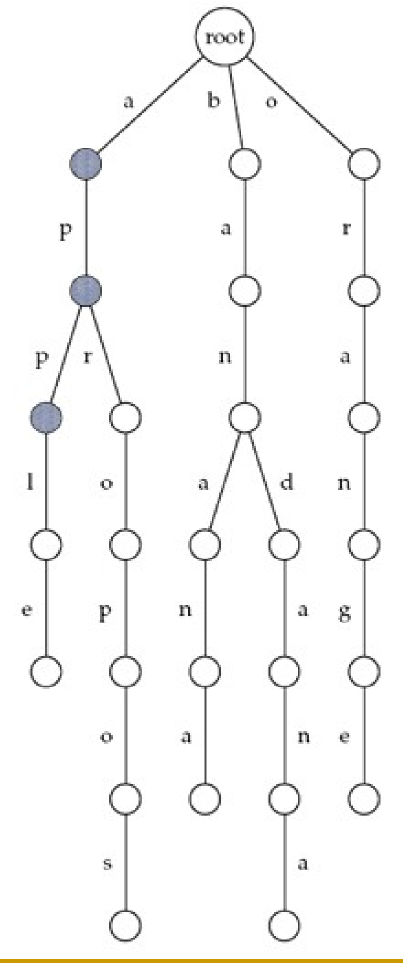
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - <u>ap</u>peal



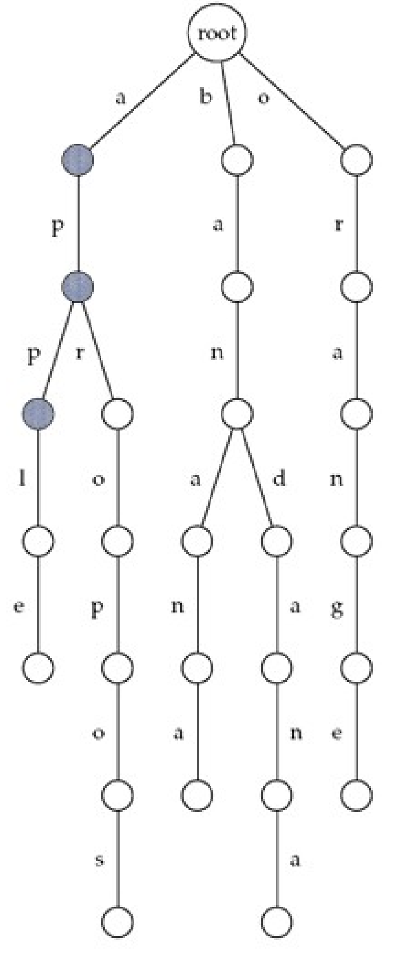
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - appeal



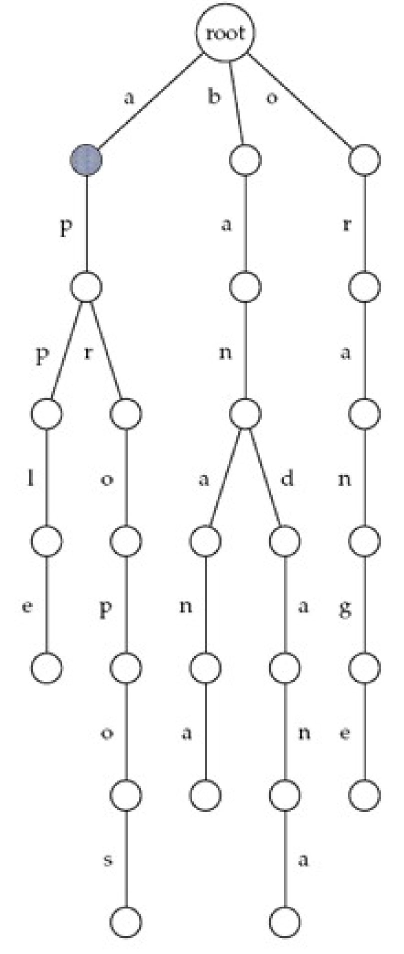
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - appeal



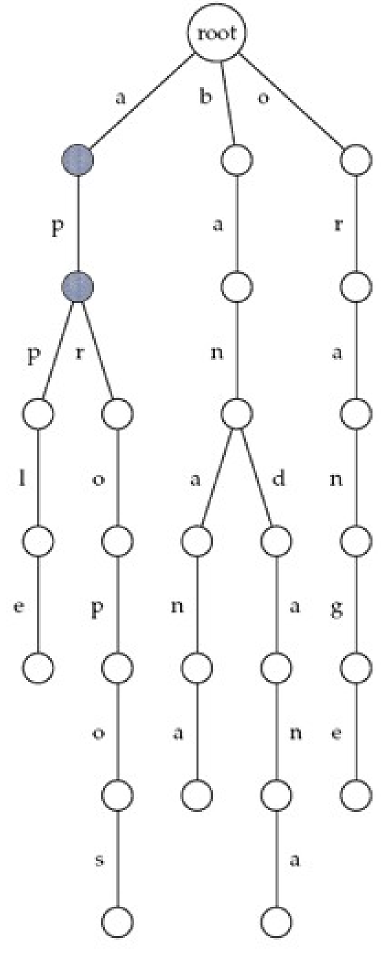
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - apple



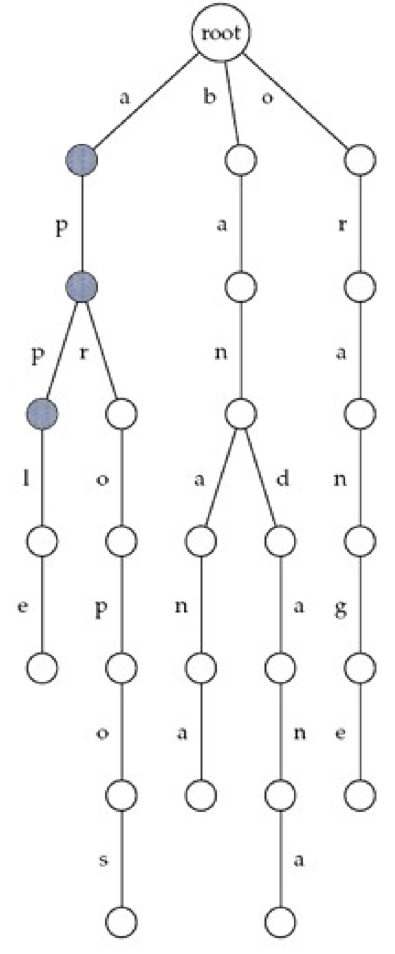
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - <u>app</u>le



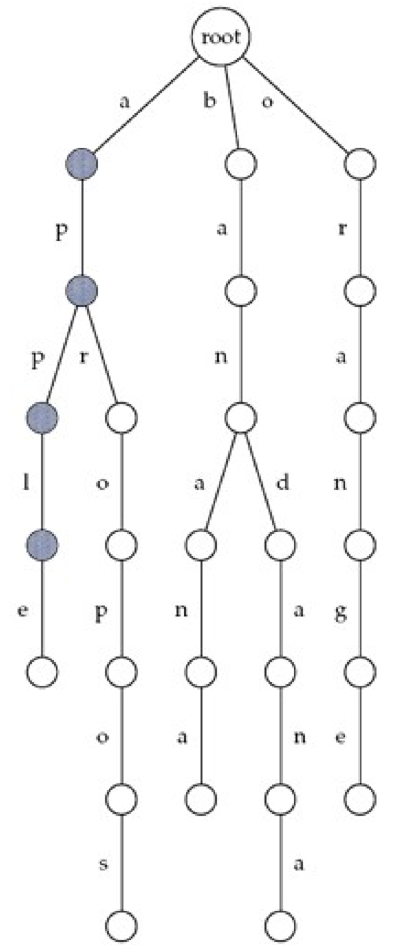
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - apple



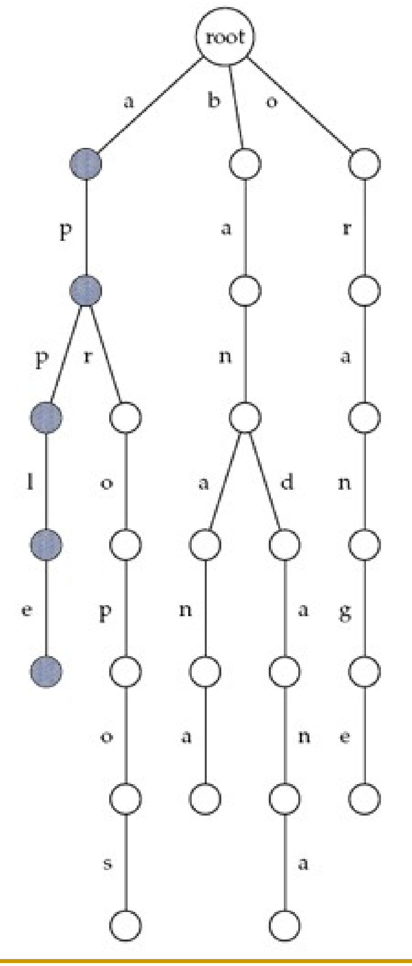
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - apple



Keyword Trees: Threading (cont'd)

- Thread "apple"
  - apple



Multiple Pattern Matching Problem

- Goal: *Given a set of patterns and a text, find all occurrences of any of patterns in text*
- Input: k patterns $\mathbf{p}^1,\ldots,\mathbf{p}^k$, and text $\mathbf{t} = t_1\ldots t_m$
- Output: Positions $1 \leq i \leq m$ where substring of $\mathbf{t}$ starting at i matches $\mathbf{p}_j$ for $1 \leq j \leq k$
- **Motivation**: Searching database for known multiple patterns

Multiple Pattern Matching: Straightforward Approach

- Can solve as *k* "Pattern Matching Problems"
  - Runtime:

    $$O(kmn)$$

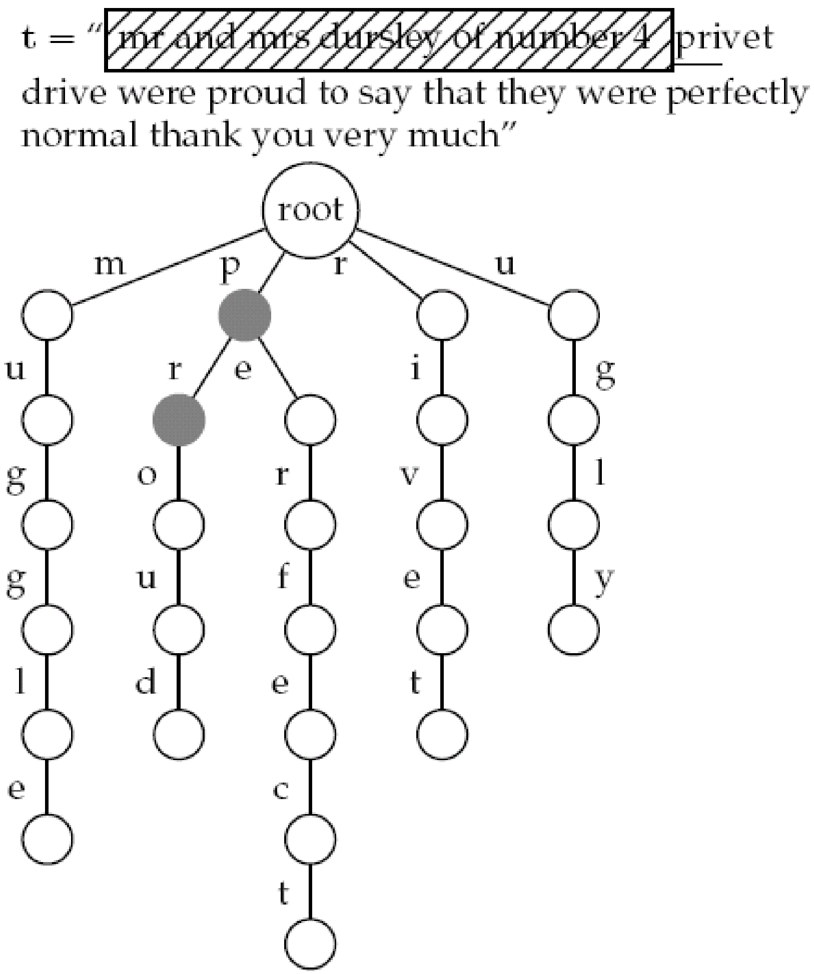
    using the *PatternMatching* algorithm *k* times
  - *m* - length of the text
  - *n* - average length of the pattern

Multiple Pattern Matching: Keyword Tree Approach

- Or, we could use keyword trees:
  - Build keyword tree in $O(N)$ time; N is total length of all patterns
  - With naive threading: $O(N + nm)$
  - Aho-Corasick algorithm: $O(N + m)$

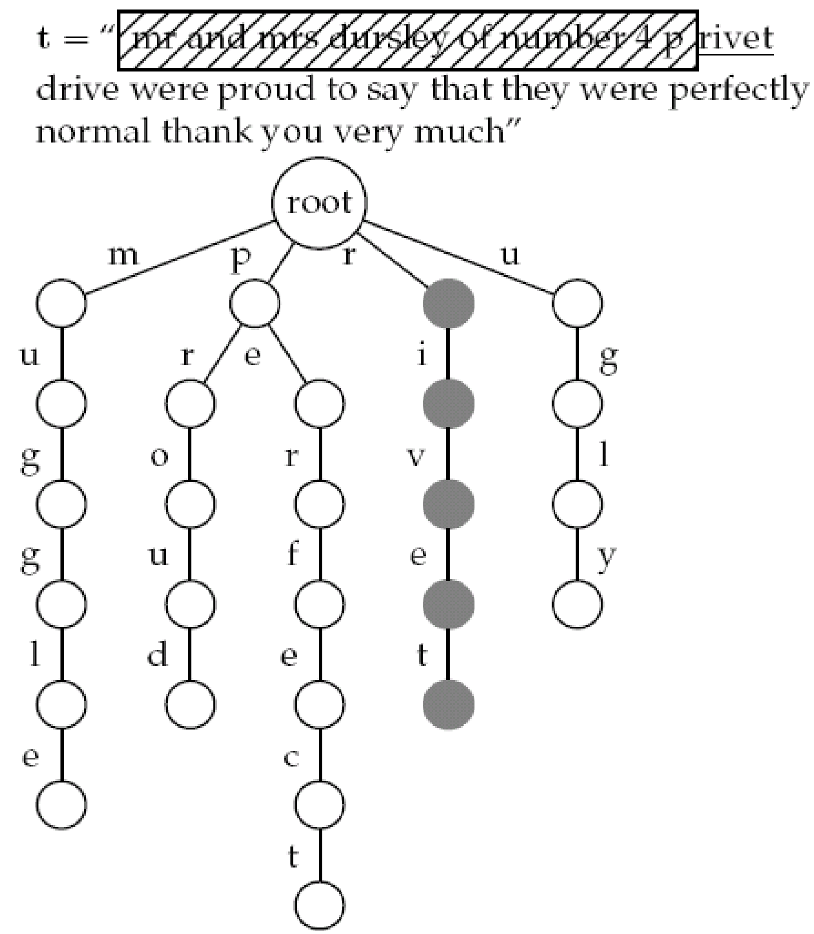
Keyword Trees: Threading

- To match patterns in a text using a keyword tree:
  - Build keyword tree of patterns
  - "Thread" the text through the keyword tree



Keyword Trees: Threading (cont'd)

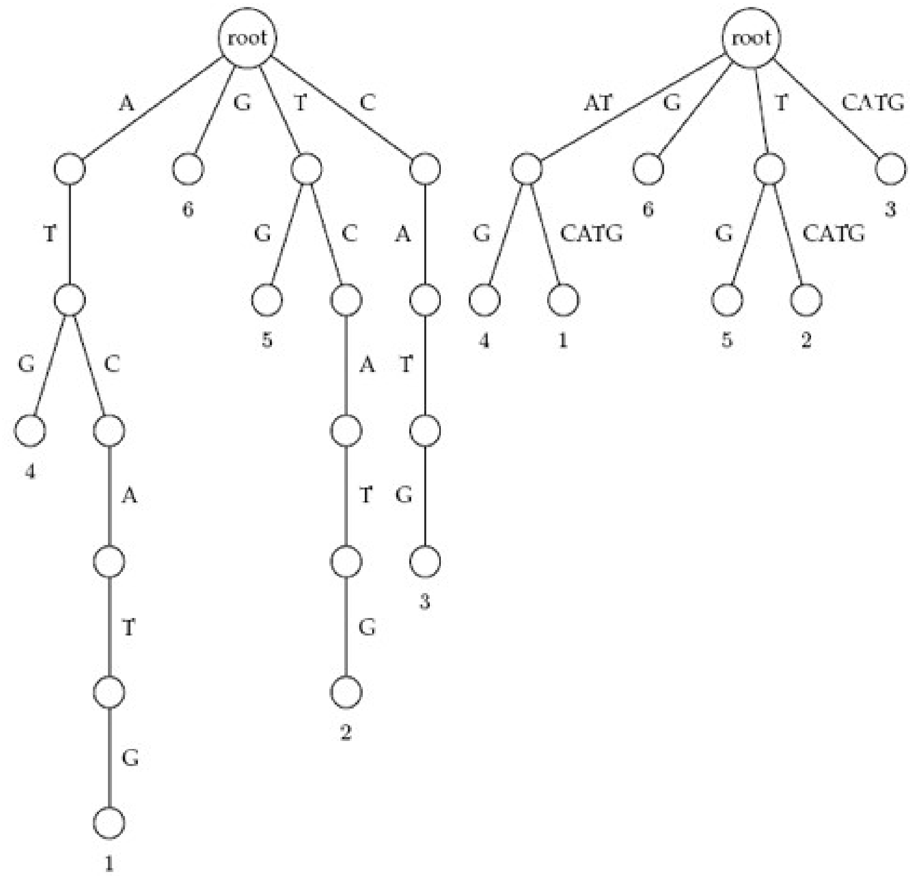
- Threading is "complete" when we reach a leaf in the keyword tree
- When threading is "complete," we've found a pattern in the text



Problem: High memory requirement when N is large

Suffix Trees=Collapsed Keyword Trees

- Similar to keyword trees, except edges that form paths are collapsed
  - Built from **text**, not patterns
  - Each edge is labeled with a *substring* of a text
  - All internal edges have at least two outgoing edges
  - Leaves labeled by the index of the pattern.



(a) Keyword tree

(b) Suffix tree

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

→ Keyword Tree → Suffix Tree

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

→ **Keyword Tree** → **Suffix Tree**

How much time does it take?

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

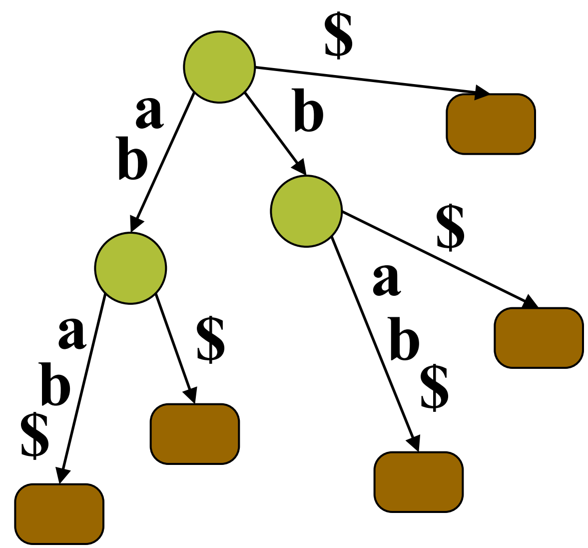
$\xrightarrow{\text{quadratic}}$ **Keyword Tree** $\longrightarrow$ **Suffix Tree**

Time is linear in the total size of all suffixes, i.e., it is quadratic in the length of the text

Suffix tree (Example)

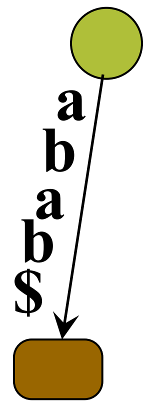
Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

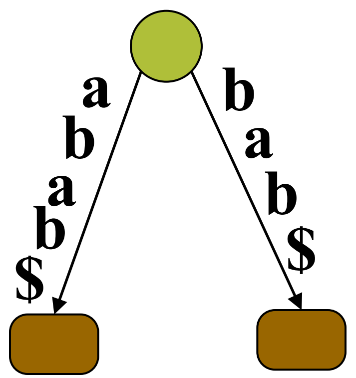


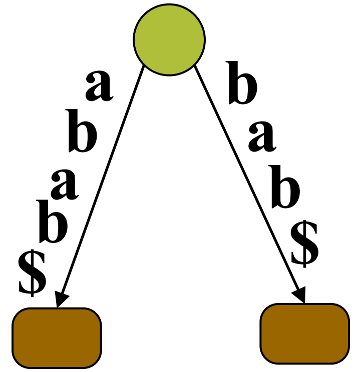
Trivial algorithm to build a Suffix tree

Put the largest suffix in

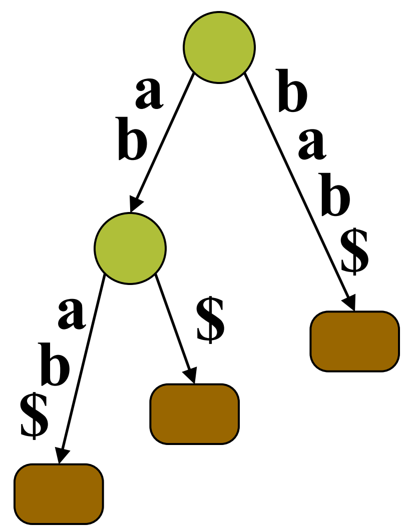


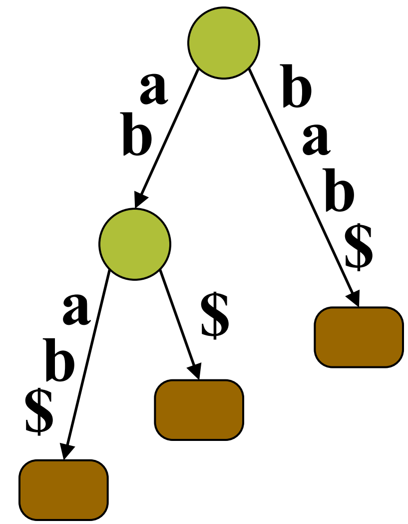
Put the suffix bab$ in



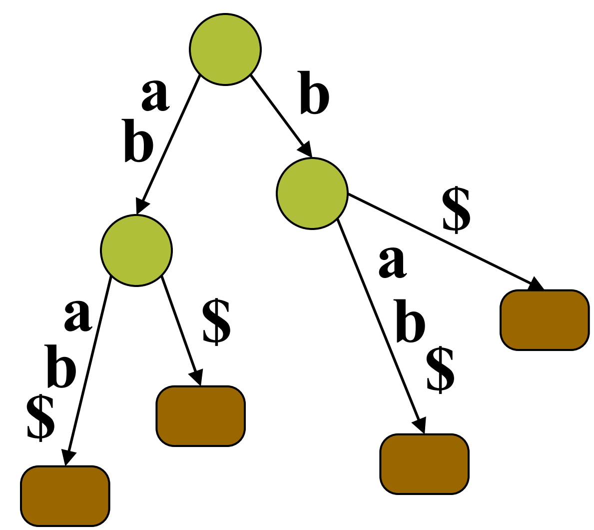


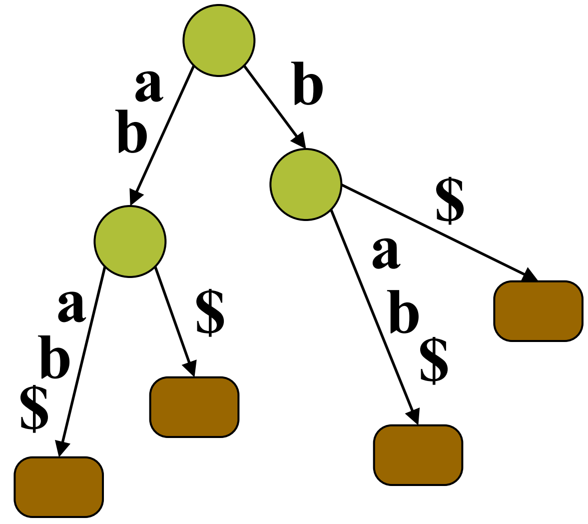
Put the suffix ab$ in



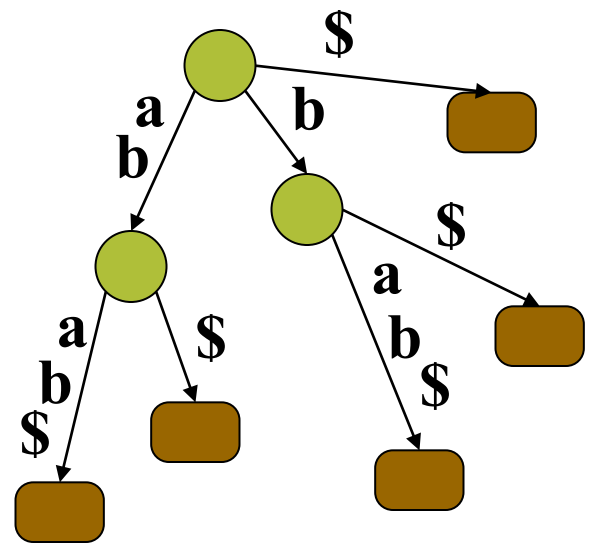


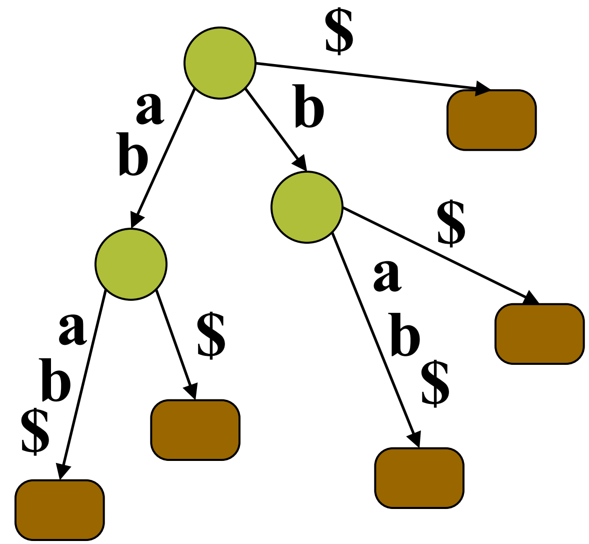
Put the suffix b$ in



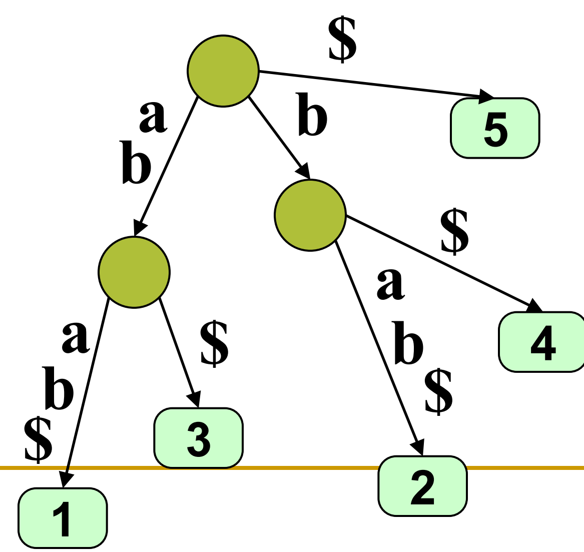


Put the suffix $ in





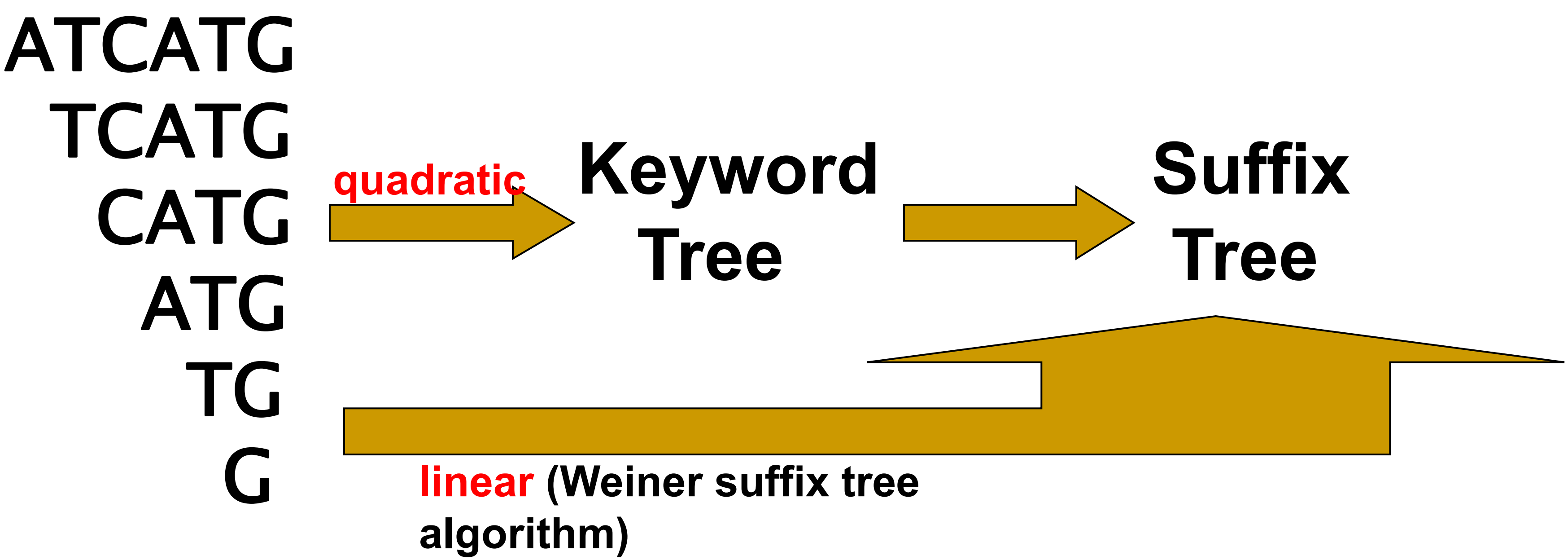
We will also label each leaf with the starting point of the corres. suffix.



Trivial algorithm: O(n²) time

Suffix Trees: Advantages

- Suffix trees of a text is constructed for all its suffixes
- Suffix trees build faster than keyword trees



Use of Suffix Trees

- Suffix trees hold all suffixes of a text
  - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
  - Builds in O(*m*) time for text of length *m*
- To find any pattern of length ***n*** in a text:
  - Build suffix tree for text
  - Thread the pattern through the suffix tree
  - Can find pattern in text in O(*n*) time!
- O(*n* + *m*) time for “Pattern Matching Problem”
  - Build suffix tree and lookup pattern

Pattern Matching with Suffix Trees

<u>SuffixTreePatternMatching(**p**,**t**)</u>

Build **suffix tree** for text **t**

Thread pattern **p** through **suffix tree**

if threading is complete

  output positions of all **p**-matching leaves in the tree

else

  output "Pattern does not appear in text"

Suffix Trees: Example

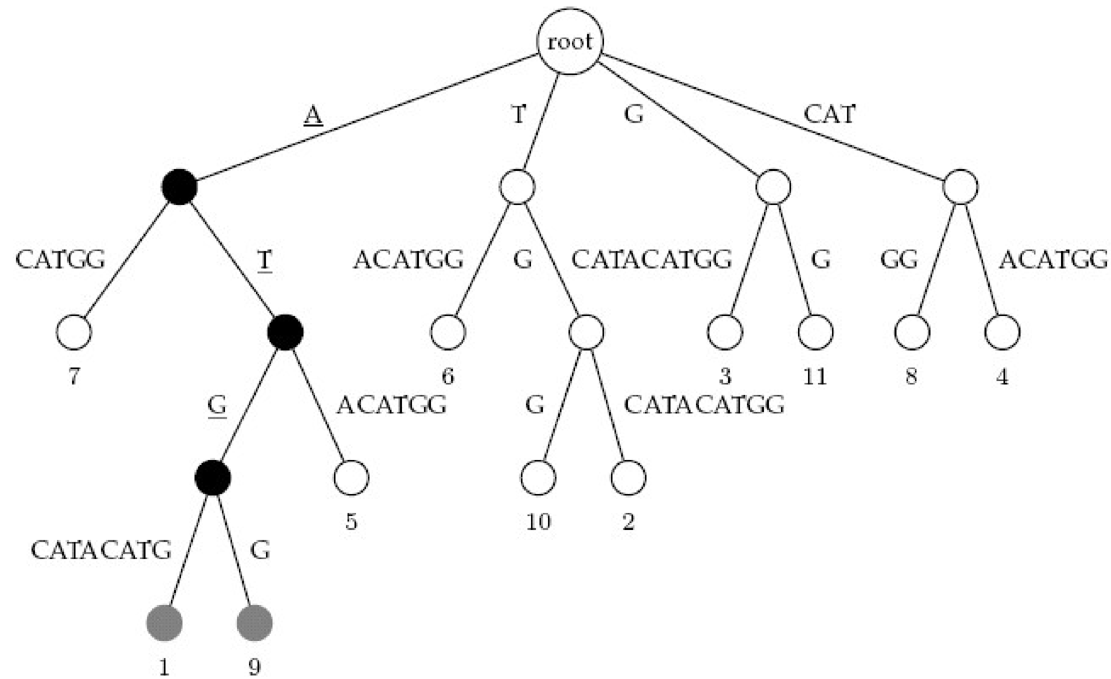


Figure 9.6 Threading the pattern ATG through the suffix tree for the text ATGCATA-CATGG. The suffixes ATGCATACATGG and ATGG both match, as noted by the gray vertices in the tree (the **p**-matching leaves). Each *p*-matching leaf corresponds to a position in the text where **p** occurs.

Generalized suffix tree

Given a set of strings S a generalized suffix tree of S is a compressed trie of all suffixes of $s \in S$

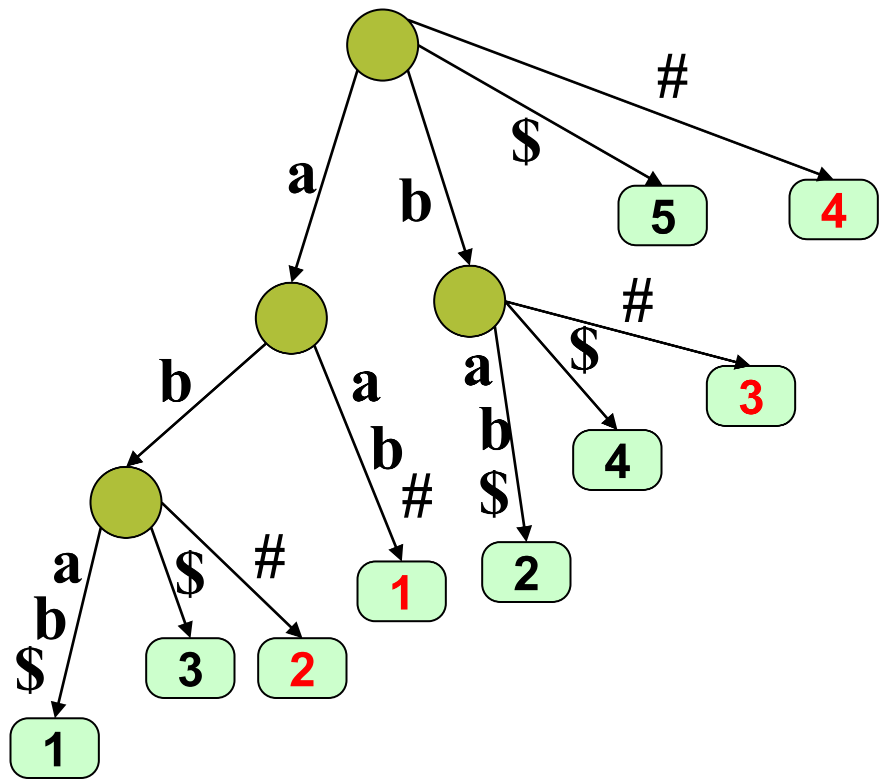
To make these suffixes prefix-free we add a special char, say $, at the end of s

To associate each suffix with a unique string in S add a different special char to each s

Generalized suffix tree (Example)

Let s_1=abab and s_2=aab here is a generalized suffix tree for s_1 and s_2

{
$ #
b$ b#
ab$ ab#
bab$ aab#
abab$
}

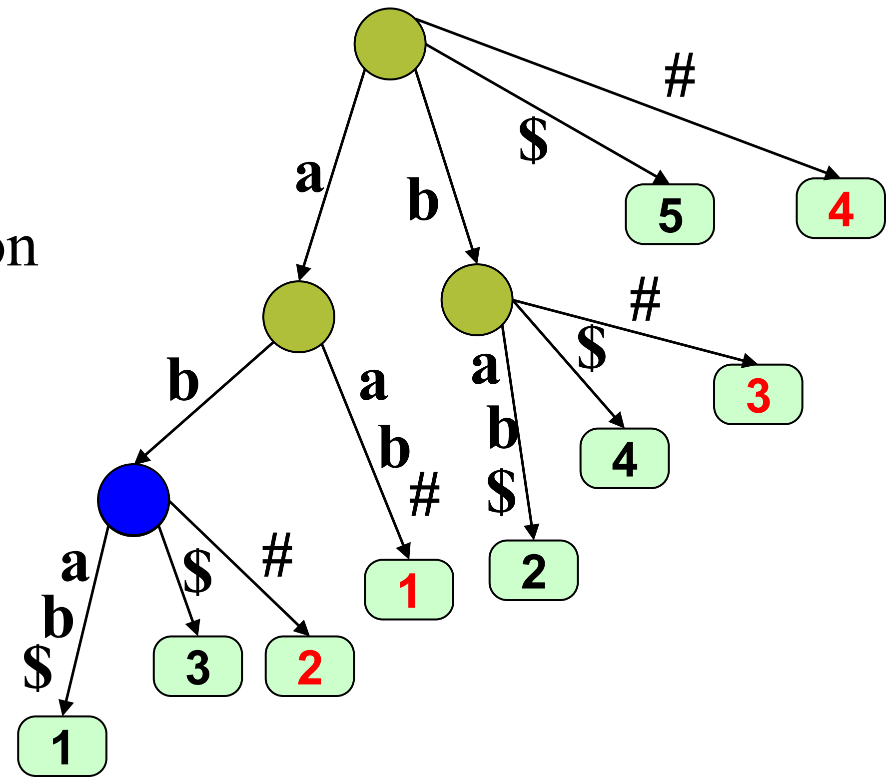


Matching a pattern against a database of strings

Longest common substring of two strings

Every node with a leaf descendant from string S_1 and a leaf descendant from string S_2 represents a maximal common substring and vice versa.

Find such node with largest “string depth”



Multiple Pattern Matching: Summary

- Keyword and suffix trees are used to find patterns in a text
- ***Keyword trees***:
  - Build keyword tree of patterns, and ***thread text*** through it
- ***Suffix trees***:
  - Build suffix tree of text, and ***thread patterns*** through it