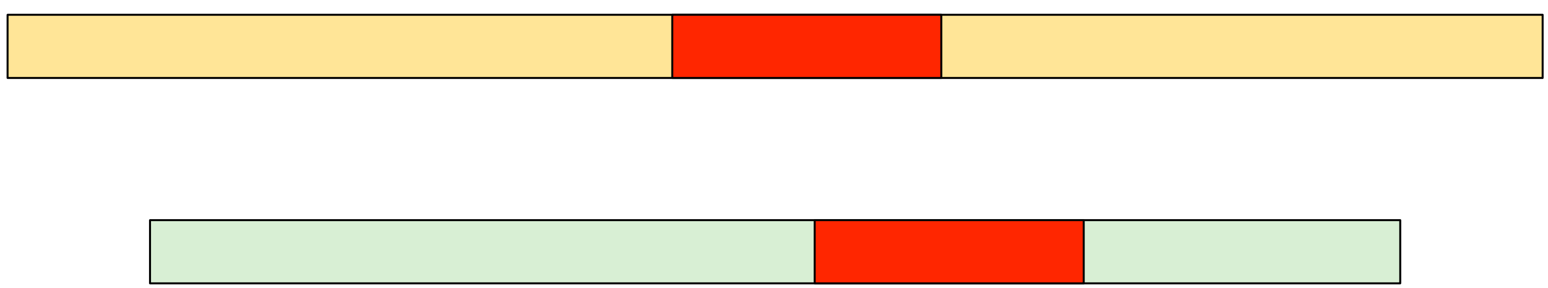
Warm up

- Find the longest common substring of two or more strings.
  - $O(n^3)$ time, $O(n^2)$ time … where n is the total length.
  - A few decades ago, Knuth conjectured that a linear time algorithm for this problem was impossible.



Full-text Indexing

- Bioinformatics research: Search the human genome (about 3 billion characters) for different genes or gene fragments of other species (say, tens to thousands of characters).
- String matching: find the occurrences of a pattern P in a text S.
- KMP algorithm: $O(n + m)$ time, where $n = |S|$ and $m=|P|$.
- Can we do better?
  - We are likely to search the human genome many times for different patterns.
  - Yes. **Build an index** for the human genome to speed up the searching.

Suffix Trees: an old solution for text indexing

- A well-studied main-memory data structures by the theoretical CS community in the 70's to 90's.
  - In recent years, the database community is also interested in suffix trees stored in external memory.
- Space: O(n) words; best implementation requires 40+ Giga bytes to index the human genome (~3G).
- Today's lecture
  - Simple applications of suffix trees: pattern matching in O(|P| + occ) time.
  - construction of suffix trees: O(n) time.

Coming lectures

- 1990's: Suffix arrays, n words (12 Gigabytes for human genome)
- Can we further reduce the space for text indexing?
- Note that just to represent the text, it requires $n \log \Sigma$ bits in the worst case, where Σ = the alphabet size.
- Open problem (before): an $O(n \log \Sigma)$-bit index or even $(n \log \Sigma)$-bit index.
- Sometimes a text (say, "aaaaaa…ab") can be compressed to occupy less than $n \log \Sigma$ bits.

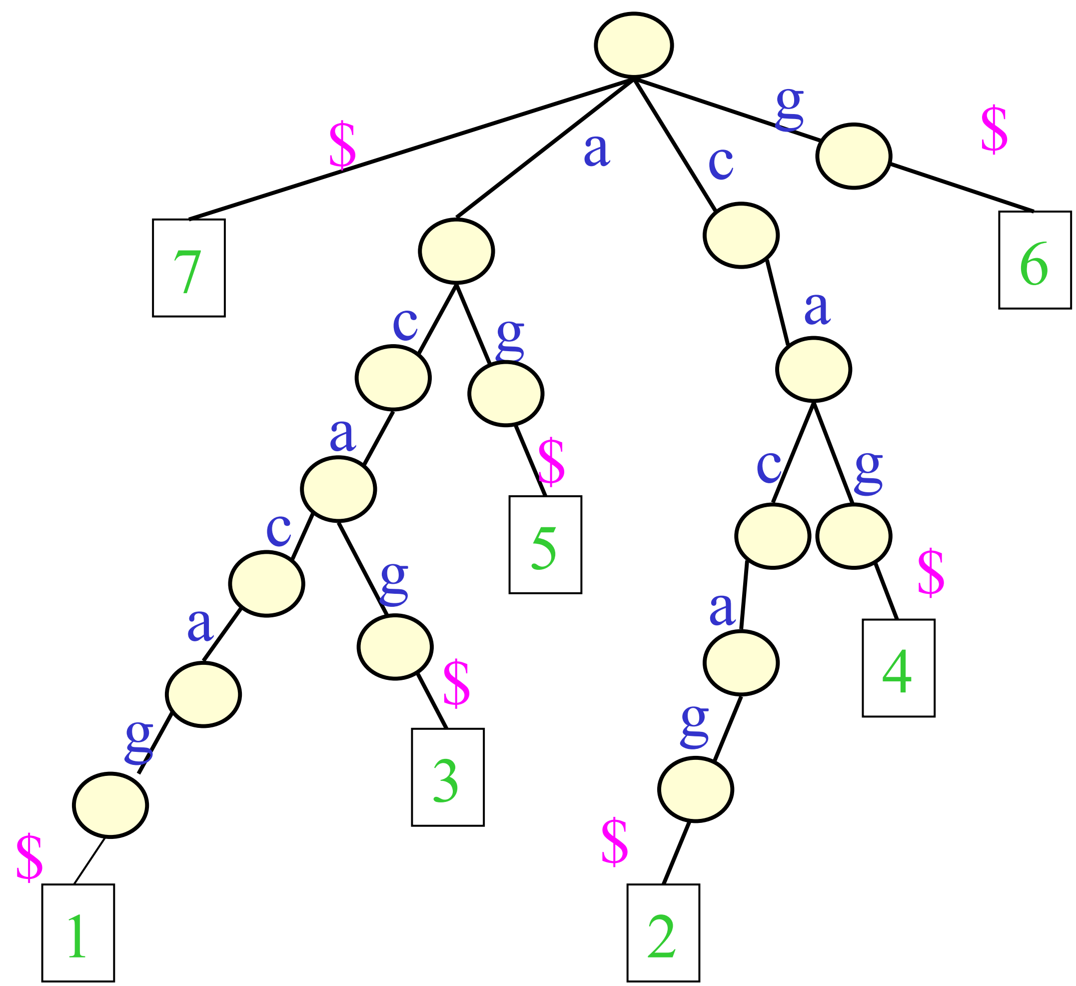
  Can we have an index whose size depends on the size of the "compressed" text?
- Breakthrough in early 2000's: FM-index, Compressed suffix arrays.

Suffix Tries

Suffix Trie: a tree of all possible suffices

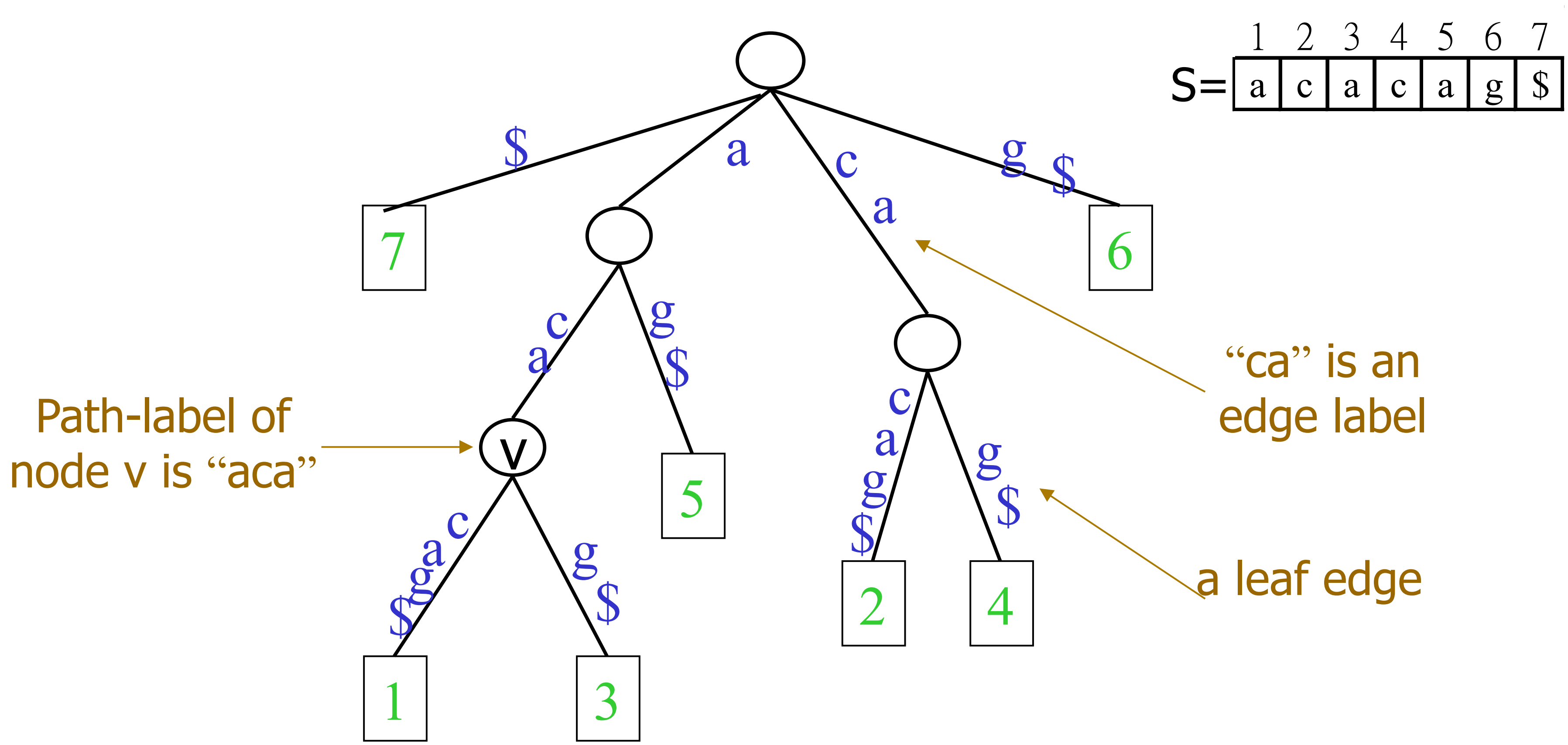
E.g. S = acacag$

	Suffix
1	acacag$
2	cacag$
3	acag$
4	cag$
5	ag$
6	g$
7	$



Suffix Trees (I)

Suffix Tree: Eliminate nodes with only one child

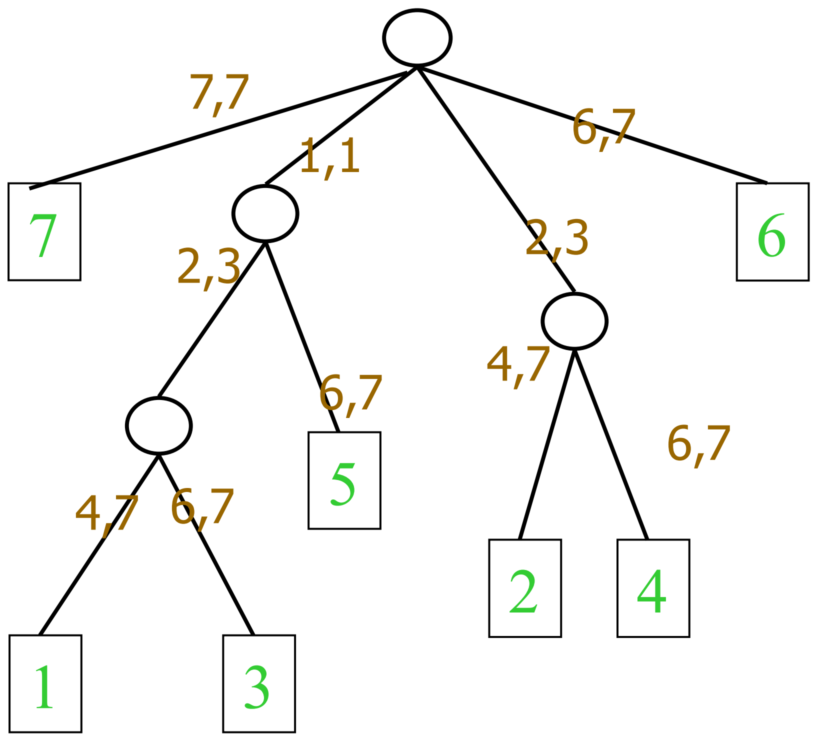


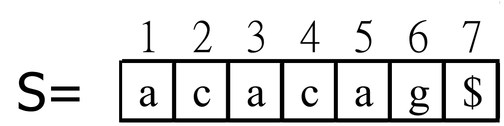
Suffix Trees (II)

A suffix tree has exactly n leaves, at most n-1 internal nodes and at most 2n-1 edges.

The label of each edge can be represented by 2 indexes.

Thus, suffix tree can be represented using O(n log n) bits

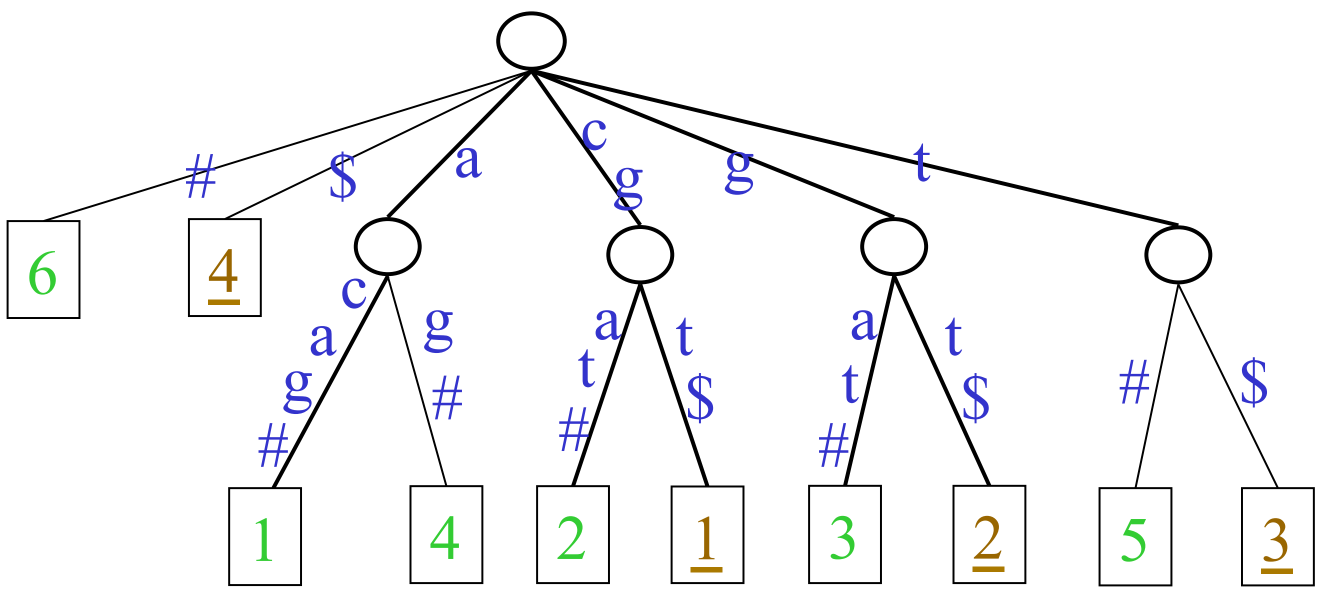




NB. For a leaf, the end index is 7.
Thus, we only store the start index.

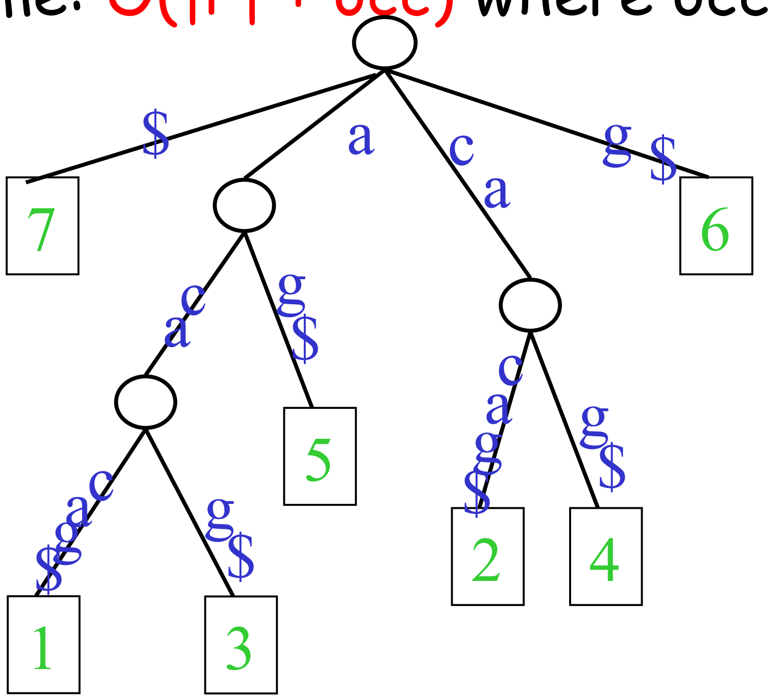
Generalized suffix tree

- Build a suffix tree for two or more strings
- E.g. S_1 = acgat#, S_2 = cgt$



Pattern matching

- Find all occurrences of a given pattern P in S
  - Traverse the suffix tree starting from the root according to P.
  - All the leaves in the subtree rooted at x are the occurrences of P.
- Time: O(|P| + occ) where occ is the no. of occurrences.



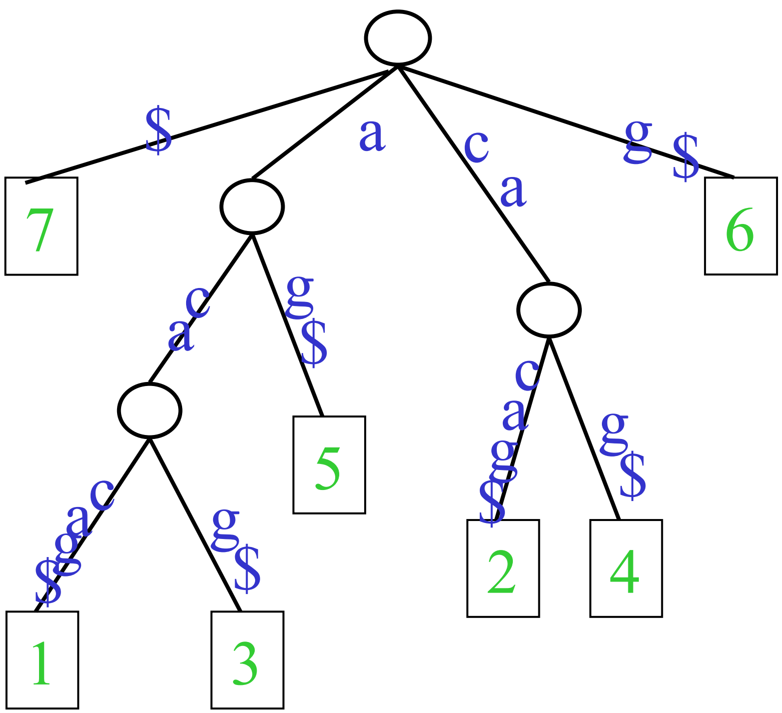
E.g. S = acacag$
P = aca
Occurrences: 1, 3

Other applications

- Find the longest repeated substring in S

Other applications

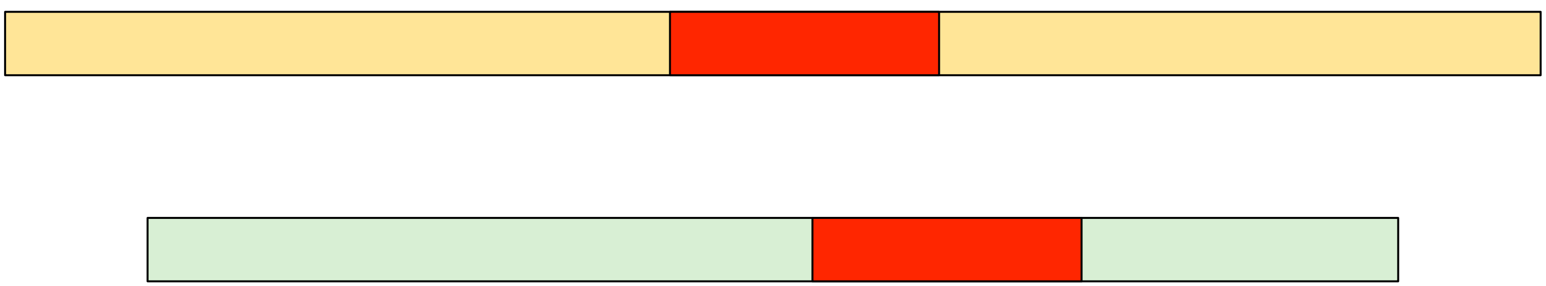
- Find the longest repeated substring in S
  - the deepest internal node
- Time: O(n)



E.g. S = acacag$
The longest repeat is aca.

Other applications

- Find the longest common substring of two or more sequences
  - About 30+ years ago, Knuth conjectured that a linear time algorithm for this problem was impossible.



Construction of suffix trees

- Consider $S = s_1s_2\ldots s_n$ where s_n=$
- Algorithm:
  - Initialize the tree with only a root
  - For i = n to 1
    - include S[i..n] into the tree
- Time: $O(n^2)$

Less than quadratic time

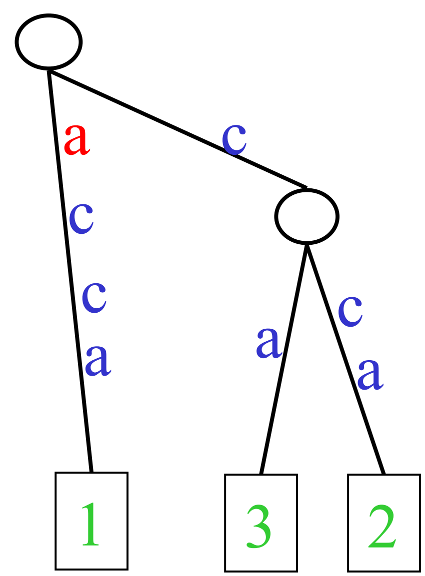
- Yes. We can construct it in O(n) time.
- Weiner [1973]
  - Linear time for constant-size alphabet, space?
- McGreight [JACM 1976]
  - Linear time for constant-size alphabet
- Ukkonen [Algorithmica, 1995]
  - Online construction, linear time for constant-size alphabet, less space
- Farach [FOCS 1997]
  - Linear time for general alphabet

- Today, we discuss Ukkonen's algorithm

Implicit suffix trees

- Let S be a string without the ending $.
- A suffix S[i..n] can be a prefix of another longer suffix S[j..n]
  - S[i..n] is excluded from the implicit suffix tree of S.
- The implicit suffix tree contains all suffixes that are not prefix of other suffixes.

S=acca



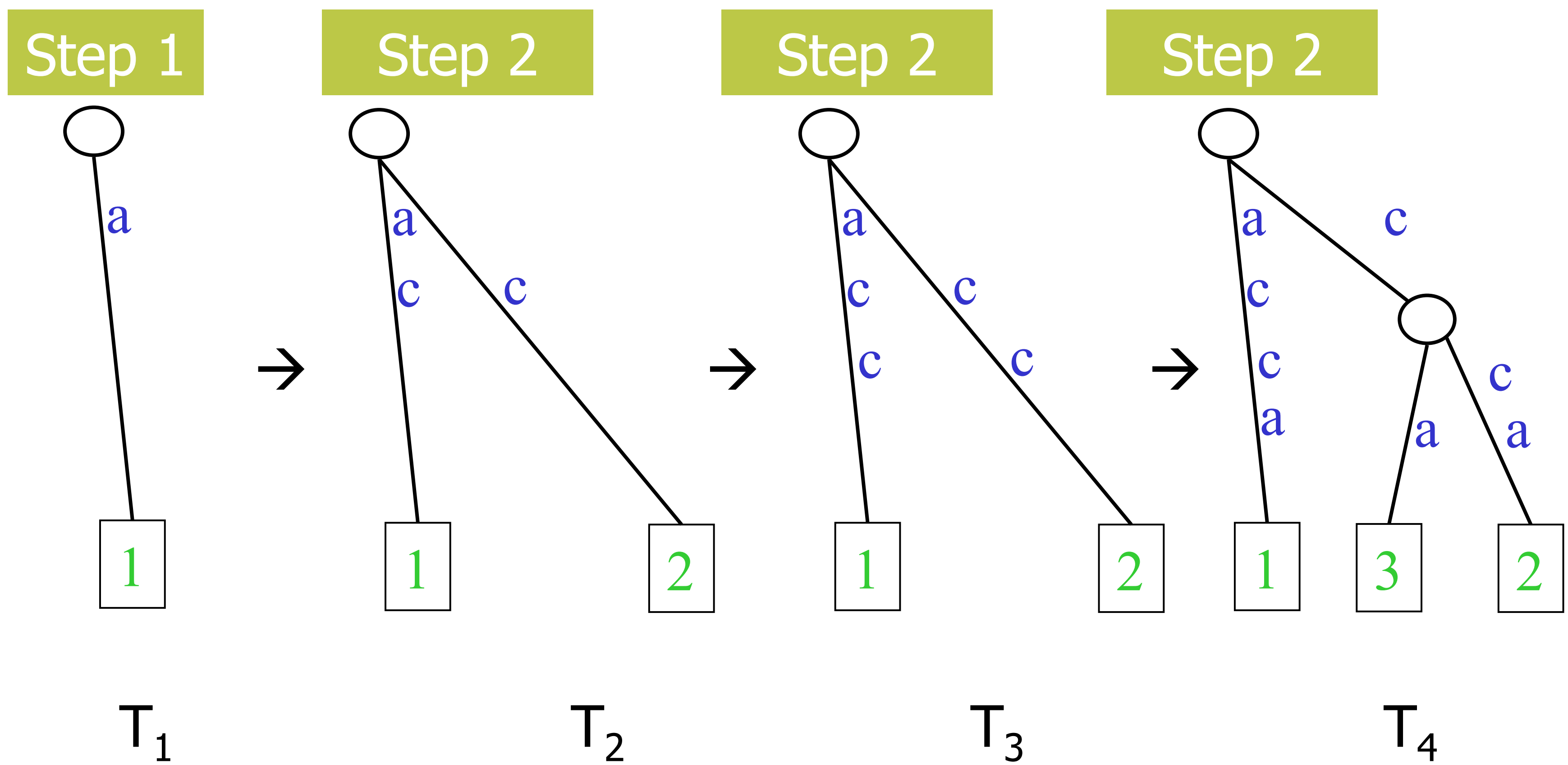
Algorithm

Denote T_i be the implicit suffix tree for S[1..i].

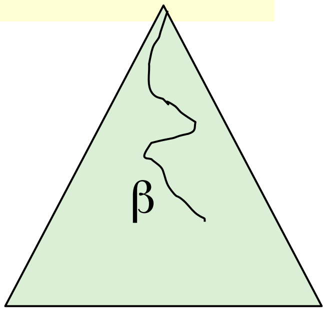
1. Construct T_1
2. For i = 1 to m-1
   - /* Phase i: Construct T_{i+1} from T_i */

Illustration

- S=acca



Constructing T_{i+1} from T_i



For j = 1 to i+1

- /* extend each suffix S[j..i] to S[j..i+1] */
- Starting from the root, find the endpoint of the path labeled β = S[j..i]
- Extend the path with character S[i+1]
  - Rule 1: If β ends at a leaf, S[i+1] is appended to the label of the last edge to the leaf.
  - Rule 2: If every path from β starts with a character ≠ S[i+1], create a new leaf and a leaf edge labeled with S[i+1].
  - Rule 3: If some path from β starts with character S[i+1], do nothing.

Example: from T_3 to T_4

Rule 1: If β ends at a leaf, S[i+1] is appended to the label of the last edge to the leaf.

Rule 2: If every path from β starts with a character ≠ S[i+1], create a new leaf and a leaf edge labeled with S[i+1].

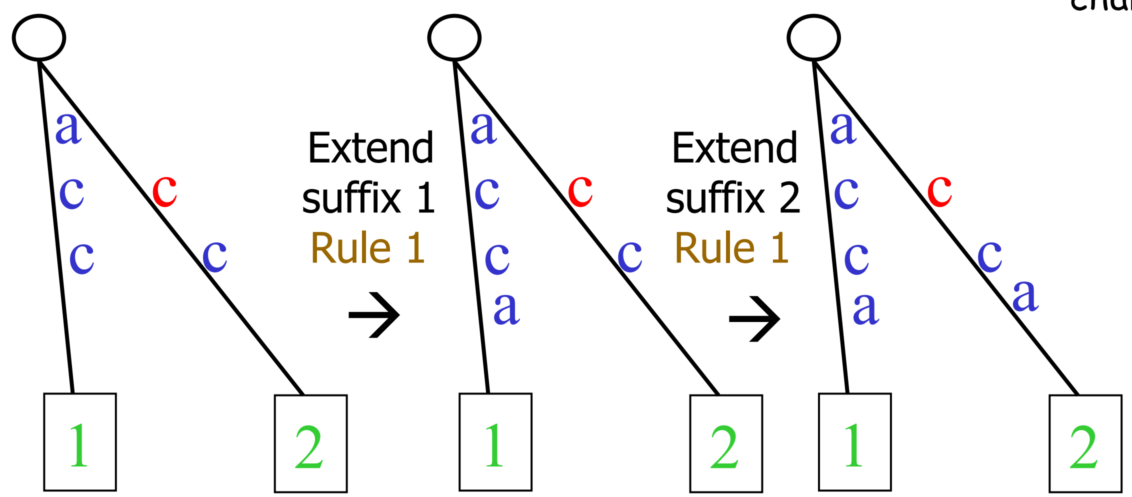
Rule 3: If some path from β starts with character S[i+1], do nothing.

S_3=

1	2	3
a	c	c

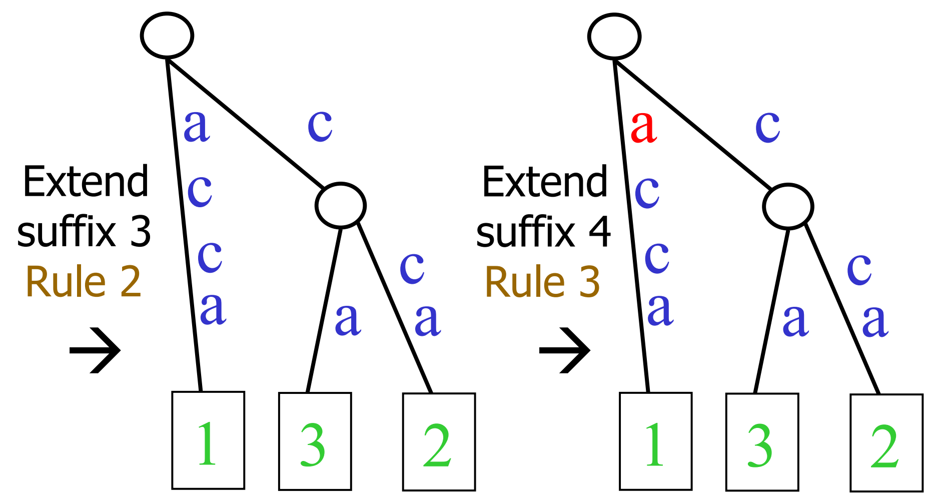
S_4=

1	2	3	4
a	c	c	a



T_3

T_4



Observation 1

- Consider Phase i (constructing T_{i+1} from T_i)

  Once we apply rule 3 to extend S[j..i], then
  - rule 3 will be applied for extending S[k..i] for k = j+1,…, i
  - Thus, nothing to be done for k = j+1,…, i

Proof:

- Since rule 3 is applied to extend S[j..i], T_i contains a path labeled S[j..i] followed by the character S[i+1].
- Thus, there is also a path for S[k..i] for k > j, followed by S[i+1].

Remark

- Based on Observation 1, in Phase i, once we have applied rule 3, we can stop.
- This saves a lot of work.

Observation 2

- Once we add a leaf for a suffix in T_i, that leaf remains in T_{i+1}, T_{i+2} , …

- Proof:
  - We never remove any leaves.

Remark

- In Phase i (i.e. constructing T_{i+1} from T_i), let H_i be the last extension that makes use of rule 1 or rule 2 to extend $S[H_i \,..i]$.
  - In other words, for extension of $j = 1$ to H_i, we do not perform any rule 3. That is, $S[\,j..i+1]$ at a leaf in T_{i+1}.
- In Phase $i+1$ (i.e. constructing T_{i+2} from T_{i+1}), for j **= 1 to** H_i, when searching T_{i+1} for $S[\,j..i+1]$, we always encounter a leaf at the end of $S[\,j..i+1]$.
  - Thus, only rule 1 is applied to extend $S[\,j..i+1]$ to $S[\,j..i+2]$, and there is no structural change to T_{i+1}
  - Structural change occurs only starting from $j > H_i$

Algorithm for Phase i

- /* For j=1...H_{i-1}, extension of j is based on rule 1. No change to the structure of the tree. */
- For j= H_{i-1} +1 to i+1,
  - Find the endpoint of the path from the root labeled with S[j..i]
  - Extend the path with character S[i+1] based on rule 1, 2, or 3
  - If we extend the path with rule 3,
    - /* extension j' for j'=j+1…i+1 are also based on rule 3. So, no need to do anything */
    - Set H_i = j-1
    - Break (exit the for loop)

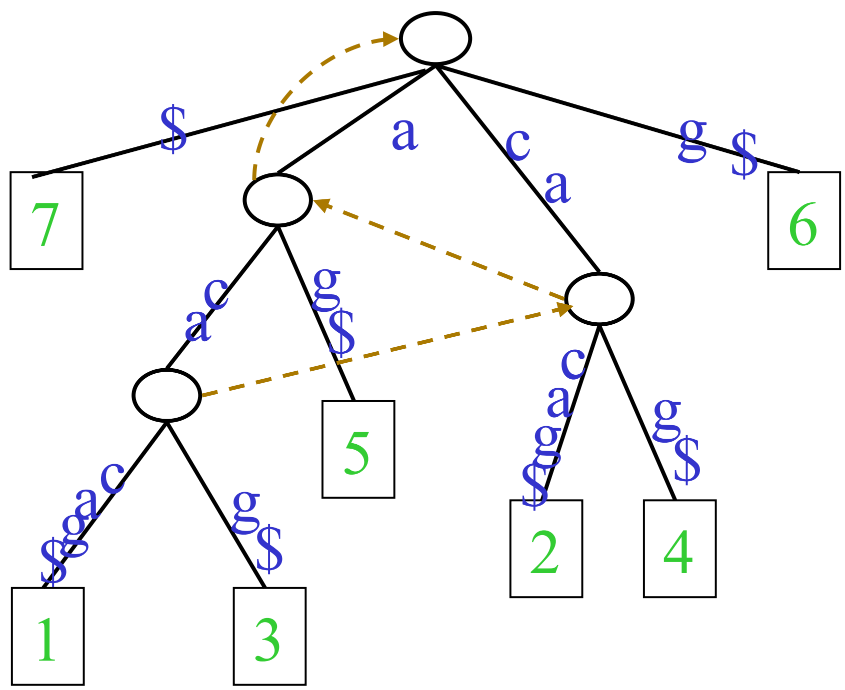
Whole process

- Summary
  - Phase 1: we compute extension for $j = 1..H_1+1$.
  - Phase 2: we compute extension for $j = H_1+1..H_2+1$.
  - Phase 3: we compute extension for $j = H_2+1..H_3+1$.
  - …
  - Phase i: we compute extension for $j = H_{i-1}+1..H_i+1$.
  - …
- In total, we will do at most 2n extensions.
- For an extension due to S[j..i], it takes O(n) time because we need to find the endpoint of S[j..i].
- The total time is $O(n^2)$.
- The process is accelerated using suffix link.

Suffix links

x is a single character

- For an internal node v with path-label xα, if there is another node s(v) with path-label α, than we create a suffix link from v to s(v).

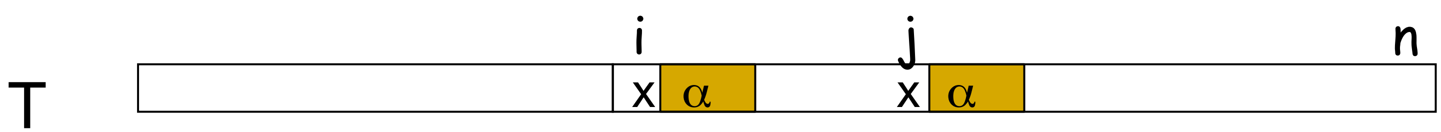


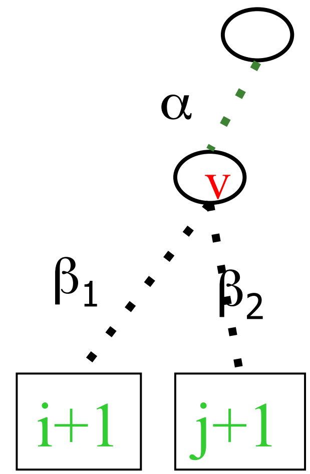
Suffix links are well defined

Lemma. In a suffix tree, every internal node u (except the root) has a suffix link v.

Proof:

- Consider any internal node u with path-label $x\alpha$.
- $x\alpha$ is the longest common prefix of some suffixes T[i..n] and T[j..n]

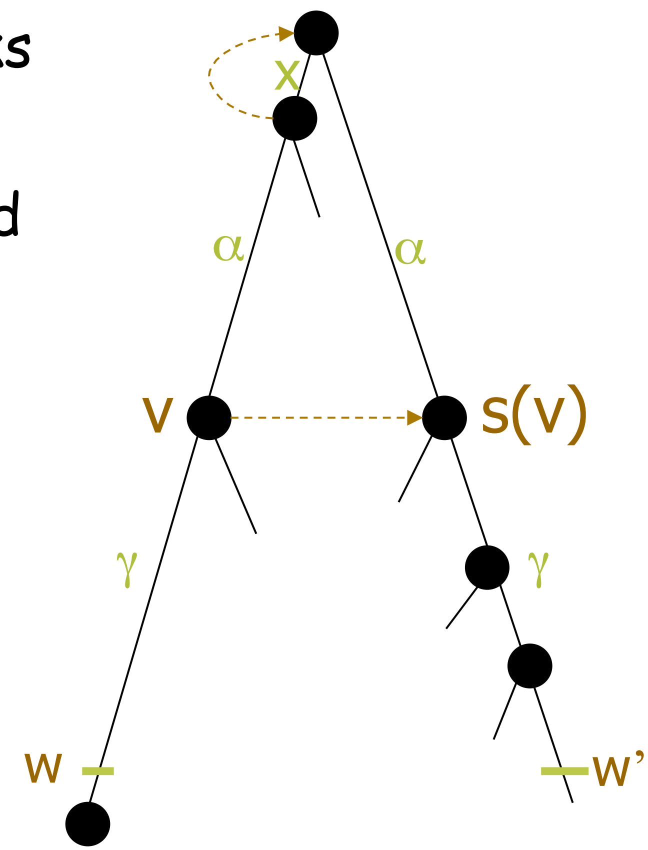




- Then α is the longest common prefix of T[**i+1**..n] and T[**j+1**..n]
- Thus, there is an internal node v with path-label α.
- suffix link of u = v

How to use suffix link?

- Assume that before extension due to S[j..i], we've maintained the suffix links for all internal nodes.
- In the extension for j, we have located the end of S[j..i], denoted w.
- To start extension for (j+1), we go to the end of S[j+1..i] as follows:
  i. From w, go up one edge to v
  ii. Through **suffix link**, go to s(v)
  iii. Go down a number of nodes until we find the end of S[j+1..i], say, w'.
  iv. If w ends at a new internal node, create a suffix link from w to w' (exists already or create it now).



Time complexity

- Find the end of S[j+1..i]:
  - Steps i, ii, and iv take O(1) time
  - Step iii takes amortized O(1) time.
- So, each extension can be solved in O(1) time.
- As there are 2n extensions, the total time is O(n).

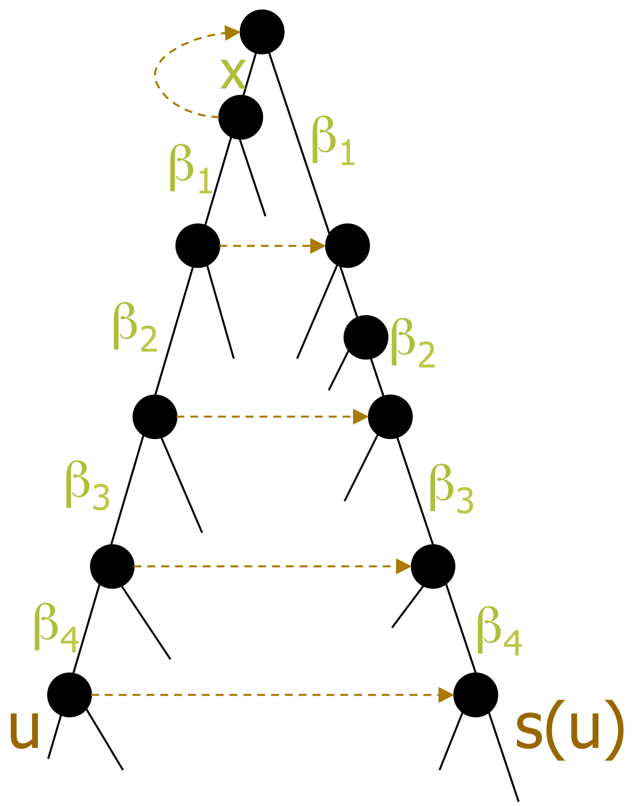
Average over all step iii

Step 3 takes amortized O(1) time

- Define node-depth to be the depth of a node from the root (we count the number of nodes).
- Note that for each extension,
  - Step i reduces the node-depth by 1.
  - Step ii reduces the node-depth by at most 1. See next slide.
  - Step iii increases the node-depth.
- Since there are at most 2n extensions,
  - All steps 1 and 2 can reduce the node-depth by at most 4n
- Since the maximum node-depth is n,
  - All steps 3 can at most increase the node-depth by 5n.
  - Each step 3 goes down O(1) nodes on average.

Suffix link and depth

- For every internal node u, depth of s(u) ≥ depth of u - 1
- Proof:
  - For every ancestor w of v except the root and the one closest to the root, s(w) should be an ancestor of s(v).



Disadvantage of suffix trees

- Suffix tree is space inefficient. It has $O(n)$ nodes and requires $O(n|\Sigma|)$ words or $O(n|\Sigma|\log n)$ bits.
- Manber and Myers (SIAM J. Comp 1993) proposes a new data structure, called suffix array, which has a similar functionality as suffix tree. Moreover, it only requires n words or $O(n \log n)$ bits.
- Compressed suffix arrays, suffix trees: $O(n)$ bits.