

Introduction to String Data Structures

Why Strings?

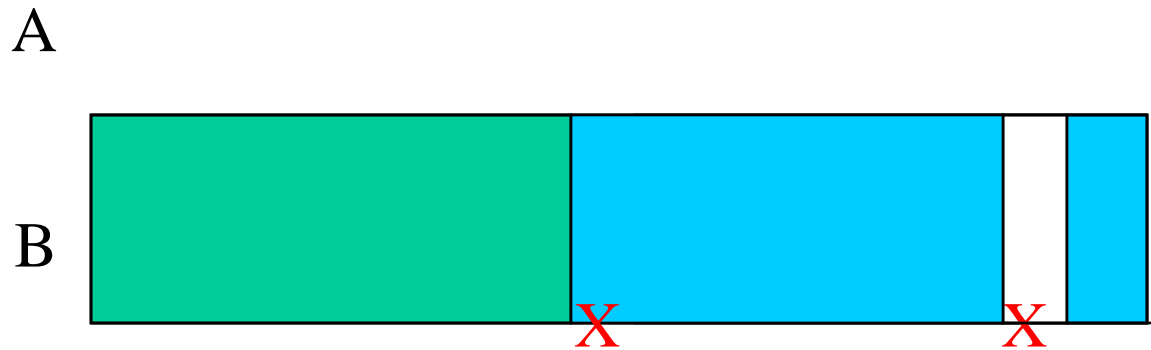
- Biological sequences can be viewed as strings, or finite sequence of characters, over an alphabet Σ .
- There is a wealth of algorithmic theory developed for general strings that we can apply to specific biological problems.

Limitations of String Data Structures

- Can only extract information in the absence of errors.
- Problems dealing with errors may be solved by decomposing into components that do not involve errors.

Example: If two sequences exhibit similarity, there must be substrings in common to them.

Example w/ Errors



Mismatch in the middle results in two exact substring matches of size $n/2$ (green)

Moving the mismatch to right or left still yields at least one match of size $n/2$

Look-up Tables

- Strings of size k over Σ can be represented by an integer index i , $0 \leq i \leq \Sigma^k - 1$.
- DNA is composed of four characters.
 - $\Sigma = \{A, G, C, T\}$ $|\Sigma| = 4$
- We can preprocess a database into a lookup table to locate all occurrences of length k strings.

Example

Let:

A = 0 (00) C = 1 (01) G = 2 (10) T = 3 (11)

Strings are converted based on the binary string they represent

String Binary Integer

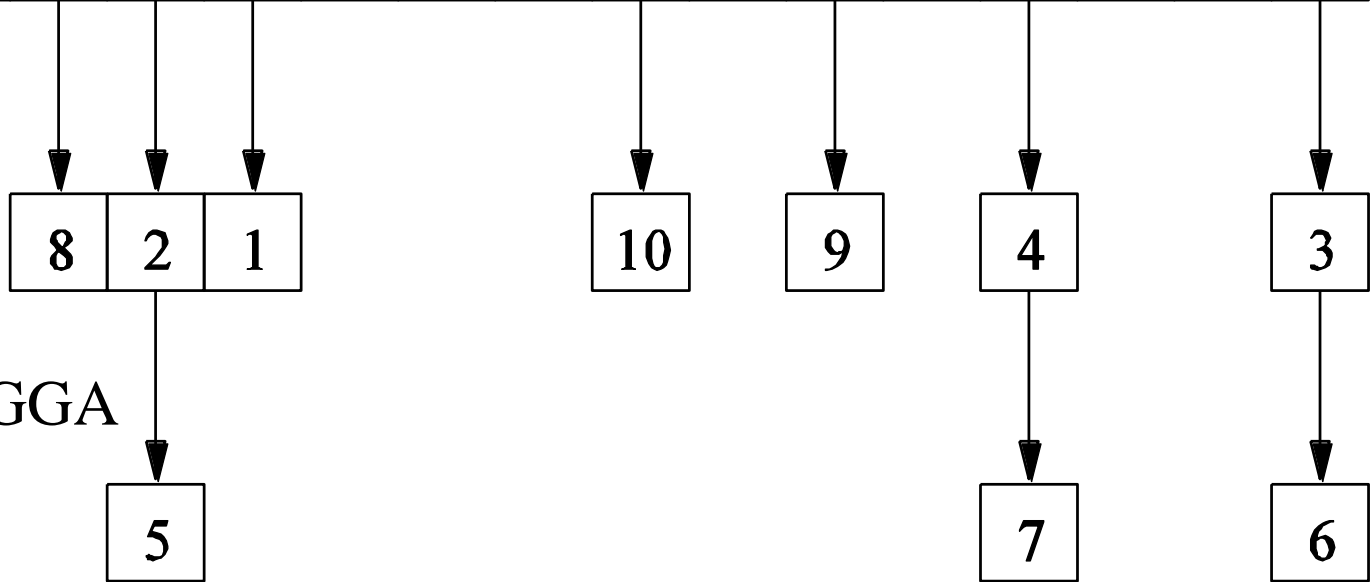
AAA = 000000 = 0

ATA = 001100 = 12

AAC = 000001 = 1

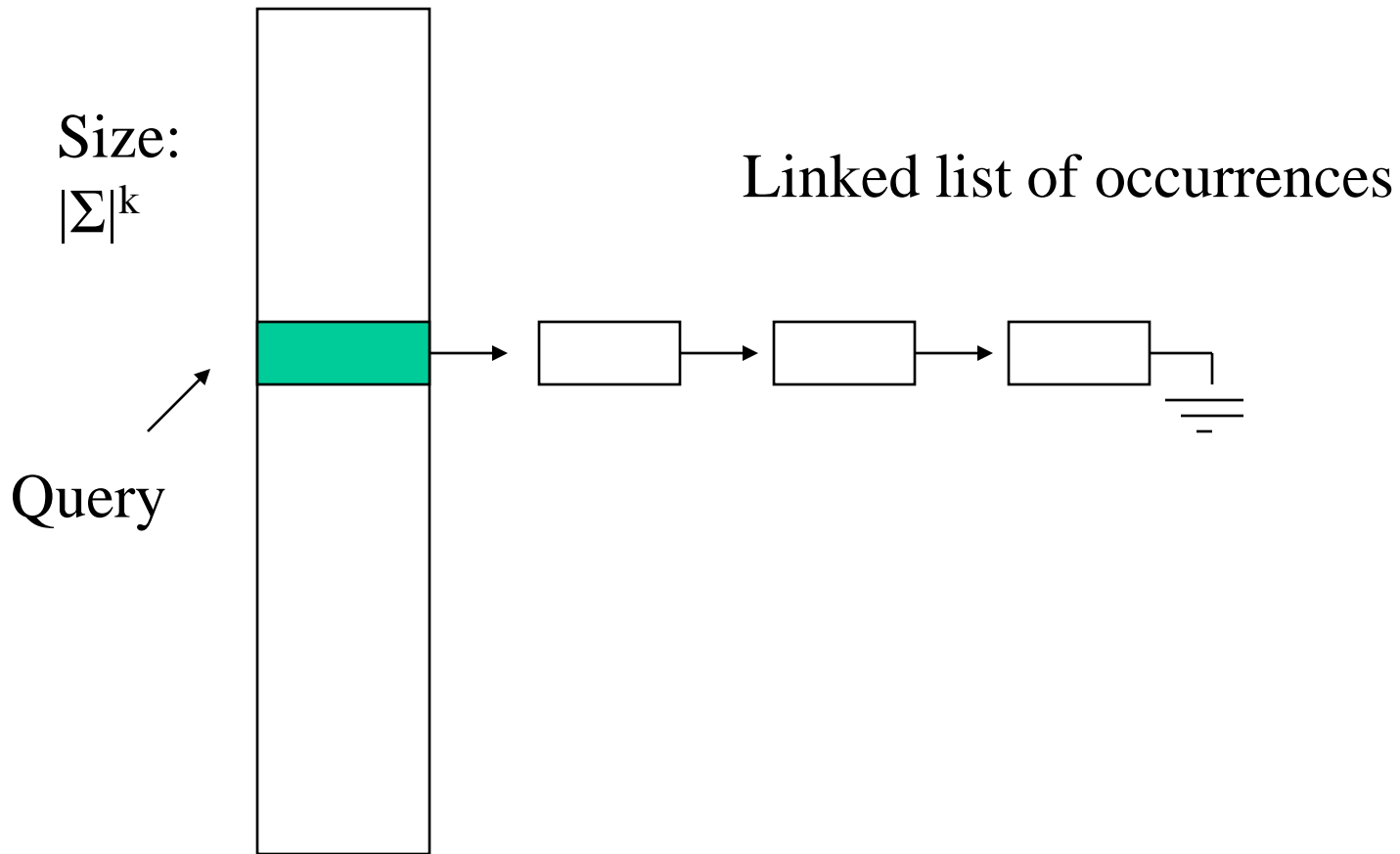
Lookup Table

AA AC AG AT CA CC CG CT GA GC GG GT TA TC TG TT
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



CATTATTAGGA

Search using an Index



Indexing discussion

- A database can be preprocessed in linear time to allow locating all instances of a short string.
- Major limitation is it limits searching to fixed length strings.

Applications of Indexing

- Seeds for searching sequence databases
 - BLAST
- Pair generation for fragment assembly
 - CAP3 sequence assembly program

Basic Terminology

Given string s of length n

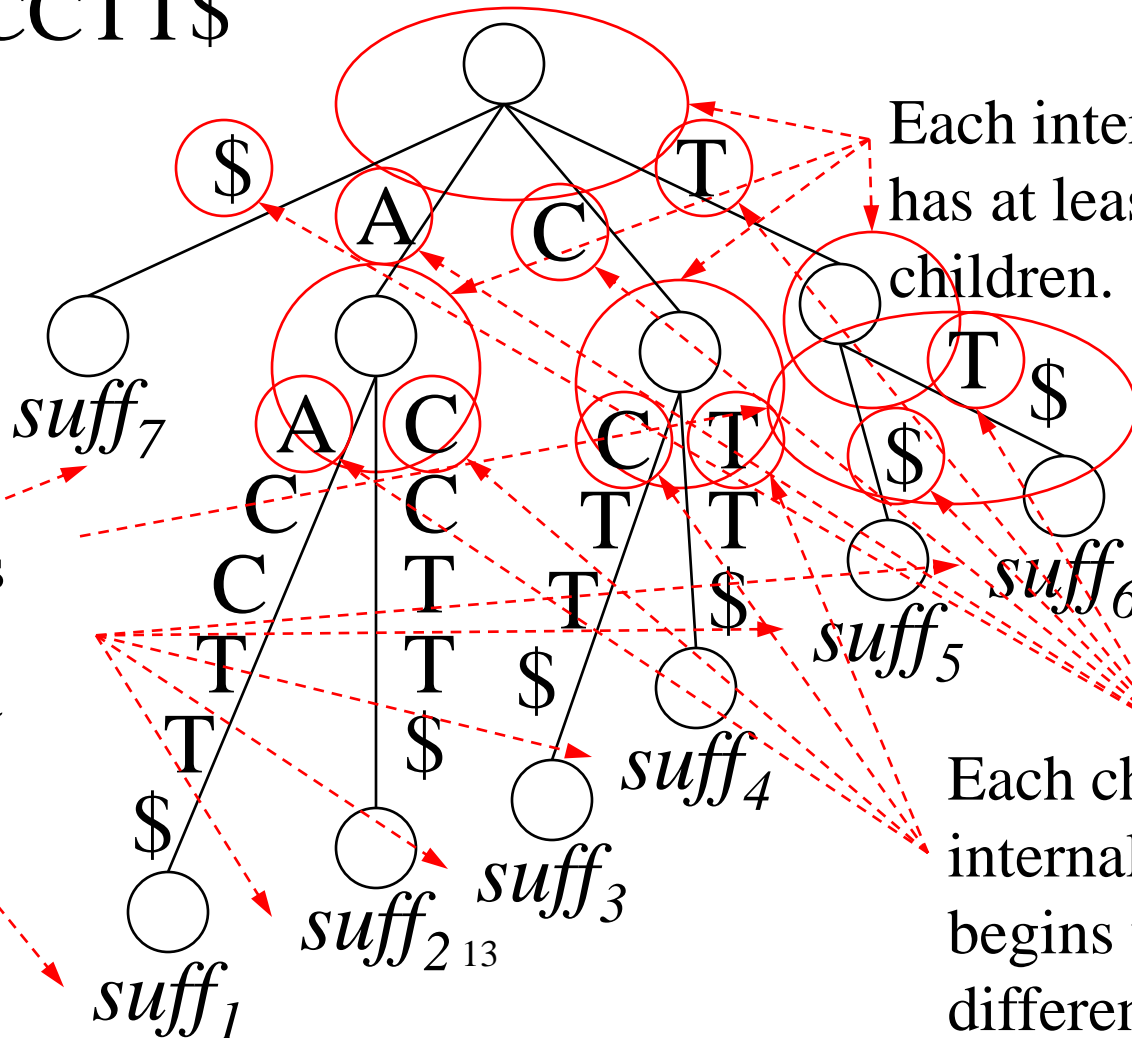
- $s[i] = i^{\text{th}}$ character of s .
- $s[i..j] = s[i]s[i+1] \dots s[j]$
- suffix: $s[i..n] = \text{suff}_i$

Exact Pattern Matching

- Is pattern p a substring of string s ?
- $p = s[i..i+|p|-1] \Leftrightarrow p$ is a prefix of $suff_i$.
- Find suffixes with prefix p .
- Can be answered efficiently if all suffixes are ordered lexicographically.
- Suffix trees and suffix arrays are based on this concept.

Suffix Tree Overview

$s = AACCTT\$$



Each internal nodes has at least 2 children.

Each child of an internal node begins with a different

The unique '\$' character prevents "Each leaf end in the middle of TT" corresponds to a suffix

Suffix tree properties

- For a string of size n , there are n leaves and at most n internal nodes.
 - therefore requires only linear space
- Each leaf represents a unique suffix.
- Concatenation of edge labels from root to a leaf spells out the suffix.
- Each internal node represents a distinct common prefix to at least two suffixes.

More Notation

- $path\text{-}label(v)$ = Concatenation of edge labels from root to v .
- $string\text{-}depth(v) = | path\text{-}label(v) |$
- $tree\text{-}depth(v)$ = Number of nodes on the path from root to v .

More Notation and Properties

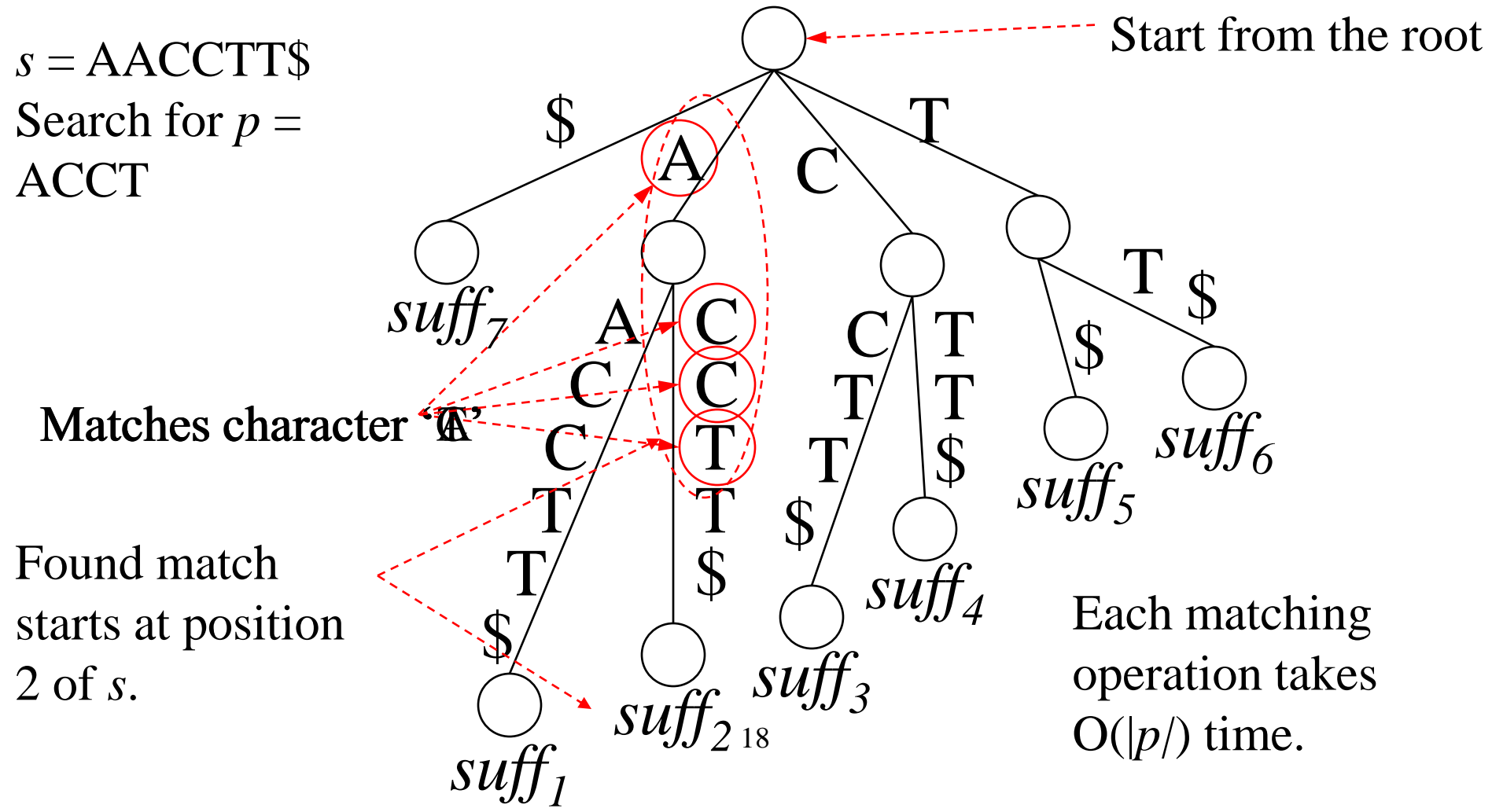
- A substring is called a repeat if it occurs two or more times.
- A repeat is *right maximal* if at least two occurrences are followed by different characters.
- There exists v with $path\text{-}label(v) = \alpha$
 $\Leftrightarrow \alpha$ is a right maximal repeat.

Finding a Pattern in a String

1. Build a suffix tree of the string.
2. Starting from the root, traverse a path matching characters of the pattern.
3. If stuck, pattern not present in string.

Otherwise, each leaf below gives a position of the pattern in the string.

Suffix Tree Pattern Matching



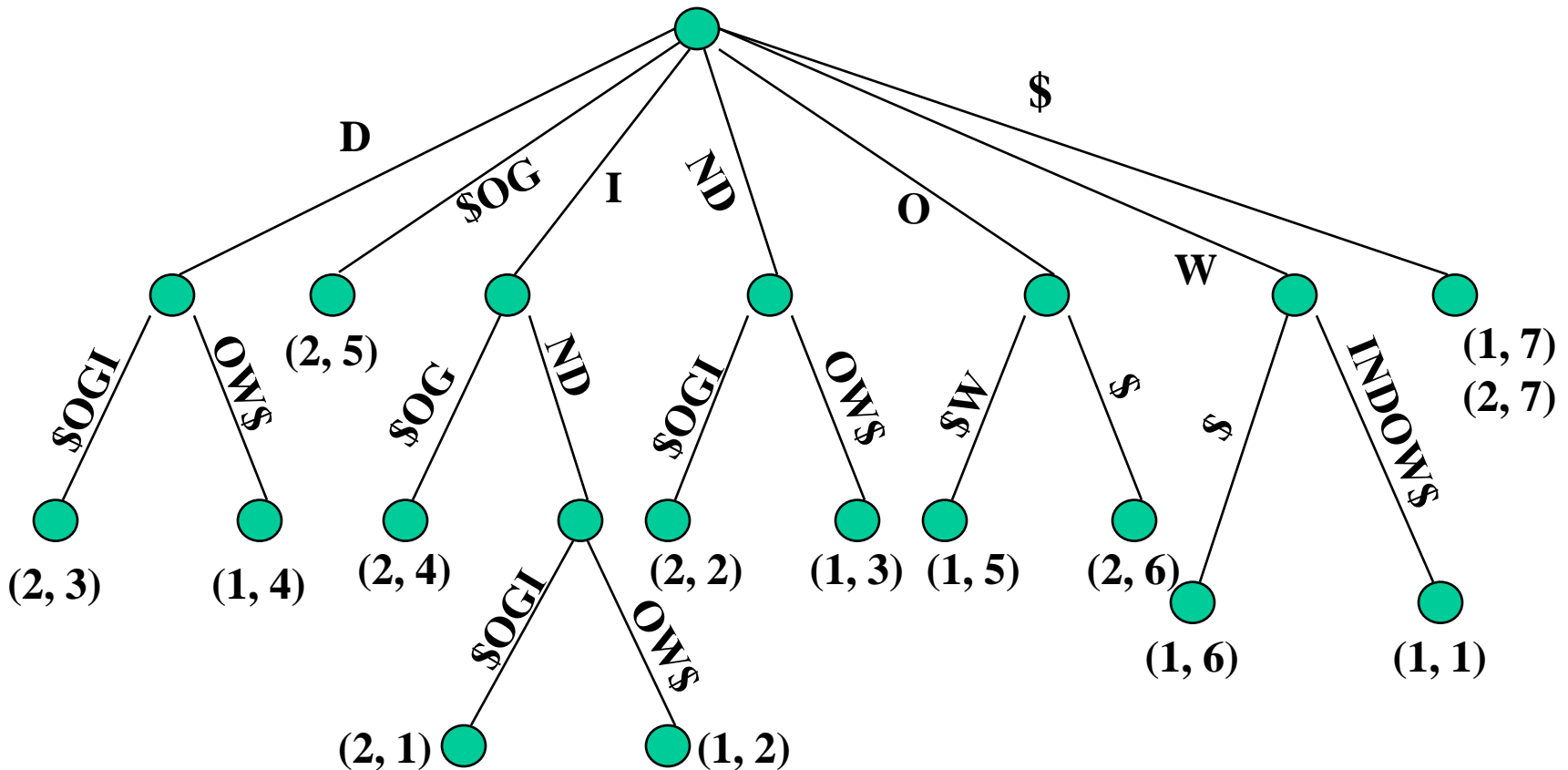
Finding Common Substrings

- Construct a generalized suffix tree for two strings (each suffix of each string is represented).
- Label each leaf with the suffix number and string label.
- Each internal node with a leaf from each string in its subtree gives a common substring.

Generalized Suffix Tree

WINDOW\$
1234567

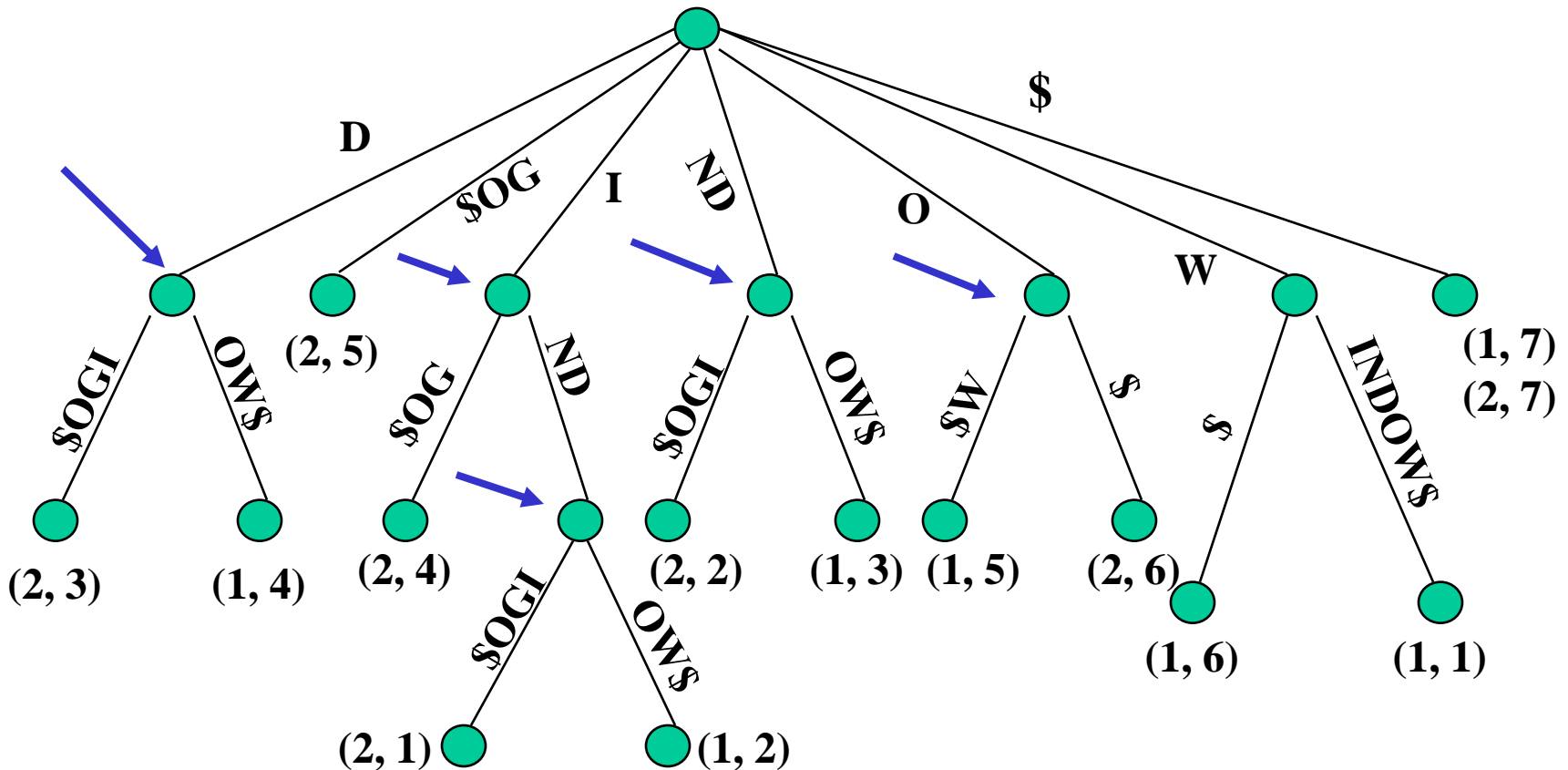
INDIGO\$
1234567



Generalized Suffix Tree

WINDOW\$
1234567

INDIGO\$
1234567



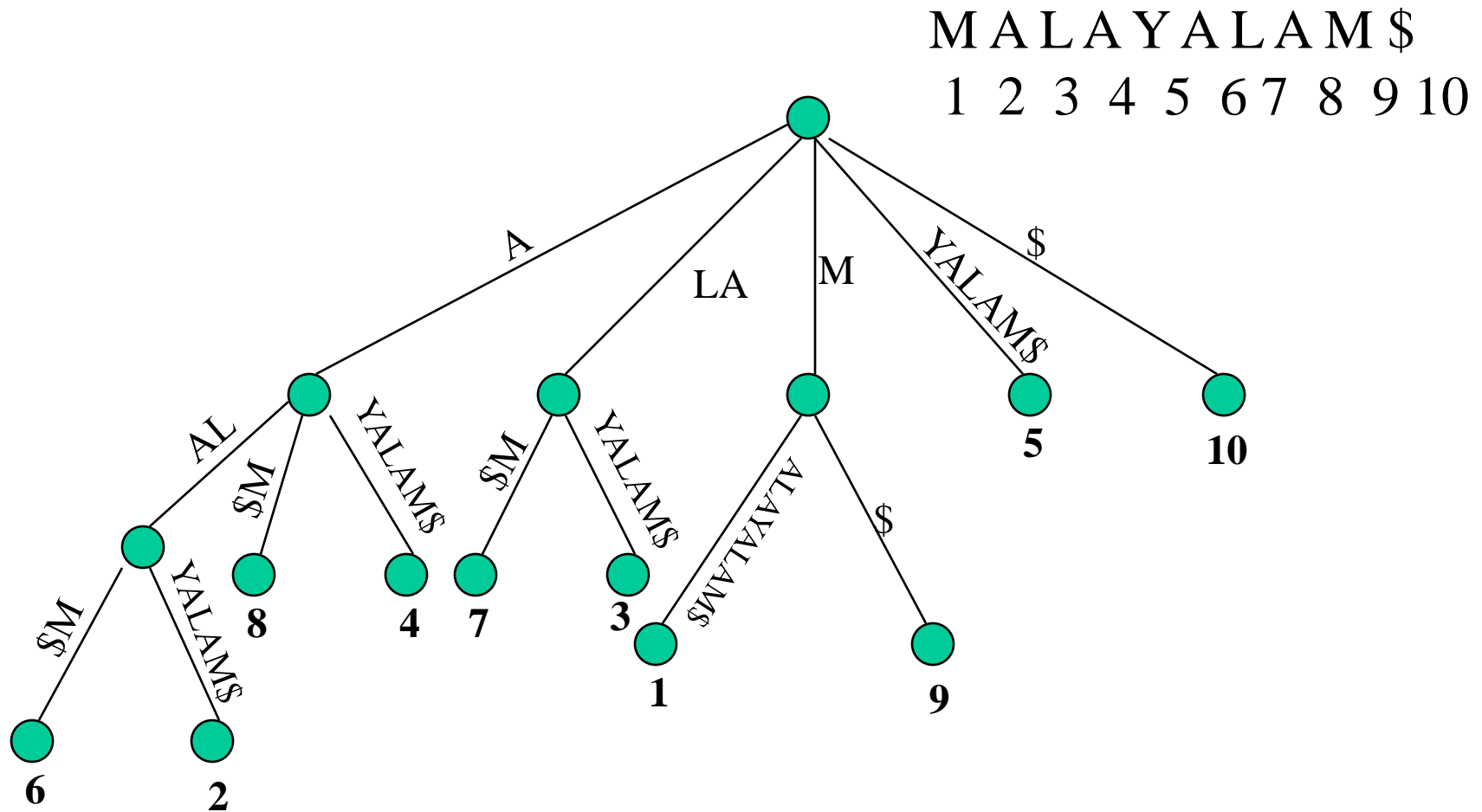
Naïve Construction algorithm

- Start with only the root node,
- Insert each suffix of s into the tree:
 - Find the longest match between the new suffix with the current tree.
 - Insert the new suffix by either:
 - Attaching the new leaf to an existing internal node, or
 - Creating a new internal node and attaching the leaf.
- Worst case run-time is $O(n^2)$, i.e. $s = A^{n-1}$

Faster Construction Algorithms

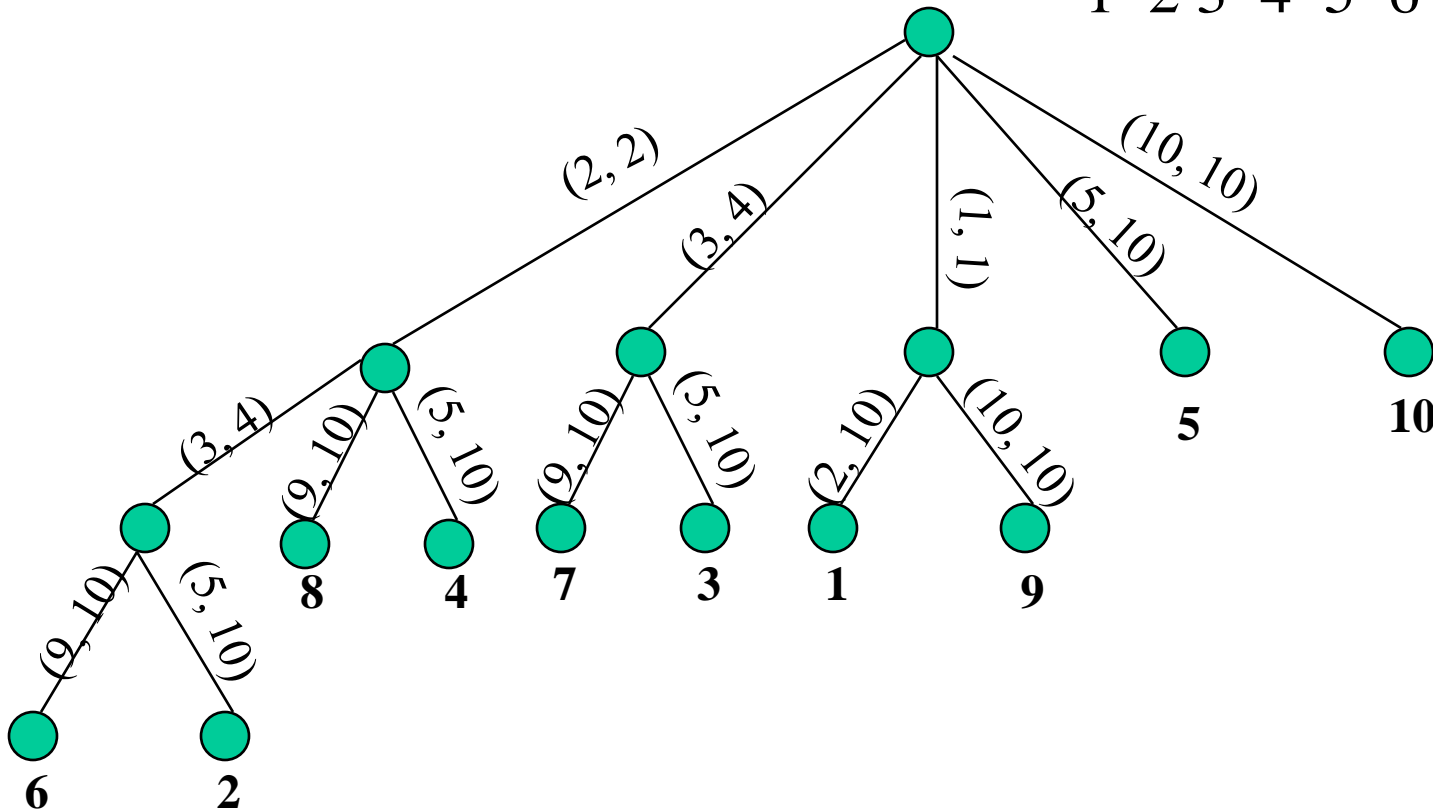
- Linear time
 - $|\Sigma| = O(1)$: Weiner [10], McCreight [7], and Ukkonen [9].
 - $|\Sigma| = O(n)$: Farach [4].

Suffix Tree with Edge Labels

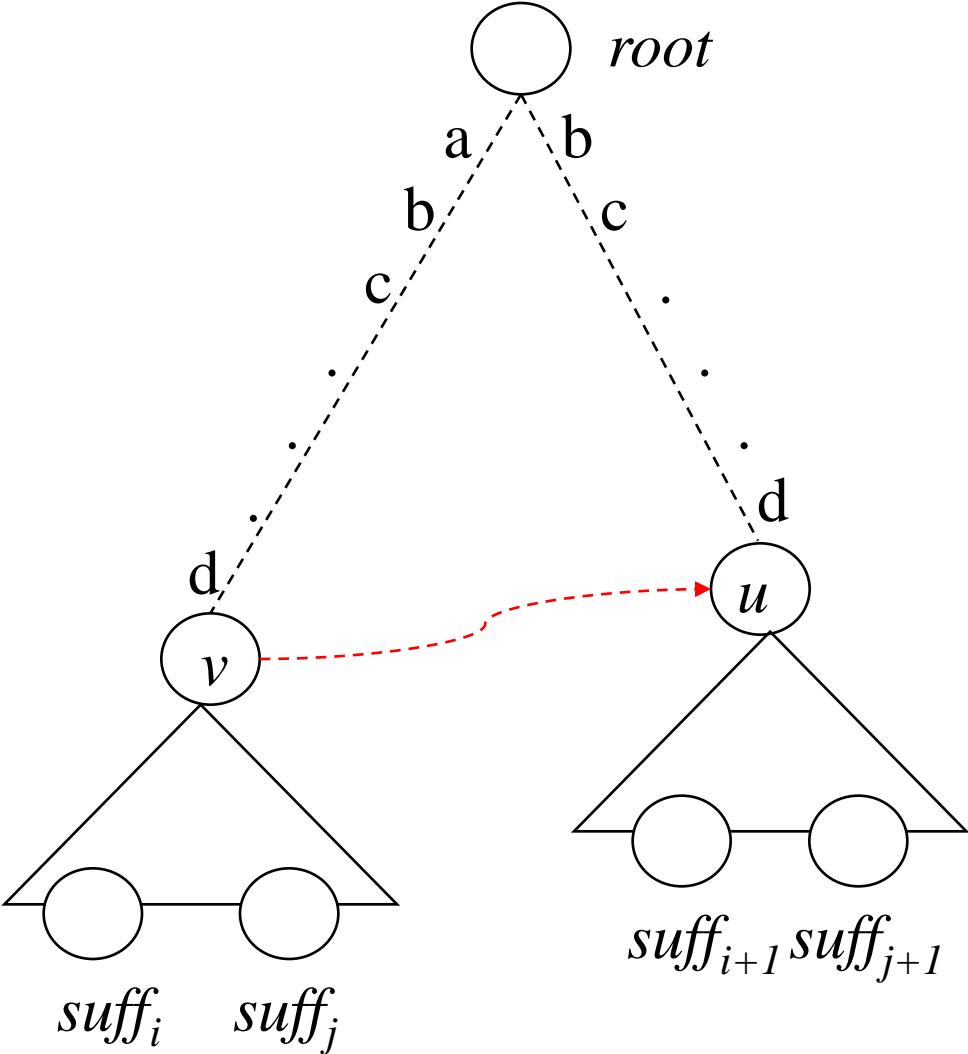


Edge encoding

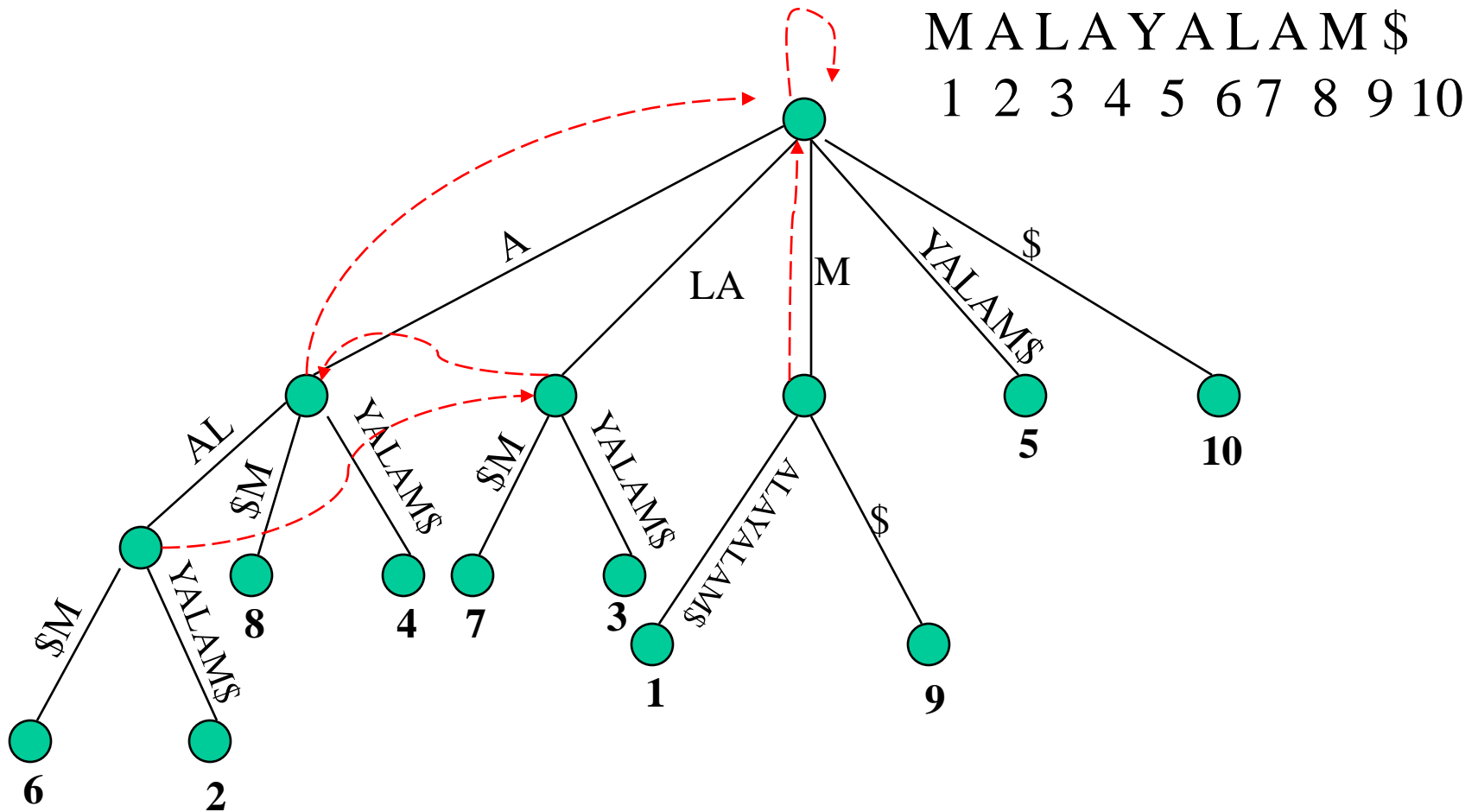
MALAYALAM\$
1 2 3 4 5 6 7 8 9 10



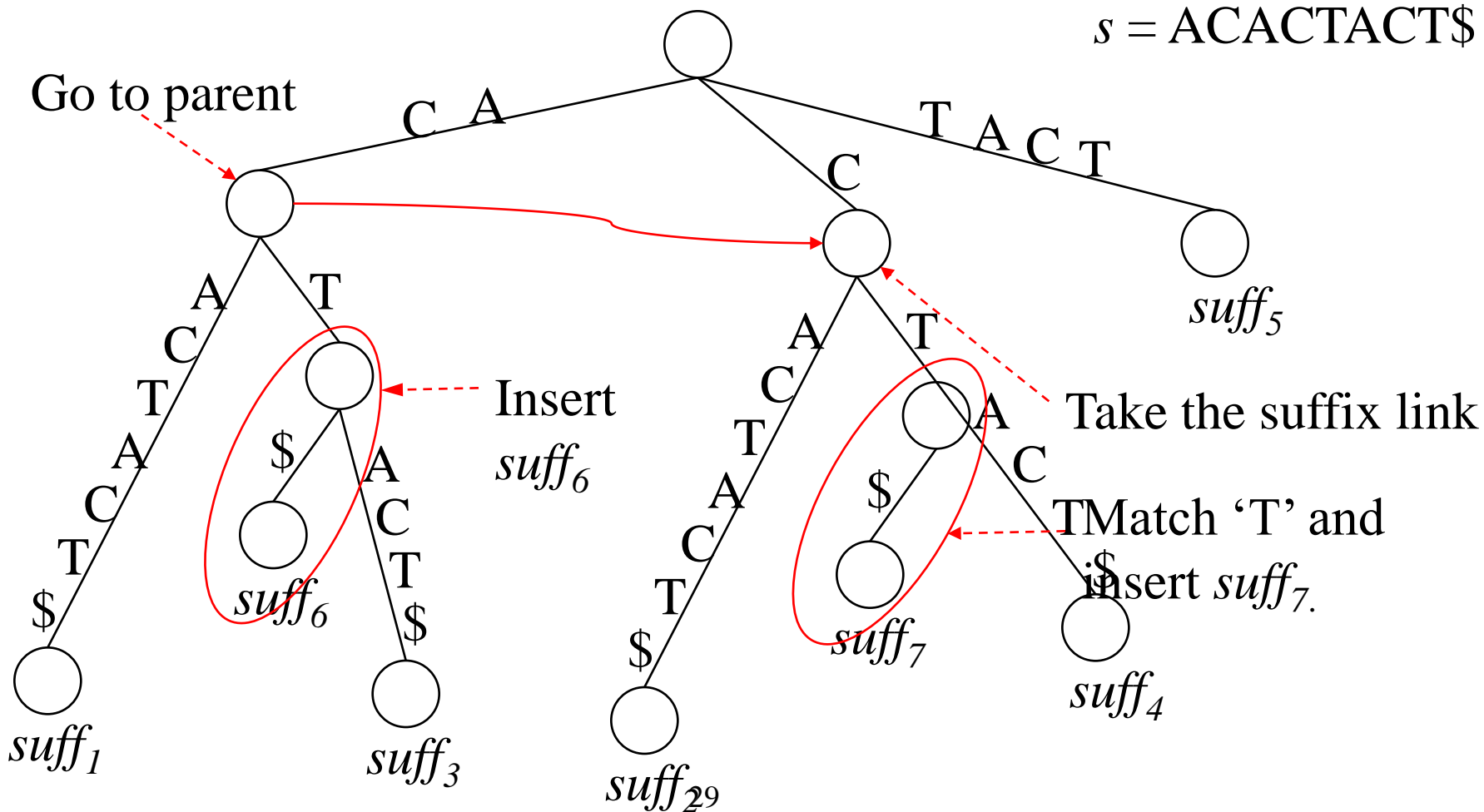
Suffix Links



Suffix Links



McCreight's Algorithm

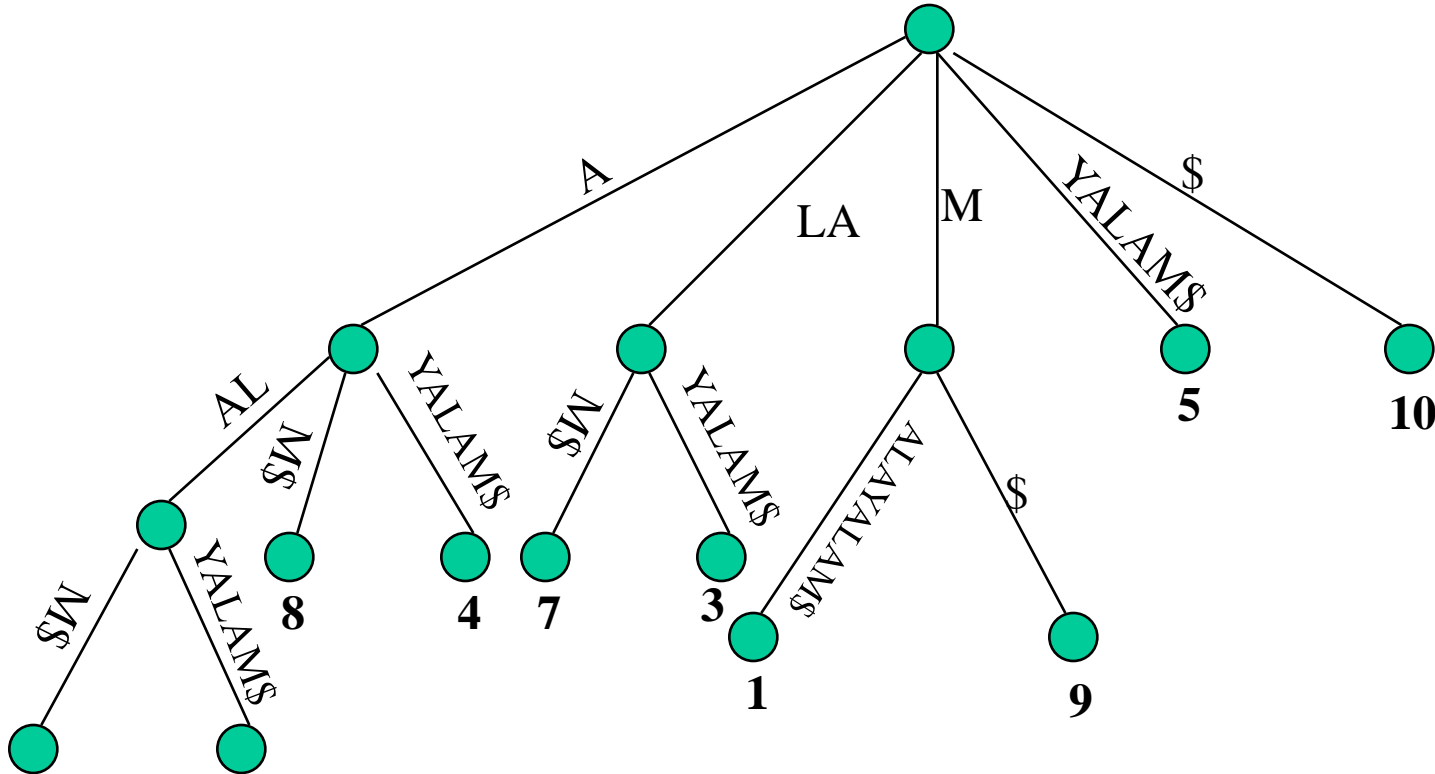


Lowest Common Ancestors

- The lowest common ancestor (*lca*) of two nodes x and y is the deepest node in a rooted tree that is an ancestor of x and y
- Concatenation of edge labels from root to the *lca* a leaf spells out the longest common prefix
- Can be found in constant time after linear preprocessing [Bender00]

A Useful Property of Suffix Trees

String depth ($\text{lca}(i, j)$) = $\text{lcp}(\text{suffix}_i, \text{suffix}_j)$

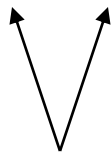


Suffix Array – Reducing Space

MALAYALAM\$
1 2 3 4 5 6 7 8 9 10

6	2	8	4	7	3	1	9	5	10
----------	----------	----------	----------	----------	----------	----------	----------	----------	-----------

Suffix Array



3	1	1	0	2	0	1	0	0	---
----------	----------	----------	----------	----------	----------	----------	----------	----------	------------

lcp Array



Suffix 6 and 2 share “ALA”

6	ALAM\$
2	ALAYALAM\$
8	AM\$
4	AYALAM\$
7	LAM\$
3	LAYALAM\$
1	MALAYALAM\$
9	M\$
5	YALAM\$
10	\$

Pattern Search in Suffix Array

- All suffixes that share a common prefix appear in consecutive positions in the array.
- Pattern P can be located in the string using a binary search on the suffix array.

Naïve Run-time = $O(|P| \times \log n)$.

Improved to $O(|P| + \log n)$ [Manber&Myers93],
and to $O(|P|)$ [Abouelhoda *et al.* 02].

Computing *lcp* Values

1. Find where T_1 is in the suffix array.
2. Compute *lcp* value of T_1 .
3. Find T_2 in the suffix array.
4. Compute *lcp* value of T_2 starting from T_1 's *lcp* - 1.
5. Repeat for all suffixes.

Run-time is linear.

Example

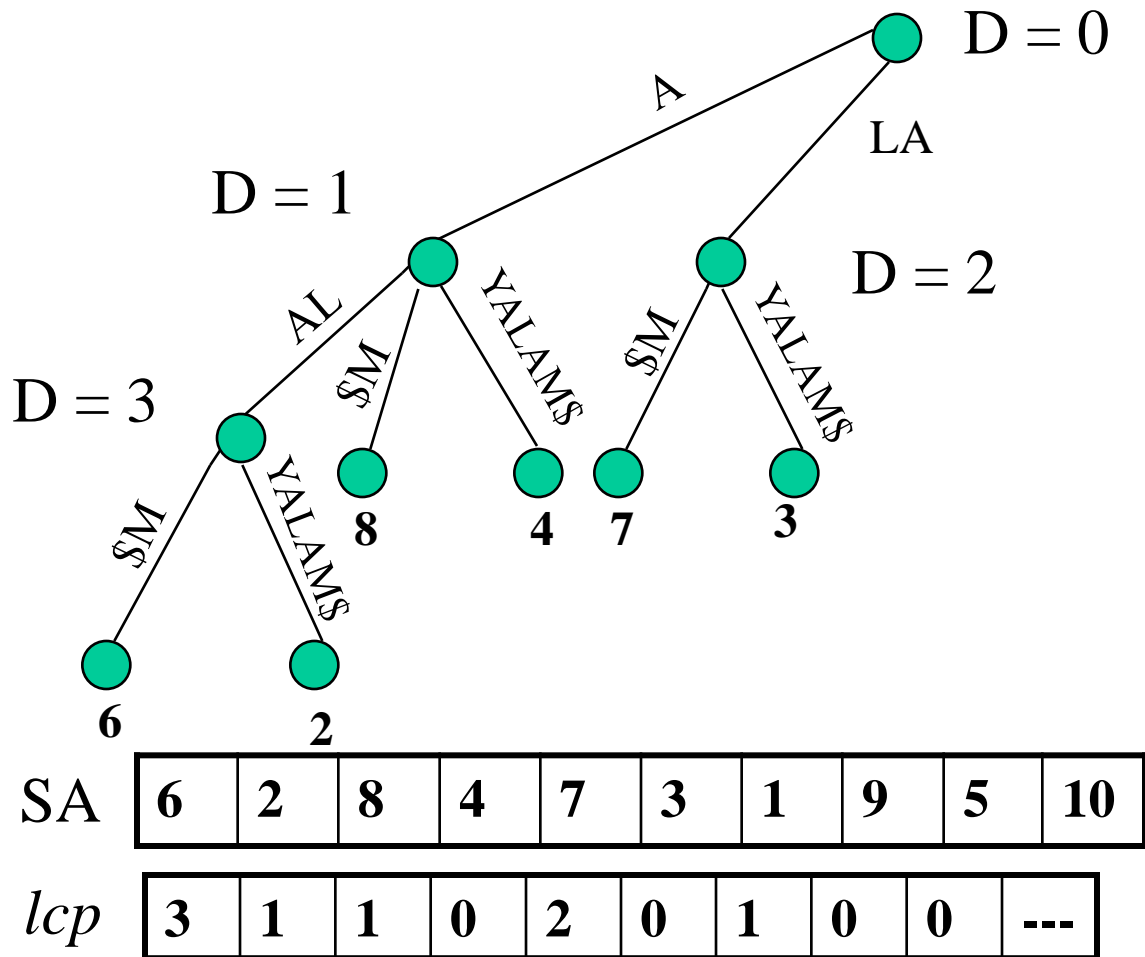
Text	M	A	L	A	Y	A	L	A	M	\$
Position	1	2	3	4	5	6	7	8	9	10
Suffix Array	6	2	8	4	7	3	1	9	5	10
<i>lcp</i> Array	3	1	1	0	2	0	1	0	0	

Suffix Trees vs. Suffix Arrays

Suffix Array = Lexicographic order of the
leaves of the Suffix Tree

Suffix Tree = Suffix Array + *lcp* Array

Building a ST from a SA and *lcp*



6	ALAM\$
2	ALAYALAM\$
8	AM\$
4	AYALAM\$
7	LAM\$
3	LAYALAM\$
1	MALAYALAM\$
9	M\$
5	YALAM\$
10	\$