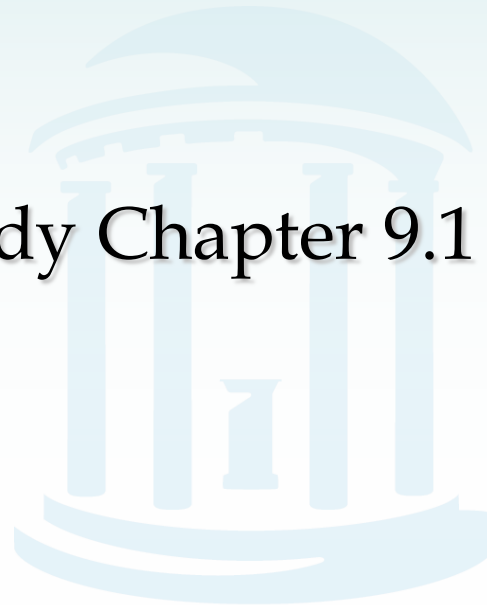




Lecture 16: Combinatorial Pattern Matching

Study Chapter 9.1 – 9.5



Repeat Finding



- Example of repeats:
 - ATGGTCTAGGTCCTAGTGGTC
- Motivation to find them:
 - Phenotypes arise from copy-number variations
 - Genomic rearrangements are often associated with repeats
 - Trace evolutionary secrets
 - Some tumors are characterized by an explosion of repeats



Repeat Finding



- Near matches are more difficult:
 - ATGGTCTAGGACCTAGTGTTTC
- Motivation to find them:
 - same as before
 - near matches are more realistic due to mutations



ℓ -mer Repeats



- Short repeats are easy to find
- Long repeats are difficult to find
- Strategy for finding long repeats:
 - Find exact repeats of short ℓ -mers (ℓ is usually 10 to 13)
 - Extend ℓ -mer repeated seeds 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



GCTTACAGATTTCAGTCTTACAGATGGT

- Extend these 4-mer matches:

GCTTACAGATTTCAGTCTTACAGATGGT

- Maximal repeat: CTTACAGAT
- Maximal repeats cannot be extended in either direction
- 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: What is it?



- How hashing works...
 - Generate an integer “key” from an arbitrary record
 - Store record in an data structure indexed by this integer key
- Hashing is a very efficient way to store and retrieve data
 - e.g., Python directories are hashes



Hashing: Definitions



- Hash table: array used in hashing
- Records: data stored in a *hash table*
- Keys: identify sets of *records*
- Hash function: uses a *key* to generate an index to insert at in *hash table*
- Collision: when more than one record is mapped to the same index in the hash table



Hashing: Example

- Where do the animals eat?
- Records: each animal
- Keys: where each animal eats



Records x	Keys $h(x)$
Penguin	1
Octopus	4
Turtle	3
Mouse	2
Snake	3
Heron	1
Tiger	2
Iguana	3
Ape	2
Cricket	4
Sparrow	1

Hashing: Maximal Repeats



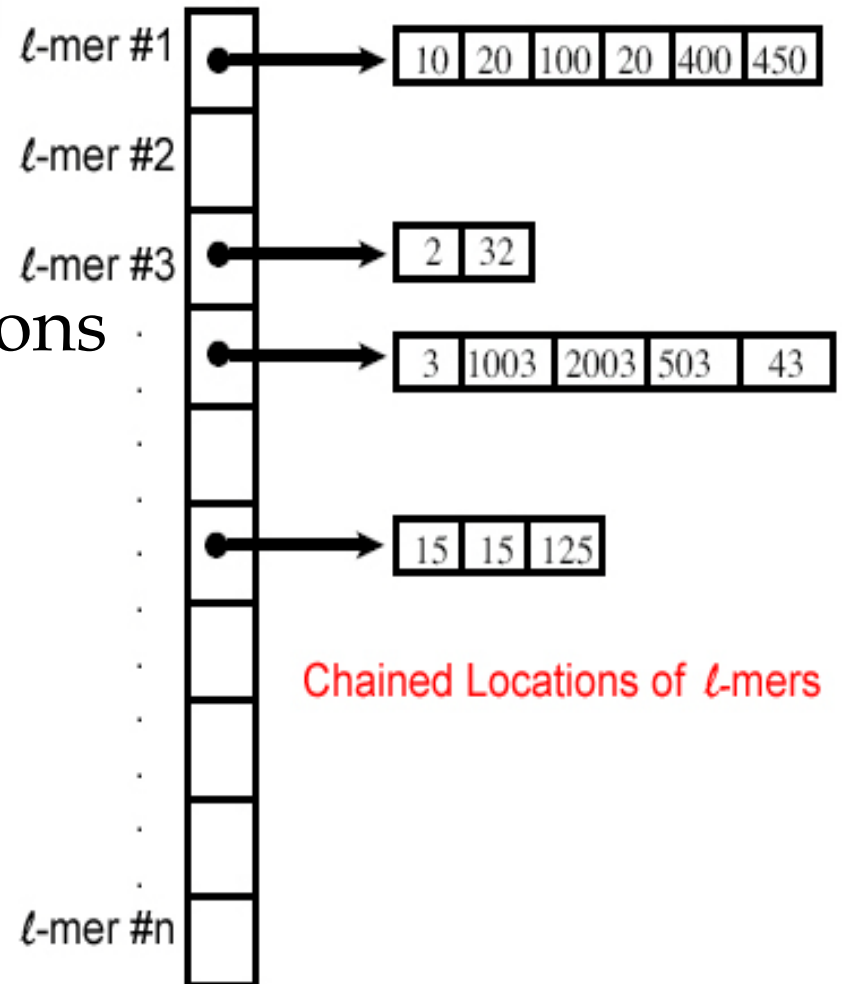
- To find repeats in a genome:
 - For all l -mers in the genome, note its starting position and the sequence
 - Generate a hash table index for each unique l -mer sequence
 - In each index of the hash table, store all genome start locations of the l -mer which generated that index
 - Extend l -mer repeats to maximal repeats
- Problem as l gets big the number of possible patterns becomes larger than the genome's length ($4^l \gg n$)



Hashing: Collisions



- Generate hash keys from a reduced space
Ex. Key = $\text{quaternary}(\text{seq}) \% (N/l)$
- Leads to possible collisions
- Dealing with collisions:
 - “Chain” tuples of (l -mer, start location) pairs in a 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 positions of a particular sequence in a given large sequence?
- 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 $p = p_1 \dots p_n$ and text $t = t_1 \dots t_m$
- Output: All positions $1 \leq i \leq (m - n + 1)$ such that the n -letter substring of t starting at i matches p
- **Motivation**: Searching database for a known pattern



Exact Pattern Matching: A Brute-Force Algorithm



PatternMatching(p,t)

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



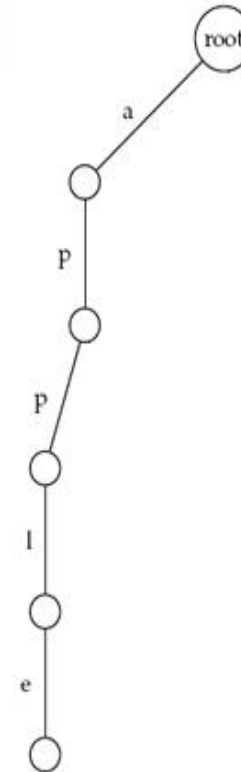
- *PatternMatching* runtime: $O(nm)$
- Probability-wise, it's more like $O(m)$
 - Rarely will there be close to n comparisons in line 4
- Worse case:
Find “AAAAT” in “AAAAAAAAAAAAAAAAAAAT”
- Better solution: **suffix trees**
 - Can solve problem in $O(m)$ time
 - Conceptually related to **keyword trees**



Keyword Trees: Example



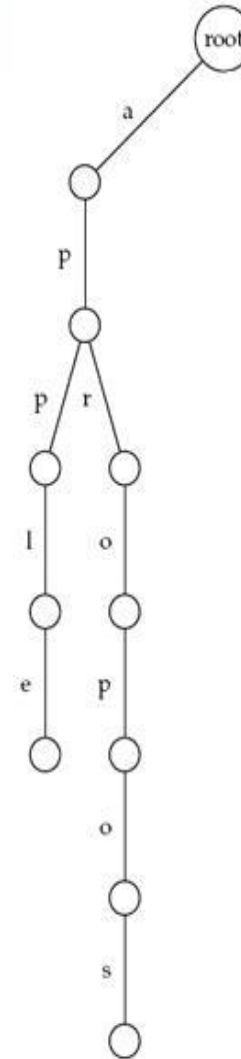
- *Keyword tree:*
 - apple



Keyword Trees: Example (cont'd)



- *Keyword tree:*
 - apple
 - apropos

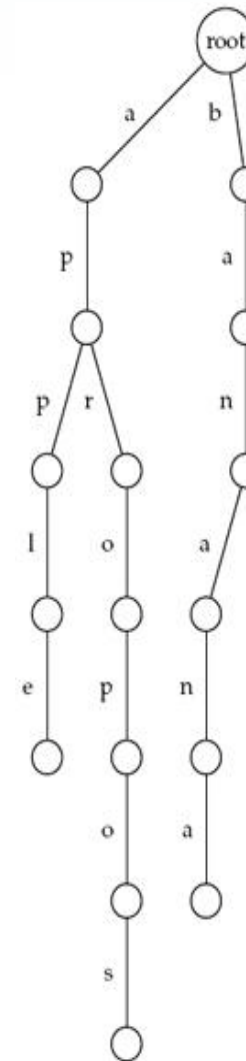


Keyword Trees: Example (cont'd)



- *Keyword tree:*

- apple
- apropos
- banana

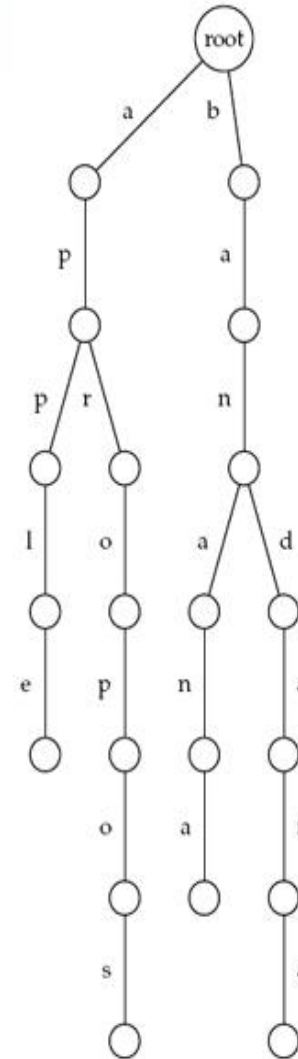


Keyword Trees: Example (cont'd)



- *Keyword tree:*

- apple
- apropos
- banana
- bandana

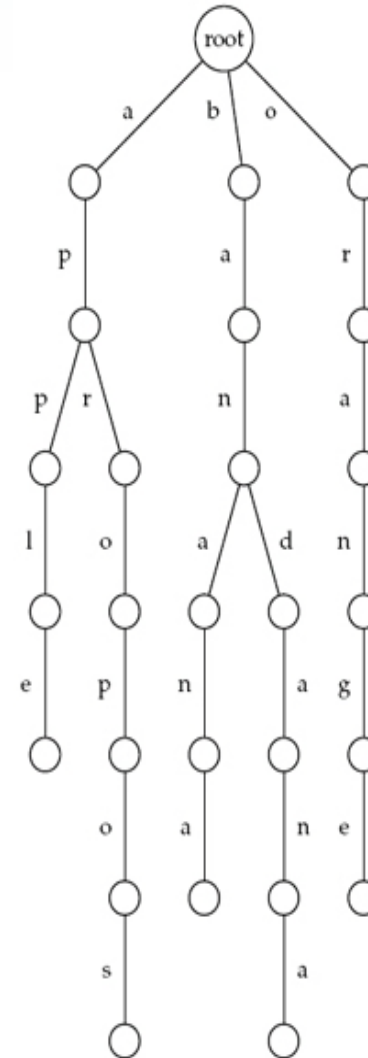


Keyword Trees: Example (cont'd)



- *Keyword tree:*

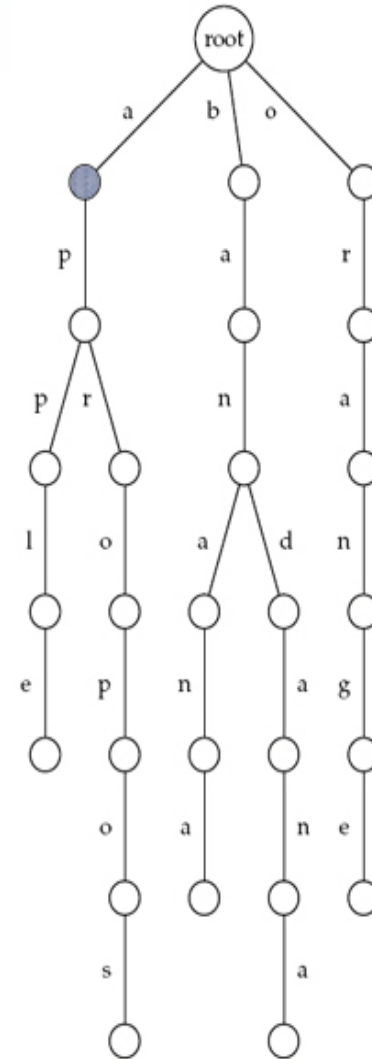
- apple
- apropos
- banana
- bandana
- orange



Keyword Trees: Threading (cont'd)



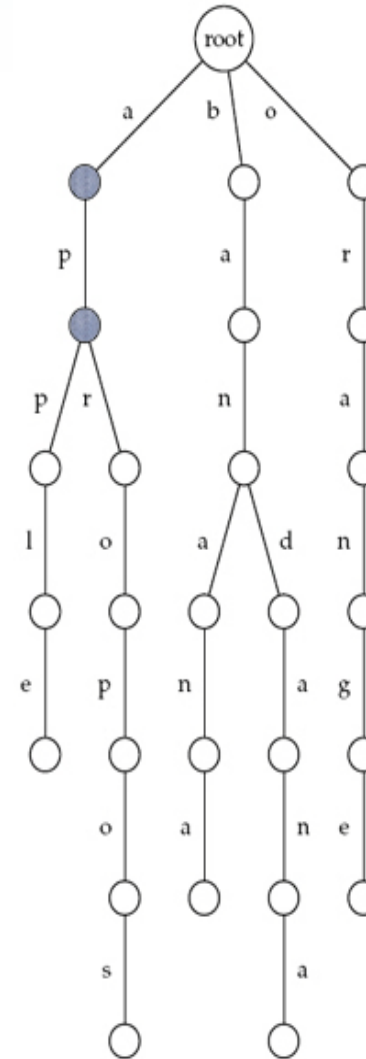
- Thread “appeal”
 - appeal



Keyword Trees: Threading (cont'd)



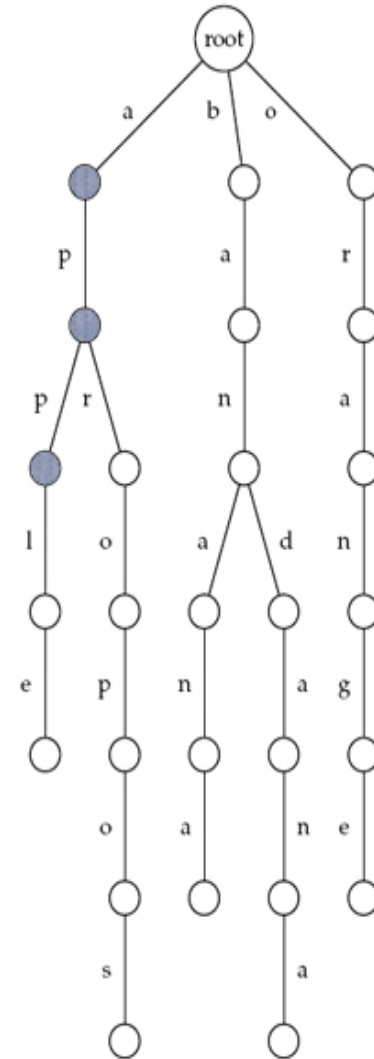
- Thread “appeal”
 - appeal



Keyword Trees: Threading (cont'd)



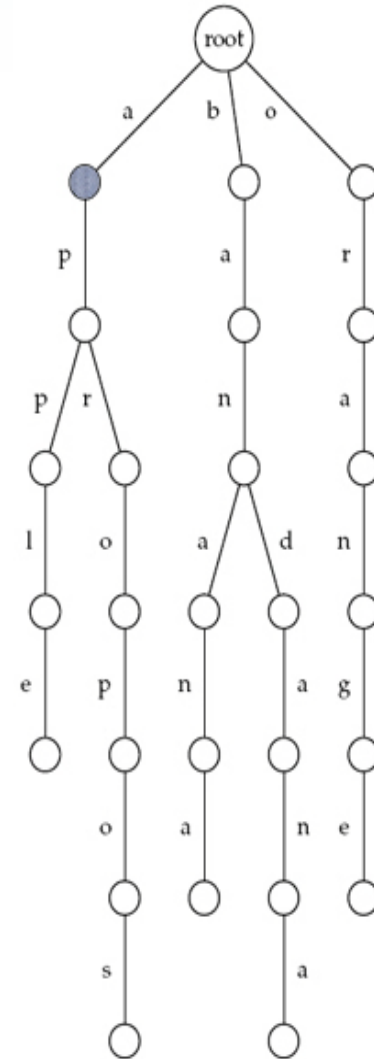
- Thread “appeal”
 - appeal



Keyword Trees: Threading (cont'd)



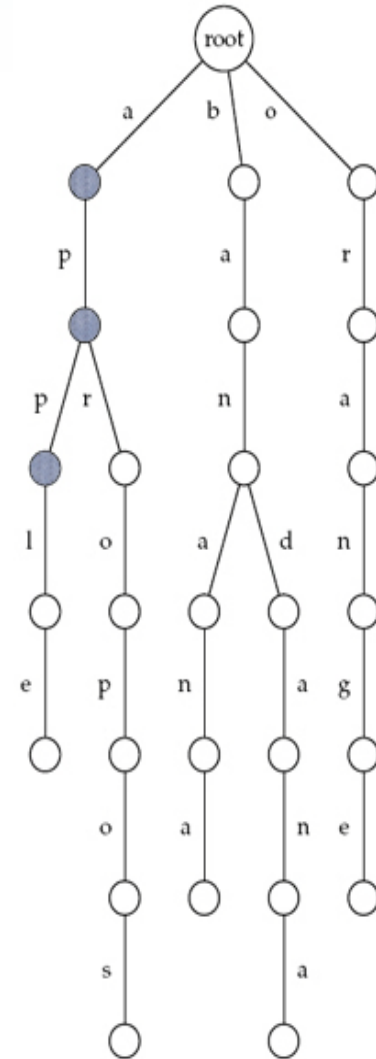
- Thread “apple”
 - apple



Keyword Trees: Threading (cont'd)



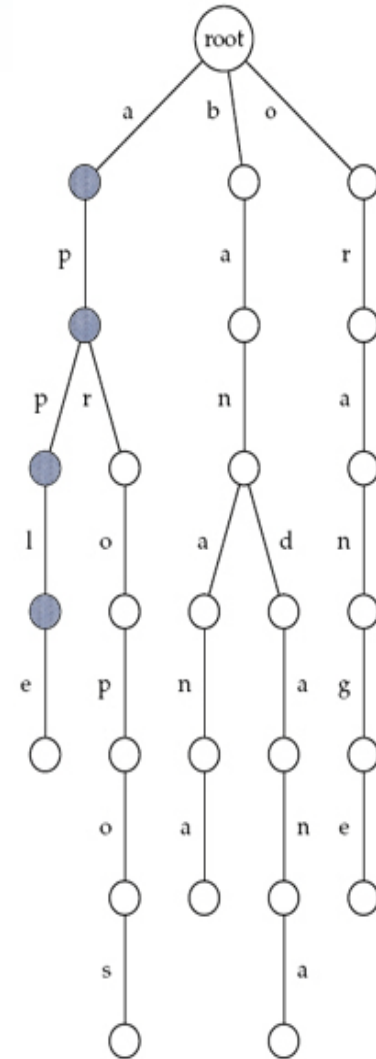
- Thread “apple”
 - apple



Keyword Trees: Threading (cont'd)



- Thread “apple”
 - apple



Multiple Pattern Matching Problem



- Goal: Given *a set of patterns* and a text, find all occurrences of any of patterns in the text
- Input: k patterns $\mathbf{p}^1, \dots, \mathbf{p}^k$, and text $\mathbf{t} = t_1 \dots t_m$
- 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:
 - First build keyword tree in $O(N)$ time; N is total length of all patterns
 - Now thread word starting at each position in the text to determine match with any pattern
 - With naive threading: $O(N + nm)$
 - Aho-Corasick algorithm: $O(N + m)$

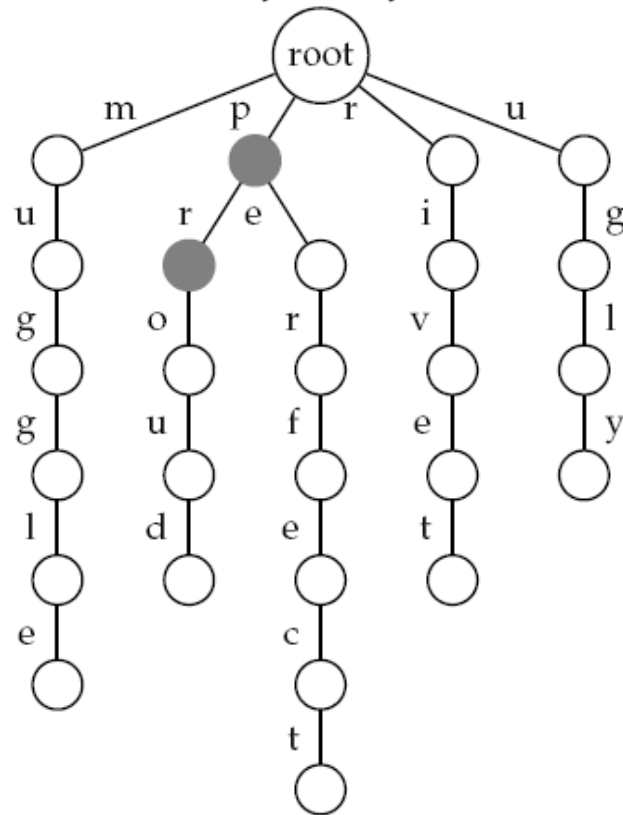


Keyword Trees: Naïve Threading



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

t = “mr and mrs dursley of number 4 privet drive were proud to say that they were perfectly normal thank you very much”

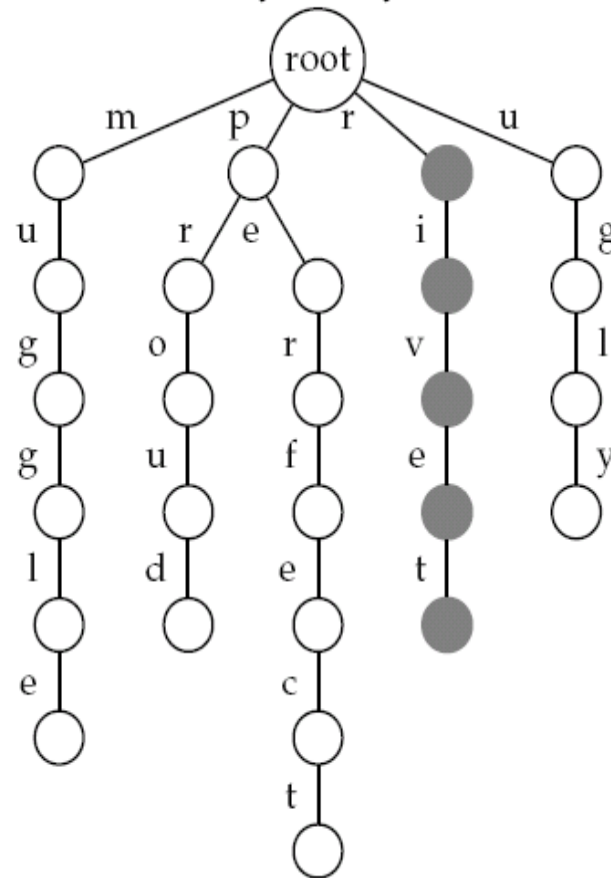


Keyword Trees: Naïve Threading



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

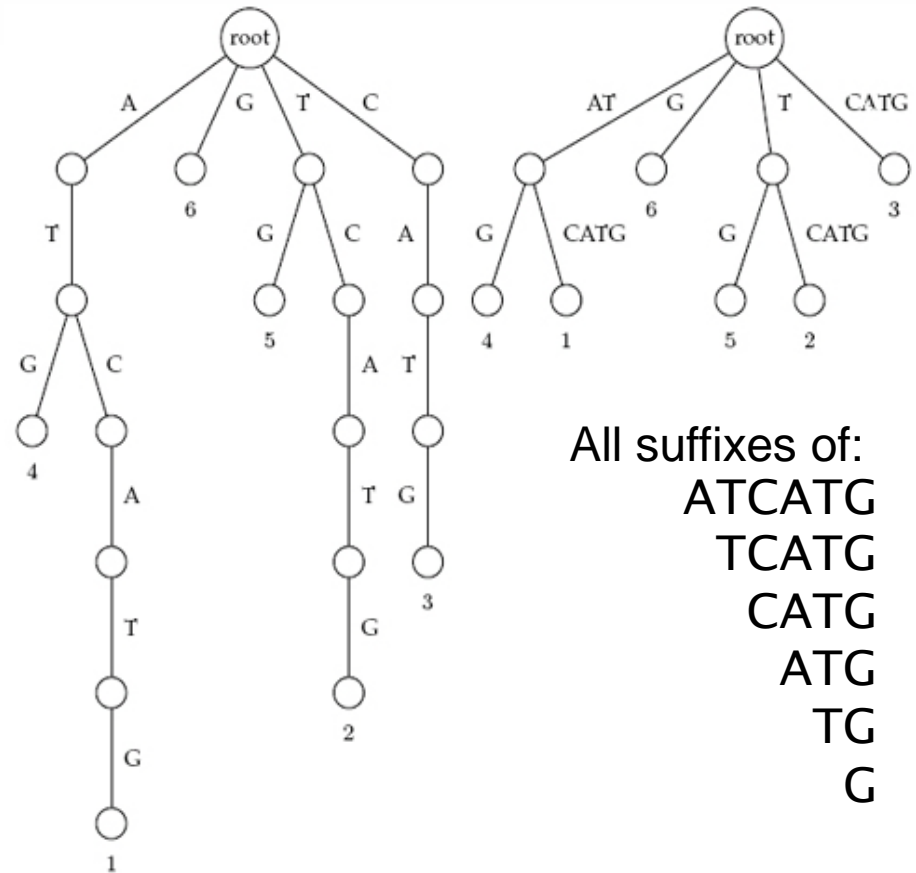
t = “mr and mrs dursley of number 4 p rivet drive were proud to say that they were perfectly normal thank you very much”



Suffix Trees=Collapsed Keyword Trees



- All suffixes of a given sequence
- Similar to keyword trees, except vertices of degree 2 are compressed
 - each edge is labeled with a *substring* of a text
 - each non-leaf has two or more children
 - leaves, are labeled by the index of the pattern.



All suffixes of:
 ATCATG
 TCATG
 CATG
 ATG
 TG
 G

(a) Keyword tree

(b) Suffix tree



Suffix Tree of a Text



- Construct a keyword tree from all suffixes of a text
- Collapse non-branching paths into an edge (path compression)

ATCATG
TCATG
CATG
ATG
TG
G



How much time does it take?

Time is linear in the total size of all suffixes, which is quadratic in the length of the text

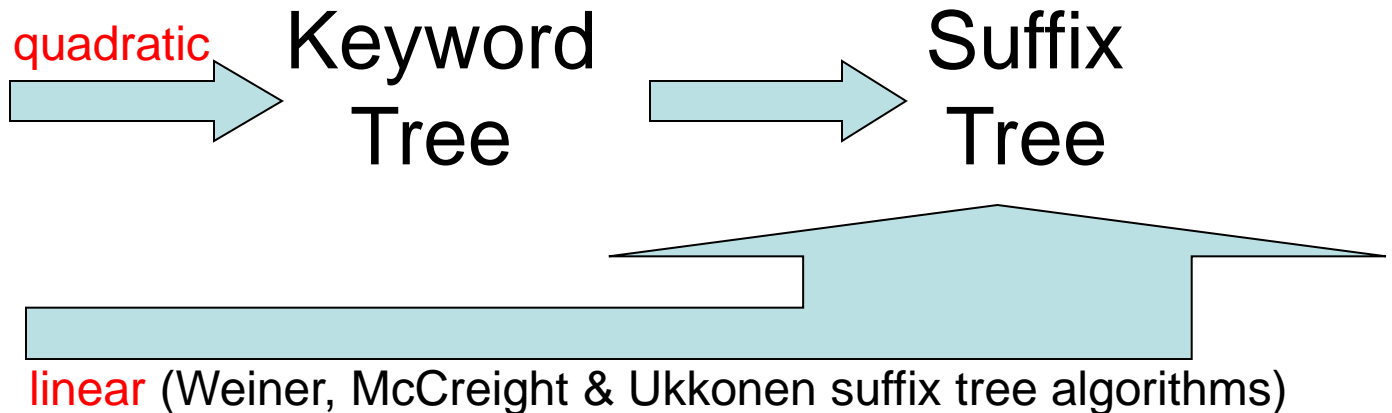


Suffix Trees: Advantages



- With careful bookkeeping a text's suffix tree can be constructed in a single pass of the text
- Thus, suffix trees can be built faster than keyword trees of suffixes and transforming them

ATCATG
TCATG
CATG
ATG
TG
G



Suffix Tree Construction



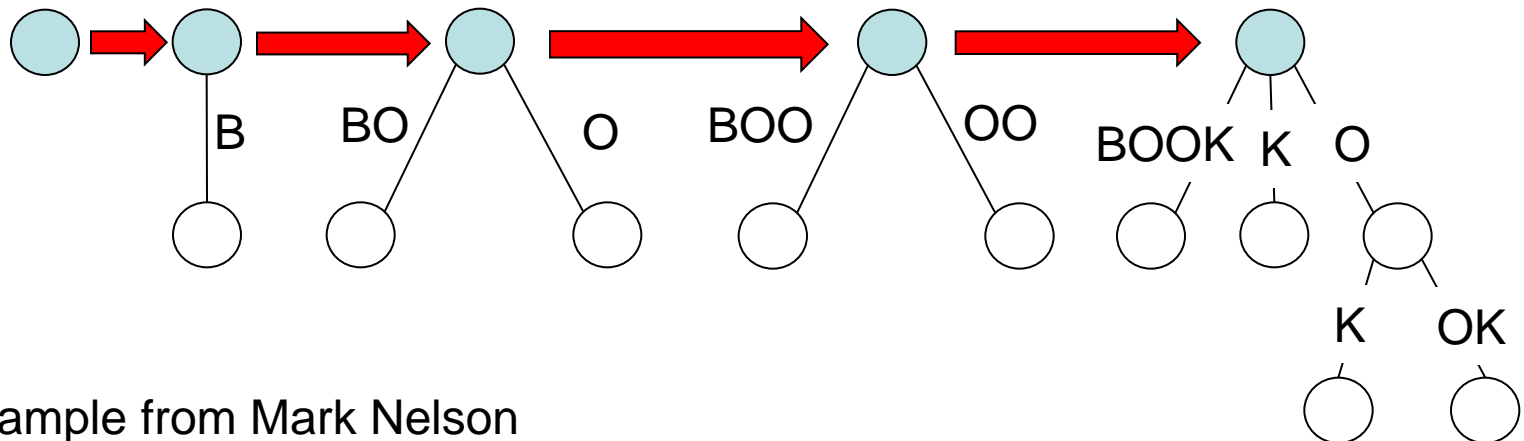
- Few books, including ours, delve into the details of suffix tree construction algorithms due to its reputation for being overly complicated.
- Weiner's and McCreight's original linear algorithms for constructing a suffix trees had some disadvantages.
- Principal among them was the requirement that the tree be built in reverse order, meaning that the tree was grown incrementally by adding characters from the end of the input.
- This ruled it out for on-line processing



Ukkonen's Clever Bookkeeping



- Esko Ukkonen's construction works from left to right.
- It's incremental. Each step transforms the Suffix Tree of the prefix ending at the i^{th} character to the Suffix Tree ending at the $i+1^{\text{th}}$.



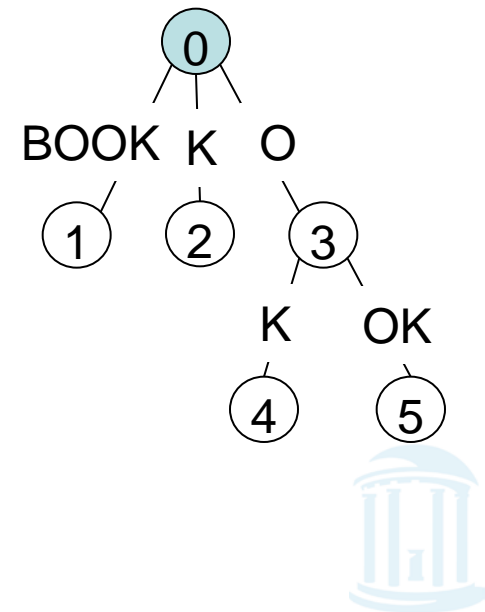
Example from Mark Nelson
[Dr. Dobb's Journal](#), August, 1996



Tree Properties



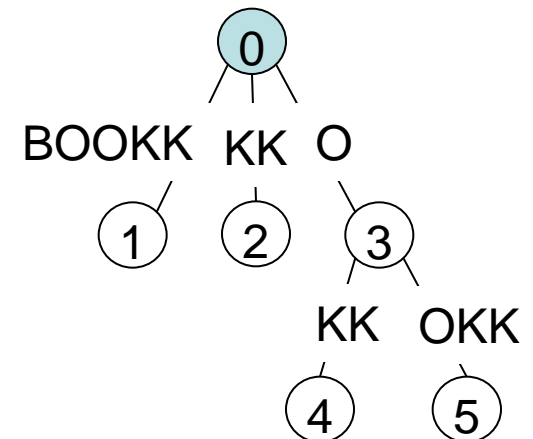
- Extensions are done by threading each new prefix through the tree and visiting each of the suffixes of the current tree.
- At each step we start at the longest suffix (BOOK), and work our way down to the shortest (empty string)
- Each ends at a node of three types:
 - A leaf node (1,2,4,5)
 - An explicit node (0, 3)
 - An implicit node (Between characters of a substring labeling an edge, such as BO, BOO, and OO).



Observations



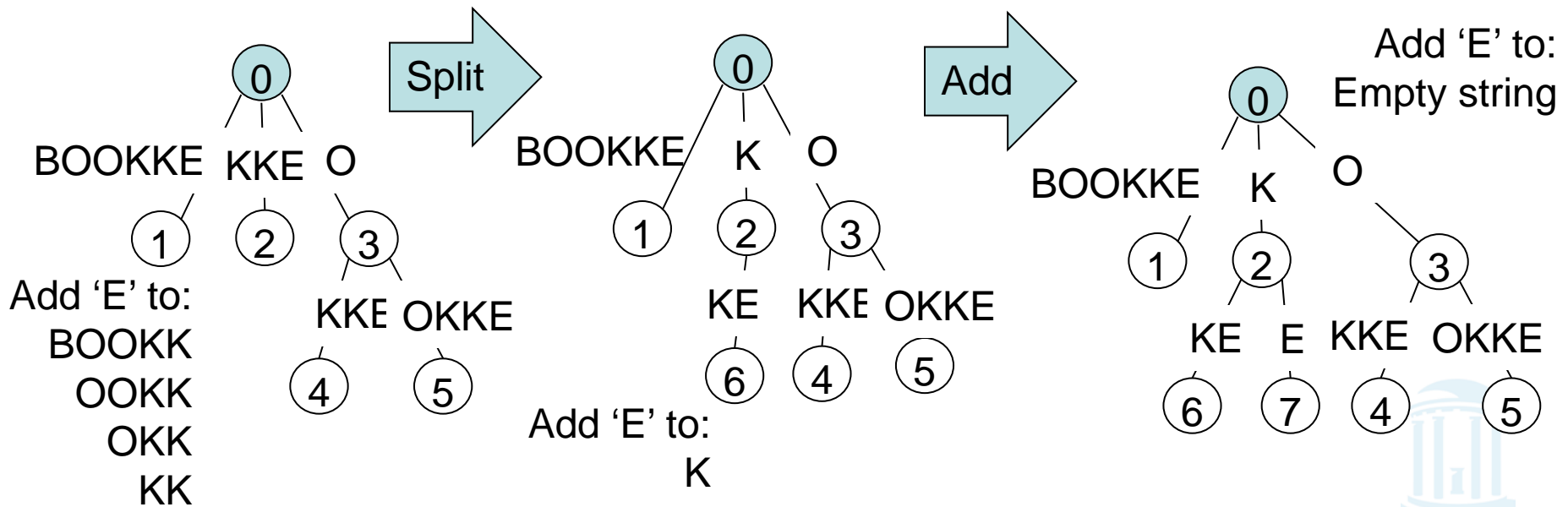
- There are 5 suffixes in the tree (including the empty string) after adding BOOK
- They are represented by the root and 4 leaves
- Adding the next letter, another 'K', requires visiting each of the suffixes in the existing tree, in order of decreasing length, and adding letter 'K' to its end.
- Adding a character to a leaf node never creates a new explicit node, regardless of the letter
- If the root already has an edge labeled 'K' we just extend it



Split and Add Update



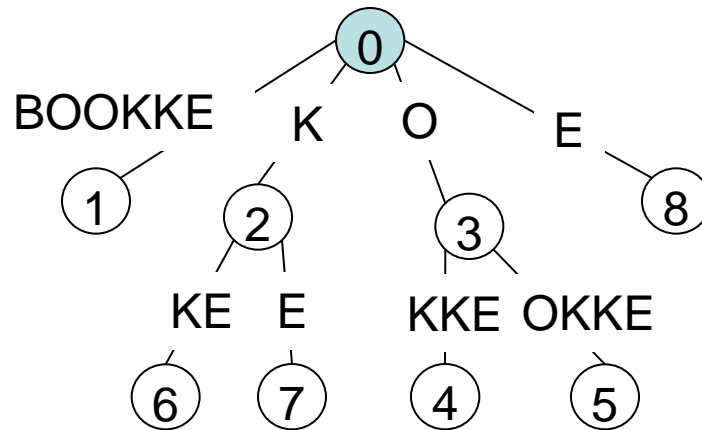
- The next step is to add an 'E' to our tree
- As before, add 'E' to each suffix in order of decreasing lengths BOOKK, OOKK, OKK, KK, K
- The first suffix that does not terminate at a leaf is called the "active point" of the suffix tree



Updating an Explicit Node



- After updating suffix K, we still have to update the next shorter suffix, which is the empty string.



Generalizing



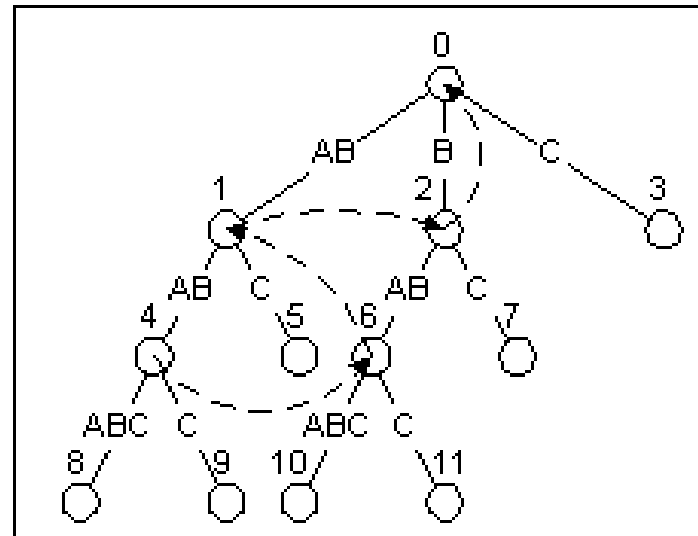
- Once a leaf node, always a leaf node
- Additional characters only extends the edge leading to the leaf (leaves are easy)
- When adding a new leaf, its edge will represent all characters from the i^{th} suffix's starting point to the $i+1^{\text{st}}$ text's end. Because of this once a leaf is created, we can just forget about it. If the edge is later split, its start may change but it will extend to the end.
- This means that we only need to keep track of the active point in each tree, and update from there.



One Last Detail



- The algorithm sketch so far glosses over one detail. At each step of an update we need to keep track of the next smaller suffix from the i^{th} update
- To do this a suffix pointer is kept at each internal node
- For Pseudo code
 - Mark Nelson, “Fast String Searching with Suffix Trees”
[Dr. Dobb's Journal](#) August, 1996
- For proofs of linear space/time performance
 - E. Ukkonen. [“On-line construction of suffix trees.”](#) Algorithmica, 14(3):249-260, September 1995.



The suffix tree for ABABABC with suffix pointers shown as dashed lines



Use of Suffix Trees



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



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 traverse all paths from the threading's endpoint to leaves and **output** their positions
- 5 **else**
- 6 **output** "Pattern does not appear in text"

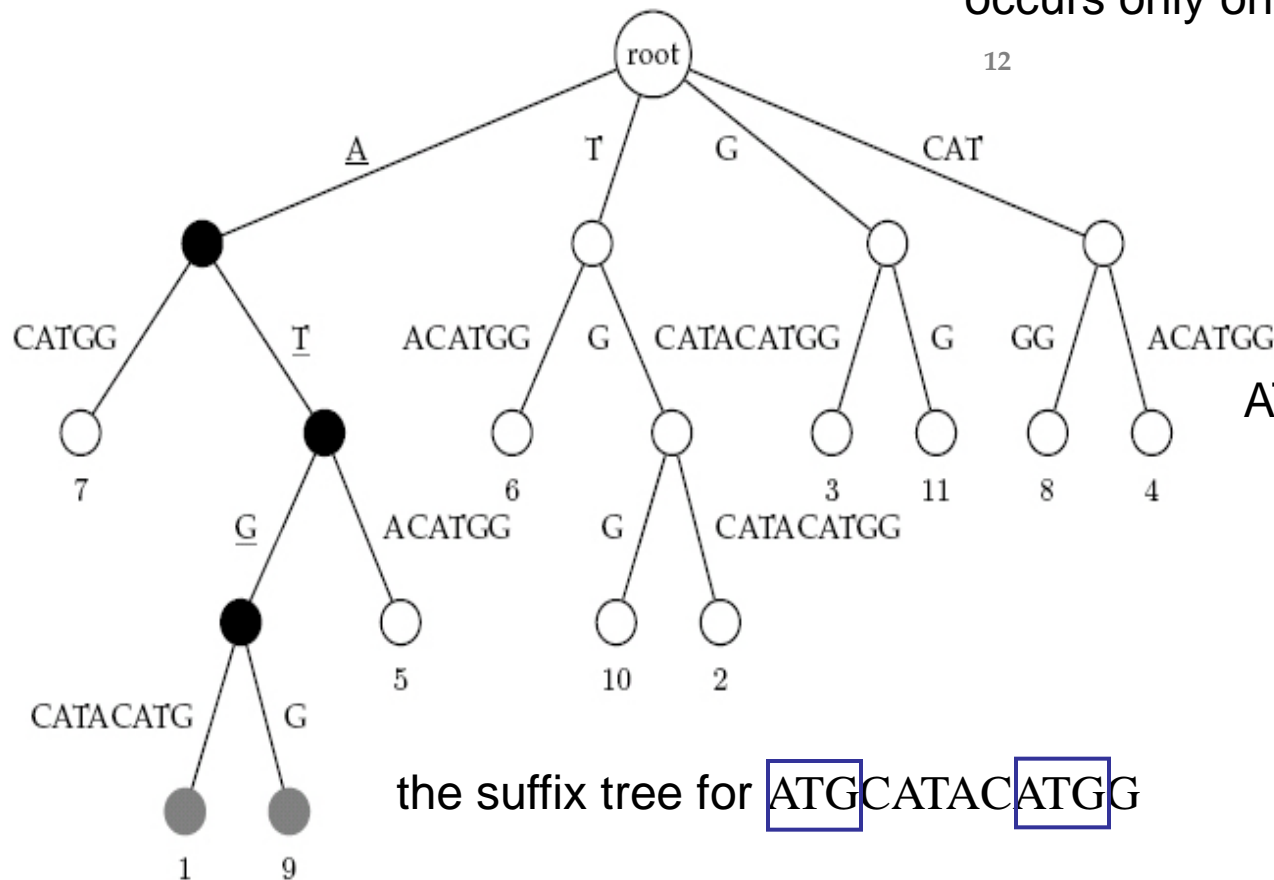


Suffix Trees: Example



Threading the pattern **ATG**

What letter occurs most frequently?
 What is the shortest pattern that occurs only once in the string?



the suffix tree for **ATG**CATAC**ATGG**

ATGCATACATGG	1
TGCATACATGG	2
GCATACATGG	3
CATACATGG	4
ATACATGG	5
TACATGG	6
ACATGG	7
CATGG	8
ATGG	9
TGG	10
GG	11
G	12

Multiple Pattern Matching: Summary



- Keyword and suffix trees are useful data structures supporting various pattern finding problems
- *Keyword trees*:
 - Build keyword tree of **patterns**, and ***thread text*** through it
- *Suffix trees*:
 - Build suffix tree of **text**, and ***thread patterns*** through it



Suffix Trees: Theory vs. Practice



- In concept, suffix trees are extremely powerful for making a variety of queries concerning a sequence
 - What is the shortest unique substring?
 - How many times does a given string appear in a text?
- Despite the existence of linear-time construction algorithms, and $O(m)$ search times, suffix trees are still rarely used for genome scale searching
 - Large storage overhead (60+ GB for human genome)
- However, close cousins of the Suffix-Tree (Suffix Arrays and Burrows-Wheeler Transforms) are more common
- Next lecture

