FIT2004: Algorithms and Data Structures

Week 10: Retrieval Data Structures for Strings

Outline

Divide and conquer (W 1-3)	Greedy algorithms (W 4-5)	Dynamic programming (W 6-7)	Network flow (W 8-9)	Data structures (W 10-11)

- Last Lecture: Circulation with Demands, Applications of Network Flow
  - Maximum Bipartite Matching
  - Circulation with Demands and Lower Bounds
  - Applications of Network Flow

- Today's Lecture: Retrieval Data Structures for Strings
  - Trie
  - Suffix Trie
  - Suffix Array

Introduction

Suppose you have a large text containing N strings. You want to pre-process it such that searching on this text is efficient.

Sorting based approach:

- Pre-processing: Sort the strings
- Searching: Binary search to find

Let M be length of strings (M can be quite large, e.g., for DNA sequences). Comparison between two strings takes O(M).

Time complexity:

Pre-processing → O(MN log N) using merge sort or O(MN) using radix sort

Searching → O(M log N)

Can we do better?

Yes! ReTrieval data structures allow answering different string queries efficiently

Outline

1. Trie
   - A. Construction
   - B. Query Processing
2. Suffix Trie
   - A. Construction
   - B. Query Processing
   - C. Suffix Tree
3. Suffix Array
   - A. Introduction
   - B. Reducing Construction Cost

Trie

- ReTRIEval tree = Trie
- Often pronounced as ‘Try'.
- Trie is an Σ-way (or multi-way) tree, where Σ is the size of the alphabet
  - E.g., Σ=2 for binary
  - Σ = 26 for English letters
  - Σ = 4 for DNA
  - Alphabet size may not always be constant, but could be part of the input to the problem! Unless otherwise specified we are focusing on alphabets of constant size in the slides.
- In a standard Trie, all words with the shared prefix fall within the same subtree/subtrie
- In fact, it is the shortest possible tree that can be constructed such that all prefixes fall within the same subtree.

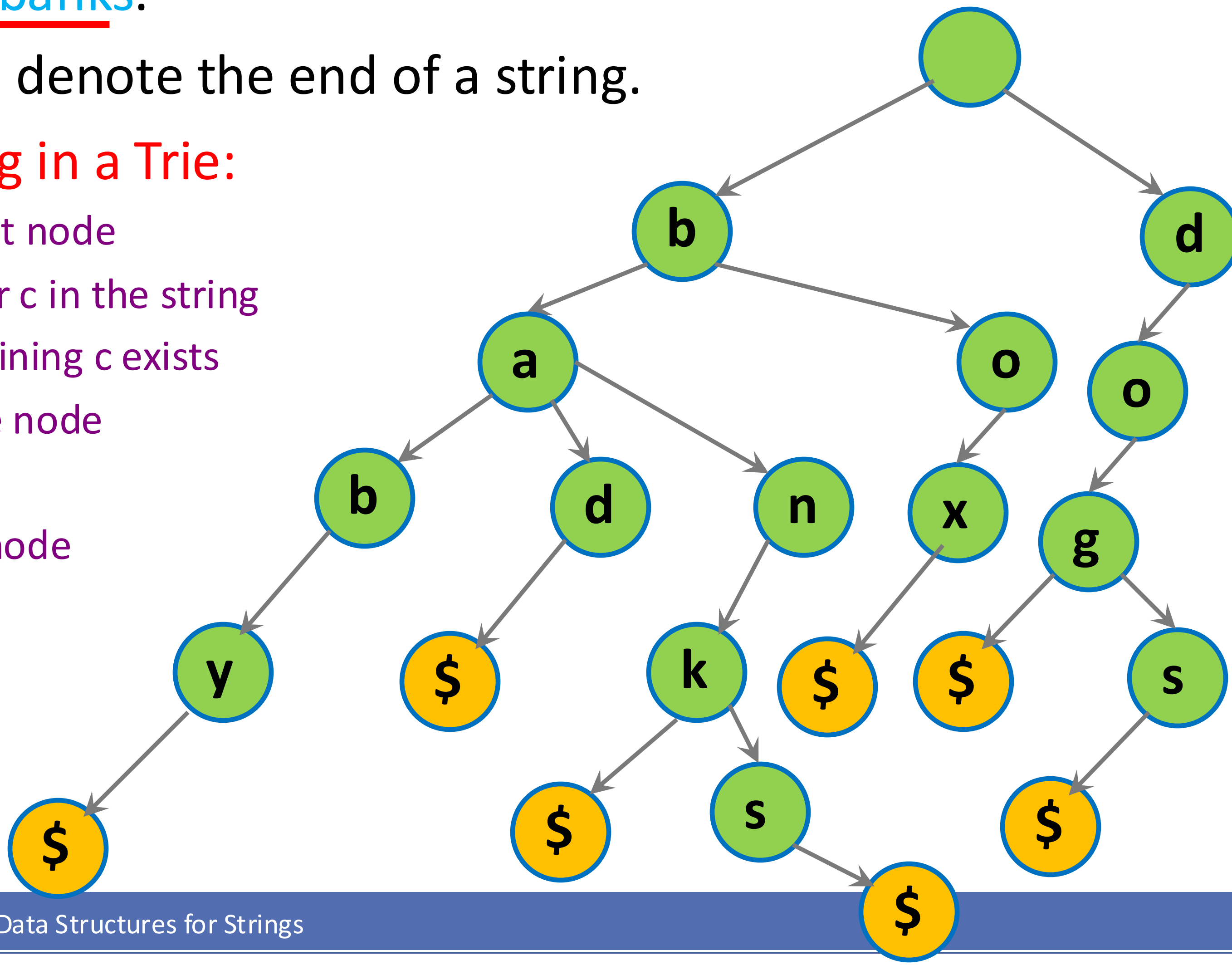
Trie Example: Insertion

Let's look at an example : a Trie that stores baby, bad, bank, box, dog, dogs, banks.

We will use $ to denote the end of a string.

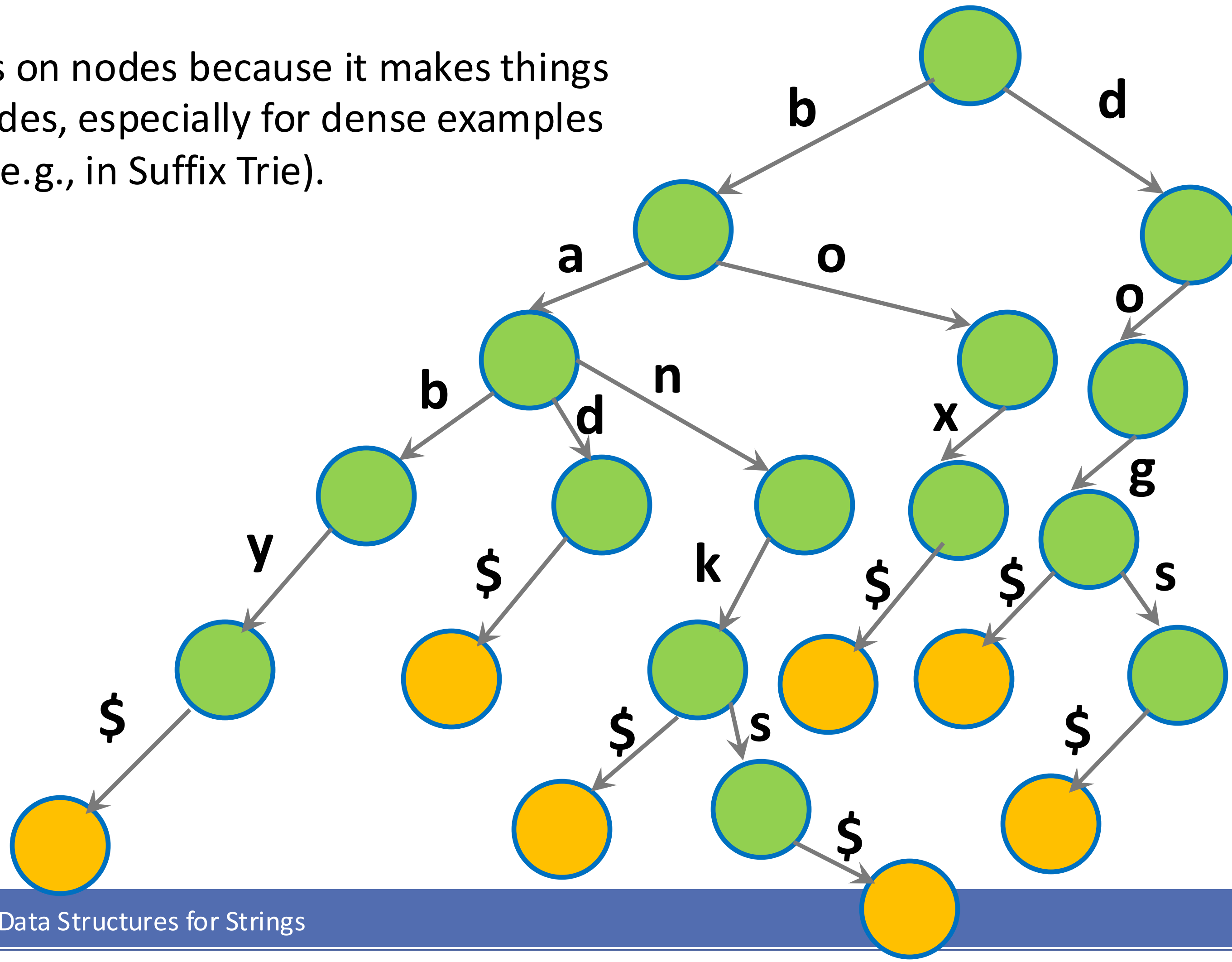
Inserting a string in a Trie:

- Start from the root node
- For each character c in the string
  - If a node containing c exists
    - Move to the node
  - Else
    - Create the node
    - Move to it



Alternative Illustration

- Traditionally, characters are shown on edges instead of nodes. However, these are just two different ways to illustrate.
- We show characters on nodes because it makes things clearer in lecture slides, especially for dense examples later in the lecture (e.g., in Suffix Trie).



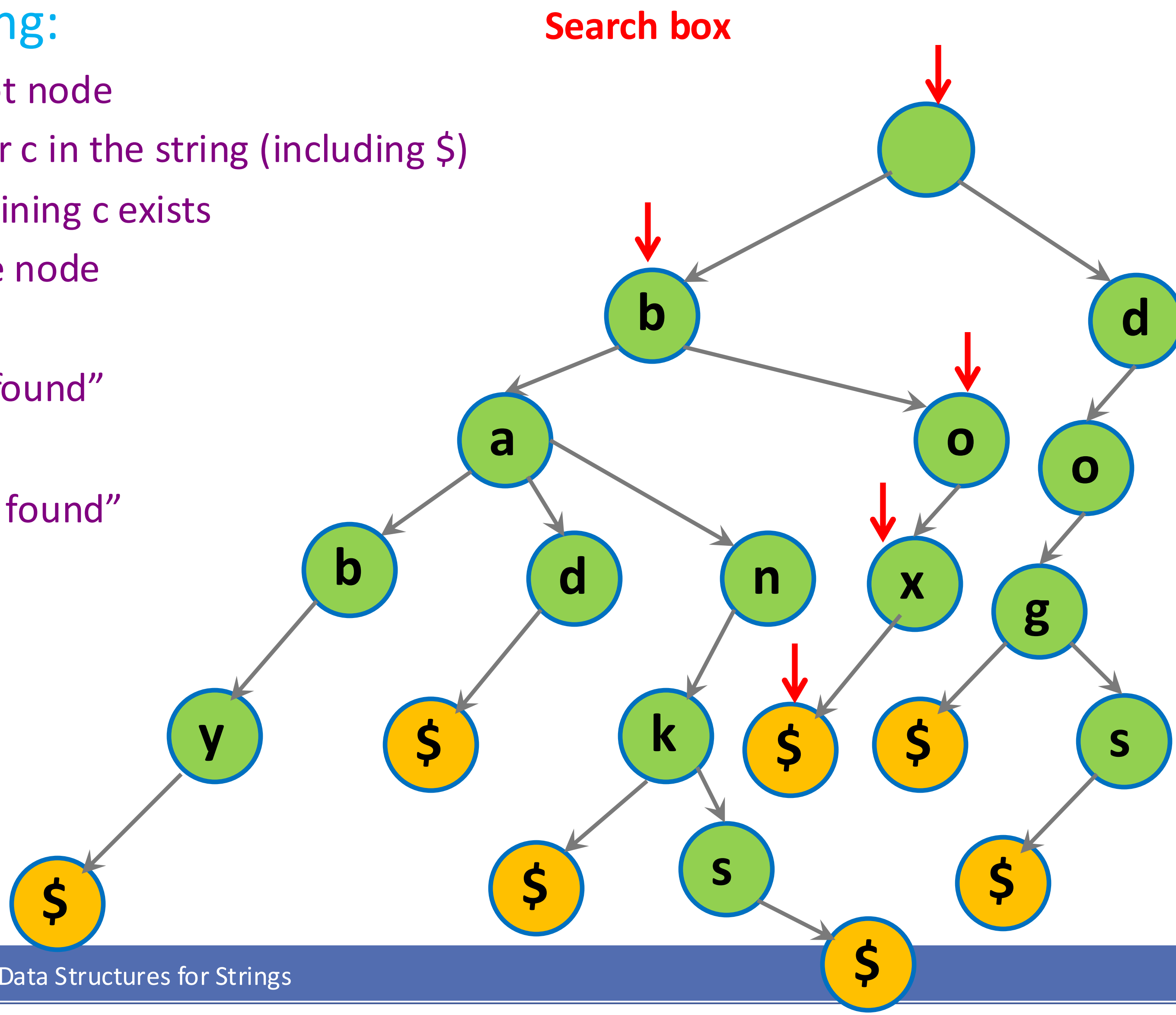
Outline

1. Trie
   - A. Construction
   - B. Query Processing
2. Suffix Trie
   - A. Construction
   - B. Query Processing
   - C. Suffix Tree
3. Suffix Array
   - A. Introduction
   - B. Reducing Construction Cost

Trie Example: Search

Searching a string:

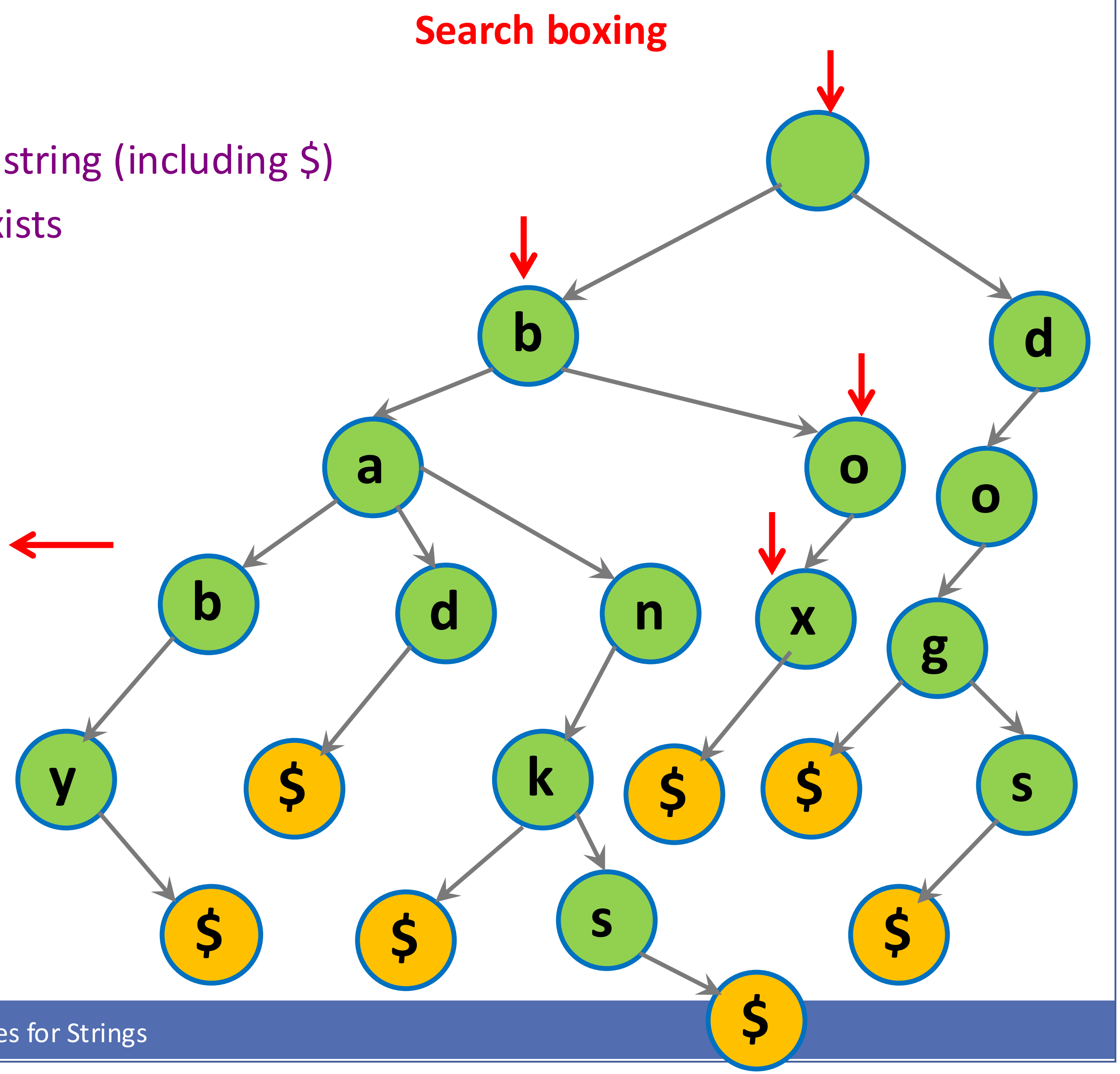
- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return “found”
  - Else
    - Return “not found”



Trie Example: Search

Searching a string:

- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return "found"
  - Else
    - Return "not found"

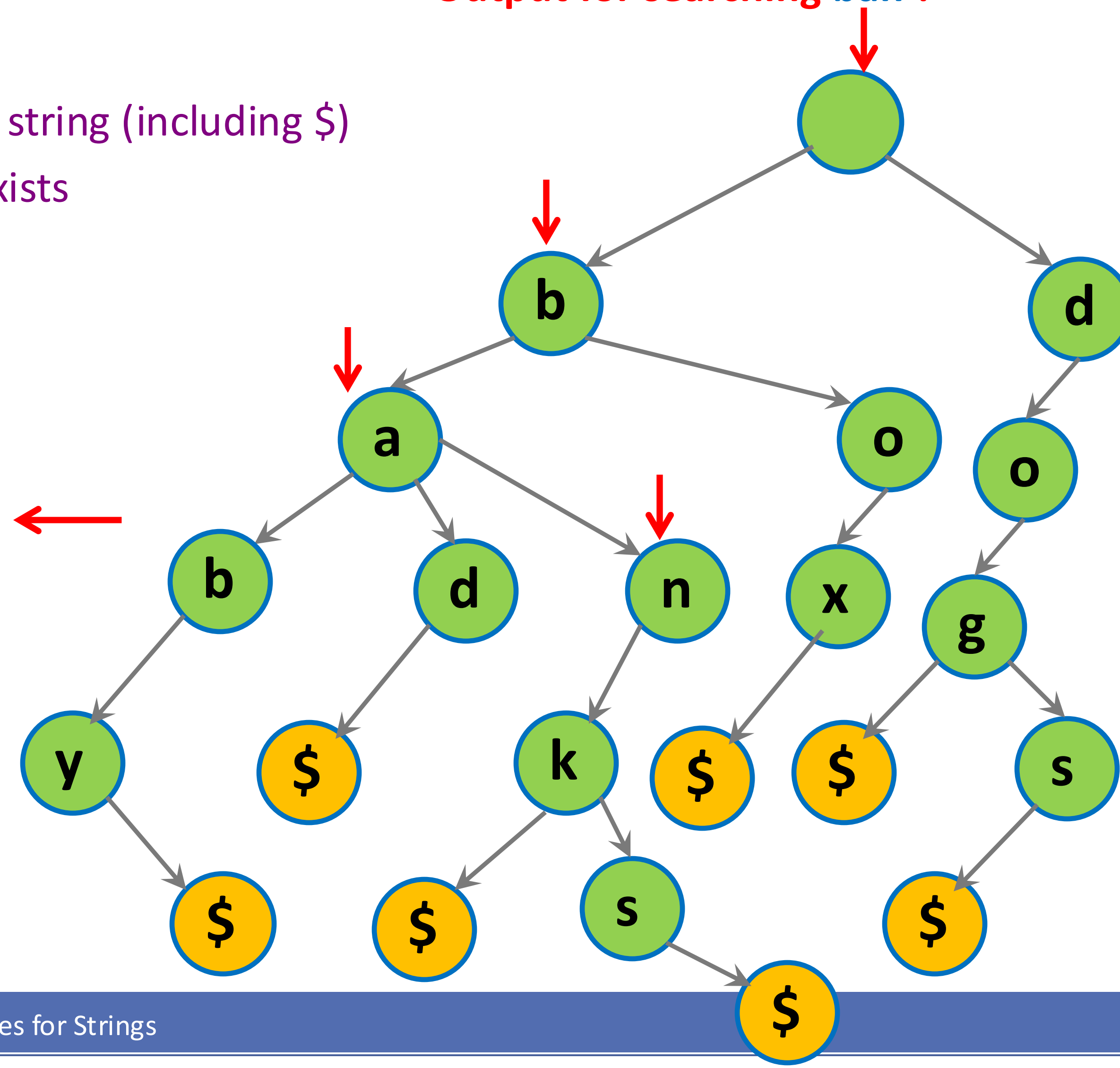


Trie Example: Search

Searching a string:

- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return “found”
  - Else
    - Return “not found”

Output for searching **ban** ?



Time Complexity?:

- For loop runs O(M) times.
- Time to check if a node containing c exists?
  - Depends on implementation, and on whether alphabet size is constant

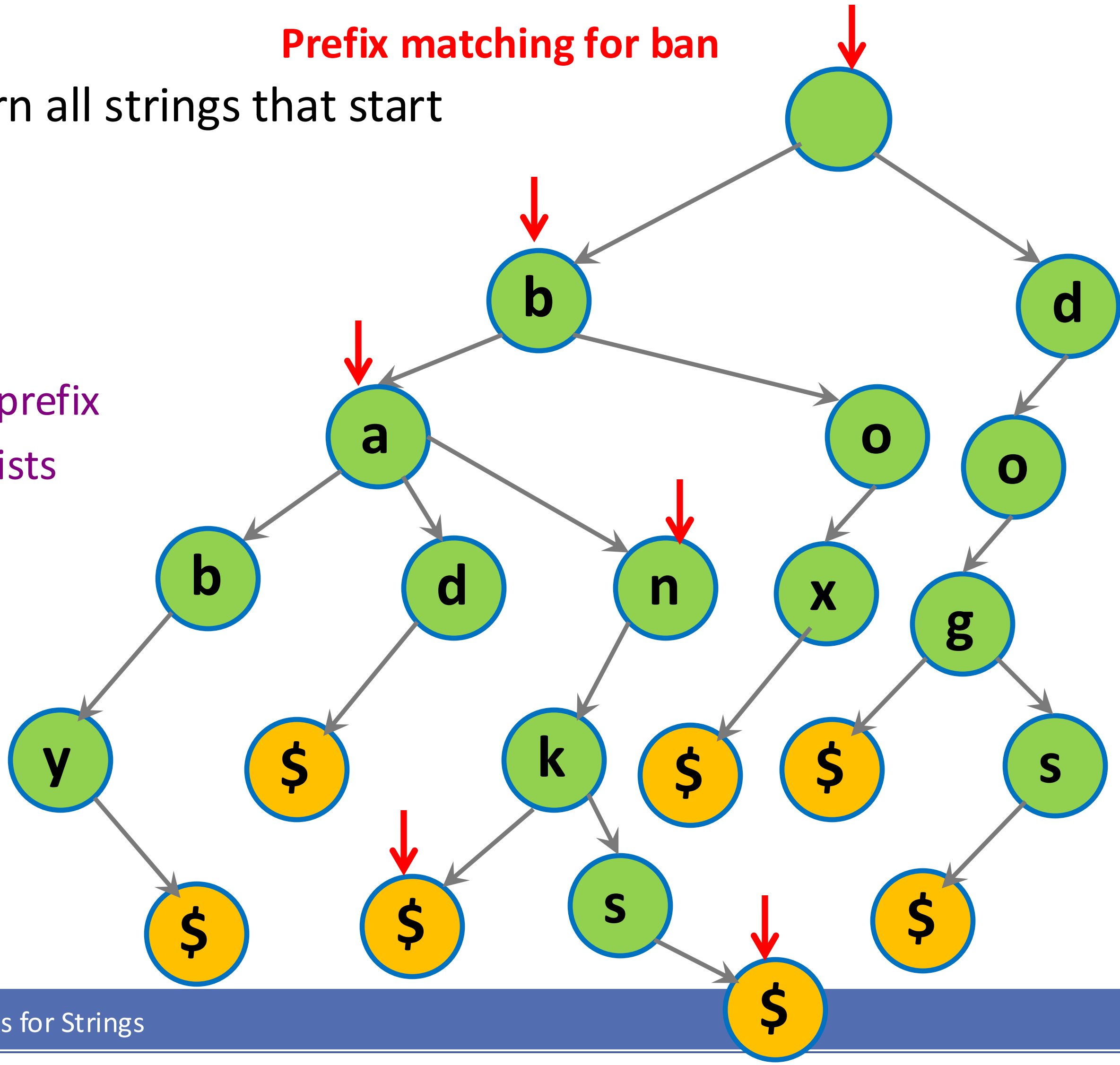
Trie Example: Prefix Matching

Prefix matching returns every string in text that has the given string as its **prefix**.

E.g., Autocompletion. Return all strings that start with "ban"

Prefix matching:

- Start from the root node
- For each character c in the prefix
  - If a node containing c exists
    - Move to the node
  - Else
    - Return "not found"
- Return all strings in the subtree rooted at the last node



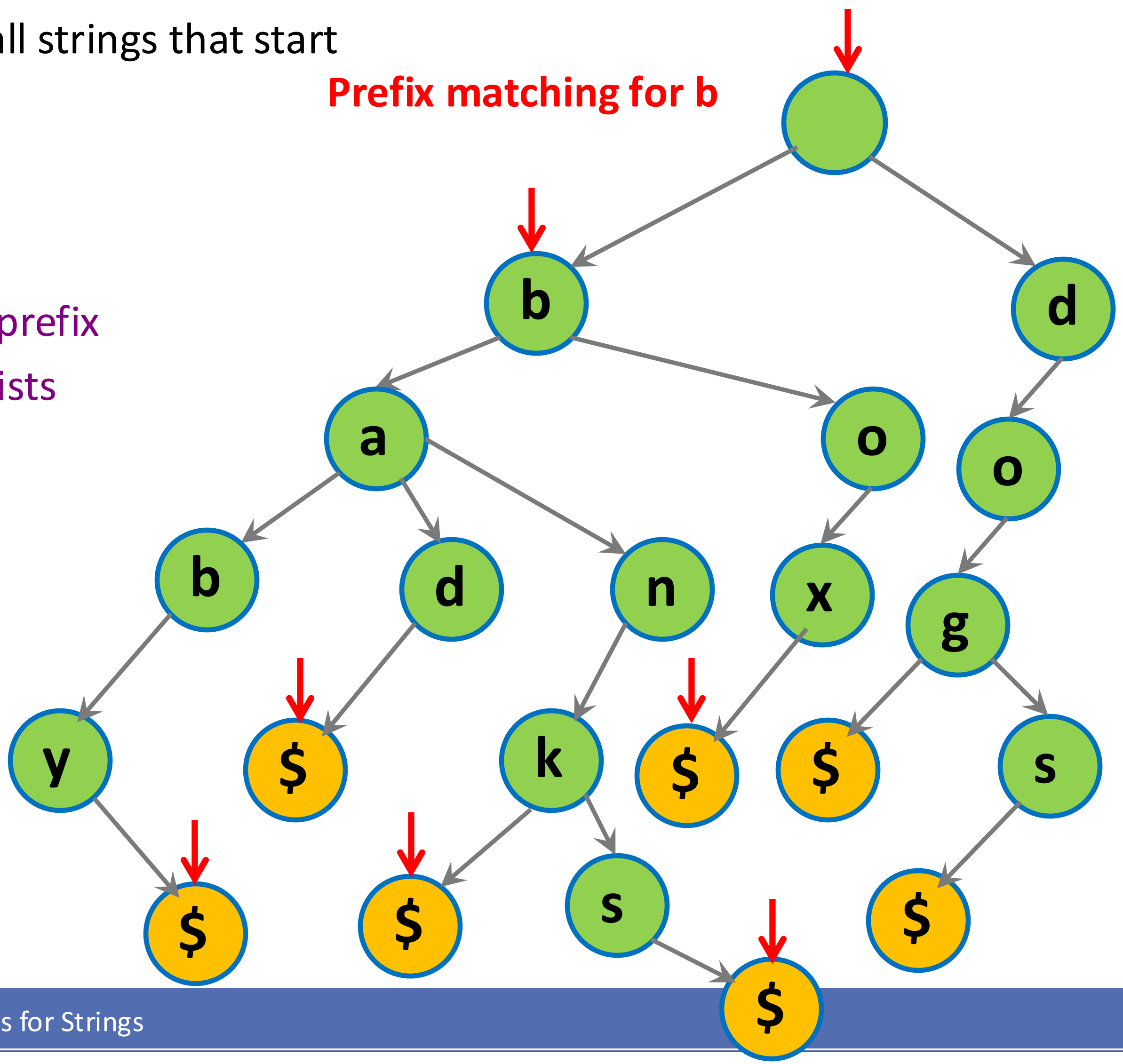
Trie Example: Prefix Matching

Prefix matching returns every string in text that has the given string as its **<u>prefix</u>**.

E.g., Autocompletion. Return all strings that start with "b"

Prefix matching:

- Start from the root node
- For each character c in the prefix
  - If a node containing c exists
    - Move to the node
  - Else
    - Return "not found"
- Return all strings in the subtree rooted at the last node



Implementing a Trie

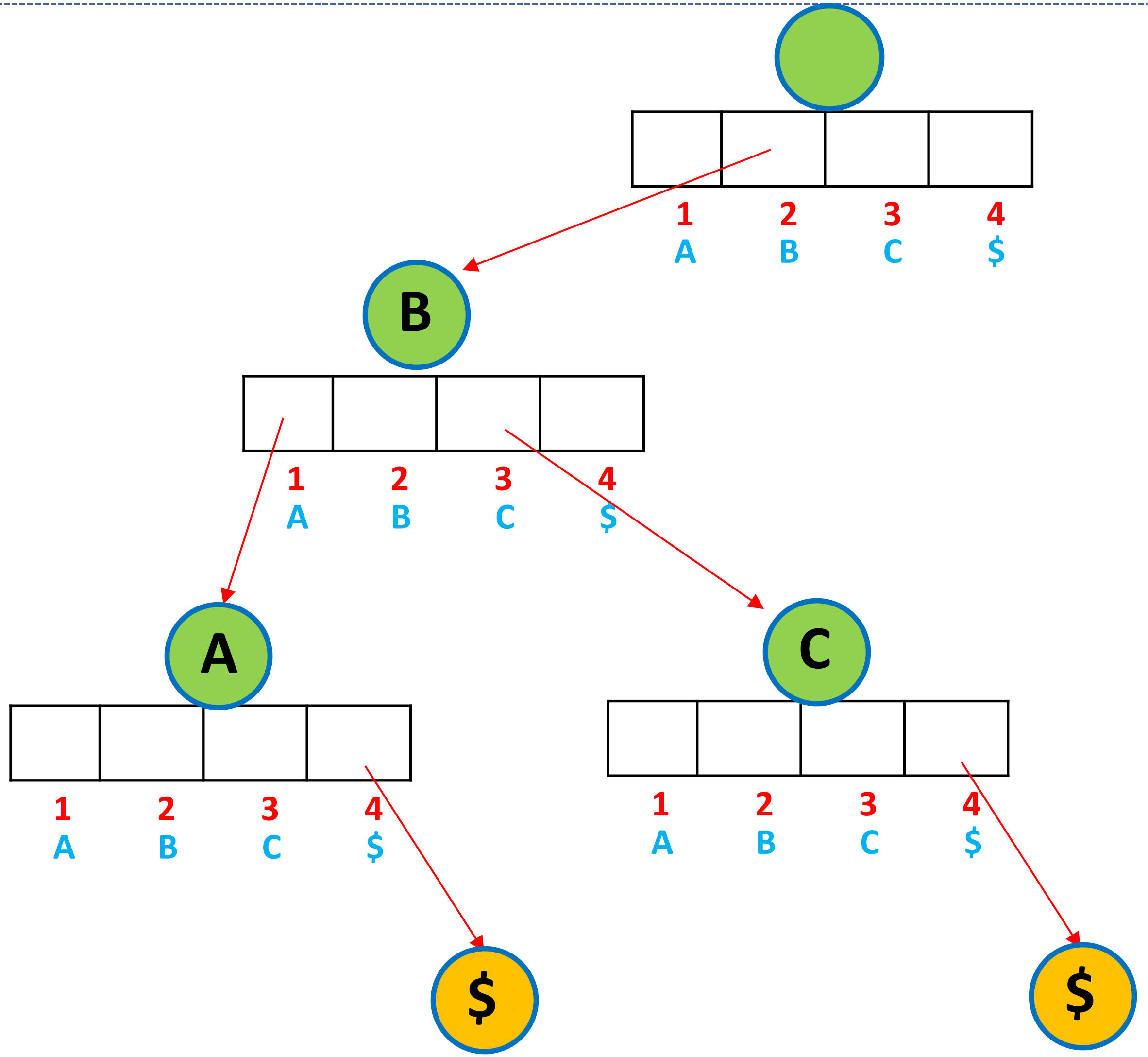
Implementation using an array:

- At each node, create an array of alphabet size (e.g., 26 for English letters, 4 for DNA strings)
- If i-th node exists, add pointer to it at array[i]
- Otherwise, array[i] = Nil.

The above implementation allows checking whether a node exists or not in O(1) (for constant-sized alphabets)

Other implementations are possible (e.g., using linked lists or hash tables).

Example: Implementing a Trie using arrays (assuming only three letters A,B,C)



Insert BA$

Insert BC$

Advantages and Disadvantages of Trie

Advantages

- A better search structure than a binary search tree (covered in Week 11) with string keys.
- A more versatile search structure than hash table.
- Allows lookup on prefix matching in O(M)-time where M is the length of prefix.
- Allows sorting collection of strings in O(MN) time where MN is the total number of characters in all strings

Disadvantages

- On average Tries can be slower (in some cases) than hash tables for looking up patterns/queries.
- Wastes space, since even when a node has few children, you need to create an array of size alphabet

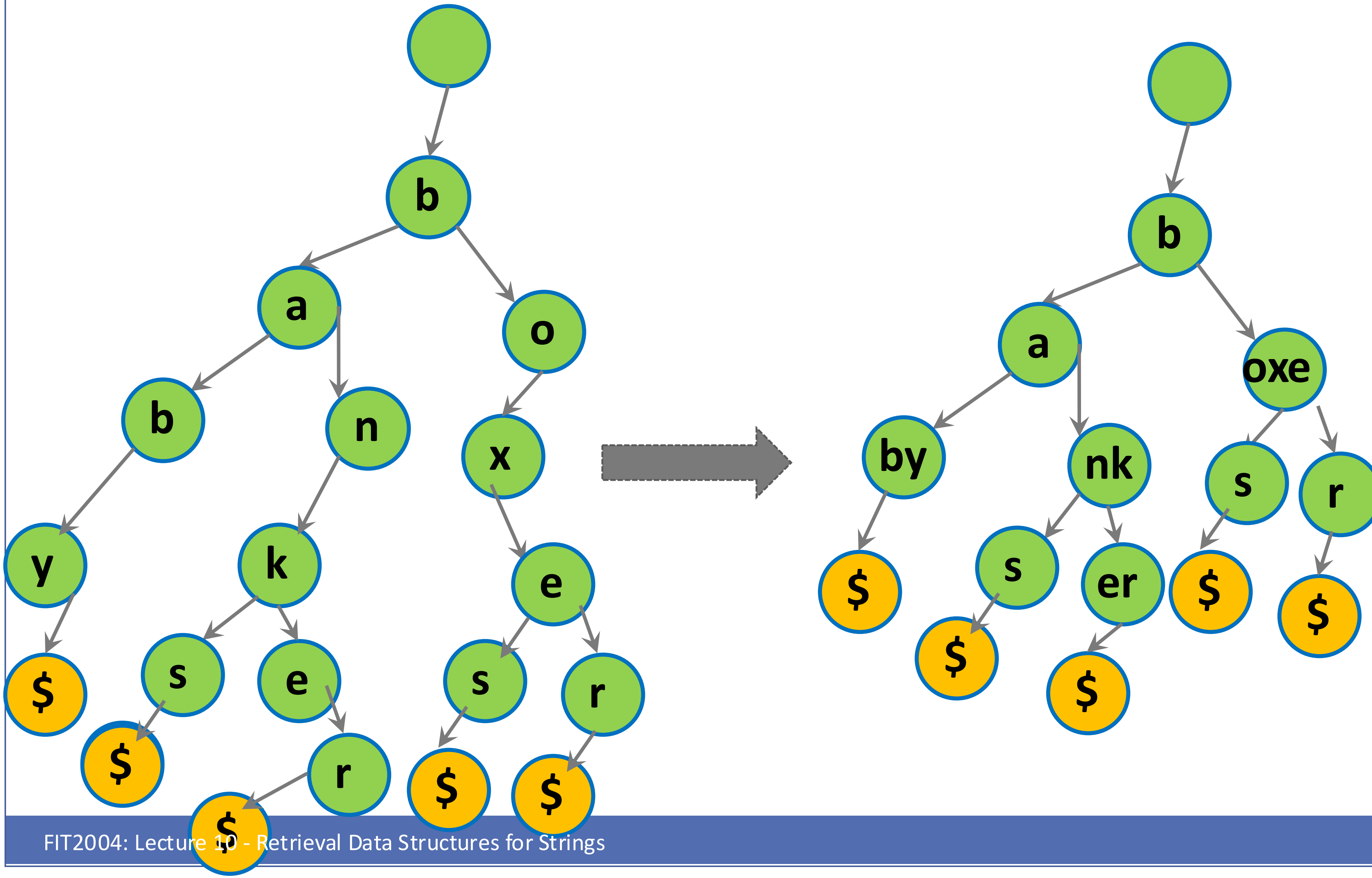
Some properties of Trie

- The maximum depth is the length of longest string in the collection.
- Insertion, Deletion, Lookup operations take time proportional to the length of the string/pattern being inserted, deleted, or searched.
- But we waste a lot of space if
  - each node has 1 pointer per symbol in the alphabet.
  - deeper nodes typically have mostly null pointers.
- Can reduce total space usage by turning each node into a linked list or binary search tree etc, trading off time for space.

Radix/PATRICIA Tree (<u>**NOT EXAMINABLE**</u> BUT WORTH MENTIONING)

- Radix/PATRICIA tree is a space-optimized/compact Trie data structure
- Unlike regular tries, edges can be labelled with substrings of characters.
- The nodes along a path having exactly one child are merged
- This makes them much more efficient for sets of strings that share long prefixes or substrings.

Radix/PATRICIA Tree (NOT EXAMINABLE BUT WORTH MENTIONING)

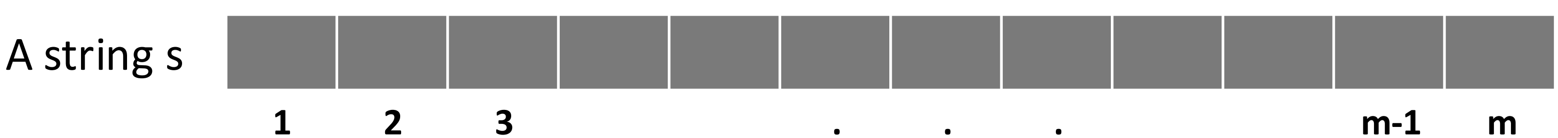


Outline

1. Trie
   A. Construction
   B. Query Processing
2. Suffix Trie
   A. Construction
   B. Query Processing
   C. Suffix Tree
3. Suffix Array
   A. Introduction
   B. Reducing Construction Cost

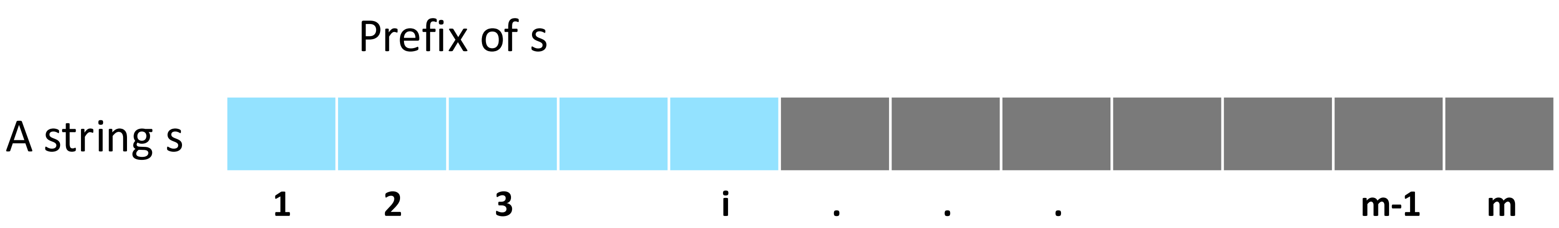
Prefixes and suffixes

- (Prefix) Tries are very useful for quickly looking up whole words, but more generally, **prefixes** of words



Prefixes and suffixes

- (Prefix) Tries are very useful for quickly looking up whole words, but more generally, **prefixes** of words
- A prefix of a word s[1..m] is some string s[1..i] where 1<=i<=m

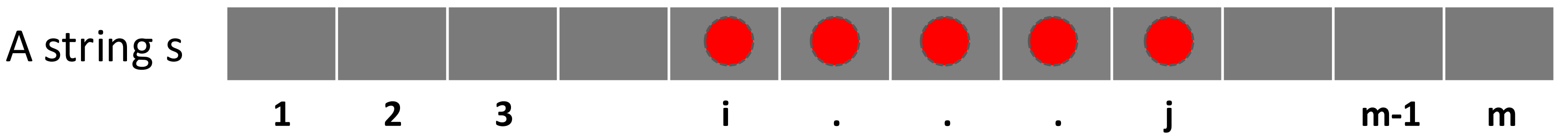


Prefixes and suffixes

- (Prefix) Tries are very useful for quickly looking up whole words, but more generally, **prefixes** of words
- A prefix of a word s[1..m] is s[1..i] where 1<=i<=m
- A suffix of a word s[1..m] is s[i..m] where 1<=i<=m

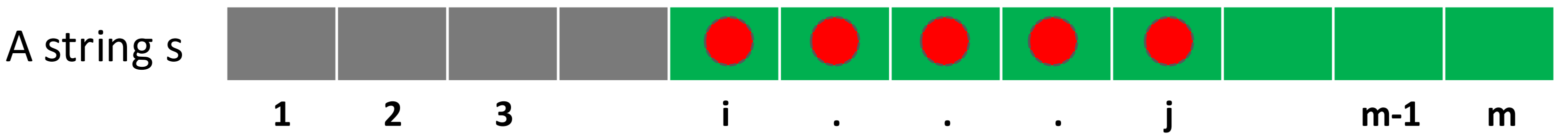
Prefixes and suffixes

- Any substring of a word is a **prefix** of some **suffix**
- In other words, a substring of s[1..m] is s[i..j]



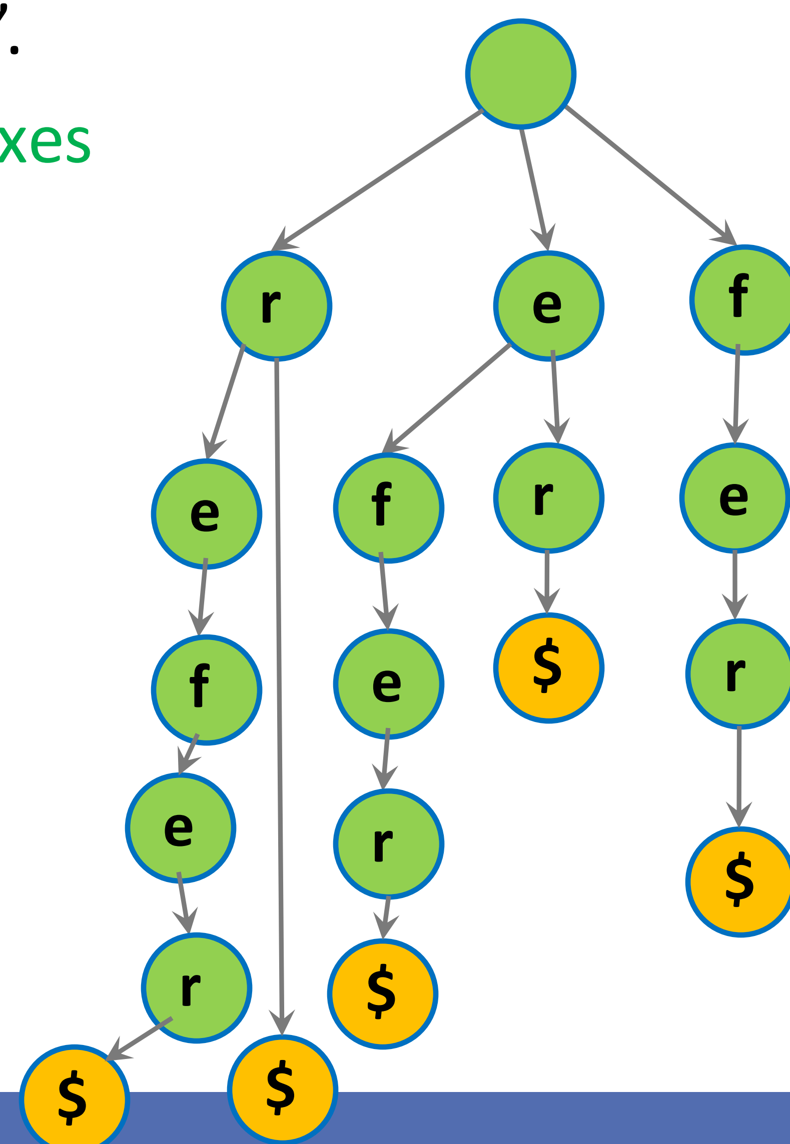
Prefixes and suffixes

- Any substring of a word is a **prefix** of some **suffix**
- In other words, a substring of s[1..m] is s[i..j]
- s[i..j] is a prefix of s[i..m] (which is a suffix of s[1..m])
- To be able to efficiently search **substrings**…
- Just make a prefix trie of suffixes



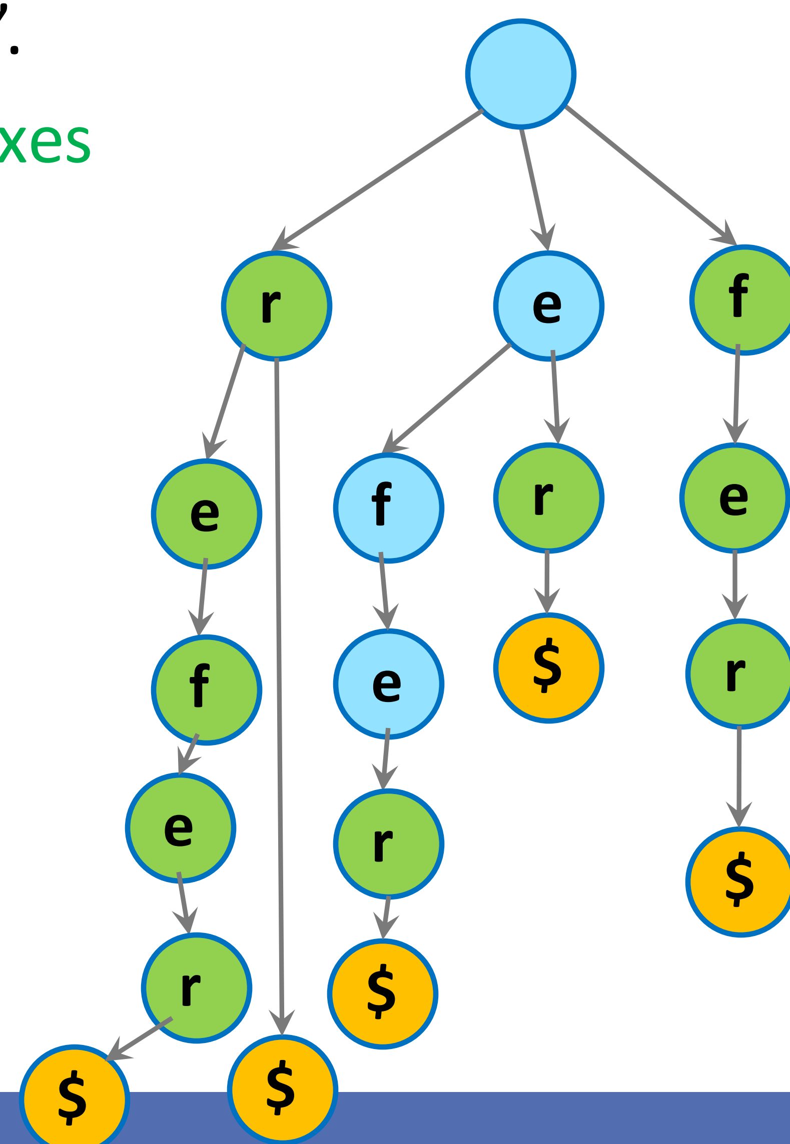
Suffix Trie

- Consider some text, e.g., “refer”.
- A Trie constructed using all suffixes of the text is called a Suffix Trie
- Pick any substring, eg “efe”



Suffix Trie

- Consider some text, e.g., “refer”.
- A Trie constructed using all suffixes of the text is called a Suffix Trie
- Pick any substring, eg “efe”
- Notice that it traces a path from root to some node



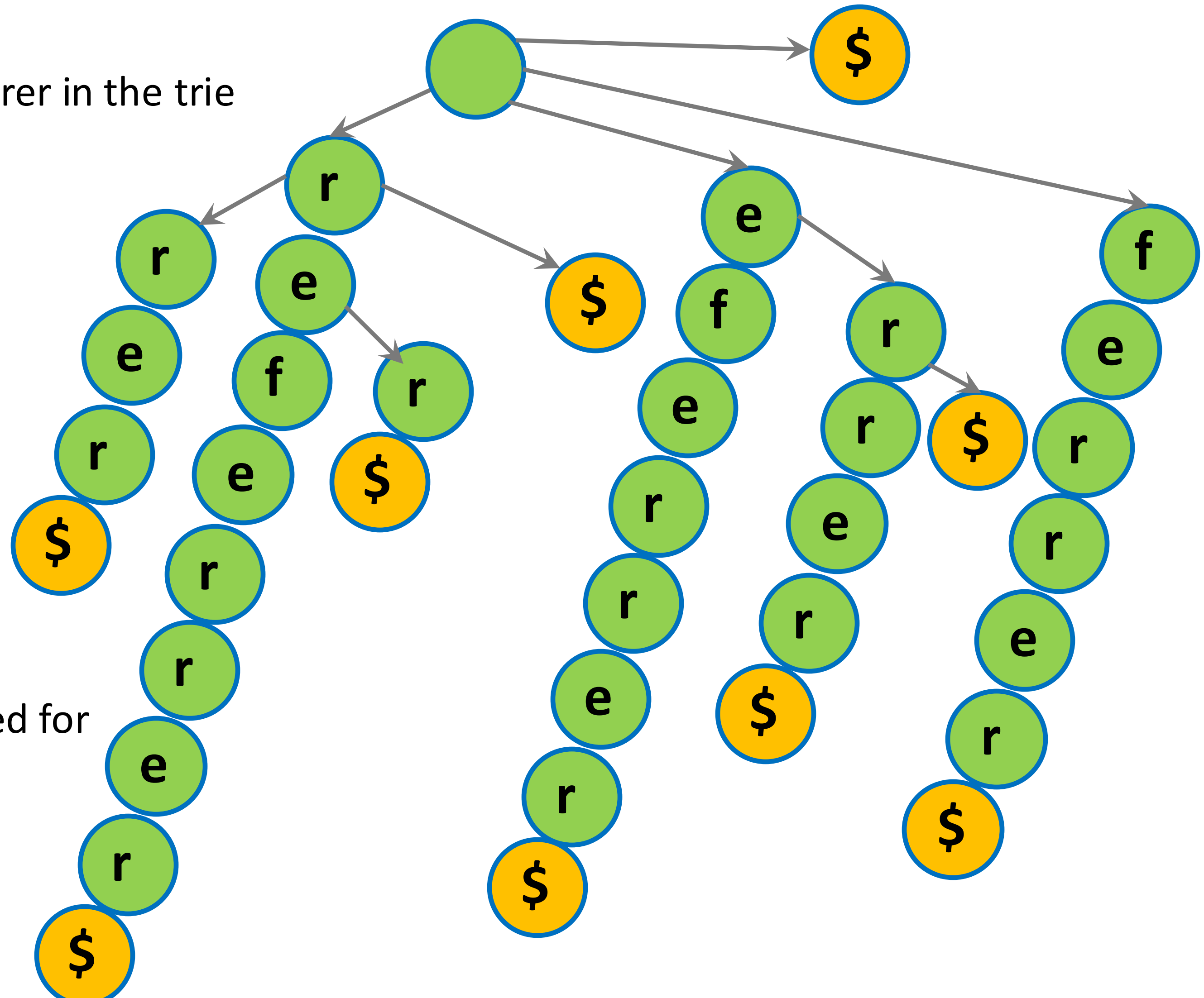
Constructing Suffix Trie

Suffix trie of referrer$

Insert all suffixes of referrer in the trie

1. referrer$
2. eferrer$
3. ferrer$
4. errer$
5. rrer$
6. rer$
7. er$
8. r$
9. $

Many arrows are removed for better visualization



Outline

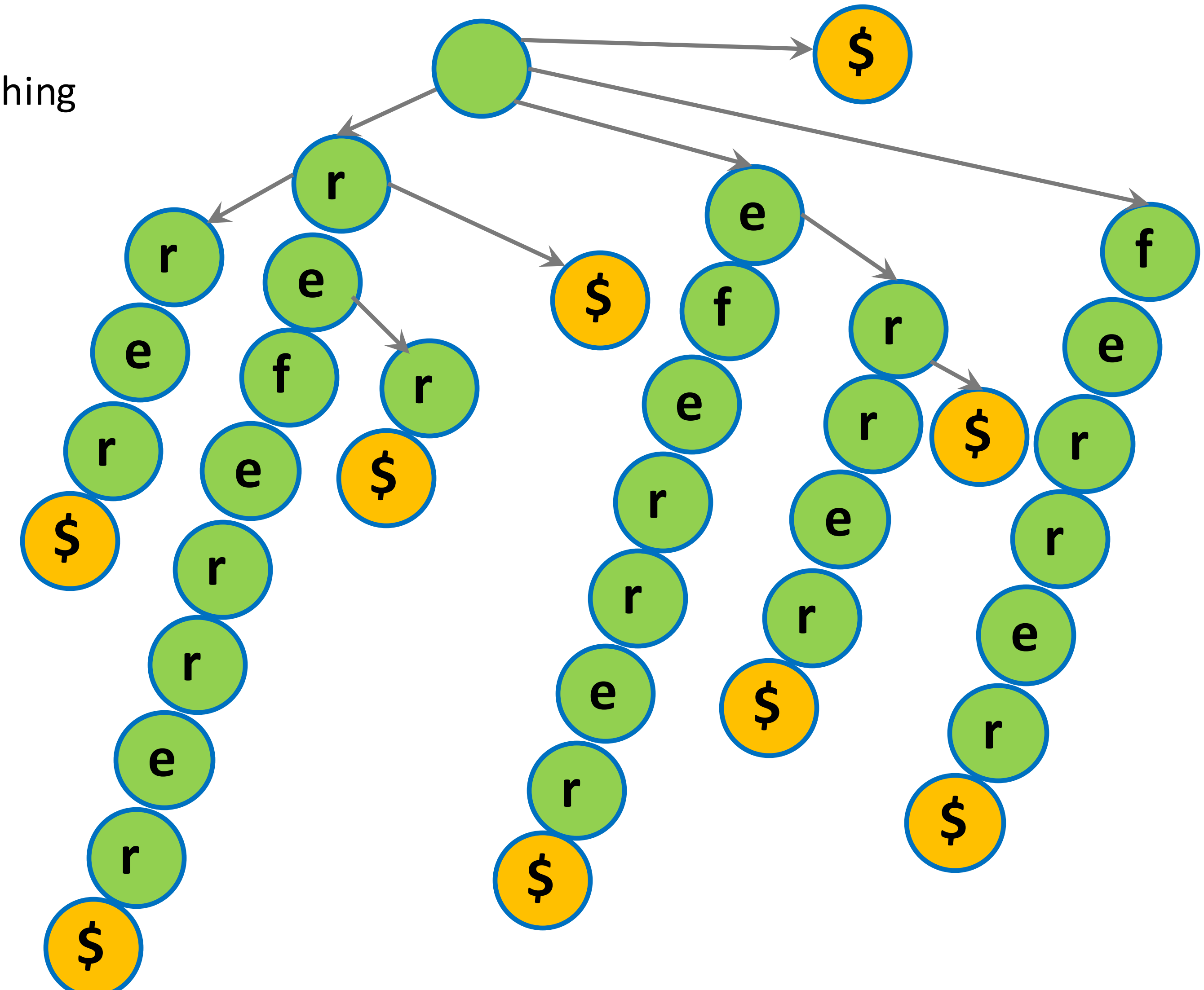
1. Trie
   - A. Construction
   - B. Query Processing
2. Suffix Trie
   - A. Construction
   - B. Query Processing
   - C. Suffix Tree
3. Suffix Array
   - A. Introduction
   - B. Reducing Construction Cost

Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

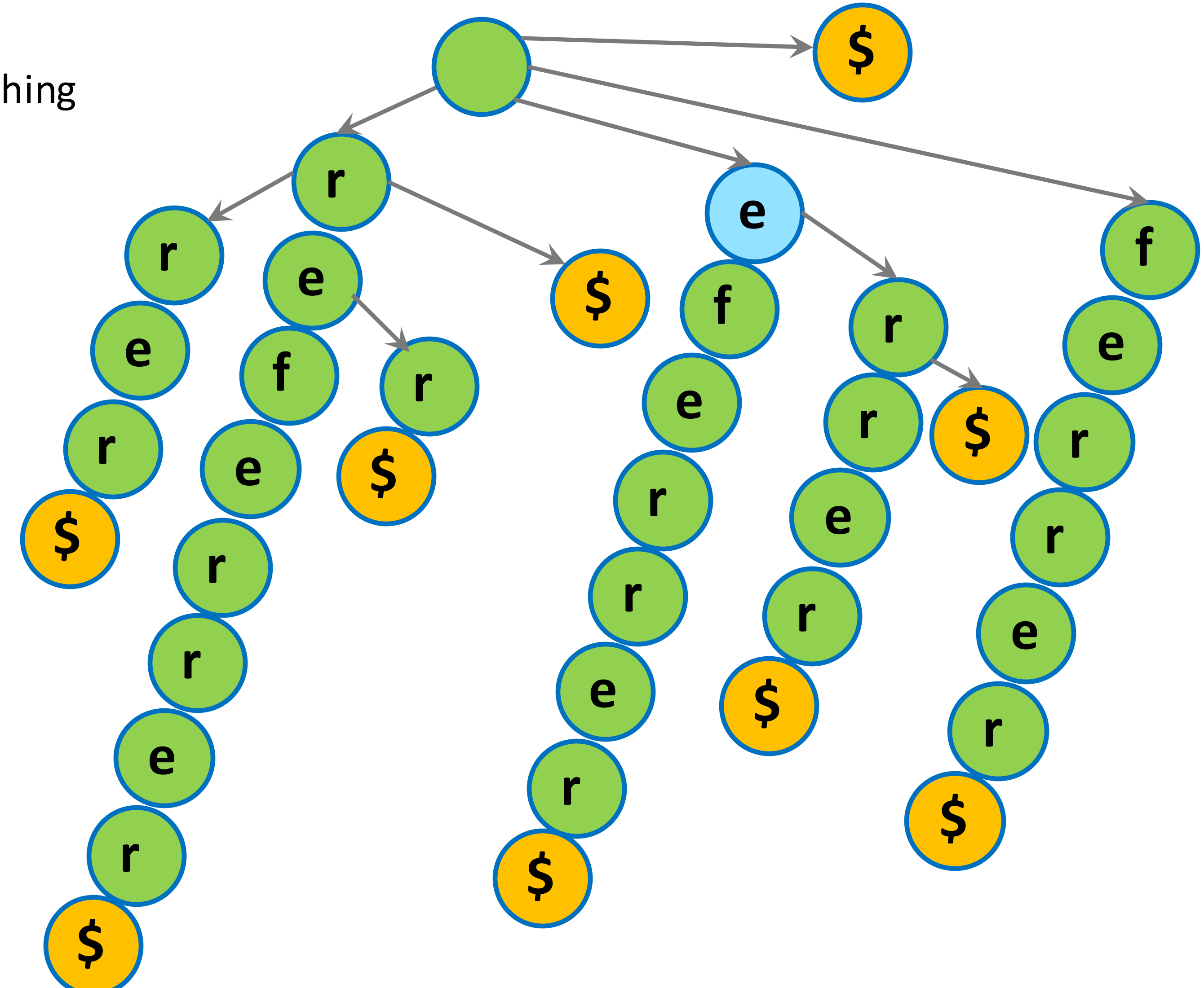


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

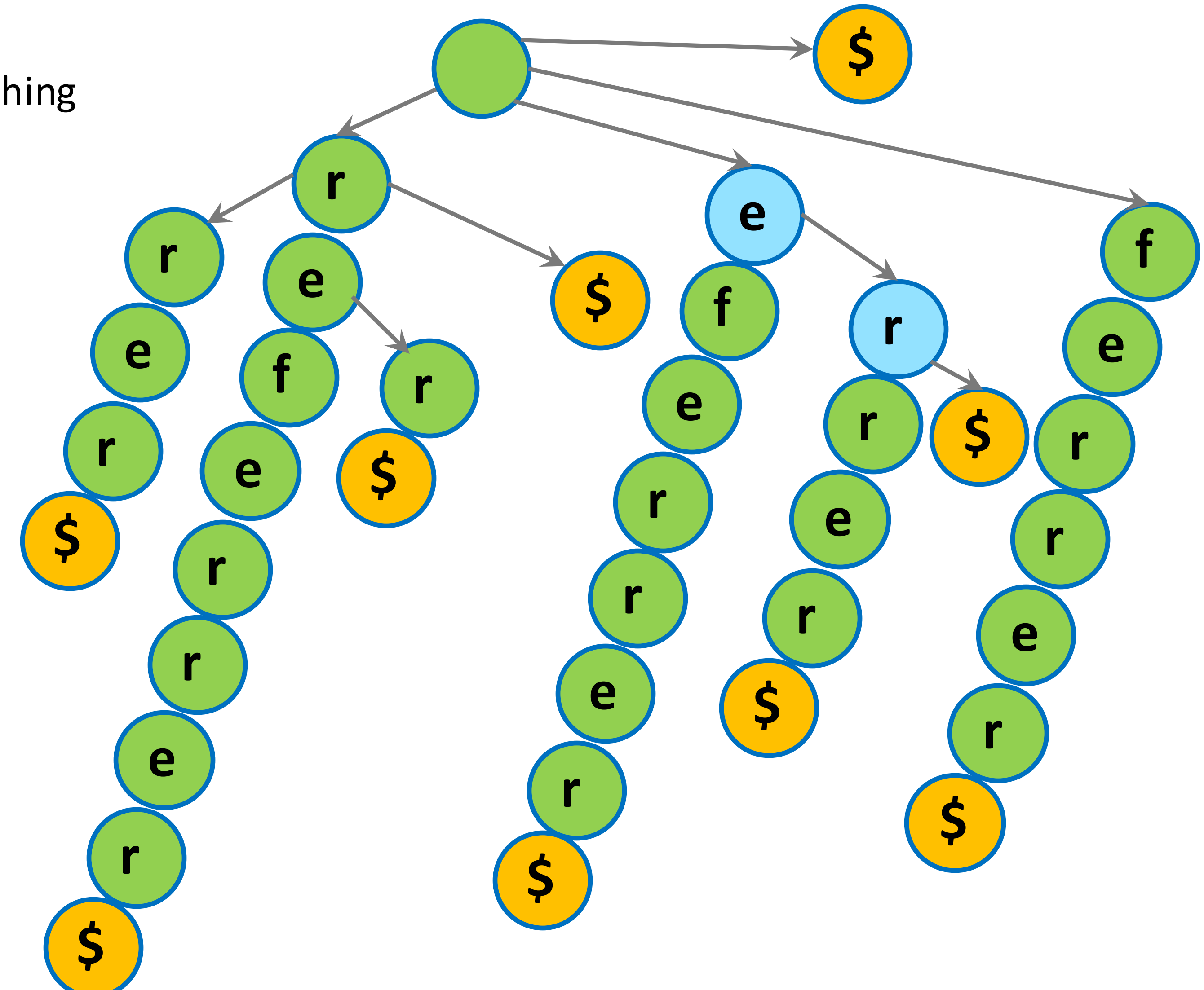


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search "err"

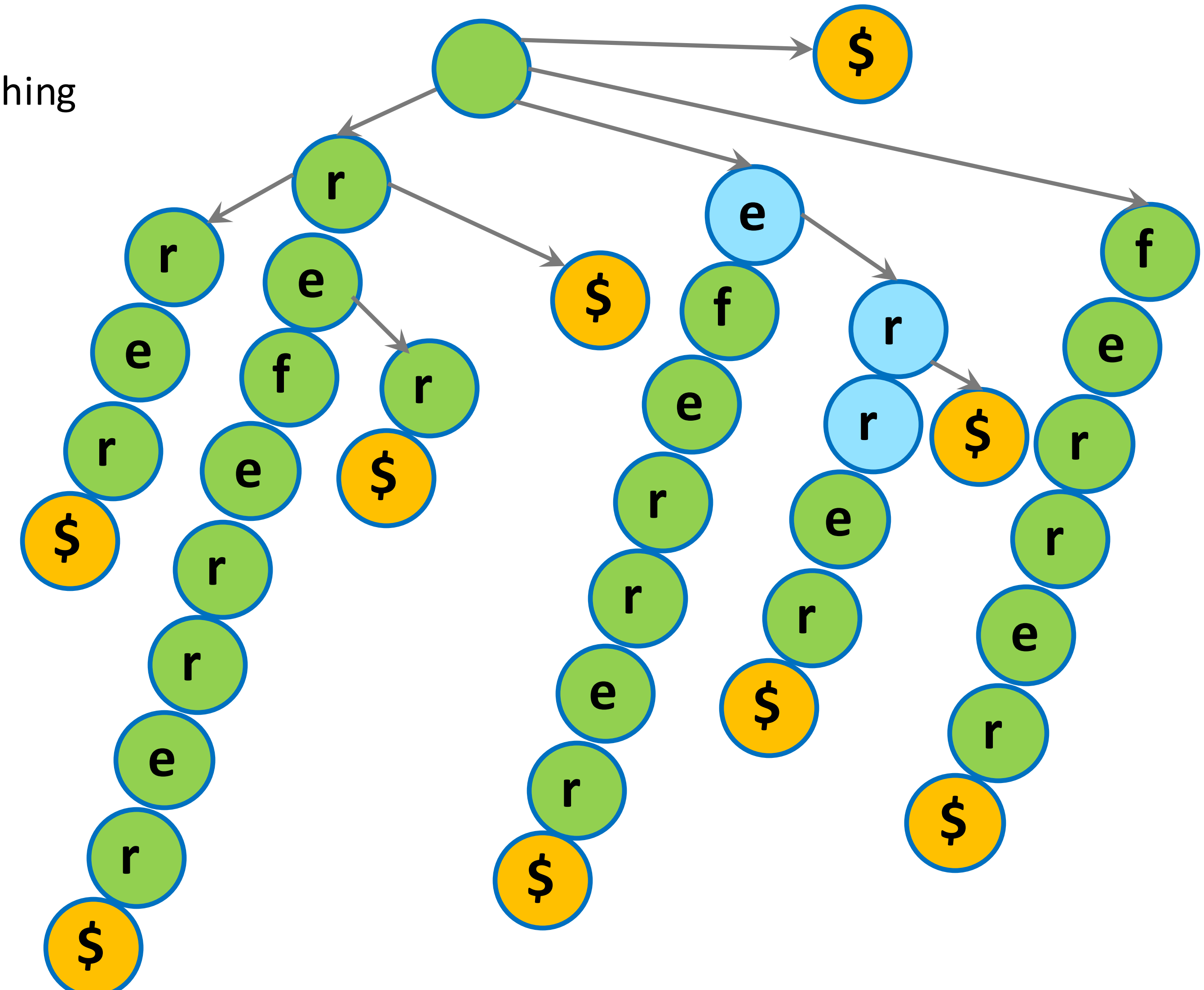


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search "err"



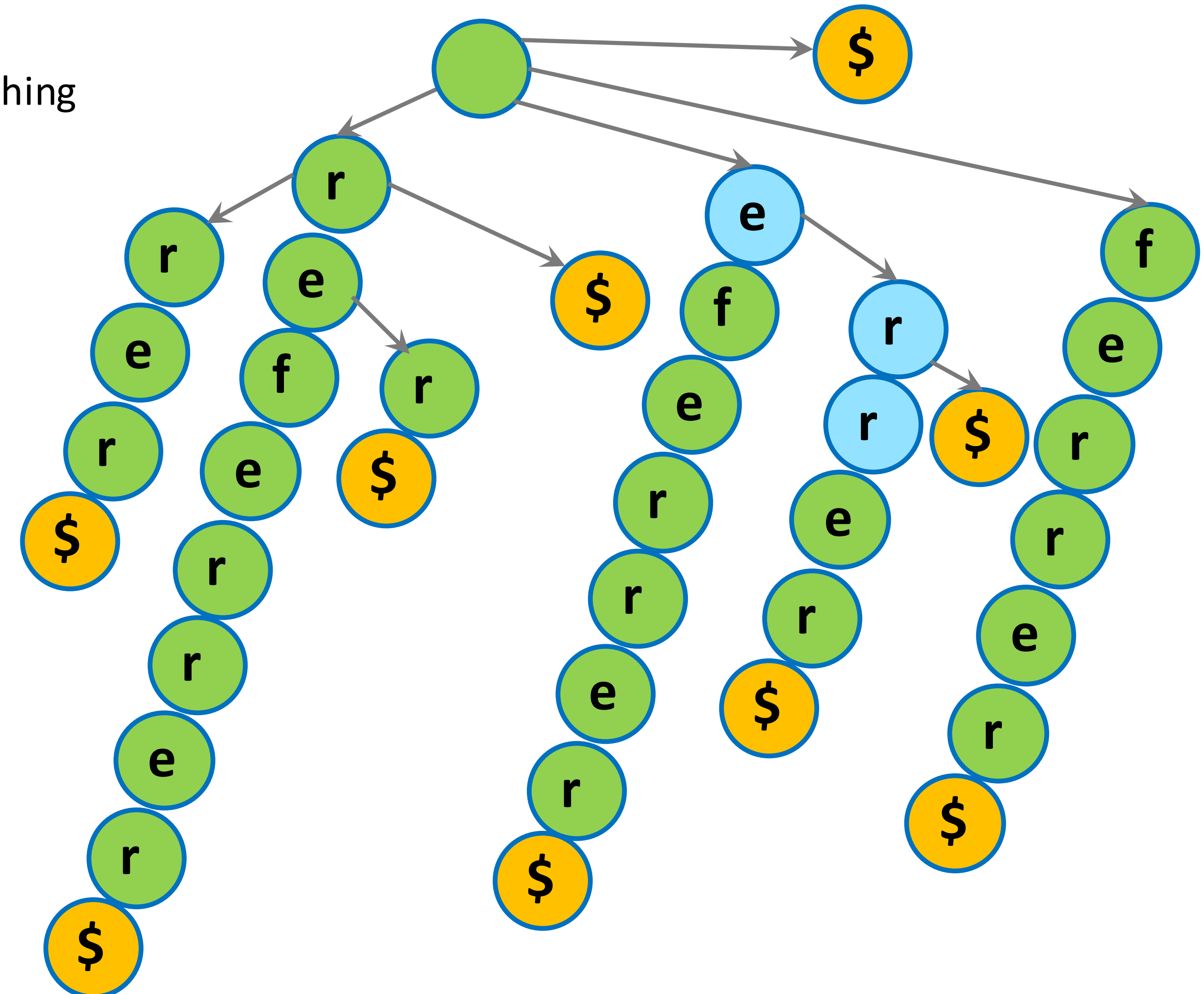
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

Found!



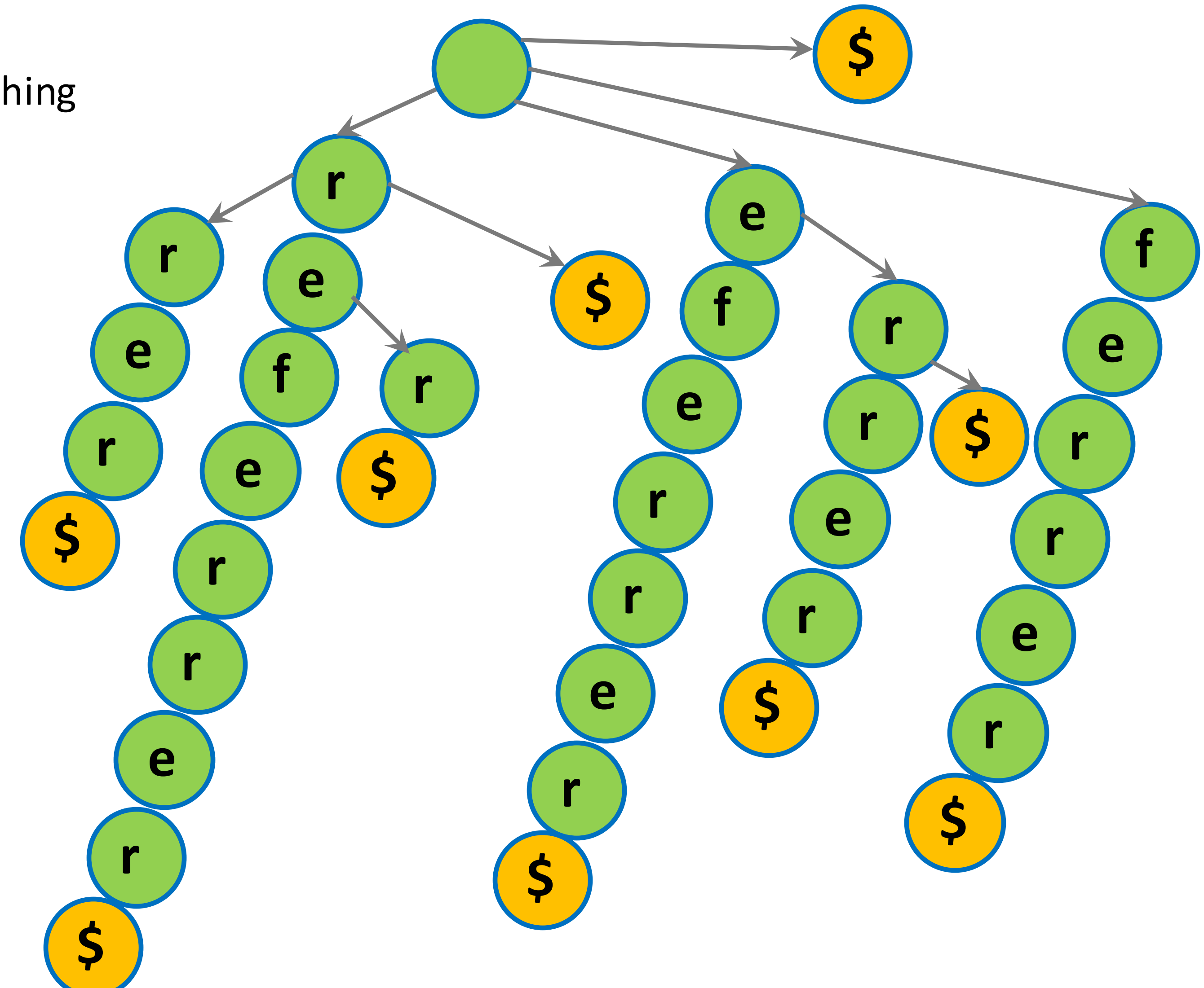
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



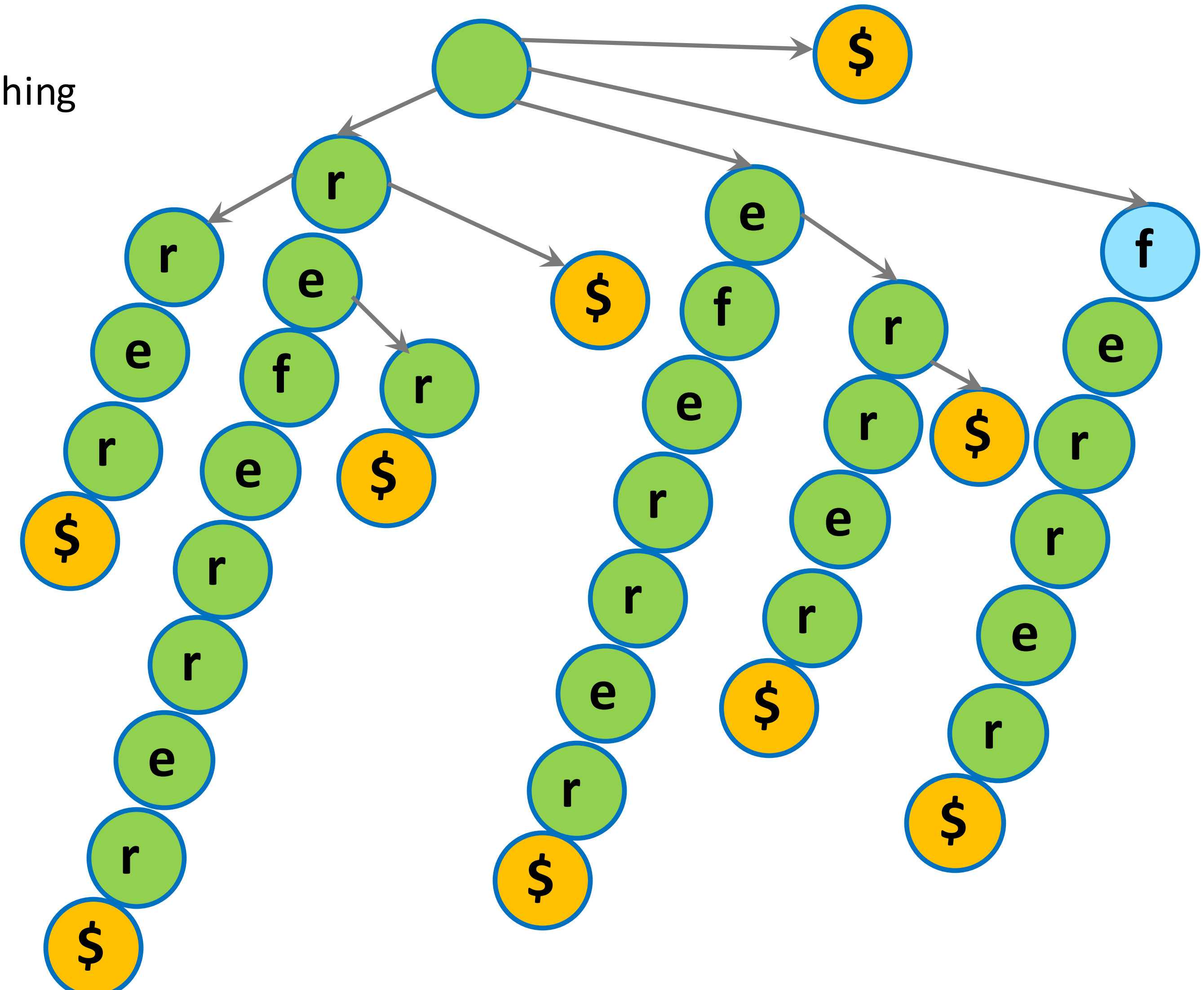
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



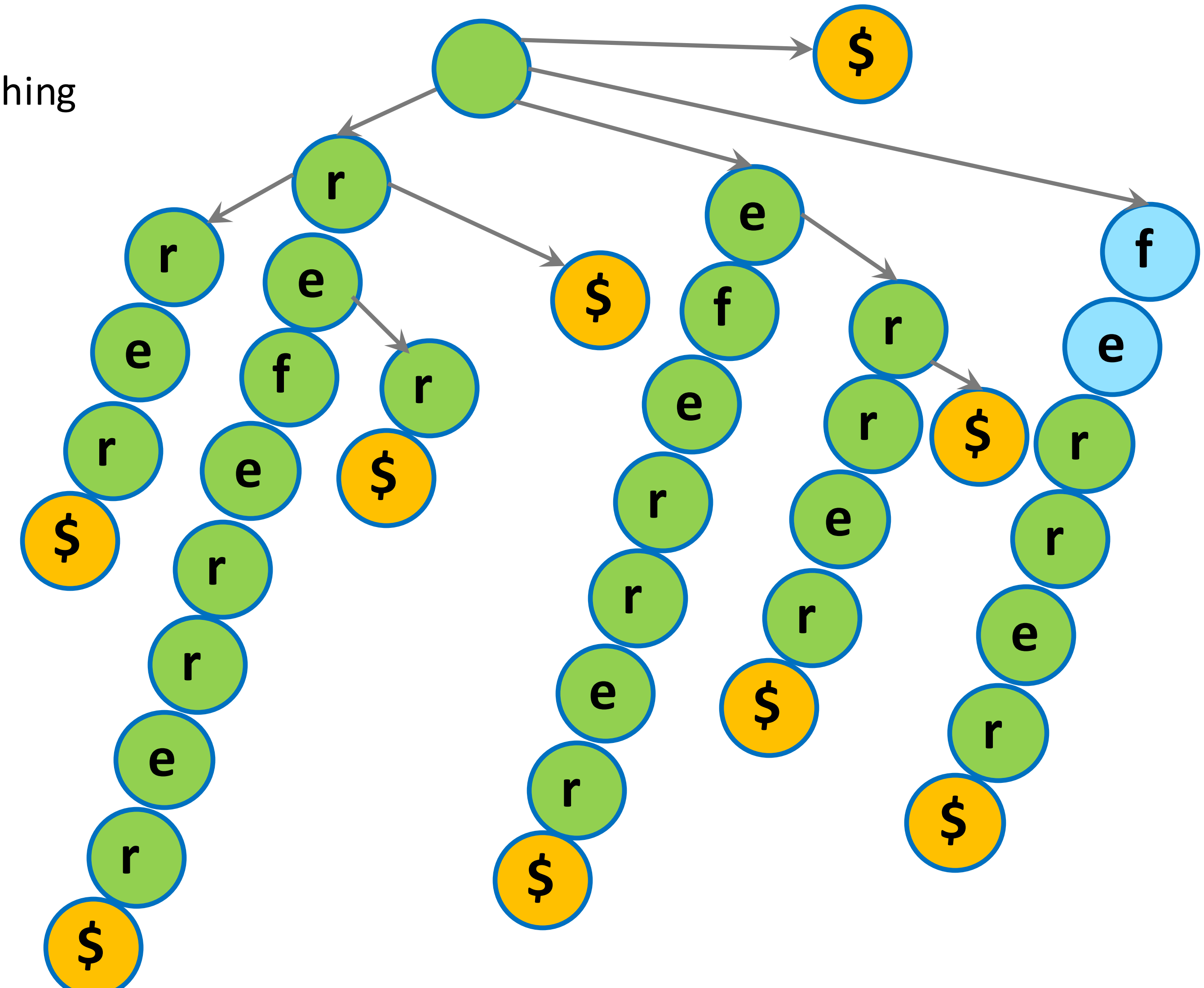
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search "err"

search "fers"



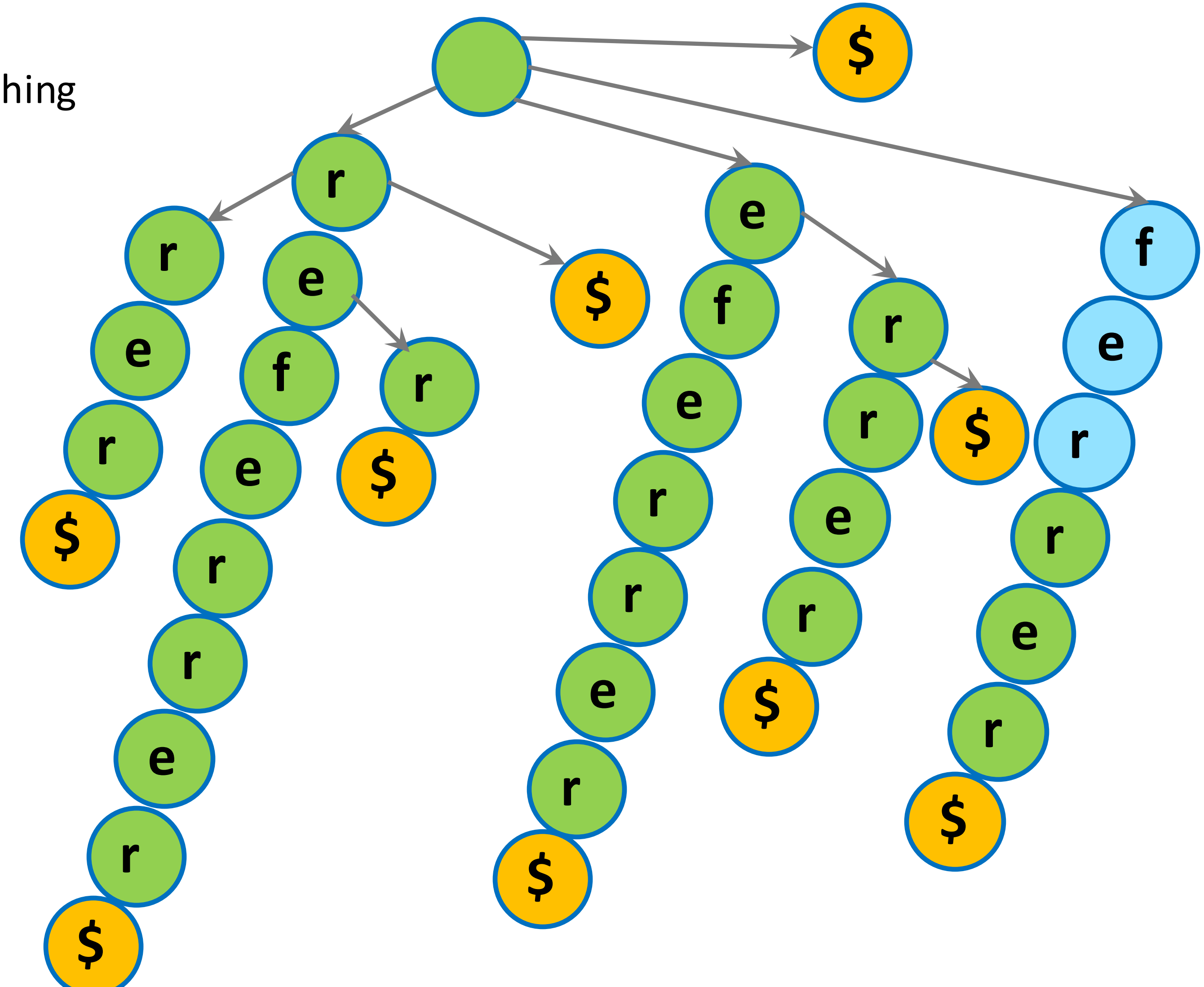
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



Substring search on Suffix Trie

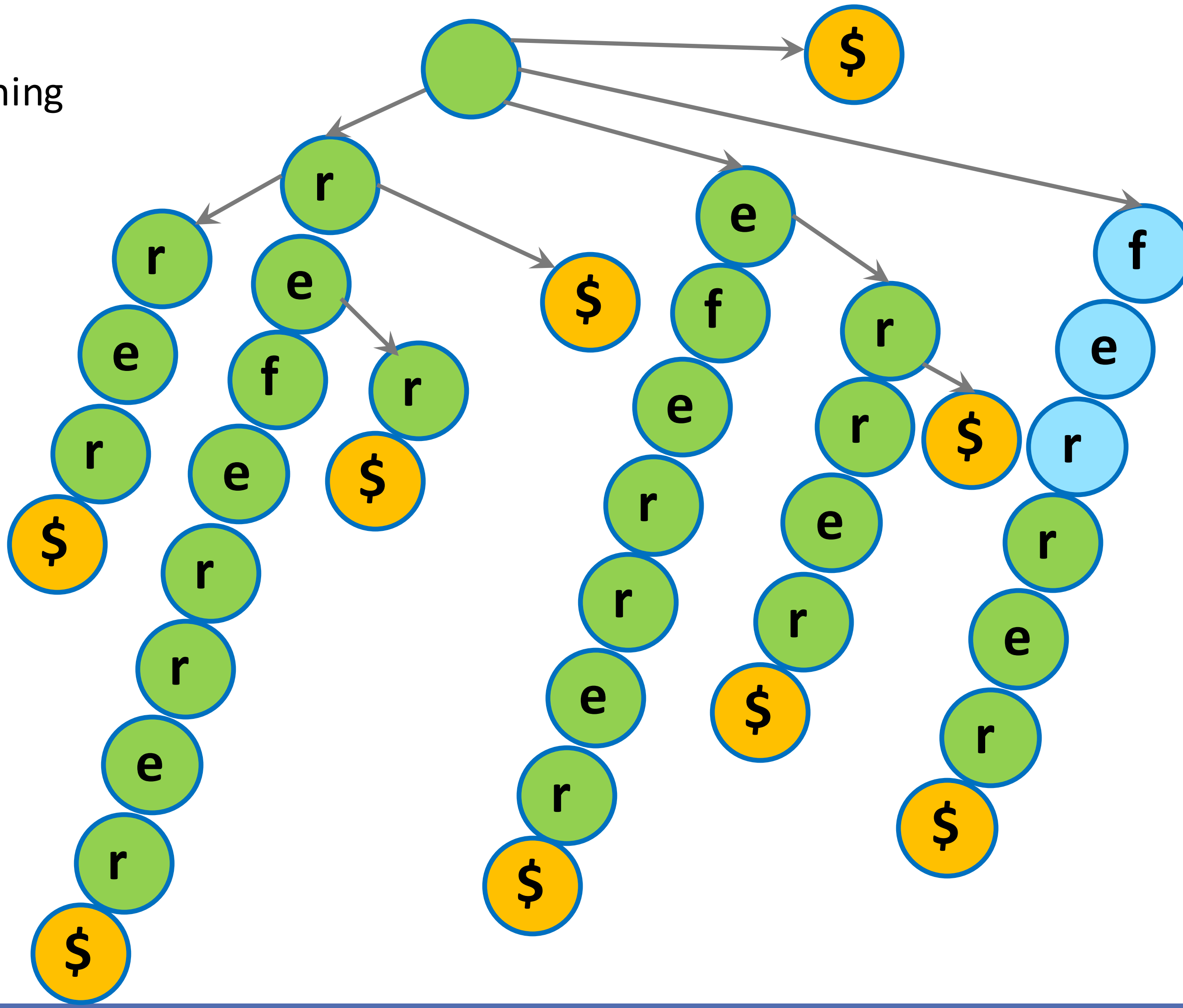
Substring search for str

- Similar to prefix matching

search “err”

search “fers”

Not found :(



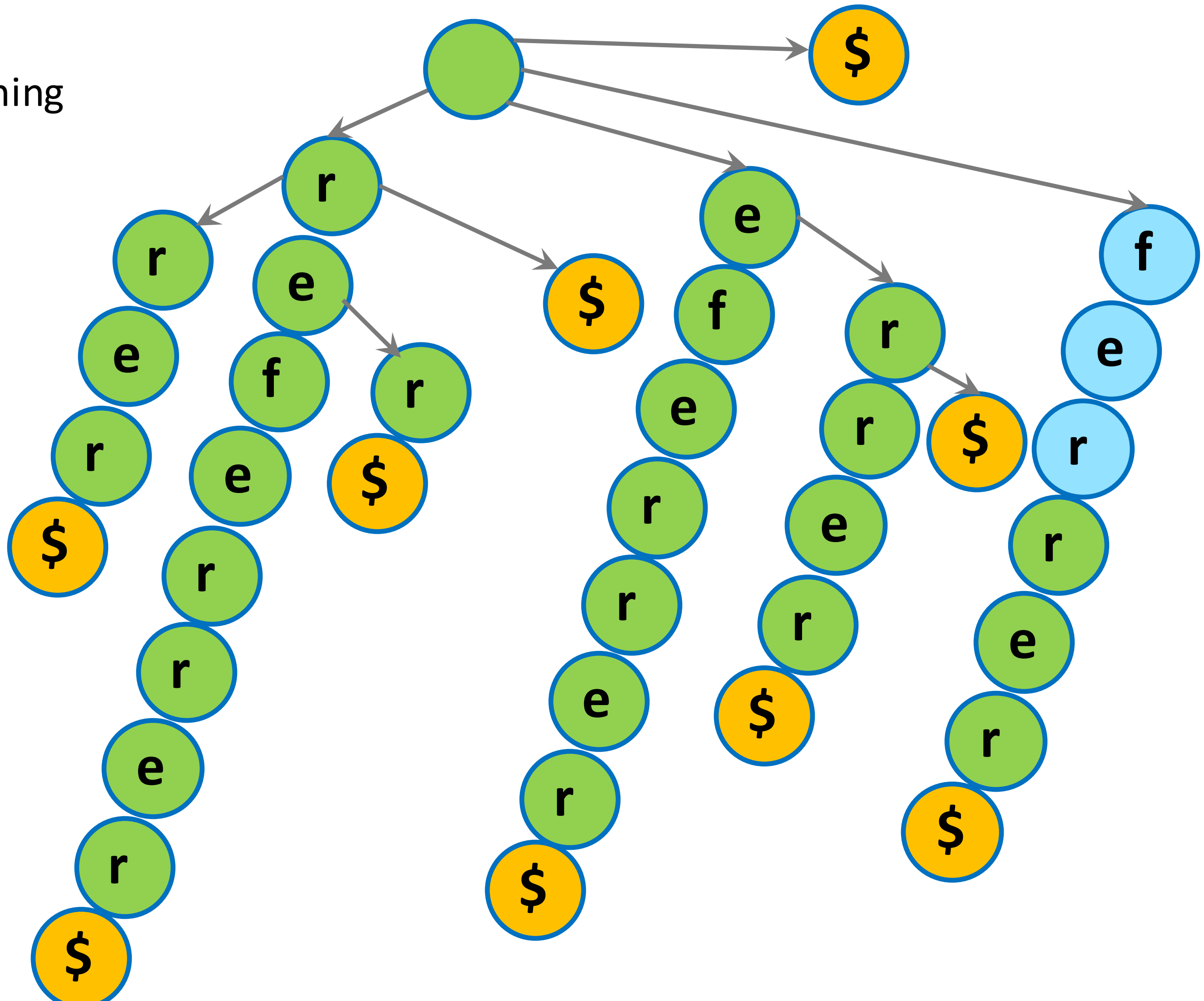
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”

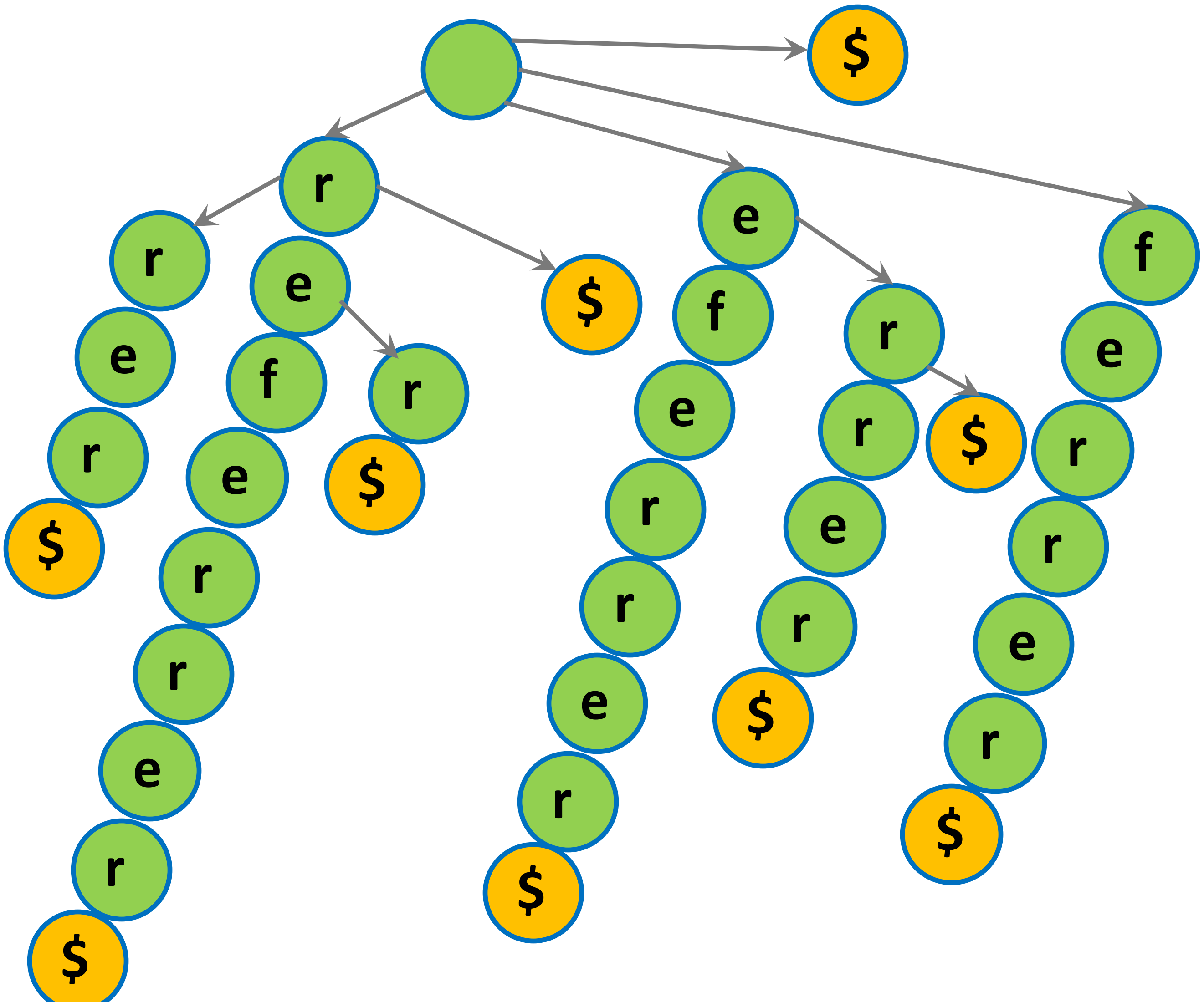


Time Complexity:

O(M) where M is the length of substring

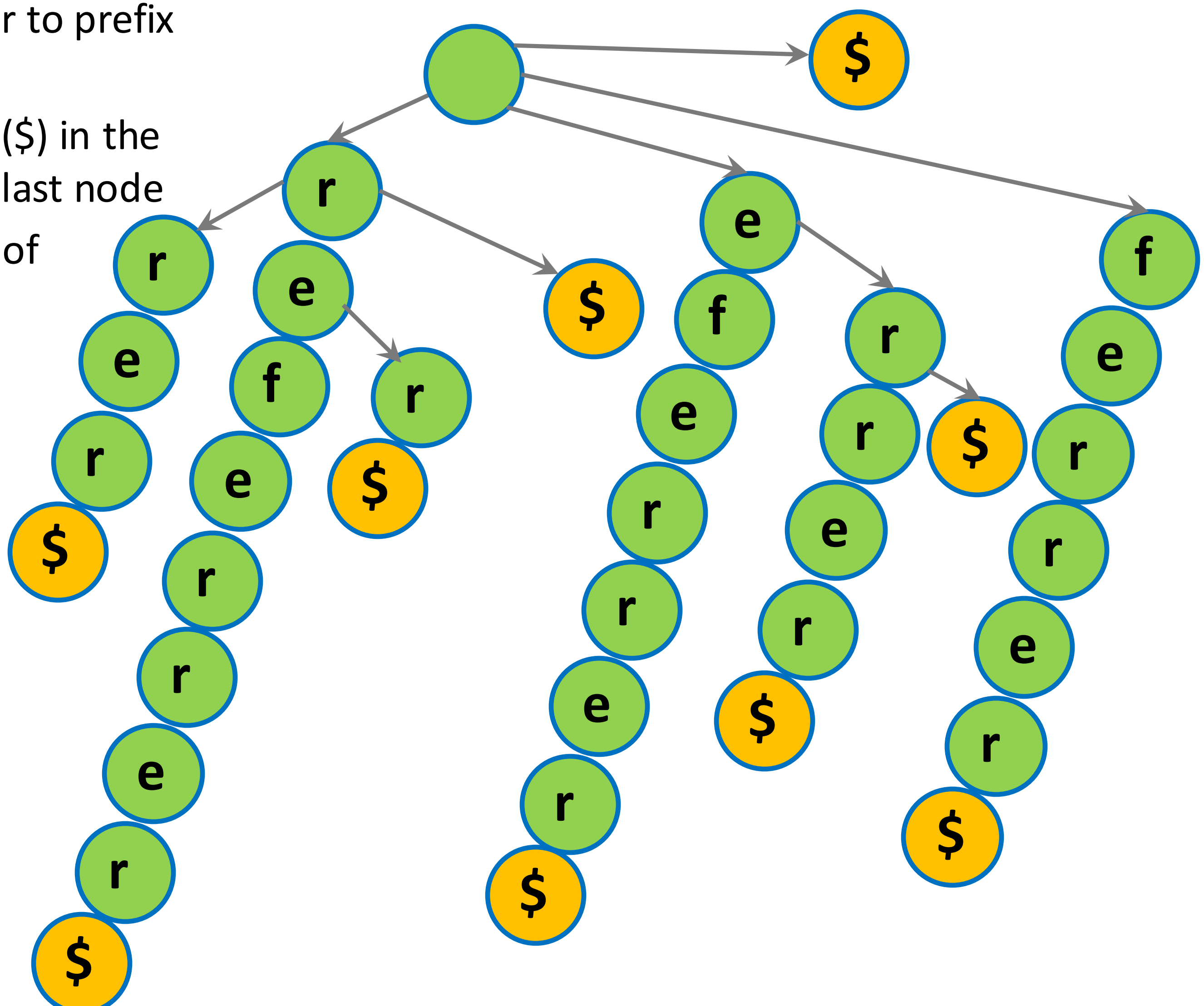
Counting # of occurences of a substring

Quiz time!



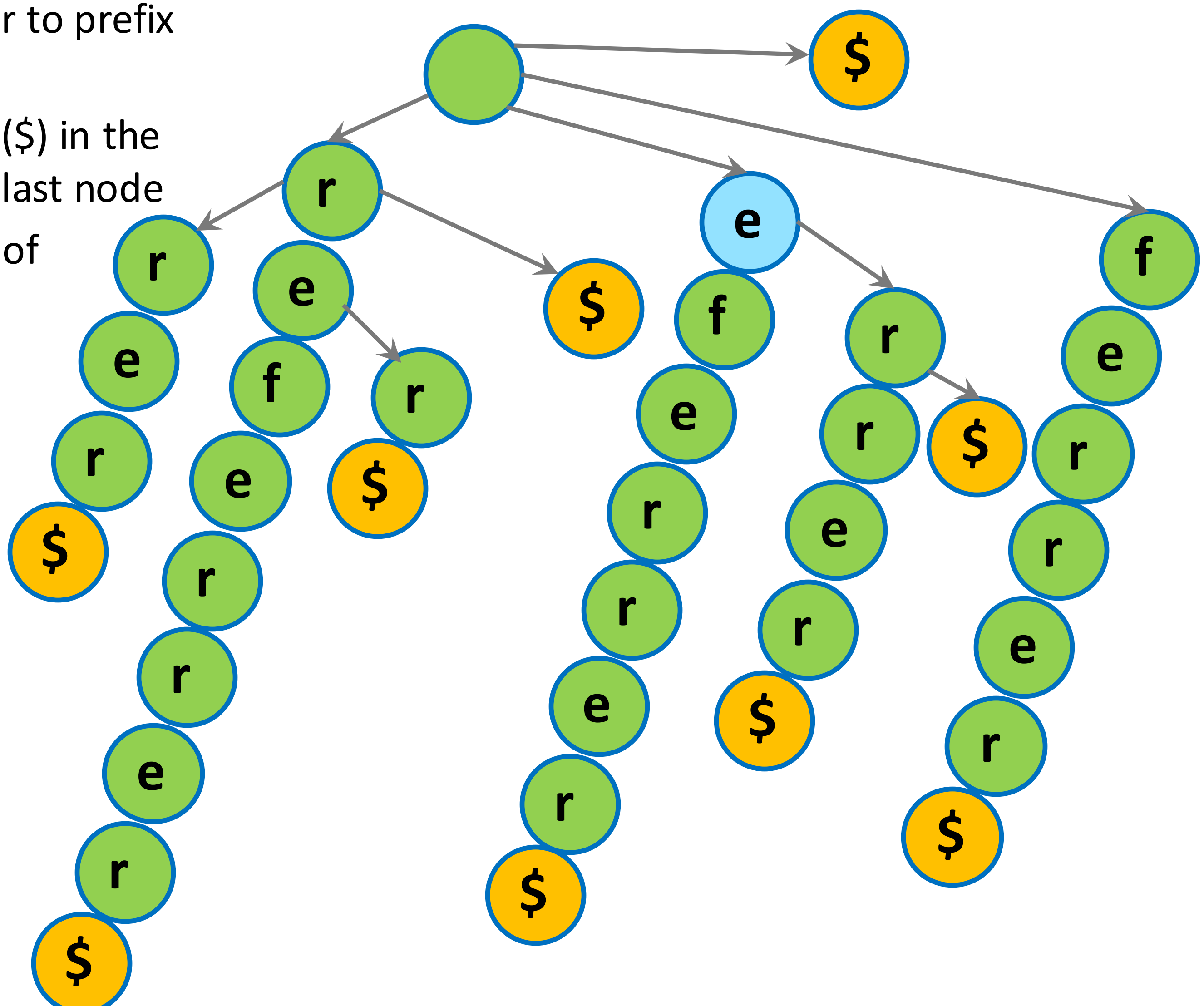
Counting # of occurences of a substring

- Follow the path similar to prefix matching
- Count # of leaf nodes ($) in the subtree rooted at the last node
- E.g Count occurences of "er"



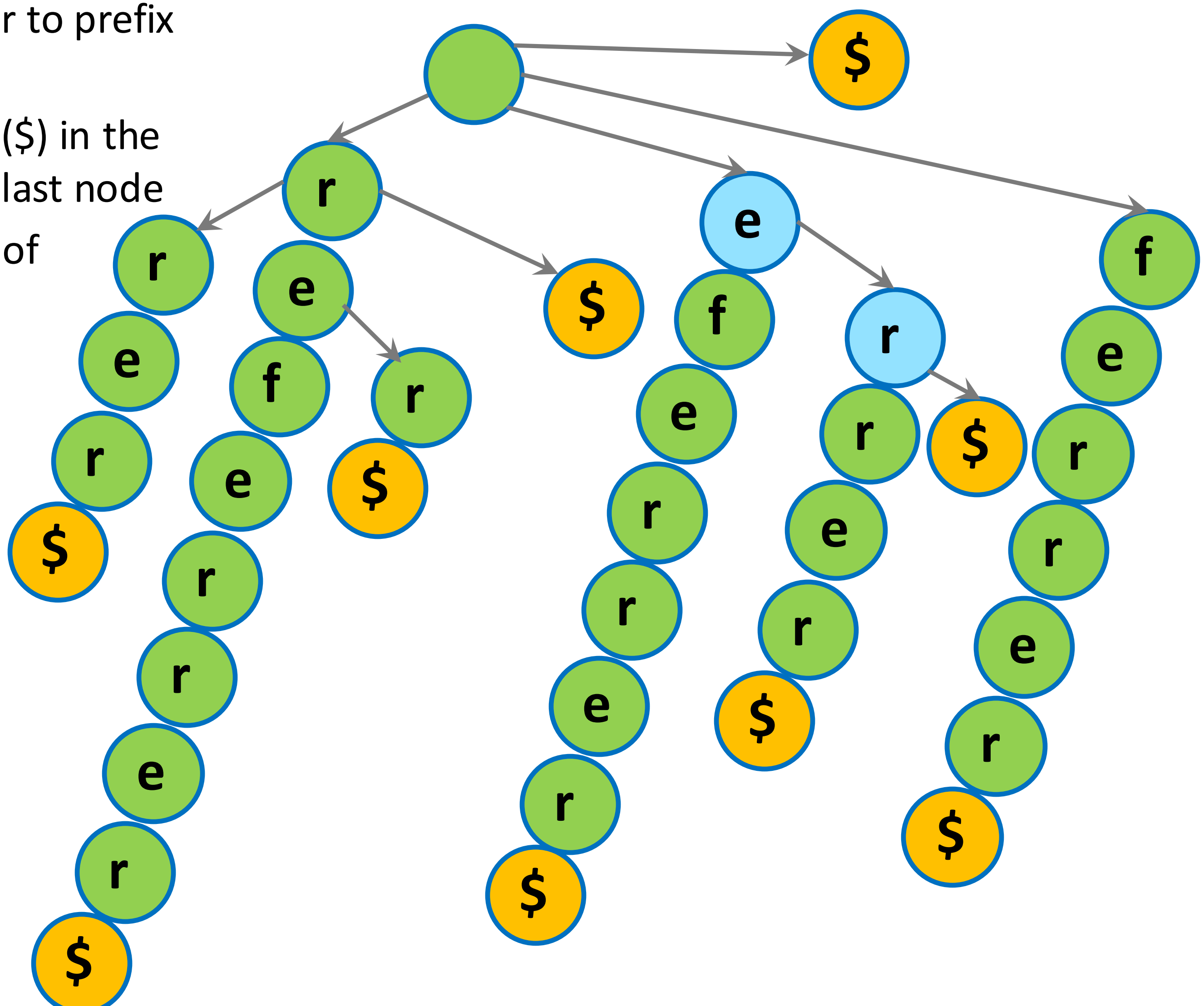
Counting # of occurences of a substring

- Follow the path similar to prefix matching
- Count # of leaf nodes ($) in the subtree rooted at the last node
- E.g Count occurences of “er”



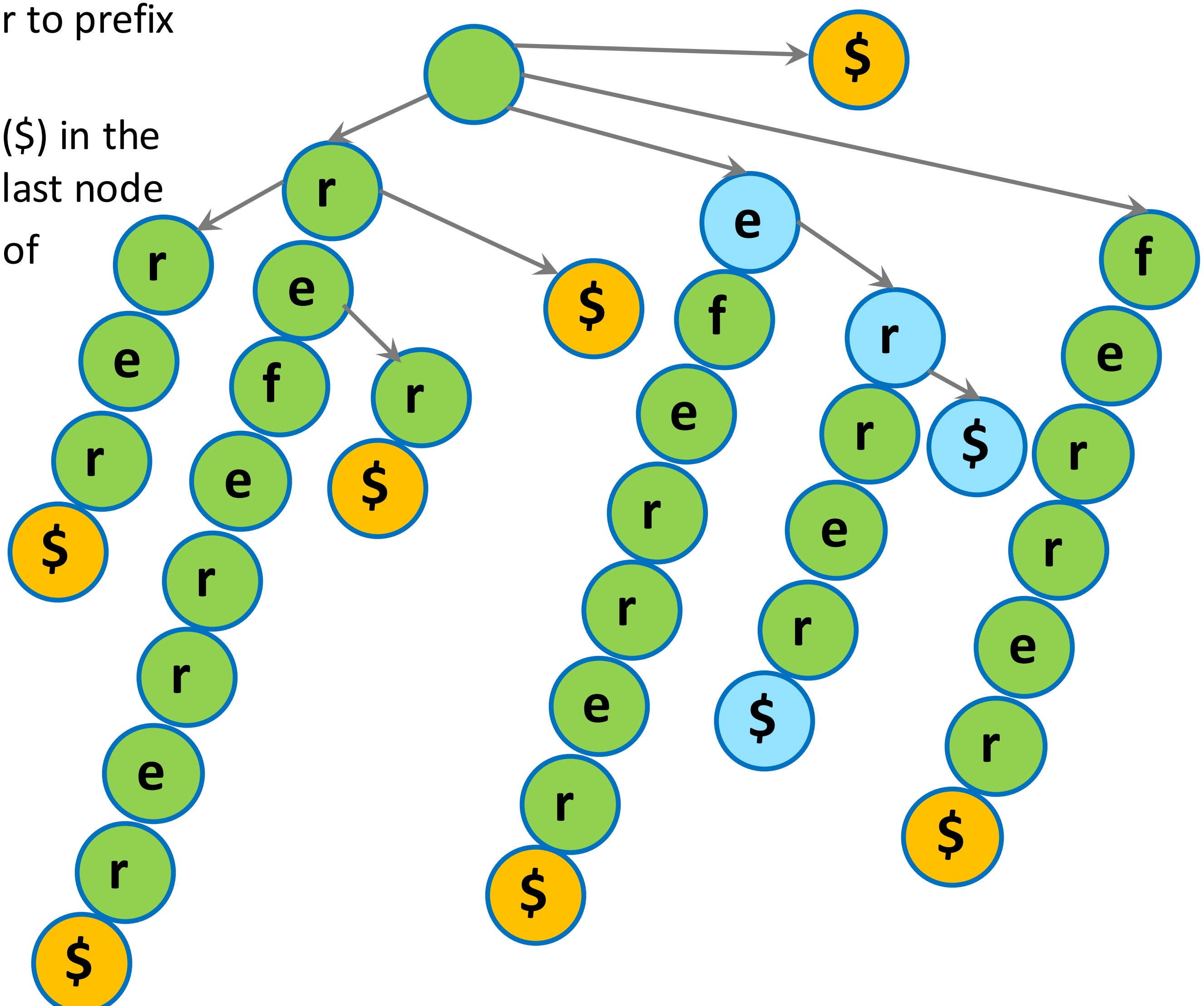
Counting # of occurences of a substring

- Follow the path similar to prefix matching
- Count # of leaf nodes ($) in the subtree rooted at the last node
- E.g Count occurences of "er"



Counting # of occurences of a substring

- Follow the path similar to prefix matching
- Count # of leaf nodes ($) in the subtree rooted at the last node
- E.g Count occurences of "er"

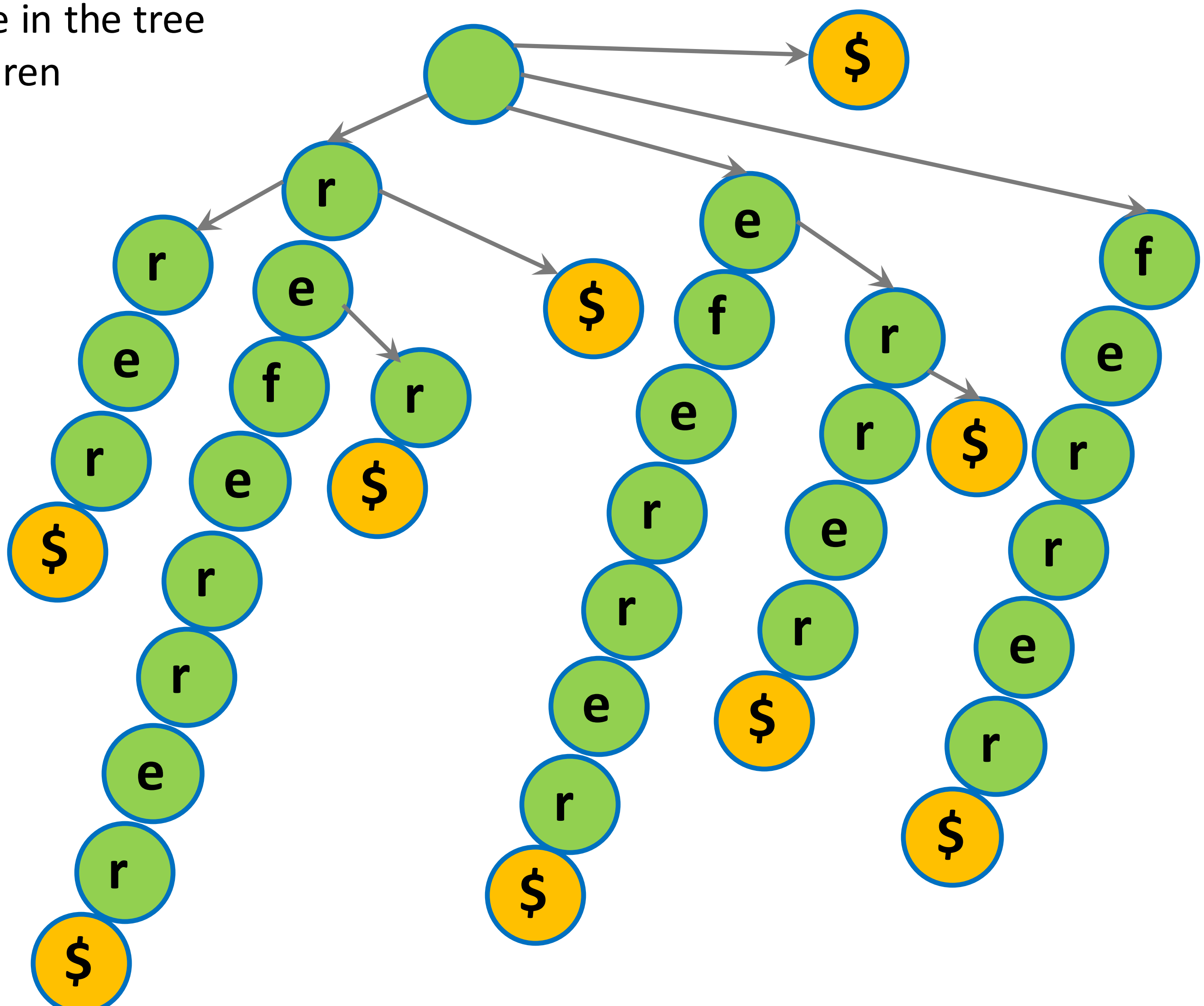


Time Complexity:

Can be done in O(M) if number of leaf nodes is maintained during construction of suffix trie

Finding longest repeated substring

- Find the deepest node in the tree with at least two children



Space complexity of suffix trie

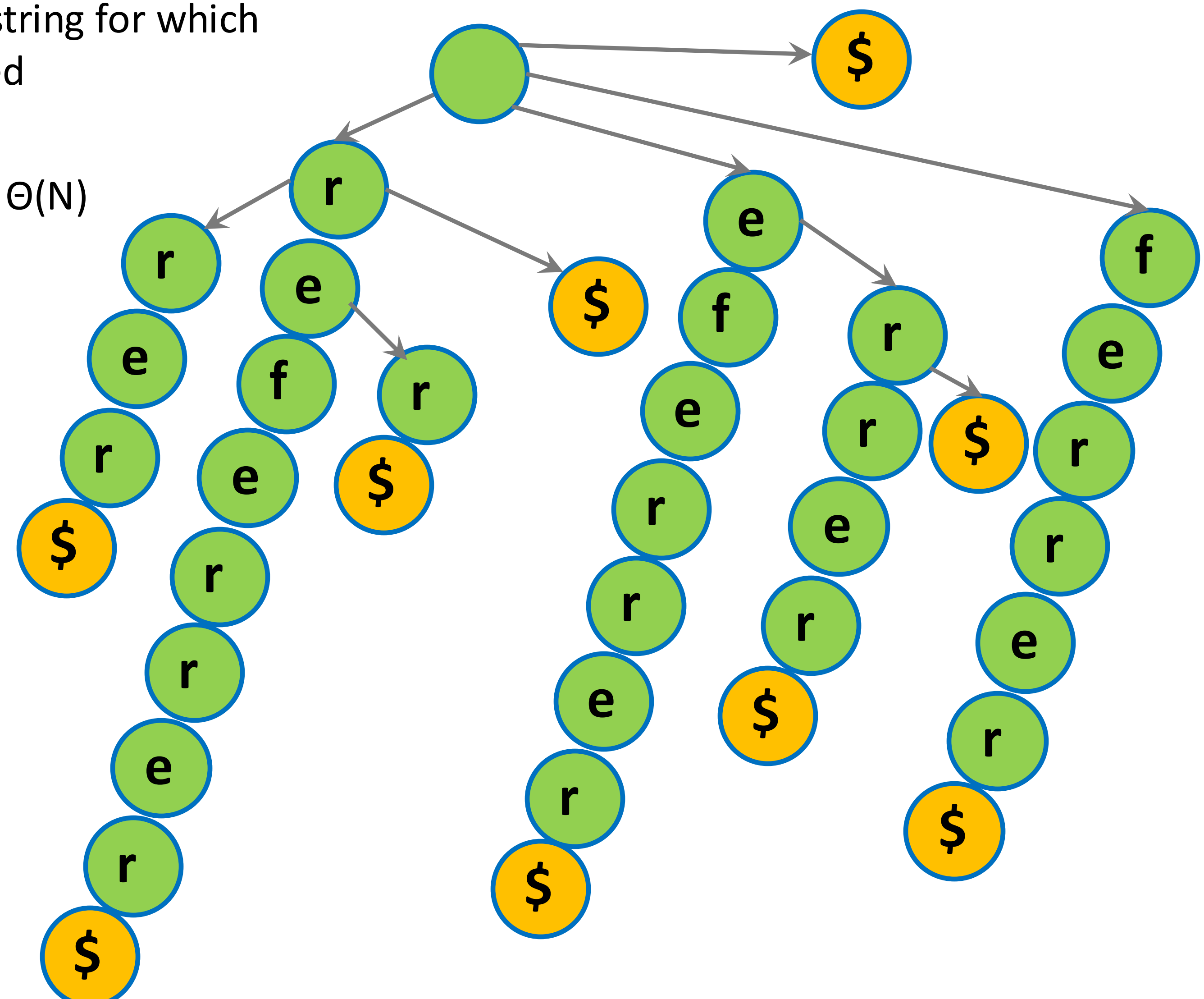
Let N be the size of the string for which a suffix trie is constructed

Space complexity?

- # number of suffixes: $\Theta(N)$
- Cost for each suffix is

linear to its size

Total space cost: $\Theta(N^2)$

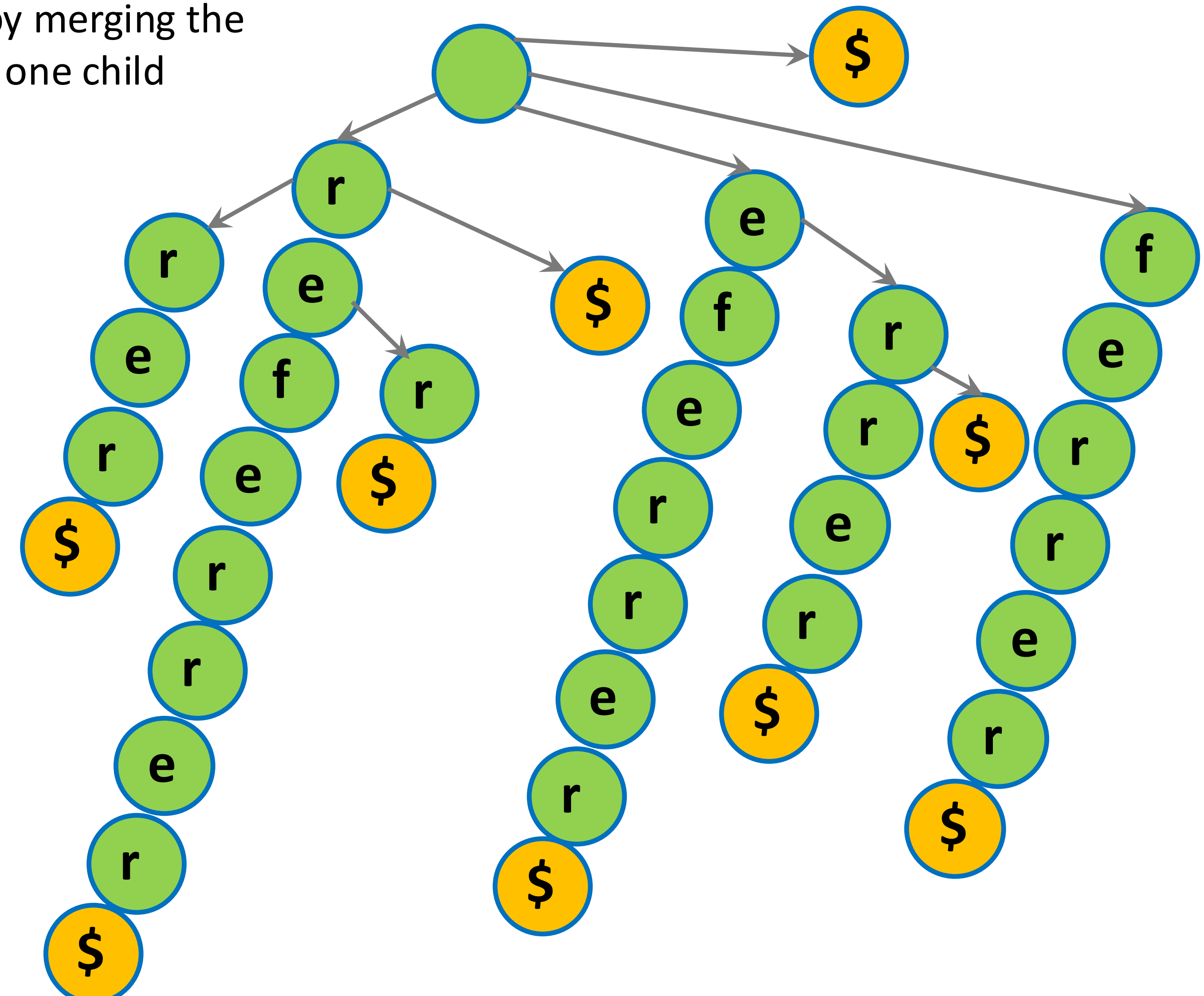


Outline

1. Trie
   A. Construction
   B. Query Processing
2. Suffix Trie
   A. Construction
   B. Query Processing
   C. Suffix Tree
3. Suffix Array
   A. Introduction
   B. Reducing Construction Cost

Suffix Tree is a compact Suffix Trie

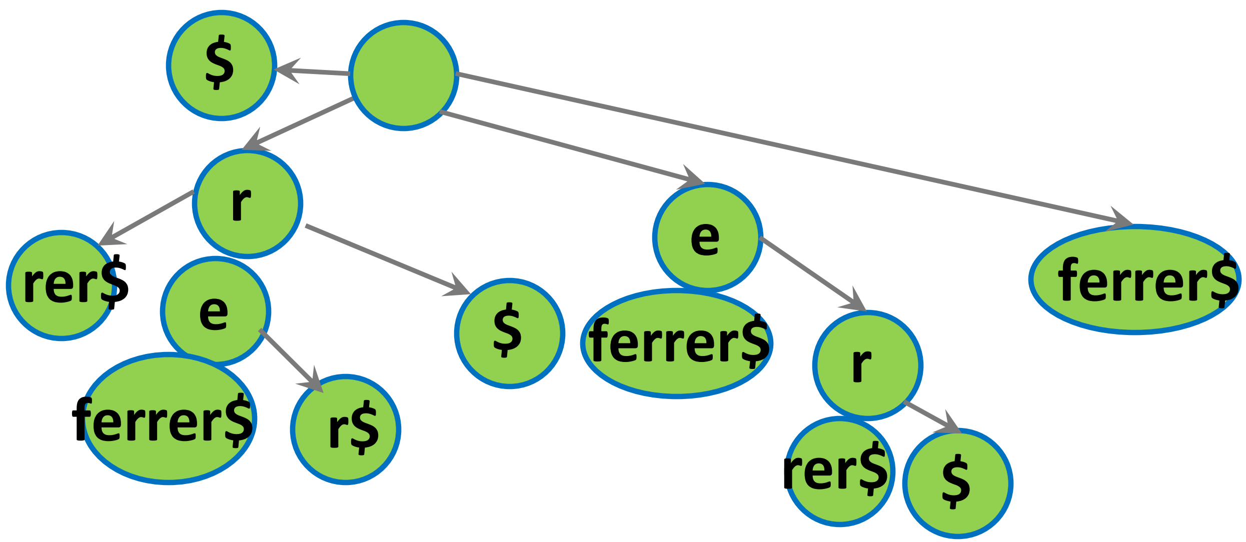
- Compress branches by merging the nodes that have only one child



Suffix Tree

- Compress branches by merging the nodes that have only one child
- But the total complexity is still the same as the same number of letters are stored

Quiz time!



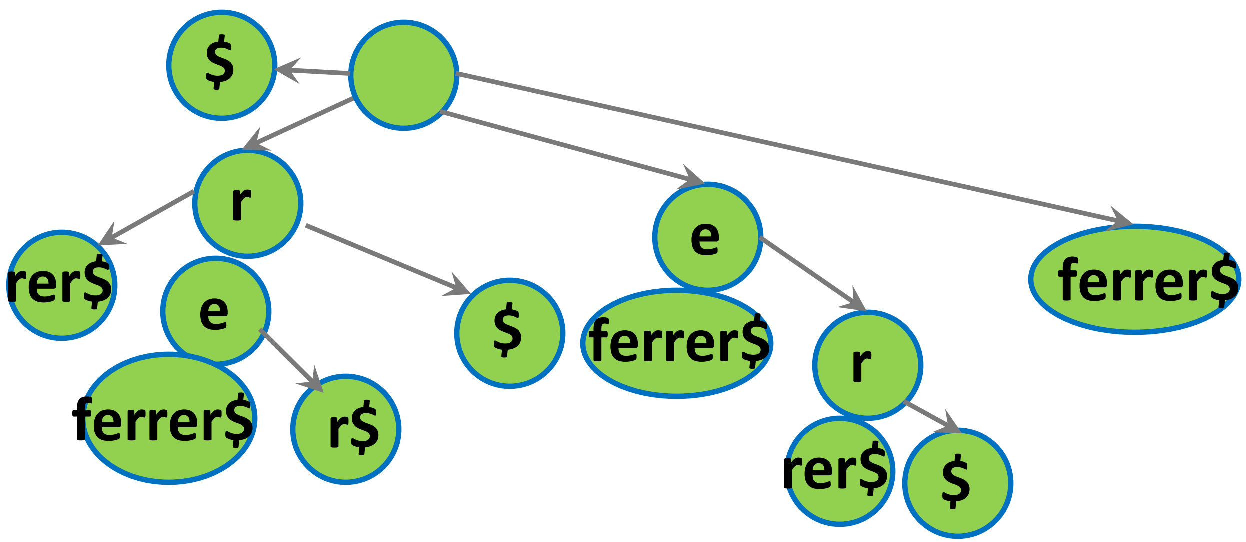
Space complexity of suffix tree

- Compress branches by merging the nodes that have only one child
- But the total complexity is still the same as the same number of letters are stored
- Replace every substring with numbers (x,y) where x is the starting index of the substring and y is its length

e.g., ferrer$ is represented as (3,7)

rer$ is represented as (6,4)

r	e	f	e	r	r	e	r	$
1	2	3	4	5	6	7	8	9

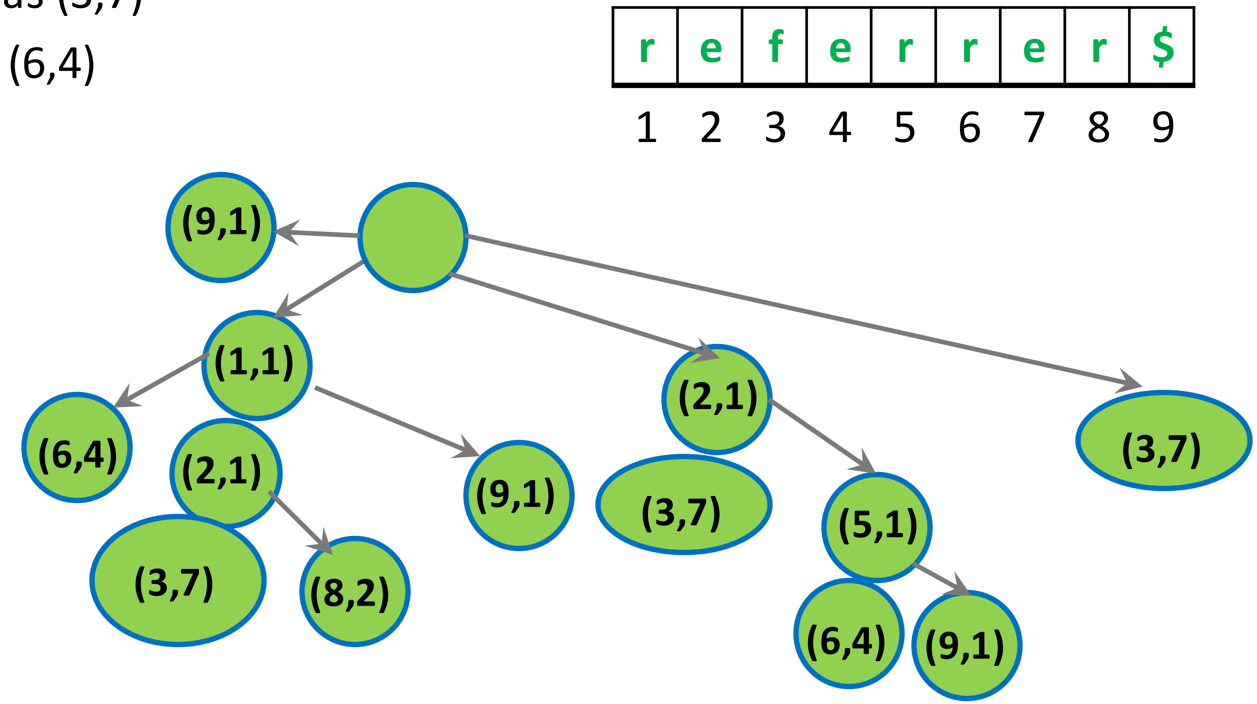


Space complexity of suffix tree

- Compress branches by merging the nodes that have only one child
- But the total complexity is still the same as the same number of letters are stored
- Replace every substring with numbers (x,y) where x is the starting index of the substring and y is its length

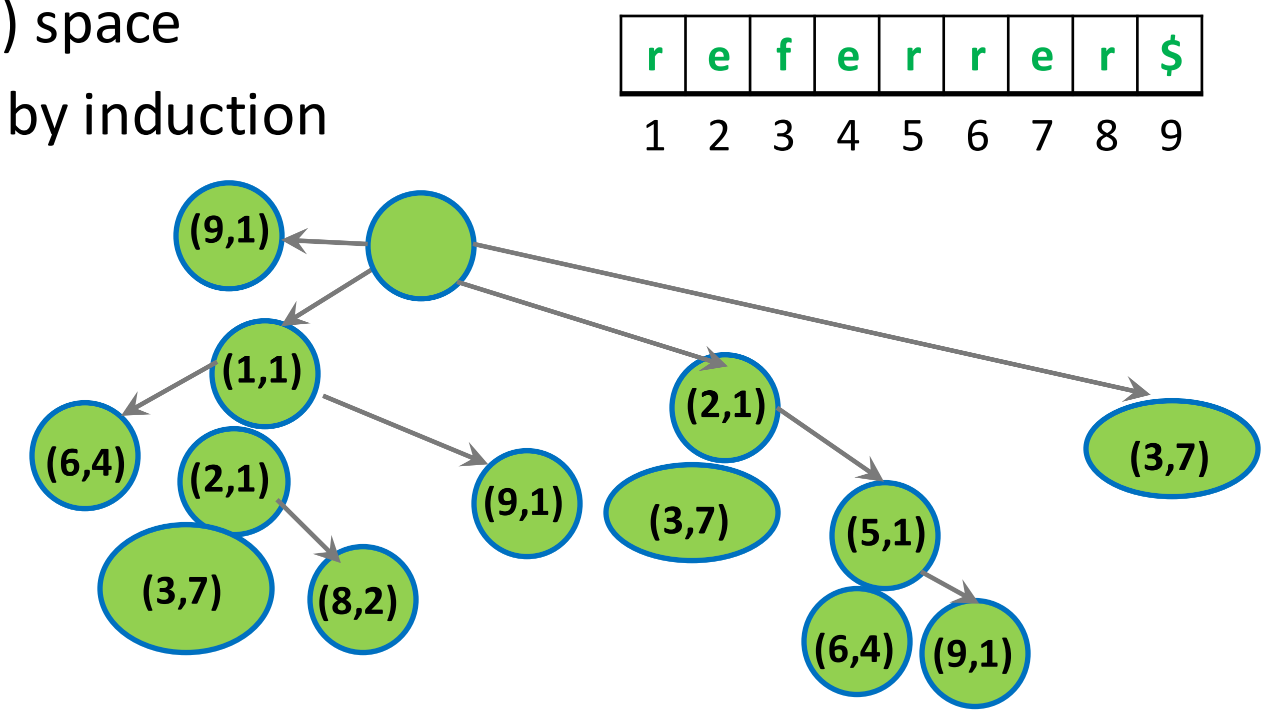
e.g., ferrer$ is represented as (3,7)

rer$ is represented as (6,4)



Space complexity of suffix tree

- Every (internal) node has at least 2 children
- There are exactly n+1 leaves
- There are at most n internal nodes
- Suffix trees are Θ(N) space
- Exercise: Prove this by induction



Time complexity of constructing suffix tree

- The algorithm described earlier inserts $\Theta(N)$ suffixes .
- Insertion cost of each suffix is linear in the size of suffix.
- Average suffix size is $\Theta(N)$.
- Compressing the trie requires traversing it, $\Theta(N^2)$.
- Thus, total time complexity of that algorithm to construct the **suffix tree** is $\Theta(N^2)$.

It is possible to construct suffix tree in $\Theta(N)$:

- In 1995 Esko Ukkonen presented a very beautiful algorithm to construct a Suffix Tree in linear time.

Outline

1. Trie
   - A. Construction
   - B. Query Processing
2. Suffix Trie
   - A. Construction
   - B. Query Processing
   - C. Suffix Tree
3. Suffix Array
   - A. Introduction
   - B. Reducing Construction Cost

Sorted Suffixes

String	M	I	S	S	I	S	S	I	P	P	I	$

1 M I S S I S S I P P I $
2 I S S I S S I P P I $
3 S S I S S I P P I $
4 S I S S I P P I $
5 I S S I P P I $
6 S S I P P I $
7 S I P P I $
8 I P P I $
9 P P I $
10 P I $
11 I $
12 $

Sort →

$
I $
I P P I $
I S S I P P I $
I S S I S S I P P I $
M I S S I S S I P P I $
P I $
P P I $
S I P P I $
S I S S I P P I $
S S I P P I $
S S I S S I P P I $

Querying on Sorted Suffixes

String	M	I	S	S	I	S	S	I	P	P	I	$

Substring search:

- Is "IPP" in the String?
  - Binary search on sorted suffices
- Let M be the number of characters in substring and N be the size of string.
- Worst-case cost of substring search is?
  - O (M log N)

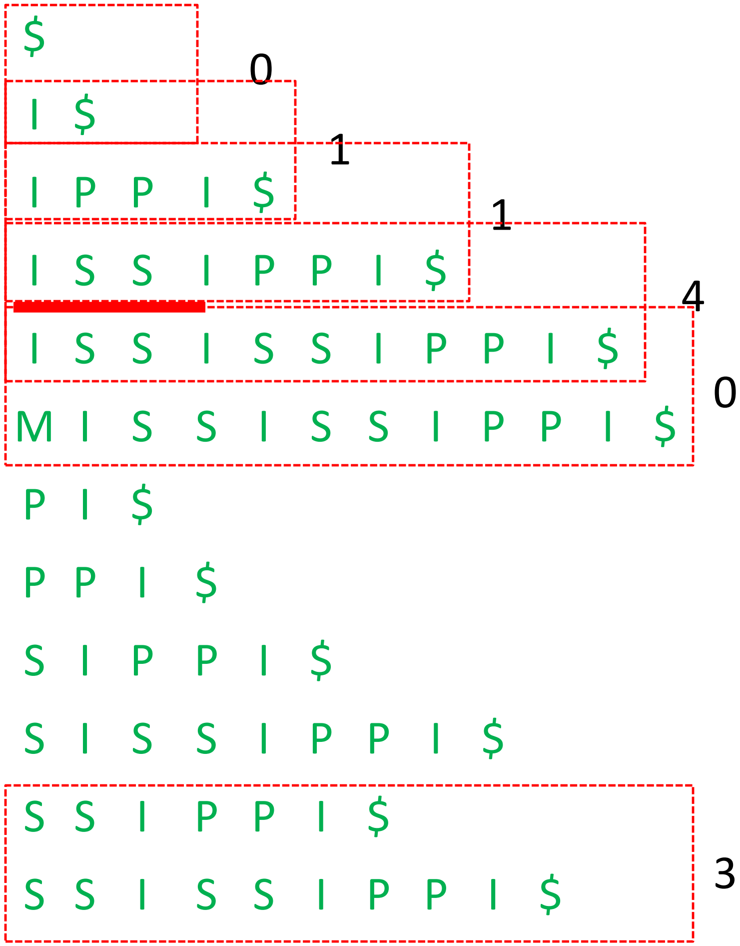
$											
I	$										
I	P	P	I	$							
I	S	S	I	P	P	I	$				
I	S	S	I	S	S	I	P	P	I	$	
M	I	S	S	I	S	S	I	P	P	I	$
P	I	$									
P	P	I	$								
S	I	P	P	I	$						
S	I	S	S	I	P	P	I	$			
S	S	I	P	P	I	$					
S	S	I	S	S	I	P	P	I	$		

Querying on Sorted Suffixes

String	**M**	**I**	**S**	**S**	**I**	**S**	**S**	**I**	**P**	**P**	**I**	**$**

Longest repeated substring:

- For each consecutive pair in sorted suffices
  - Compute the size of longest common prefix (LCP) among the pair
  - Maintain the one with the maximum size
- Scan the LCP for the maximum
- Complexity:
  - Cost of building sorted suffixes+ cost of building LCP array + O(N)



Sorted Suffixes

String	M	I	S	S	I	S	S	I	P	P	I	$

Space complexity of Sorted Suffixes:

- ○ $\Theta(N^2)$

- Can we do better?

Yes! Suffix Array reduces it to $\Theta(N)$ without losing effectiveness

$
I $
I P P I $
I S S I P P I $
I S S I S S I P P I $
M I S S I S S I P P I $
P I $
P P I $
S I P P I $
S I S S I P P I $
S S I P P I $
S S I S S I P P I $

Suffix Array

Suffix ID

index	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Suffix ID	
1	M I S S I S S I P P I $
2	I S S I S S I P P I $
3	S S I S S I P P I $
4	S I S S I P P I $
5	I S S I P P I $
6	S S I P P I $
7	S I P P I $
8	I P P I $
9	P P I $
10	P I $
11	I $
12	$

Sort

12	$
11	I $
8	I P P I $
5	I S S I P P I $
2	I S S I S S I P P I $
1	M I S S I S S I P P I $
10	P I $
9	P P I $
7	S I P P I $
4	S I S S I P P I $
6	S S I P P I $
3	S S I S S I P P I $

Suffix Array:
Only stores IDs of suffixes. The sorted suffices are shown just for illustration

Practice

What will be the suffix array of ABAB$?

Quiz time!

Outline

1. Trie
   A. Construction
   B. Query Processing
2. Suffix Trie
   A. Construction
   B. Query Processing
   C. Suffix Tree
3. Suffix Array
   A. Introduction
   B. Reducing Construction Cost

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Ranks

- Tell us the relative order of strings
- Only with respect to the chars which have been considered so far

Rank	ID	
-	1	M I S S I S S I P P I $
-	2	I S S I S S I P P I $
-	3	S S I S S I P P I $
-	4	S I S S I P P I $
-	5	I S S I P P I $
-	6	S S I P P I $
-	7	S I P P I $
-	8	I P P I $
-	9	P P I $
-	10	P I $
-	11	I $
-	12	$

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes

Rank	ID	Suffix
-	1	M I S S I S S I P P I $
-	2	I S S I S S I P P I $
-	3	S S I S S I P P I $
-	4	S I S S I P P I $
-	5	I S S I P P I $
-	6	S S I P P I $
-	7	S I P P I $
-	8	I P P I $
-	9	P P I $
-	10	P I $
-	11	I $
-	12	$

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks (pertaining to the first character)

Rank	ID	
77	1	M I S S I S S I P P I $
73	2	I S S I S S I P P I $
83	3	S S I S S I P P I $
83	4	S I S S I P P I $
73	5	I S S I P P I $
83	6	S S I P P I $
83	7	S I P P I $
73	8	I P P I $
80	9	P P I $
80	10	P I $
73	11	I $
36	12	$

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char

Rank	ID	Suffix
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	**M**	**I**	**S**	**S**	**I**	**S**	**S**	**I**	**P**	**P**	**I**	**$**

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
  - Update ranks
  - Ranks now pertain to relative order of first 2 chars

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	4	S I S S I P P I $
8	7	S I P P I $
9	3	S S I S S I P P I $
9	6	S S I P P I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
  - Update ranks
  - Ranks now pertain to relative order of first 2 chars
- Sort strings on first 4 chars, using ranks for first 2 chars

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
8	4	S I S S I P P I $
9	6	S S I P P I $
9	3	S S I S S I P P I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
  - Update ranks
  - Ranks now pertain to relative order of first 2 chars
- Sort strings on first 4 chars, using ranks for first 2 chars
  - Update ranks
  - Ranks now pertain to relative order of first 4 chars

Rank	ID												
1	12	$											
2	11	I	$										
3	8	I	P	P	I	$							
4	2	I	S	S	I	S	S	I	P	P	I	$	
4	5	I	S	S	I	P	P	I	$				
5	1	M	I	S	S	I	S	S	I	P	P	I	$
6	10	P	I	$									
7	9	P	P	I	$								
8	7	S	I	P	P	I	$						
9	4	S	I	S	S	I	P	P	I	$			
10	6	S	S	I	P	P	I	$					
11	3	S	S	I	S	S	I	P	P	I	$		

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	**M**	**I**	**S**	**S**	**I**	**S**	**S**	**I**	**P**	**P**	**I**	**$**

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
  - Update ranks
  - Ranks now pertain to relative order of first 2 chars
- Sort strings on first 4 chars, using ranks for first 2 chars
  - Update ranks
  - Ranks now pertain to relative order of first 4 chars

...

Rank	ID		
1	**12**	$	
2	**11**	I $	
3	**8**	I P P I $	
4	**5**	I S S I P P I $	
4	**2**	I S S I S S I P	P I $
5	**1**	M I S S I S S I	P P I $
6	**10**	P I $	
7	**9**	P P I $	
8	**7**	S I P P I $	
9	**4**	S I S S I P P I	$
10	**6**	S S I P P I $	
11	**3**	S S I S S I P P	I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
  - Update ranks
  - Ranks now pertain to relative order of first 2 chars
- Sort strings on first 4 chars, using ranks for first 2 chars
  - Update ranks
  - Ranks now pertain to relative order of first 4 chars

…

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	5	I S S I P P I $
5	2	I S S I S S I P P I $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	7	S I P P I $
10	4	S I S S I P P I $
11	6	S S I P P I $
12	3	S S I S S I P P I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Basic Idea:

- Generate suffixes
- Use ascii values of first chars to set up ranks
- Sort the strings on first 2 chars, using ranks for first char
    - Update ranks
    - Ranks now pertain to relative order of first 2 chars
- Sort strings on first 4 chars, using ranks for first 2 chars
    - Update ranks
    - Ranks now pertain to relative order of first 4 chars

…

Rank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	5	I S S I P P I $
5	2	I S S I S S I P P I $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	7	S I P P I $
10	4	S I S S I P P I $
11	6	S S I P P I $
12	3	S S I S S I P P I $

Constructing Suffix Array: Prefix Doubling

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

- What do we use the ranks for?
- Why do we do all these intermediate sorts?
- Naïve sorting would take N*N time just for the final sort (with radix sort)
- We need to speed up the comparisons
- What if comparisons were O(1)?
- logN sorts

Rank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	5	I S S I P P I $
5	2	I S S I S S I P P I $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	7	S I P P I $
10	4	S I S S I P P I $
11	6	S S I P P I $
12	3	S S I S S I P P I $

O(1) Comparison

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Comparing suffixes in O(1):

- Suppose already sorted on first k characters (2 in this example)
- We have ranks for first 2 characters
- Now sorting on 2k characters (4 in this example)

Observation 1:

- If current ranks are different, suffix with smaller rank is smaller (because its first k characters are smaller)
  - E.g., PPI$ < SSIP
  - Note comparison cost is O(1)

Rank	ID												
1	12	$											
2	11	I	$										
3	8	I	P	P	I	$							
4	2	I	S	S	I	S	S	I	P	P	I	$	
4	5	I	S	S	I	P	P	I	$				
5	1	M	I	S	S	I	S	S	I	P	P	I	$
6	10	P	I	$									
7	9	P	P	I	$								
8	4	S	I	S	S	I	P	P	I	$			
8	7	S	I	P	P	I	$						
9	3	S	S	I	S	S	I	P	P	I	$		
9	6	S	S	I	P	P	I	$					

O(1) Comparison

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Observation 2:

If current ranks are the same

- First k characters must be the same
- The tie is to be broken on the next k characters, e.g.,

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	4	S I S S I P P I $
8	7	S I P P I $
9	3	S S I S S I P P I $
9	6	S S I P P I $

O(1) Comparison

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Observation 2:

If current ranks are the same

- First k characters must be the same
- The tie is to be broken on the next k characters, e.g.,
  - We need to compare “SSIPPI$” and “PPI$” on the first 2 characters

Rank	ID		
1	12	$	
2	11	I $	
3	8	I P	P I $
4	2	I S	S I S S I P P I $
4	5	I S	S I P P I $
5	1	M I	S S I S S I P P I $
6	10	P I	$
7	9	P P	I $
8	4	S I	S S I P P I $
8	7	S I	P P I $
9	3	S S	I S S I P P I $
9	6	S S	I P P I $

O(1) Comparison

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Observation 2:

If current ranks are the same

- First k characters must be the same
- The tie is to be broken on the next k characters, e.g.,
  - We need to compare “SSIPPI$” and “PPI$” on the first 2 characters
  - SSIPPI$ and PPI$ are suffixes and are already ranked on first 2 characters
    - E.g., PPI$ < SSIPPI$ because its rank is smaller
    - Therefore, suffix #7< suffix #4

Rank	ID		
1	12	$	
2	11	I $	
3	8	I P	P I $
4	2	I S	S I S S I P P I $
4	5	I S	S I P P I $
5	1	M I	S S I S S I P P I $
6	10	P I	$
7	9	P P	I $
8	4	S I	S S I P P I $
8	7	S I	P P I $
9	3	S S	I S S I P P I $
9	6	S S	I P P I $

Practice

	1	2	3	4	5	6	7	8	9	10	11	12
String	**M**	**I**	**S**	**S**	**I**	**S**	**S**	**I**	**P**	**P**	**I**	**$**

- **BUT WAIT!**
- **How did we do that quickly? Surely looking up the "second half" suffixes is O(N)?**

Rank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

Practice

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Suppose we are comparing suffix with ID 2 and 5:

- We need to compare SSIPPI$ and PPI$

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

Practice

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Suppose we are comparing suffix with ID 2 and 5:

- We need to compare SSIPPI$ and PPI$
- How do we find their ranks quickly?

Rank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

Practice

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Suppose we are comparing suffix with ID 2 and 5:

- We need to compare SSIPPI$ and PPI$
- How do we find their ranks quickly?

- We want the ranks of suffixes: 2+k and 5+k
- I.e. suffixes 6 and 9

- This means we can calculate the IDs of the suffixes we want in O(1)

- Now we need to get from IDs to ranks in O(1)

Rank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

Practice

	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

Suppose we are comparing suffix with ID 2 and 5:

- We need to compare SSIPPI$ and PPI$
- How do we find their ranks quickly?

- We want the ranks of suffixes: 2+k and 5+k
- I.e. suffixes 6 and 9

- To have O(1) access to their ranks, we need an array indexed by ID which contains the ranks!

- In other words, the way the ranks are arranged on this slide is useless

Rank	ID	Suffix
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

O(1) Comparison

Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	5	4	11	9	4	10	8	3	7	6	2	1
String	**M**	**I**	**S**	**S**	**I**	**S**	**S**	**I**	**P**	**P**	**I**	**$**

Note: The greyed out oldRank array has been left for reference, but does not exist in implementation

- If we want the rank of ID i, look at Rank[i]
- Going back to our example…

oldRank	ID	Suffix
1	**12**	$
2	**11**	I $
3	**8**	I P P I $
4	**2**	I S S I S S I P P I $
4	**5**	I S S I P P I $
5	**1**	M I S S I S S I P P I $
6	**10**	P I $
7	**9**	P P I $
8	**7**	S I P P I $
9	**4**	S I S S I P P I $
10	**6**	S S I P P I $
11	**3**	S S I S S I P P I $

O(1) Comparison

Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	5	4	11	9	4	10	8	3	7	6	2	1
String	M	I	S	S	I	S	S	I	P	P	I	$

oldRank	ID											
1	12	$										
2	11	I	$									
3	8	I	P	P	I	$						
4	2	I	S	S	I	S	S	I	P	P	I	$
4	5	I	S	S	I	P	P	I	$			
5	1	M	I	S	S	I	S	S	I	P	P	I	$
6	10	P	I	$								
7	9	P	P	I	$							
8	7	S	I	P	P	I	$					
9	4	S	I	S	S	I	P	P	I	$		
10	6	S	S	I	P	P	I	$				
11	3	S	S	I	S	S	I	P	P	I	$	

Note: The greyed out oldRank array has been left for reference, but does not exist in implementation

- If we want the rank of ID i, look at Rank[i]
- Going back to our example…
- We wanted to find the second parts of suffixes 2 and 5

O(1) Comparison

Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	5	4	11	9	4	10	8	3	7	6	2	1
String	M	I	S	S	I	S	S	I	P	P	I	$

Note: The greyed out oldRank array has been left for reference, but does not exist in implementation

- If we want the rank of ID i, look at Rank[i]
- Going back to our example…

- We wanted to find the second parts of suffixes 2 and 5
- I.e. ID 6 and 9

- Rank[9] < rank[6]
- So ID 5 should come before ID 2 in the suffix array

oldRank	ID	
1	12	$
2	11	I $
3	8	I P P I $
4	2	I S S I S S I P P I $
4	5	I S S I P P I $
5	1	M I S S I S S I P P I $
6	10	P I $
7	9	P P I $
8	7	S I P P I $
9	4	S I S S I P P I $
10	6	S S I P P I $
11	3	S S I S S I P P I $

Lets do an example

ID	String	Rank
1	M	-
2	I	-
3	S	-
4	S	-
5	I	-
6	S	-
7	S	-
8	I	-
9	P	-
10	P	-
11	I	-
12	$	-

REMEMBER:
This rank array does not exist

It contains the same values as the other rank array, its just to make the algorithm easier to follow

Rank	SA	
-	1	M I S S I S S I P P I $
-	2	I S S I S S I P P I $
-	3	S S I S S I P P I $
-	4	S I S S I P P I $
-	5	I S S I P P I $
-	6	S S I P P I $
-	7	S I P P I $
-	8	I P P I $
-	9	P P I $
-	10	P I $
-	11	I $
-	12	$

Lets do an example

ID	String	Rank
1	M	-
2	I	-
3	S	-
4	S	-
5	I	-
6	S	-
7	S	-
8	I	-
9	P	-
10	P	-
11	I	-
12	$	-

Rank the first characters of each suffix

Rank	SA	
-	1	M I S S I S S I P P I $
-	2	I S S I S S I P P I $
-	3	S S I S S I P P I $
-	4	S I S S I P P I $
-	5	I S S I P P I $
-	6	S S I P P I $
-	7	S I P P I $
-	8	I P P I $
-	9	P P I $
-	10	P I $
-	11	I $
-	12	$

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Rank the first characters of each suffix using ord()

Rank	SA	
77	1	M I S S I S S I P P I $
73	2	I S S I S S I P P I $
83	3	S S I S S I P P I $
83	4	S I S S I P P I $
73	5	I S S I P P I $
83	6	S S I P P I $
83	7	S I P P I $
73	8	I P P I $
80	9	P P I $
80	10	P I $
73	11	I $
36	12	$

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Sort SA

Since we have ranks for the first character, our sort will sort the suffixes based on their first **two characters** (using the doubling trick)

Rank	SA	
77	1	M I S S I S S I P P I $
73	2	I S S I S S I P P I $
83	3	S S I S S I P P I $
83	4	S I S S I P P I $
73	5	I S S I P P I $
83	6	S S I P P I $
83	7	S I P P I $
73	8	I P P I $
80	9	P P I $
80	10	P I $
73	11	I $
36	12	$

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Sort SA by ranks

Note that this does not change the rank array, since IDs have kept the same ranks

We just rearranged the SA

Ranks still relate only to first char

Rank	SA	Suffix
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Now we update the ranks to relate to first 2 chars

Rank	SA												
36	12	$											
73	11	I	$										
73	8	I	P	P	I	$							
73	2	I	S	S	I	S	S	I	P	P	I	$	
73	5	I	S	S	I	P	P	I	$				
77	1	M	I	S	S	I	S	S	I	P	P	I	$
80	10	P	I	$									
80	9	P	P	I	$								
83	4	S	I	S	S	I	P	P	I	$			
83	7	S	I	P	P	I	$						
83	3	S	S	I	S	S	I	P	P	I	$		
83	6	S	S	I	P	P	I	$					

Lets do an example

Make an array, "Temp" to hold the new ranks

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	1
12	$	36	1

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
1
1
1
1
1
1
1
1
1
1
1
1

Rank	SA											
36	12	$										
73	11	I	$									
73	8	I	P	P	I	$						
73	2	I	S	S	I	S	S	I	P	P	I	$
73	5	I	S	S	I	P	P	I	$			
77	1	M	I	S	S	I	S	S	I	P	P	I	$
80	10	P	I	$								
80	9	P	P	I	$							
83	4	S	I	S	S	I	P	P	I	$		
83	7	S	I	P	P	I	$					
83	3	S	S	I	S	S	I	P	P	I	$	
83	6	S	S	I	P	P	I	$				

For each pair of adjacent suffixes, compare them

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	1
12	$	36	1

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

If they have different ranks already, then the second suffix is certainly larger

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Set Temp[11] to Temp[12]+1

Rank	SA	
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

If they have the same rank

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Rank	SA	Suffix
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

We need to use the O(1) trick

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Rank	SA			
36	12	$		
73	11	I	$	
73	8	I	P	P I $
73	2	I	S	S I S S I P P I $
73	5	I	S	S I P P I $
77	1	M	I	S S I S S I P P I $
80	10	P	I	$
80	9	P	P	I $
83	4	S	I	S S I P P I $
83	7	S	I	P P I $
83	3	S	S	I S S I P P I $
83	6	S	S	I P P I $

First characters have the same rank, so compare the suffixes which start with next characters

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	1
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

$ vs PPI$ (ID 11+1 and 8+1)

Rank	SA			
36	12	$		
73	11	I	$	
73	8	I	P	P I $
73	2	I	S	S I S S I P P I $
73	5	I	S	S I P P I $
77	1	M	I	S S I S S I P P I $
80	10	P	I	$
80	9	P	P	I $
83	4	S	I	S S I P P I $
83	7	S	I	P P I $
83	3	S	S	I S S I P P I $
83	6	S	S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
1
1
1
1
1
1
1
1
1
1
2
1

Rank	SA		
36	12	$	
73	11	I	$
73	8	I	P P I $
73	2	I	S S I S S I P P I $
73	5	I	S S I P P I $
77	1	M	I S S I S S I P P I $
80	10	P	I $
80	9	P	P I $
83	4	S	I S S I P P I $
83	7	S	I P P I $
83	3	S	S I S S I P P I $
83	6	S	S I P P I $

12 has lower rank than 9, so 11 should have lower rank than 8

Lets do an example

Set Temp[8] = Temp[11] + 1

ID	String	Rank	Temp
1	M	77	1
2	I	73	1
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	3
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Rank	SA	Suffix
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

Lets do an example

ID	String	Rank	Temp
1	M	77	1
2	I	73	4
3	S	83	1
4	S	83	1
5	I	73	1
6	S	83	1
7	S	83	1
8	I	73	3
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Continue in this way

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
1
4
1
1
4
1
1
3
1
1
2
1

Continue in this way

Rank	SA	
36	12	$
73	11	I $
73	8	I P P I $
73	2	I S S I S S I P P I $
73	5	I S S I P P I $
77	1	M I S S I S S I P P I $
80	10	P I $
80	9	P P I $
83	4	S I S S I P P I $
83	7	S I P P I $
83	3	S S I S S I P P I $
83	6	S S I P P I $

Lets do an example

ID	String	Rank	Temp
1	M	77	5
2	I	73	4
3	S	83	1
4	S	83	1
5	I	73	4
6	S	83	1
7	S	83	1
8	I	73	3
9	P	80	1
10	P	80	1
11	I	73	2
12	$	36	1

Continue in this way

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
5
4
1
1
4
1
1
3
1
6
2
1

Continue in this way

Rank	SA												
36	12	$											
73	11	I	$										
73	8	I	P	P	I	$							
73	2	I	S	S	I	S	S	I	P	P	I	$	
73	5	I	S	S	I	P	P	I	$				
77	1	M	I	S	S	I	S	S	I	P	P	I	$
80	10	P	I	$									
80	9	P	P	I	$								
83	4	S	I	S	S	I	P	P	I	$			
83	7	S	I	P	P	I	$						
83	3	S	S	I	S	S	I	P	P	I	$		
83	6	S	S	I	P	P	I	$					

Lets do an example

ID	String	Rank	Temp
1	M	77	5
2	I	73	4
3	S	83	1
4	S	83	1
5	I	73	4
6	S	83	1
7	S	83	1
8	I	73	3
9	P	80	7
10	P	80	6
11	I	73	2
12	$	36	1

Continue in this way

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank	Temp
1	M	77	5
2	I	73	4
3	S	83	1
4	S	83	8
5	I	73	4
6	S	83	1
7	S	83	1
8	I	73	3
9	P	80	7
10	P	80	6
11	I	73	2
12	$	36	1

Continue in this way

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
5
4
1
8
4
1
8
3
7
6
2
1

Continue in this way

Rank	SA												
36	12	$											
73	11	I	$										
73	8	I	P	P	I	$							
73	2	I	S	S	I	S	S	I	P	P	I	$	
73	5	I	S	S	I	P	P	I	$				
77	1	M	I	S	S	I	S	S	I	P	P	I	$
80	10	P	I	$									
80	9	P	P	I	$								
83	4	S	I	S	S	I	P	P	I	$			
83	7	S	I	P	P	I	$						
83	3	S	S	I	S	S	I	P	P	I	$		
83	6	S	S	I	P	P	I	$					

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
5
4
9
8
4
1
8
3
7
6
2
1

Continue in this way

Rank	SA												
36	12	$											
73	11	I	$										
73	8	I	P	P	I	$							
73	2	I	S	S	I	S	S	I	P	P	I	$	
73	5	I	S	S	I	P	P	I	$				
77	1	M	I	S	S	I	S	S	I	P	P	I	$
80	10	P	I	$									
80	9	P	P	I	$								
83	4	S	I	S	S	I	P	P	I	$			
83	7	S	I	P	P	I	$						
83	3	S	S	I	S	S	I	P	P	I	$		
83	6	S	S	I	P	P	I	$					

Lets do an example

ID	String	Rank	Temp
1	M	77	5
2	I	73	4
3	S	83	9
4	S	83	8
5	I	73	4
6	S	83	9
7	S	83	8
8	I	73	3
9	P	80	7
10	P	80	6
11	I	73	2
12	$	36	1

Continue in this way

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

Lets do an example

ID	String	Rank
1	M	77
2	I	73
3	S	83
4	S	83
5	I	73
6	S	83
7	S	83
8	I	73
9	P	80
10	P	80
11	I	73
12	$	36

Temp
5
4
9
8
4
9
8
3
7
6
2
1

Rank	SA		
36	12	$	
73	11	I $	
73	8	I P	P I $
73	2	I S	S I S S I P P I $
73	5	I S	S I P P I $
77	1	M I	S S I S S I P P I $
80	10	P I	$
80	9	P P	I $
83	4	S I	S S I P P I $
83	7	S I	P P I $
83	3	S S	I S S I P P I $
83	6	S S	I P P I $

This is our new Rank array, so overwrite the old one

Lets do an example

ID	String	Rank
1	M	5
2	I	4
3	S	9
4	S	8
5	I	4
6	S	9
7	S	8
8	I	3
9	P	7
10	P	6
11	I	2
12	$	1

Temp
5
4
9
8
4
9
8
3
7
6
2
1

Rank	SA			
1	12	$		
2	11	I	$	
3	8	I	P	P I $
4	2	I	S	S I S S I P P I $
4	5	I	S	S I P P I $
5	1	M	I	S S I S S I P P I $
6	10	P	I	$
7	9	P	P	I $
8	4	S	I	S S I P P I $
8	7	S	I	P P I $
9	3	S	S	I S S I P P I $
9	6	S	S	I P P I $

This is our new Rank array, so overwrite the old one

Lets do an example

ID	String	Rank
1	M	5
2	I	4
3	S	9
4	S	8
5	I	4
6	S	9
7	S	8
8	I	3
9	P	7
10	P	6
11	I	2
12	$	1

Rank	SA												
1	12	$											
2	11	I	$										
3	8	I	P	P	I	$							
4	2	I	S	S	I	S	S	I	P	P	I	$	
4	5	I	S	S	I	P	P	I	$				
5	1	M	I	S	S	I	S	S	I	P	P	I	$
6	10	P	I	$									
7	9	P	P	I	$								
8	4	S	I	S	S	I	P	P	I	$			
8	7	S	I	P	P	I	$						
9	3	S	S	I	S	S	I	P	P	I	$		
9	6	S	S	I	P	P	I	$					

Now we have ranks for 2 characters, we can sort on 4 characters

Lets do an example

ID	String	Rank
1	M	5
2	I	4
3	S	11
4	S	9
5	I	4
6	S	10
7	S	8
8	I	3
9	P	7
10	P	6
11	I	2
12	$	1

Rank	SA												
1	12	$											
2	11	I	$										
3	8	I	P	P	I	$							
4	2	I	S	S	I	S	S	I	P	P	I	$	
4	5	I	S	S	I	P	P	I	$				
5	1	M	I	S	S	I	S	S	I	P	P	I	$
6	10	P	I	$									
7	9	P	P	I	$								
8	7	S	I	P	P	I	$						
9	4	S	I	S	S	I	P	P	I	$			
10	6	S	S	I	P	P	I	$					
11	3	S	S	I	S	S	I	P	P	I	$		

Lets do an example

ID	String	Rank
1	M	6
2	I	5
3	S	12
4	S	10
5	I	4
6	S	11
7	S	9
8	I	3
9	P	8
10	P	7
11	I	2
12	$	1

Rank	SA	
1	12	$
2	11	I $
3	8	I P P I $
4	5	I S S I P P I $
5	2	I S S I S S I P P I $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	7	S I P P I $
10	4	S I S S I P P I $
11	6	S S I P P I $
12	3	S S I S S I P P I $

Lets do an example

ID	String	Rank
1	M	6
2	I	5
3	S	12
4	S	10
5	I	4
6	S	11
7	S	9
8	I	3
9	P	8
10	P	7
11	I	2
12	$	1

Rank	SA	
1	12	$
2	11	I $
3	8	I P P I $
4	5	I S S I P P I $
5	2	I S S I S S I P P I $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	7	S I P P I $
10	4	S I S S I P P I $
11	6	S S I P P I $
12	3	S S I S S I P P I $

Suffix arrays

- Prefix doubling allows construction in Nlog(N) time
- The O(1) comparison idea is very powerful

- Linear time construction algorithms exist for suffix array
- In order to match the speed of a suffix tree, LCP array is necessary

- Suffix arrays are more compact than suffix trees (no links)
- Suffix arrays are localised in memory

Reading

- Course Notes: Chapters 11 and 12
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Chapter 18
- For a more advanced treatment of trie and suffix trees: Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press. Book available in the library!

Summary

Take home message

- Tries, suffix trees and suffix arrays provide efficient text search and pattern matching (typically linear in number of characters in string)
- Linear time construction for both is possible, but beyond the scope of this unit

Things to do (this list is not exhaustive)

- Implement tries, suffix trees and suffix arrays and run various pattern matching queries

Coming Up Next

- Search Trees