

COMBINATORIAL PATTERN MATCHING

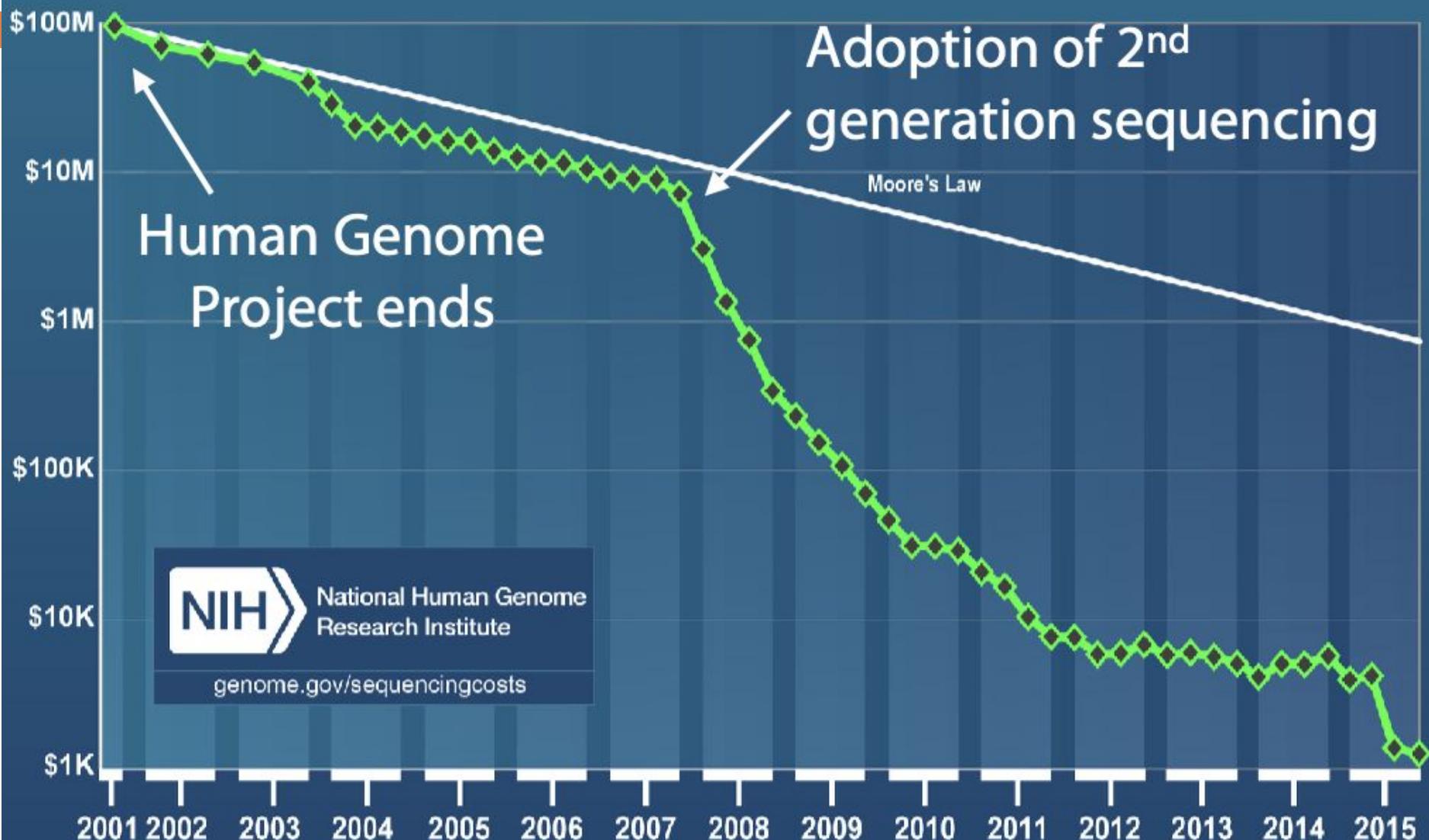
CSE 463: INTRODUCTION TO BIOINFORMATICS

Dr M A Nayeem

Outline

- Hash Tables
- Exact Pattern Matching
- Keyword Trees
- Suffix Trees
- Suffix Array
- Burrows-Wheeler Transform (BWT)
- FM Index

Cost per Genome



Why Personal Genomics

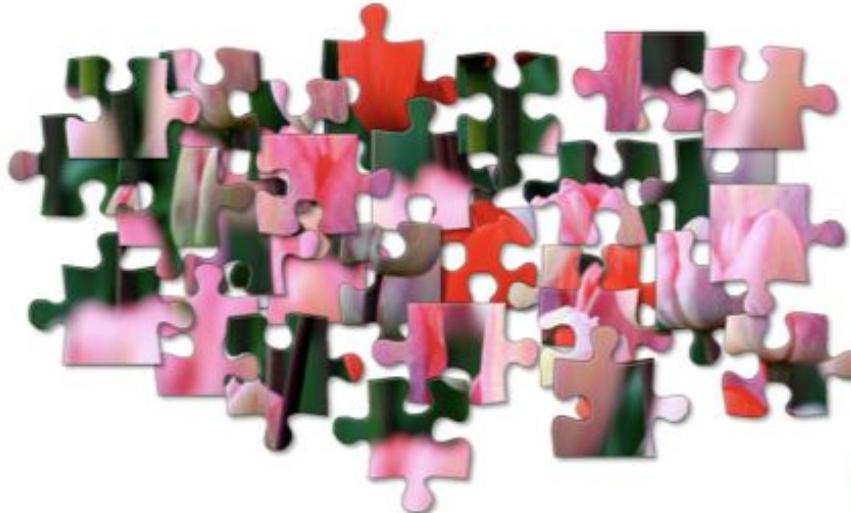
- Nicholas Volker: first person to have life saved by genome sequencing (2010).
- Personal genomics can help us understand the genetic basis of diseases.



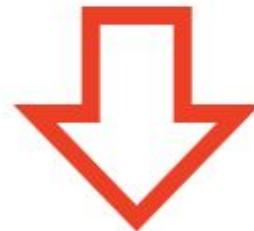
Short read mapping

- Short read
 - Second generation or mainly Illumina reads
 - 35-300bp
- Mapping
 - Finding the positions in the genome reads are coming from
- Applications
 - Reference guided genome assembly
 - Looking for SNPs
 - ...

Reference guided assembly



+



ATATCTTAGAGGGAGGGCTGAGGGTTGAAGTCCAACCTCTAACGCCAGTGCCAGAAGAGCCAAGGACAGGTACGGCTGTC
ATCACTTAGACCTCACCTGTGGAGCCACACCCTAGGGTTGGCCAATCTACTCCCAGGAGCAGGGAGGGCAGGAGCCAGG
GCTGGGCATAAAAGTCAGGGCAGAGCCATCTATTGCTTACATTGCTTCTGACACAACGTGTTCACTAGCAACCTCAA
CAGACACCATGGTGCATCTGACTCCTGAGGAGAAGTCTGCCGTTACTGCCCTGTGGGCAAGGTGAACGTGGATGAAGTT
GGTGGTGAGGCCCTGGGAGGTTGGTATCAAGGTTACAAGACAGGTTAAGGAGACCAATAGAAAACGGCATGTGGAGA
CAGAGAAGACTCTTGGGTTCTGATAGGCACTGACTCTCTGCCTATTGGTCTATTCCCACCCCTAGGCTGCTGGTG
GTCTACCCCTGGACCCAGAGGTTCTTGAGTCCTTGGGATCTGCCACTCCTGATGCTTATGGCAACCTAAGGT
GAAGGCTCATGGCAAGAAA **G**TGCTCGGTGCCCTTAGTGTAGTGGCCTGGCTCACCTGGACAACCTCAAGGGCACCTTGC
CACTGAGTGAGCTGCACTGTGACAAGCTGCACGTGGATCCTGAGAACCTCAGGGTGAGTCTATGGGACGCTTGATGTTT
CTTCCCCCTTTCTATGGTTAAGTTATGTGATAGGAAGGGGATAAGTAACAGGGTACAGTTAGAATGGGAAACAG
ACGAATGATTGCATCAGTGTGGAAAGTCTCAGGATCGTTAGTTCTTATTGCTGTTCATAACAAATTGTTTCTT
GTTAATTCTGCTTCTTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTT
CCGCAATTAAACTATTACTATTACTTAATGCCTAACATTGTGTATAACAAA
AGGAAATATCTCTGAGATACTTAAGTAACCTAAAAAAACTTACACAGTCTGCCTAGTACATTACTATTGGAATAT
ATGTGTGCTTATTGCATATTGATCATATTCTAACATCTCCCTACTTATTCTTATTCTTATTGATACATAATTACATAT
TTATGGGTTAAAGTGTAAATTGTTAATATGTGTACACATATTGACCAAATCAGGGTAATTGCTTGTAAATTGTTAATTT
AATGCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTT
TGATACAATGTATCATGCCTCTTGCACCATTCTAAAGAATAACAGTGTAAATTCTGGGTTAAGGCAATAGCAATATCT
CTGCATATAAAATTCTGCATATAAAATTGTAACGTGTAAAGAGGTTCATATTGCTAATAGCAGCTACAATCCAGCTA
CCATTCTGCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTTCTT
TACCTCTTATCTCCTCCCACAGCTCTGGCAACGTGCTGGTCTGTGTGGCCCATCACTTGGCAAAGAATTCA
CCACCAAGTGCAGGCTGCCTATCAGAAAGTGGTGGCTGGCTAATGCCCTGGCCCACAAGTATCACTAAGCTCGCTT
TCTTGTGCTTCAATTCTATTAAAGGTTCTTGTGCTTCTTAAGTCCAACACTAAACTGGGGGATATTATGAAGGGCCTT
GAGCATCTGGATTCTGCCTAATAAAAAACAT **T**TATTTCATTGCAATGATGTATTAAATTATTCTGAATATTCT
AAAAGGGAATGTGGGAGGTCACTGCATTAAAACATAAAGAAATGAAGAGCTAGTTCAAACCTTGGAAAACACTATA

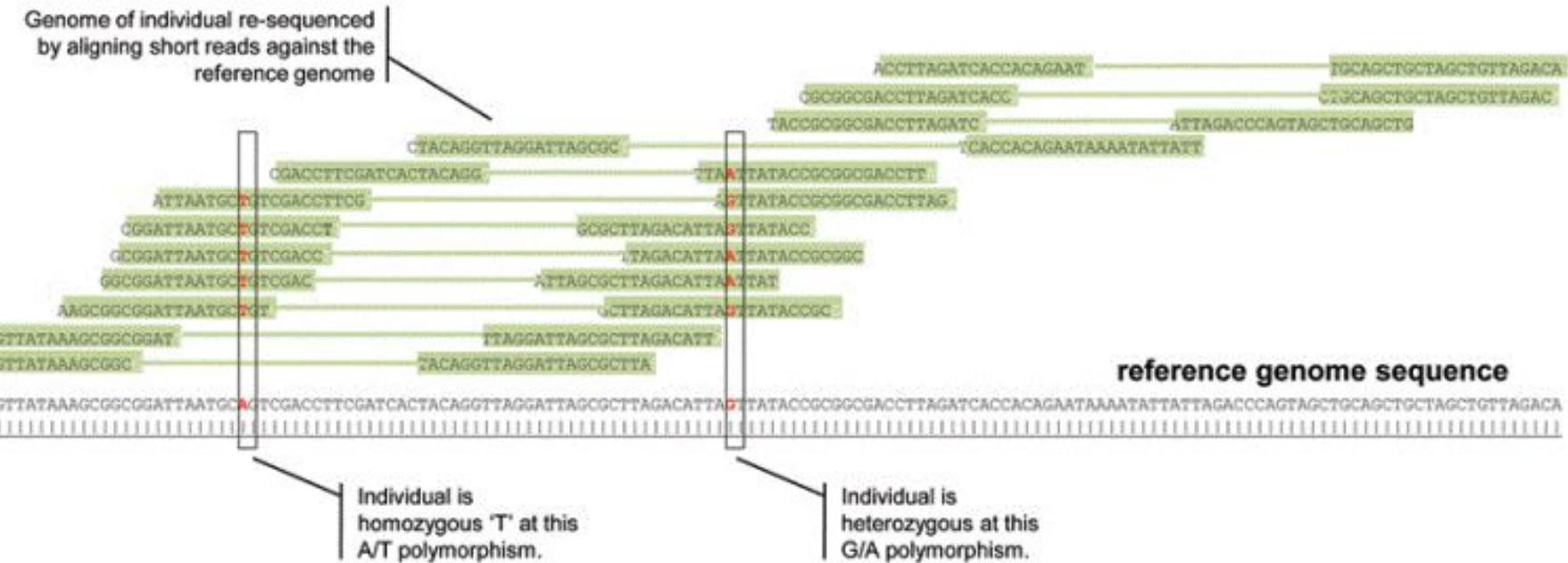
Unrelated humans have genomes that are 99.8-99.9% similar

Reference guided assembly

TATGTCGAGTATCTG
TATGTCGAGTATCTT
TATGTCGAGTATCTG
TATGTCGAGTATCTG
CCGGACACCCATA
ACACCCATGTCGA
ACACCCATGTCGA
TATGTCGAGTATCTG
ACACCCATGTCGA
CCGGACACCCATA
GTCGAGTATCTGNN
TGTCGAGTATCTGTC

GATCACAGGTCTATCACCTATTACCCTCACGGGAGCTCTCCATGCATTGGTATTT
CGTCTGGGGGTATGCACCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCATGTC
GCACTATCTGCTTGATTCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACACTAAAGTGTGTTAATTAAATTAAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCGACAGCCACTTCCACACAGACATCATAACAAAAAATTCCACCA
AACCCCCCTCCCCGCTTCTGCCACAGCACTAAACACATCTCTGCCAAACCCCCAAA
ACAAAGAACCTAACACCGCTAACAGATTCAAATTATCTTTGGCGGTATGCAC
TTTAACAGTCACCCCCCAACTAACACATTATTTCCCCTCCACTCCCATACTACTAAT
CTCATCAATACAACCCCCGCCATCCTACCCAGCACACACACACCGCTGCTAACCCCATA
CCCCGAACCAACCAACCCCCAAAGACACCCCCACAGTTATGTAGCTTACCTCCTCAA
GCAATACACTGACCCGCTCAAACCTCTGGATTGGATCCACCCAGCGCCTTGGCTAA
CTAGCTTCTATTAGCTCTTAGTAAGATTACACATGCAAGCATCCCCGTTCCAGTGAGT
TCACCCCTAAATCACCACGATCAAAGGAACAAGCATCAAGCACGCAAGCAATGCACTC
AAACGCTTAGCCTAGCCACACCCCCACGGGAAACAGCAGTGATTAACCTTAGCAATAA
ACGAAAGTTAACTAAGCTATACTAACCCCAGGGTTGGTCAATTCTGCTGCCAGCCACCGC
GGTCACACGATTAACCCAAAGTCATAAGAAGCCGGCGTAAAGAGTGTTTAGATACCCCC
TCCCCATAAAAGCTAAACCTCACCTGAGTTGAAAAAACTCCAGTTGACACAAAAATAGAC
TACGAAAGTGGCTTAACTATCTGAACACACAAATAGCTAAGACCCAAACTGGGATTAGA
TACCCCCACTATGCTTAGCCCTAAACCTCAACAGTTAAATCAACAAAATGCTCGCCAGAA
CACTACGAGCCACAGCTTAAACCTCAAGGACCTGGCGGTGCTTCAATCCCTAGAGG
AGCCCTGTTCTGTTAGCTAAACCCGATCAACCCCTACCCACCTCTGCTCAACCTATAA

Reference guided assembly



Exact pattern matching

- Given a pattern (read) and a text (genome), find all positions in the text that matches the pattern?
- This leads us to the **Pattern Matching Problem**.

Pattern Matching Problem

- **Goal:** Find all occurrences of a pattern in a text.
- **Input:**
 - Pattern $p = p_1 \dots p_m$ of length m
 - Text $t = t_1 \dots t_n$ of length n
- **Output:** All positions $1 \leq i \leq (n - m + 1)$ such that the m -letter substring of text t starting at i matches the pattern p .

Exact Pattern Matching: Brute Force

PatternMatching(p,t)

- 1 $m \square$ length of pattern **p**
- 2 $n \square$ length of text **t**
- 3 for $i \square 1$ to $(n - m + 1)$
- 4 if $t_i \dots t_{i+m-1} = p$
- 5 output i

Exact Pattern Matching: An Example

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

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCC**GCATCT**
- **GCAT**

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCCG**C**CATCT
- **G**CAT

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AG**CCGC**ATCT
 - **GCAT**

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCC**CGCA**TCT
 - **GCAT**

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCC**GCAT**CT
- **GCAT**

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCCG**CATCT**
 - **GCAT**

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
 - Pattern GCAT
 - Text AGCCGCATCT
- AGCCGC**A**TCT
 - **GCAT**

Exact Pattern Matching: Running Time

- **PatternMatching runtime:** $O(nm)$
 - In the **worst** case, we have to check m characters at each of the n letters of the text.
 - **Example:** Text = AAAAAAAAAAAAAAAA, Pattern = AAAC
- This is rare; on average, the run time is more like $O(m)$.
 - Rarely will there be close to m comparisons at each step.

Knuth-Morris-Pratt Algorithm

- Linear time algorithm
- Avoids comparisons with positions in the text that we have already checked



- AGCAGCTAGCTAGT
- AGCTAGT



Knuth-Morris-Pratt Algorithm

- Linear time algorithm
- Avoids comparisons with positions in the text that we have already checked



- AGCAGCTAGCTAGT
- AGCTAGT



Knuth-Morris-Pratt Algorithm

Pattern Matched so far

AGCTAGT AGCTAG _

Proper
prefix Proper
suffix

- Compute length of the longest proper prefix of the pattern P that is a proper suffix of $P[1:q]$
- $P[1:q]$ matched with the text at some point, but $p[q+1]$ failed.

Knuth-Morris-Pratt Algorithm

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)

Knuth-Morris-Pratt Algorithm

KMP-MATCHER(T, P)

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$                                 // number of characters matched
5  for  $i = 1$  to  $n$                   // scan the text from left to right
6    while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7       $q = \pi[q]$                       // next character does not match
8      if  $P[q + 1] == T[i]$ 
9         $q = q + 1$                     // next character matches
10       if  $q == m$                       // is all of  $P$  matched?
11         print "Pattern occurs with shift"  $i - m$ 
12          $q = \pi[q]$                   // look for the next match
```

Knuth-Morris-Pratt Algorithm

COMPUTE-PREFIX-FUNCTION(P)

```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6      while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7           $k = \pi[k]$ 
8          if  $P[k + 1] == P[q]$ 
9               $k = k + 1$ 
10          $\pi[q] = k$ 
11 return  $\pi$ 
```

Short read mapping

- KMP runs in $O(n+m)$
 - Preprocesses the pattern
 - To map k -reads $O(k(n+m))$ time is needed
- But typically we need to map many reads to a single genome
- Better to index the genome
 - Hash tables
 - Suffix trees
 - Suffix arrays
 - BWT-FM index

Multiple Pattern Matching (MPM) Problem

- **Goal:** Given a set of patterns and a text, find all occurrences of any of patterns in text.
- **Input:** k patterns p^1, \dots, p^k , and text of n characters $t = t_1 \dots t_n$
- **Output:** Positions $1 \leq i \leq n$ where the substring of text t starting at i matches one of the patterns p_i for $1 \leq i \leq k$.

Hashing

- Hashing is a very efficient way to store and retrieve data.
- What does hashing do?
 - Say we are given a collection of data.
 - For different entries, generate a **unique integer** and store the entry in an array at this integer location

Hashing: Definitions

- **Records:** **data** stored in a *hash table*.
- **Keys:** Integers identifying set of *records*.
- **Hash Table:** The array of keys used in hashing.
- **Hash Function:** Assigns each record to a key.
- **Collision:** Occurs when more than one record is mapped to the same index in the hash table.

Hashing DNA sequences

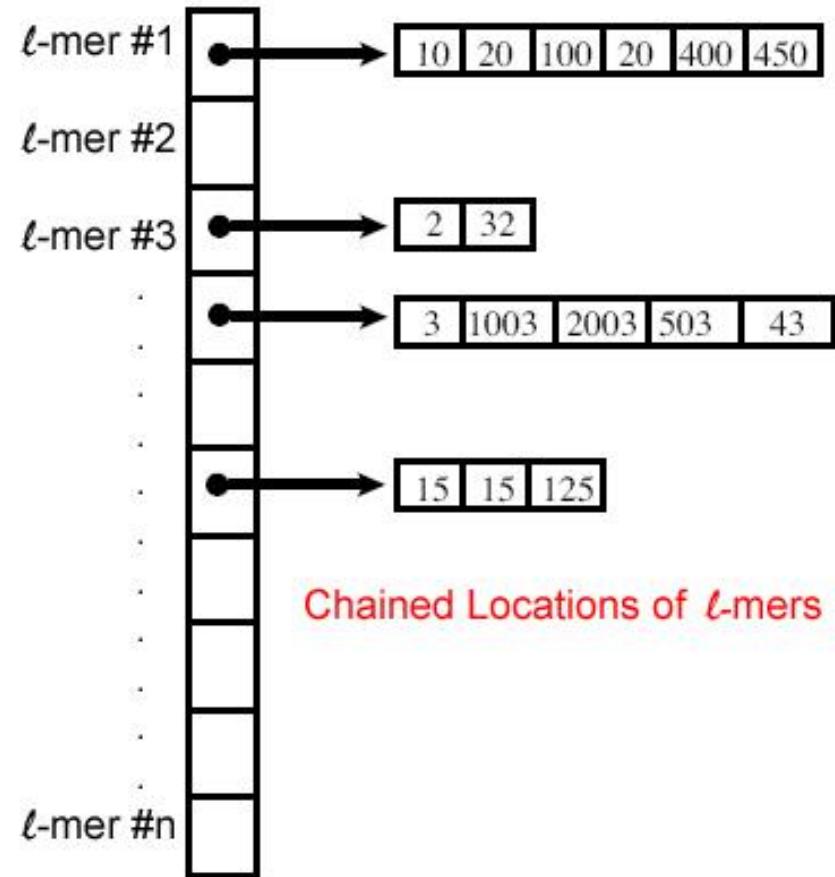
- Each **k-mer** can be translated into a **binary** string:
 - A can be represented as **00**
 - T can be represented as **01**
 - G can be represented as **10**
 - C can be represented as **11**
 - **Example:** ATGCTA = 000110110100
- After assigning a **unique** integer to each **k-mer**, it is easy to obtain all **start locations** of each **k-mer** in a genome.

Hashing: Read Mapping

- To map reads exactly:
 1. For all *k-mers* in the genome, note the **start position** and the sequence where *k* is the read length
 2. Generate a hash table index for each *k-mer* sequence.
 3. At each index of the hash table, store **all** genome start locations of the *k-mer* that generated the index.
 4. Look up all start positions using the read as the key
- How do we deal with collisions?

Hashing: Collisions

- In order to deal with collisions:
- “Chain” all start locations of k -mers.
- This can be done via a data structure called a **linked list**.



Hashing: Read Mapping

A	0
A	1
C	2
G	3
T	4
A	5
C	6
G	7
G	8
A	9
G	1
T	0
T	1
A	2
C	3
G	4
T	5

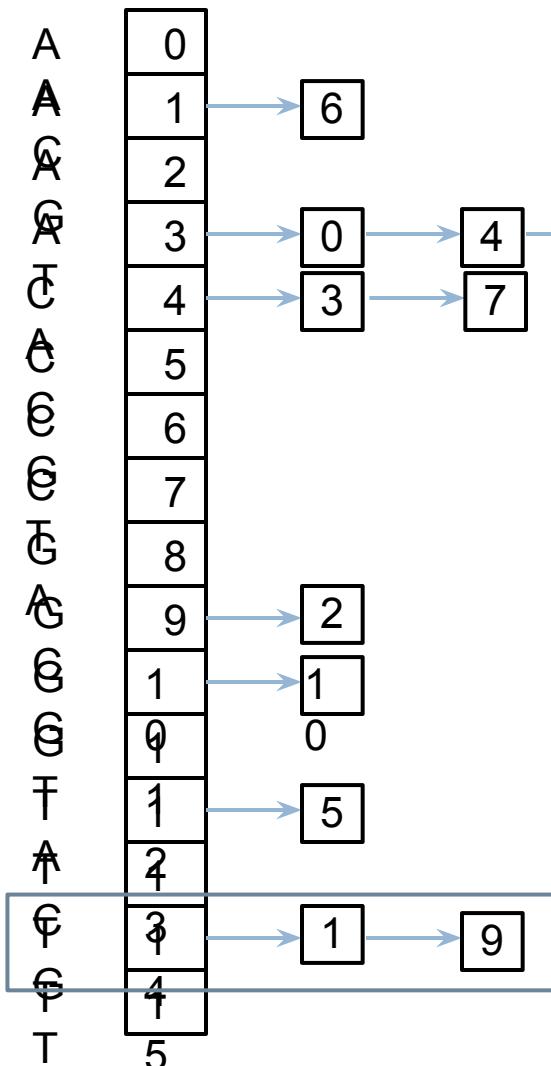
- **Example:** Hash table for
ATGCATACATGG

Hashing: Read Mapping

A	0
A	1
C	2
G	3
T	4
A	5
C	6
G	7
G	8
A	9
G	1
T	0
T	1
A	2
C	3
T	4
T	5

- **Example:** Hash table for ATGCATACATGG

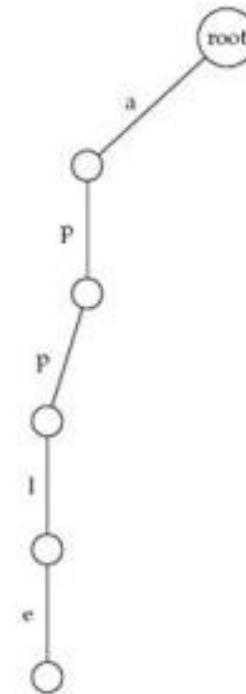
Hashing: Read Mapping



- **Example:** Hash table for ATGCATACATGG
- **Map TG**
 - Lookup TG
 - At positions 1, 9
- **Map GCAT**
 - Lookup GC and AT

Keyword Trees: Example

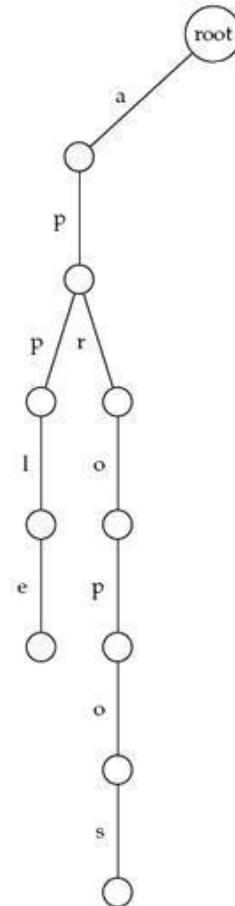
- **Keyword tree:**
 - Apple



Keyword Trees: Example (cont'd)

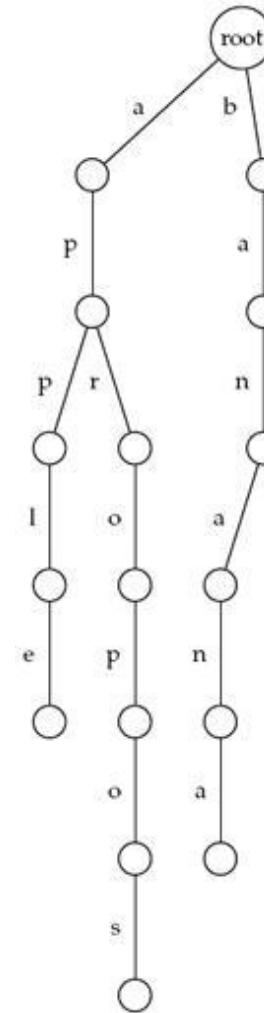
- **Keyword tree:**

- Apple
 - Apropos



Keyword Trees: Example (cont'd)

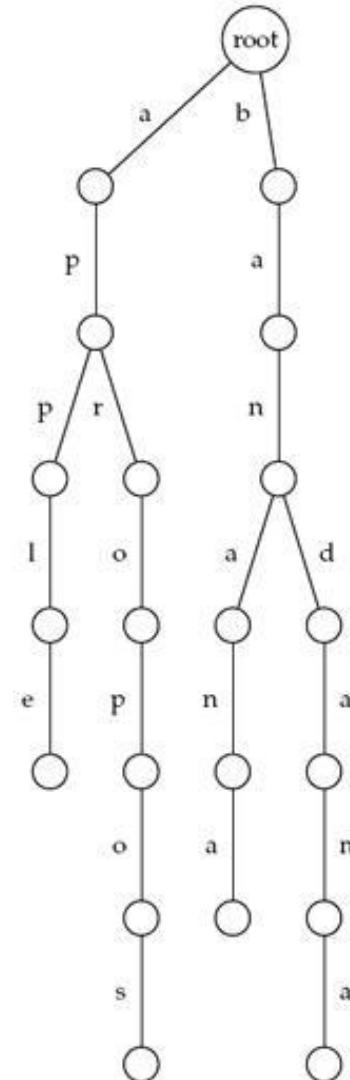
- **Keyword tree:**
 - Apple
 - Apropos
 - Banana



Keyword Trees: Example (cont'd)

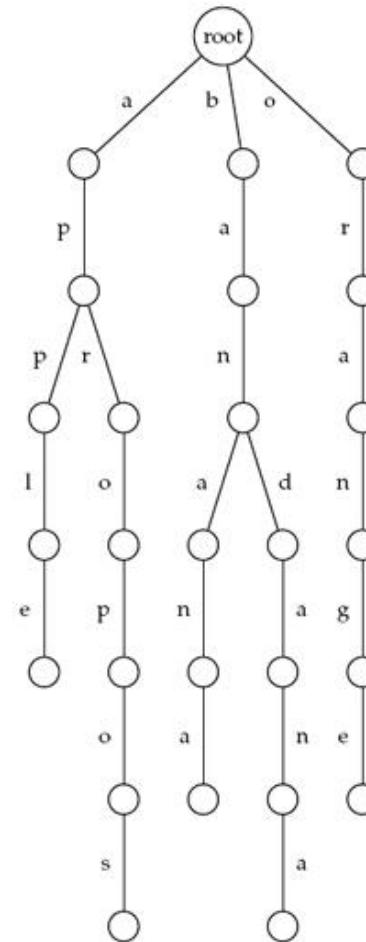
- **Keyword tree:**

- Apple
- Apropos
- Banana
- Bandana



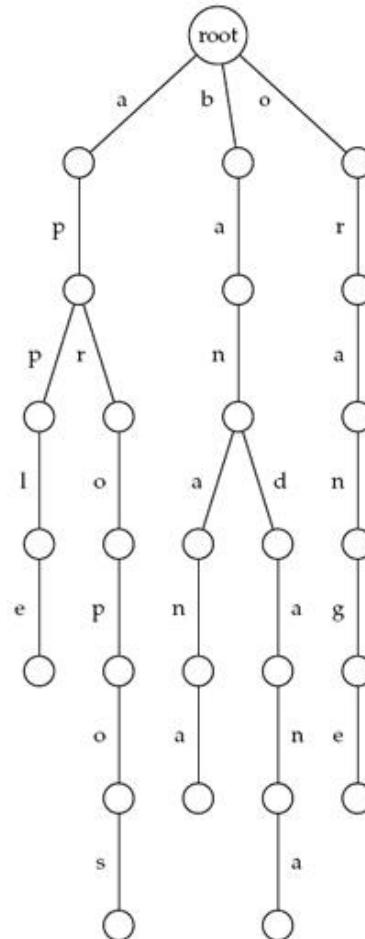
Keyword Trees: Example (cont'd)

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



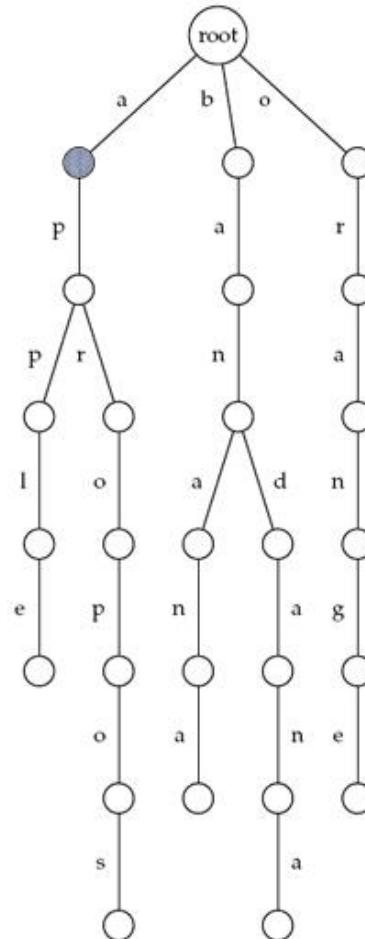
Keyword Trees: Properties

- 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



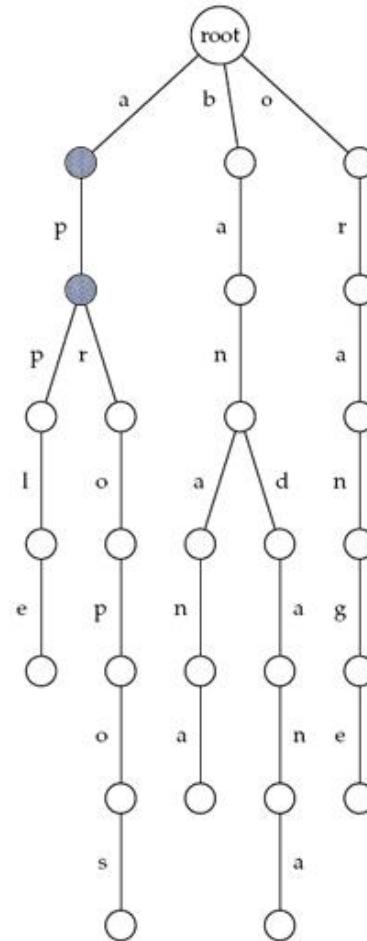
Keyword Trees: Threading (cont'd)

- Thread “appeal”
 - appeal



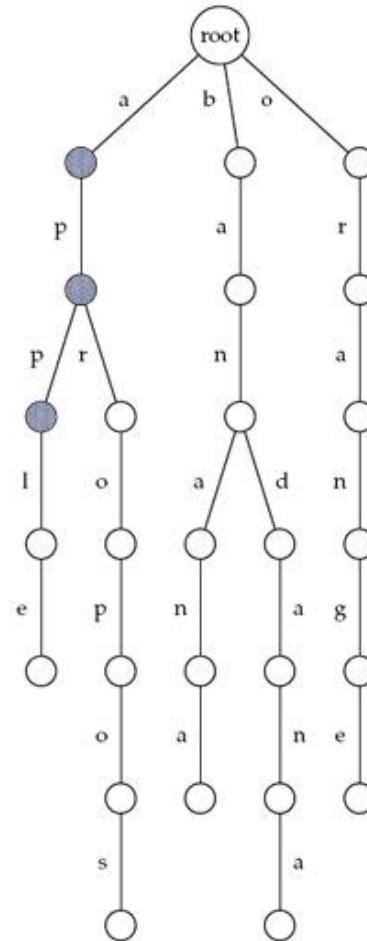
Keyword Trees: Threading (cont'd)

- Thread “appeal”
 - appeal



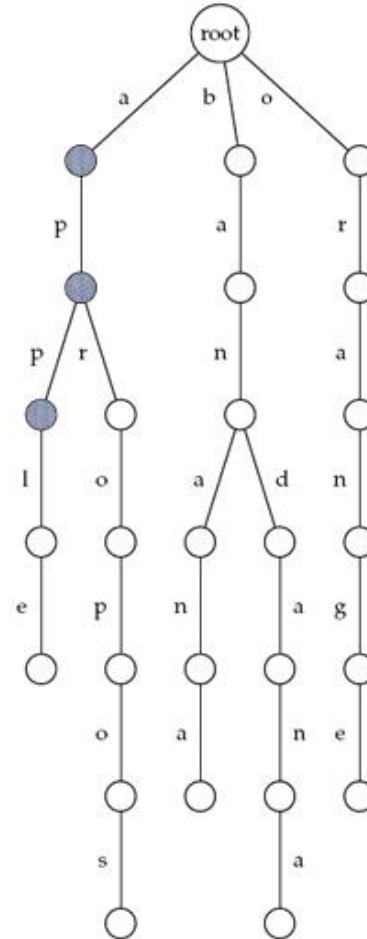
Keyword Trees: Threading (cont'd)

- Thread “appeal”
 - appeal



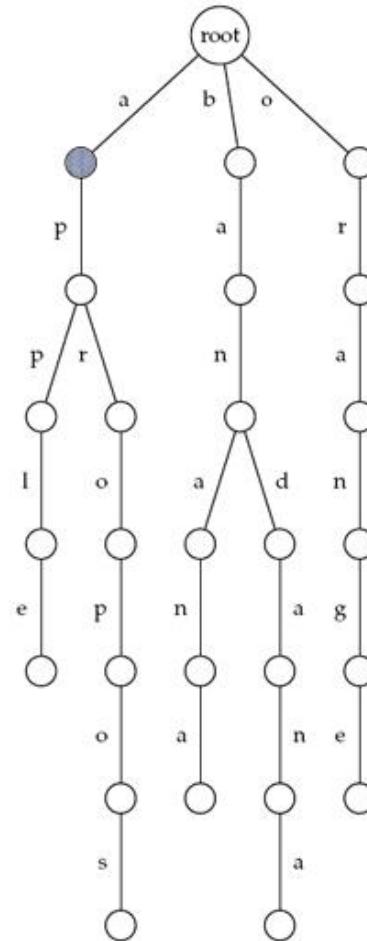
Keyword Trees: Threading (cont'd)

- Thread “appeal”
 - appeal



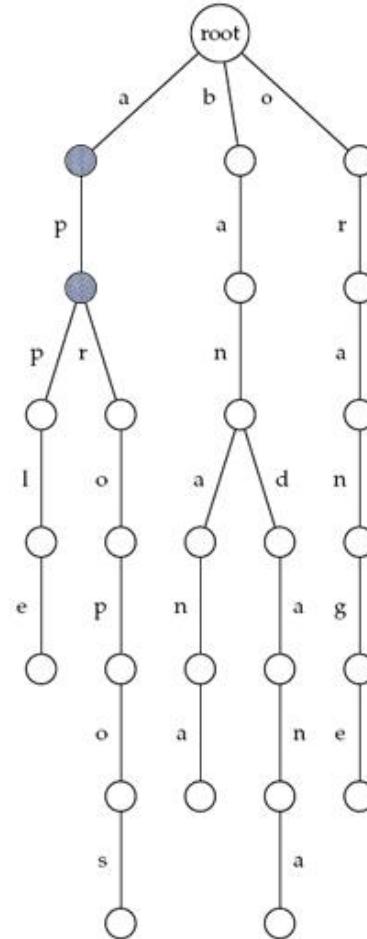
Keyword Trees: Threading (cont'd)

- Thread “apple”
 - apple



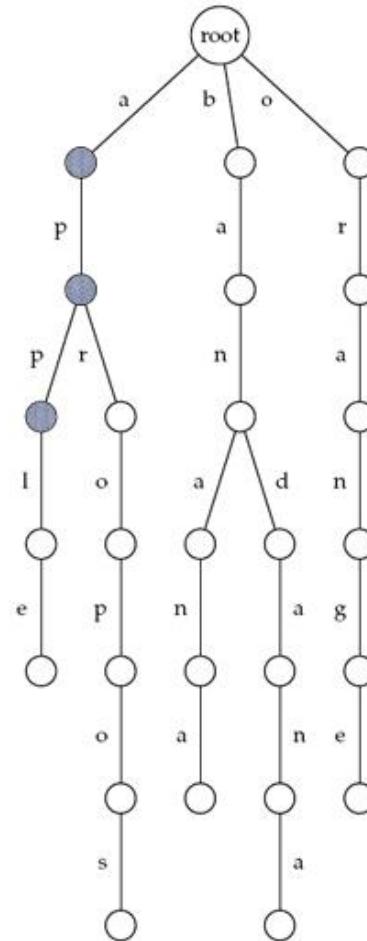
Keyword Trees: Threading (cont'd)

- Thread “apple”
 - apple



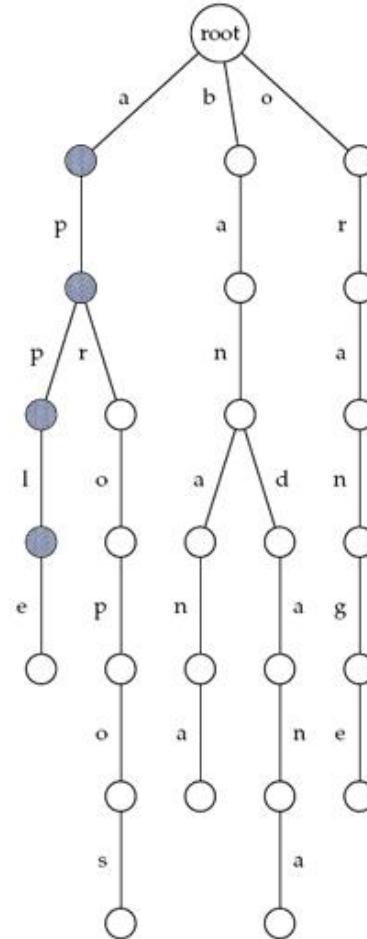
Keyword Trees: Threading (cont'd)

- Thread “apple”
 - apple



Keyword Trees: Threading (cont'd)

- Thread “apple”
 - apple



Keyword Trees: Threading (cont'd)

- Thread “apple”
 - apple

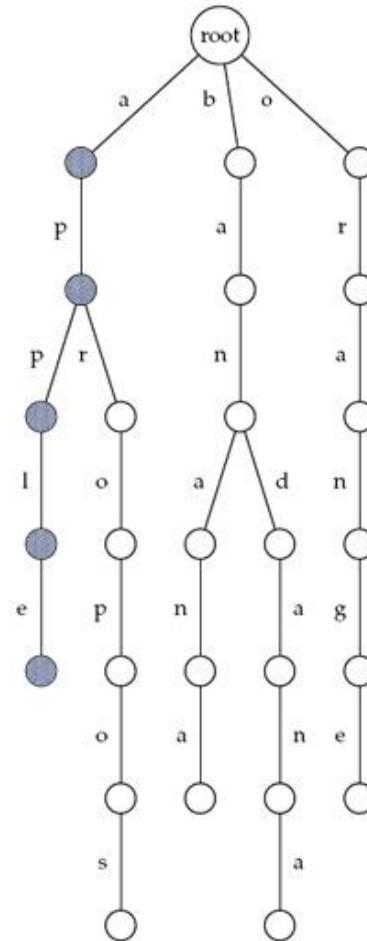


FIGURE 9.1 The trie for the following collection of strings Patterns: "ananas", "and", "antenna", "banana", "bandana", "nab", "nana", "pan".

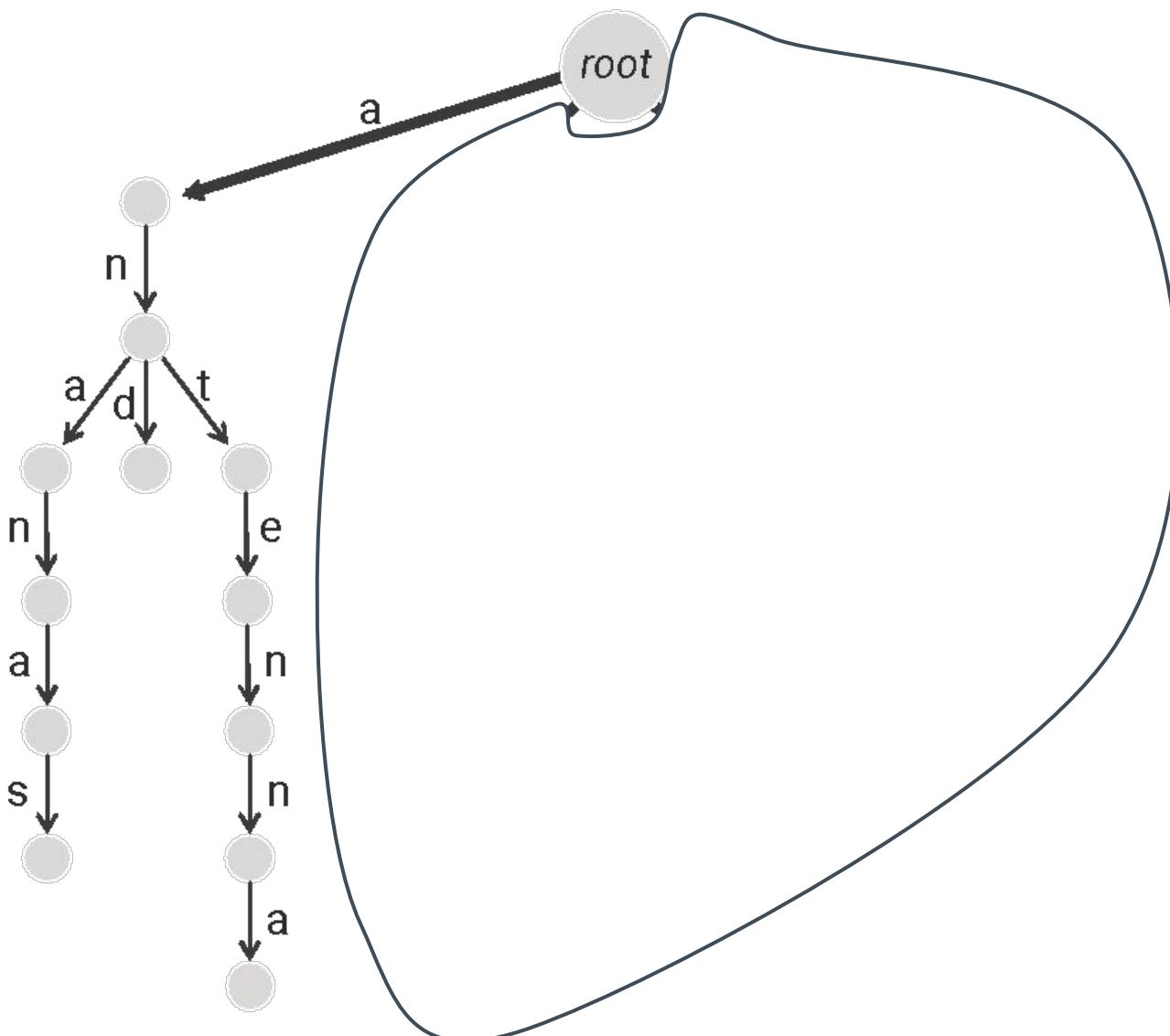


FIGURE 9.1 The trie for the following collection of strings Patterns: "ananas", "and", "antenna", "banana", "bandana", "nab", "nana", "pan".

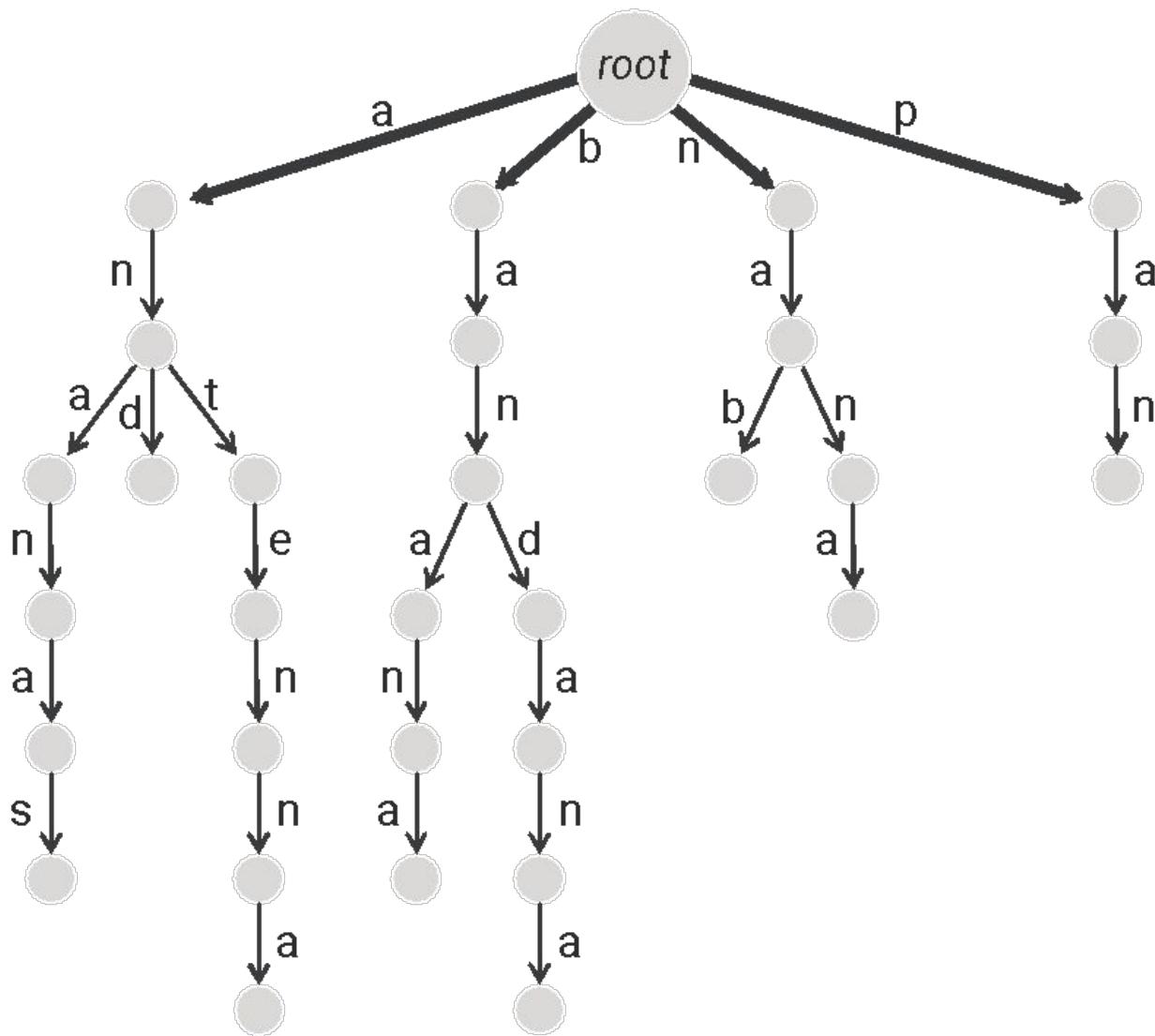


FIGURE 9.1 The trie for the following collection of strings Patterns: "ananas", "and", "antenna", "banana", "bandana", "nab", "nana", "pan".

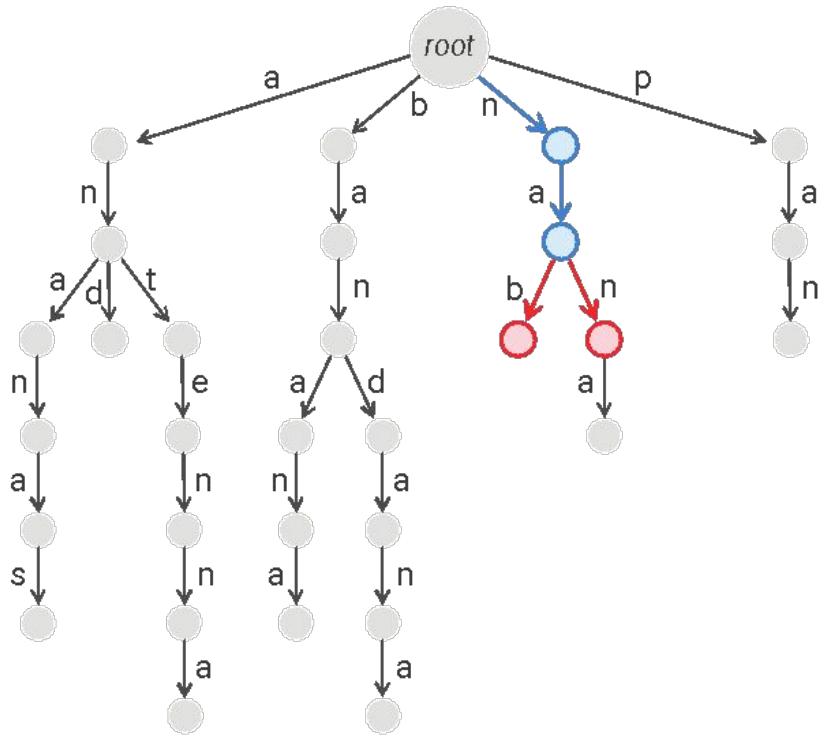
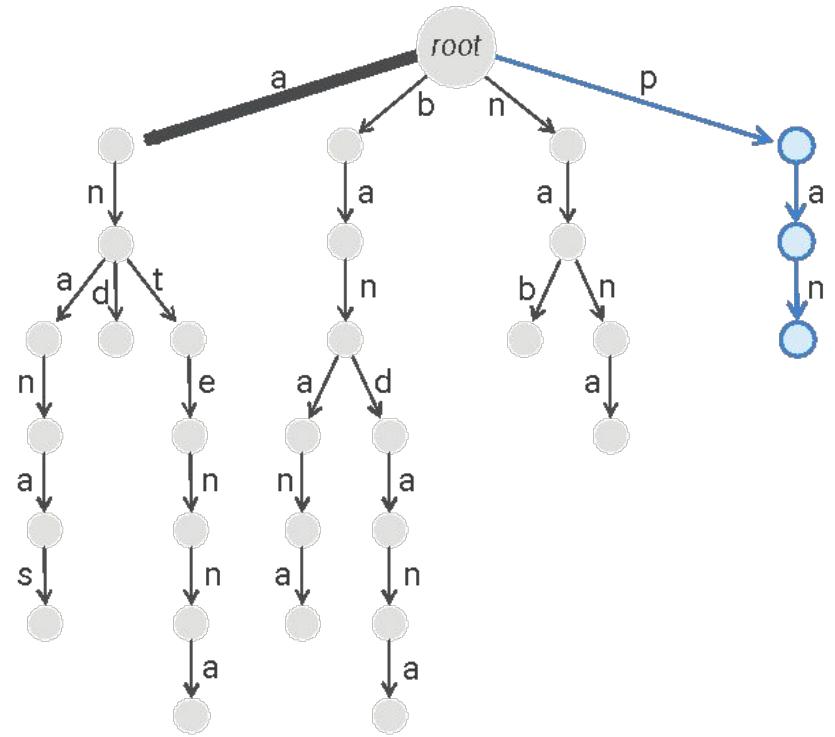


FIGURE 9.2 (Left) The pattern "pan" matches *Text* = "panamabanas" when starting at the beginning of *Text*. (Right) No pattern match is found in $\text{TRIE}(\text{Patterns})$ for the strings *Patterns* from Figure 9.1 when starting at the third symbol of "panamabanas".

Memory?

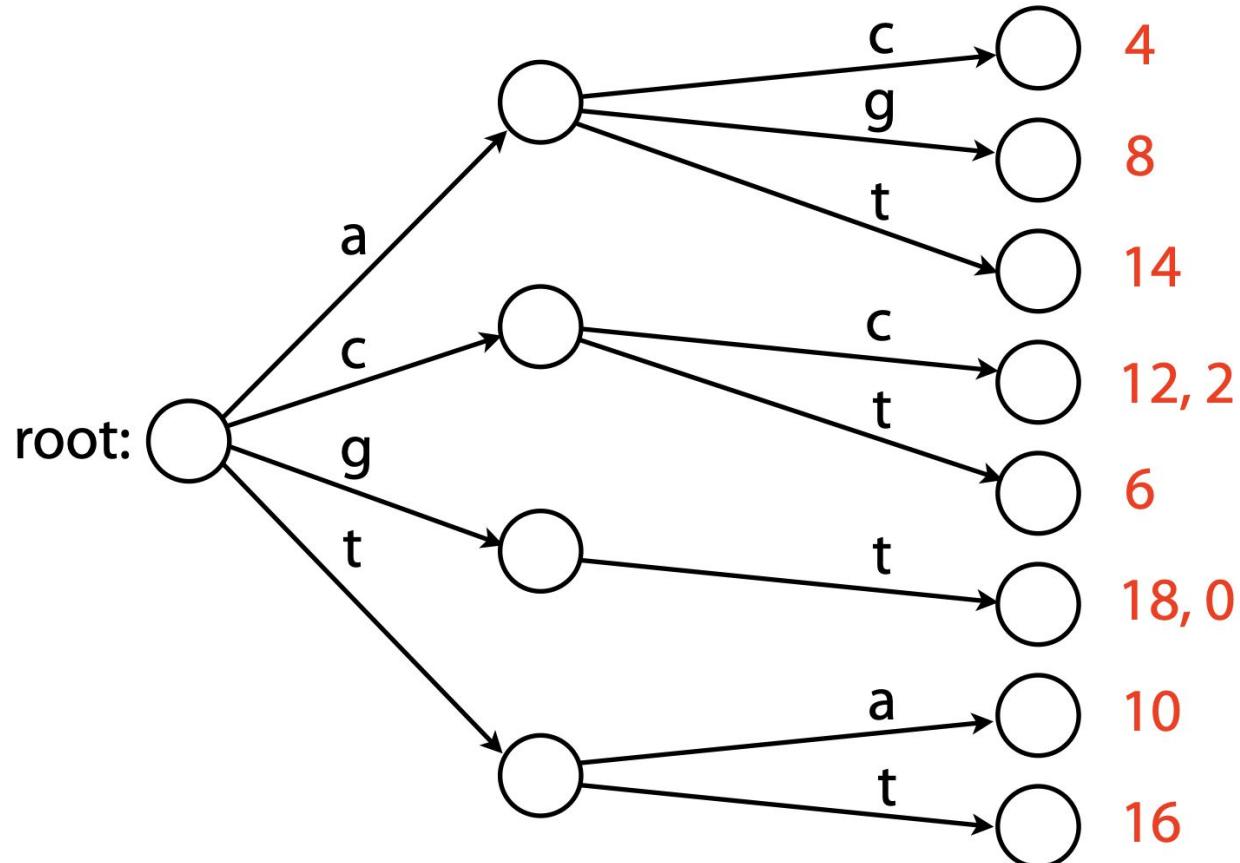
- 1 TB

Could use trie to represent k -mer index.

Map k -mers to offsets where they occur

ac	4
ag	8
at	14
cc	12
cc	2
ct	6
gt	18
gt	0
ta	10
tt	16

Index



Suffix Trie

Build a **trie** containing all **suffixes** of a text T

$T: GTTATAGCTGATCGCGGC GTAGCGG\$$

$GTTATAGCTGATCGCGGC GTAGCGG\$$

$TTATAGCTGATCGCGGC GTAGCGG\$$

$TATAGCTGATCGCGGC GTAGCGG\$$

$ATAGCTGATCGCGGC GTAGCGG\$$

$TAGCTGATCGCGGC GTAGCGG\$$

$AGCTGATCGCGGC GTAGCGG\$$

$GCTGATCGCGGC GTAGCGG\$$

$CTGATCGCGGC GTAGCGG\$$

$TGATCGCGGC GTAGCGG\$$

$GATCGCGGC GTAGCGG\$$

$ATCGCGGC GTAGCGG\$$

$TCGCGGC GTAGCGG\$$

$CGCGGC GTAGCGG\$$

$CGGGCG GTAGCGG\$$

$CGGC GTAGCGG\$$

$GGCG GTAGCGG\$$

$GCGT AGCGG\$$

$CGTAGCGG\$$

$GTAGCGG\$$

$TAGCGG\$$

$AGCGG\$$

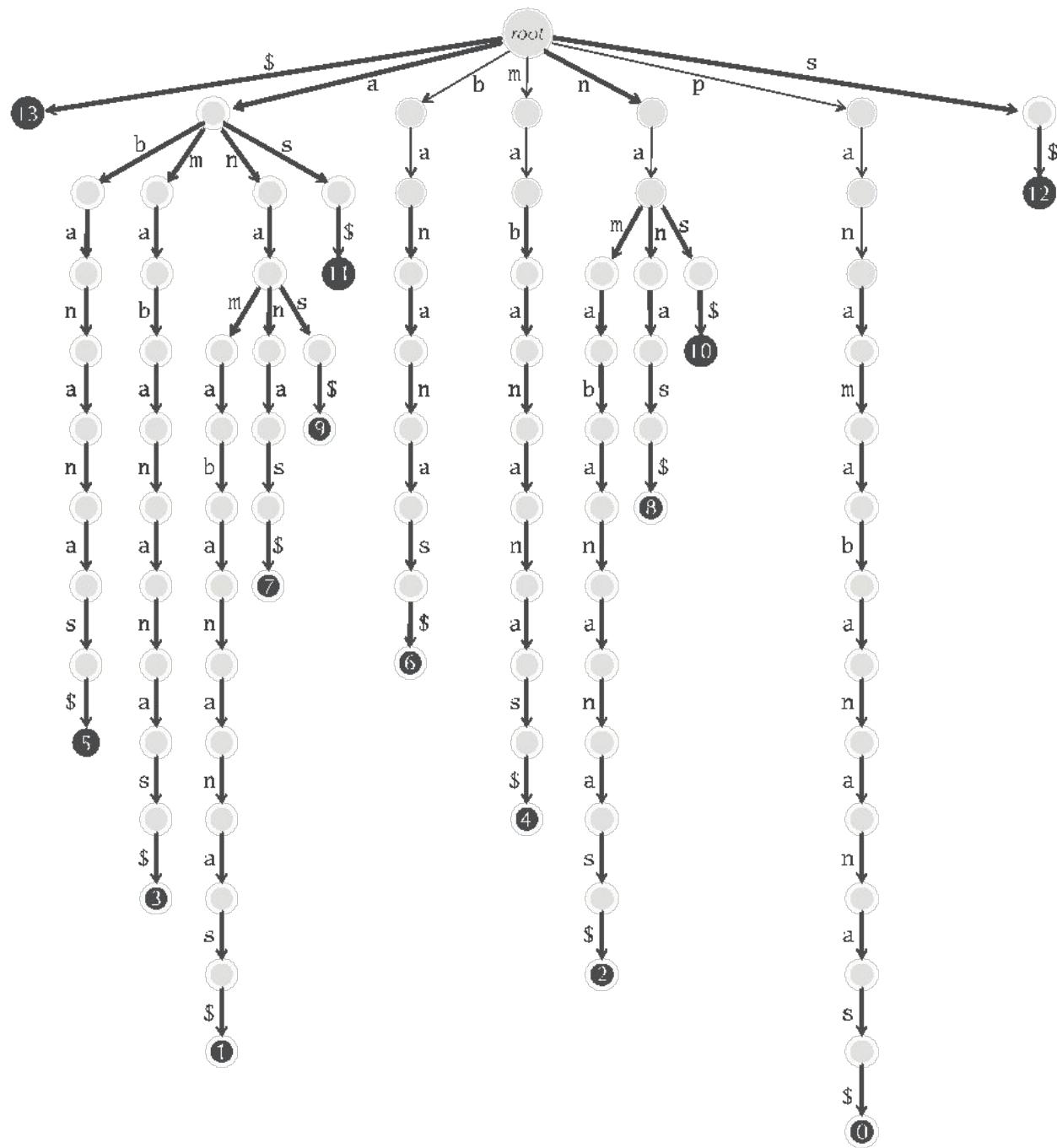
$GC GG\$$

$GG \$$

$G \$$

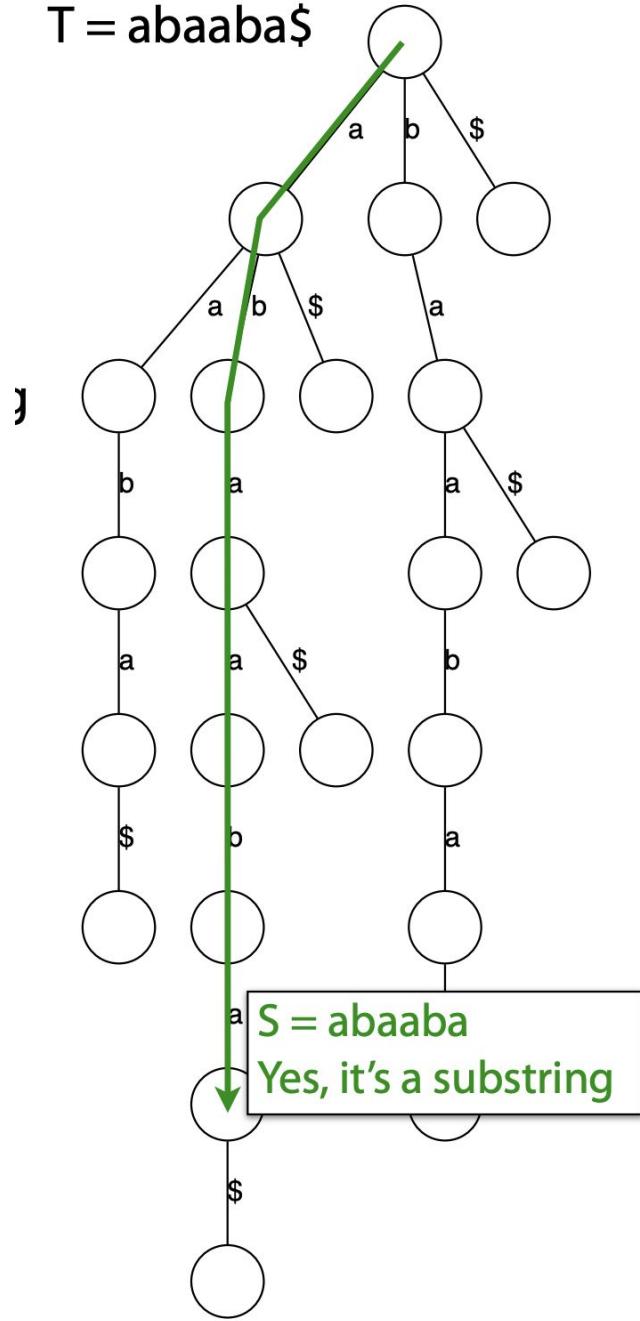
$\$$

$m(m+1)/2$
chars



- ant
- Nana
- ana

$T = abaaba\$$



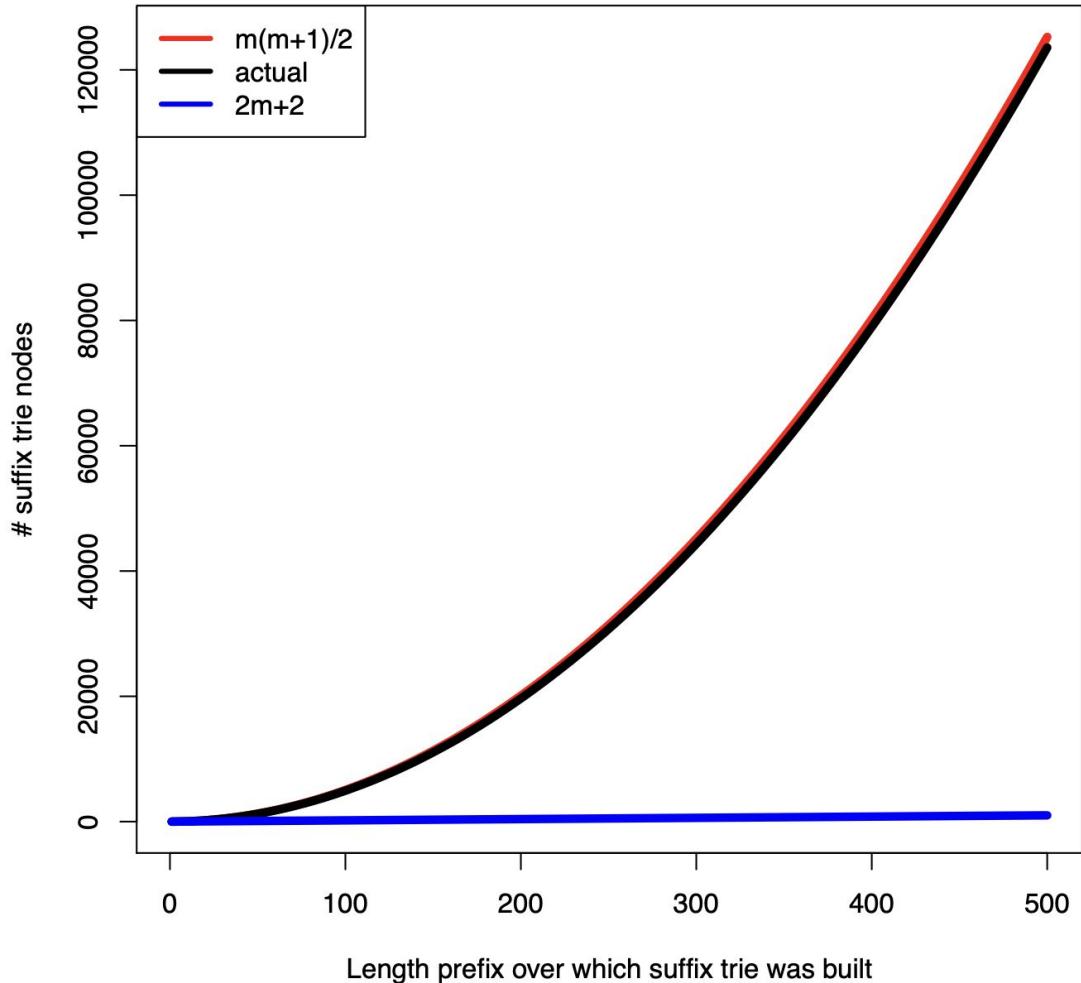
- baa
- aba

Suffix Trie

We saw the suffix trie, but we also saw its size grows *quadratically* with the length of the string

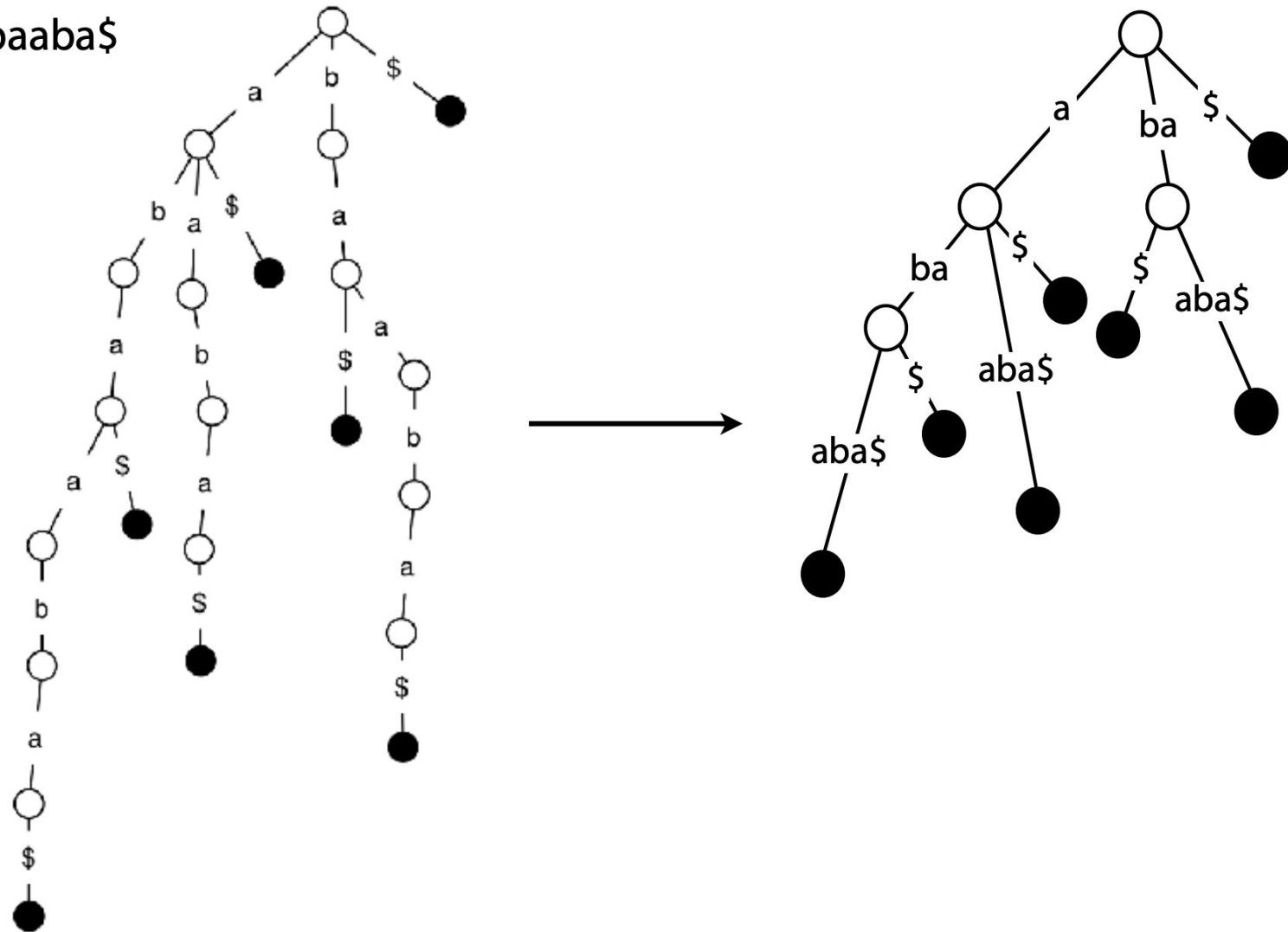
Human genome is $3 \cdot 10^9$ bases long.

If $m = 3 \cdot 10^9$, m^2 is way huge, far beyond what we can store in memory



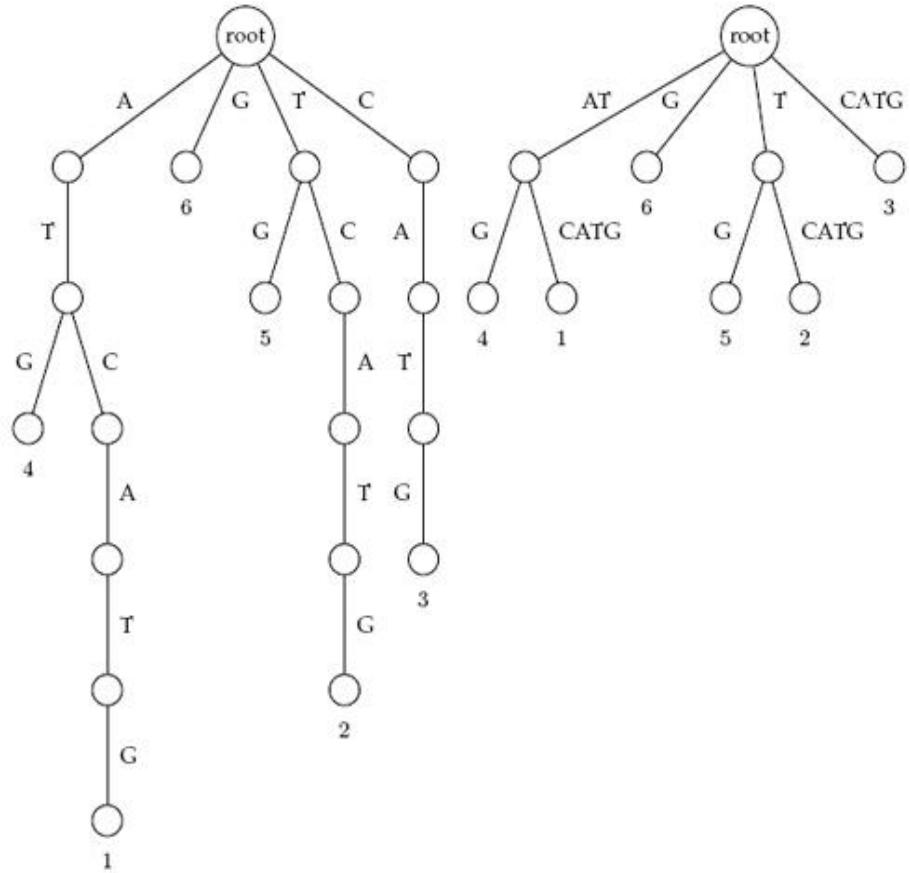
Making it smaller

$T = abaaba\$$



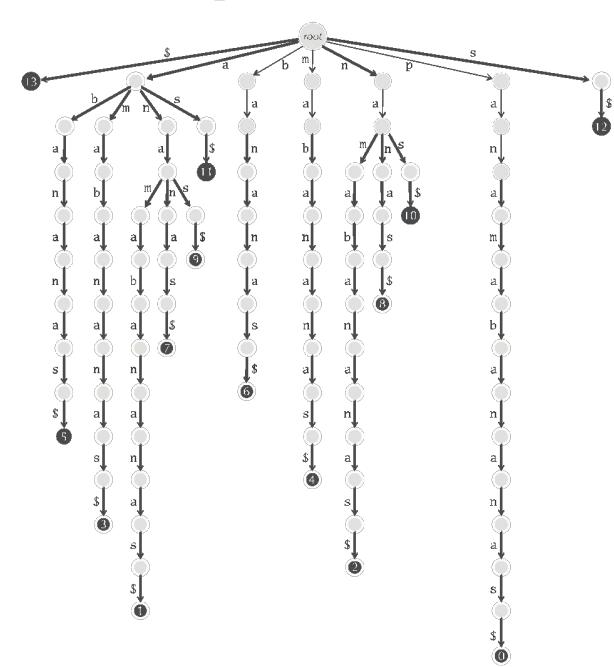
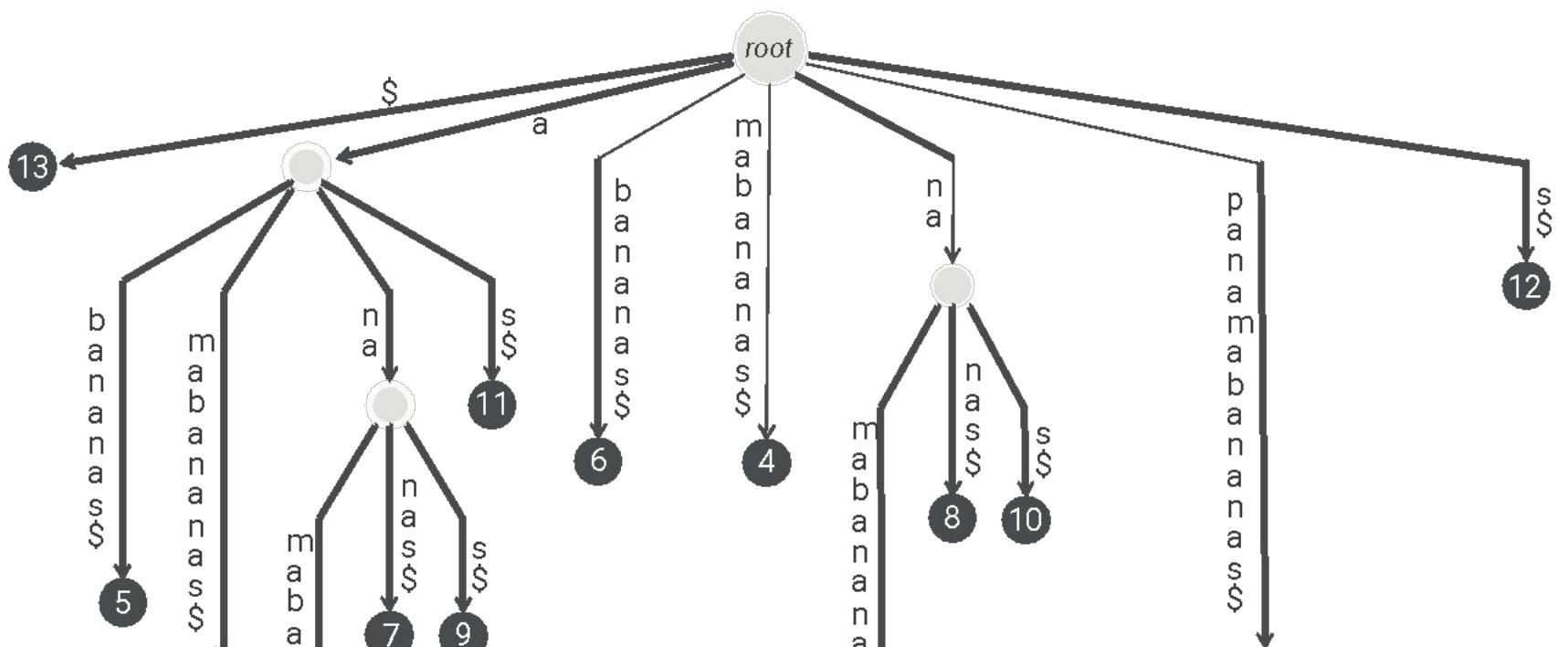
Suffix Tree=Collapsed Suffix Trie

- Similar to keyword trees, except edges that form paths are collapsed
 - Each edge is labeled with a *substring* of a text
 - All internal nodes have at least two outgoing edges
 - Leaves labeled by the index of the pattern.

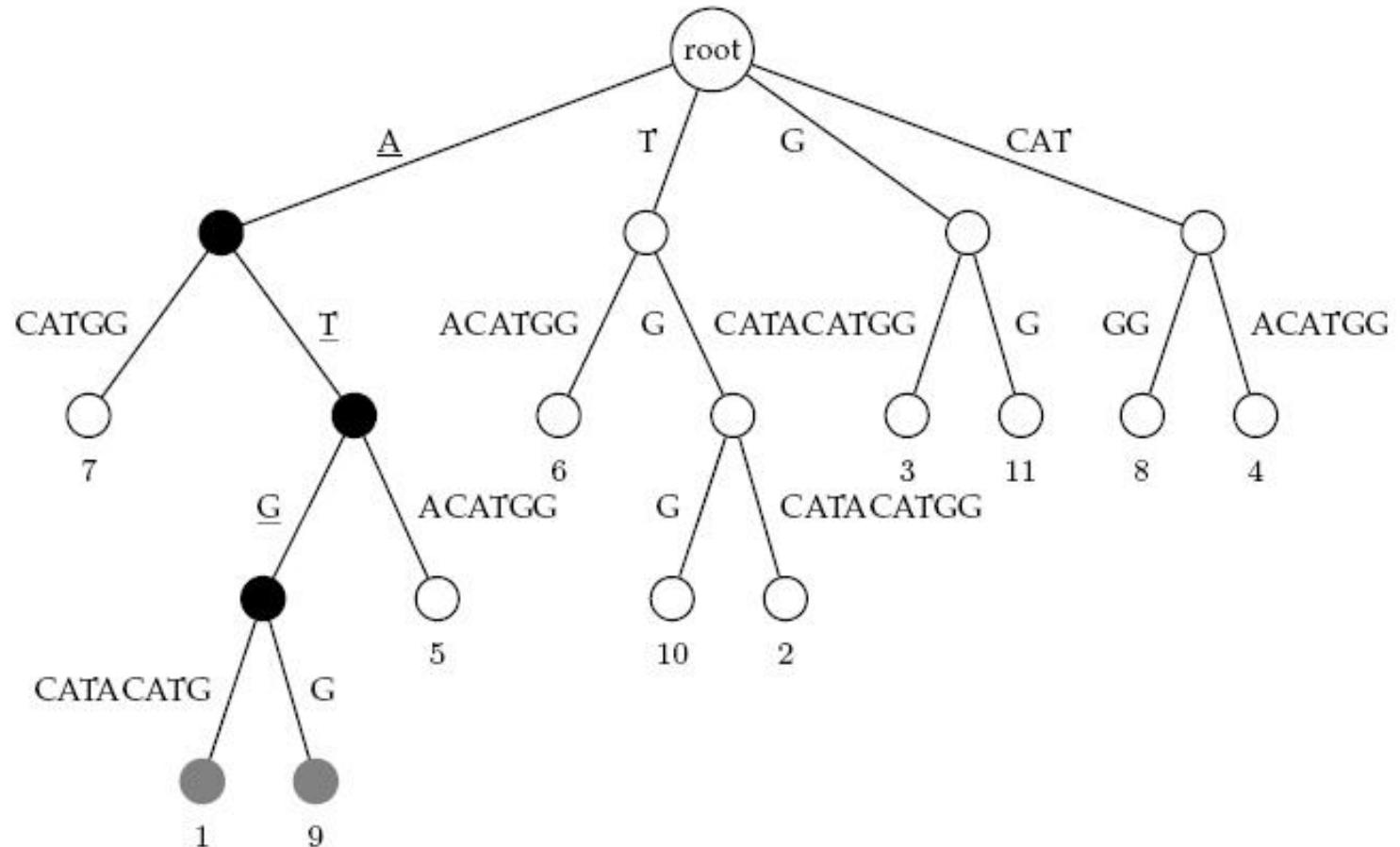


(a) Keyword tree

(b) Suffix tree

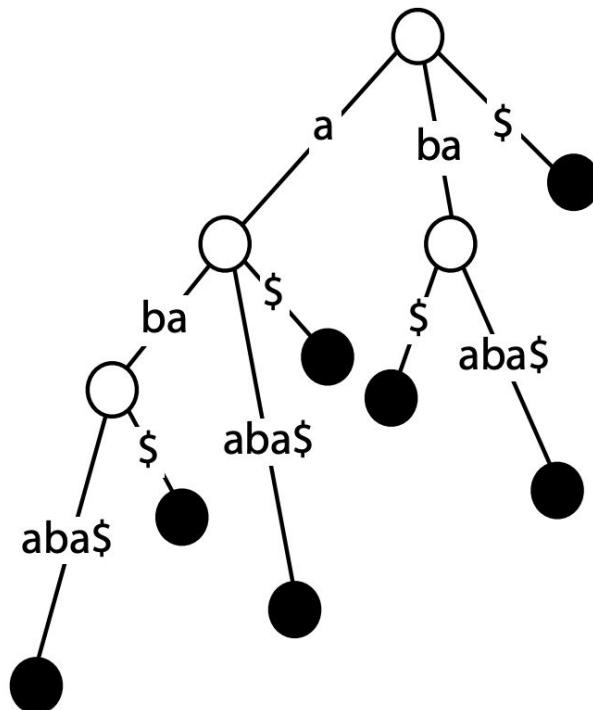


Threading ATG through a Suffix Tree



Suffix Tree

$$T = \text{abaaba\$} \quad |T| = m$$



Is *total size O(m)* now?

leaves? m

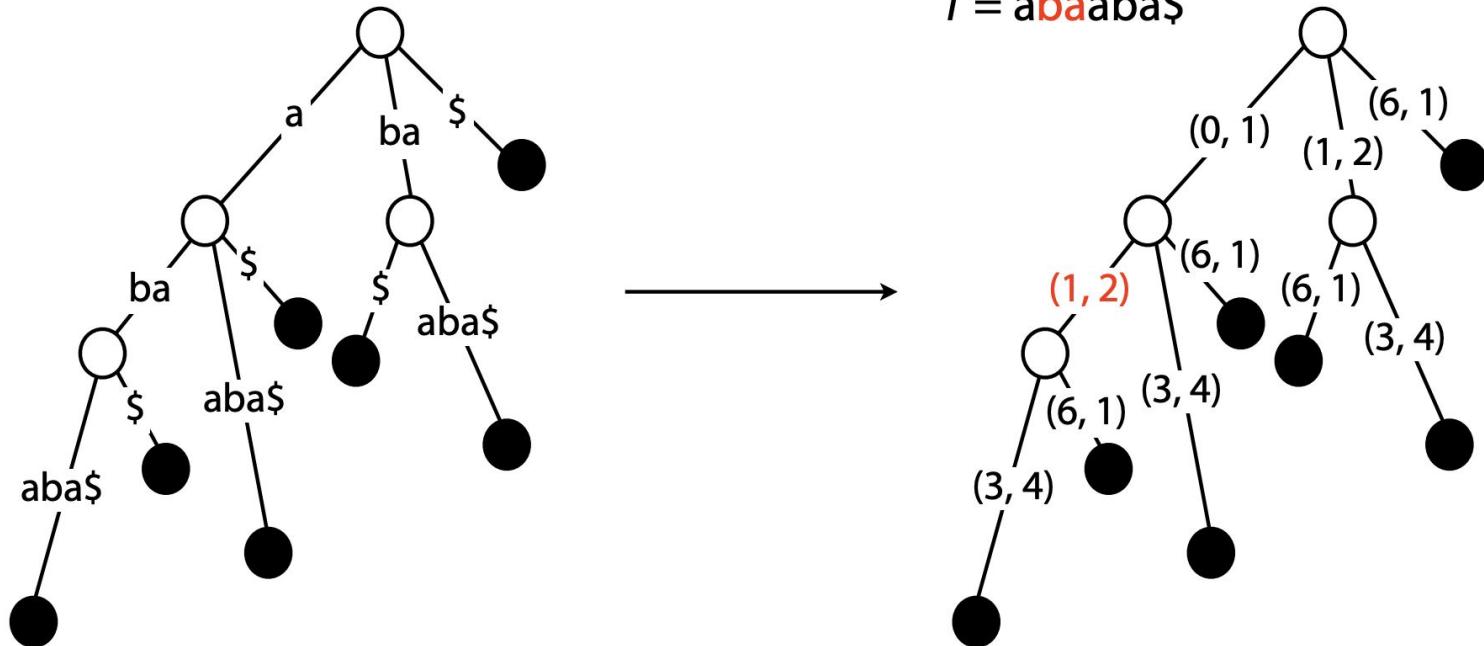
non-leaf nodes (bound)? $\leq m - 1$

$\leq 2m - 1$ nodes total — $O(m)$

No: total length of edge labels grows with m^2

Suffix Tree

Idea 2: Store T itself in addition to the tree. Convert tree's edge labels to (offset, length) pairs with respect to T .

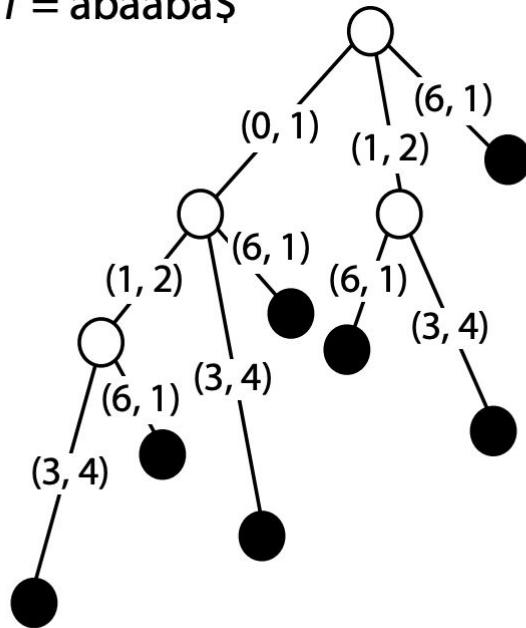


Space is now $O(m)$ Suffix trie was $O(m^2)$!

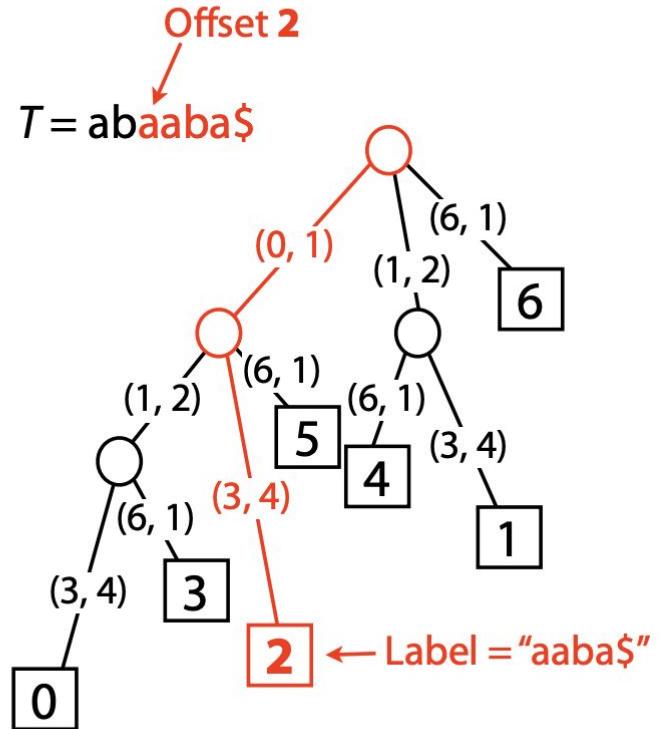
Suffix Tree

Suffix tree: leaves hold offsets

$T = abaaba\$$



$T = aba\textcolor{red}{a}ba\$$



Space is now $O(m)$ Suffix trie was $O(m^2)$!

Suffix Tree of a Text

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

ATCATG
TCATG
CATG
ATG
TG
G



Suffix Tree of a Text

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

ATCATG
TCATG
CATG
ATG
TG
G



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

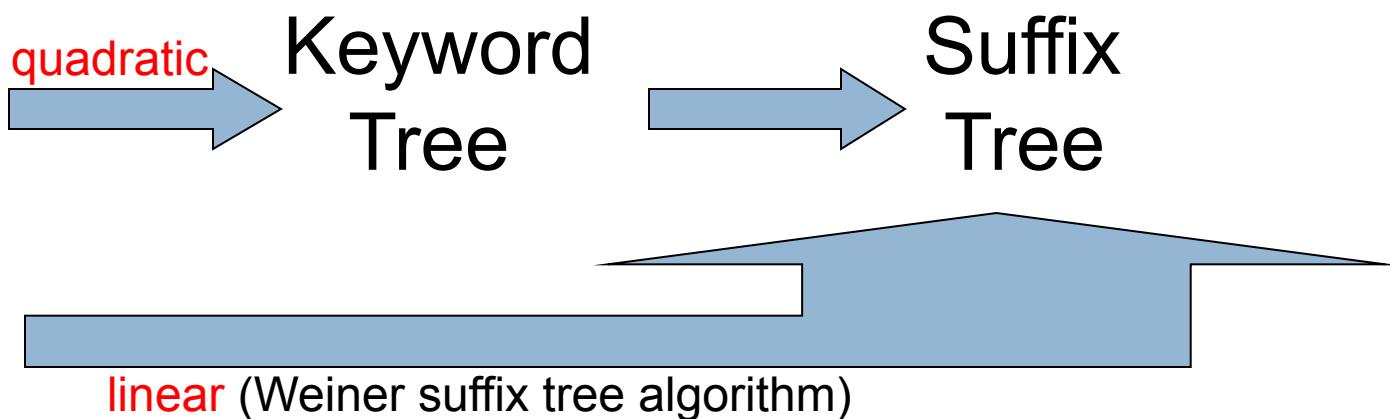


Time is linear in the total size of all suffixes,
i.e., it is quadratic in the length of the text

Suffix Trees: Advantages

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

ATCATG
TCATG
CATG
ATG
TG
G



On-Line Construction of Suffix Trees¹

E. Ukkonen²

Abstract. An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix tries. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWGs).

Key Words. Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

Canonical algorithm for $O(m)$ time & space suffix tree construction

Use of Suffix Trees

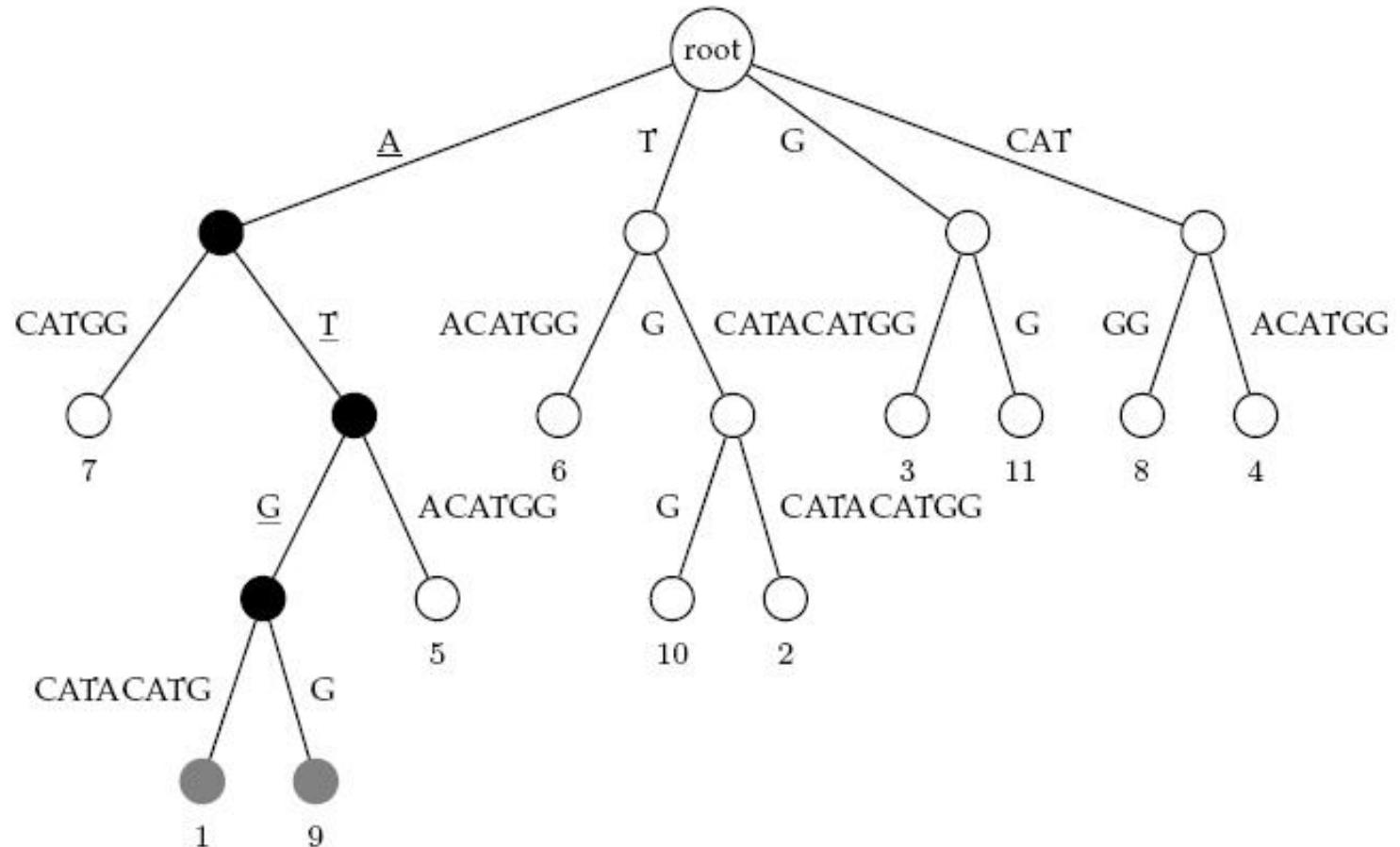
- Suffix trees hold all suffixes of a text
 - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
 - Builds in $O(m)$ time for text of length m
- To find any pattern of length n in a text:
 - Build suffix tree for text
 - Thread the pattern through the suffix tree
 - Can find pattern in text in $O(n)$ time!
- $O(n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

Pattern Matching with Suffix Trees

SuffixTreePatternMatching(p,t)

- 1 Build **suffix tree** for text **t**
- 2 Thread pattern **p** through **suffix tree**
- 3 **if** threading is complete
 - 4 **output** positions of all **p**-matching **leaves** in the tree
- 5 **else**
 - 6 **output** “Pattern does not appear in text”

Threading ATG through a Suffix Tree

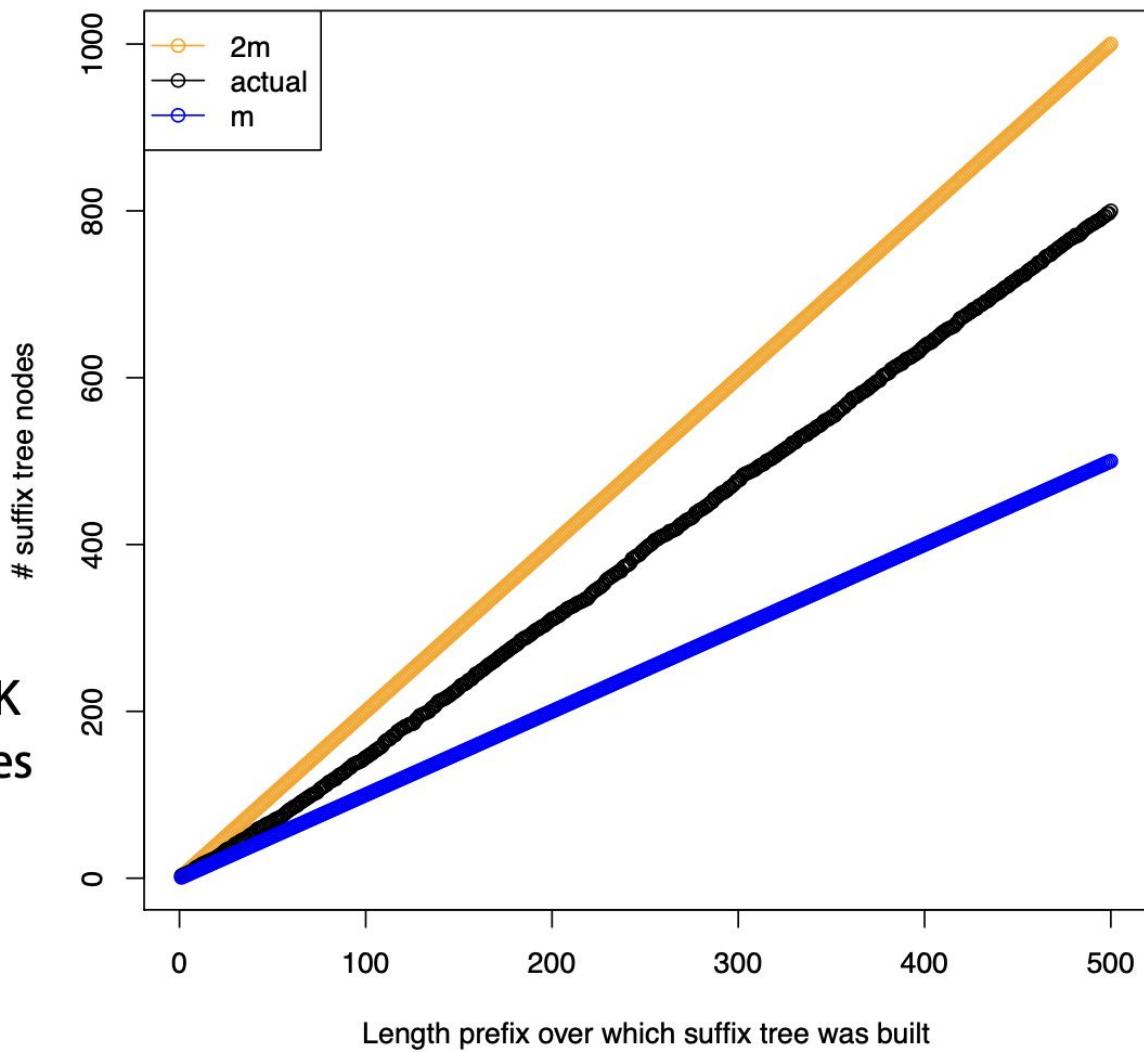
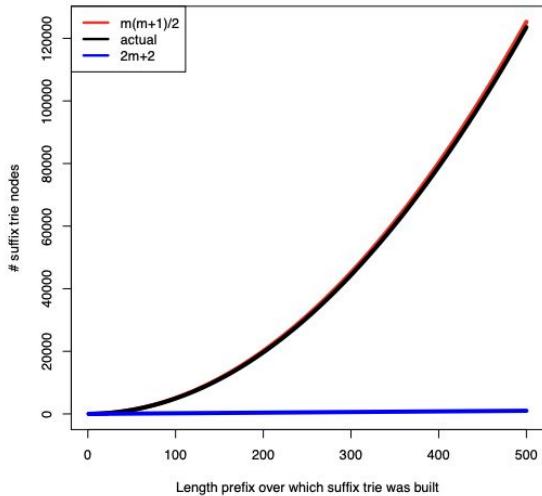


Suffix tree: actual growth

Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Remember suffix trie plot:



Suffix Array

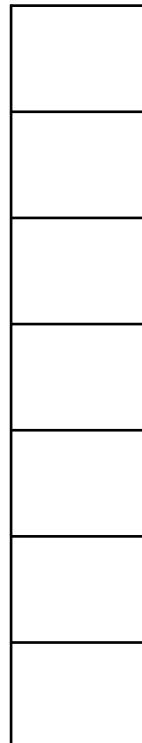
- More space efficient than suffix tree
 - Suffix tree index for human genome $\geq 47\text{GB}$
- Lexicographically sort all the suffixes
- Store the starting indices of the suffixes along with the original string

Suffix Array

$T = \text{abaaba\$}$
0123456

As with suffix tree,
 T is part of index

$\text{SA}(T) =$



Suffix array

$T = \text{abaaba\$}$ ← As with suffix tree,
 T is part of index
0123456

$\text{SA}(T) =$	<table border="1"><tr><td>6</td><td>\$</td></tr><tr><td>5</td><td>a \$</td></tr><tr><td>2</td><td>a a b a \$</td></tr><tr><td>3</td><td>a b a \$</td></tr><tr><td>0</td><td>a b a a b a \$</td></tr><tr><td>4</td><td>b a \$</td></tr><tr><td>1</td><td>b a a b a \$</td></tr></table>	6	\$	5	a \$	2	a a b a \$	3	a b a \$	0	a b a a b a \$	4	b a \$	1	b a a b a \$	m integers
6	\$															
5	a \$															
2	a a b a \$															
3	a b a \$															
0	a b a a b a \$															
4	b a \$															
1	b a a b a \$															

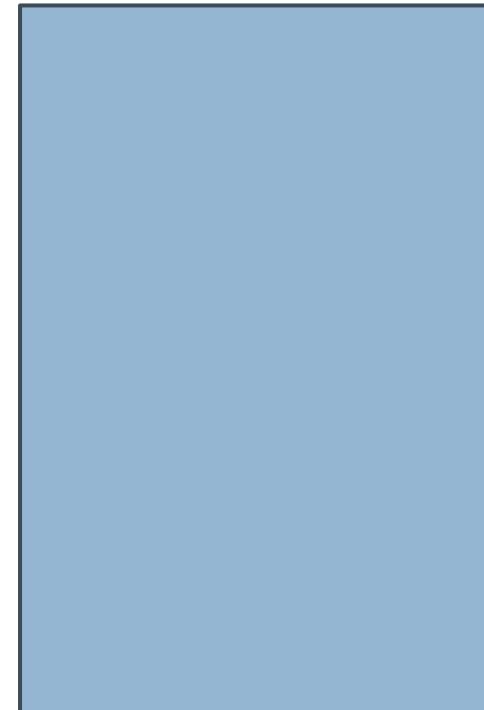
Suffix array of T is an array of integers in $[0, m)$ specifying lexicographic (alphabetical) order of T 's suffixes

Suffix Array

- ATCATG

1	ATCATG\$
2	TCATG\$
3	CATG\$
4	ATG\$
5	TG\$
6	G\$
7	\$

Sort the
suffixes
lexicographical
y



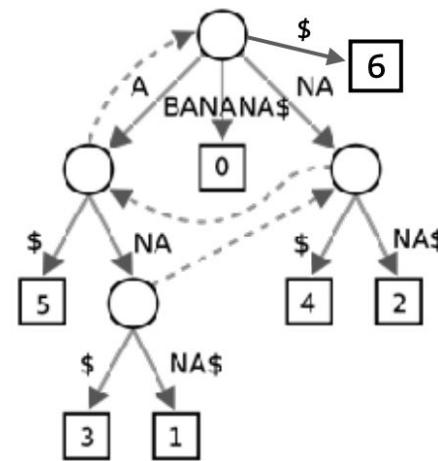
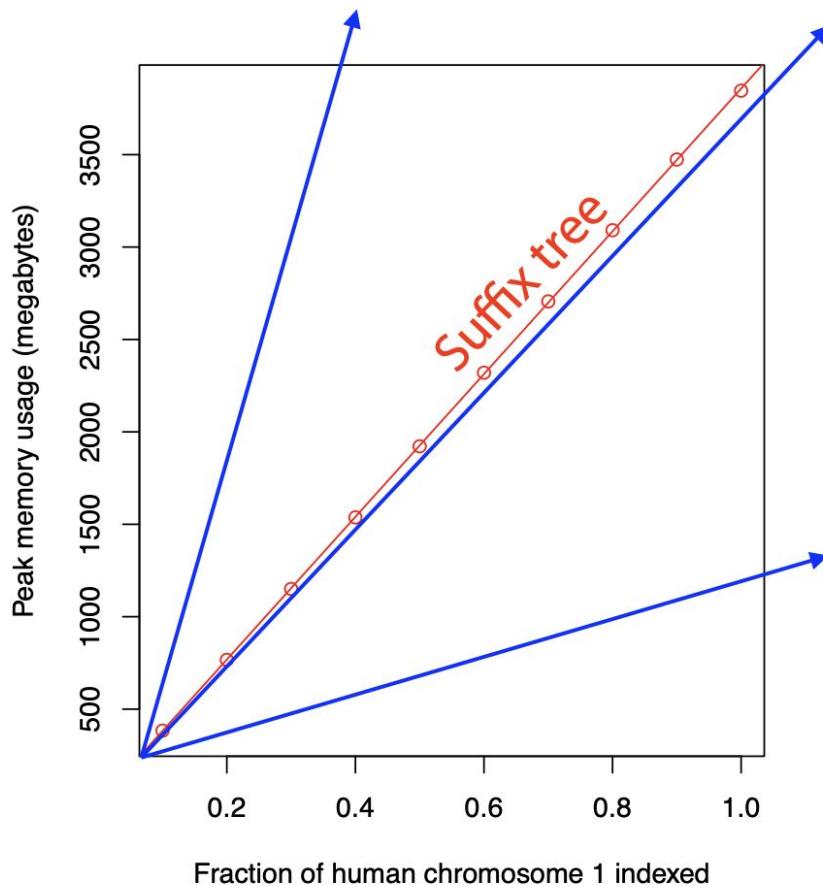
Sorted Suffixes	Starting Positions
\$	13
abananas\$	5
amabananas\$	3
anamabananas\$	1
anas\$	7
anas\$	9
as\$	11
bananas\$	6
mabananas\$	4
namabananas\$	2
nanas\$	8
nas\$	10
panamabananas\$	0
s\$	12

SUFFIXARRAY("panamabananas\$") = [13, 5, 3, 1, 7, 9, 11, 6, 4, 2, 8, 10, 0, 12].

Suffix array

$O(m)$ space, like suffix tree

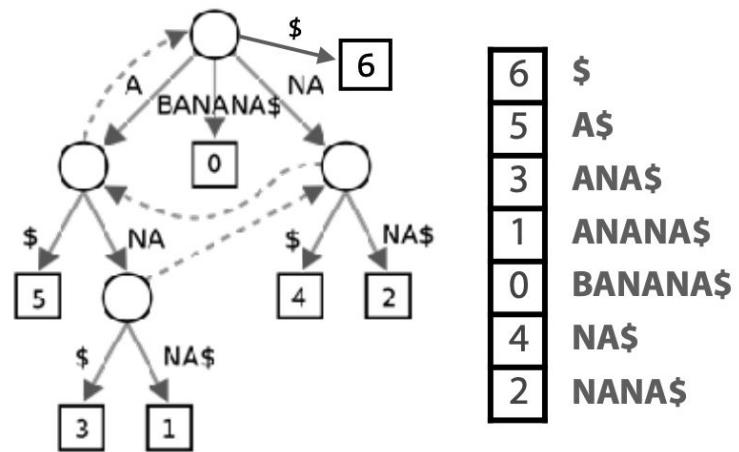
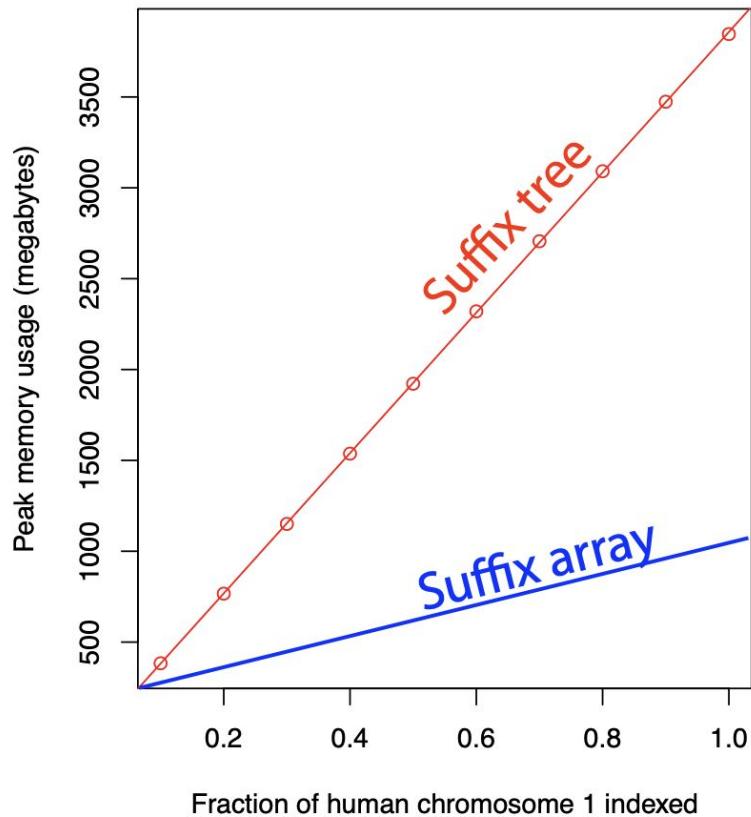
Is “constant factor” worse, better, same?



6	\$
5	A\$
3	ANAS\$
1	ANANA\$
0	BANANA\$
4	NA\$
2	NANA\$

Suffix array

32-bit integers sufficient for human genome, so fits in
~4 bytes/base \times 3 billion bases \approx 12 GB. Suffix tree is >45 GB.



Suffix Array

- There exists algorithms to construct suffix array in $O(n)$ time
- Searching is done using binary search
 - Basic binary search would take $O(m\log n)$
 - Can be done in $O(m + \log n)$

PATTERNMATCHINGWITHSUFFIXARRAY(*Text*, *Pattern*, **SUFFIXARRAY**)

```
minIndex ← 0
maxIndex ← |Text|
while minIndex < maxIndex
    midIndex ← (minIndex + maxIndex) / 2
    if Pattern > suffix of Text starting at position SUFFIXARRAY(midIndex)
        minIndex ← midIndex + 1
    else
        maxIndex ← midIndex
    first ← minIndex
    maxIndex ← |Text|
    while minIndex < maxIndex
        midIndex ← (minIndex + maxIndex) / 2
        if Pattern < suffix of Text starting at position SUFFIXARRAY(midIndex)
            maxIndex ← midIndex
        else
            minIndex ← midIndex + 1
    last ← maxIndex
    if first > last
        return "Pattern does not appear in Text"
    else
        return (first, last)
```

$T = \text{panamabananas\$}$

Cyclic Rotations

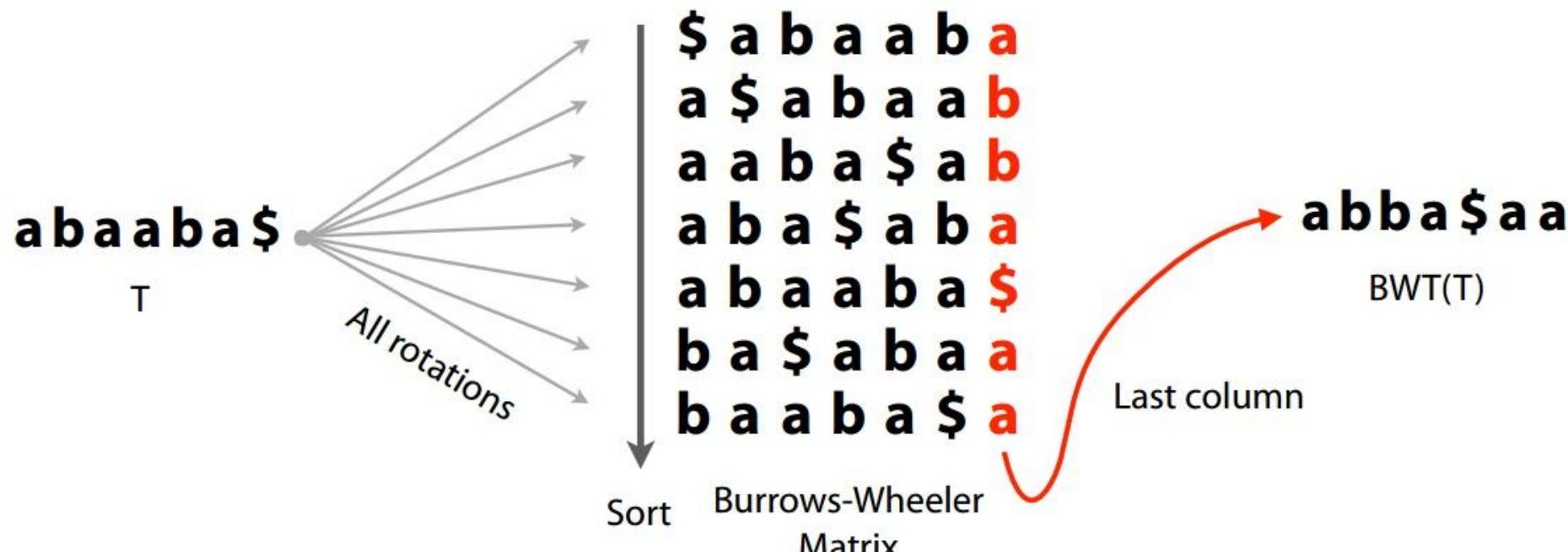
panamabananas\$

$M(\text{" panamabananas\$" })$



Burrows-Wheeler Transform (BWT)

- Lexicographically sort all the *rotations*



How is it useful for compression?

How is it reversible?

How is it an index?

Burrows-Wheeler Transform (BWT)

- Sort order is the same for rotations and suffixes
 - Since \$ precedes others

T = a b a a b a \$

\$ a b a a b a
a \$ a b a a b
a a b a \$ a b
a b a \$ a b a
a b a a b a \$
b a \$ a b a a
b a a b a \$ a

BWM(T)

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

SA(T)

Burrows-Wheeler Transform (BWT)

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases}$$

T = a b a a b a \$

\$ a b a a b a
a \$ a b a a b
a a b a \$ a b
a b a \$ a b a
a b a a b a \$
b a \$ a b a a
b a a b a \$ a

BWM(T)

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

SA(T)

Burrows-Wheeler Transform (BWT)

- Original motivation was compression
- Rotating and sorting tend to increase run lengths

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set \$L[i]\$ to be the
i	n turn, set \$R[i]\$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector \$R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with \$ch\$ appear in the {\em same order}
i	n with \$ch\$. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell-\cite{bell}.

Figure 1: Example of sorted rotations. Twenty consecutive rotations from the sorted list of rotations of a version of this paper are shown, together with the final character of each rotation.

Figure from original paper on
BWT

Burrows-Wheeler Transform (BWT)

nd Corey (1). They kindly made their manuscript available a
nd criticism especially on interatomic distances. We a
nd cytosine. The sequence of bases on a single chain and a
nd experimentally (3, 4) that the ratio of the amounts of u
nd for this reason we shall not comment on it. We wish a
nd guanine (purine) with cytosine (pyrimidine). In other a
nd ideas of Dr. M H F. Wilkins, Dr. R E. Franklin a
nd its water content is rather high. At lower water content a
nd pyrimidine bases. The planes of the bases are perpendicular a
nd stereochemical arguments. It has not escaped our notice a
nd that only specific pairs of bases can bond together a
nd the atoms near it is close to Furberg's 'standard conformation' a
nd the bases on the inside, linked together by hydrogen bonds a
nd the bases on the outside. In our opinion, this structure a
nd the other a pyrimidine for bonding to occur. The hydroxyl groups a
nd the phosphates on the outside. The configuration of a
nd the ration of guanine to cytosine, are always very close a
nd the same axis (see diagram). We have made the usual a
nd their co-workers at King's College, London. One of a

LF mapping

- Last to first mapping

\$ a b a a b a
a \$ a b a a b
a a b a \$ a b
a b a \$ a b a
a b a a b a \$
b a \$ a b a a
b a a b a \$ a

LF mapping

- Assign numbers to distinguish characters

- $a\ b_0\ a\ a\ b_1\ a\ \$$

$\$ \ a\ b\ a\ a\ b\ a$
 $a\ \$\ a\ b\ a\ a\ b_1$
 $a\ a\ b\ a\ \$\ a\ b_0$
 $a\ b\ a\ \$\ a\ b\ a$
 $a\ b\ a\ a\ b\ a\ \$$
 $b_1\ a\ \$\ a\ b\ a\ a$
 $b_0\ a\ a\ b\ a\ \$\ a$

LF mapping

- They will be in same order in L and F. Why?

- a b₁ a a b₂ a \$

\$ a b a a b a
a \$ a b a a b₁
a a b a \$ a b₀
a b a \$ a b a
a b a a b a \$
b₁ a \$ a b a a
b₀ a a b a \$ a

LF Mapping

BWM with T-ranking:

<i>F</i>	<i>L</i>
\$ a ₀ b ₀ a ₁ a ₂ b ₁ a ₃	
a ₃ \$ a ₀ b ₀ a ₁ a ₂ b ₁	
a ₁ a ₂ b ₁ a ₃ \$ a ₀ b ₀	
a ₂ b ₁ a ₃ \$ a ₀ b ₀ a ₁	
a ₀ b ₀ a ₁ a ₂ b ₁ a ₃ \$	
b ₁ a ₃ \$ a ₀ b ₀ a ₁ a ₂	
b ₀ a ₁ a ₂ b ₁ a ₃ \$ a ₀	

LF mapping

- Assign numbers to distinguish characters

- $a_0 \ b_0 \ a_1 \ a_2 \ b_1 \ a_3 \ \$$

$\$ \ a \ b \ a \ a \ b \ a_3$
 $a_3 \ \$ \ a \ b \ a \ a \ b_1$
 $a_1 \ a \ b \ a \ \$ \ a \ b_0$
 $a_2 \ b \ a \ \$ \ a \ b \ a_1$
 $a_0 \ b \ a \ a \ b \ a \ \$$
 $b_1 \ a \ \$ \ a \ b \ a \ a_2$
 $b_0 \ a \ a \ b \ a \ \$ \ a_0$

LF mapping

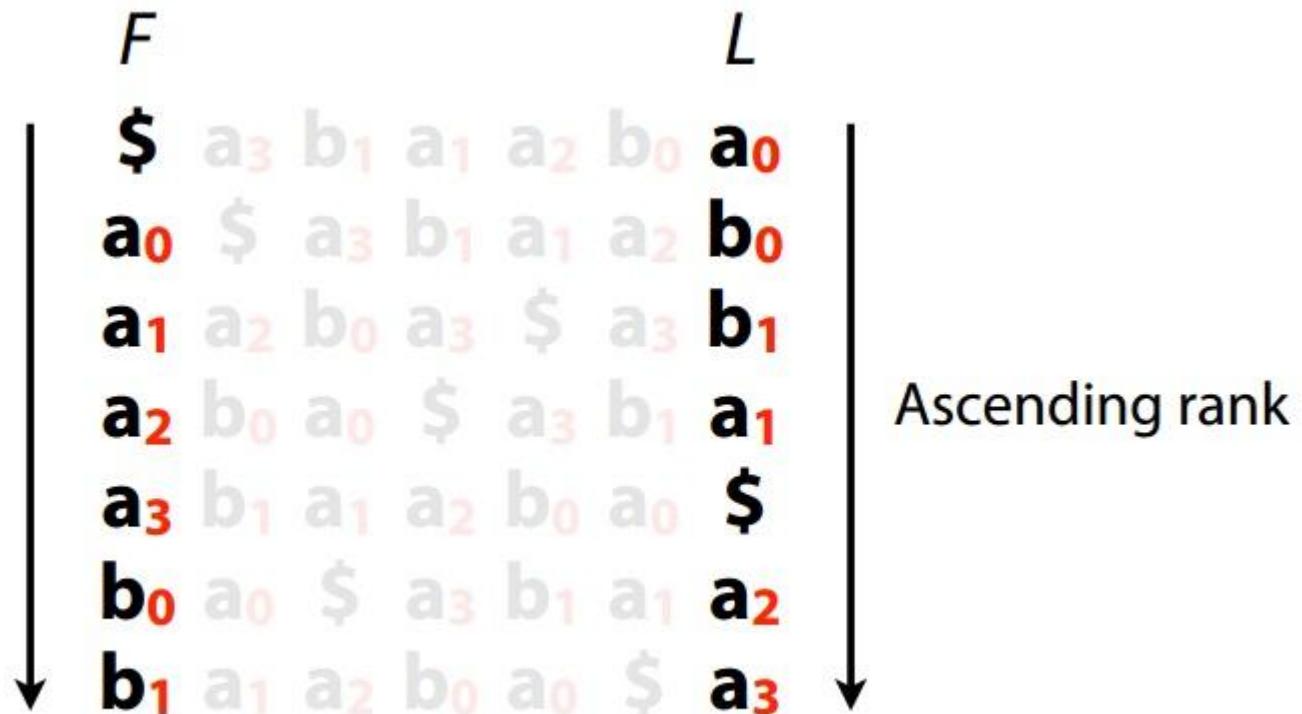


Figure from slide by Ben Langmead

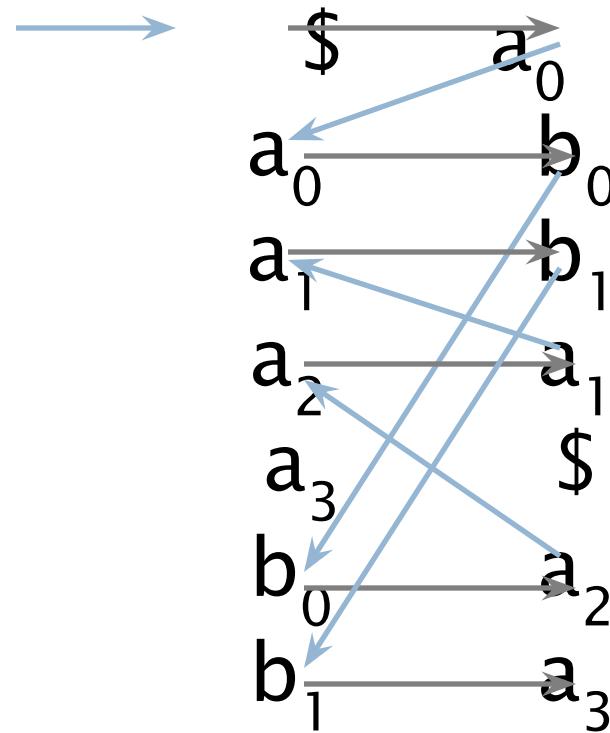
LF mapping

F	L	
\$	a_0	
a_0	b_0	
a_1	b_1	← Which BWM row <i>begins</i> with b_1 ?
a_2	a_1	Skip row starting with \$ (1 row)
a_3	\$	Skip rows starting with a (4 rows)
b_0	a_2	Skip row starting with b_0 (1 row)
row 6 → b_1	a_3	Answer: row 6

Figure from slide by Ben Langmead

BWT reversing

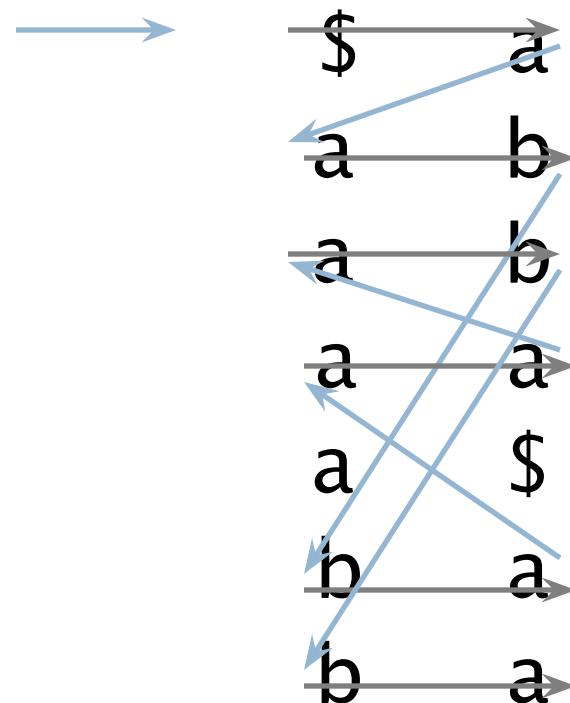
- Reverse BWT from right to left



a₃b₁a₁a₂b₀a₀\$

BWT reversing

- Reverse BWT from right to left



abaaba\$

Burrows-Wheeler Transform: reversing

Reverse BWT(T) starting at right-hand-side of T and moving left

Start in first row. F must have $\$$.

L contains character just prior to $\$$: a_0

Jump to row beginning with a_0 .

L contains character just prior to a_0 : b_0 .

Repeat for b_0 , get a_2

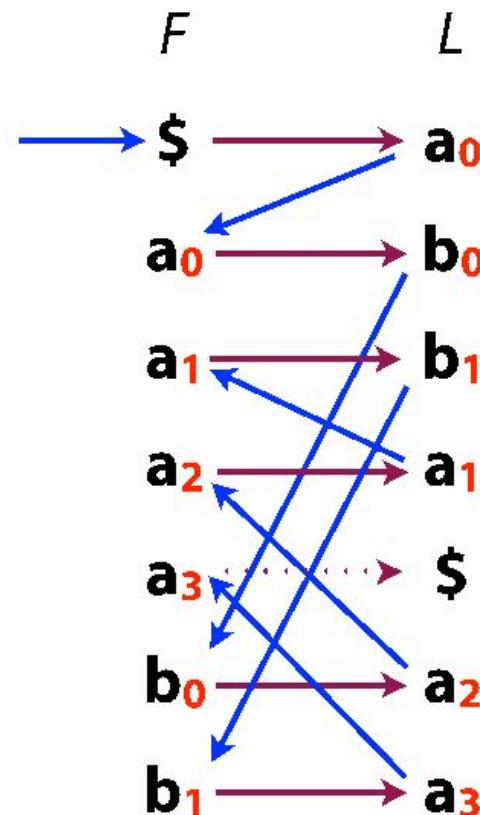
Repeat for a_2 , get a_1

Repeat for a_1 , get b_1

Repeat for b_1 , get a_3

Repeat for a_3 , get $\$$ (done)

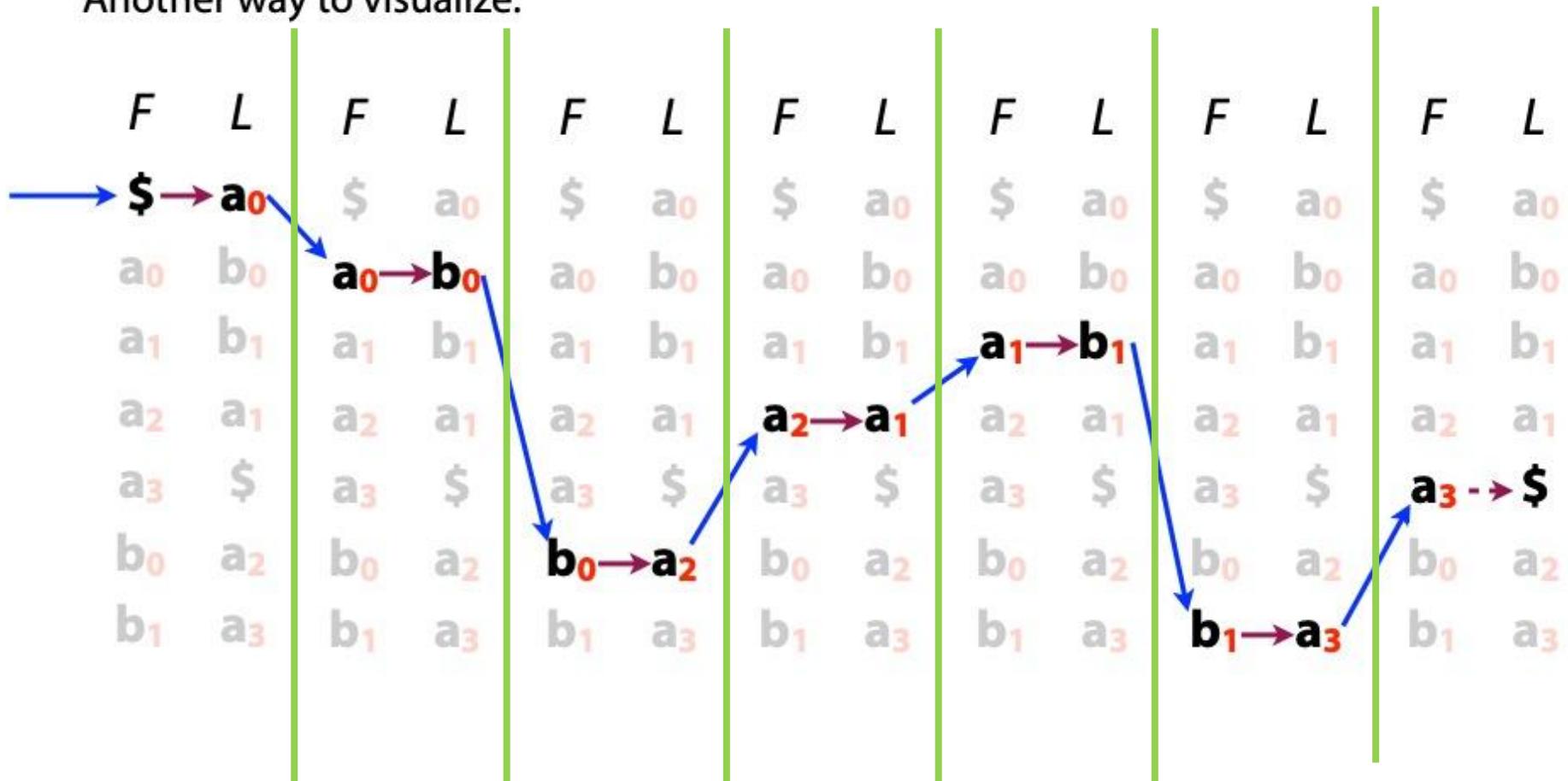
$T: a_3 \ b_1 \ a_1 \ a_2 \ b_0 \ a_0 \ \$$



In reverse order, we saw = $a_3 \ b_1 \ a_1 \ a_2 \ b_0 \ a_0 \ \$ = T$

Burrows-Wheeler Transform: reversing

Another way to visualize:



$T: a_3 b_1 a_1 a_2 b_0 a_0 \$$

FM index

- FM index
 - Combines the BWT with a few small auxiliary data structures
 - Ferragina and Manzini
- F and L from BWT
 - Both can be compressed

FM index querying

Easy to find all the rows beginning with **a**, thanks to F 's simple structure

$P = \mathbf{aba}$	F	L
	\$	a b a a b a ₃
	a ₀	\$ a b a a b ₀
	a ₁	a b a \$ a b ₁
	a ₂	b a \$ a b a ₁
	a ₃	b a a b a \$
	b ₀	a \$ a b a a ₂
	b ₁	a a b a \$ a ₀

Figure from slide by Ben Langmead

FM index querying

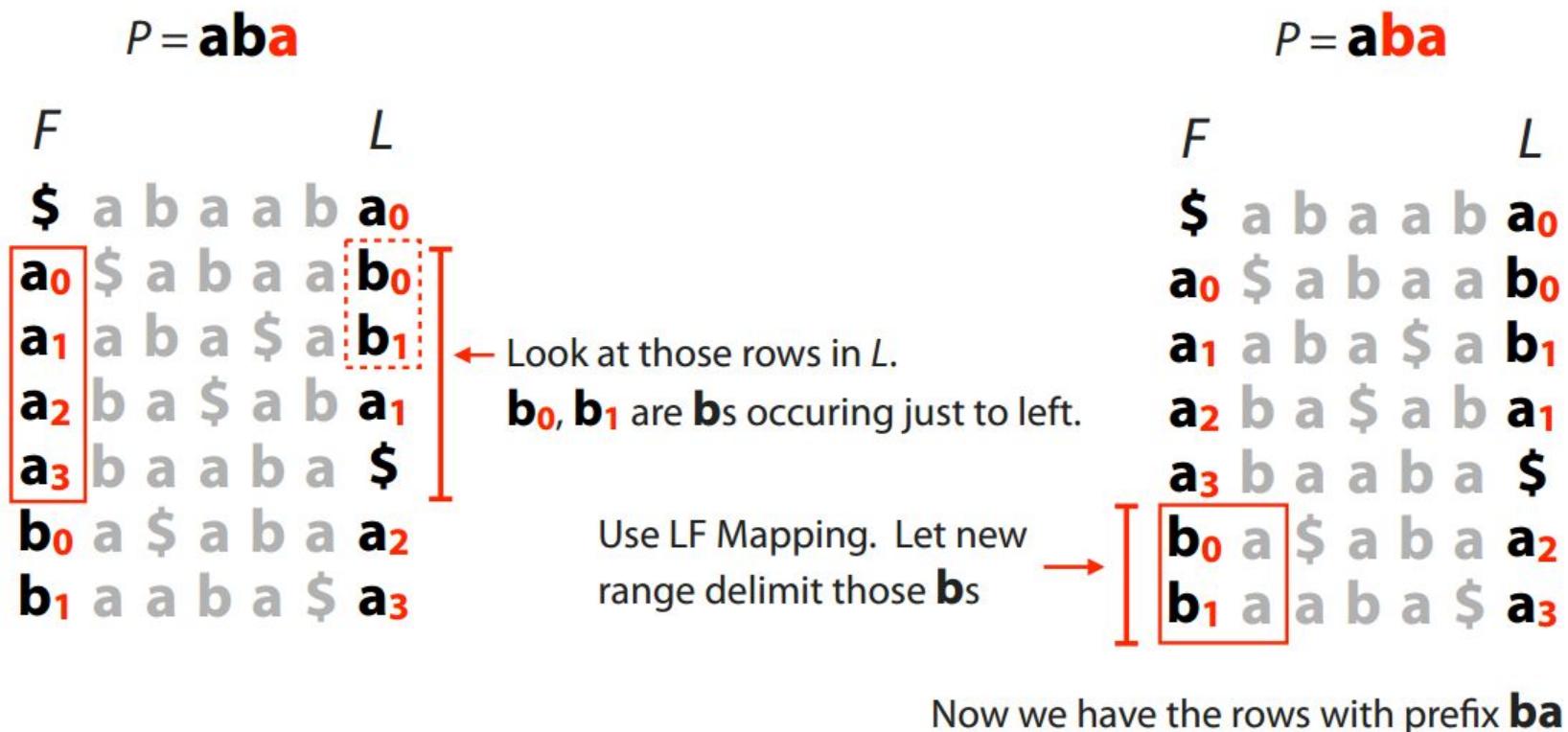


Figure from slide by Ben Langmead

FM index querying

$$P = \mathbf{aba}$$

F	L
\$ a b a a b	a_0
a_0 \$ a b a a	b_0
a_1 a b a \$ a	b_1
a_2 b a \$ a b	a_1
a_3 b a a b a	\$
b_0 a \$ a b a	a_2
b_1 a a b a \$	a_3

Use LF Mapping →

a_2, a_3 occur just to left.

$$P = \mathbf{aba}$$

F	L
\$ a b a a b	a_0
a_0 \$ a b a a	b_0
a_1 a b a \$ a	b_1
a_2 b a \$ a b	a_1
a_3 b a a b a	\$
b_0 a \$ a b a	a_2
b_1 a a b a \$	a_3

Now we have the rows with prefix **aba**

Figure from slide by Ben Langmead

ana

1 → \$₁panamabananas₁
a₁bananas\$panam₁
a₂mabananas\$pan₁
a₃namabananas\$p₁
a₄nanas\$panamab₁
6 → a₅nas\$panamaba₂
a₆s\$panamabana₃
b₁ananas\$panama₁
m₁abanananas\$pana₂
n₁amabananas\$pa₃
n₂anas\$panamaba₄
n₃as\$panamabana₅
p₁anamabananas\$₁
s₁\$panamabanana₆

FM Index: querying

When P does not occur in T , we eventually fail to find next character in L :

$$P = \mathbf{bba}$$

F	L
\$	a b a a b a ₀
a ₀	\$ a b a a b ₀
a ₁	a b a \$ a b ₁
a ₂	b a \$ a b a ₁
a ₃	b a a b a \$
b ₀	a \$ a b a a ₂
b ₁	a a b a \$ a ₃

Rows with **ba** prefix

I [b₀ a \$ a b a a₂] ← No bs!

I [b₁ a a b a \$ a₃]

FM index issues

- Where are the matches?
 - Store positions as in suffix array?

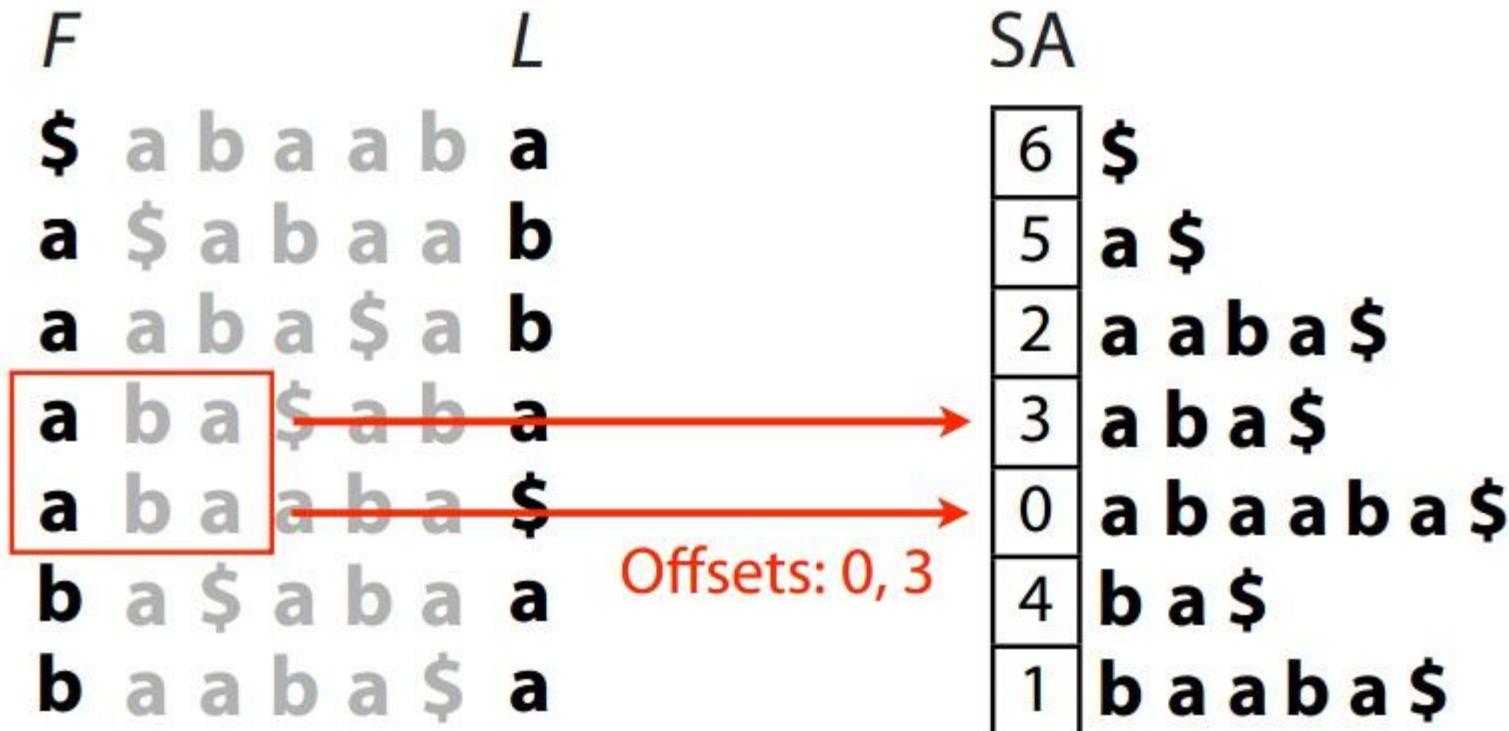


Figure from slide by Ben Langmead

FM index issues

(1) Scanning for preceding character is slow

The diagram shows a string T and its BWT representation bw . The string T is: $\$ a b a a b a_0 a_1 a_2 a_3 b_0 b_1 a a b a \$ a_3$. The BWT bw is: $\$ a b a a b_0 b_1 a_1 a_0 a_2 a_3 b_0 b_1 a_2 a_3$. A red box highlights the character a_0 in T , and a red arrow points from this character to the corresponding character b_0 in bw . The text "O(m)" is written next to the arrow, and the word "scan" is written below the arrow.

(2) Storing ranks takes too much space

```
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

A callout arrow points from the variable m in the text "integers" to the line `ranks, tots = rankBwt(bw)` in the code.

(3) Need way to find where matches occur in T :

The diagram shows the same string T and BWT bw as before. A red box highlights the character a_2 in T , and a red arrow points from this character to the corresponding character a_2 in bw . The word "Where?" is written to the left of the bw string.

FM Index: fast rank calculations

Is there an fast way to determine which **b**s precede the **a**s in our range?

<i>F</i>		<i>L</i>
\$	a b a a b	a₀
a₀	\$ a b a a	b₀
a₁	a b a \$ a	b₁
a₂	b a \$ a b	a₁
a₃	b a a b a	\$
b₀	a \$ a b a	a₂
b₁	a a b a \$	a₃

FM Index: fast rank calculations

		<i>Tally</i>	
<i>F</i>	<i>L</i>	a	b
\$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	\$	2	2
b	a	3	2
b	a	4	2

0 **b**s up to & including this row

2 **b**s up to & including this row

So **b₀** and **b₁** must be in there!

FM Index: fast rank calculations

		<i>Tally</i>	
<i>F</i>	<i>L</i>	a	b
\$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	\$	2	2
b	a	3	2
b	a	4	2

$\vdash |\Sigma| \dashv$

Tally is $m \times |\Sigma|$ integers
Too big!

FM Index: fast rank calculations

		Tally	
F	L	a	b
\$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	\$	2	2
b	a	3	2
b	a	4	2

2 a's up to & including this row

4 a's up to & including this row

So **a₂** and **a₃** must be in there!

O(1) time; 2 lookups
regardless of range size

FM index issues

- How to get numbers (ranks) for LF mapping?
 - Calculate numbers of each character up to that row

Idea: pre-calculate # **a**s, **b**s in L up to every row:

Tally

F	L	a	b
\$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	\$	2	2
b	a	3	2
b	a	4	2

We infer **b_0** and **b_1** appear in L in this range

$O(1)$ time, but requires $m \times |\Sigma|$ integers

Figure from slide by Ben Langmead

FM Index: fast rank calculations

Next idea: pre-calculate # **a**s, **b**s in L up to some rows, e.g. every 5th row. Call pre-calculated rows *checkpoints*.

Tally

F	L	a	b
\$	a	1	0
a	b		
a	b		
a	a		
a	\$		
b	a	3	2
b	a		

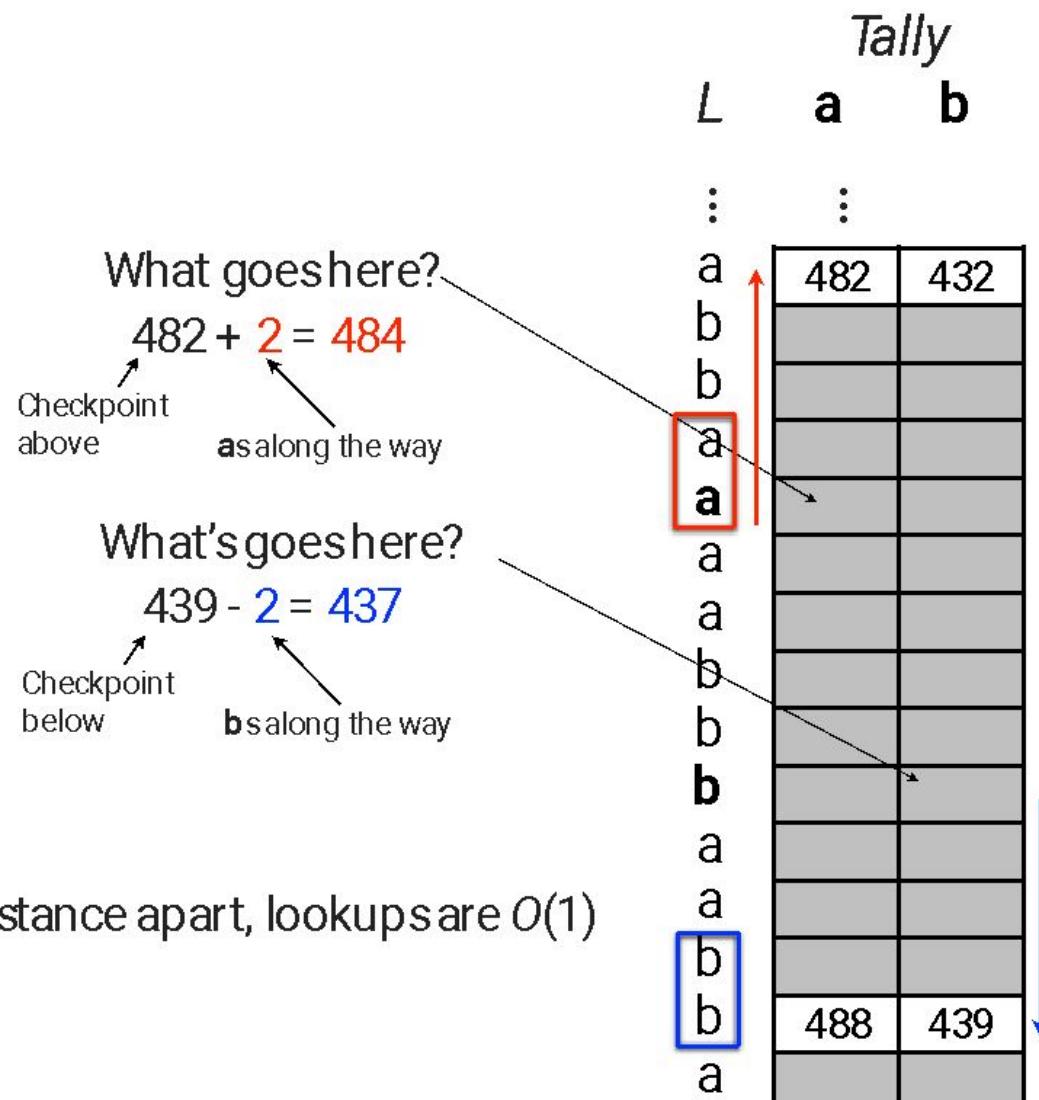
← Lookup here succeeds as usual

← Oops: not a checkpoint

← But there's one nearby

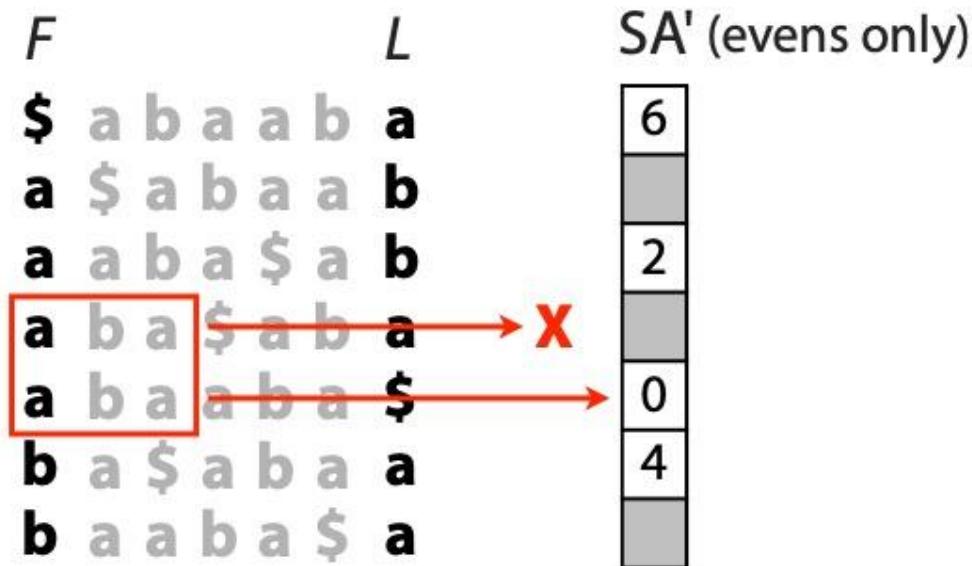
To resolve a lookup for a non-checkpoint row, walk to nearest checkpoint. Use tally at that checkpoint, *adjusted for characters we saw along the way*.

FM Index: fast rank calculations



FM Index: resolving offsets

Idea: store some suffix array elements, but not all

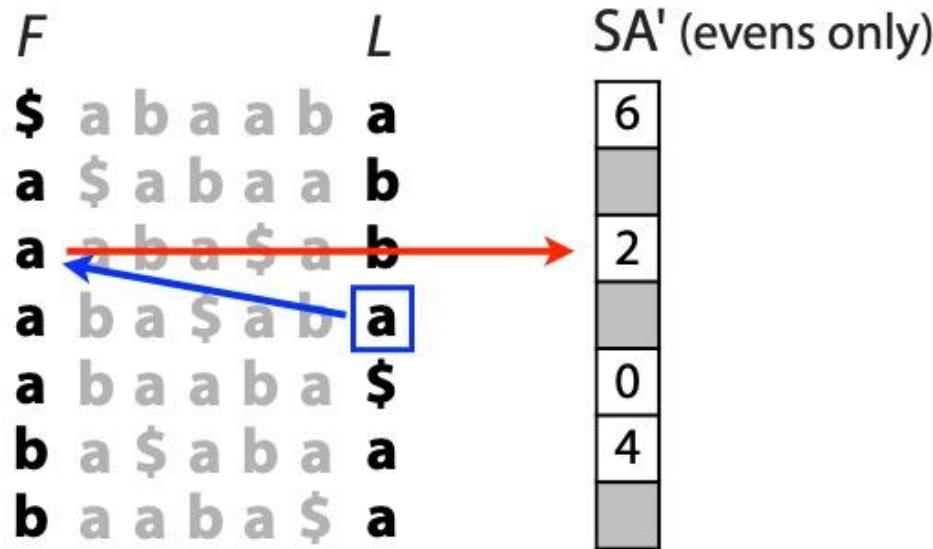


Lookup for row 4 succeeds

Lookup for row 3 fails - SA entry was discarded

FM Index: resolving offsets

LF Mapping tells us that “a” at the end of row 3 corresponds to...
... “a” at the beginning of row 2



Row 2 of suffix array = 2

Missing value in row 3 = 2 (row 2's SA val) + 1 (# steps to row 2) = 3

If saved SA values are O(1) positions apart in *T*, resolving offset is O(1) time

FM index

- FM index for human genome can be as small as 1.5 GB
 - With some space-time tradeoff

FM Index

Components of FM Index:

First column (F): $\sim |\Sigma|$ integers

Last column (L): m characters

SA sample: $m \cdot a$ integers, a is fraction of SA elements kept

Checkpoints: $m \cdot |\Sigma| \cdot b$ integers, b is fraction of tallies kept

For DNA alphabet (2 bits / nt), T = human genome, $a = 1/32$, $b = 1/128$:

First column (F): 16 bytes

Last column (L): 2 bits * 3 billion chars = 750 MB

SA sample: 3 billion chars * 4 bytes / 32 = ~ 400 MB

Checkpoints: 3 billion * 4 alphabet chars * 4 bytes / 128 = ~ 400 MB

Total ≈ 1.5 GB

~0.5 bytes per input char

(blue indicates what we can
adjust by changing a & b)

Short read mapping tools

- Hash based
 - MAQ
 - SOAP
- BWT based
 - Bowtie, Bowtie2
 - SOAP2
 - BWA

References

- **Suffix Tree**
 - Bioinformatics Algorithms (Part II) - Chapter 9
 - Part I of Algorithms on Strings, Trees and Sequences by Dan Gusfield
- **BWT, FM index**
 - http://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf
- **Special Thanks to**
 - Atif sir, Saifur sir