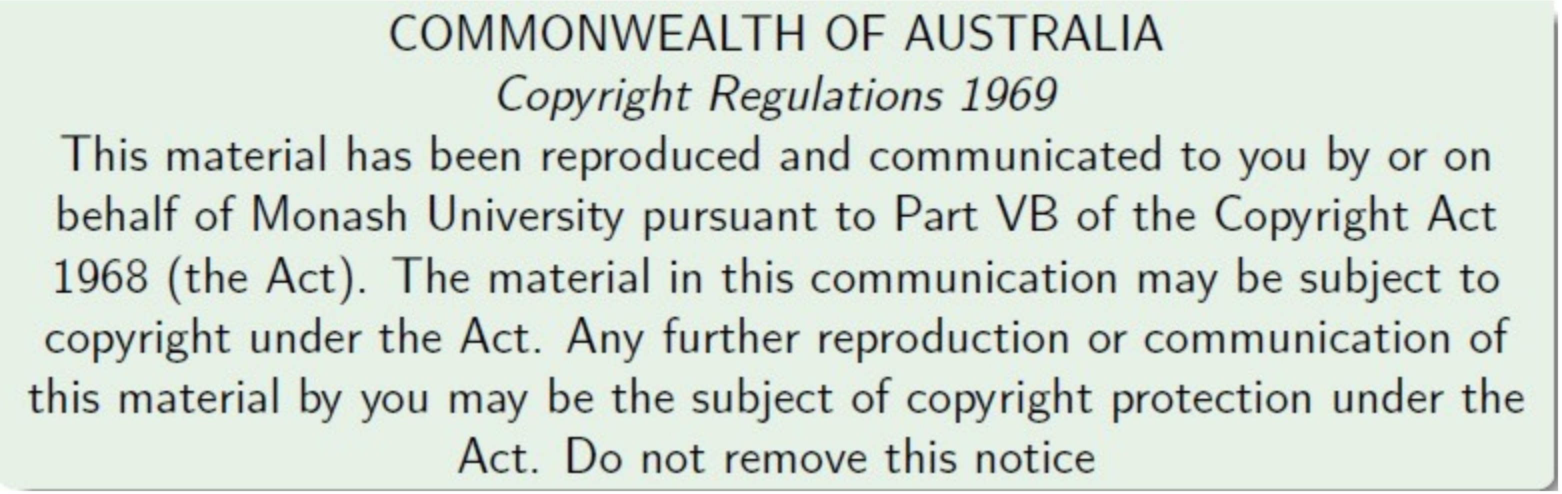
Faculty of Information Technology, Monash University



FIT2004: Algorithms and Data Structures

Week 6: Retrieval Data Structures for Strings

Lecturer: Reza Haffari

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)
- Tries: http://en.wikipedia.org/wiki/Trie/
- Suffix Trees: http://www.allisons.org/ll/AlgDS/Tree/Suffix/
- 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.

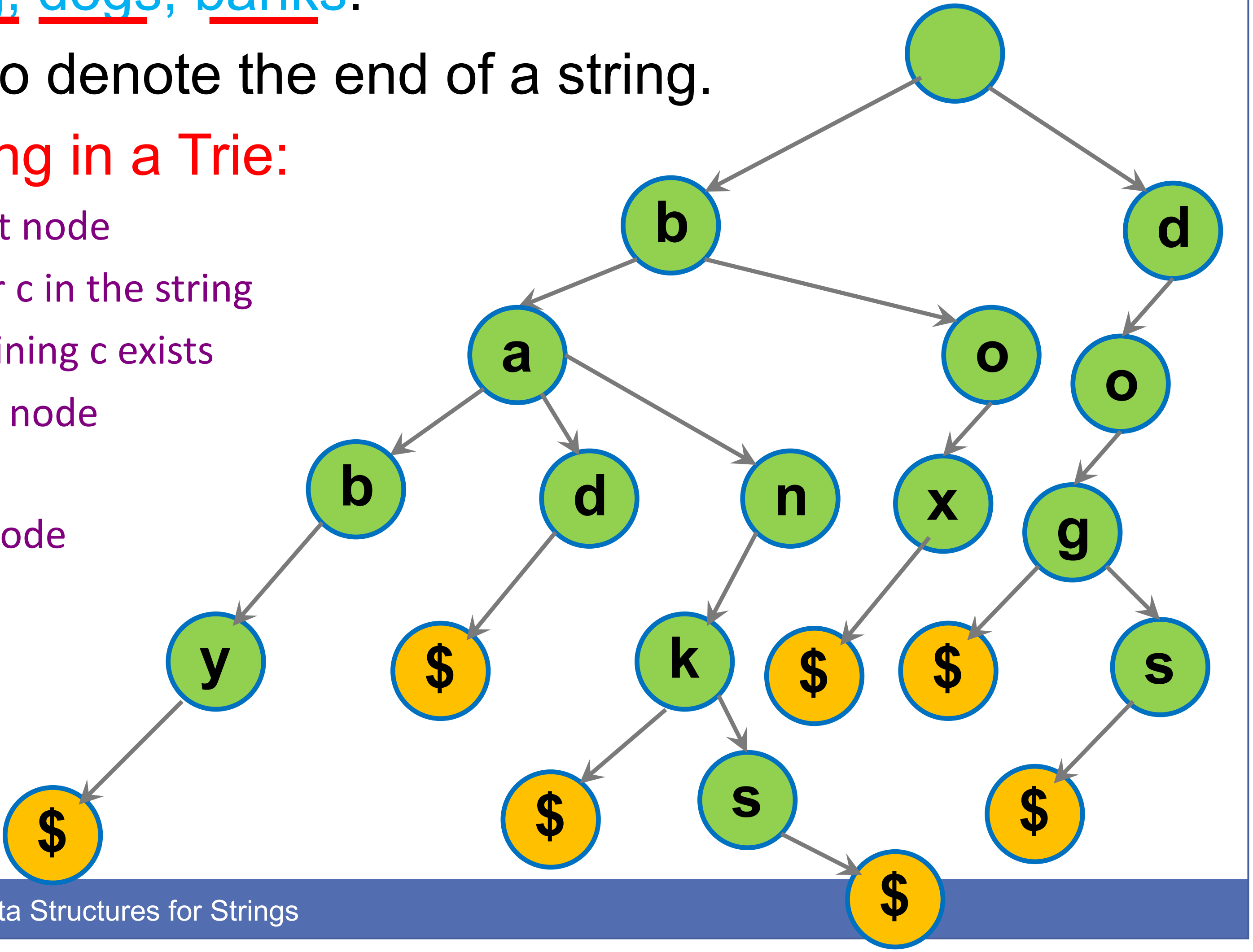
Trie Example: Insertion

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.

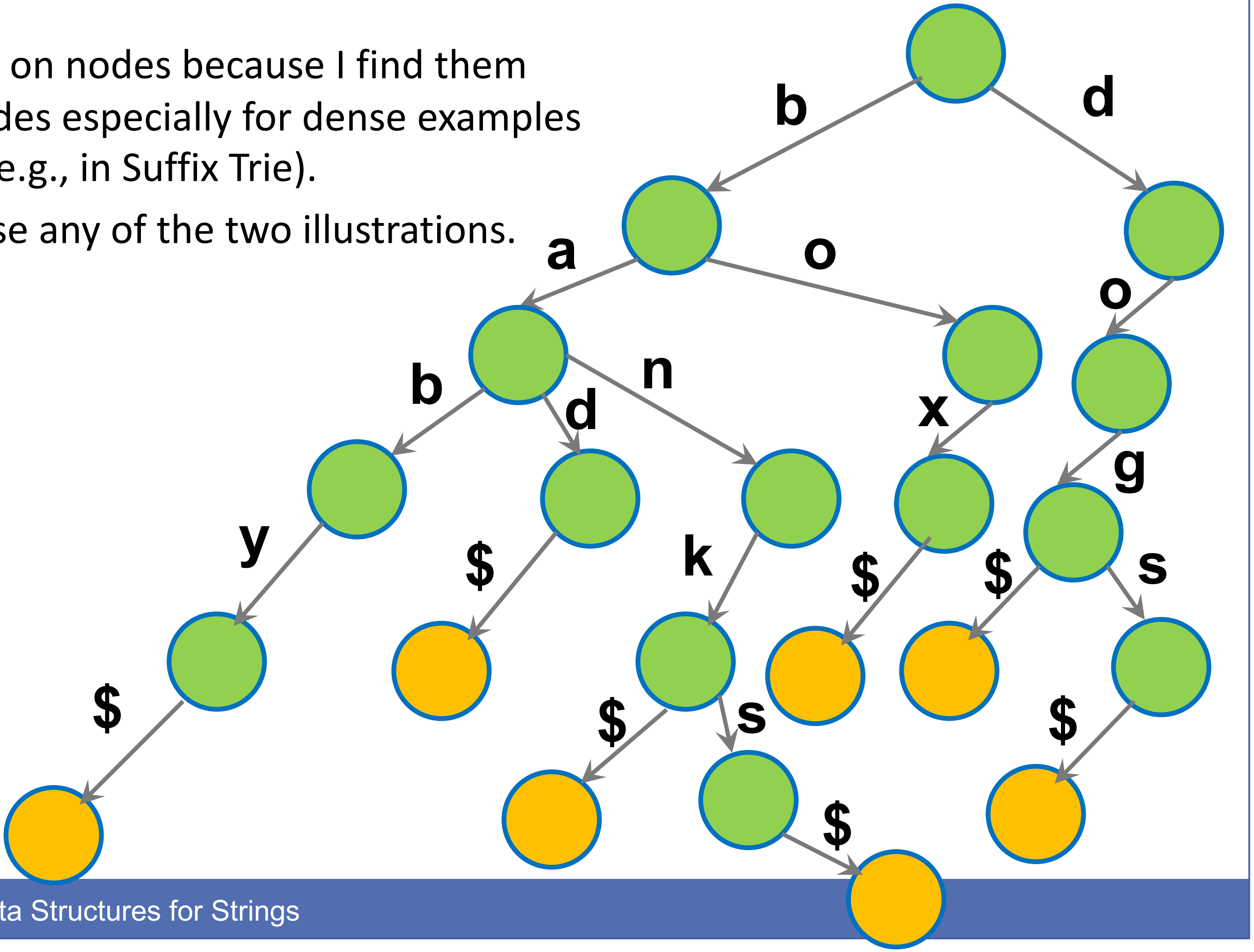
Inserting a string in a Trie:

- 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.



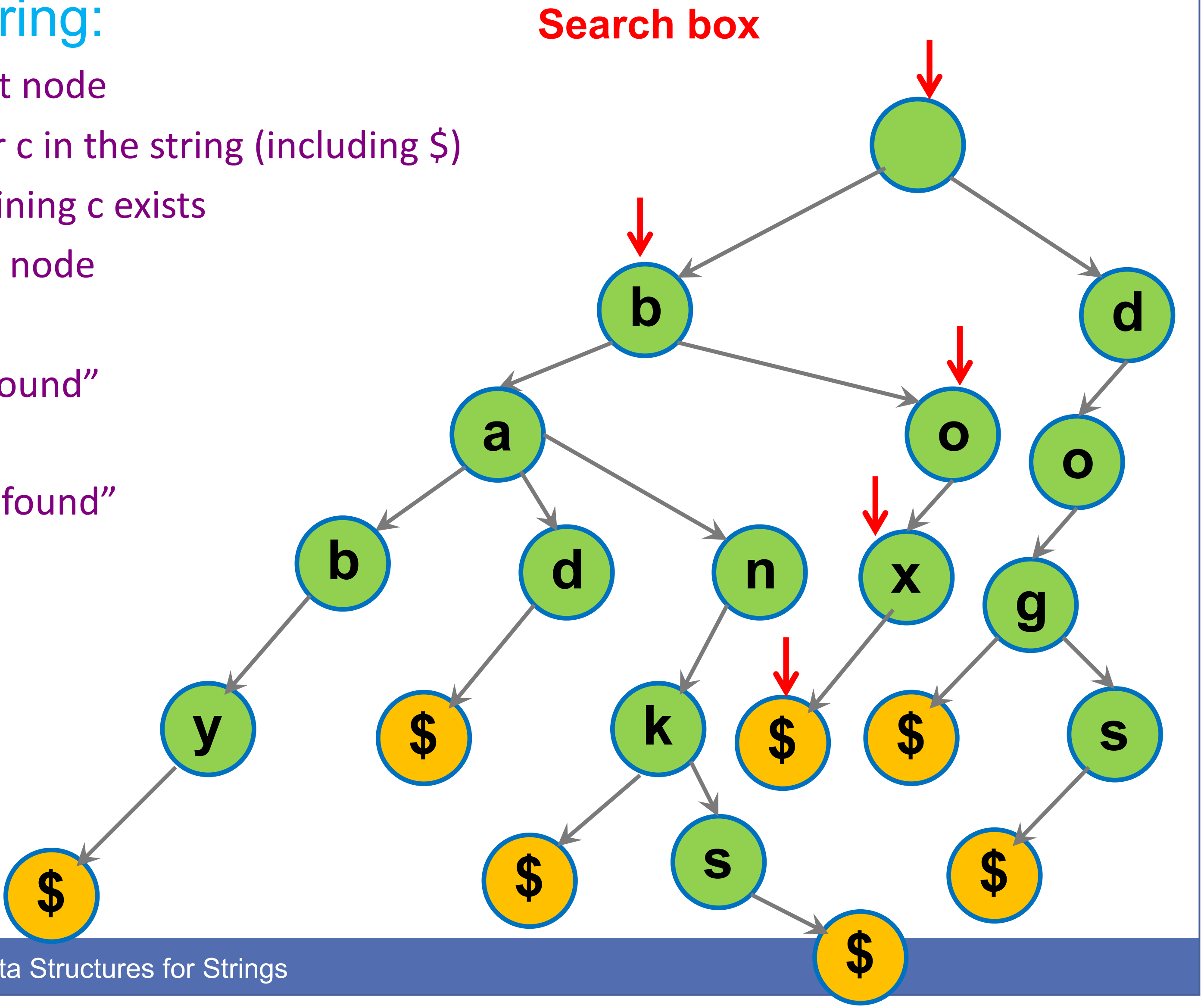
Outline

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”

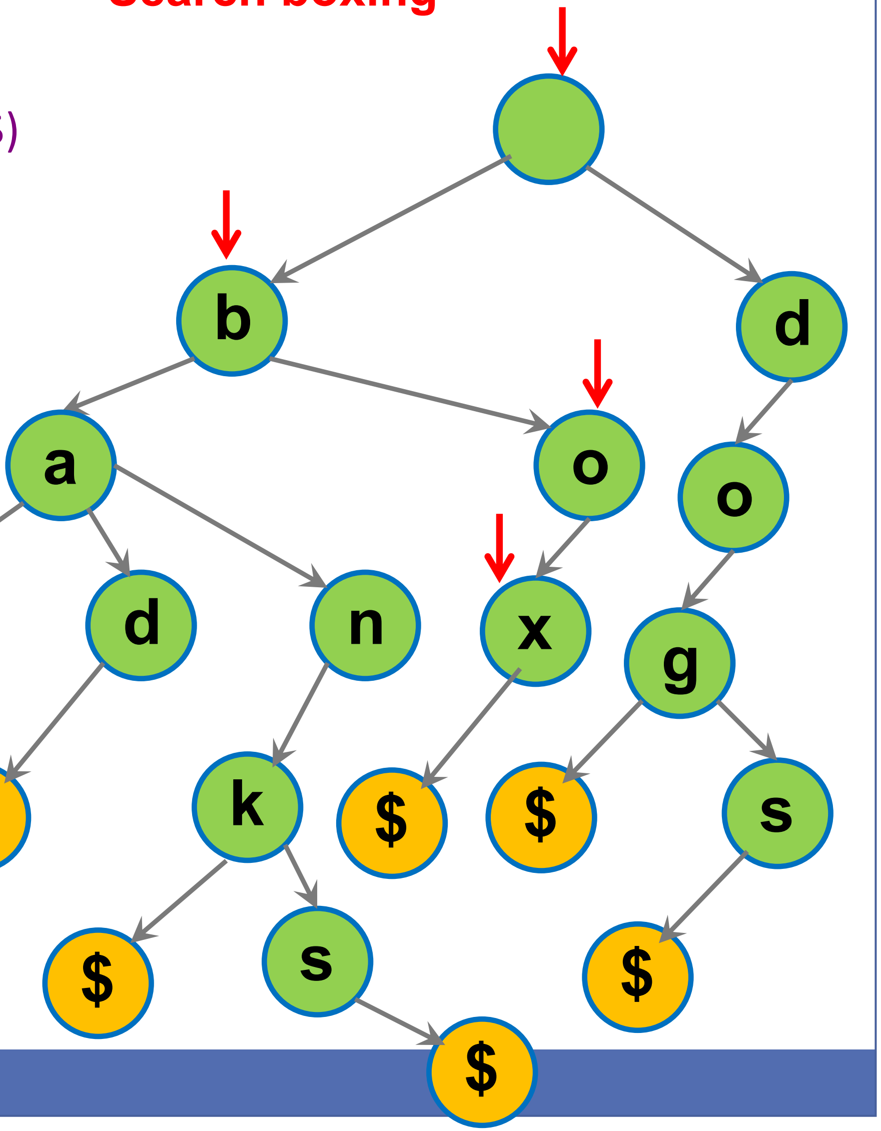


Trie Example: Search

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" ←

Search boxing



Trie Example: Search

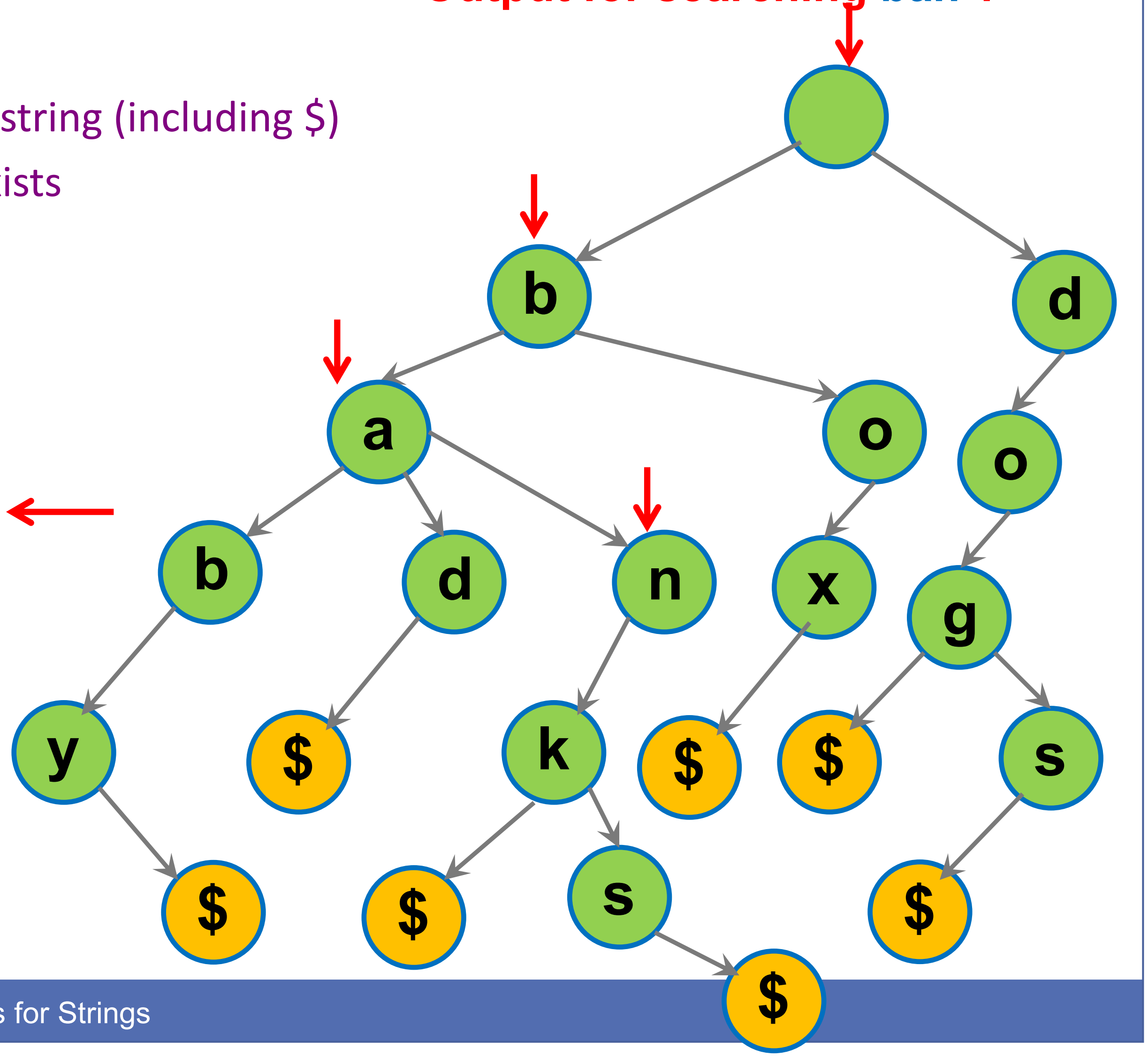
Searching a string:

- 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?
  - O(1) if using an array implementation (e.g., direct-addresing) or considering alphabet size a constant (e.g., 26 for English)

Output for searching ban ?



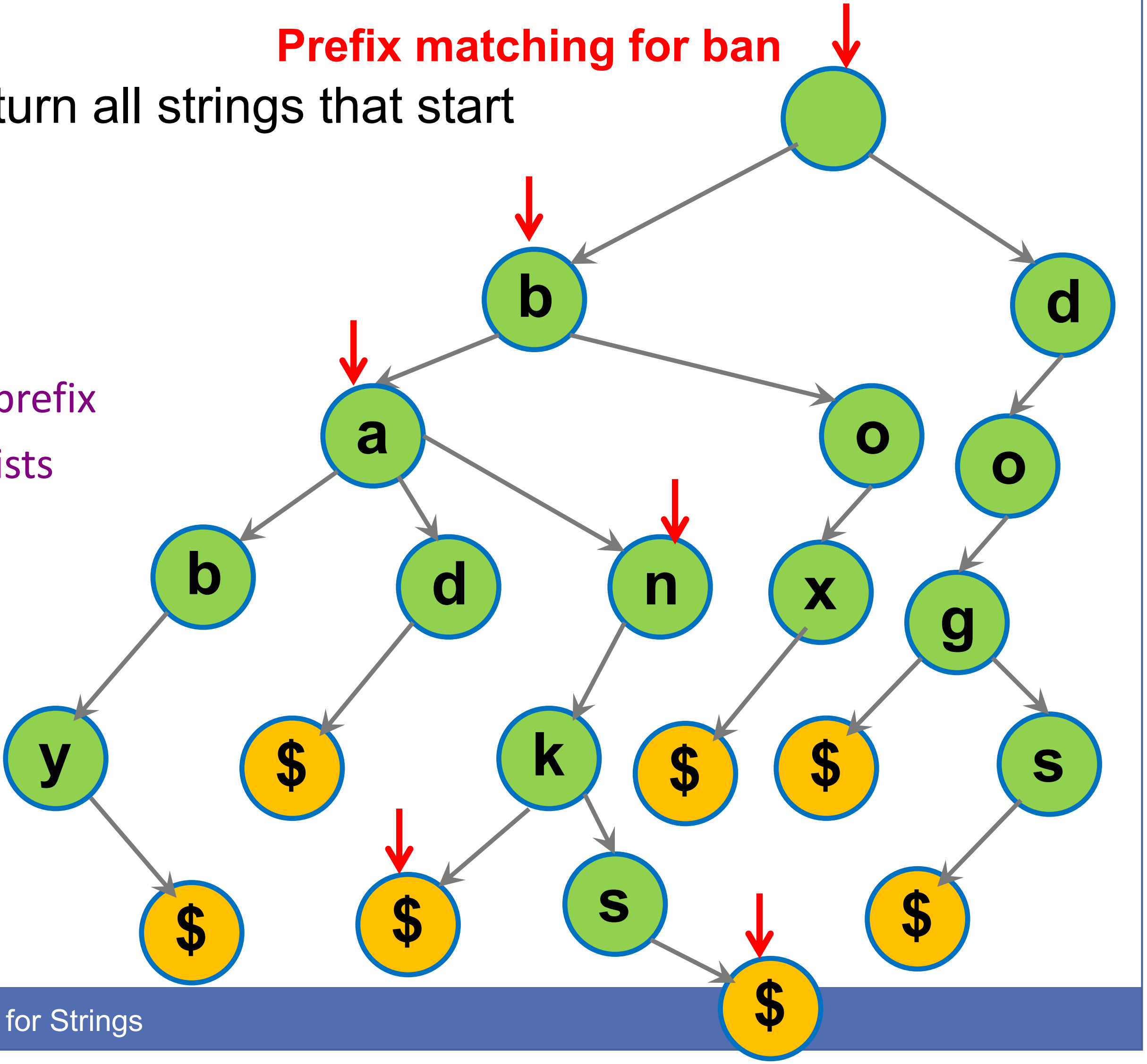
Trie Example: Prefix Matching

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

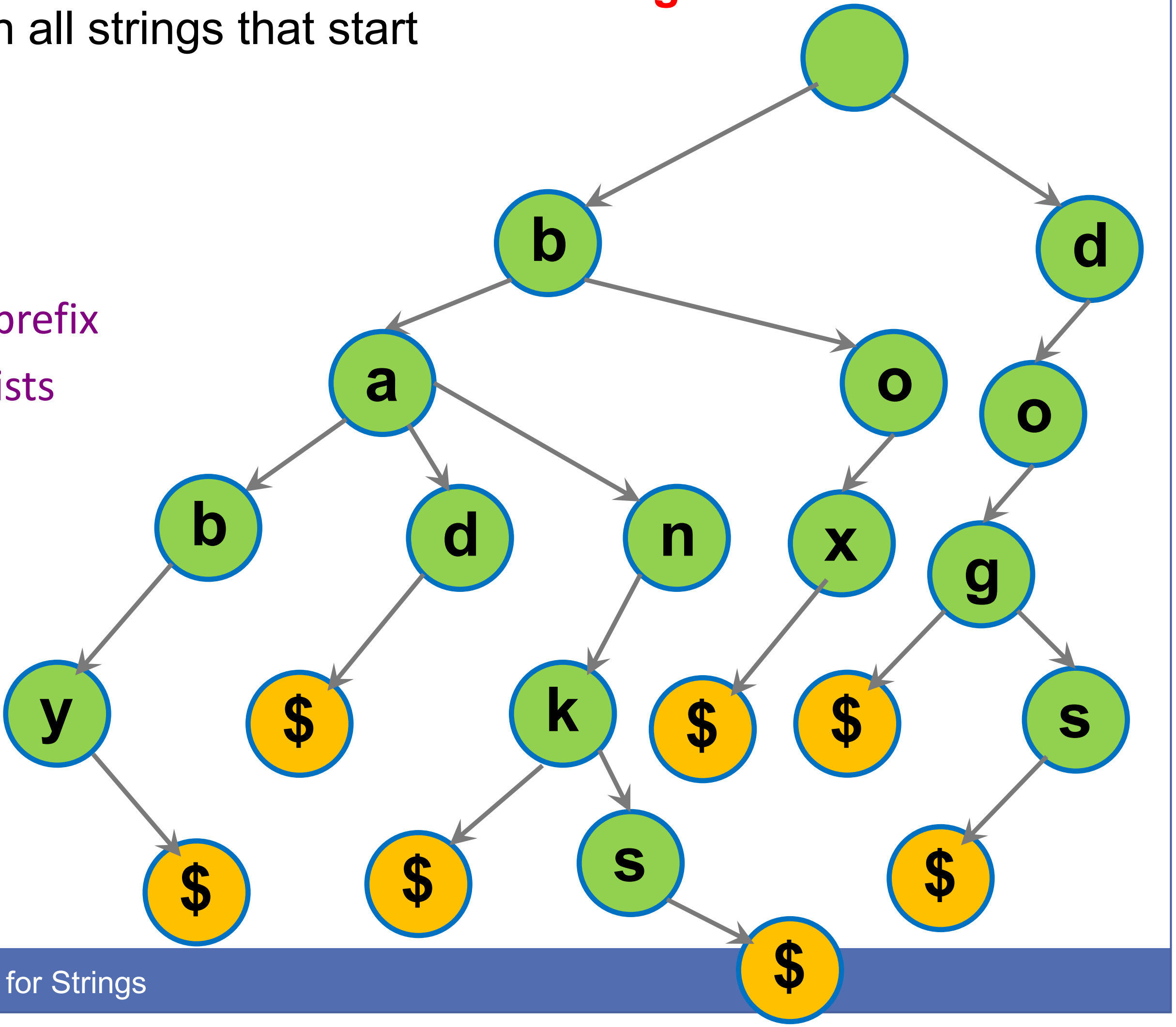
Prefix matching returns every string in text that has the given string as its **prefix**.

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

Prefix matching for b



Implementing a Trie

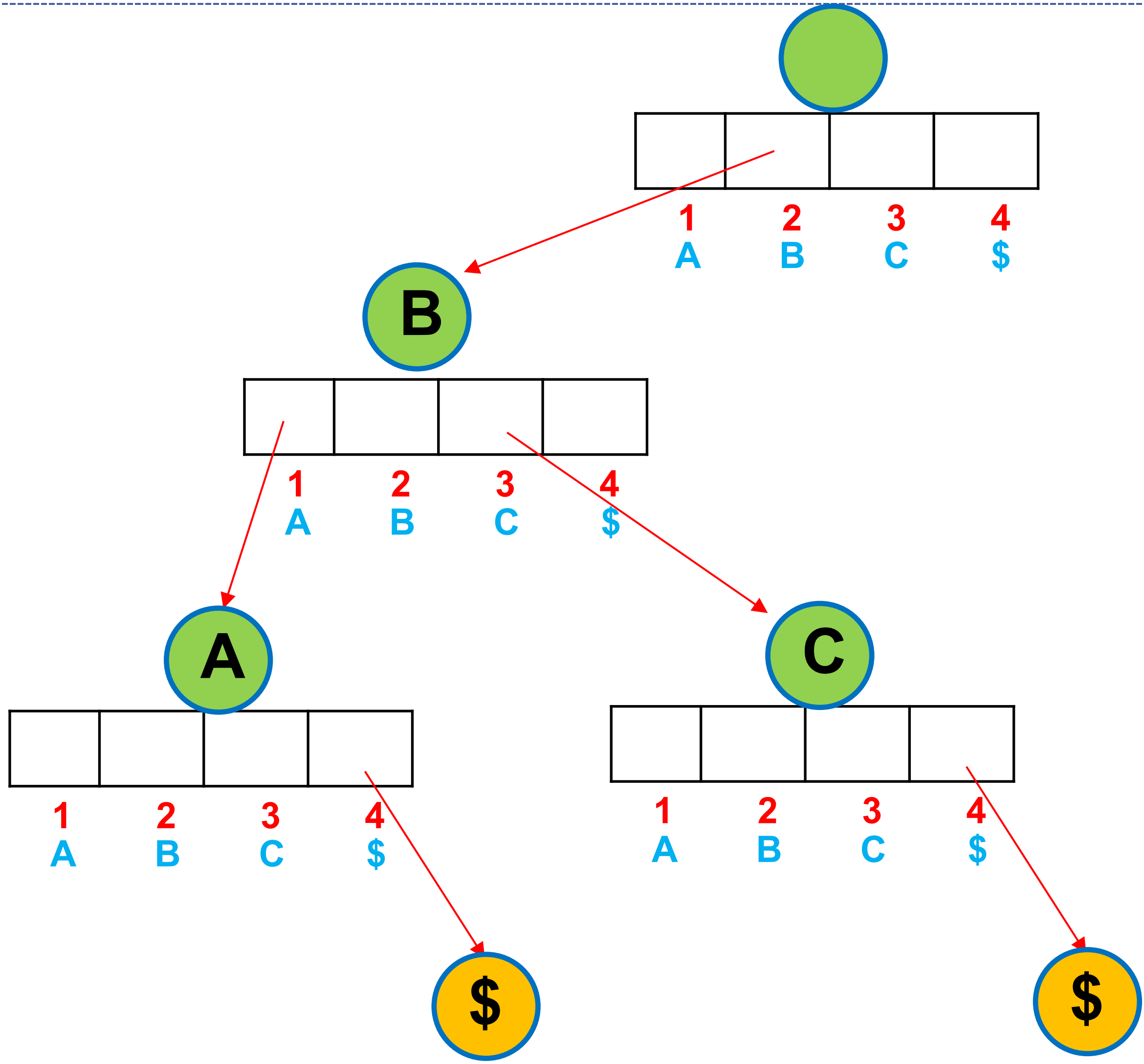
Implementation using an array:

- 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.
- Requires a lot of wasteful space, as many nodes, as you descend a trie, will have more and more children set to nil.

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, much wasted space with a simple implementation of a Trie, where
  - 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.

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 Trie

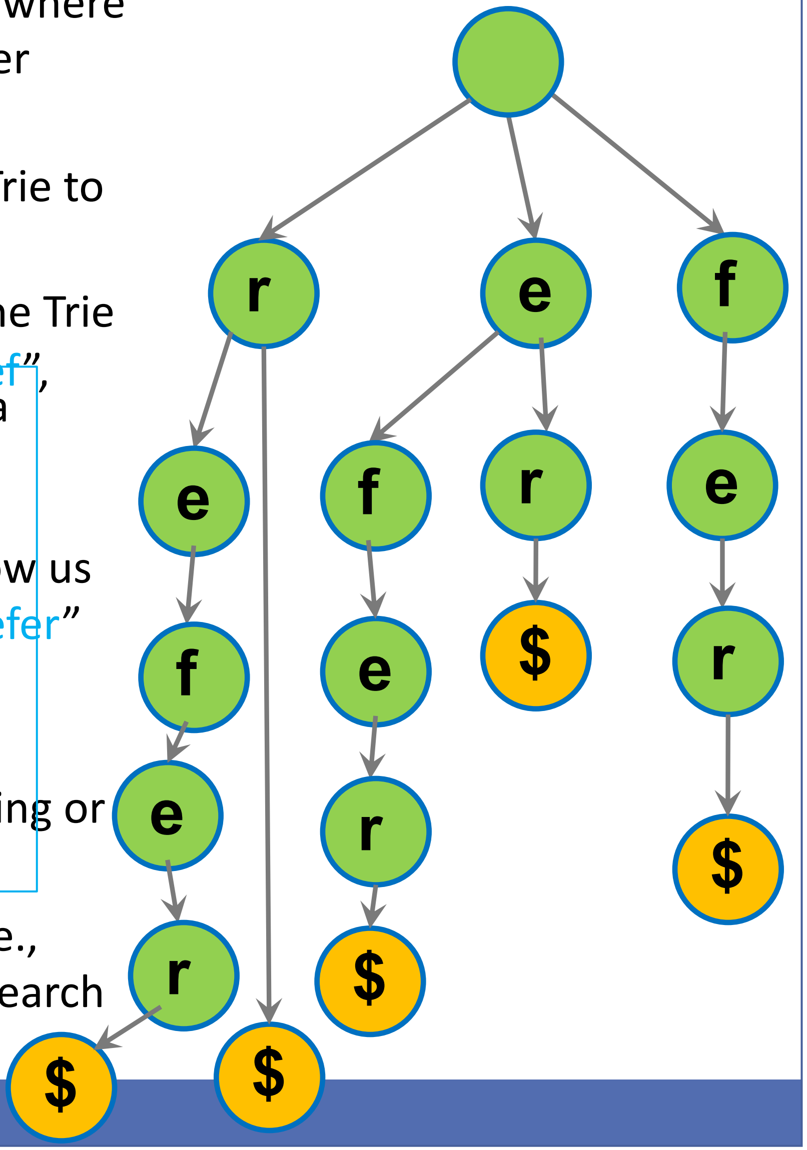
- We saw that a Trie allows prefix matching in O(M) where M is the length of the prefix, e.g., checking whether "ref" is a prefix of "refer".
- What about substring matching? Can we use the Trie to check if "ef" is a substring of "refer" in O(M).
- No! because "ef" is not a prefix of "refer". Using the Trie of "refer", we can only check whether "r", "re", "ref", "refe", and "refer" are the substrings.

Idea:

- What about if we add "efer" in the Trie? It will allow us efficiently checking whether "e", "ef", "efe", and "efer" are also in the string.
- What about if we also add "fer". This will allow us checking whether "f", "fe", and "fer" are in the string or not.
- In short, if we add all suffixes of the string refer (i.e., refer, efer, fer, er, r) in the Trie, we can efficiently search every substring of refer in the Trie.

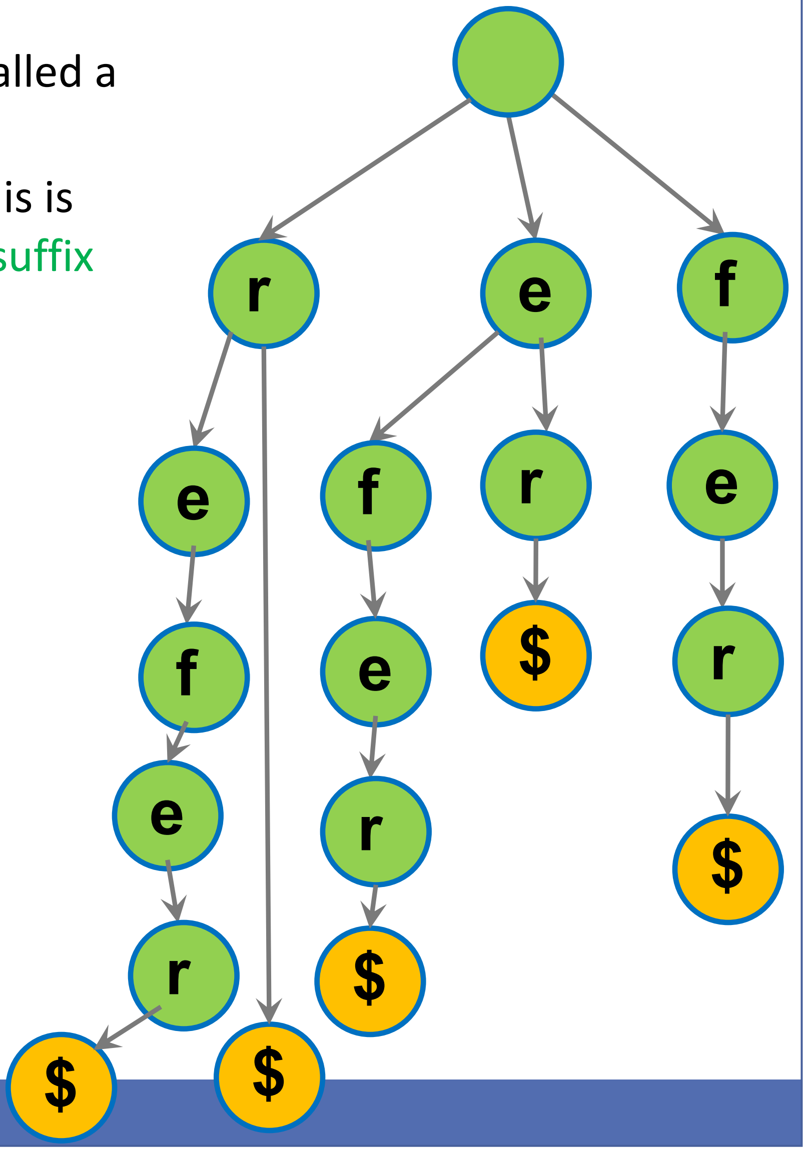
Prefix for a string s[1…M] is a string s[1…X] where X≤M. (e.g., refe is a prefix of refer.)

Suffix for a string s[1…M] is a string s[X…M]. E.g., fer is a suffix of refer.



Suffix Trie

- Consider some text, e.g., “refer”.
- A Trie constructed using all suffixes of the text is called a Suffix Trie.
- A suffix trie allows efficient substring matching. This is because every substring is a prefix of at least one suffix present in the tree.



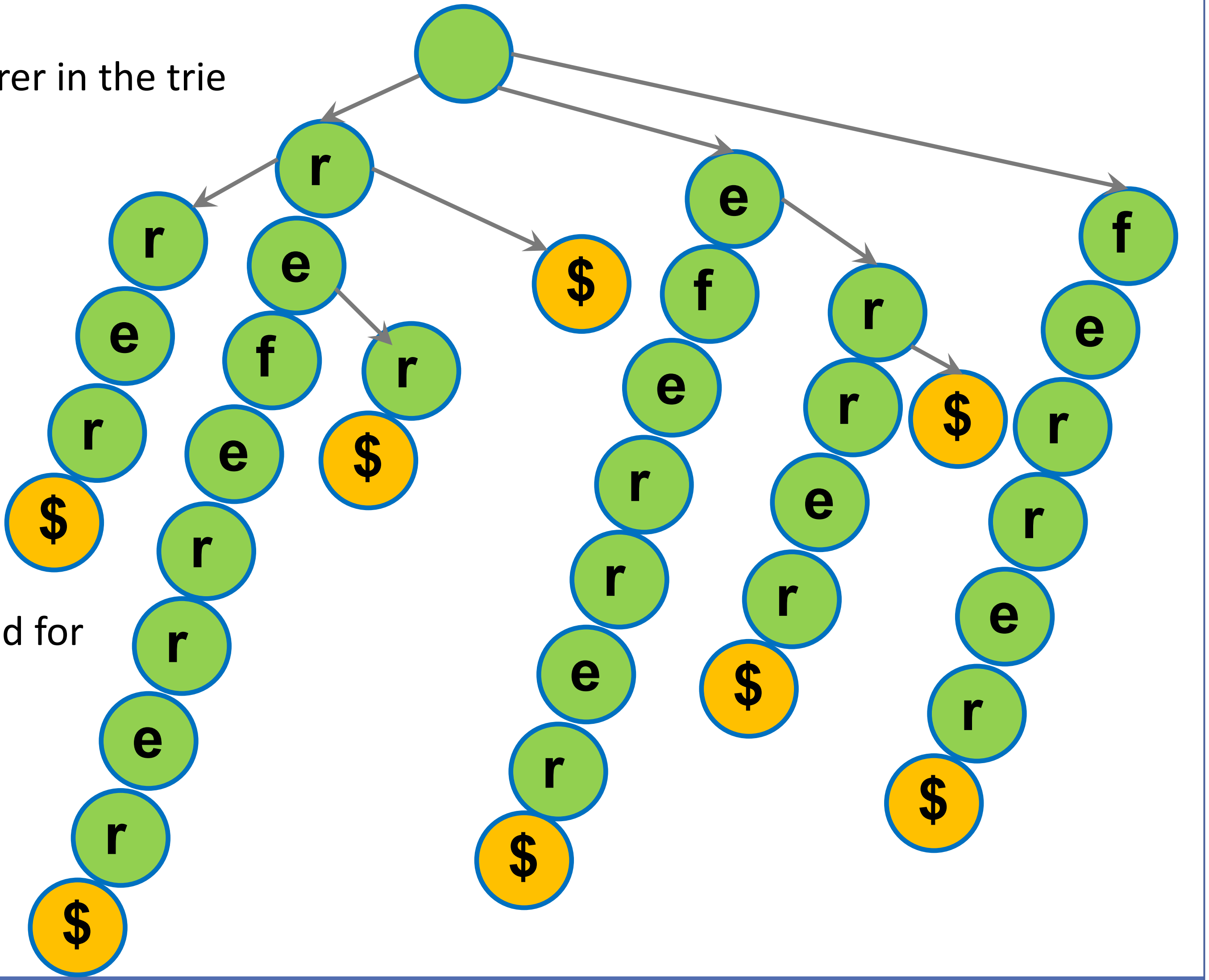
Constructing Suffix Trie

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

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

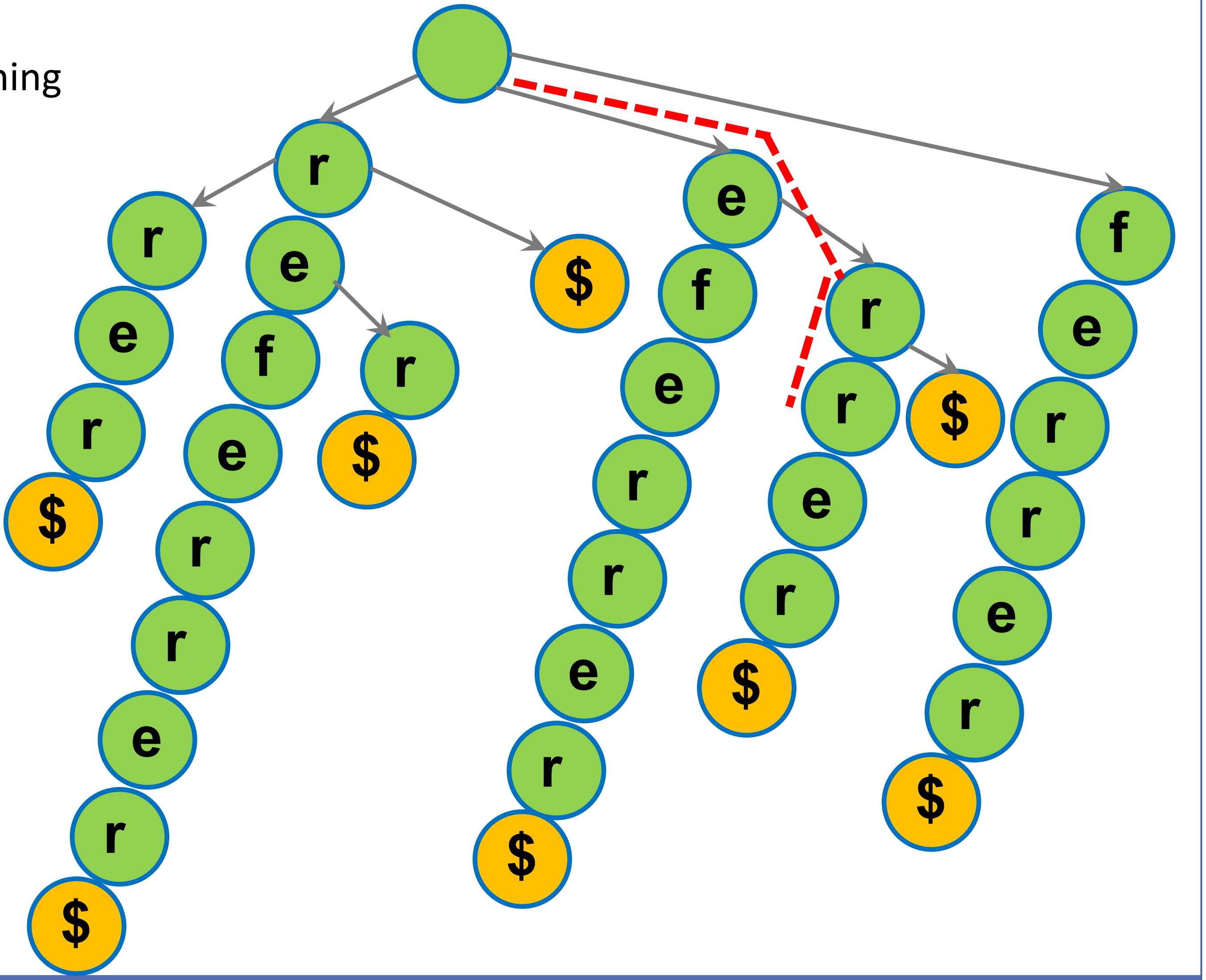
Substring search on Suffix Trie

Substring search for str

- Similar to prefix matching

E.g., search “err”

search “fers”



Time Complexity:

O(M) where M is the length of substring

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 “e”

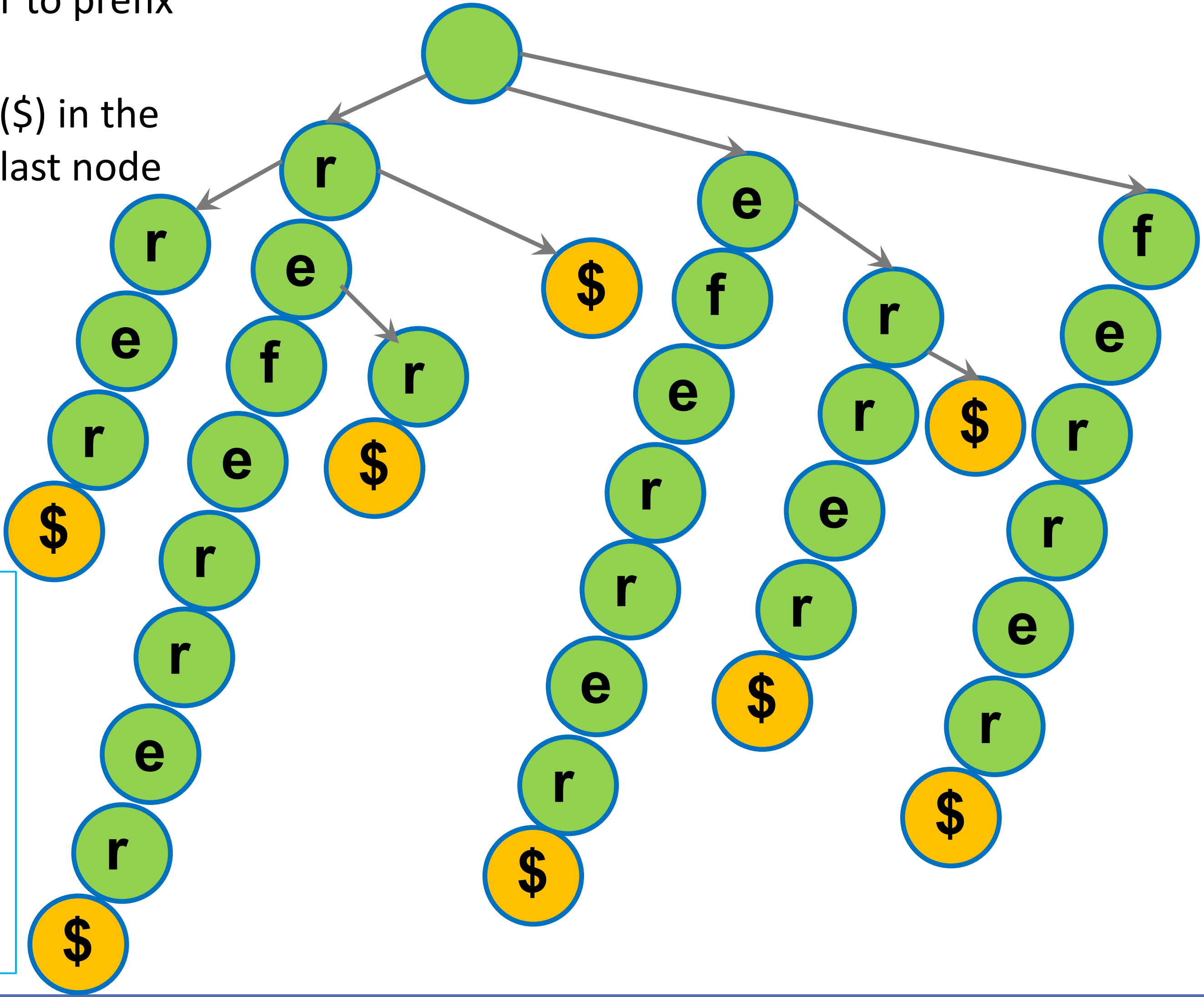
Count “r”

Count “er”

Count “re”

Count “err”

Count “efr”



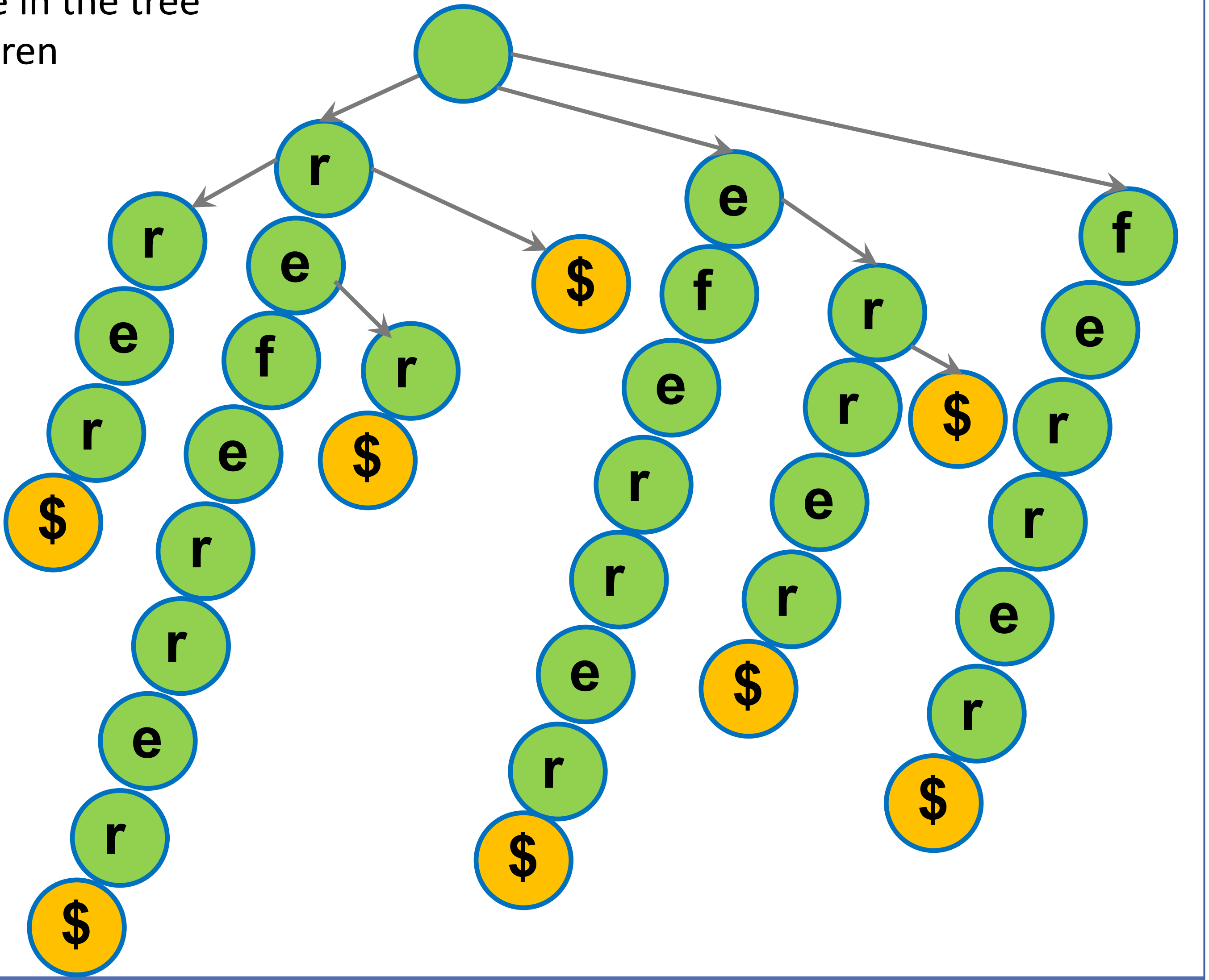
Time Complexity:

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

E.g., “re” and “er”



Space complexity of suffix trie

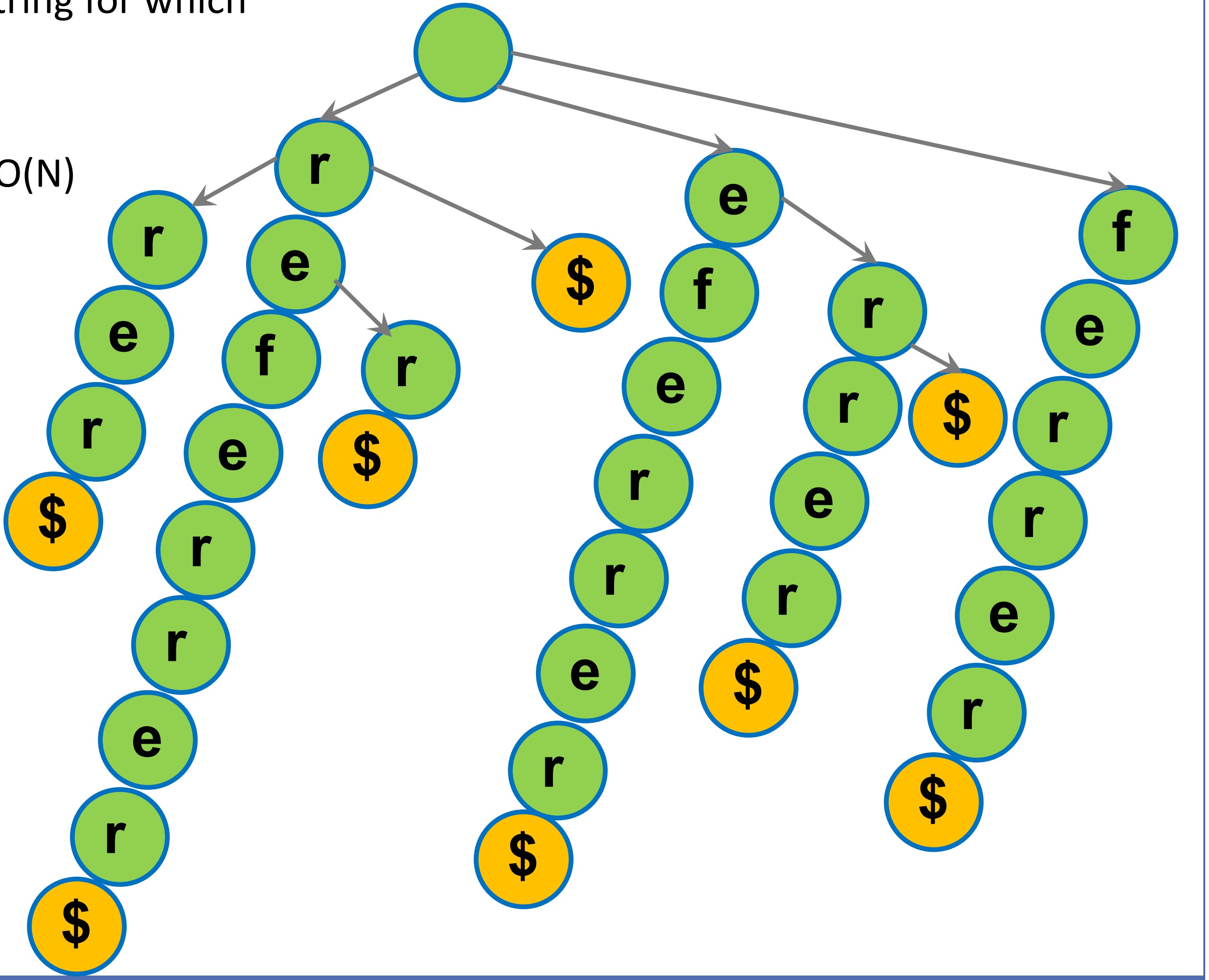
Let N be the size of the string for which suffix trie is constructed

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

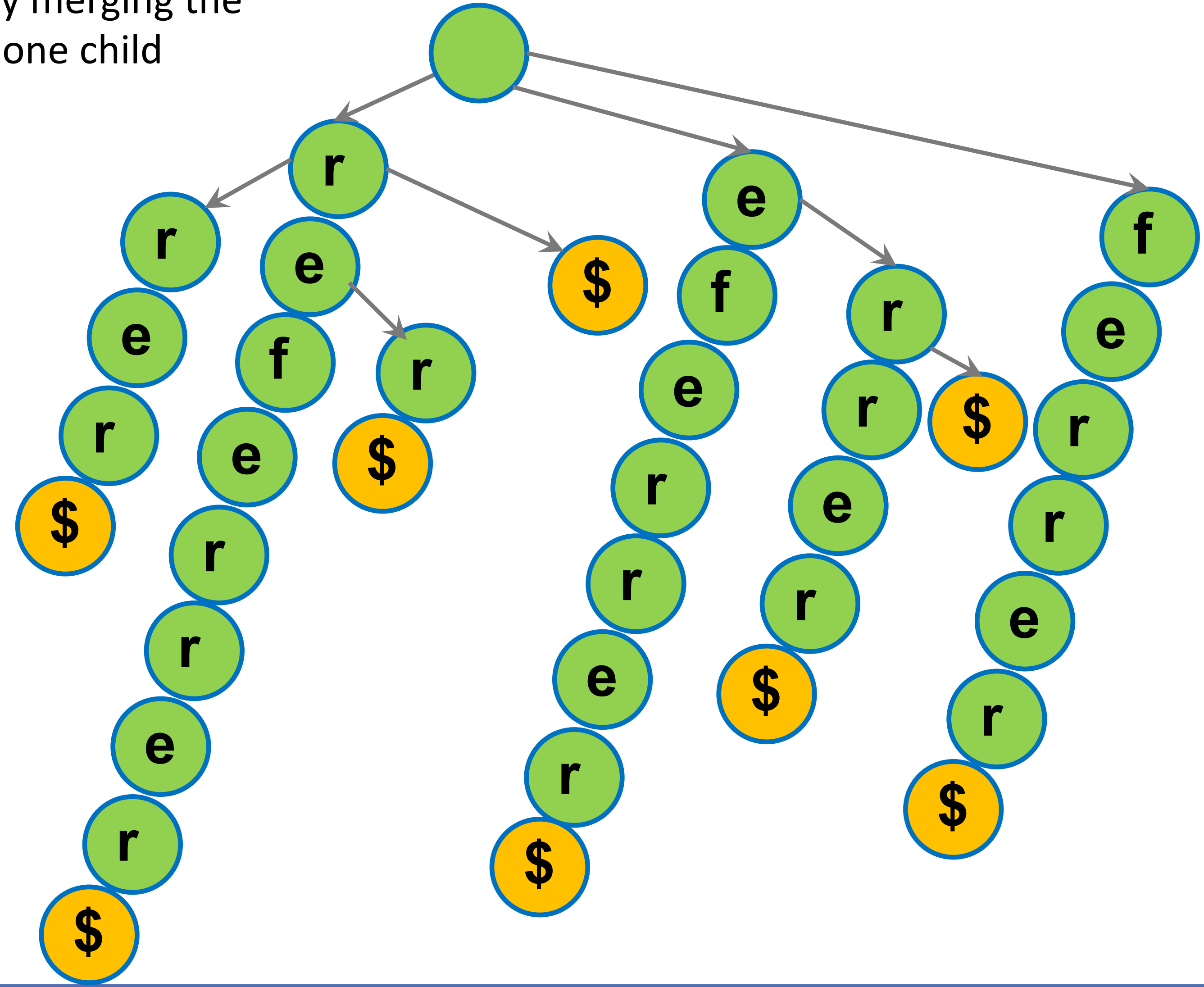
Hi there, I am pretty good at string queries.
Me too AND I occupy way less space. Don't Trie him at home.

Suffix Trie

Suffix Tree

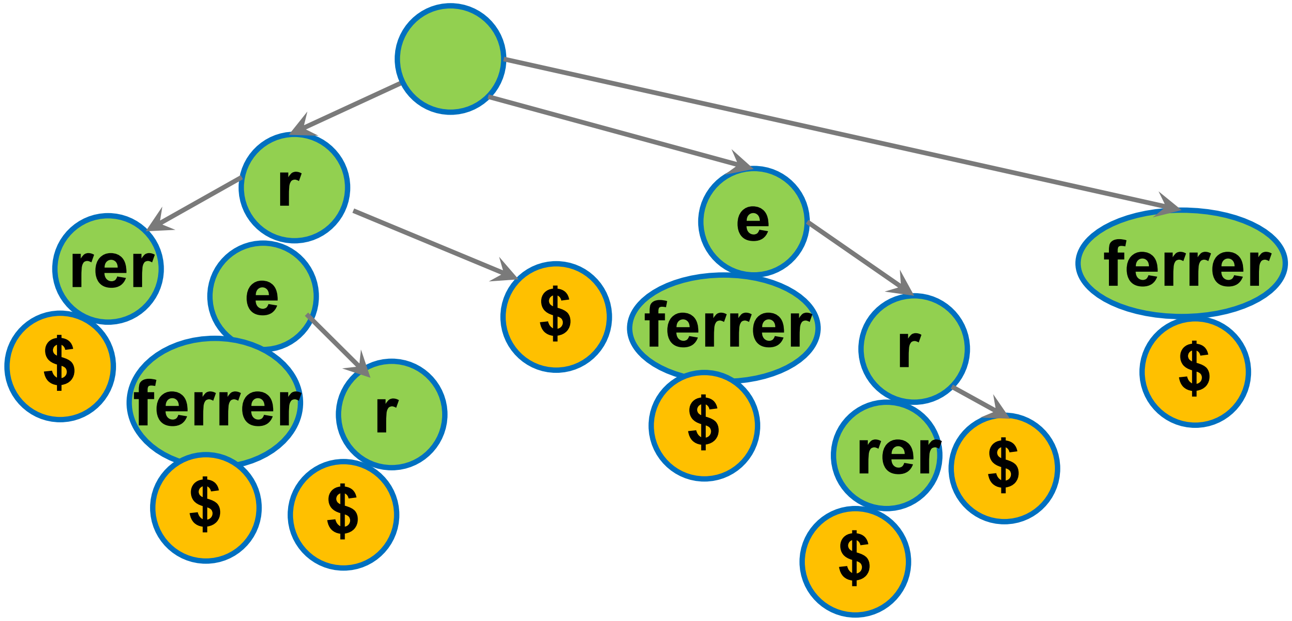
Suffix Tree is a compact Suffix Trie

- 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



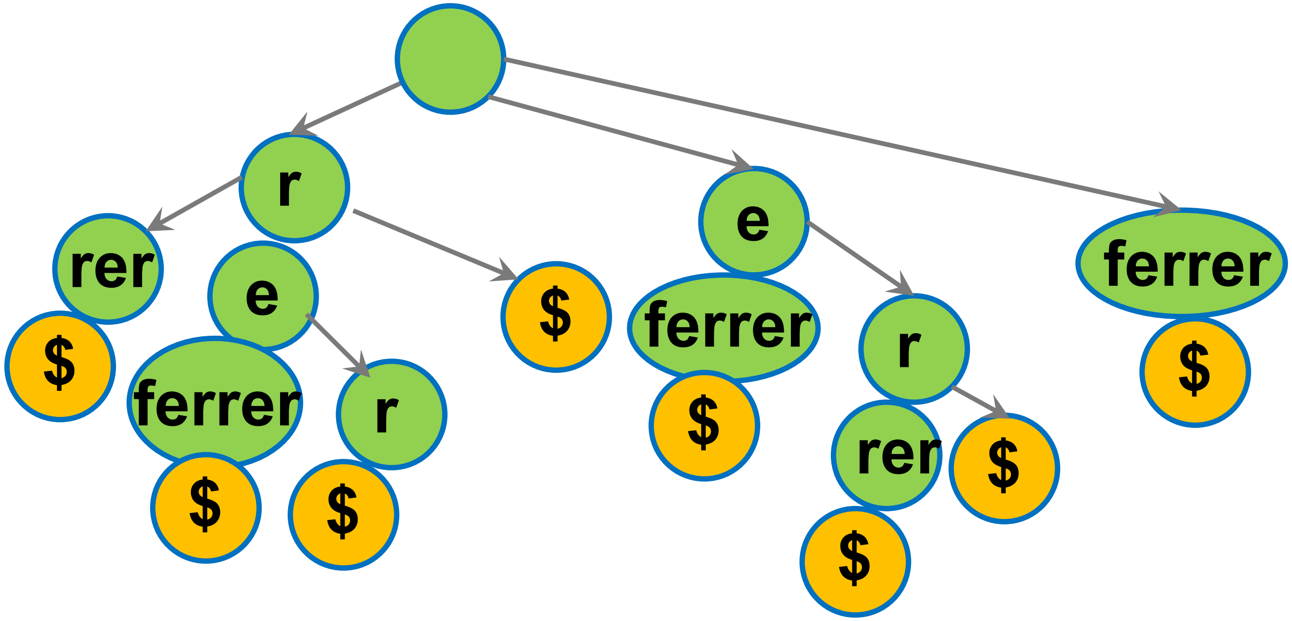
Space complexity of suffix tree

- 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



Space complexity of suffix tree

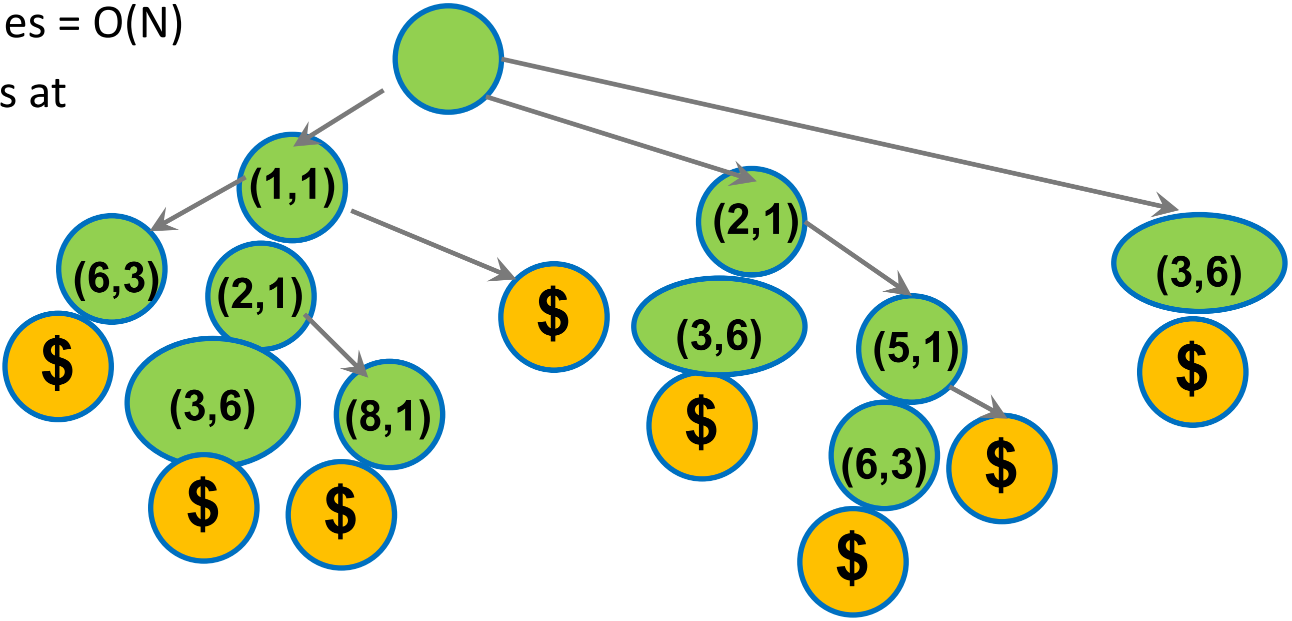
- Compress branches by merging the nodes that have only one child
- But the total complexity is still the same as the same number of letters are stored
- Replace every substring with numbers (x,y) where x is the starting index of the substring and y is its length

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

- Total number of leaf nodes = # of suffixes
- Total number of leaf nodes = O(N)
- Each node in the tree has at least two children
- So, total # of nodes is O(N + N/2 + N/4 + …) = O(N)



Time complexity of constructing suffix tree

- The algorithm described earlier inserts O(N) suffixes
- Insertion cost of each suffix is linear to the size of suffix
- 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 MISSISSIPPI$

2 ISSISSIPPI$

3 SSISSIPPI$

4 SISSIPPI$

5 ISSIPPI$

6 SSIPPI$

7 SIPPI$

8 IPPI$

9 PPI$

10 PI$

11 I$

12 $

Sort

$

I$

IPPI$

ISSIPPI$

ISSISSIPPI$

MISSISSIPPI$

PI$

PPI$

SIPPI$

SISSIPPI$

SSIPPI$

SSISSIPPI$

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)

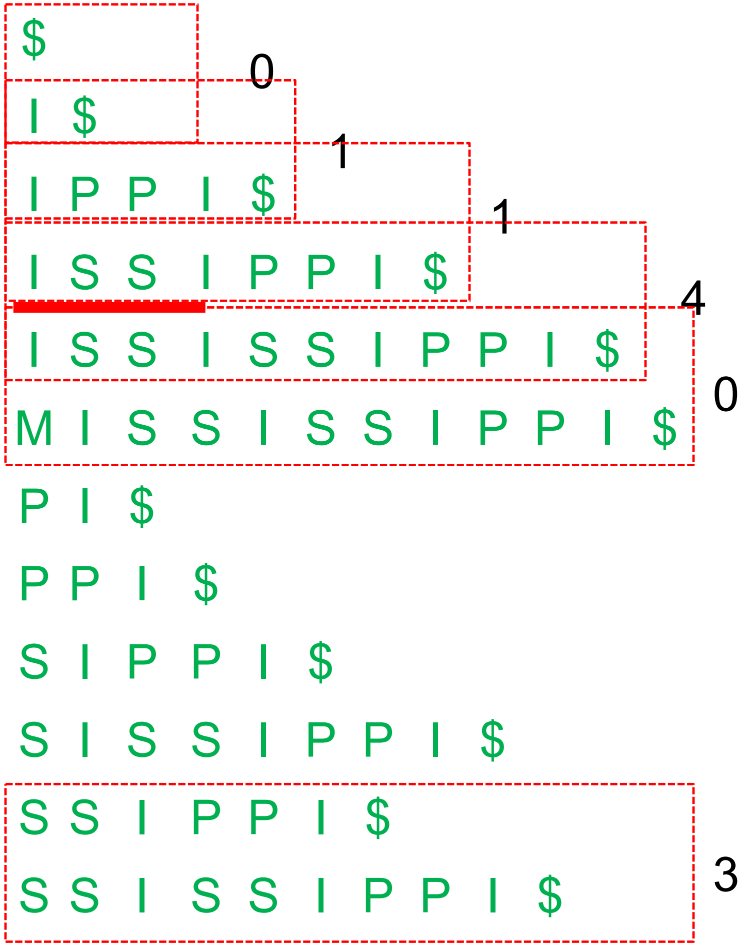
$
I$
IPPI$
ISSIPPI$
ISSISSIPPI$
MISSISSIPPI$
PI$
PPI$
SIPPI$
SISSIPPI$
SSIPPI$
SSISSIPPI$

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
- Cost of computing LCP among two strings of length N characters
  - O(N)
- Total cost of longest repeated substring?
  - O(N^2)



$

0

I $

1

I P P I $

1

I S S I P P I $

4

I S S I S S I P P I $

0

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 $

3

S S I S S I P P I $

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$

IPPI$

ISSIPPI$

ISSISSIPPI$

MISSISSIPPI$

PI$

PPI$

SIPPI$

SISSIPPI$

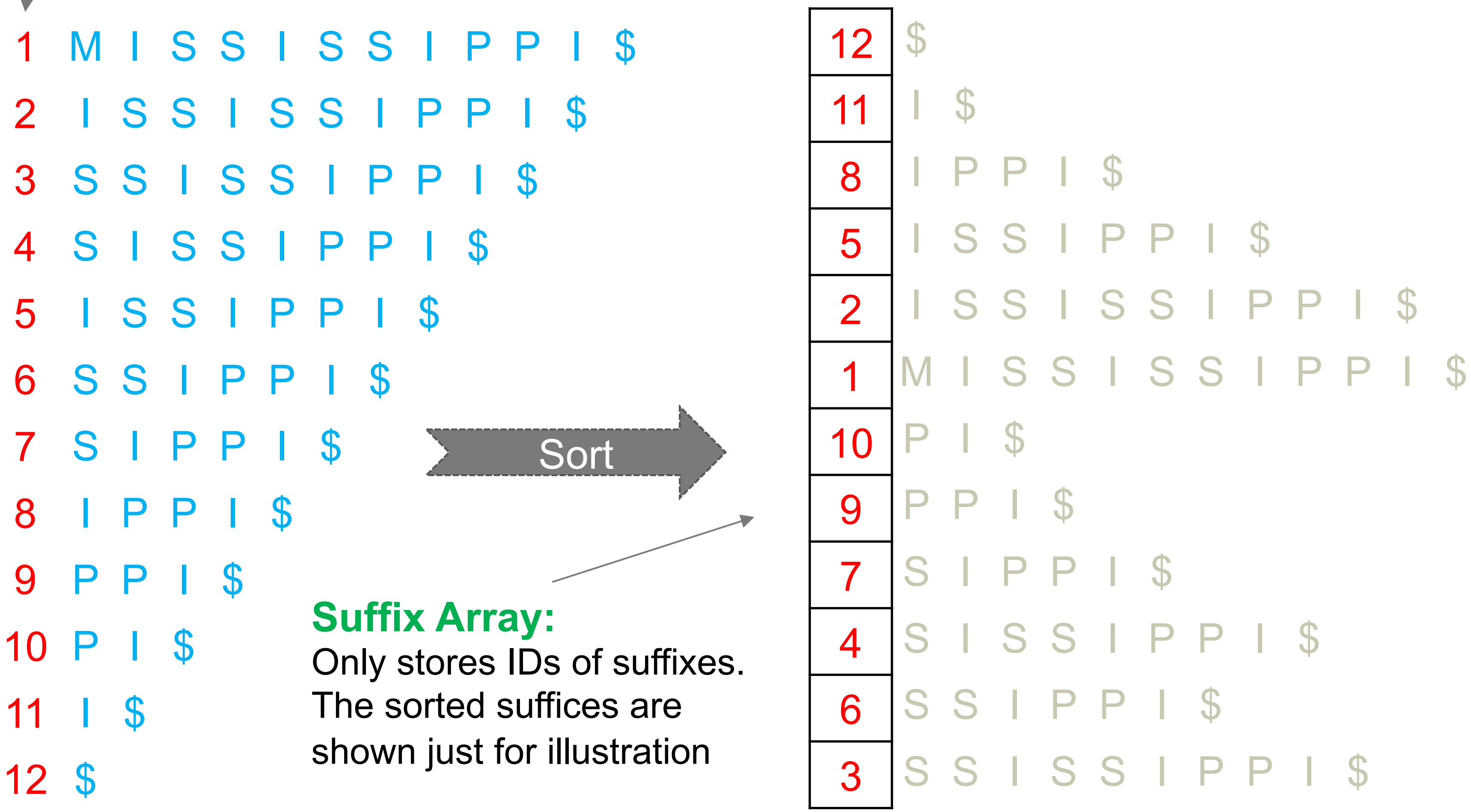
SSIPPI$

SSISSIPPI$

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	MISSISSIPPI$
2	ISSISSIPPI$
3	SSISSIPPI$
4	SISSIPPI$
5	ISSIPPI$
6	SSIPPI$
7	SIPPI$
8	IPPI$
9	PPI$
10	PI$
11	I$
12	$

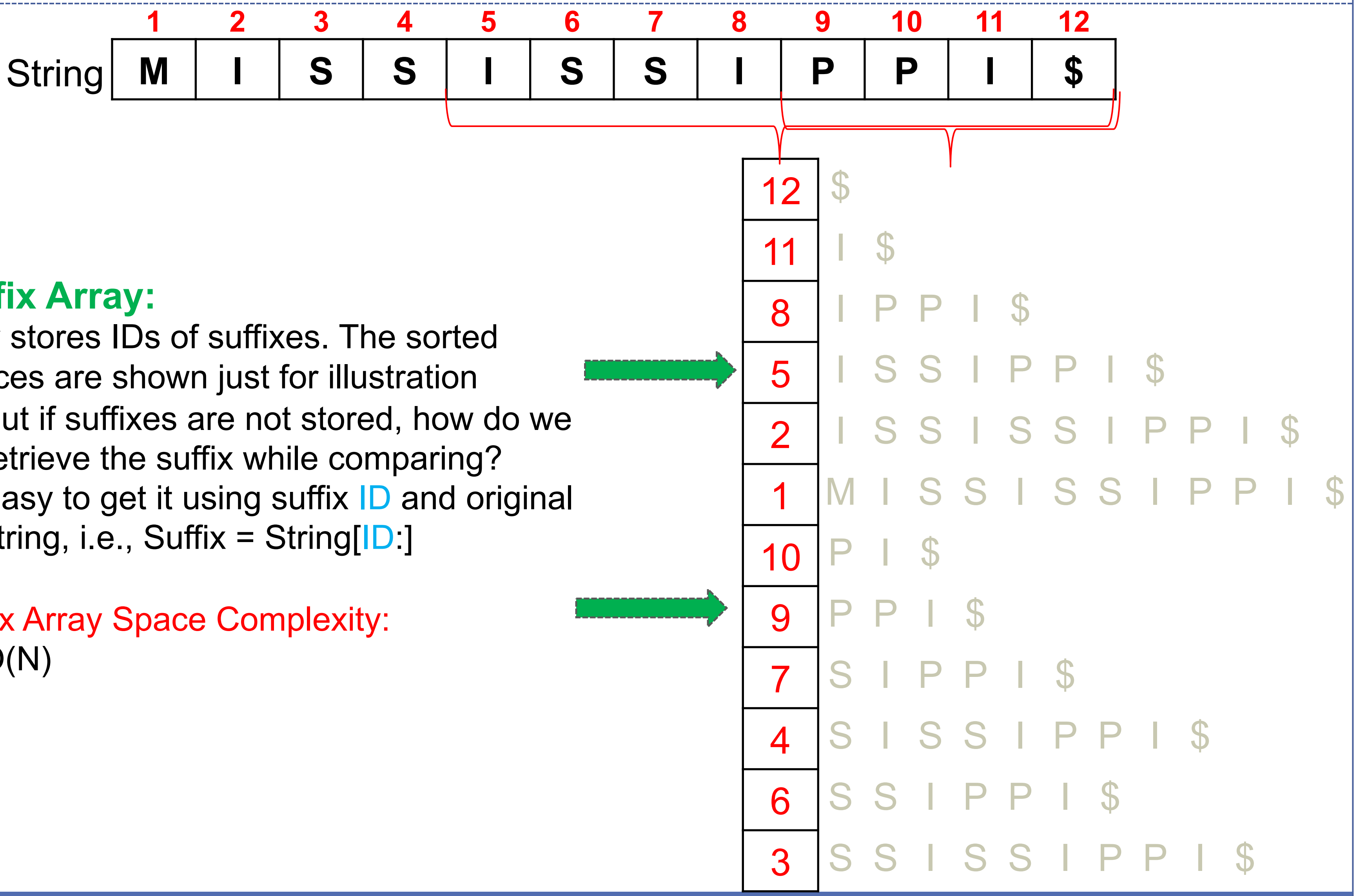
12	$
11	I$
8	IPPI$
5	ISSIPPI$
2	ISSISSIPPI$
1	MISSISSIPPI$
10	PI$
9	PPI$
7	SIPPI$
4	SISSIPPI$
6	SSIPPI$
3	SSISSIPPI$

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

Practice

What will be the suffix array of BIRD$?

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	$

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

- But if suffixes are not stored, how do we retrieve the suffix while comparing?
- Easy to get it using suffix ID and original string, i.e., Suffix = String[ID:]

Suffix Array Space Complexity:

- O(N)

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

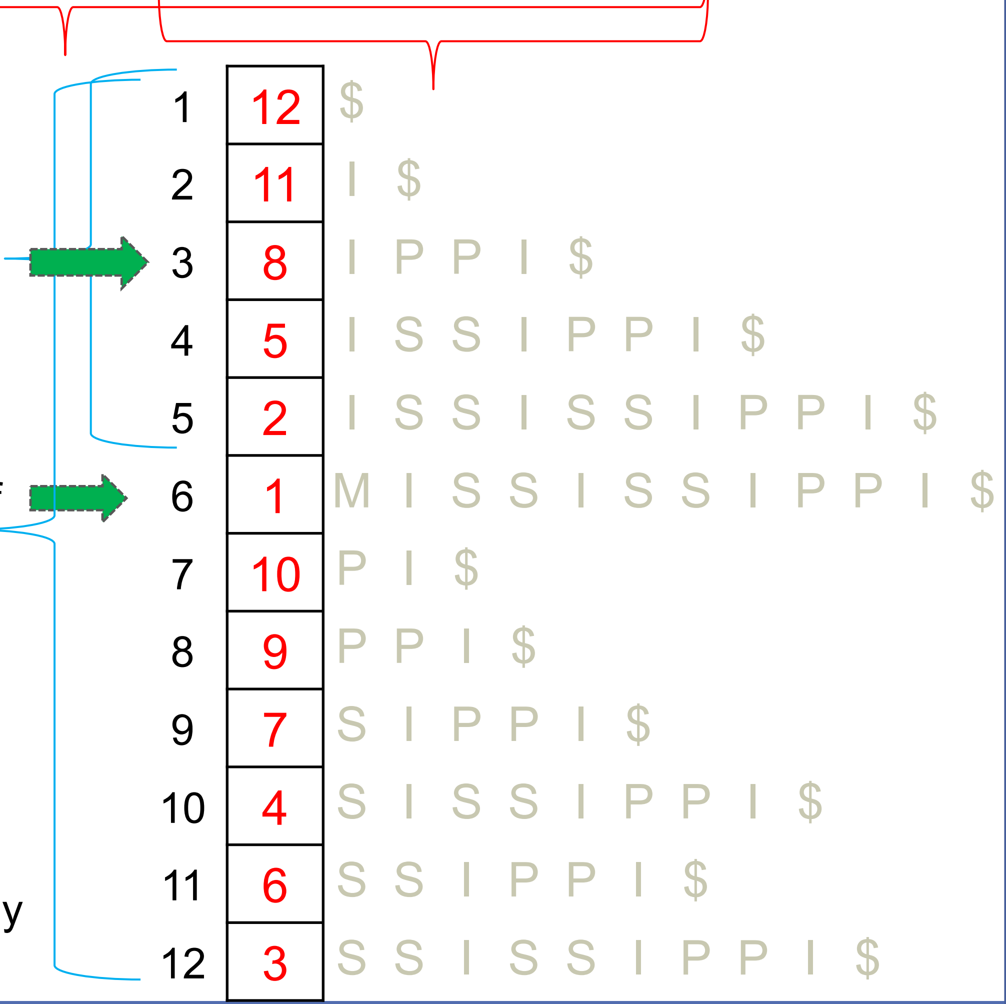
Example:

Search “IPP” in the string.

- Initially, the search space is whole Suffix Array
- 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	IPPI$
4	5	ISSIPPI$
5	2	ISSISSIPPI$
6	1	MISSISSIPPI$
7	10	PI$
8	9	PPI$
9	7	SIPPI$
10	4	SISSIPPI$
11	6	SSIPPI$
12	3	SSISSIPPI$

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

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 their 1st characters
- Sort suffixes on first 2 characters

Rank	ID		
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 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	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 their 1st characters
- Sort suffixes on first 2 characters
- Sort suffixes on first 4 characters
- …

Rank	ID	
1	12	$
2	11	I$
3	8	IPPI$
4	2	ISSISSIPPI$
4	5	ISSIPPI$
6	1	MISSISSIPPI$
7	10	PI$
8	9	PPI$
9	7	SIPPI$
10	4	SISSIPPI$
11	6	SSIPPI$
12	3	SSISSIPPI$

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 characters

Rank	ID	
1	12	$
2	11	I$
3	8	IPPI$
4	5	ISSIPPI$
5	2	ISSISSIPPI$
6	1	MISSISSIPPI$
7	10	PI$
8	9	PPI$
9	7	SIPPI$
10	4	SISSIPPI$
11	6	SSIPPI$
12	3	SSISSIPPI$

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

Time complexity:

- Cost of first sort (1 character)
  - 1.N log N
- Cost of second sort (2 characters)
  - 2.N log N
- Cost of i-th sort (2^{i-1} characters)
  - 2^{i-1} N log N
- Total cost:
- NlogN+2NlogN+4NlogN+…+N.NlogN
- (1 + 2 + 4 +… + N/2 + N)* N log N
  - (N+N/2 +N/4 + … + 1) → O(N)
- Total cost is still O(N^2 log 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 $

Constructing Suffix Array: Prefix Doubling

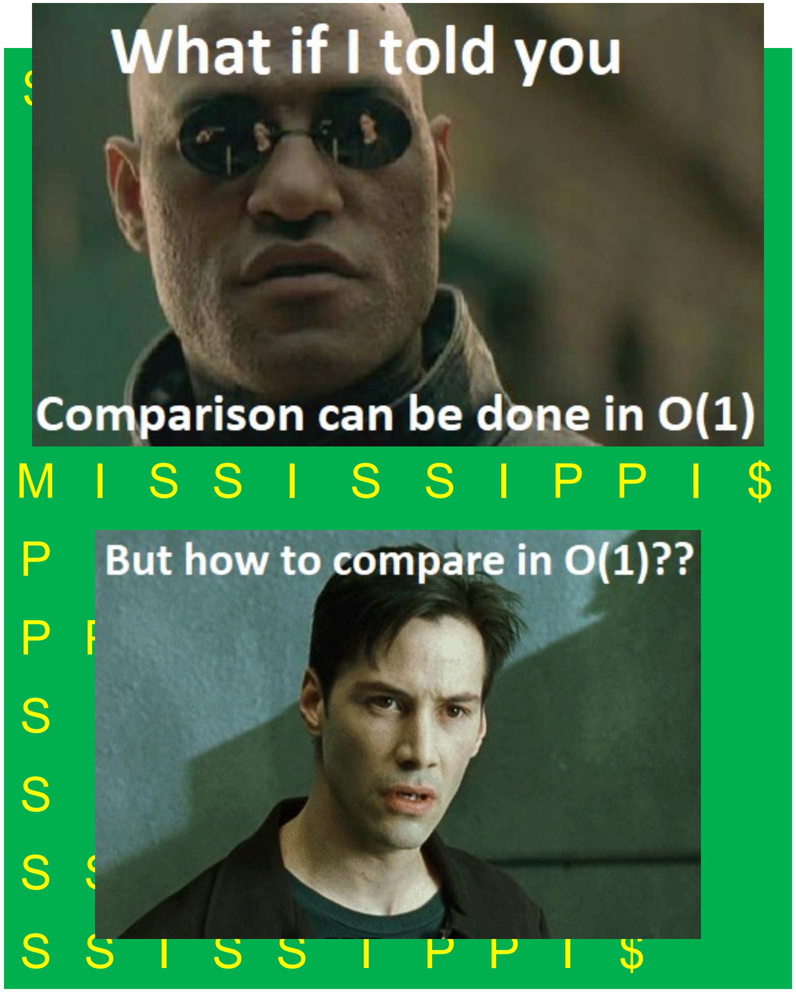
	1	2	3	4	5	6	7	8	9	10	11	12
String	M	I	S	S	I	S	S	I	P	P	I	$

What if we can compare any two suffixes in O(1)?

Time complexity:

- Cost of first sort (1 character)
  - 1.N log N
- Cost of second sort (2 characters)
  - 1.N log N
- Cost of i-th sort (2^i characters)
  - 1.N log N
- Total # of sorting required
  - O(log N)
    - Sort on 1 character
    - Sort on 2 characters
    - ….
    - Sort on N/2 characters
    - Sort on N characters
- Cost for each sort O(N log N)
- Total cost: O(N $\log^2$ N)

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



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 < SSIPPI$
  - 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 $
11	3	S S	I S S I P P I $
11	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	$

Recall: k=2 in this example, and we are sorting on first 2k=4 characters

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 $
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 $

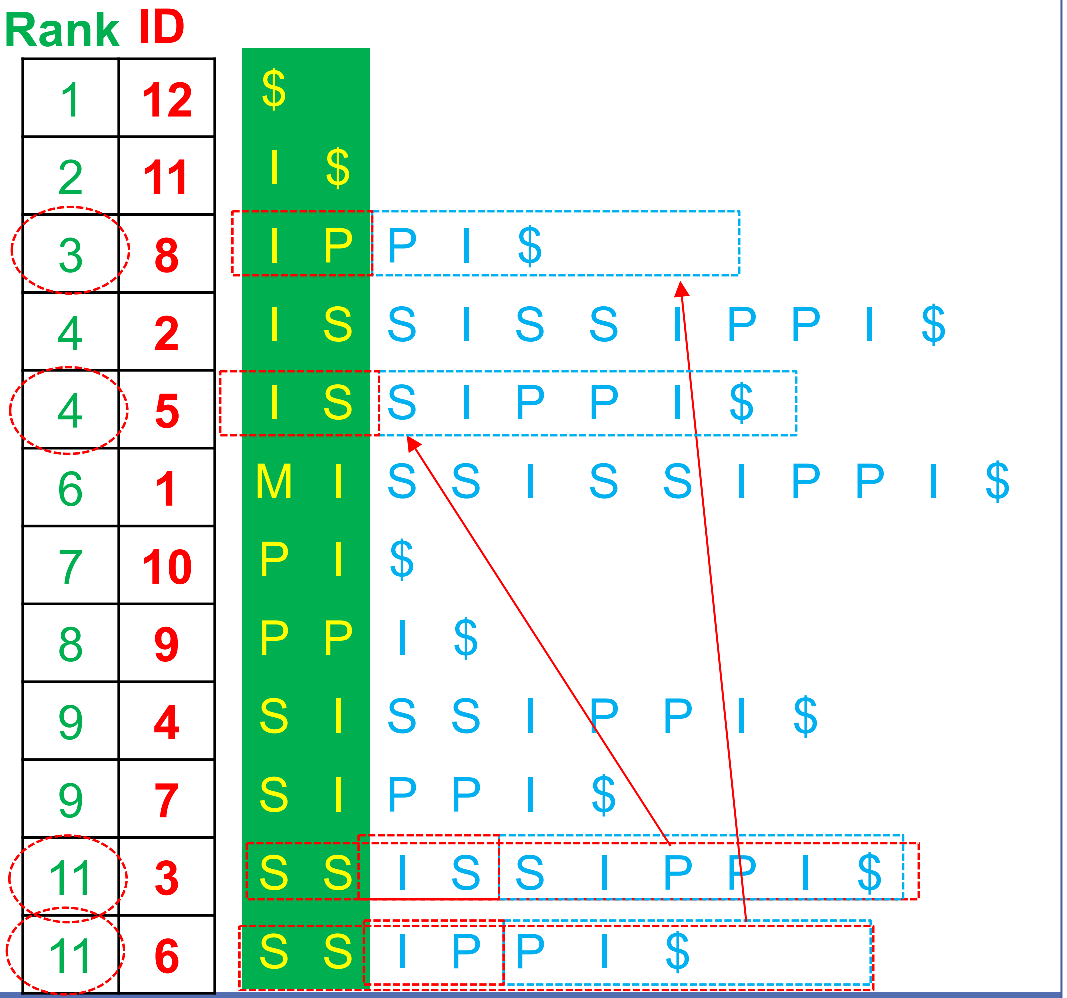
O(1) Comparison

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

Another example:

Suppose we are comparing Suffix with ID 3 and Suffix with ID 6.

- First k characters must be the same
- The tie is to be broken on the next k characters, e.g.,
  - We need to compare "ISSIPPI$" and "IPPI$" on the first 2 characters
  - ISSIPPI$ and IPPI$ are suffixes and are already ranked on first 2 characters
    - E.g., IPPI$ < ISSIPPI$ because its rank is smaller
    - Therefore, SSIPPI$ < SSISSIPPI$



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 $
11	3	S S I S S I P P I $
11	6	S S I P P I $

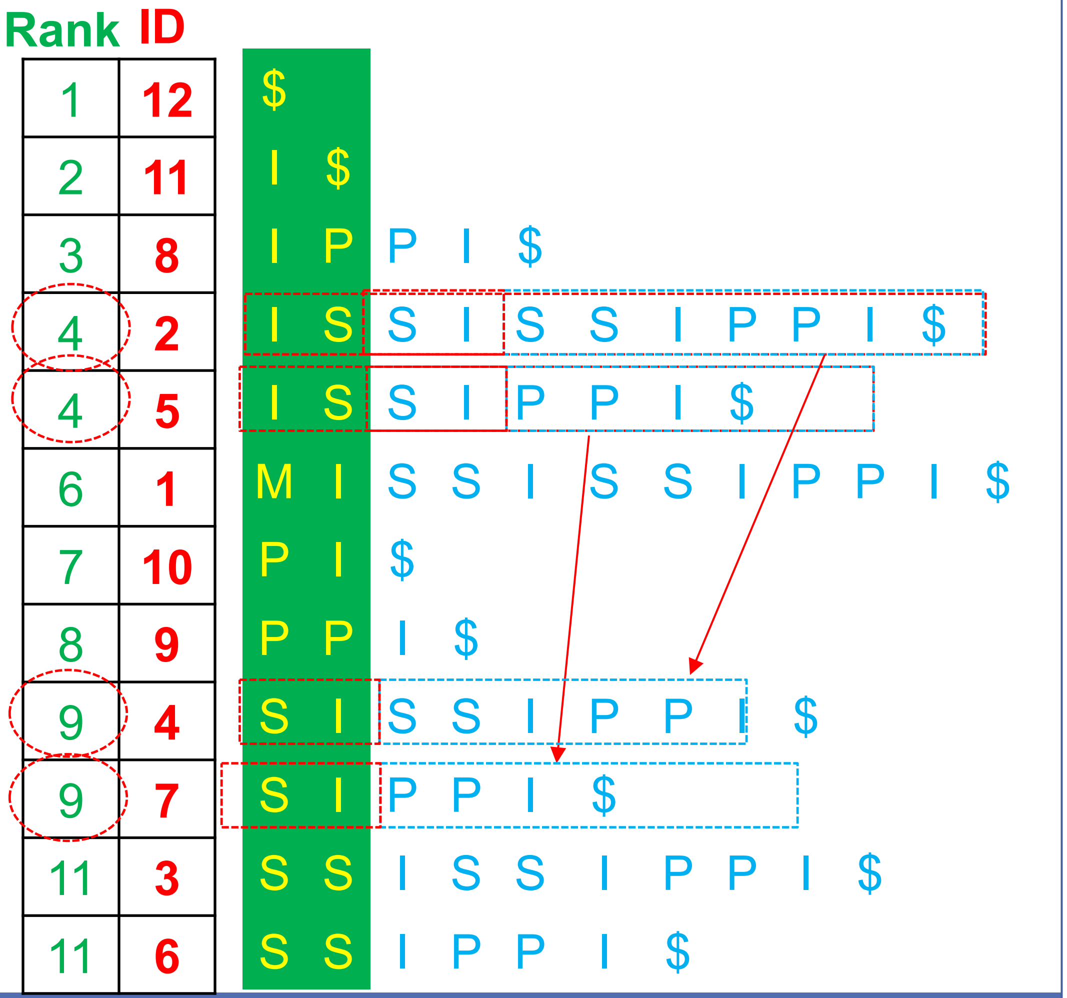
O(1) Comparison

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

Yet another example:

Suppose we are comparing Suffix with ID 2 and Suffix with ID 5.

- First k characters must be the same
- The tie is to be broken on the next k characters, e.g.,
  - We need to compare "SISSIPPI$" and "SIPPI$" on the first 2 characters
  - SISSIPPI$ and SIPPI$ are suffixes and are already ranked on first 2 characters
    - E.g., SIPPI$ = SISSIPPI$ on **first 2** characters
    - Therefore, ISSIPPI$ = ISSISSIPPI$ 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	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 $

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	$

Suffixes are sorted on first 4 characters and we are comparing them on first 8 characters.

Suppose we are comparing suffix with ID 2 and 5:

- Are they ranked the same at first 4 characters?
- Let's compare them on next 4 characters.
  - What are the suffixes (give their IDs) whose rank we need to compare?

How do we efficiently determine suffix IDs and their ranks?

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	11	9	4	11	9	3	8	7	2	1
String	M	I	S	S	I	S	S	I	P	P	I	$

Create a new array called **Rank** where

- Index corresponds to ID of each suffix
- At each index, the current rank of the suffix is recorded

This array can be used to get current rank of any suffix.

Consider Suffix with ID 2 "ISSISSIPPI$":

- Its current rank can be found at **Rank[2]**
- How do we know the ID/rank of suffix "SISSIPPI$"?
- Since it is 2 characters "off" from "ISSISSPPI$" (ID 2), its ID is 2+2=4 and rank is **Rank[4]** = 9
- In general, a suffix k characters "off" from a suffix with ID **x** will have an ID x + k.
    - E.g., ID of suffix "IPPI$"?
    - It is 6 characters "off" from suffix with ID 2 so its ID is 2 + 6 = 8

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 $
11	3	S S I S S I P P I $
11	6	S S I P P I $

Given SSISSIPPI$ with ID 3, what is the ID of PPI$?

O(1) Comparison

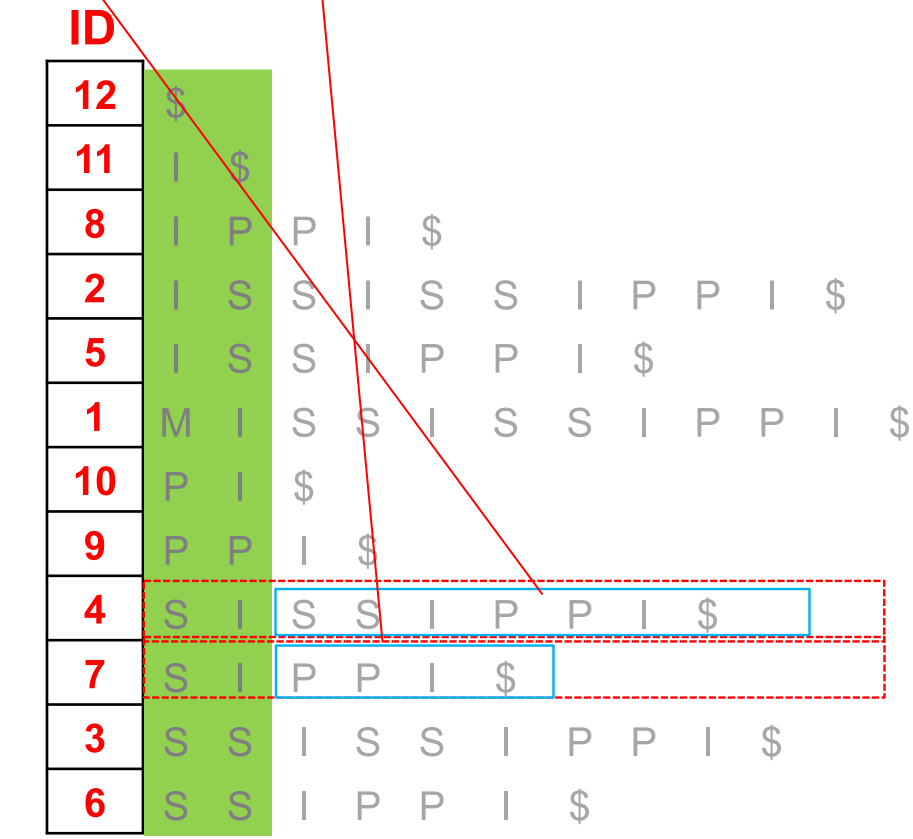
Index/ID	1	2	3	4	5	6	7	8	9	10	11	12
Rank	6	4	11	9	4	11	9	3	8	7	2	1
String	M	I	S	S	I	S	S	I	P	P	I	$

Note: We don't need to store the suffixes (shown grey) – we only need Suffix IDs.

Comparing Suffix with IDs 4 and 7:

- Their ranks are equal.
  - **Rank[4]** = **Rank [7]**
  - E.g., they are the same on first 2 characters
- We need to compare them on the next 2 characters
- Compare ranks of suffixes 4+2=6 and 7+2=9
  - **Rank[6]** > **Rank[9]**
  - So, Suffix #7 is smaller than #4

Note: Comparison takes O(1) and we do not need to store all suffixes – space used is O(N)



ID	Suffix
12	$
11	I $
8	I P P I $
2	I S S I S S I P P I $
5	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 $
4	S I S S I P P I $
7	S I P P I $
3	S S I S S I P P I $
6	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	$

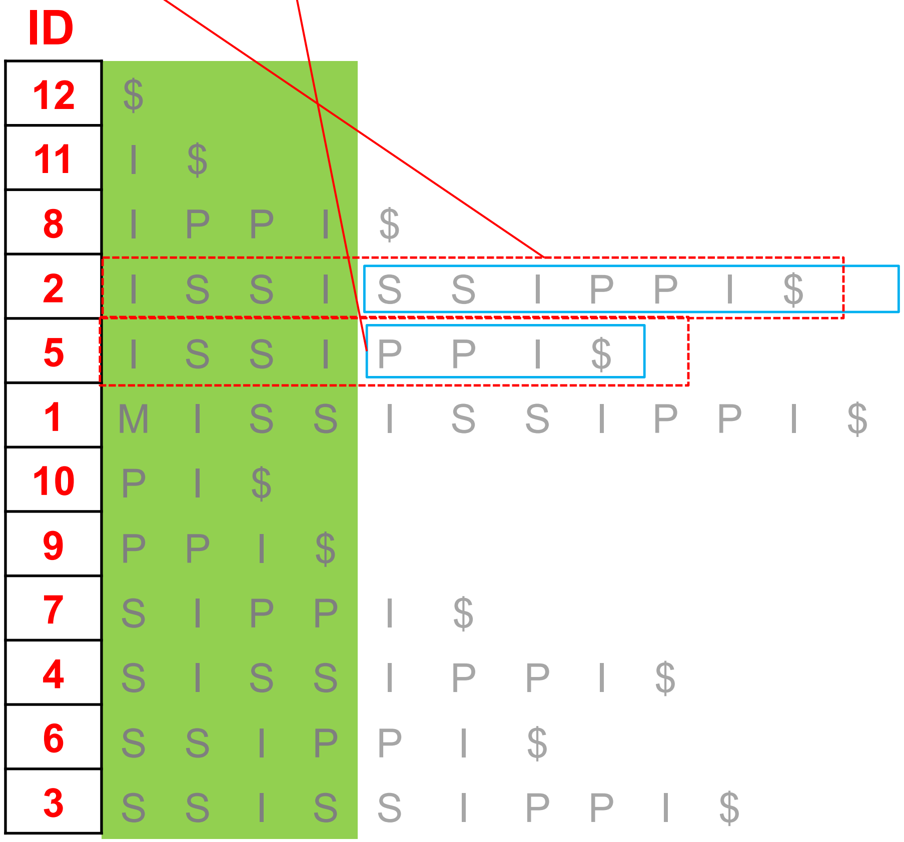
Note: We don't need to store the suffixes (shown grey) – we only need Suffix IDs.

Suppose array has been sorted on first 4 characters.

Comparing Suffix with IDs 2 and 5:

- Their ranks are equal.
  - **Rank[2] = Rank [5]**
  - E.g., they are the same on first 4 characters
- We need to compare them on the next 4 characters
  - Should we instead compare them on all remaining characters???
    - No, array is sorted on first 4 only
- Compare ranks of suffixes 2+4=6 and 5+4=9
  - **Rank[6] > Rank[9]**
  - So, Suffix #5 is smaller than #2

Note: Each comparison takes O(1) and we do not need to store all suffixes – space used is O(N)



ID	
12	$
11	I $
8	I P P I $
2	I S S I S S I P P I $
5	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 $

Construction Cost of Suffix Array

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

Time Complexity (prefix doubling):

- We need to sort O(log N) times
    - Sort on 1 characters
    - Sort on 2 characters
    - …
    - Sort on N/2 characters
    - Sort on N characters
- Each sorting requires O(N log N) comparisons
- Each comparison takes O(1)
- Total cost: O(N log^2 N)

Space Complexity:

- O(N)

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 $

Note

- Suffix trie/tree/array can be constructed for long text (e.g., paragraphs) assuming it to be a single string.

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