Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 6: Retrieval Data Structures for Strings

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Recommended readings

- Unit notes (Chapters 9&10)
- Cormen et al. “Introduction to Algorithms" (Chapter 18)
- For a more advanced treatment of Trie and suffix trees: Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press. (Chapter 5) - Book available in the library!

Outline

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

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

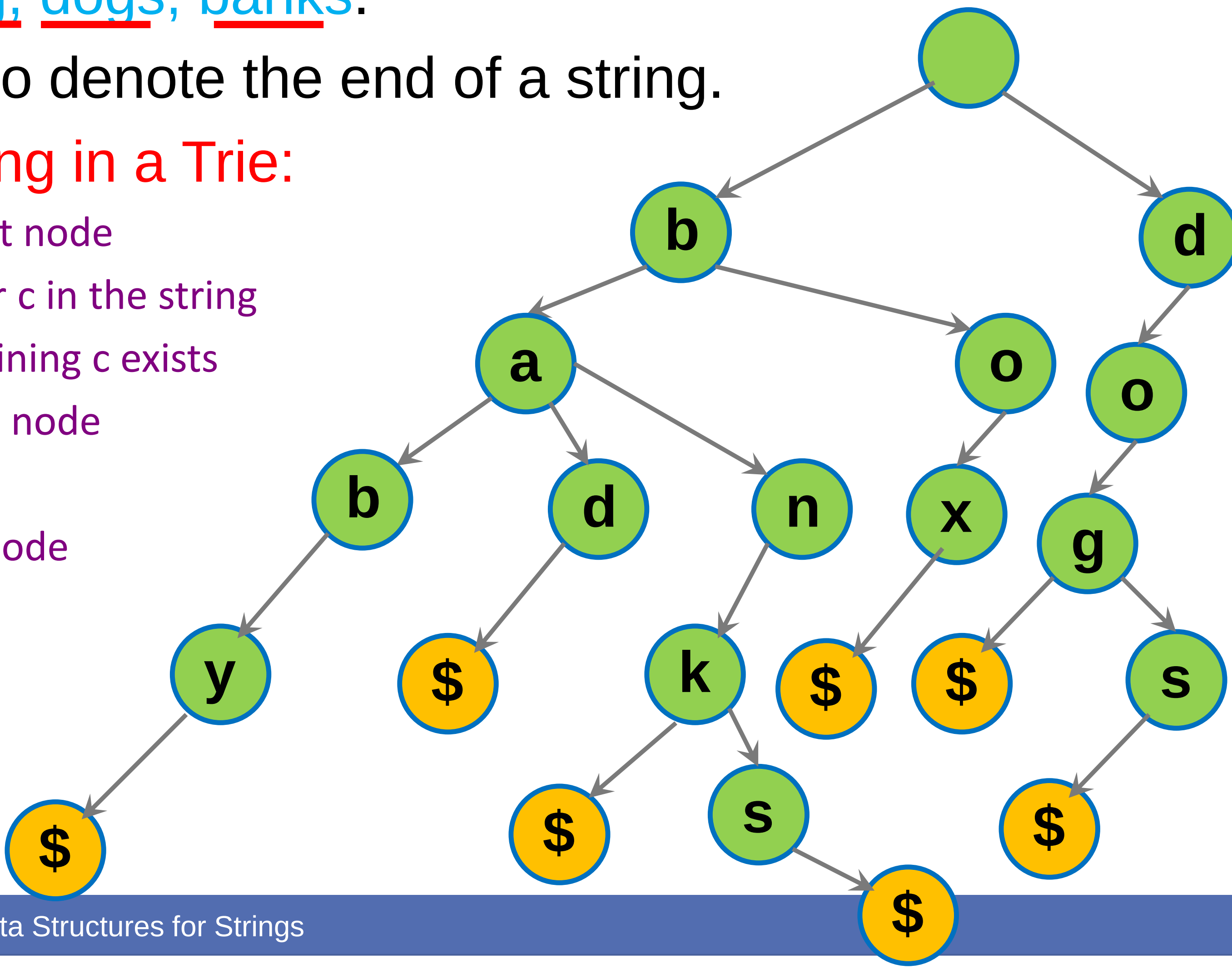
Trie

- ReTRIEval tree = Trie
- Often pronounced as ‘Try'.
- Trie is an N-way (or multi-way) tree, where N is the size of the alphabet
  - E.g., N=2 for binary
  - N = 26 for English letters
  - N = 4 for DNA
- 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.

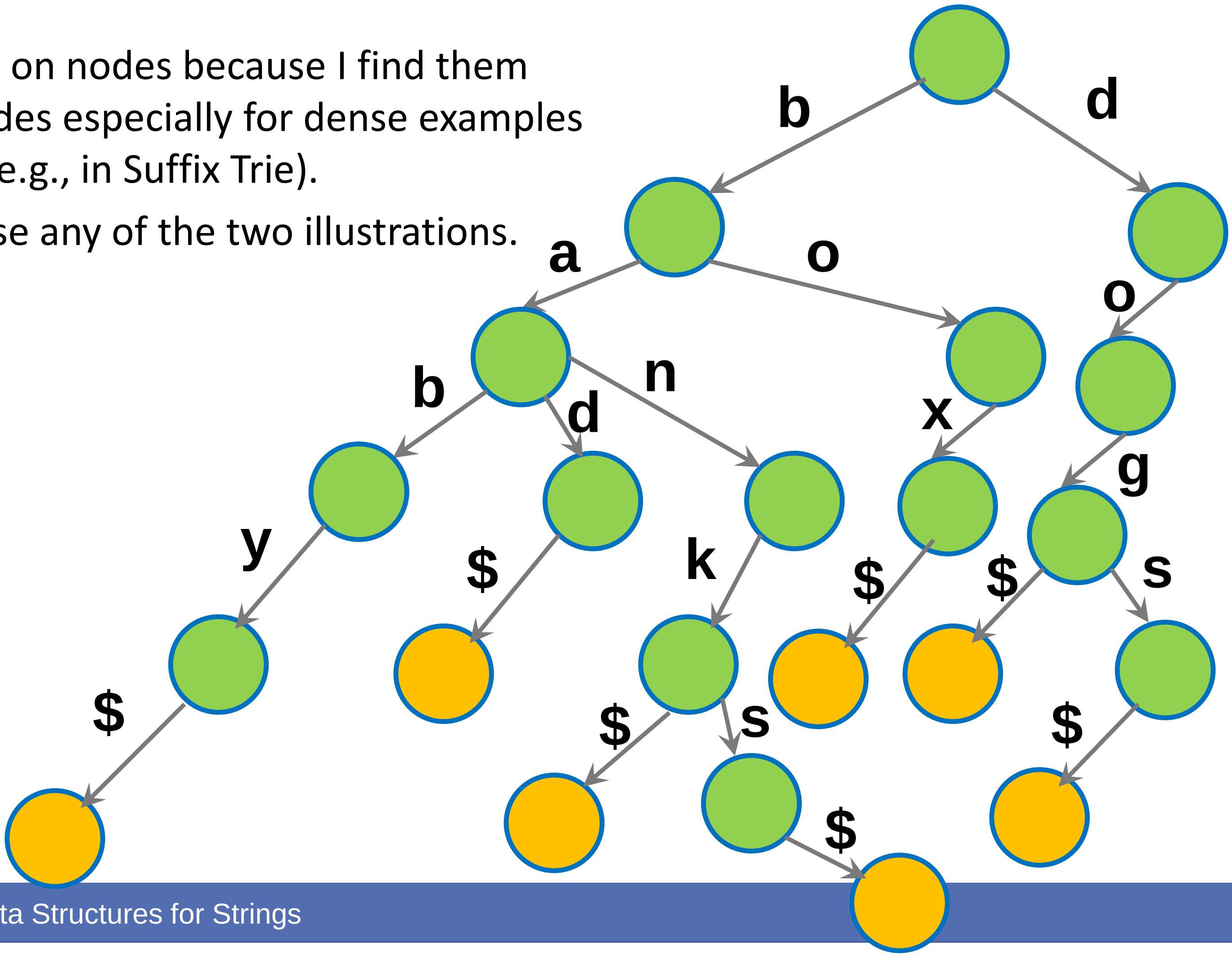


Inserting a string in a Trie:

- 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 I find them clearer in lecture slides especially for dense examples later in the lecture (e.g., in Suffix Trie).
- In exams, you can use any of the two illustrations.



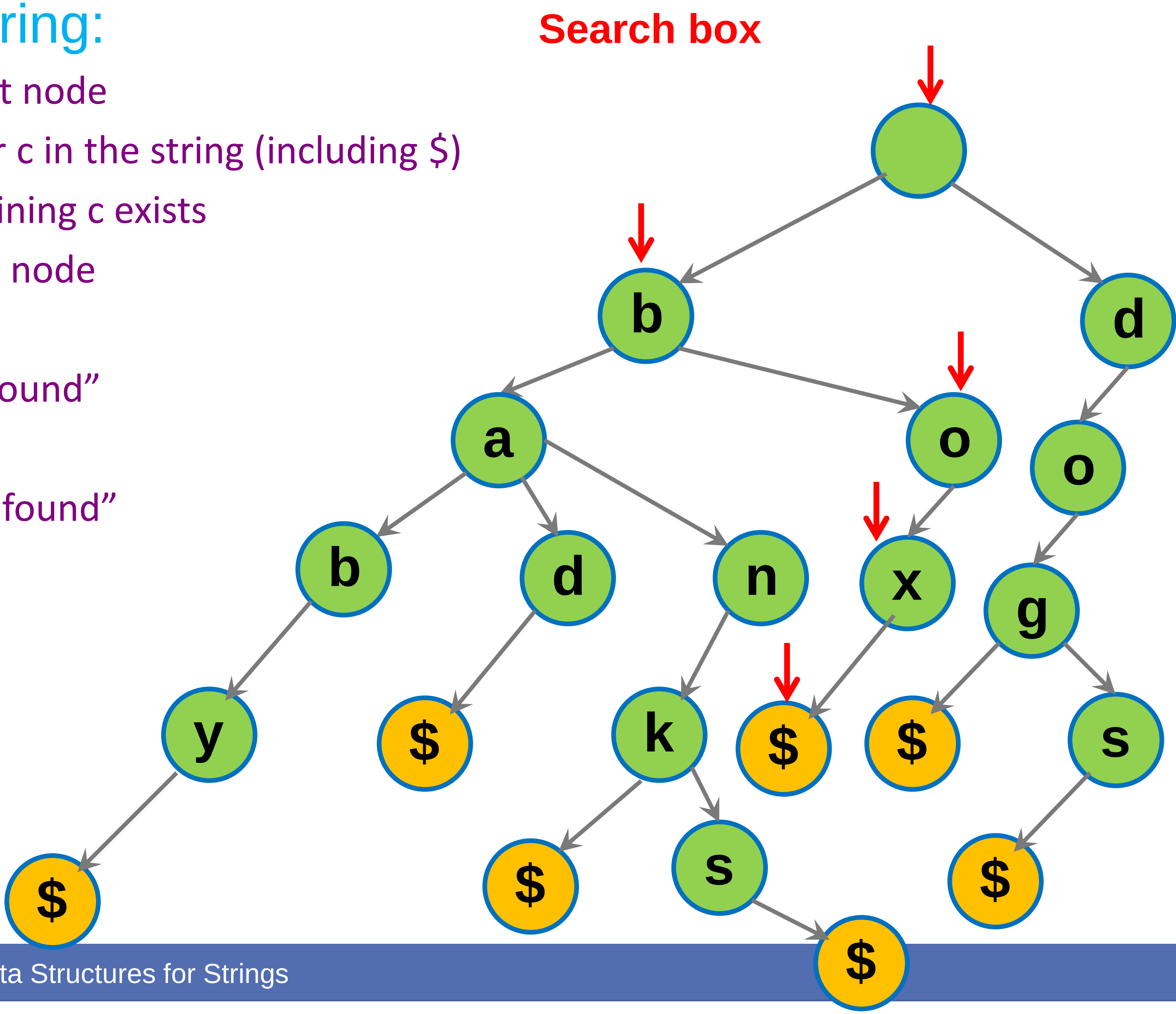
Outline

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

Trie Example: Search

Searching a string:

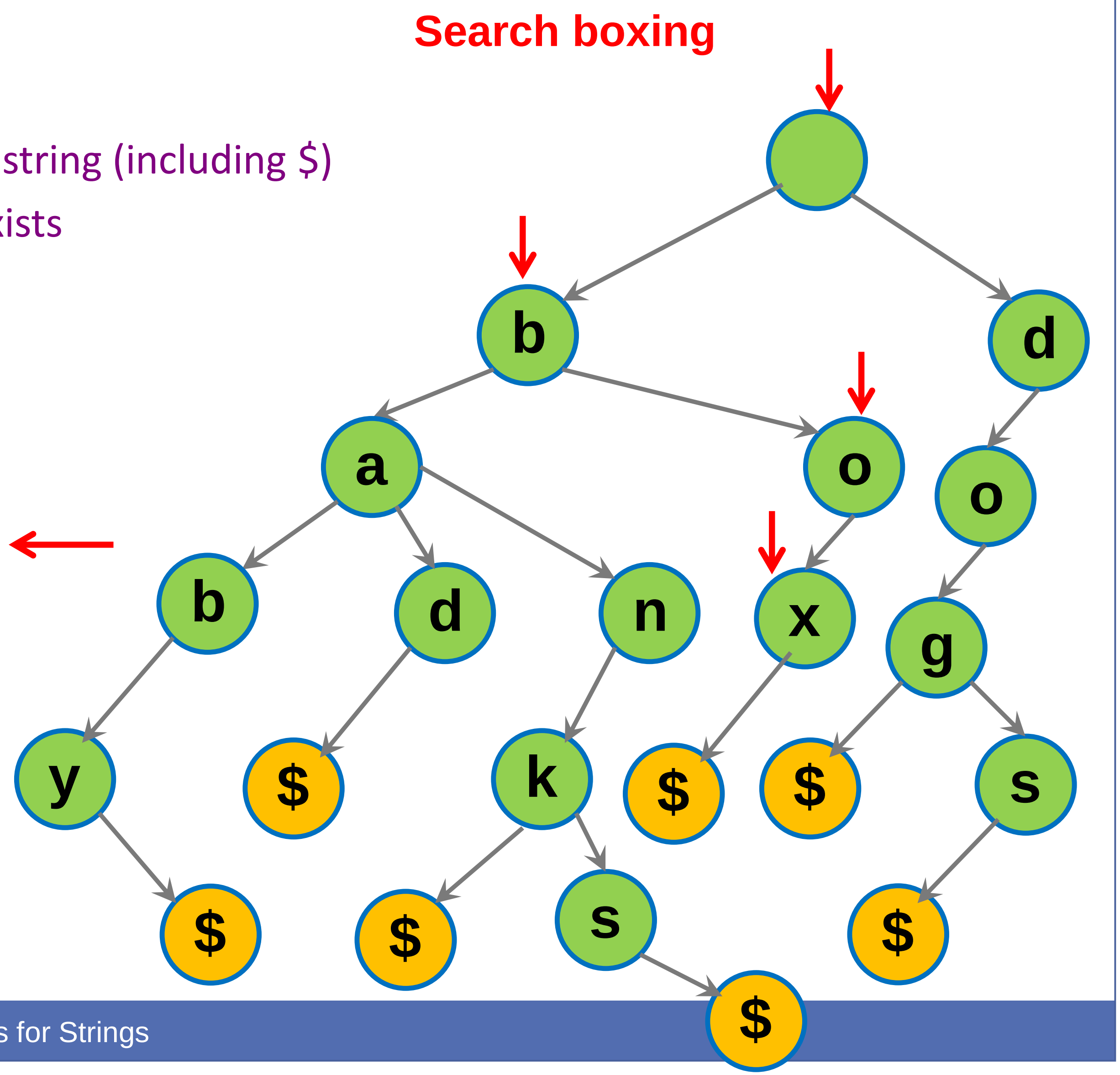
- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return “found”
  - Else
    - Return “not found”



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

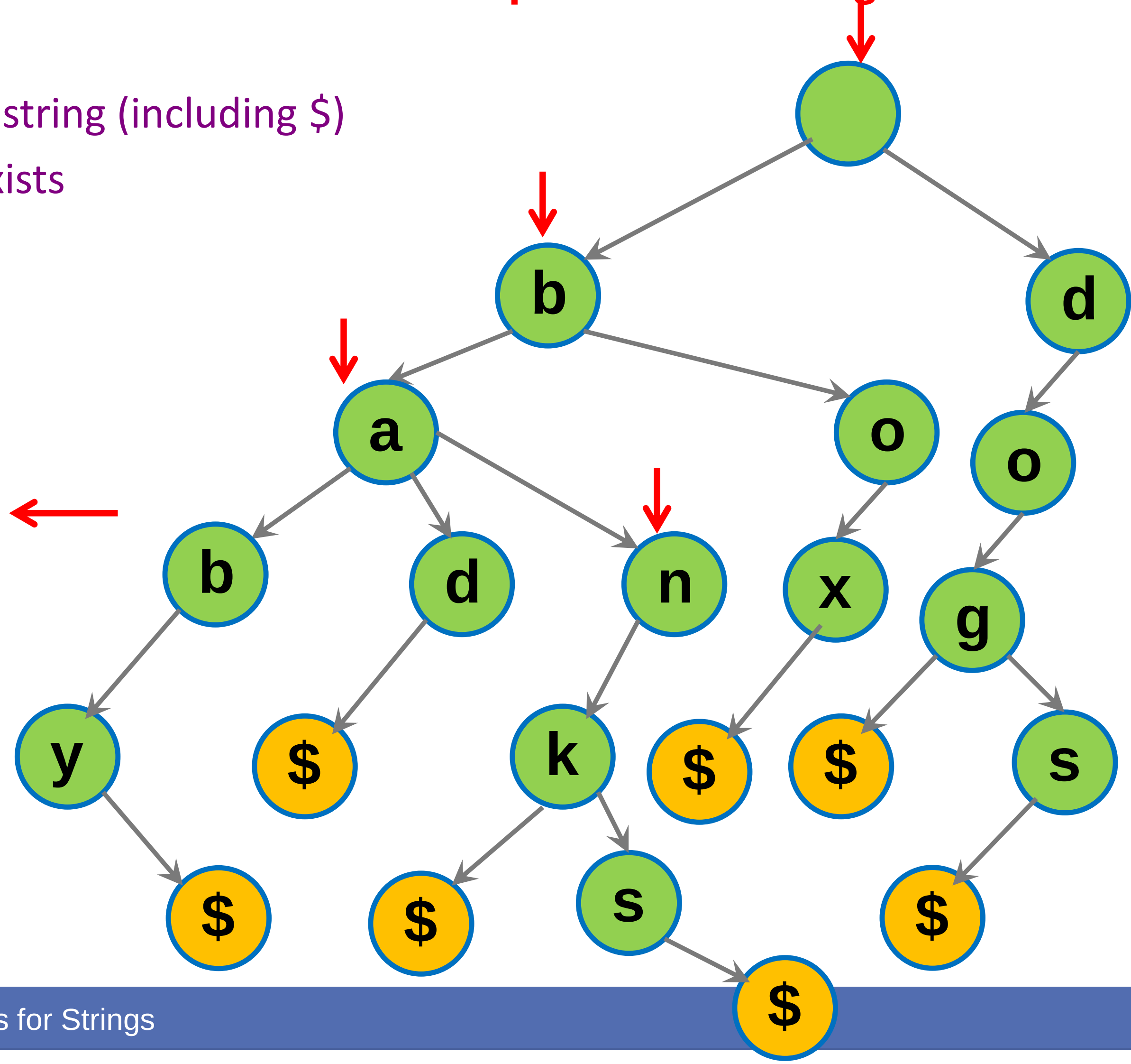
Searching a string:

- 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”

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

Output for searching ban ?



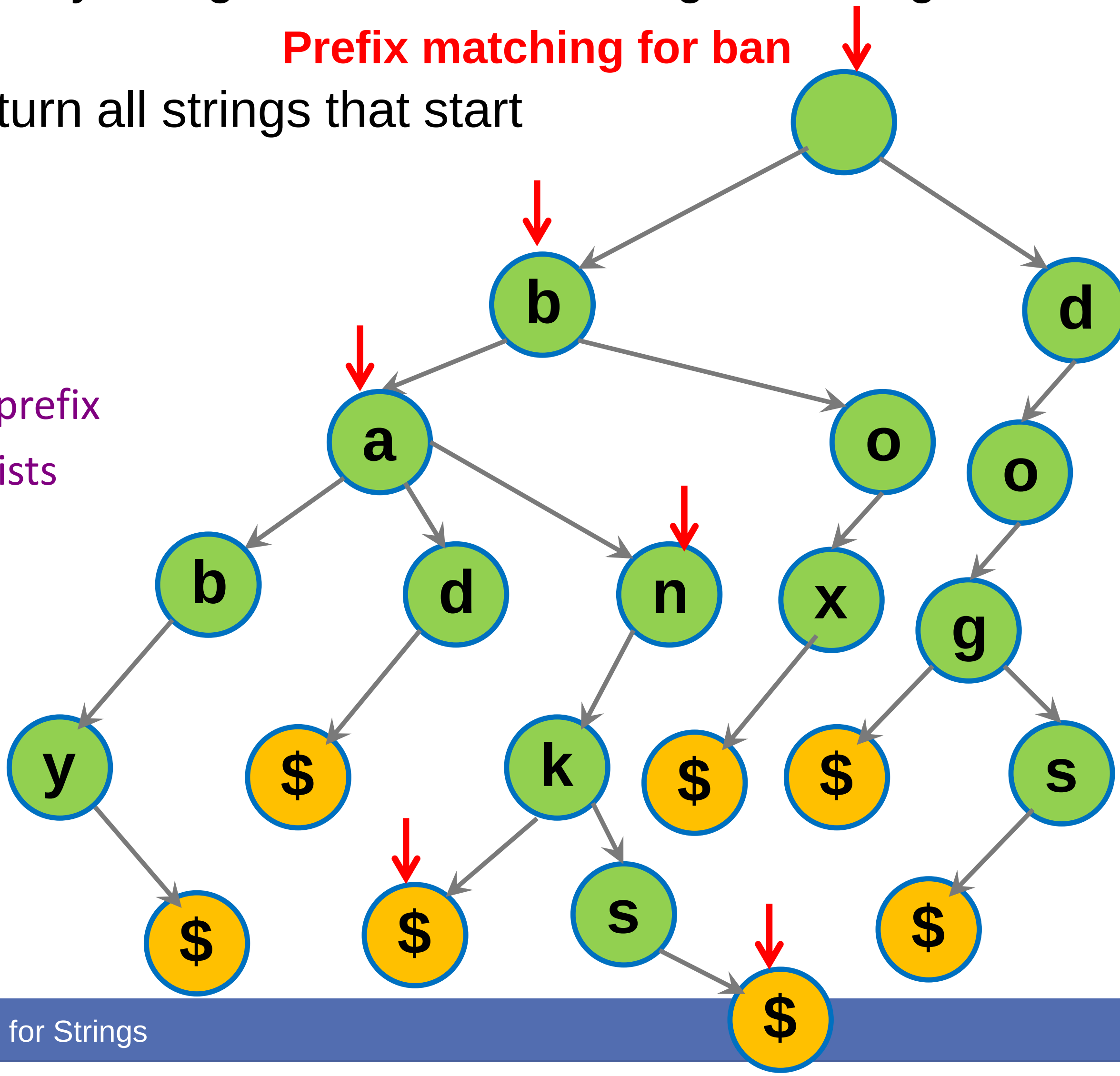
Trie Example: Prefix Matching

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



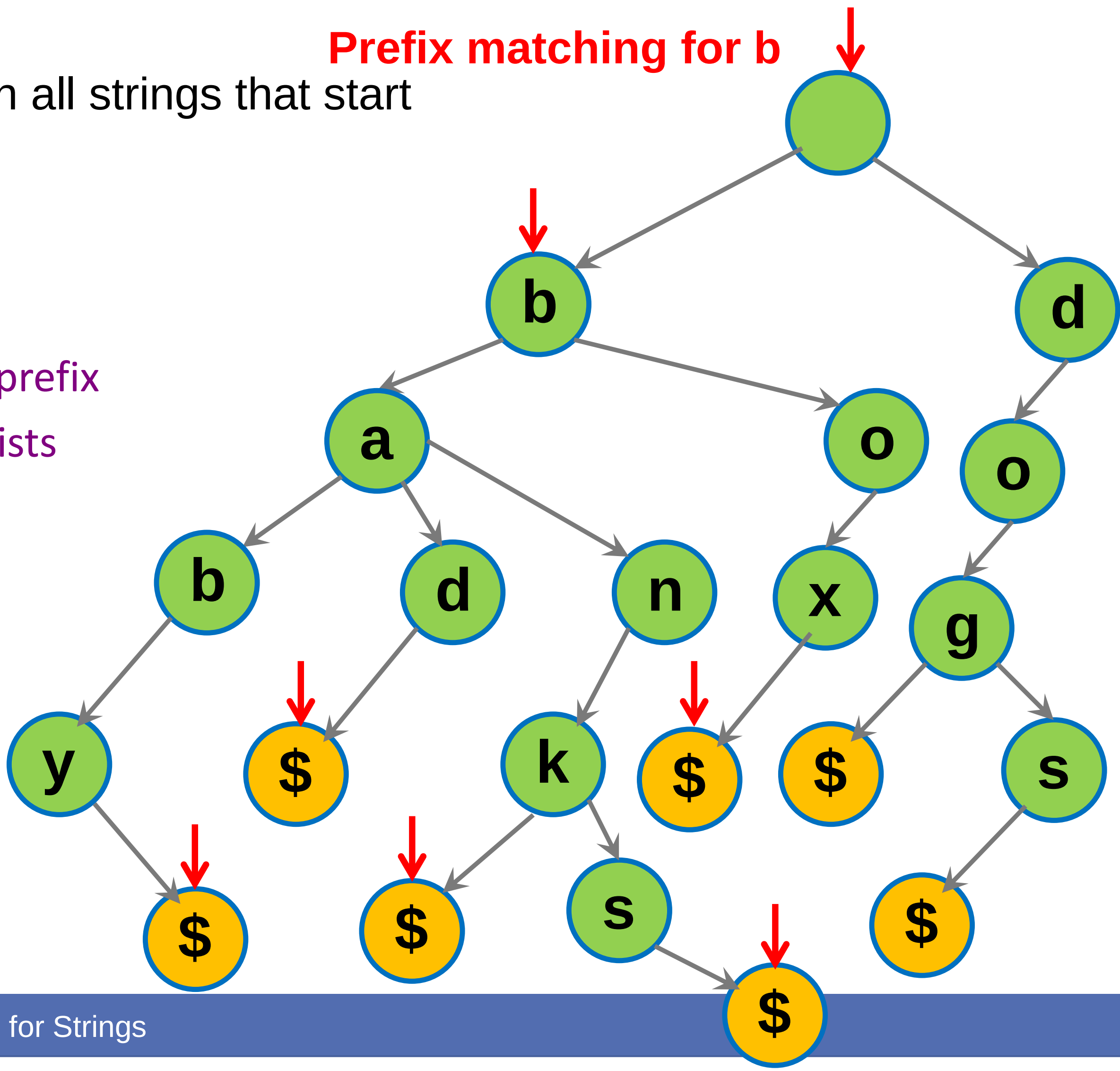
Trie Example: Prefix Matching

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

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

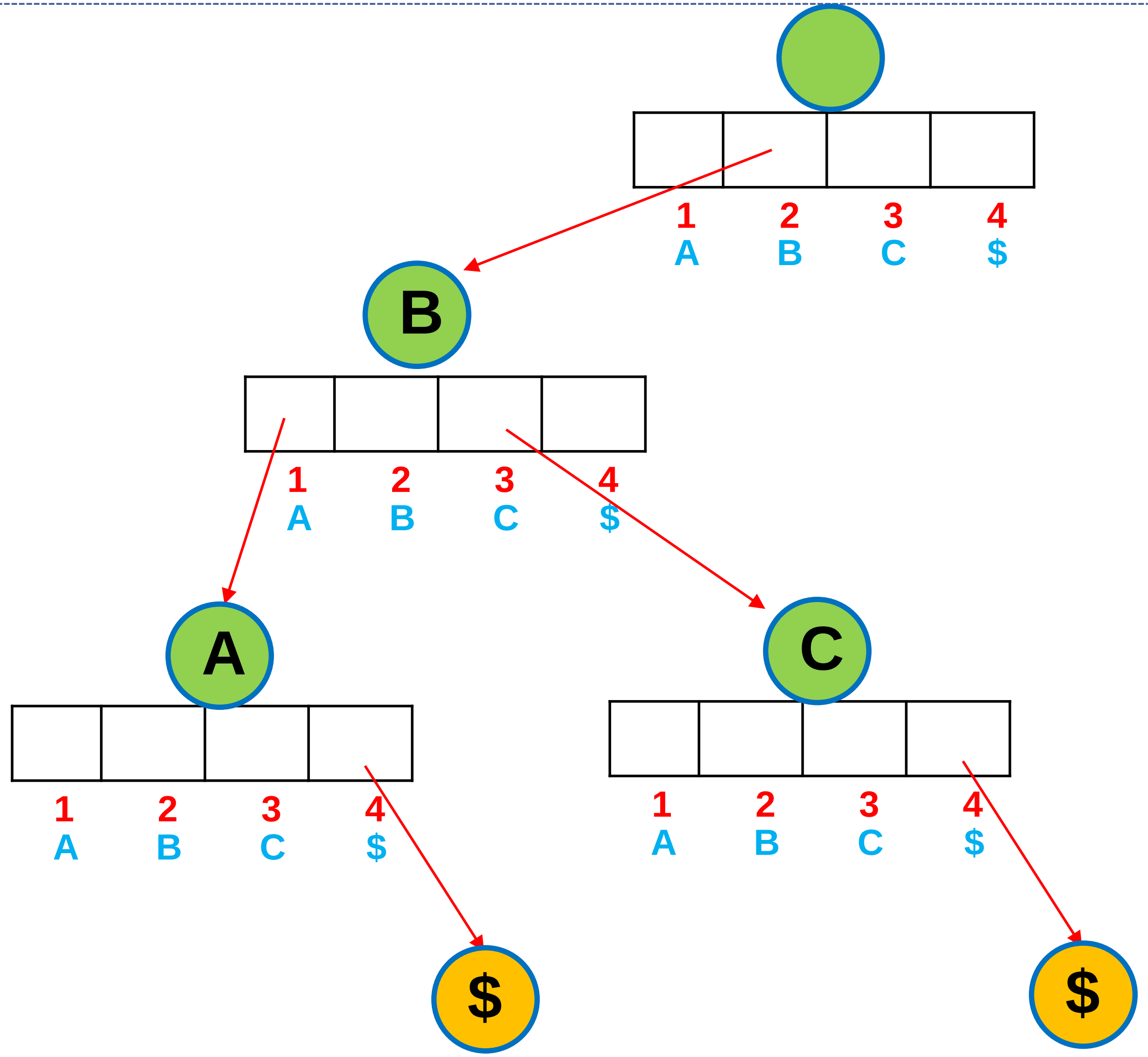
Implementation using an array:

- At each node, create an array of alphabets 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).

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 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

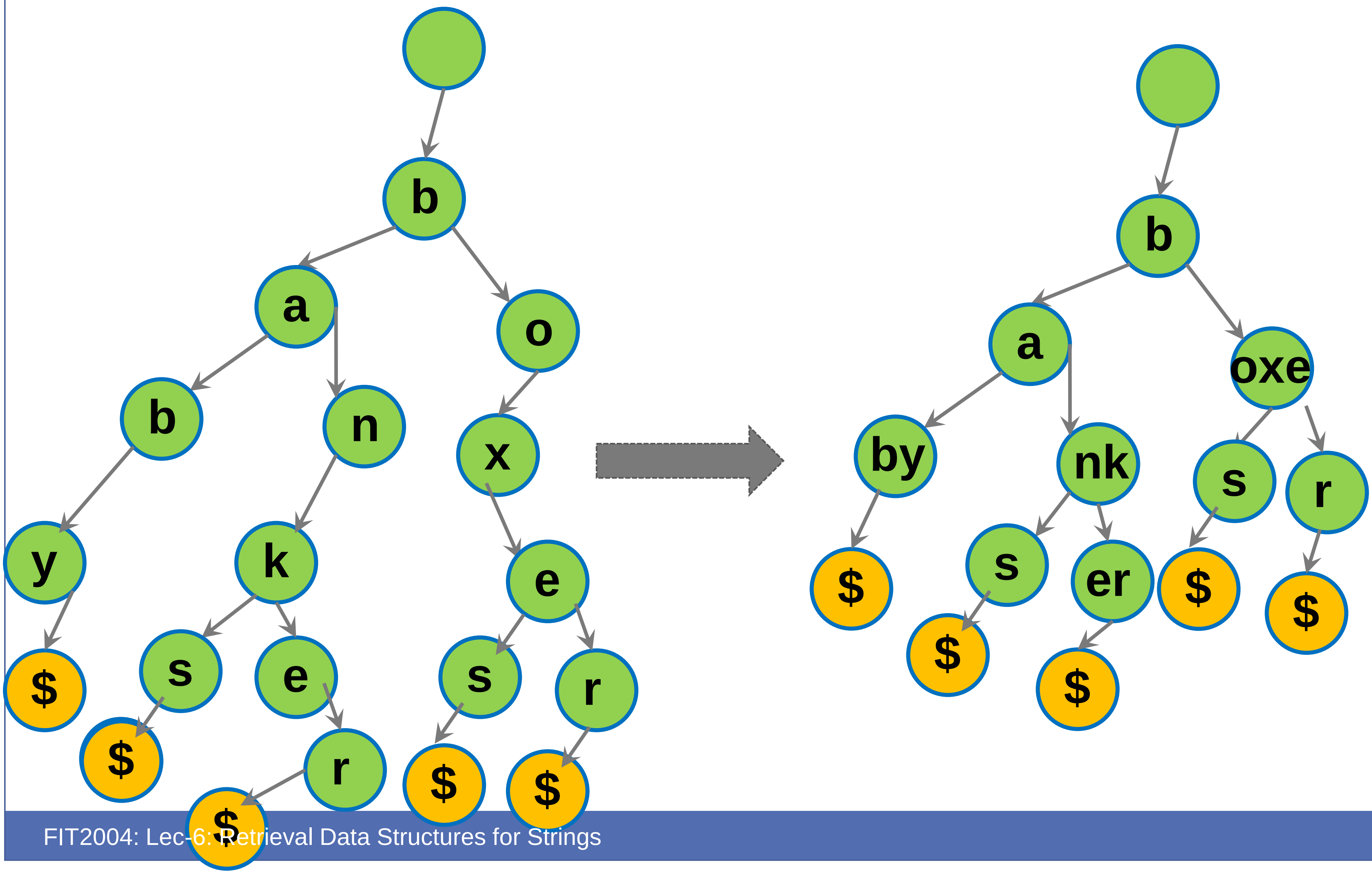
Some properties of Trie

- 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 (NOT EXAMINABLE BUT WORTH MENTIONING)

- Radix/PATRICIA tree is a space-optimized/compact Trie data structure
- Unlike regular tries, edges can be labeled 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 (<u>NOT EXAMINABLE</u> BUT WORTH MENTIONING)



Outline

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

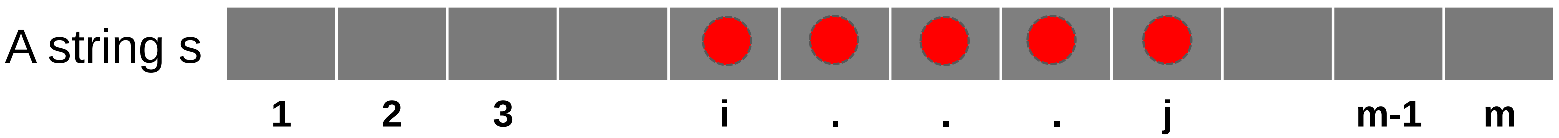
- (Prefix) Tries are very useful for quickly looking up whole words, but more generally, **prefixes** of words
- A prefix of a word s[1..m] is some string s[1..i] where 1<=i<=m

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

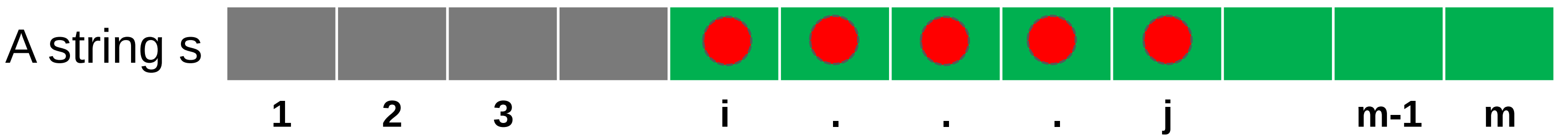
Prefixes and suffixes

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



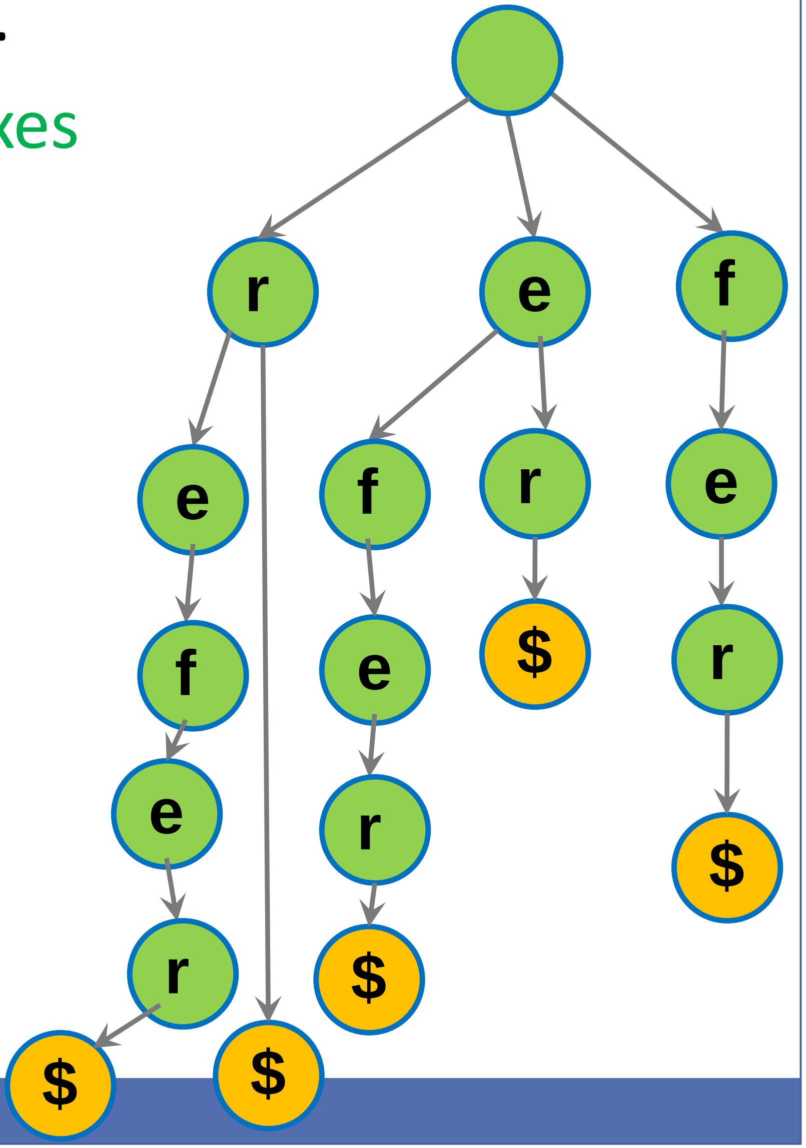
Prefixes and suffixes

- 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



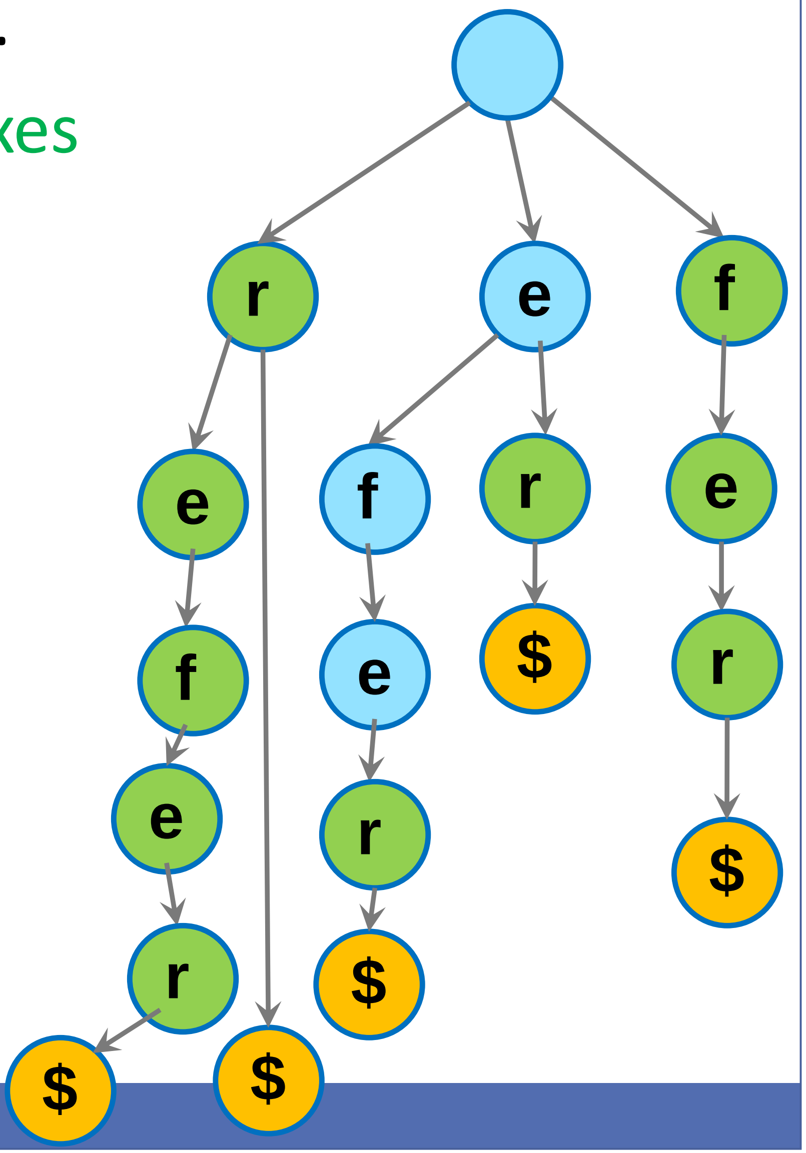
Suffix Trie

- 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



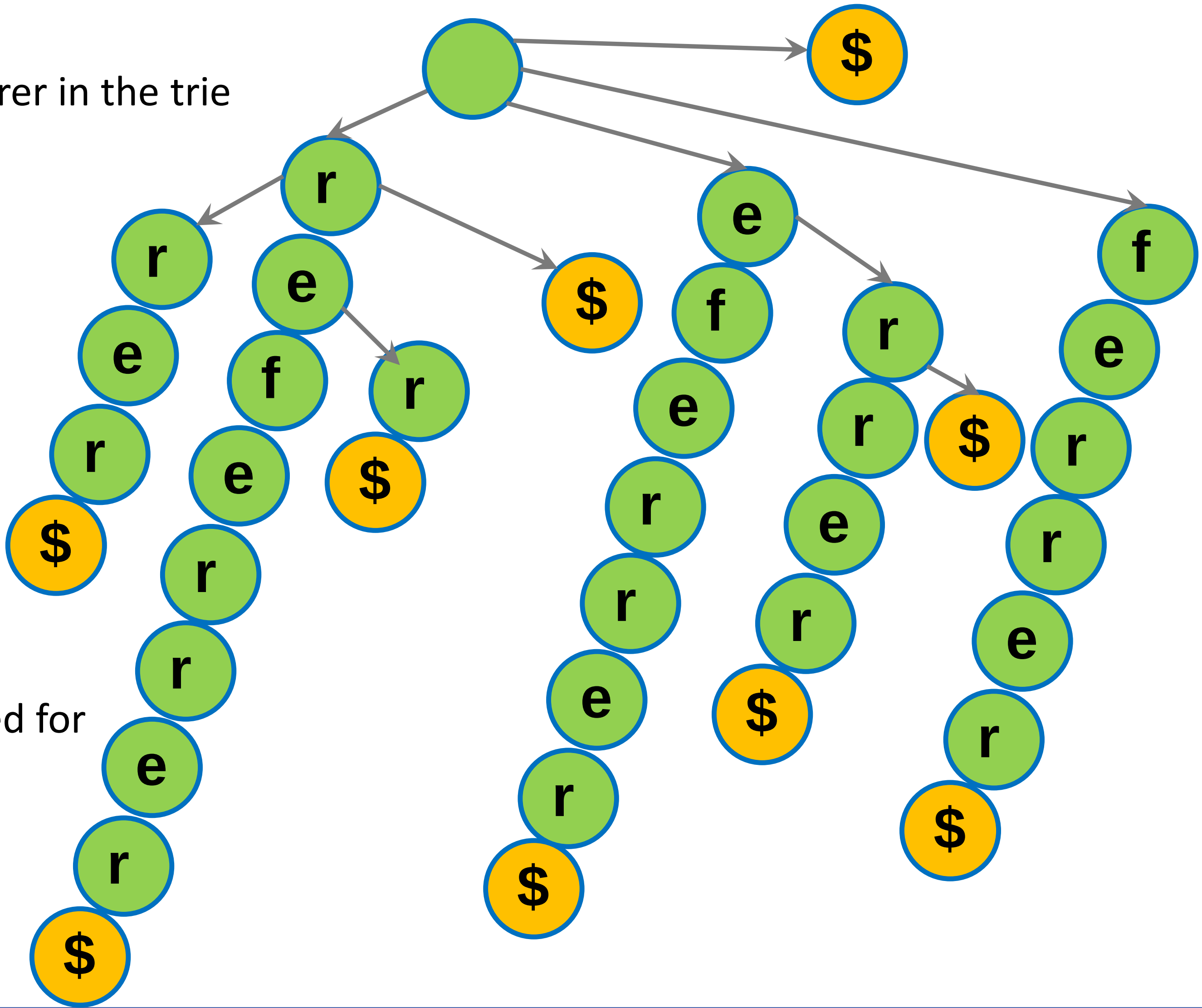
Constructing Suffix Trie

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

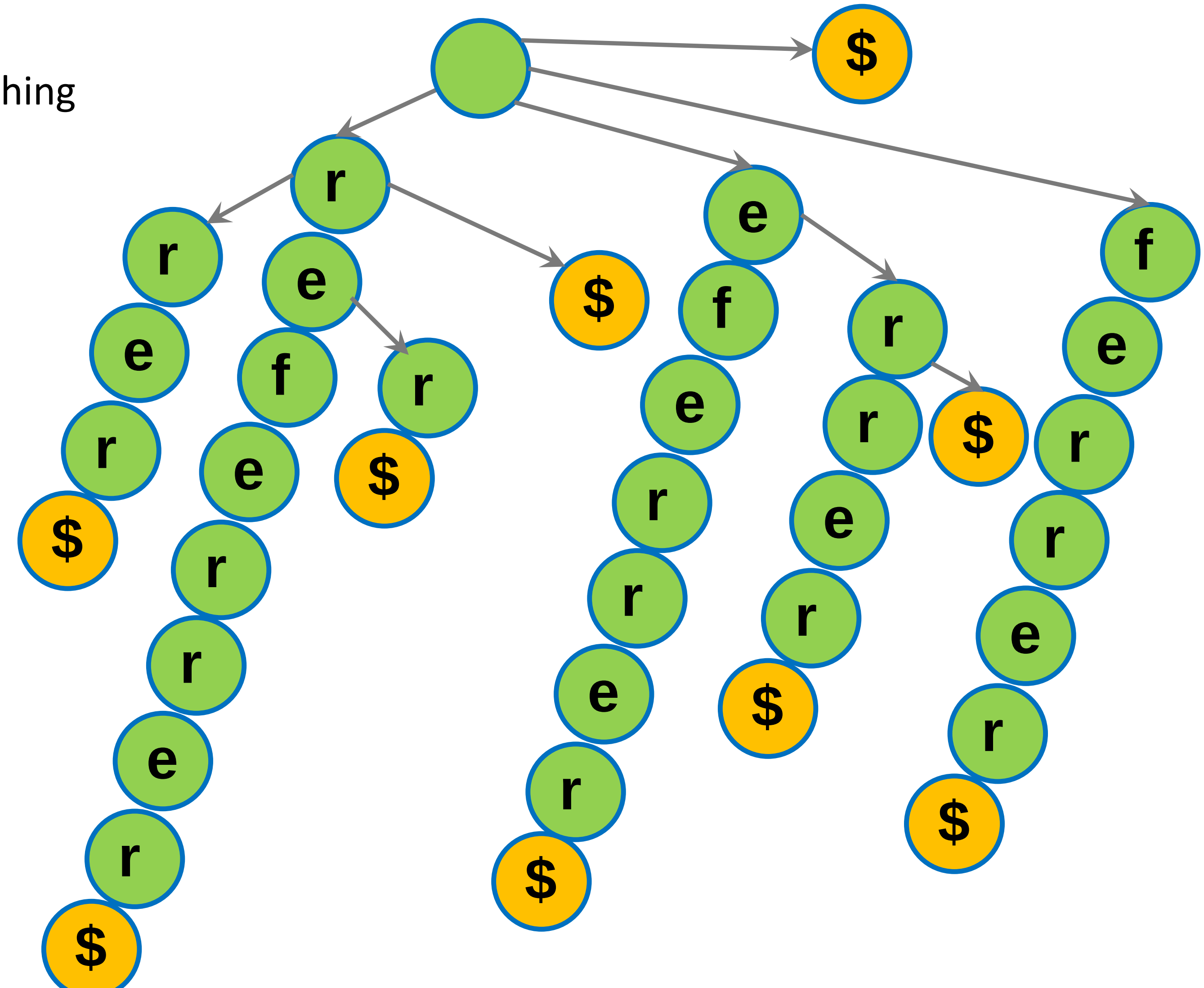
1. Introduction
2. Trie
   A. Construction
   B. Query Processing
3. Suffix Trie
   A. Construction
   B. Query Processing
   C. Suffix Tree
4. Suffix Array
   A. Introduction
   B. Query Processing
   C. Reducing Construction Cost

Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search "err"

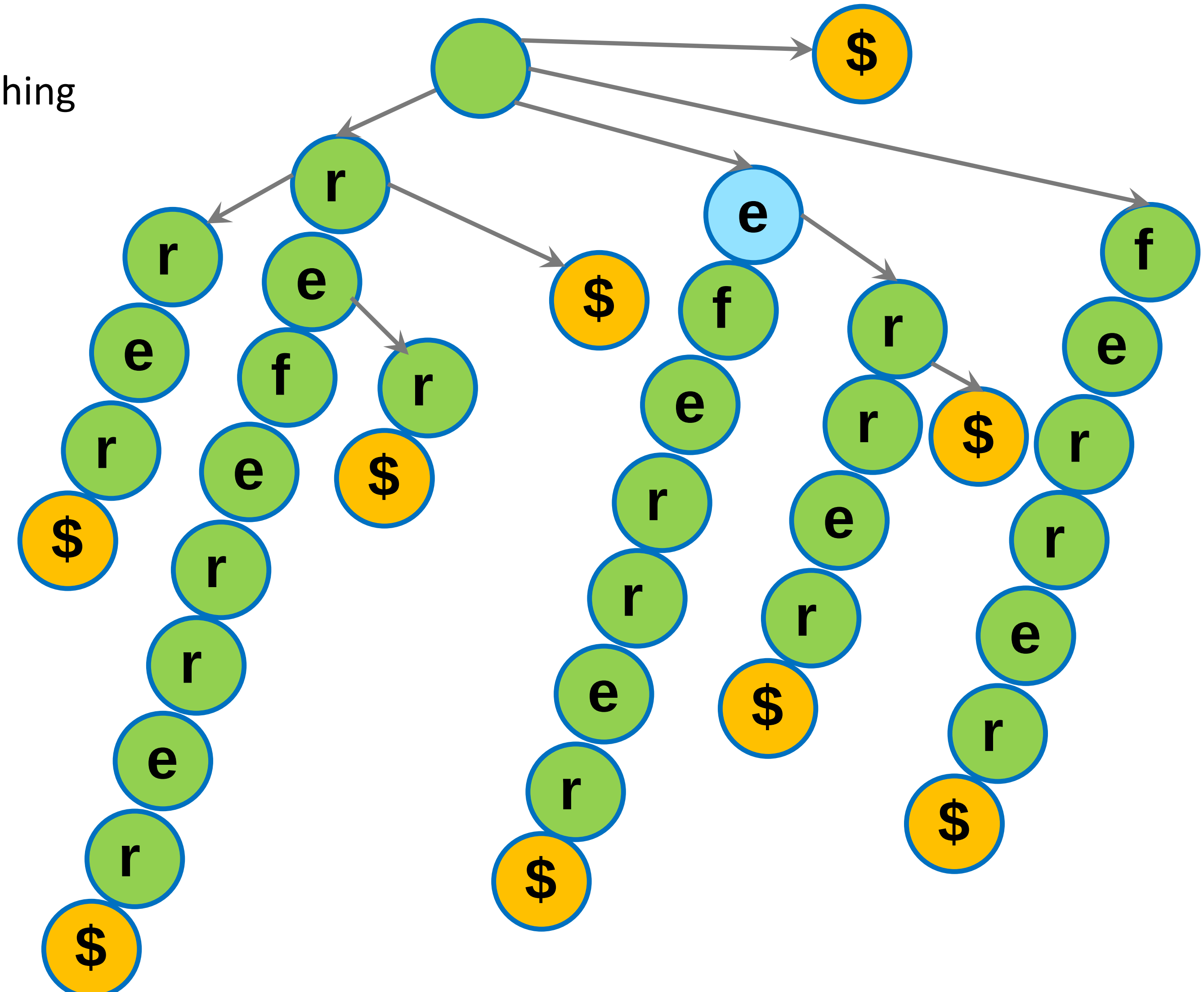


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search "err"

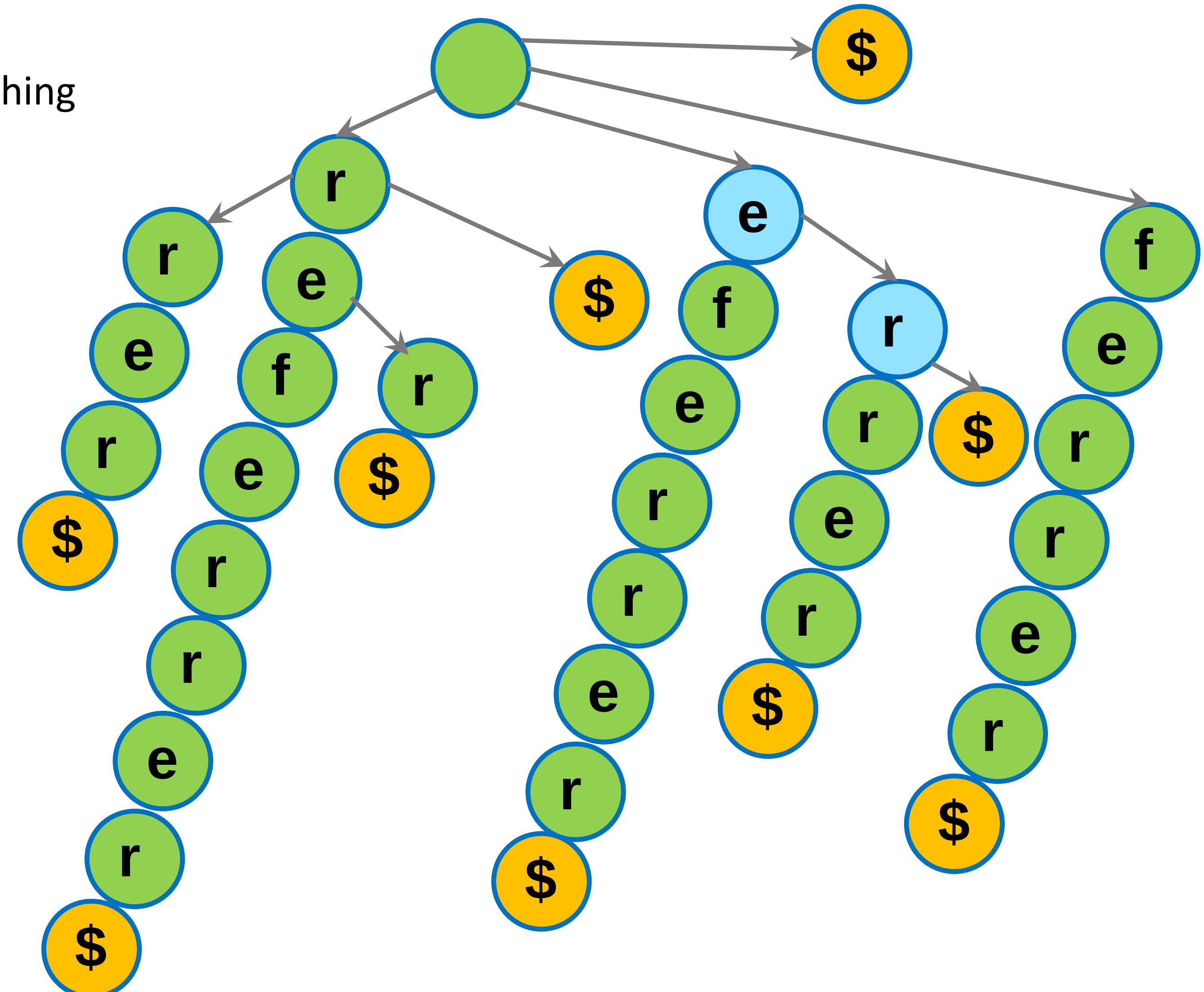


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

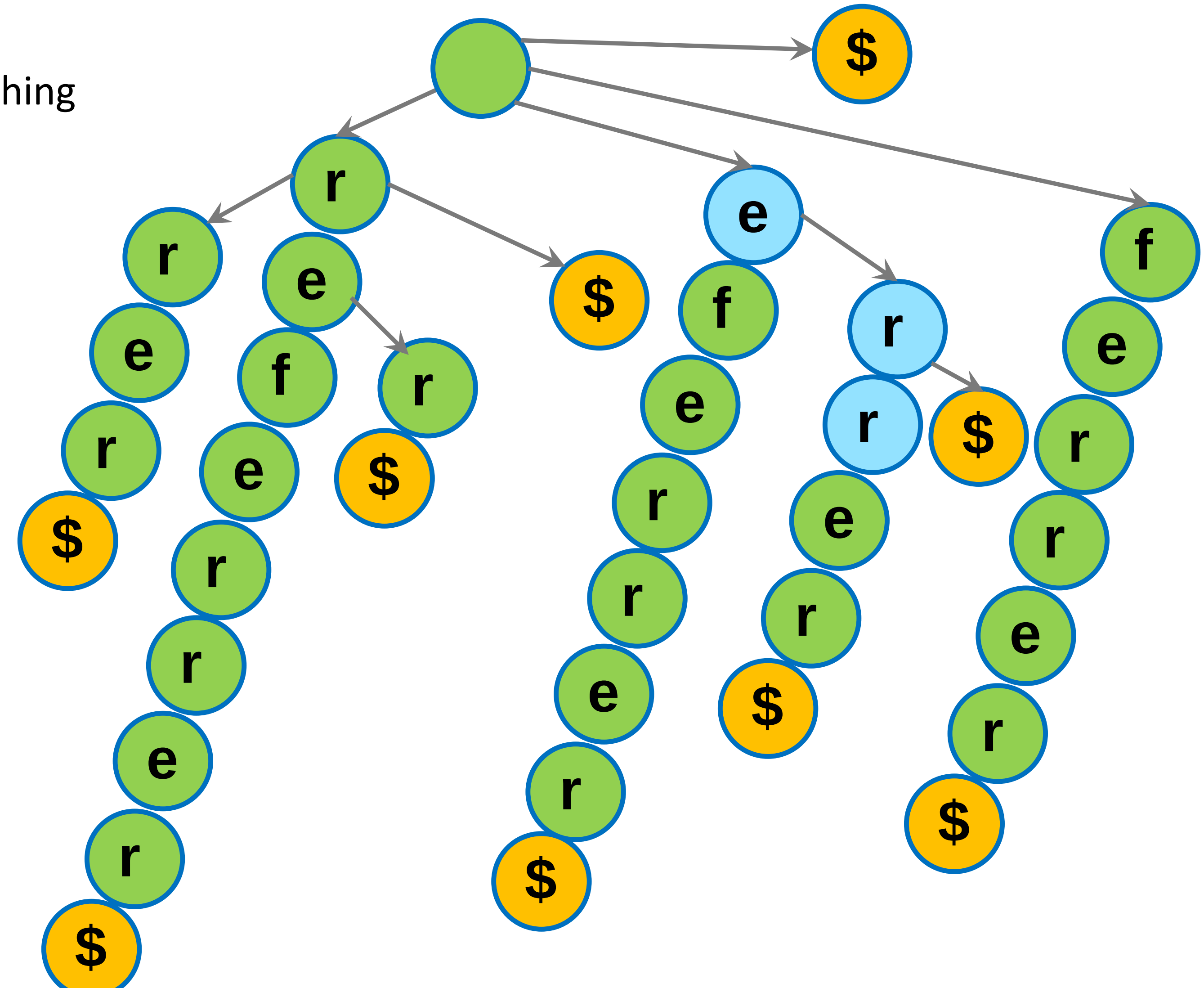


Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”



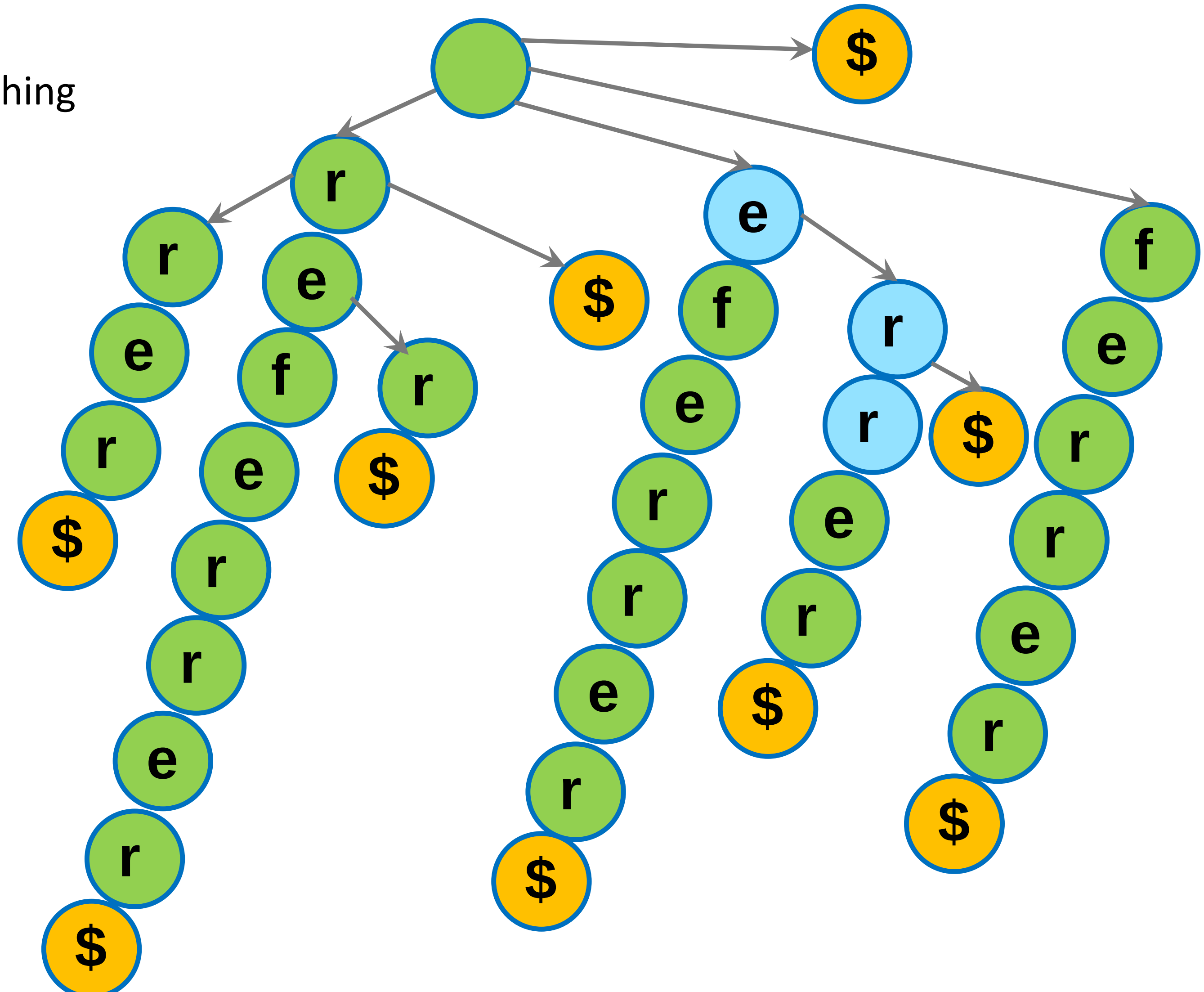
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

Found!



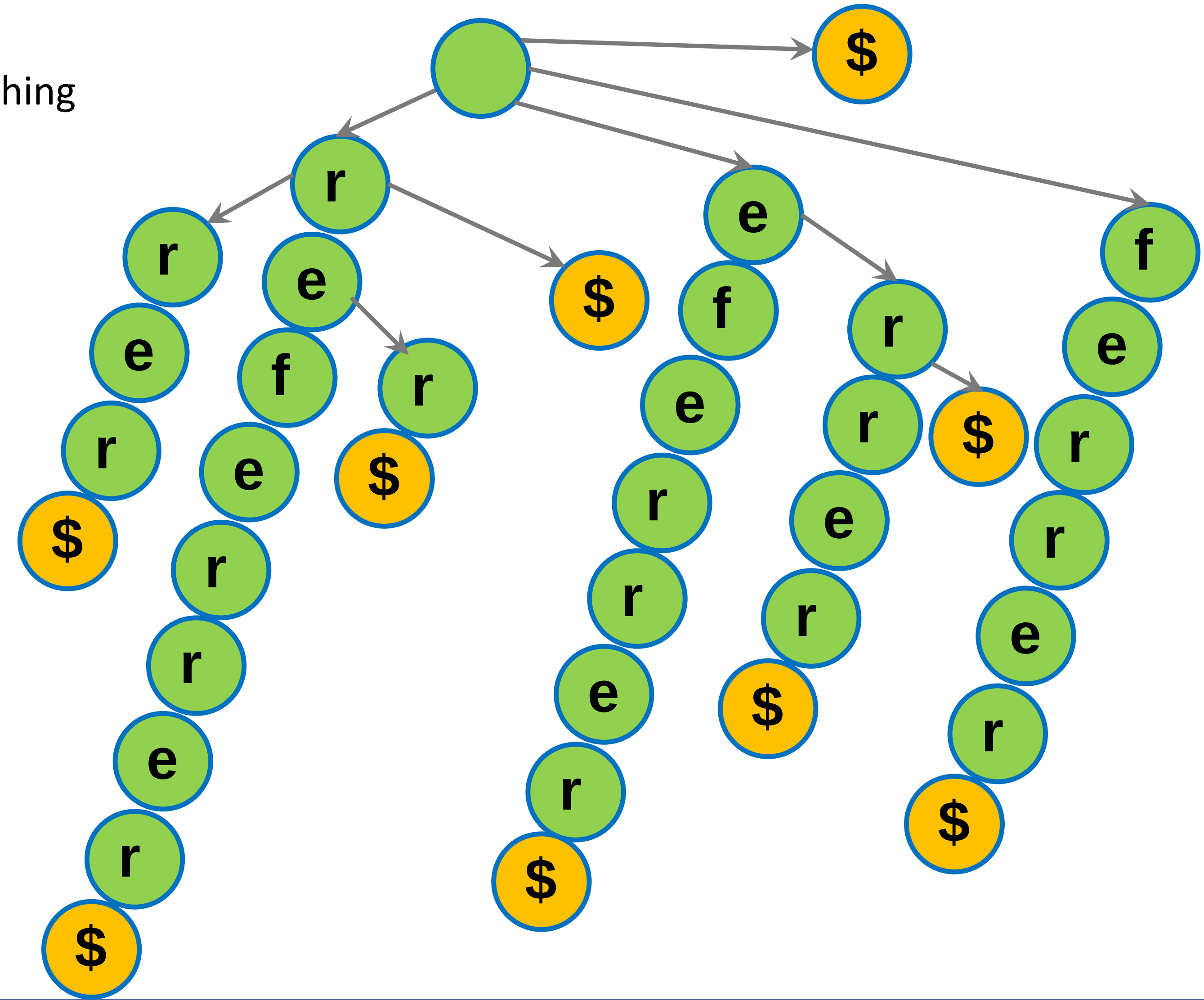
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



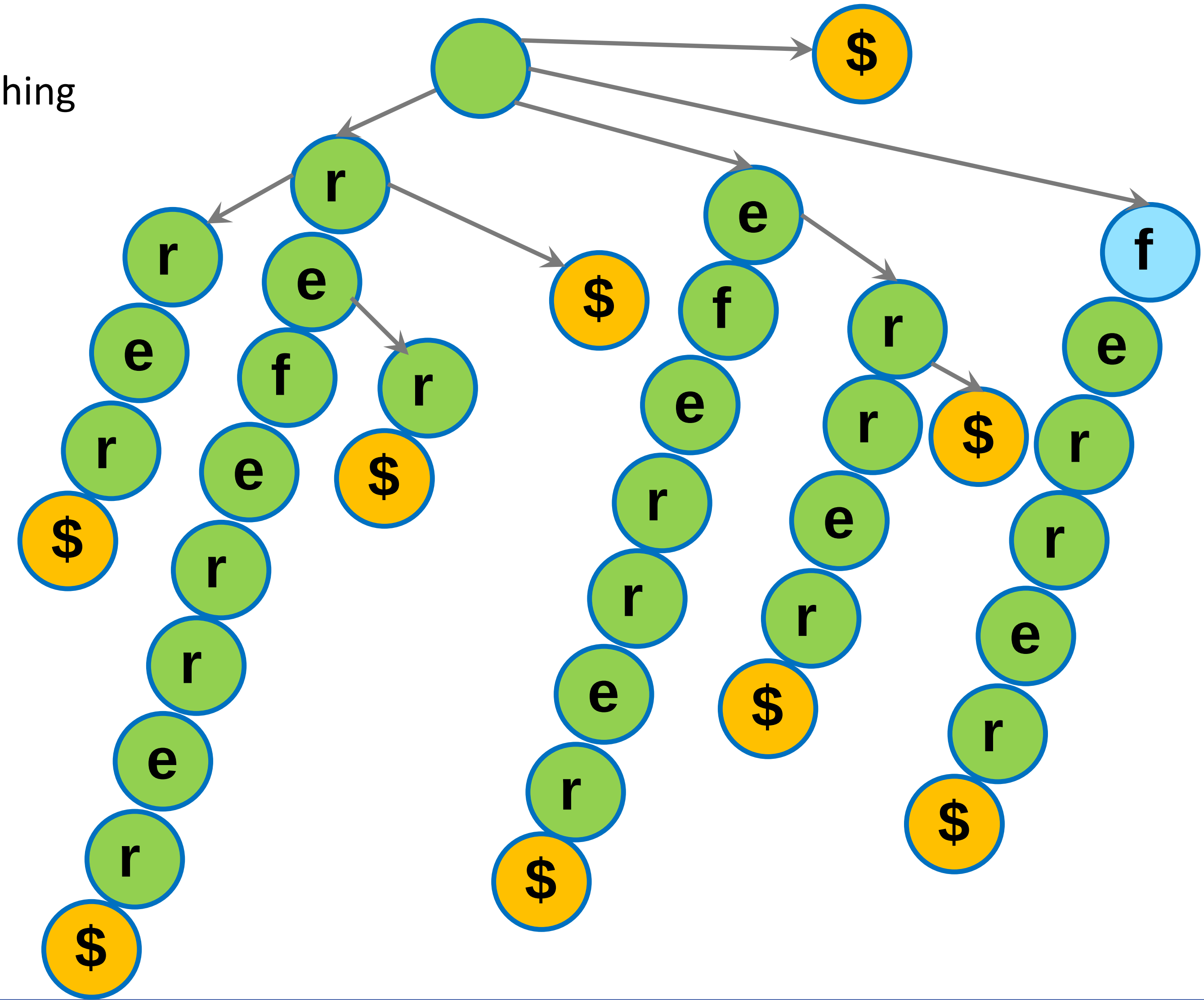
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



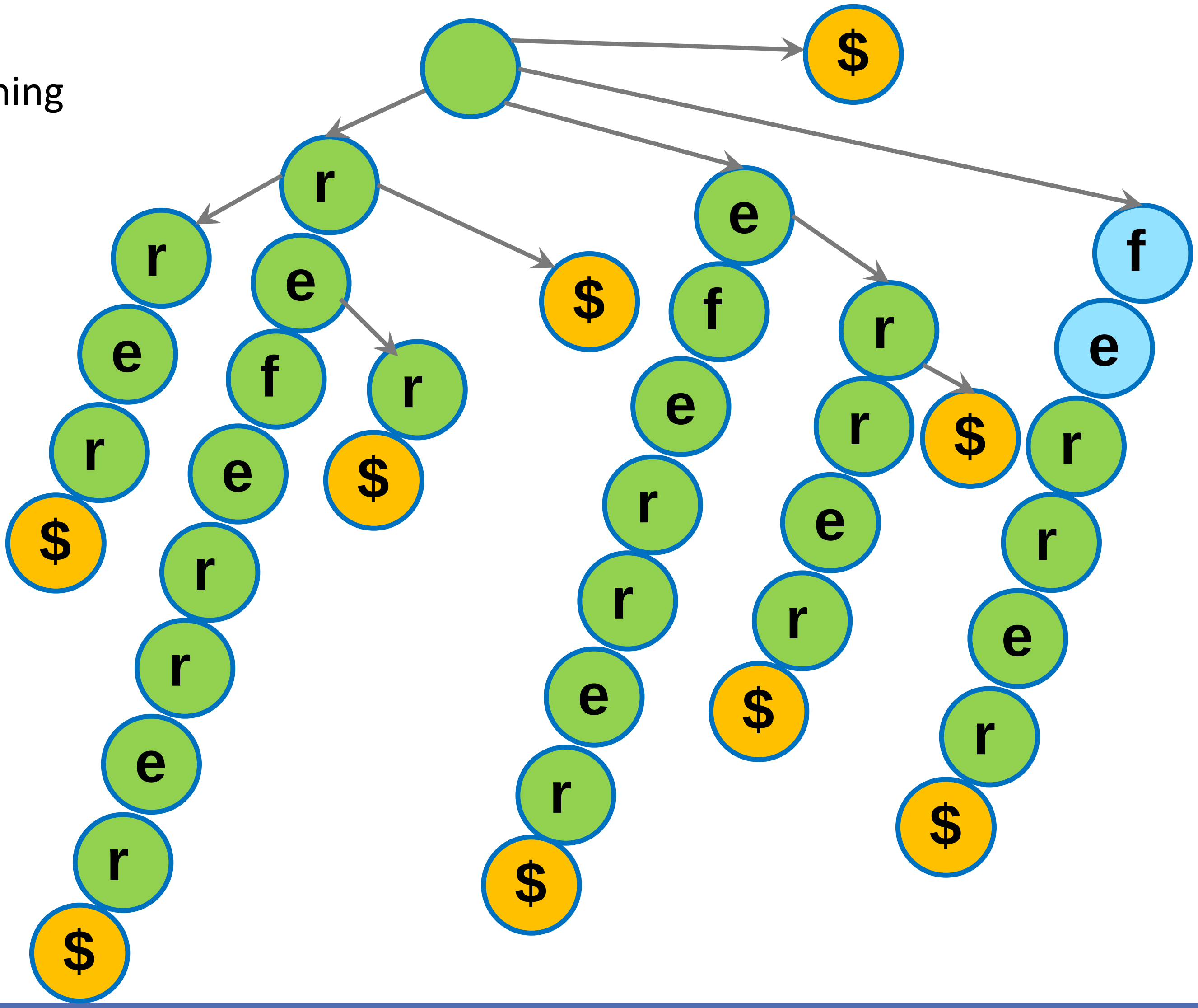
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



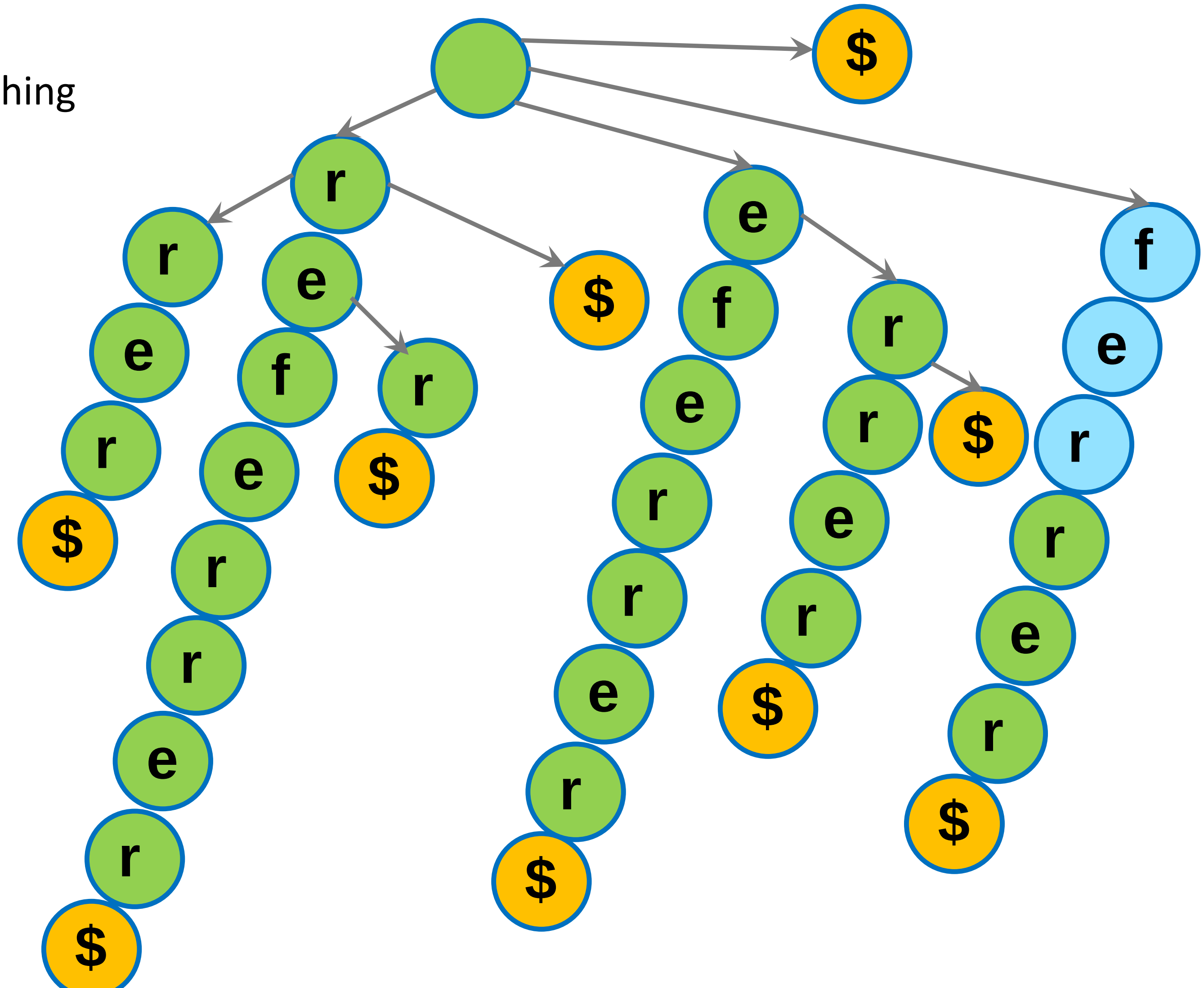
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”



Substring search on Suffix Trie

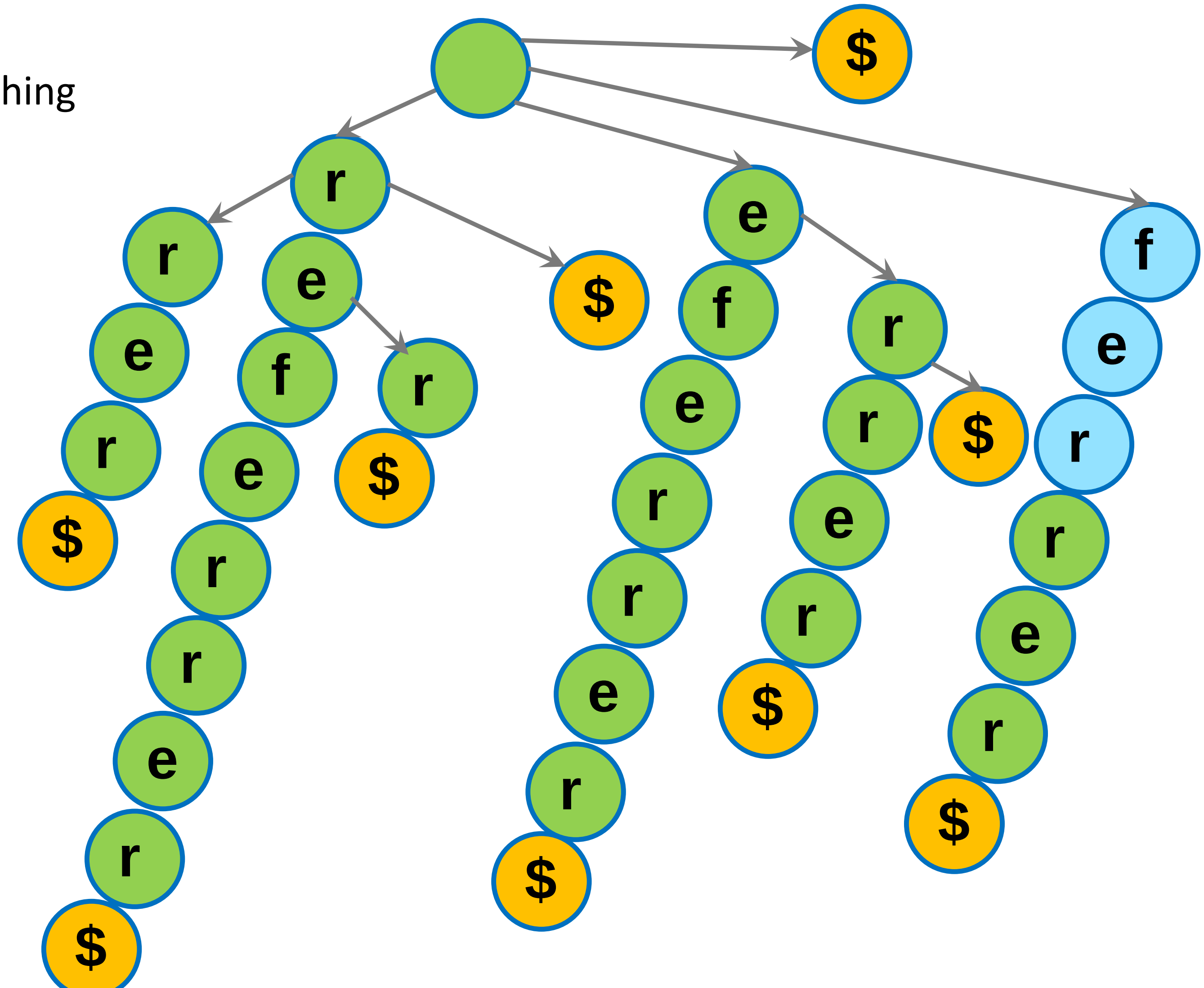
Substring search for str

- Similar to prefix matching

search "err"

search "fers"

Not found :(



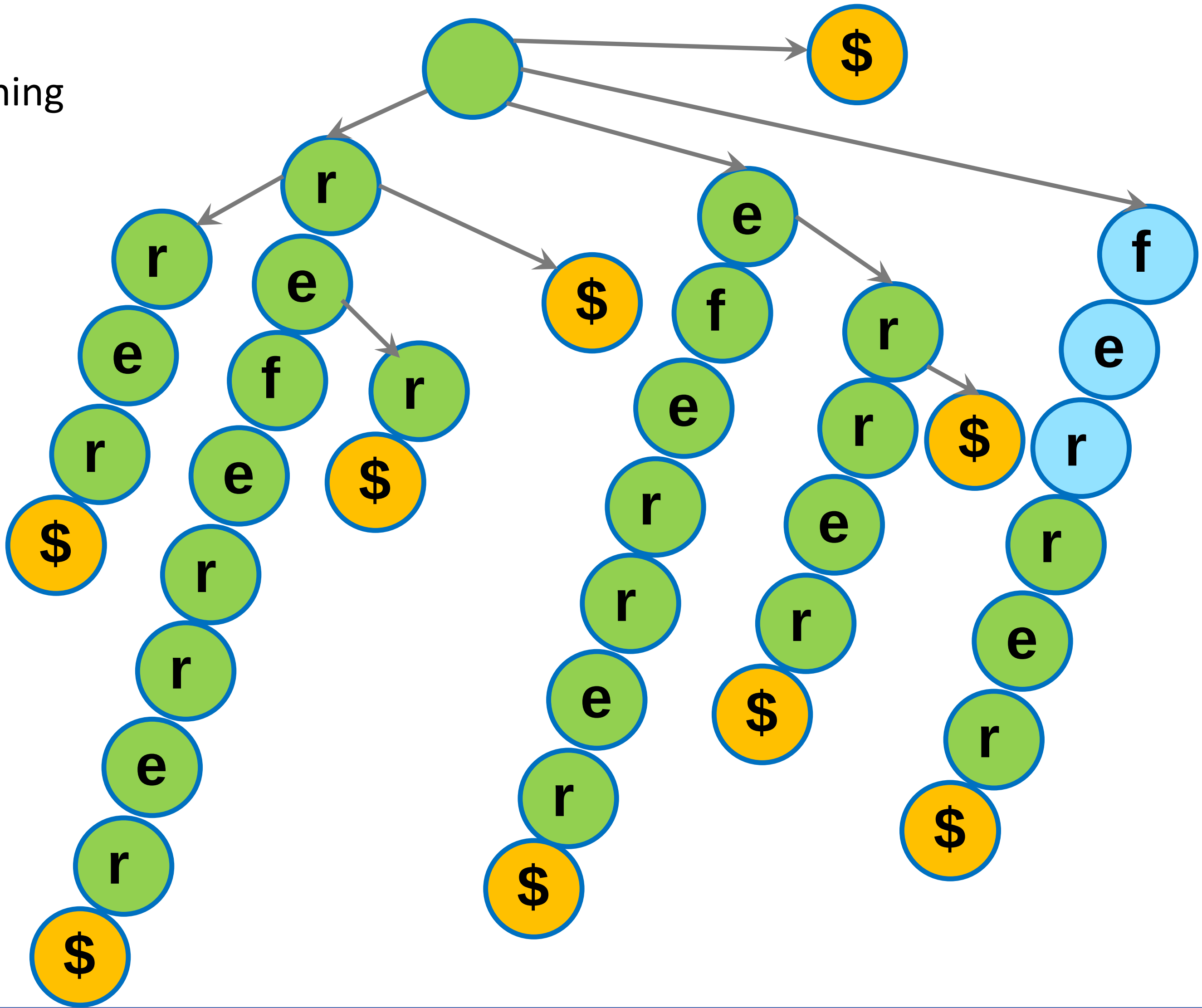
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

search “err”

search “fers”

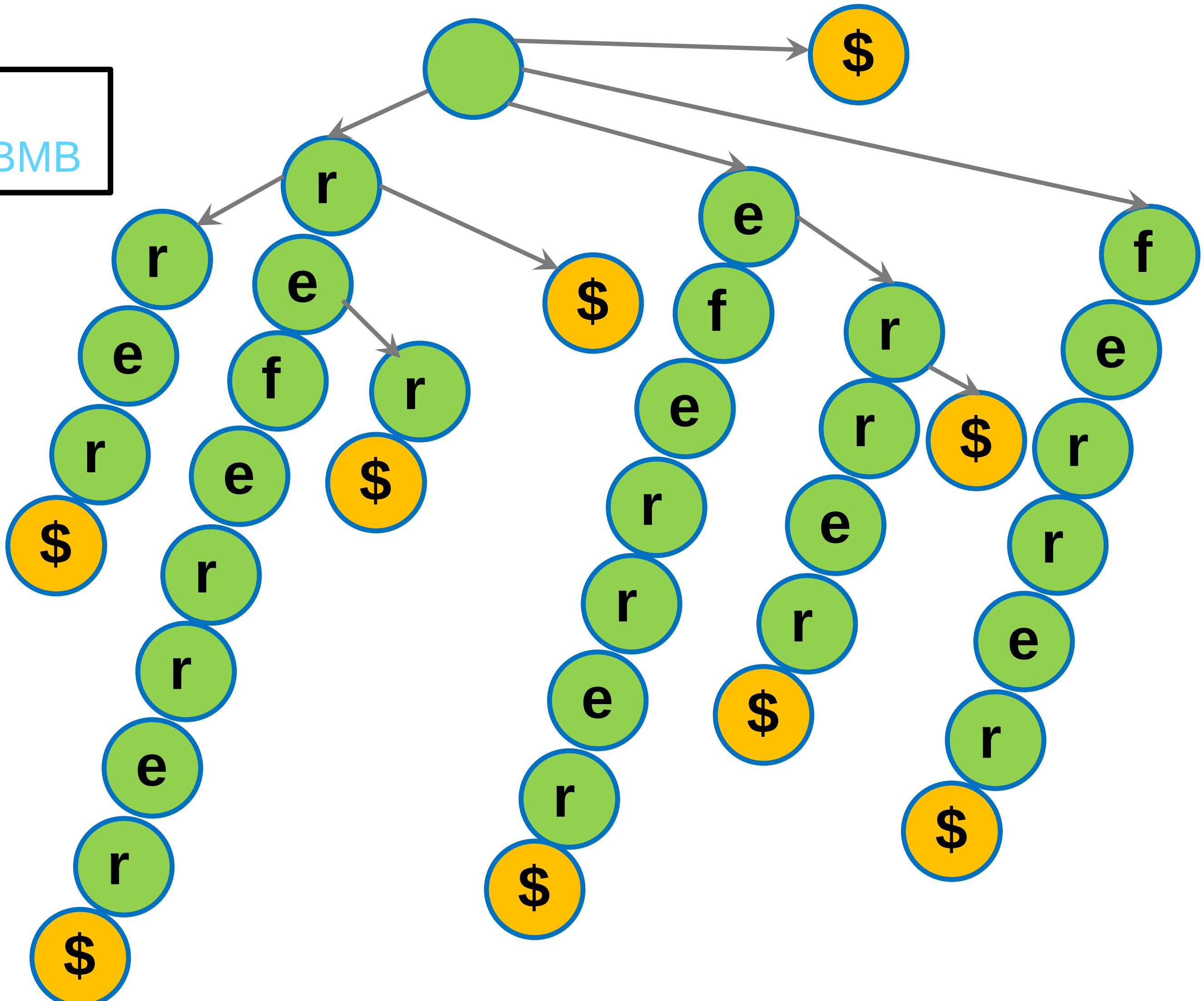


Time Complexity:

O(M) where M is the length of substring

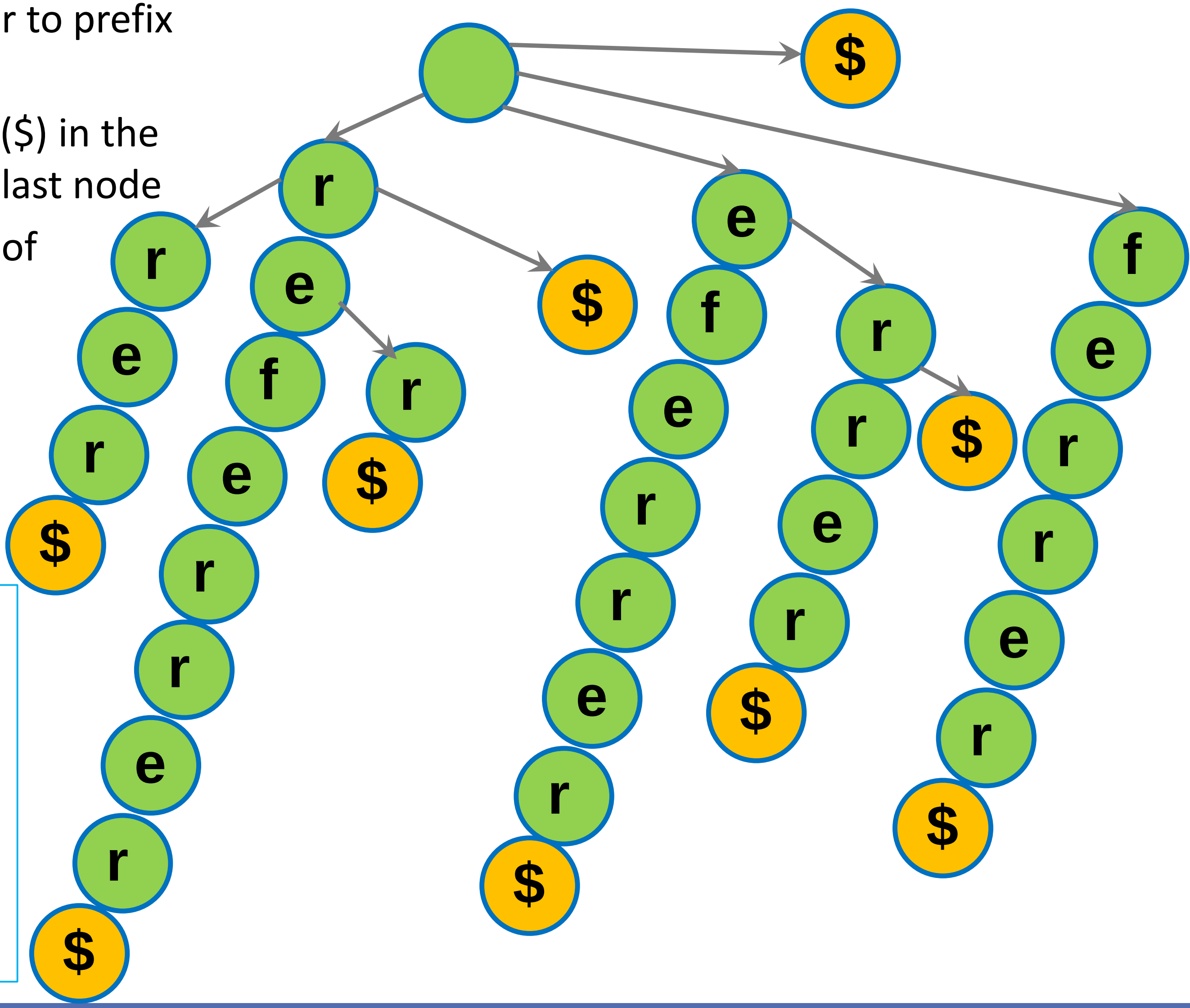
Counting # of occurences of a substring

Quiz time!
https://flux.qa - RFIBMB



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"

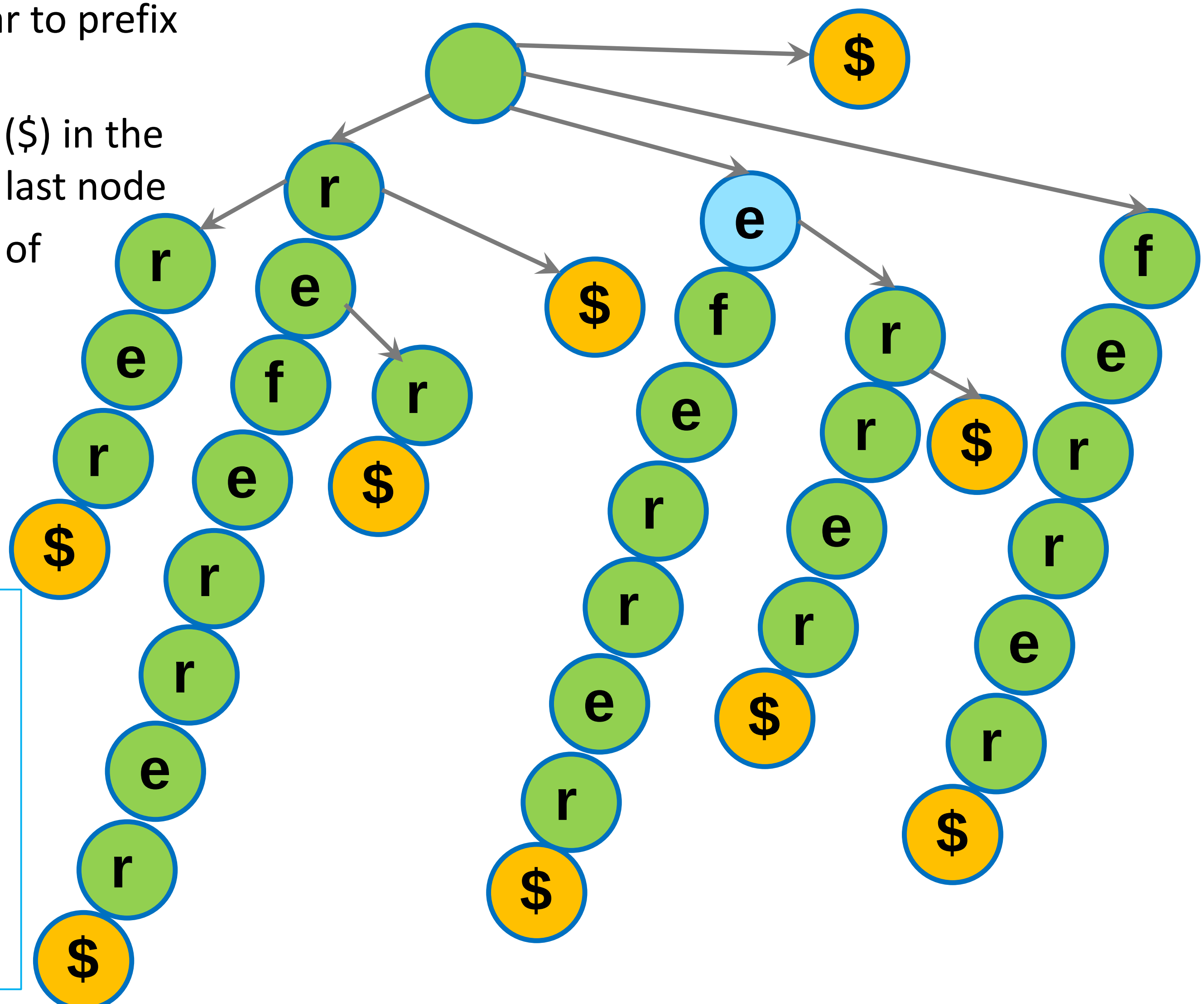


Time Complexity:

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

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”

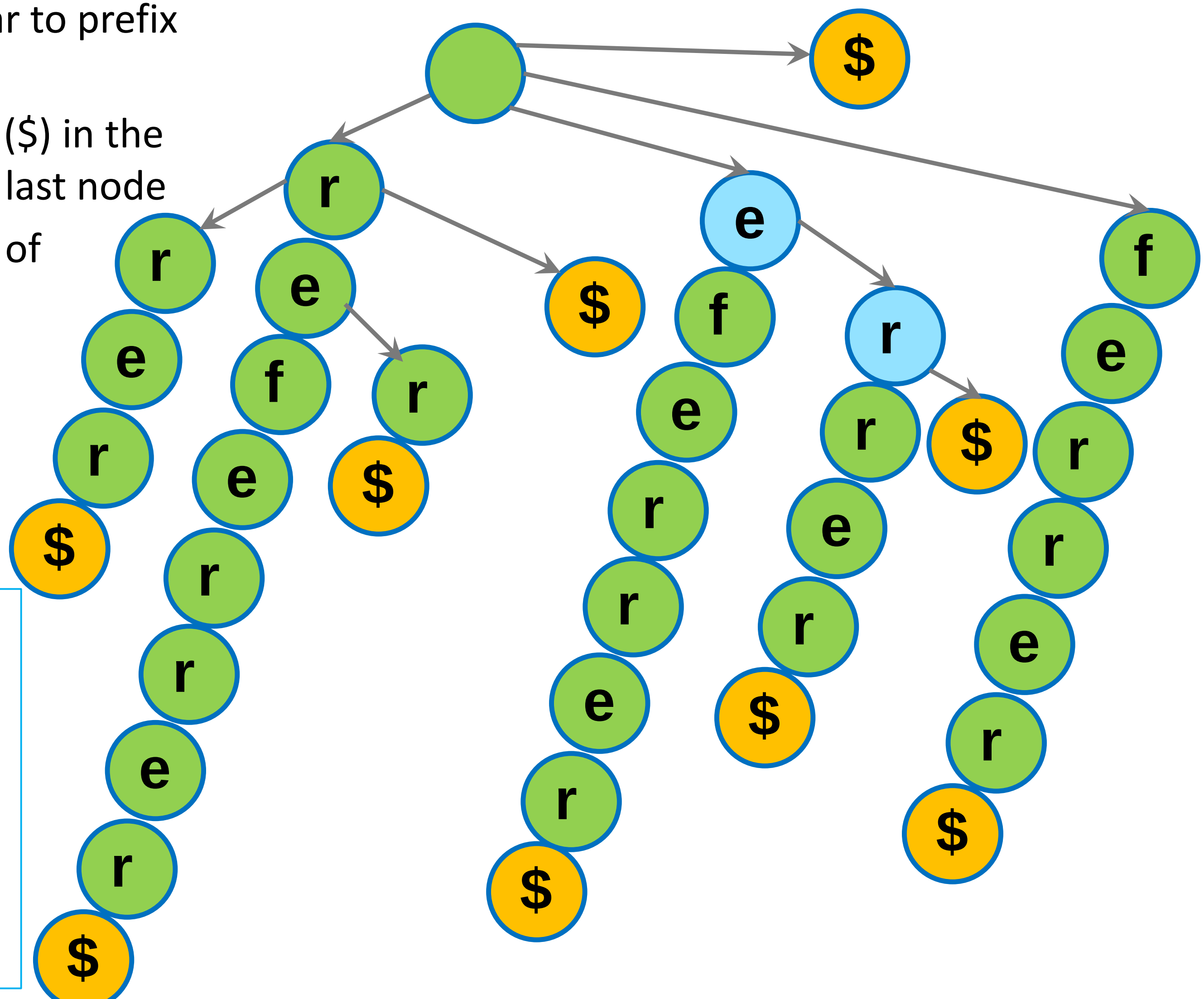


Time Complexity:

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

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"

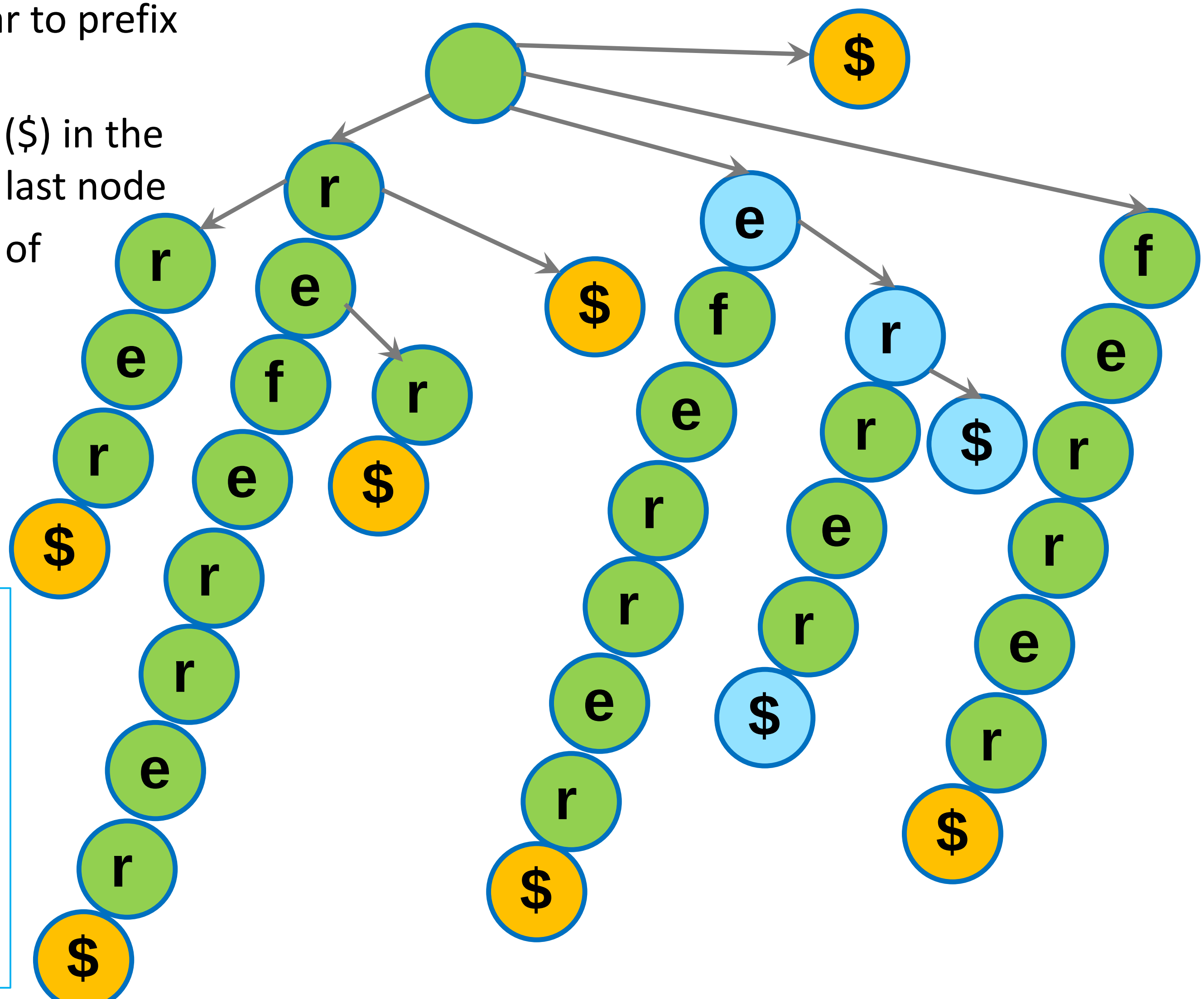


Time Complexity:

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

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"

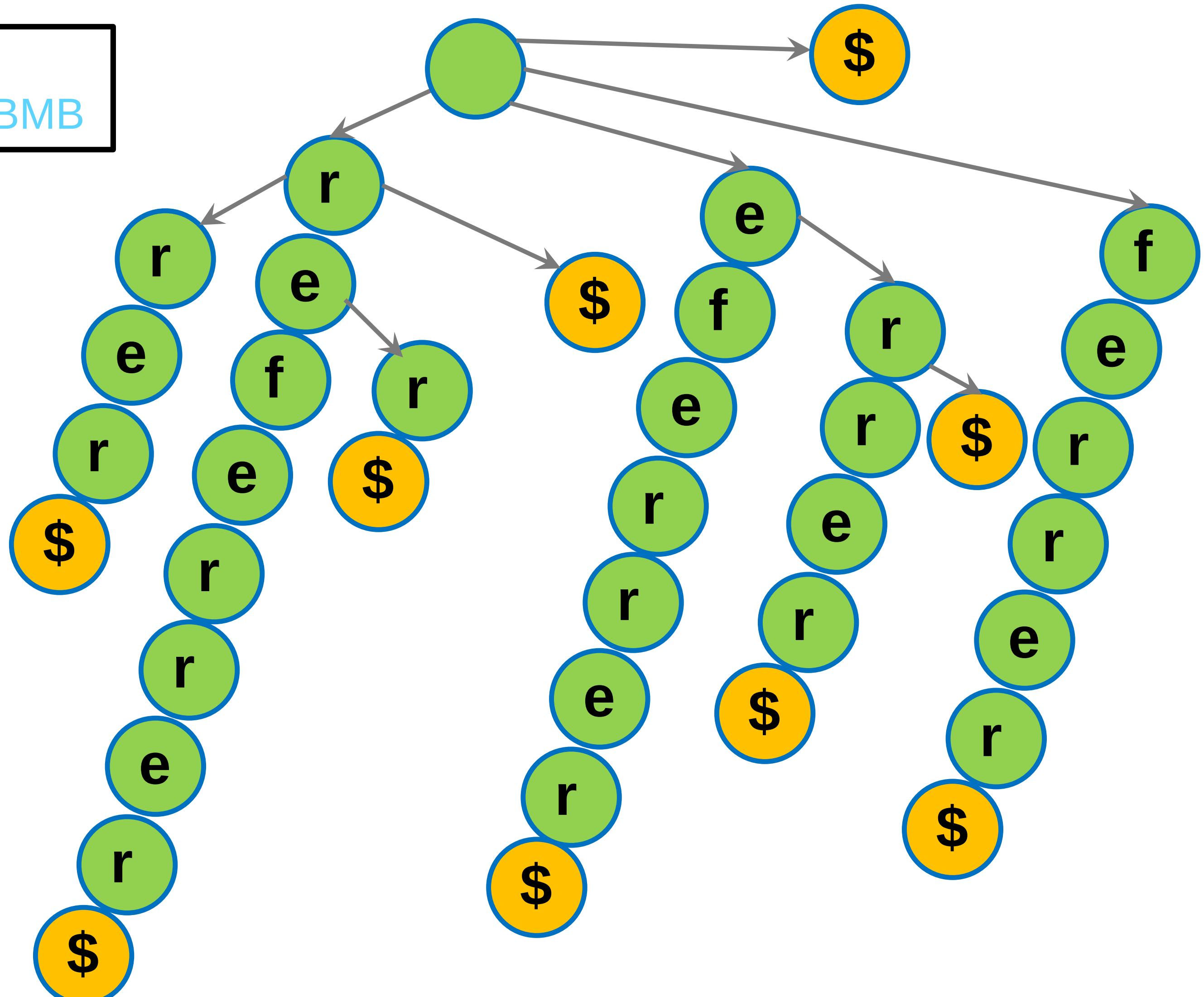


Time Complexity:

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

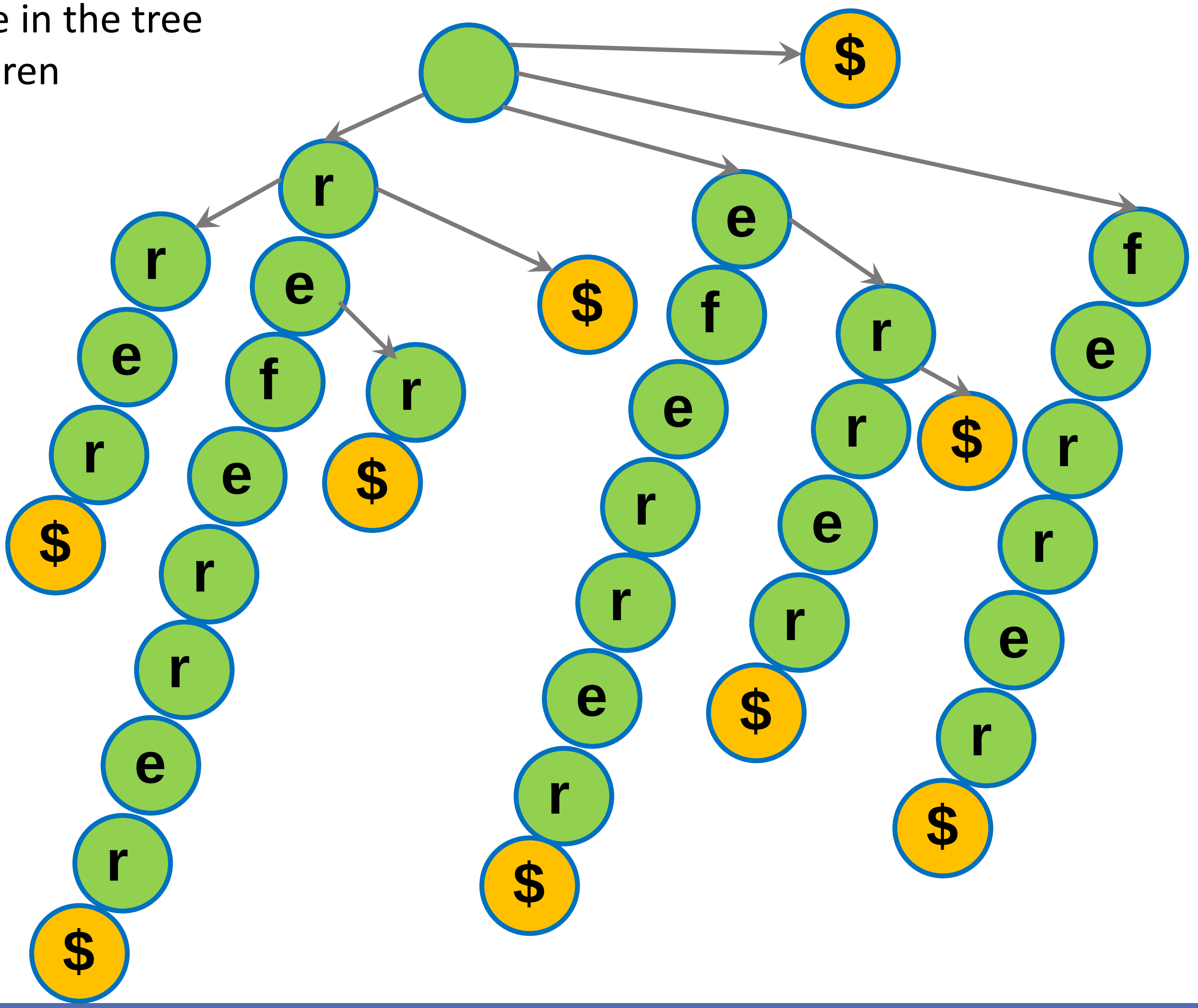
Finding longest repeated substring

Quiz time!
https://flux.qa - RFIBMB



Finding longest repeated substring

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



Space complexity of suffix trie

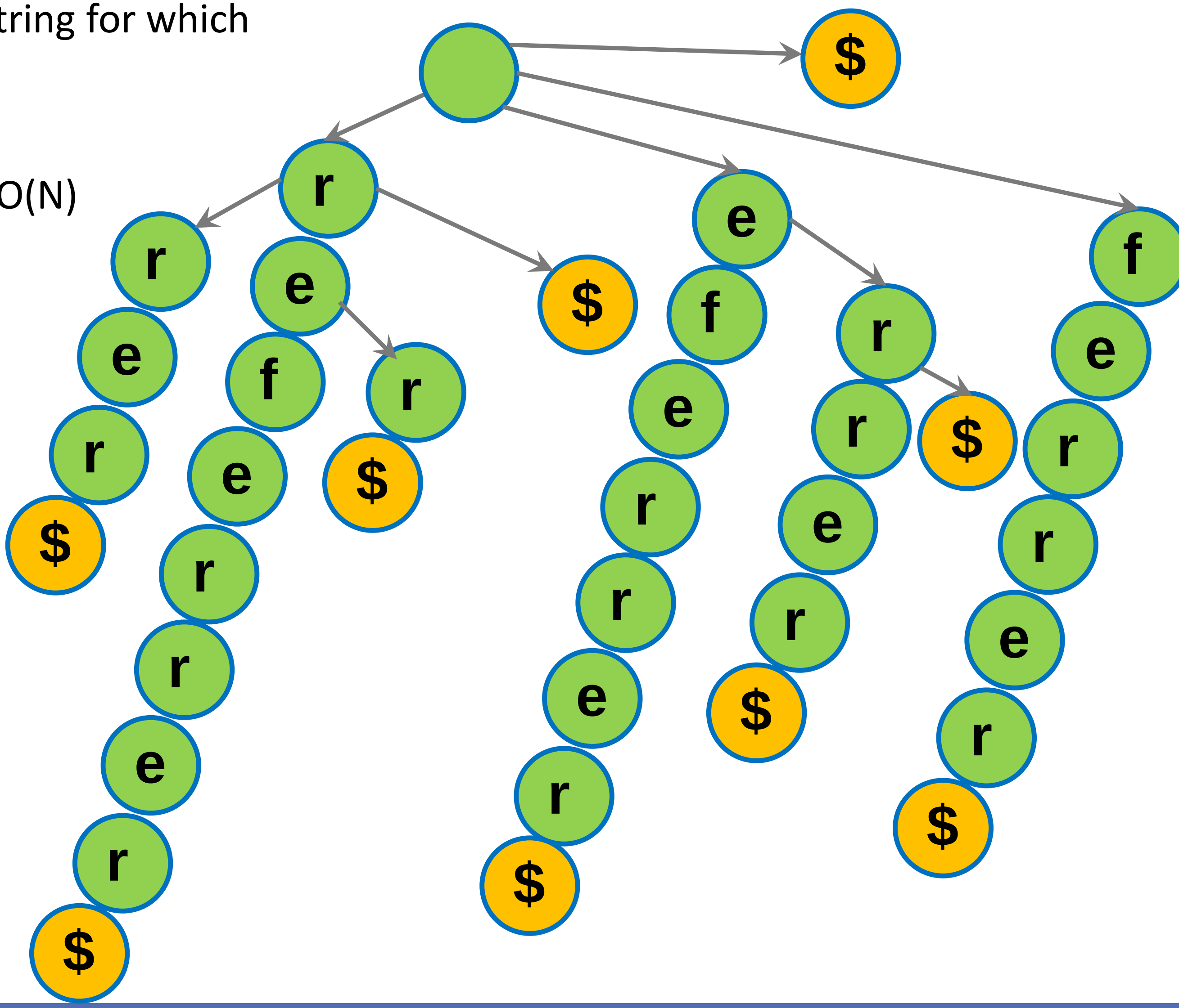
Let N be the size of the string for which suffix trie is constructed

Space complexity?:

- # number of suffixes: O(N)
- Cost for each suffix is

linear to its size

Total space cost: $O(N^2)$

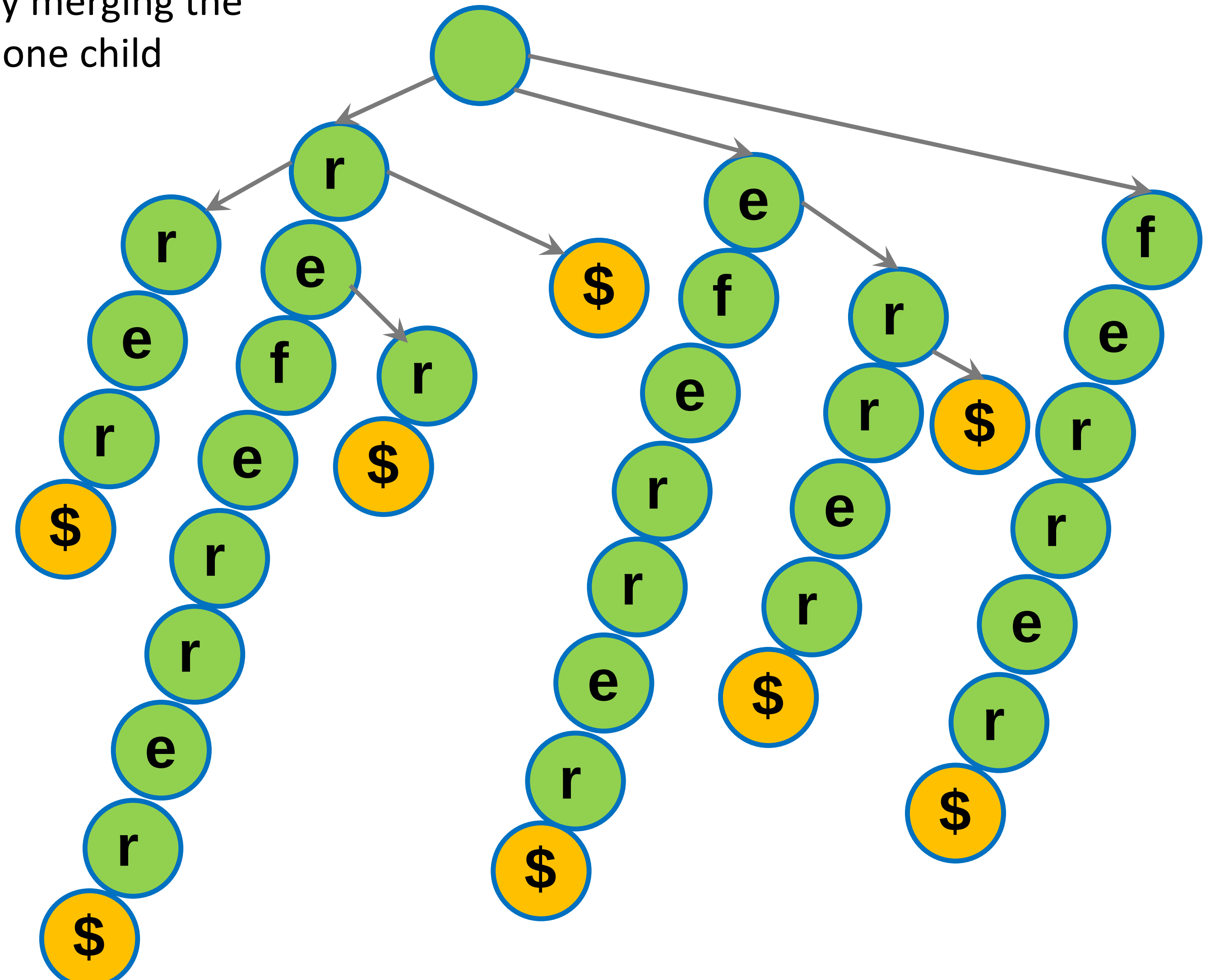


Outline

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

Suffix Tree is a compact Suffix Trie

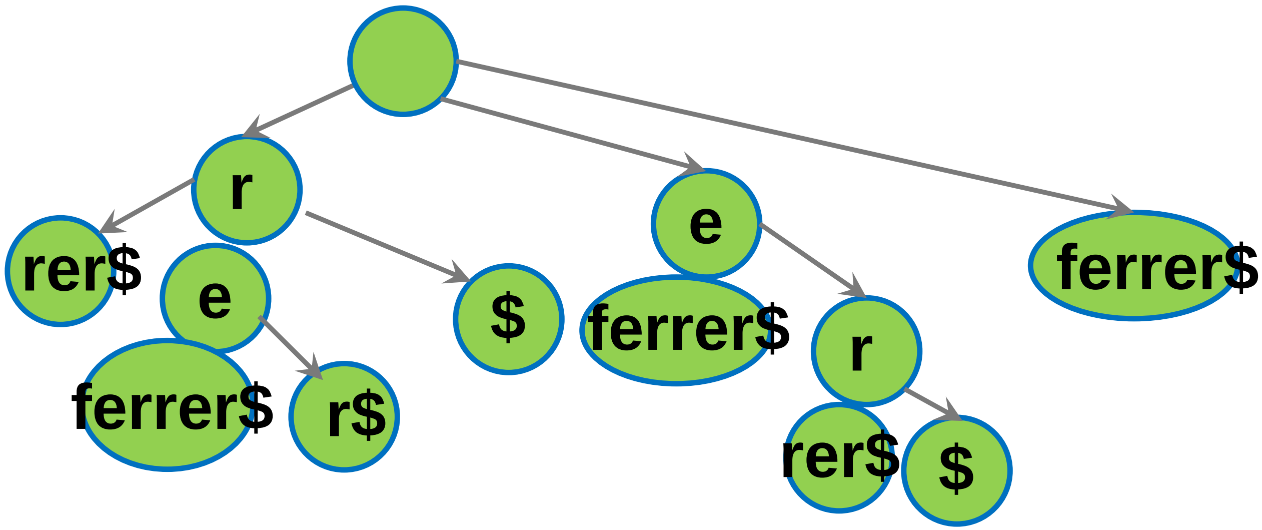
- Compress branches by merging the nodes that have only one child



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!
https://flux.qa - RFIBMB



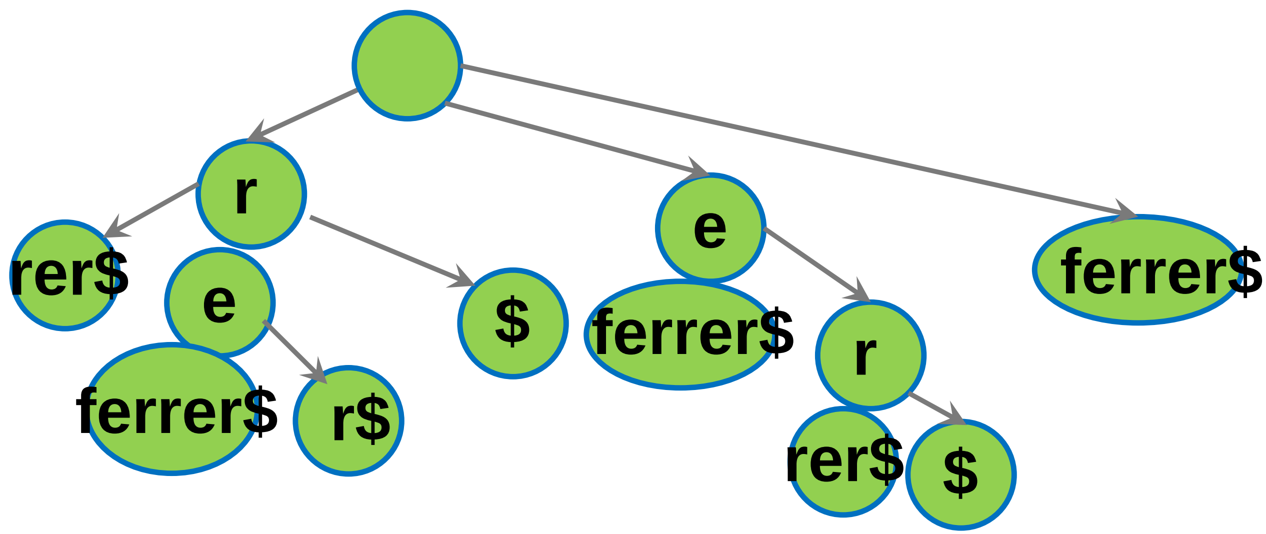
Space complexity of suffix tree

- 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 (3,6)

rer is represented as (6,3)

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



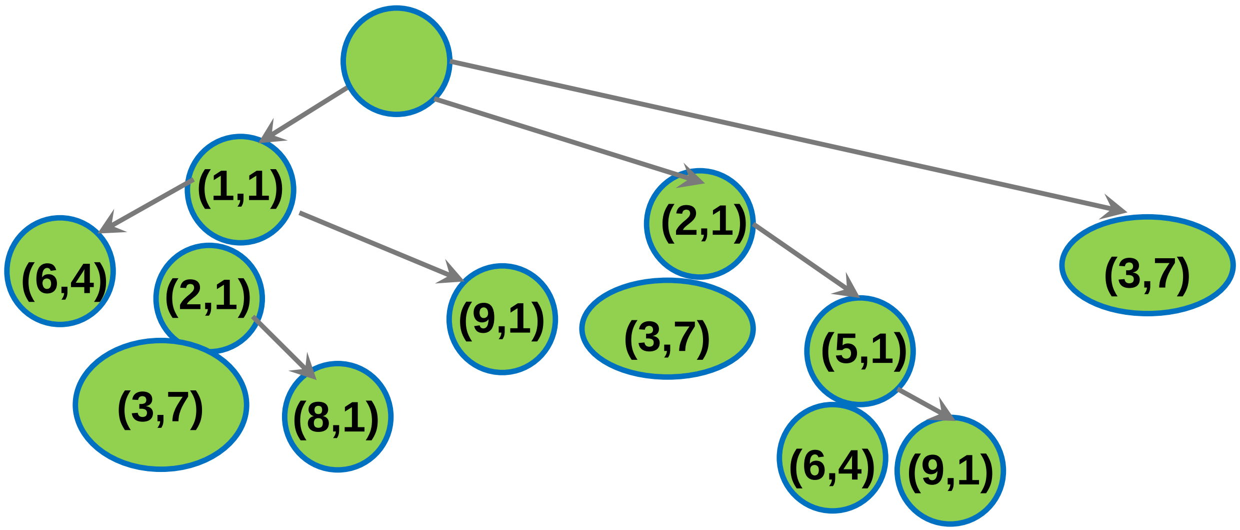
Space complexity of suffix tree

- 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)

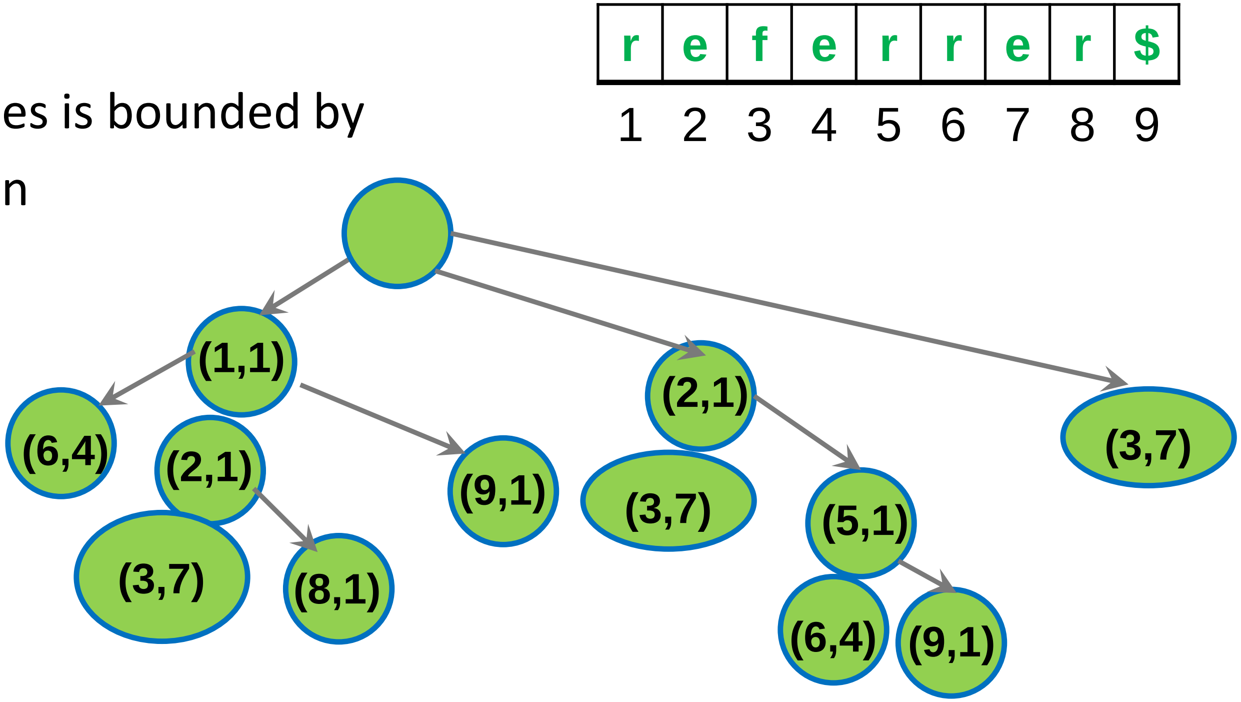
rer$ is represented as (6,4)

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



Space complexity of suffix tree

- Every (internal) node has at least 2 children
- There are exactly n leaves
- There are at most n/2 nodes in the layer above the leaves
- There are at most n/4 notes in the layer above that
- …
- The total number of nodes is bounded by
- N + n/2 + n/4 + … + 1 < 2n



Time complexity of constructing suffix tree

- The algorithm described earlier inserts O(N) suffixes
- Insertion cost of each suffix is linear in the size of suffix
- Average suffix size is O(N)
- Thus, total time complexity is O(N^2)

It is possible to construct suffix tree in O(N)

- Esko Ukkonen in 1995 gave a beautiful (but involved) algorithm to construct a Suffix Tree in linear time. If you every get interested in doing this in linear time, consider reading the source:

Ukkonen, E. (1995). "On-line construction of sux trees". Algorithmica 14 (3): 249260.

Outline

1. Introduction
2. Trie
   A. Construction
   B. Query Processing
3. Suffix Trie
   A. Construction
   B. Query Processing
   C. Suffix Tree
4. Suffix Array
   A. Introduction
   B. Query Processing
   C. 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)

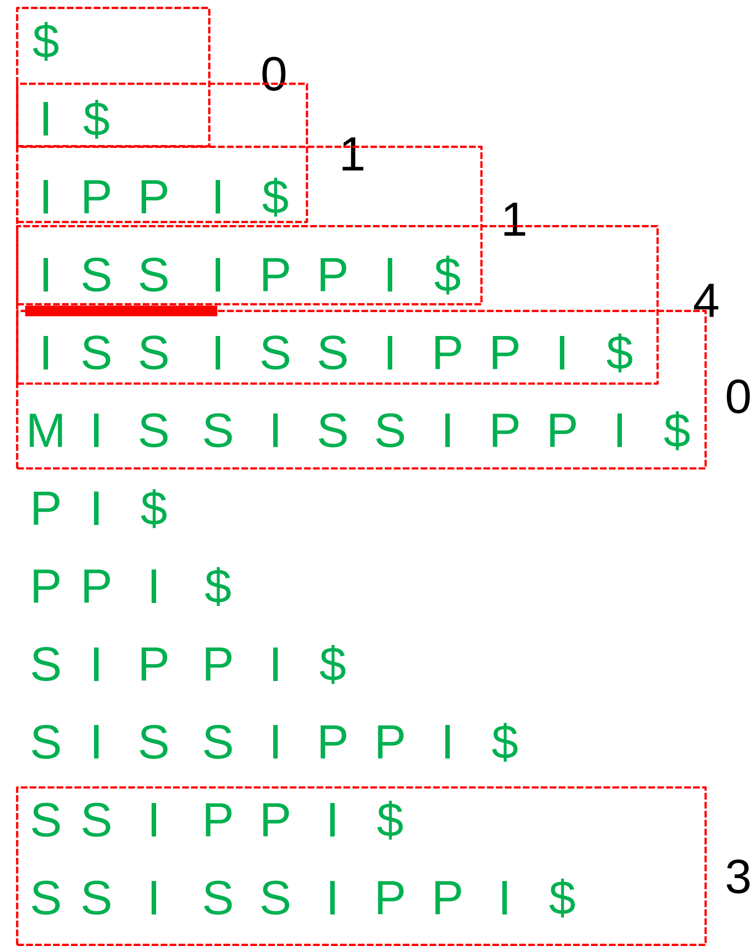
$
I $
I P P I $
I S S I P P I $
I S S I S S I P P I $
M I S S I S S I P P I $
P I $
P P I $
S I P P I $
S I S S I P P I $
S S I P P I $
S S I S S I P P I $

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 suffix array + 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:

- $O(N^2)$

- Can we do better?

Yes! Suffix Array reduces it to O(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	$

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!

https://flux.qa - RFIBMB

Outline

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

Suffix Array

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

Substring Search:

- Do a binary search

Example:

Search “IPP” in the string.

- Initially, the search space is whole Suffix Array

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 Array

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

Substring Search:

- Do a binary search

Example:

Search “IPP” in the string.

- Initially, the search space is whole Suffix Array
- Look at the middle element in range, i.e.,
  - At index 6 in Suffix array
  - This is Suffix #1 (“MISSISSIPPI$”)

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 Array

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

Substring Search:

- Do a binary search

Example:

Search “IPP” in the string.

- Initially, the search space is whole Suffix Array
- Look at the middle element in range, i.e.,
  - At index 6 in Suffix array
  - This is Suffix #1 (“MISSISSIPPI$”)
- Since “IPP” < “MISSISSIPPI”, substring if present must be above this element
- Look at the middle element in range, i.e.,
  - At index 3 in Suffix array
  - This is suffix with ID 8 (“IPPI$”)

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 Array

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

Substring Search:

- Do a binary search

Example:

Search “IPP” in the string.

- Initially, the search space is whole Suffix Array
- Look at the middle element in range, i.e.,
  - At index 6 in Suffix array
  - This is Suffix #1 (“MISSISSIPPI$”)
- Since “IPP” < “MISSISSIPPI”, substring if present must be above this element
- Look at the middle element in range, i.e.,
  - At index 3 in Suffix array
  - This is suffix with ID 8 (“IPPI$”)
- Found!!!

Time Complexity:

- O(M log N)
- Can be improved to O(M) using LCP array (beyond the scope of this unit)

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 Array

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

Longest Repeated Substring:

- Algorithm is same as on “Sorted Suffixes”
- Time complexity is also the same

Time Complexity:

- $O(N^2)$
- Can be improved to O(N) using LCP array (beyond the scope of this unit)

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 $

Construction Cost of Suffix Array

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

We need to generate N suffixes and then sort all N suffixes.

Time Complexity (with Merge Sort):
- O(N log N) comparisons
- Each comparison takes O(N)
- Total cost: $O(N^2 \log N)$

Time Complexity (with Radix Sort):
- O(N) passes
- Each pass takes O(N)
- Total cost: $O(N^2)$

Space required during construction:
- $O(N^2)$ – we need all suffixes during sorting

Can we do better?
- Yes, using prefix doubling approach
- $O(N \log^2 N)$ time complexity
- O(N) space required during construction

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 $

Outline

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
- Sort suffixes on their 1st characters

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
- Sort suffixes on first 1 characters

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

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 $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	4	S I S S I P P I $
9	7	S I P P I $
11	3	S S I S S I P P I $
11	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
- Sort suffixes on first 1 characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 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 $
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
- Sort suffixes on their 1st characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 characters
- …

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	$

Basic Idea:

- Generate suffixes
- Sort suffixes on their 1st characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 characters
- …
- Sort suffixes on all N characters

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	$

Basic Idea:

- Generate suffixes
- Sort suffixes on their 1st characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 characters
- …
- Sort suffixes on all N characters

- The last sort **alone** takes NlogN with a comparison cost of N
- This is N^2logN

- We need to speed up the comparisons

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	$

Basic Idea:

- Generate suffixes
- Sort suffixes on their 1st characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 characters
- …
- Sort suffixes on all N characters

- **Suppose we could compare in O(1)**

- logN sorts
- O(NlogN) for each
- O(NlogNlogN) = O(Nlog^{2}N)

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)
- 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 $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	4	S I S S I P P I $
9	7	S I P P I $
10	3	S S I S S I P P I $
10	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		
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 $
6	1	M I	S S I S S I P P I $
7	10	P I	$
8	9	P P	I $
9	4	S I	S S I P P I $
9	7	S I	P P I $
10	3	S S	I S S I P P I $
10	6	S S	I P P I $

O(1) Comparison

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

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	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 $
6	1	M I S S I S S I P P I $
7	10	P I $
8	9	P P I $
9	4	S I S S I P P I $
9	7	S I P P I $
10	3	S S I S S I P P I $
10	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	
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 $
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 $

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	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 $
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 $

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 $
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 $

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	
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 $
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

Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	6	4	12	10	4	11	9	3	8	7	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	
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	6	4	12	10	4	11	9	3	8	7	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

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 $

O(1) Comparison

Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	6	4	12	10	4	11	9	3	8	7	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 (by hand in lecture)

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 (by hand in lecture)

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 (by hand in lecture)

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

Rank the first characters of each suffix

In practice we would do this using ord(), but since ranks are only comparative, the actual values don't matter, just their order. So we use numbers starting at 1

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

Lets do an example (by hand in lecture)

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

Sort SA by ranks

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

Lets do an example (by hand in lecture)

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

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

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

Lets do an example (by hand in lecture)

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

Now sort on first 2 characters

For each comparison, we use the trick outlined in the previous section

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

Lets do an example (by hand in lecture)

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

Now we update the ranks

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

Lets do an example (by hand in lecture)

Make an array, “Temp” to hold the new ranks

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

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

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

Lets do an example (by hand in lecture)

For each pair of adjacent suffixes, compare them

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

Lets do an example (by hand in lecture)

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

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

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

Lets do an example (by hand in lecture)

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

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

Lets do an example (by hand in lecture)

If they have the same rank

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

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

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

Lets do an example (by hand in lecture)

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

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

We need to use the O(1) trick

Lets do an example (by hand in lecture)

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

Rank	SA			
1	12	$		
2	11	I	$	
2	8	I	P	P I $
2	2	I	S	S I S S I P P I $
2	5	I	S	S I P P I $
3	1	M	I	S S I S S I P P I $
4	10	P	I	$
4	9	P	P	I $
5	4	S	I	S S I P P I $
5	7	S	I	P P I $
5	3	S	S	I S S I P P I $
5	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 (by hand in lecture)

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

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

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

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

Lets do an example (by hand in lecture)

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

Rank	SA		
1	12	$	
2	11	I $	
2	8	I P	P I $
2	2	I S	S I S S I P P I $
2	5	I S	S I P P I $
3	1	M I	S S I S S I P P I $
4	10	P I	$
4	9	P P	I $
5	4	S I	S S I P P I $
5	7	S I	P P I $
5	3	S S	I S S I P P I $
5	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 (by hand in lecture)

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

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

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

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

Lets do an example (by hand in lecture)

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

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

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

Continue in this way

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

Lets do an example (by hand in lecture)

Continue in this way

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

Lets do an example (by hand in lecture)

Continue in this way

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

Lets do an example (by hand in lecture)

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

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

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

Continue in this way

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

Lets do an example (by hand in lecture)

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

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

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

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

Lets do an example (by hand in lecture)

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

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

Lets do an example (by hand in lecture)

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 the suffixes sorted by the first 2. Go for 4!

Lets do an example (by hand in lecture)

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	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 (by hand in lecture)

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	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 $

Lets do an example (by hand in lecture)

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	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 $

Summary

Take home message

- Tries, Suffix trees and Suffix array provide efficient text search and pattern matching (typically linear in number of characters in string)

Things to do (this list is not exhaustive)

- Implement Trie, Suffix trees and Suffix array and run various pattern matching queries

Coming Up Next

- Burrows-Wheeler Transform - A beautiful space-time efficient pattern matching algorithm on text