Tries and Suffix Trees

Inge Li Gørtz

String indexing problem

- String matching problem. Given strings T (text) and P (pattern) over an alphabet Σ, report starting positions of all occurrences of P in T.
  - *Finite automaton:* $O(m\Sigma + n)$ time and space
  - *KMP:* $O(m+n)$ time and space

- String indexing problem. Given a string S of characters from an alphabet Σ. Preprocess S into a data structure to support
  - Search(P): Return starting position of all occurrences of P in S.

- Today: Data structure using $O(n)$ space and supporting Search(P) in $O(m)$ time.

- Applications:
  - Search engines, e.g. prefix searches.
  - Finding common substrings of many biological strings
  - Finding repeating substructures in biological strings
  - Detecting DNA contamination

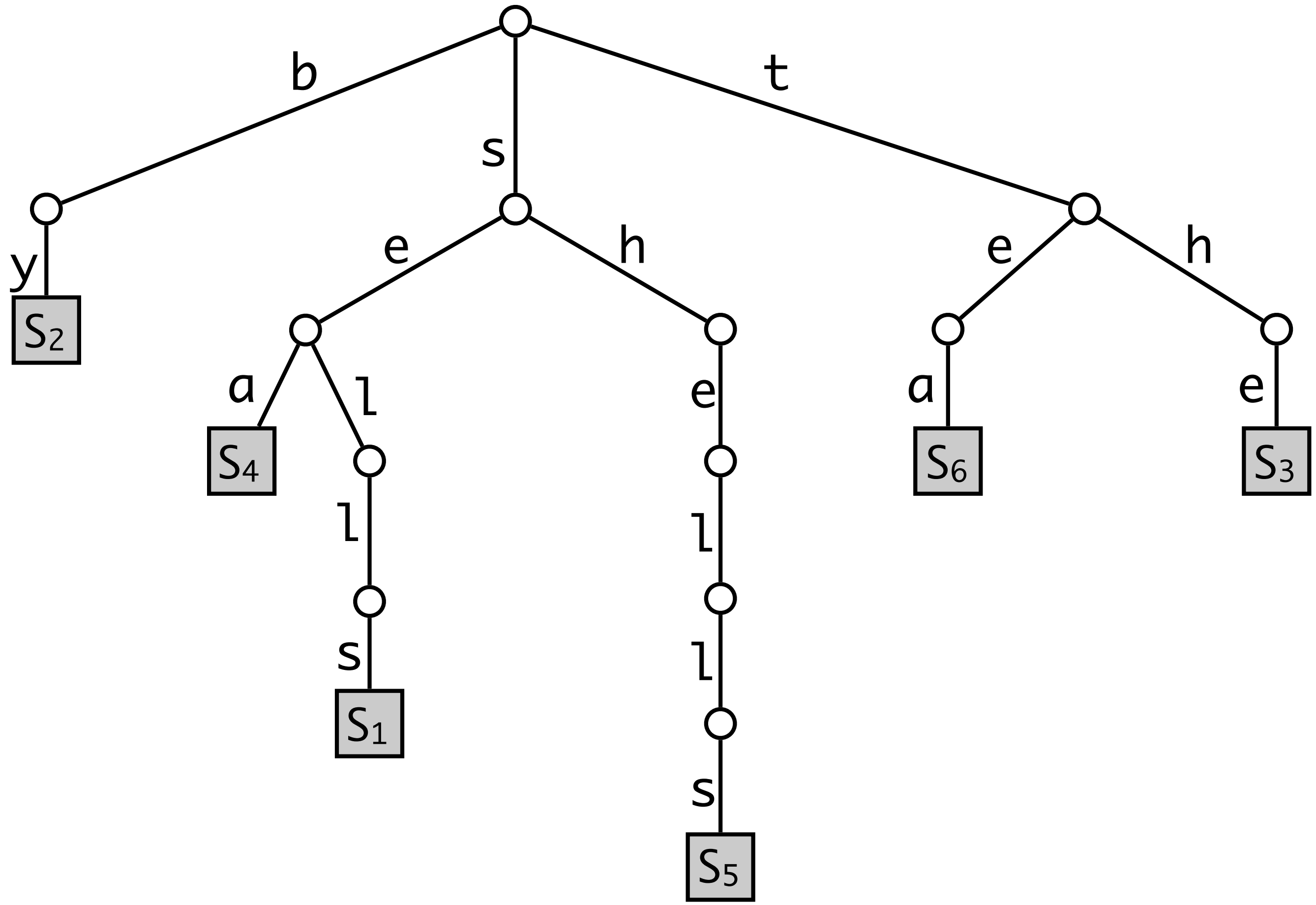
Outline

- Tries
- Compressed tries
- Suffix trees
- Applications of suffix trees

Tries

Tries

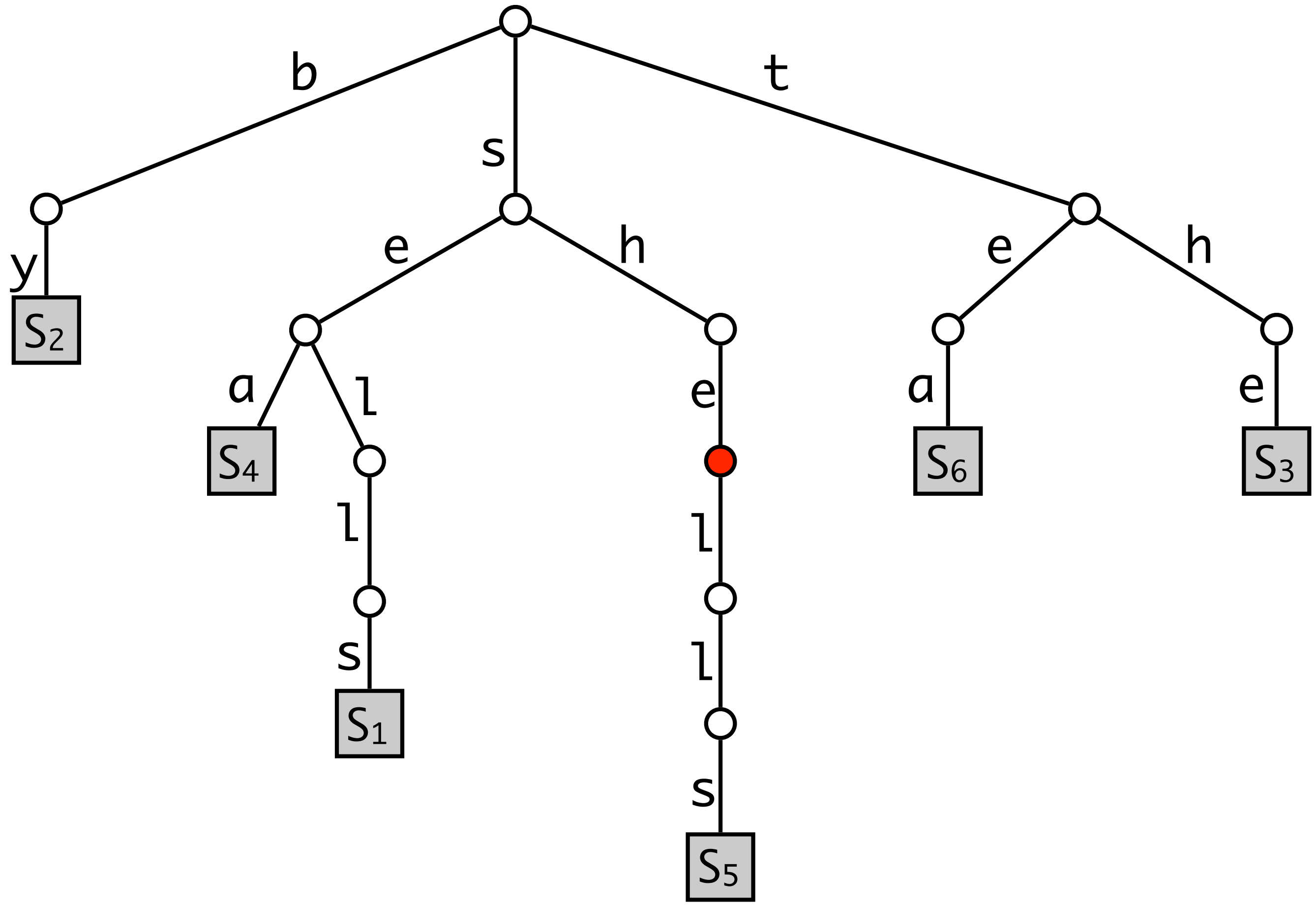
- Text re**trie**val



- Trie over the strings: sells, by, the, sea, shells, tea.

Tries

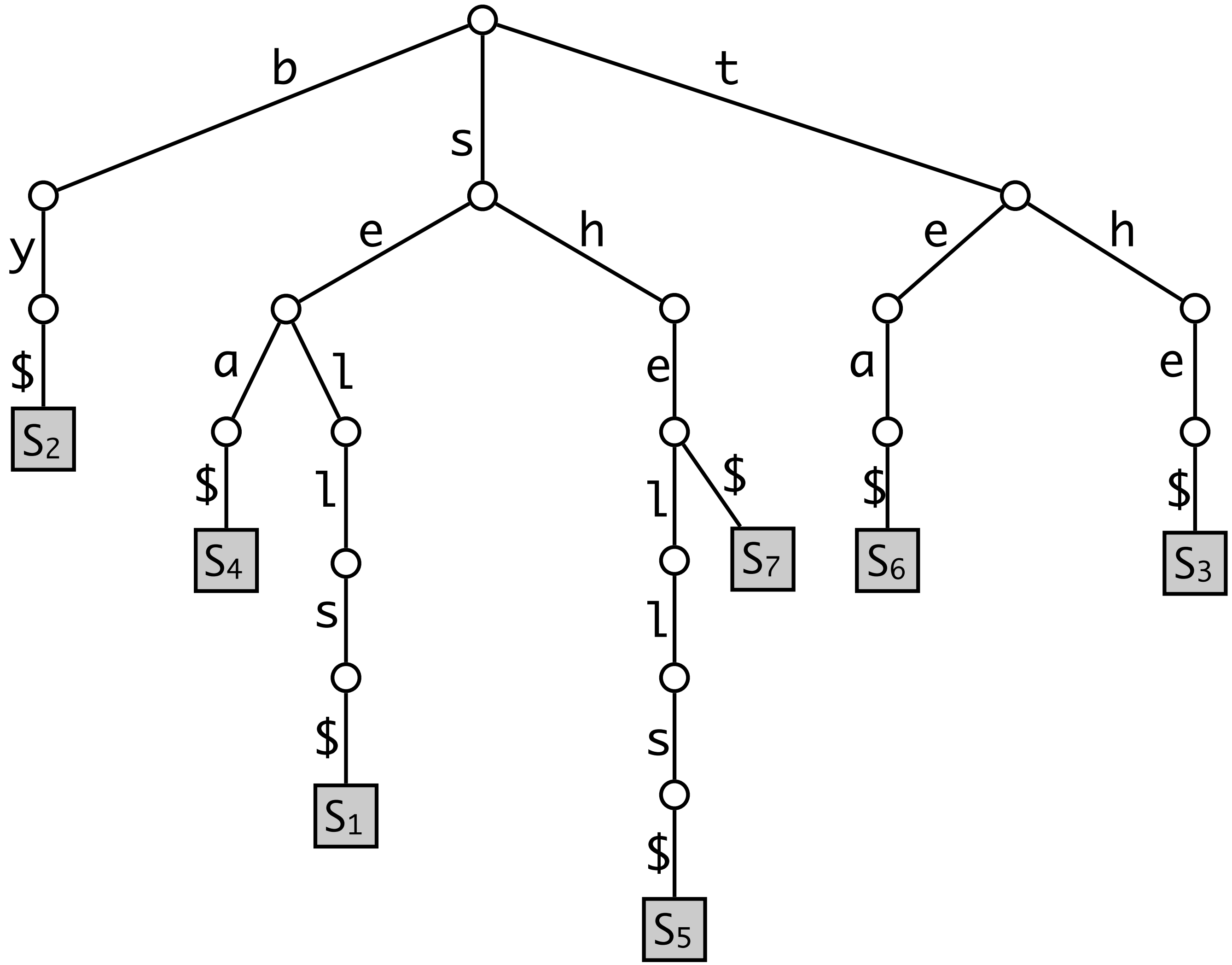
- Text retrieval
- Prefix-free?



- Trie over the strings: sells, by, the, sea, shells, tea, *she*.

Tries

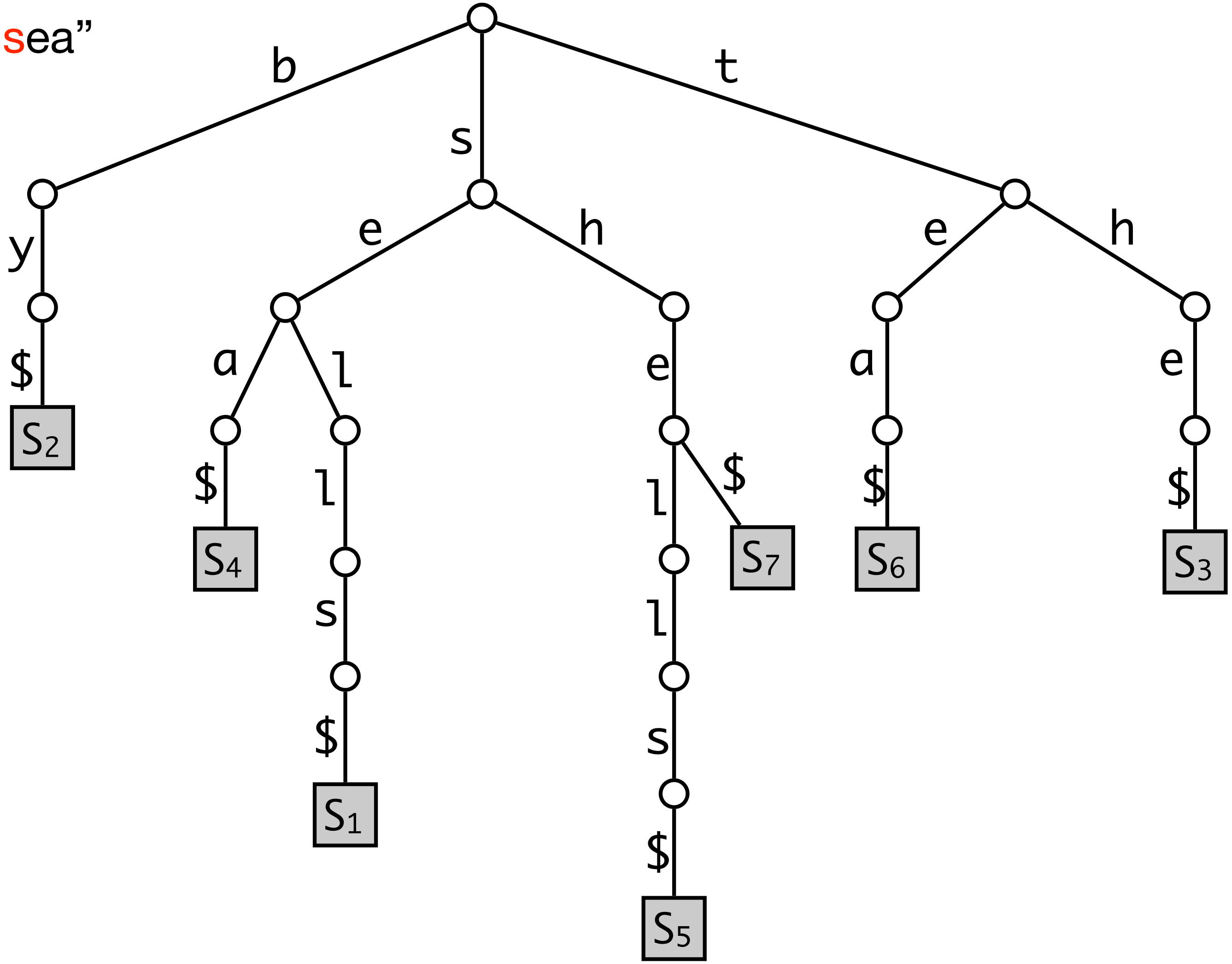
- Text retrieval
- Prefix-free?



- Trie over the strings: sells, by, the, sea, shells, tea, *she*.

Tries

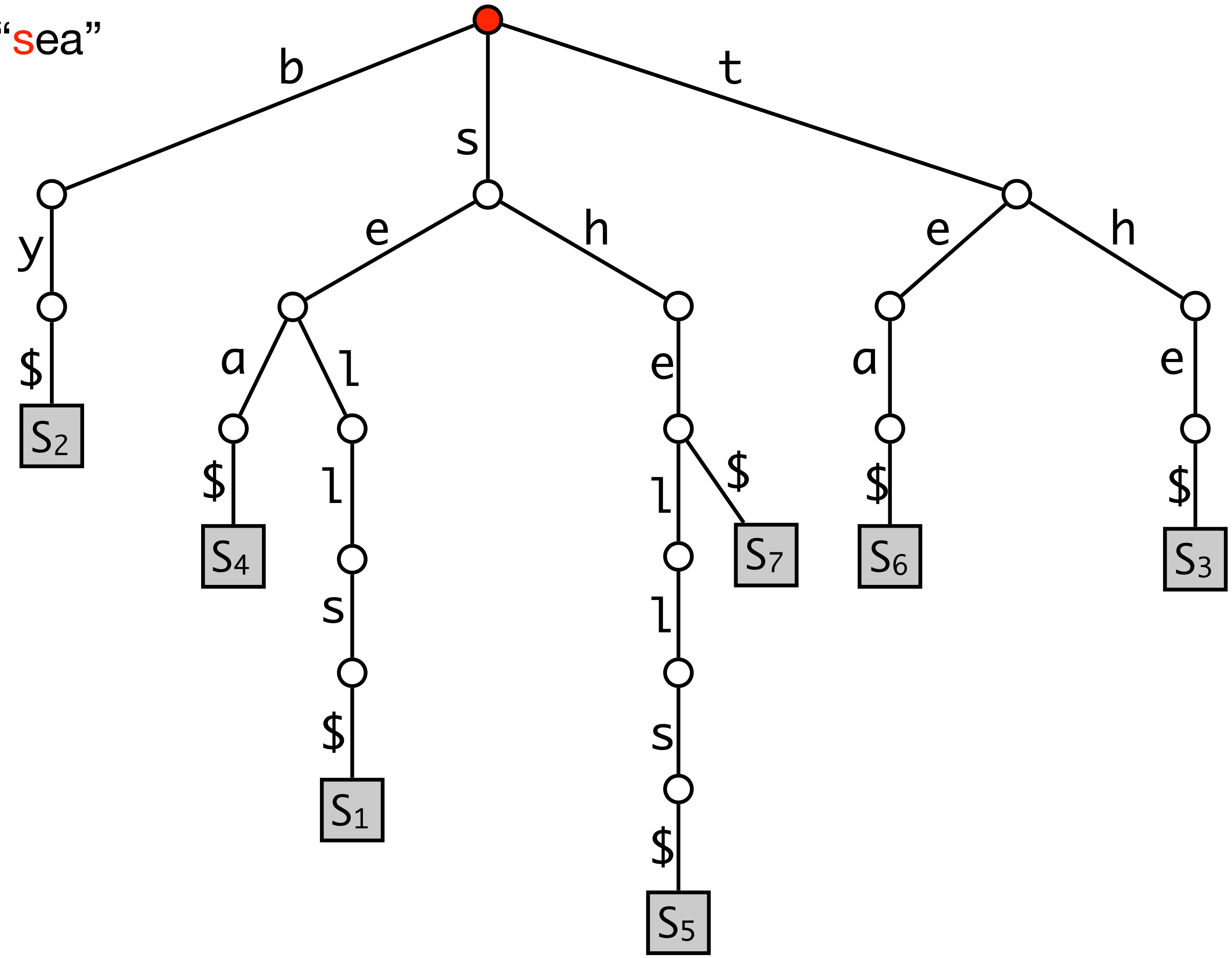
- Text re**trie**val
- Search for "**s**ea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

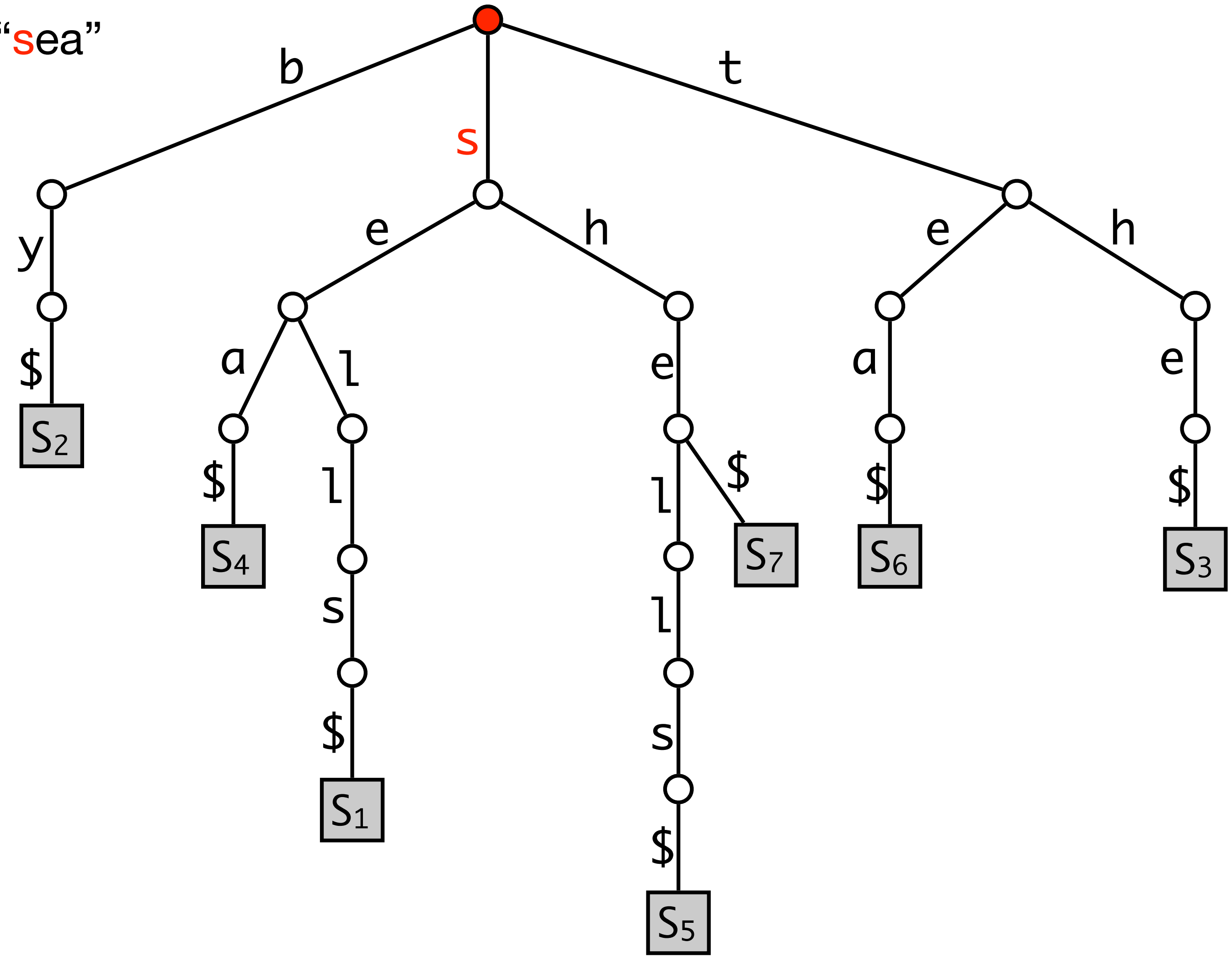
- Text retrieval
- Search for "sea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

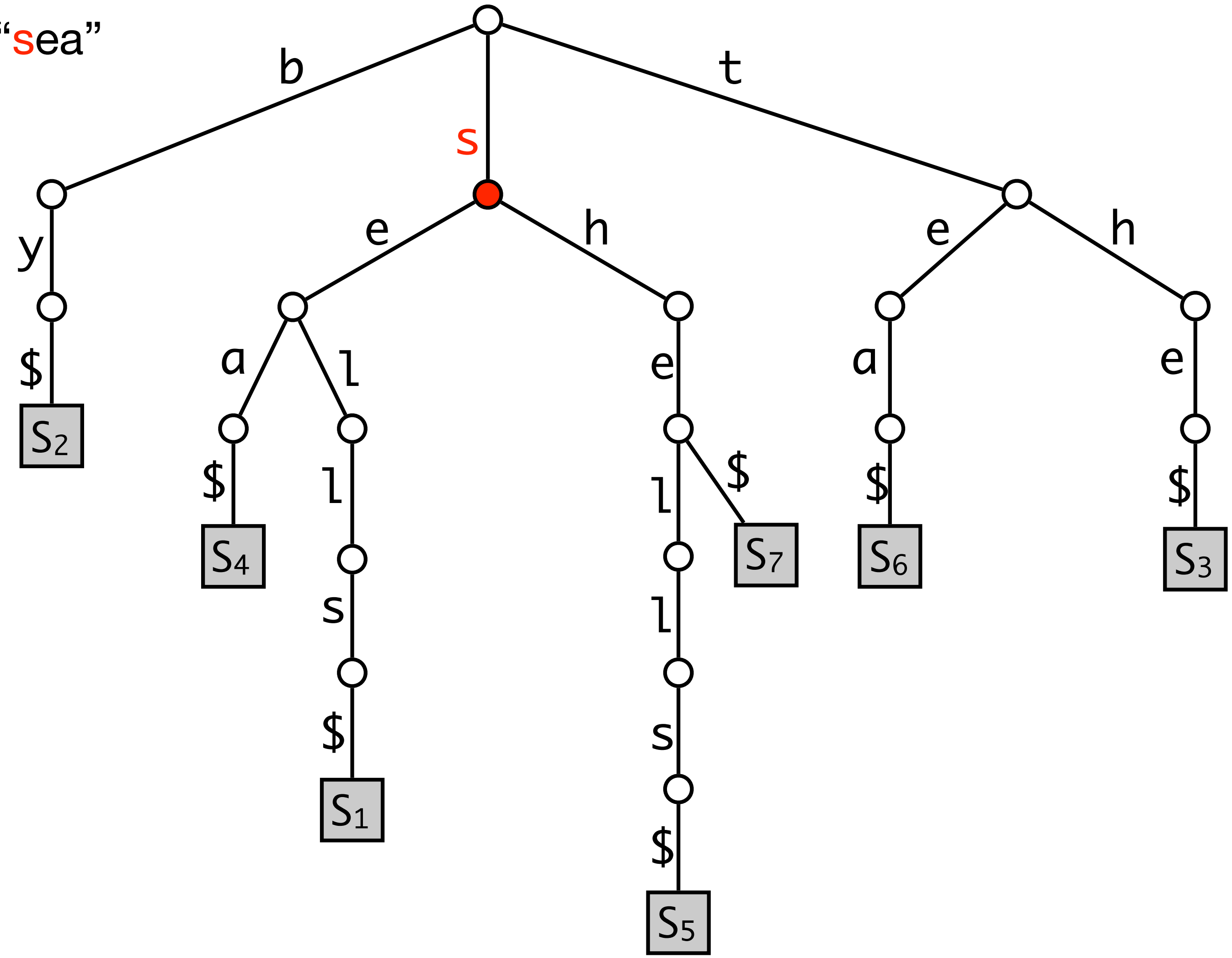
- Text retrieval
- Search for "sea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

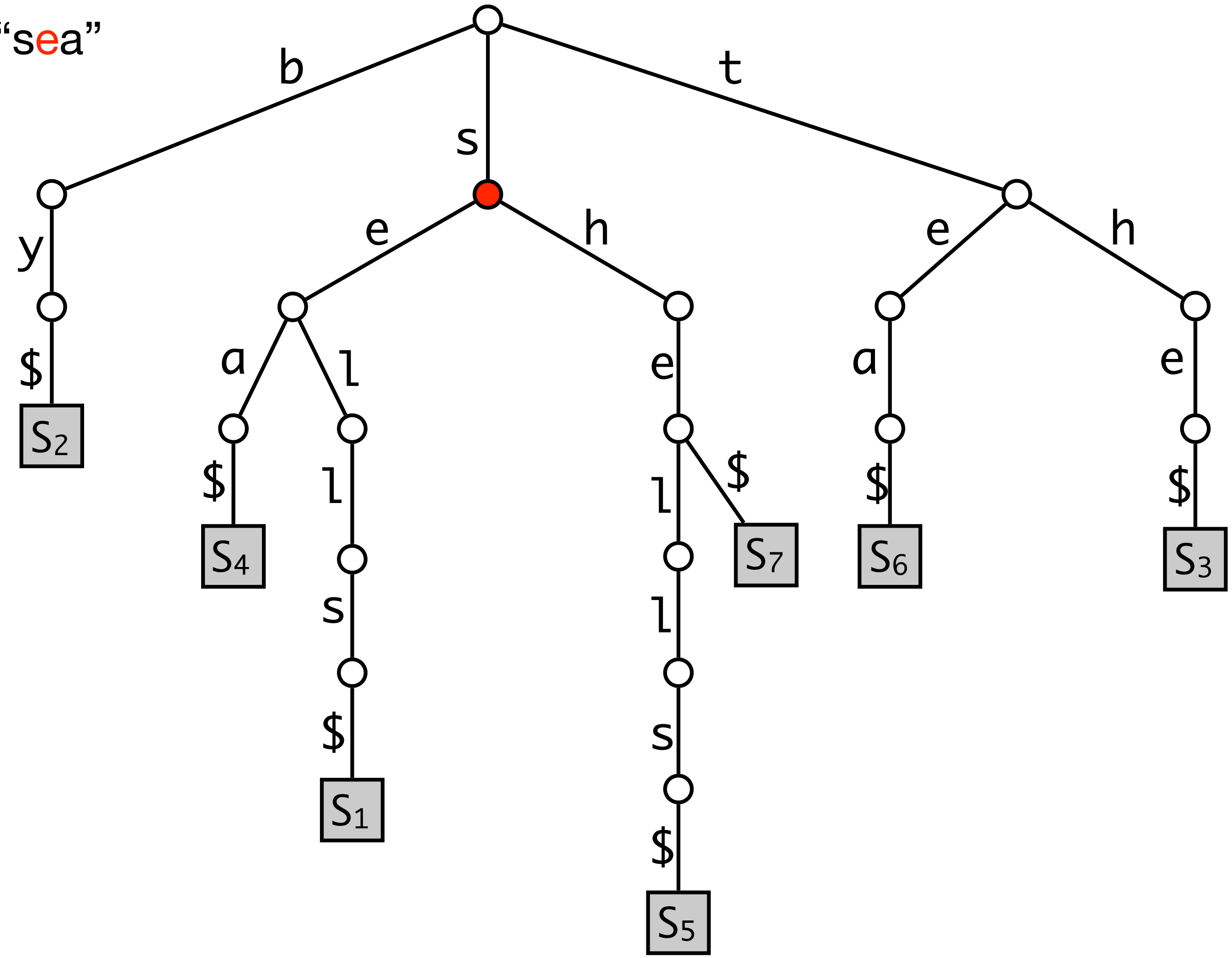
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

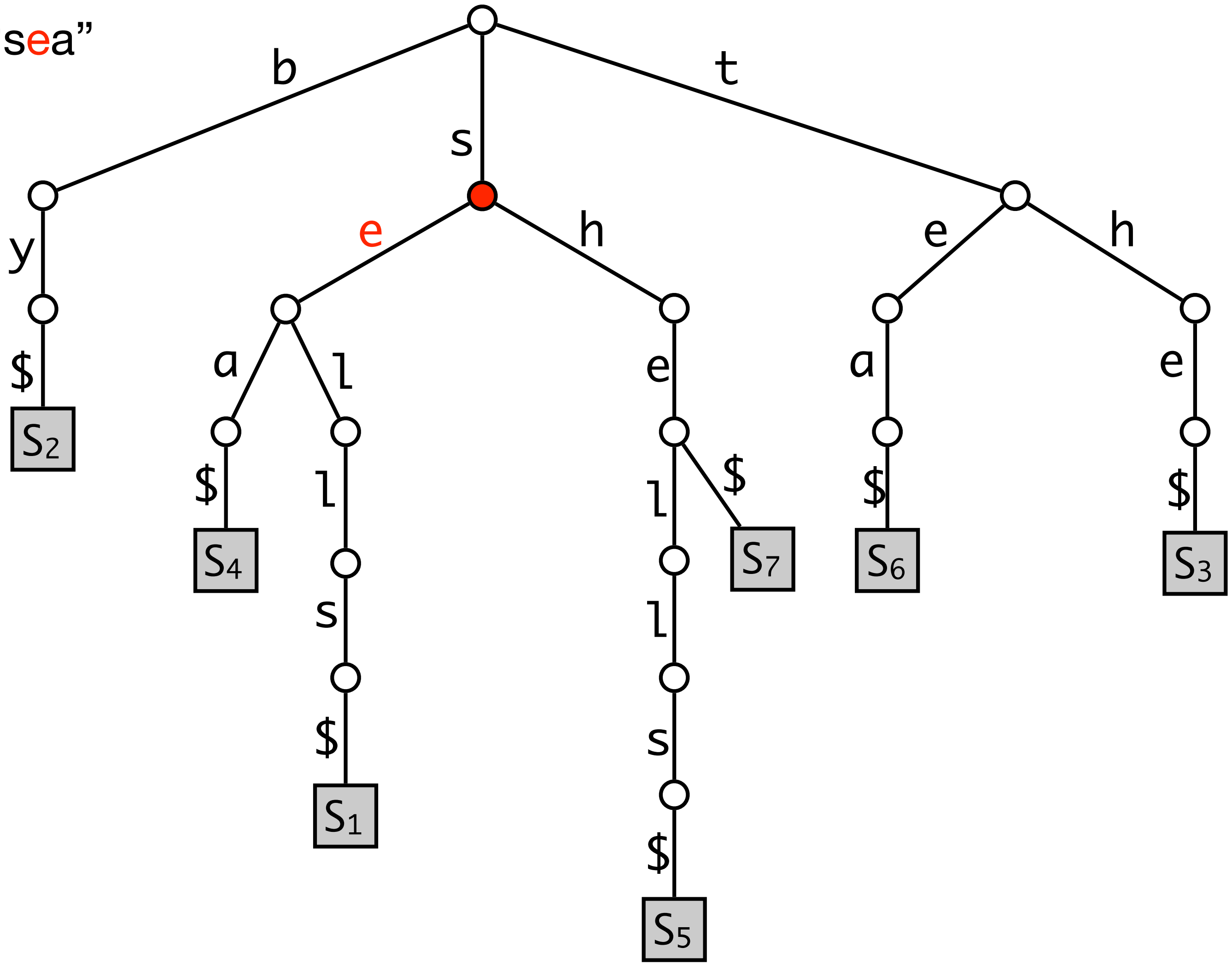
- Text retrieval
- Search for "sea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

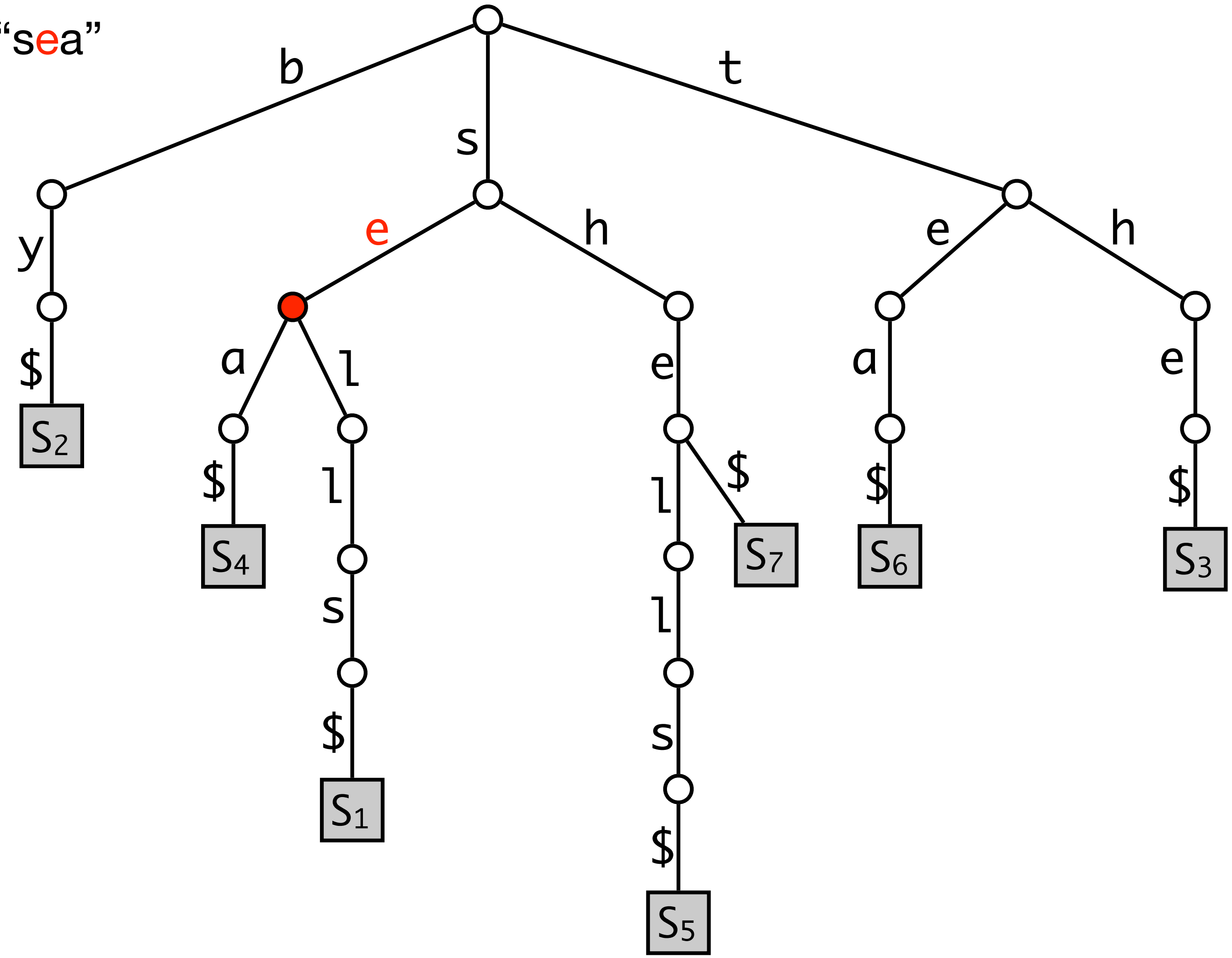
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

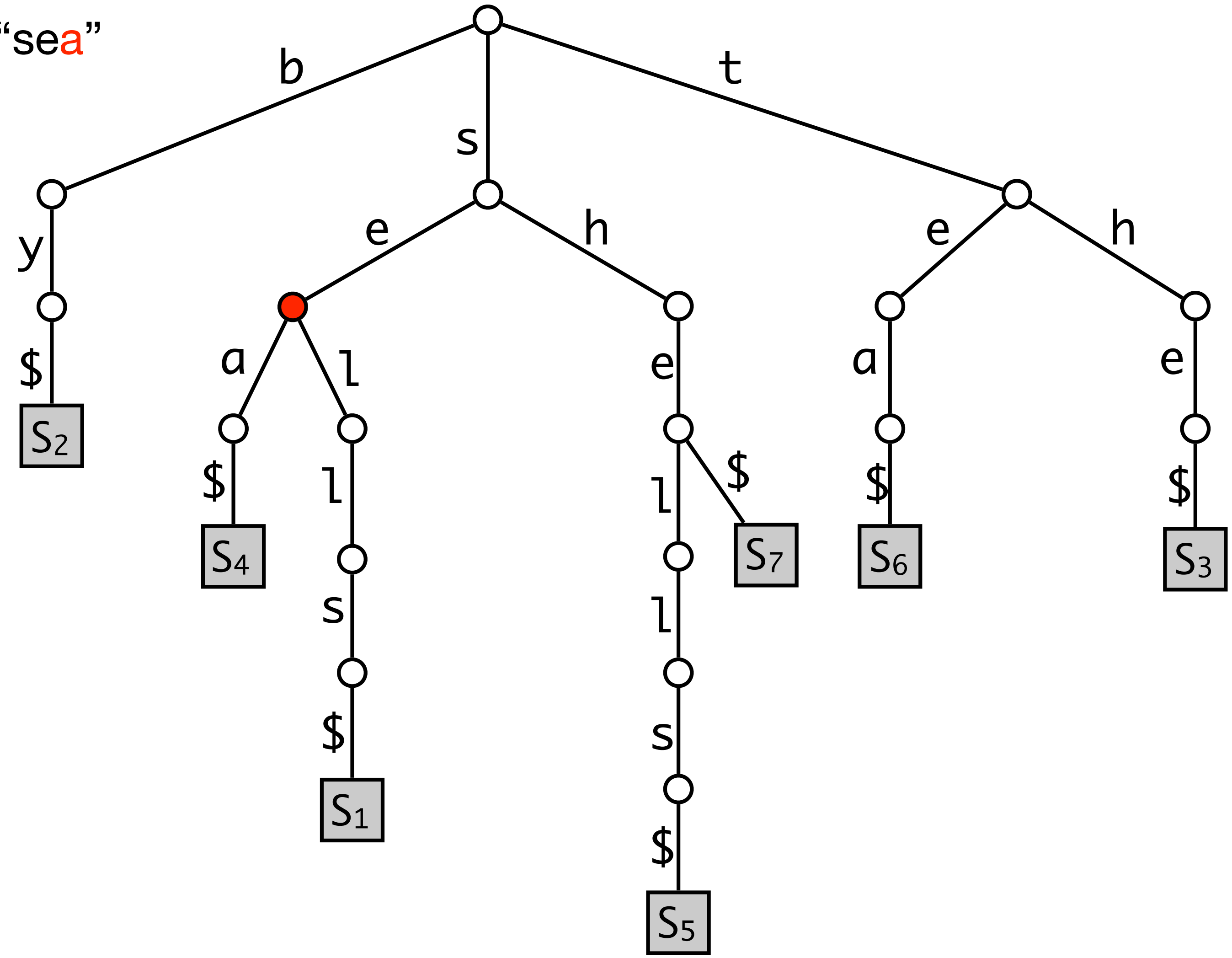
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

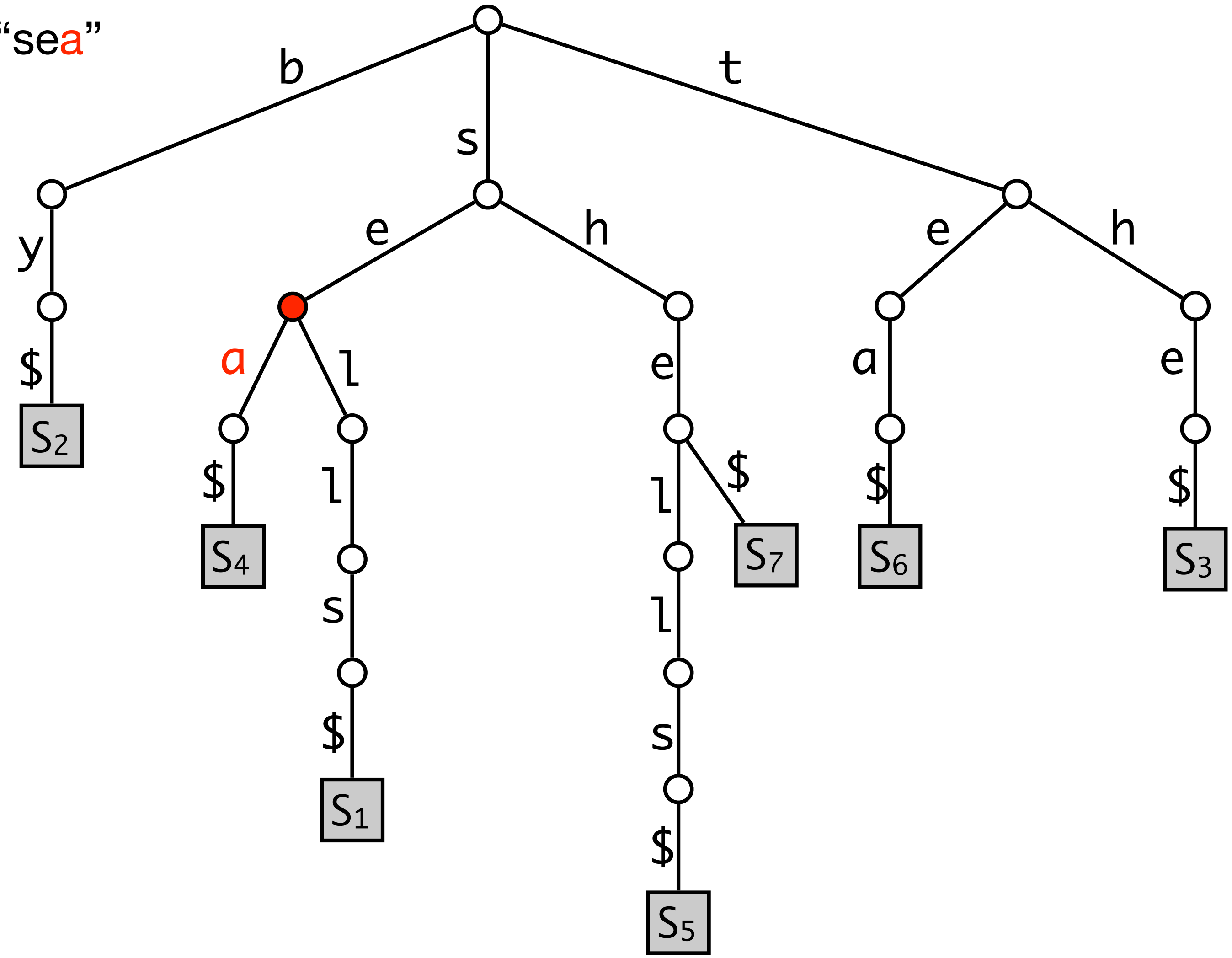
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

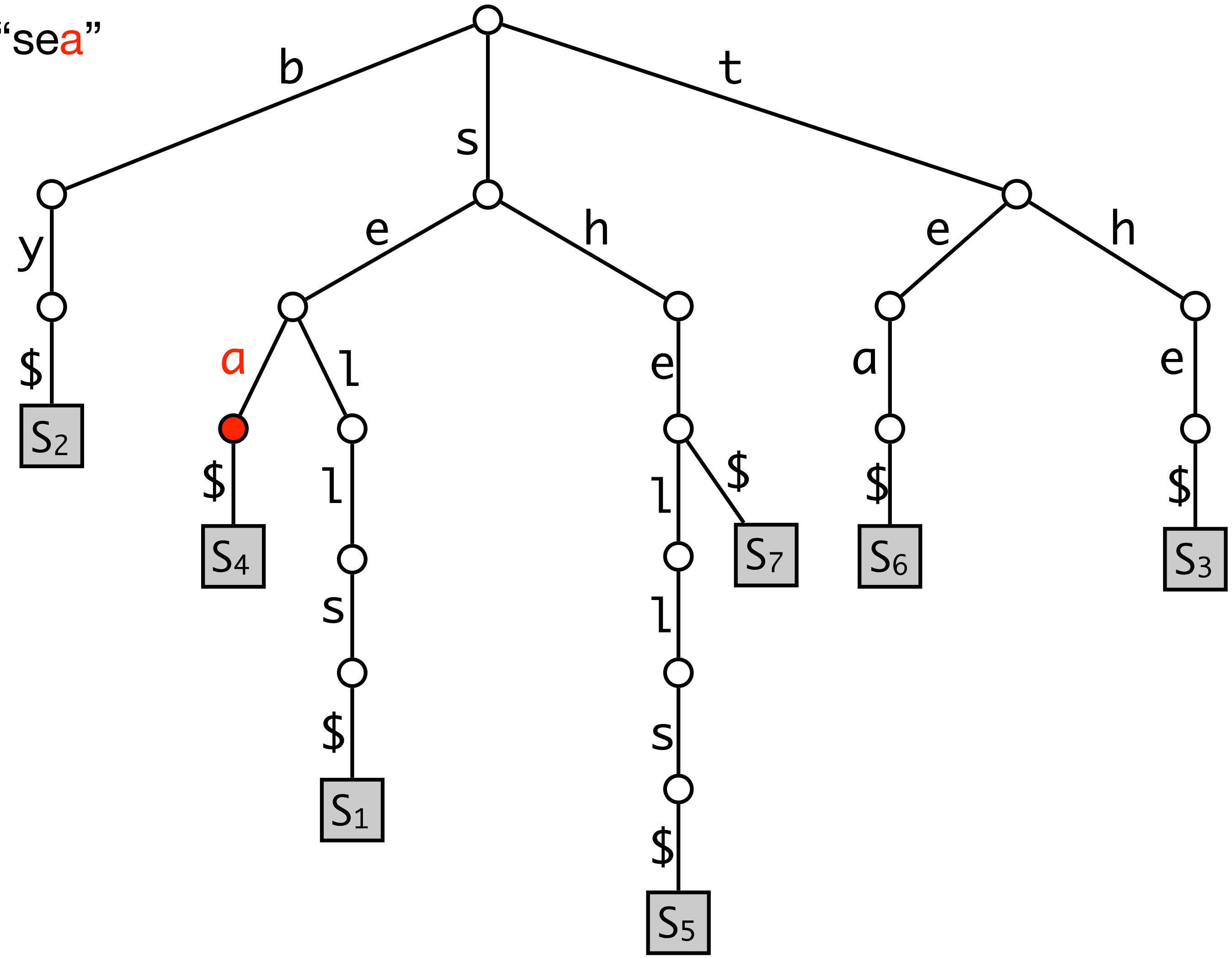
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

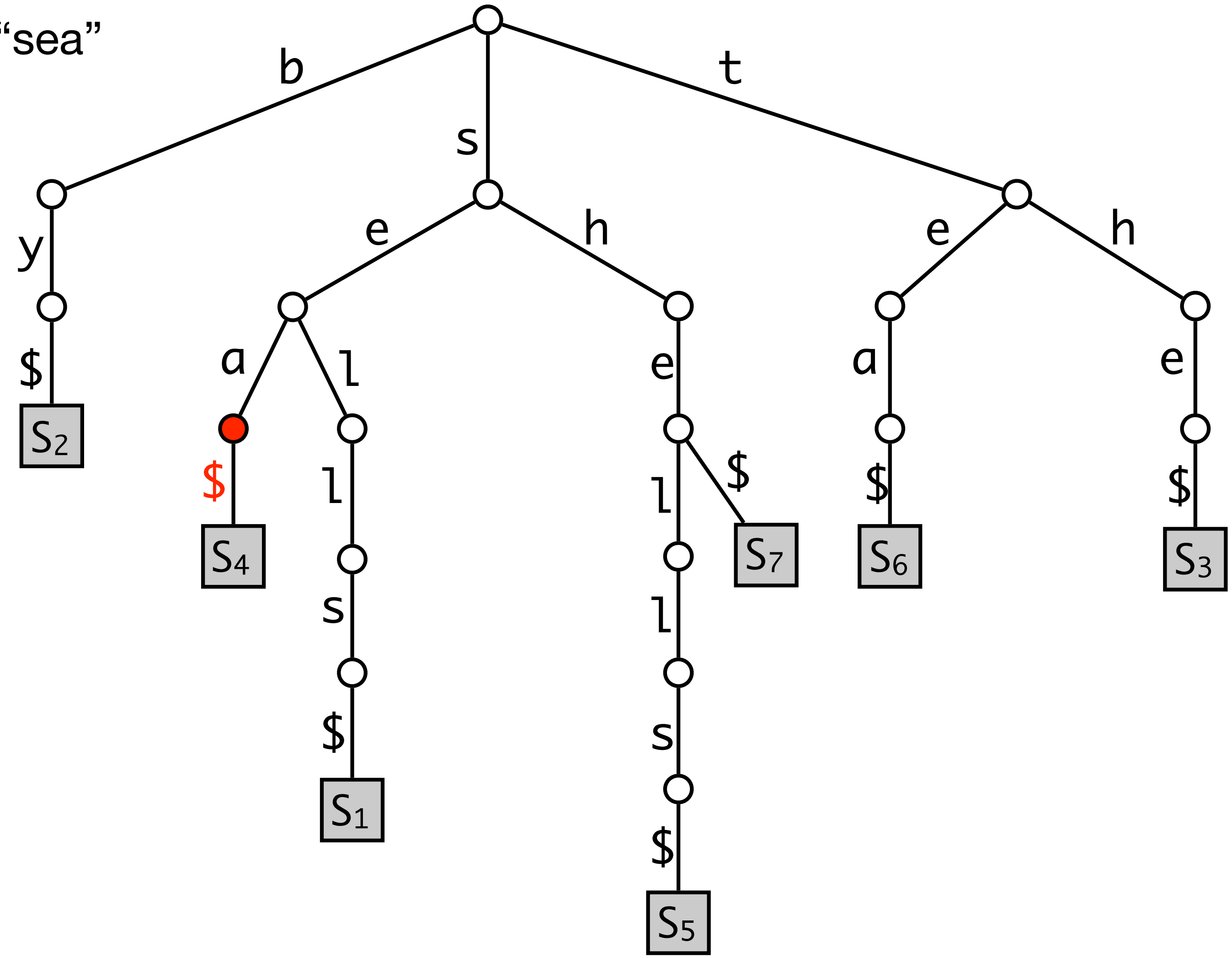
- Text retrieval
- Search for “sea”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

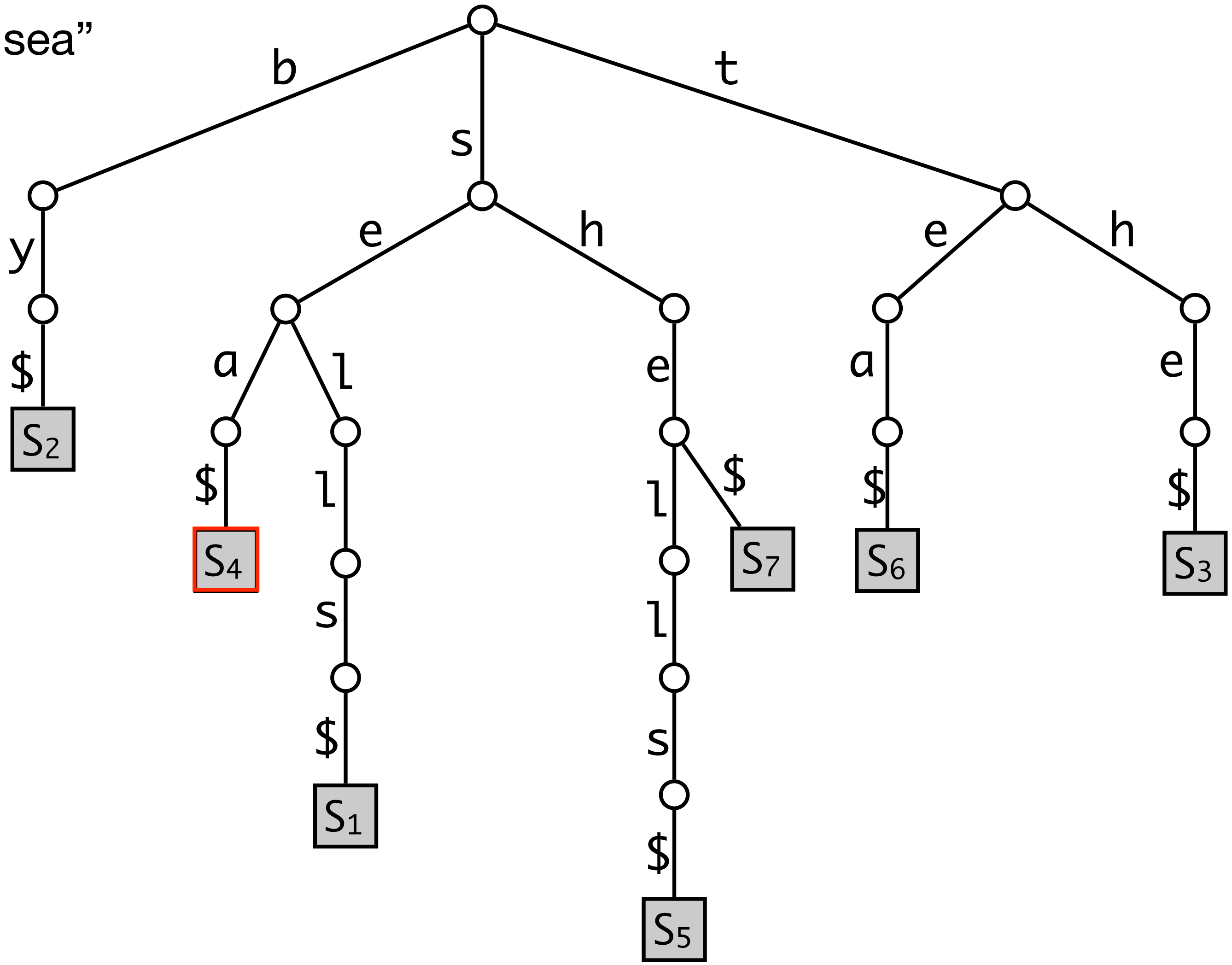
- Text retrieval
- Search for "sea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

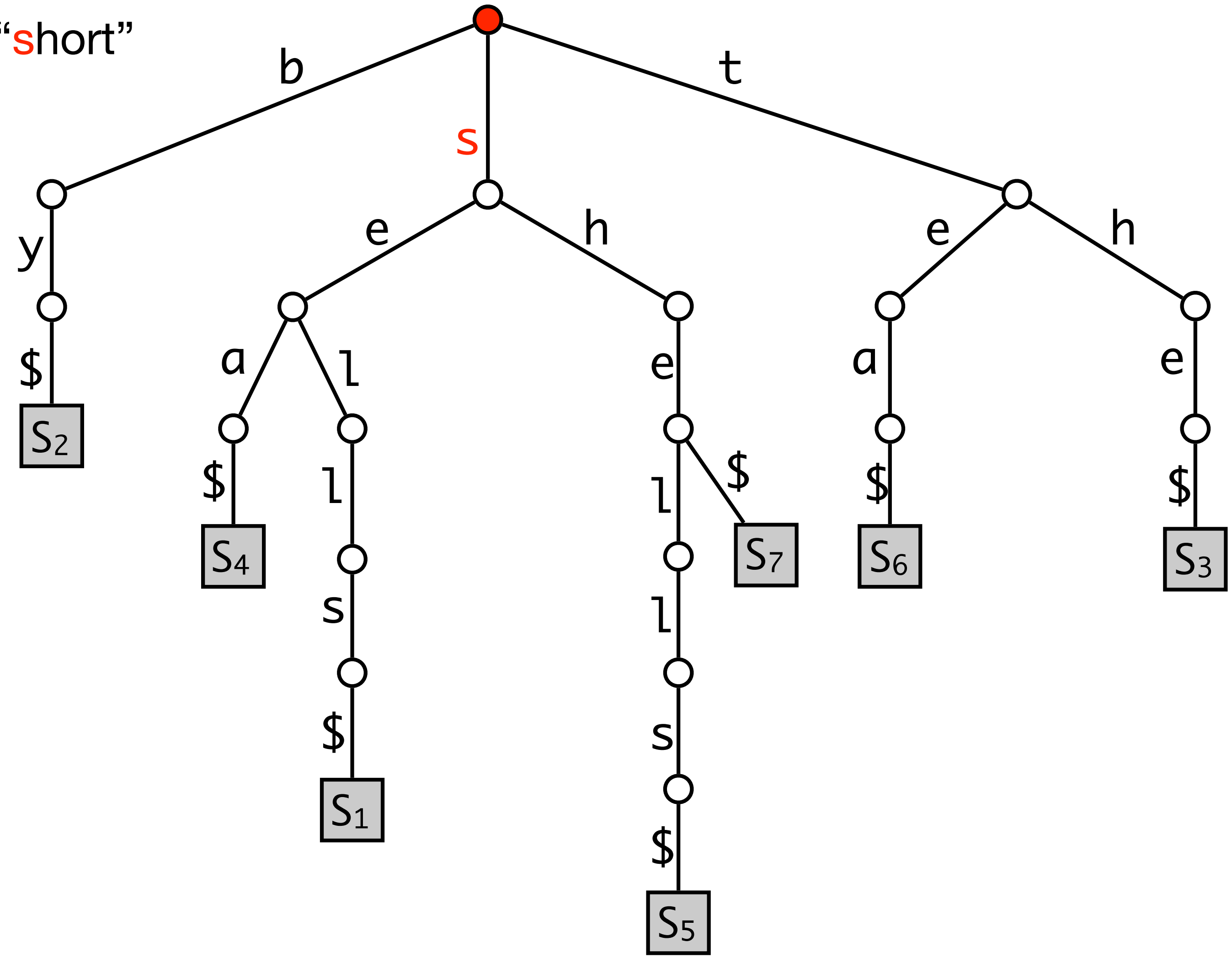
- Text retrieval
- Search for "sea"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

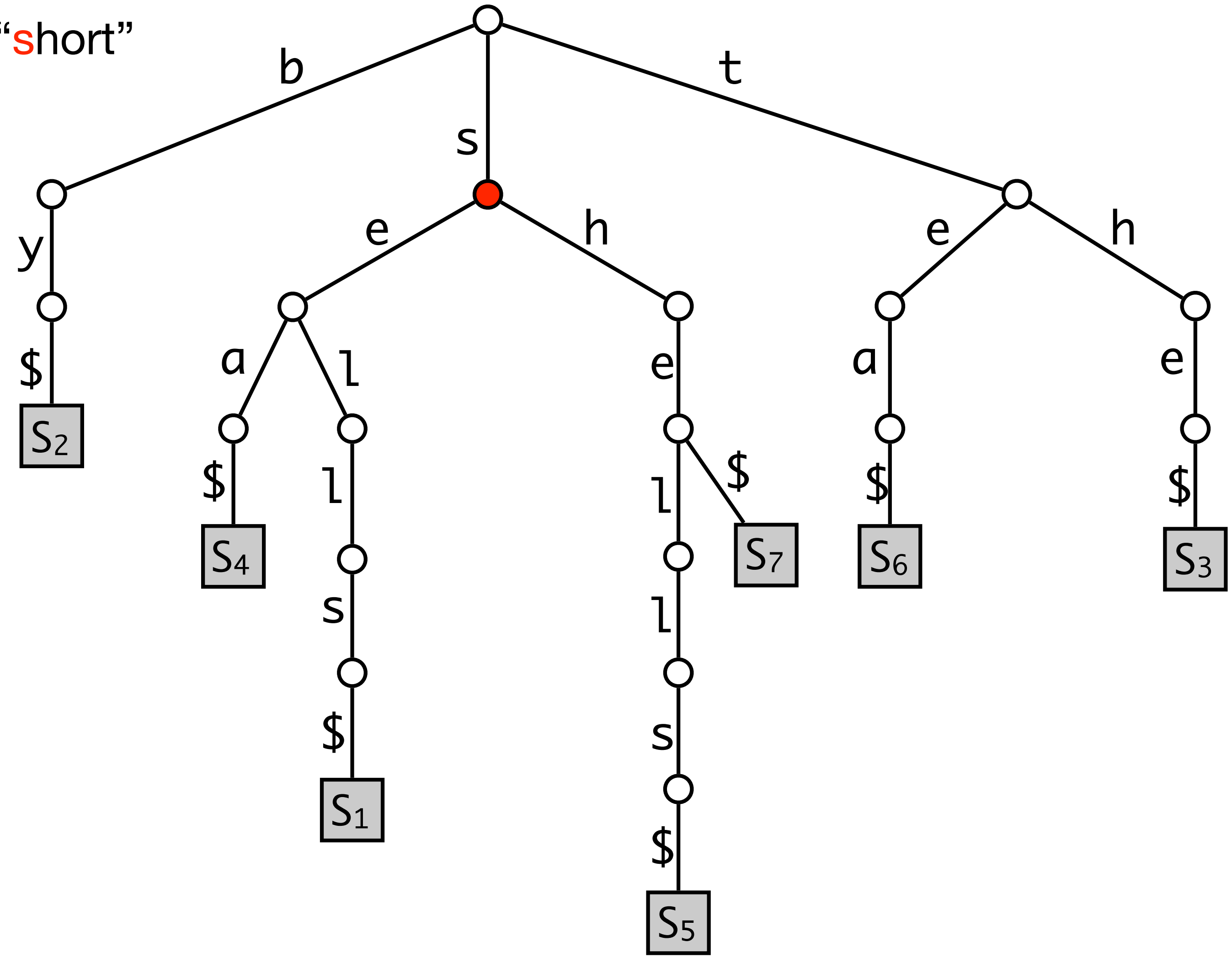
- Text retrieval
- Search for “short”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

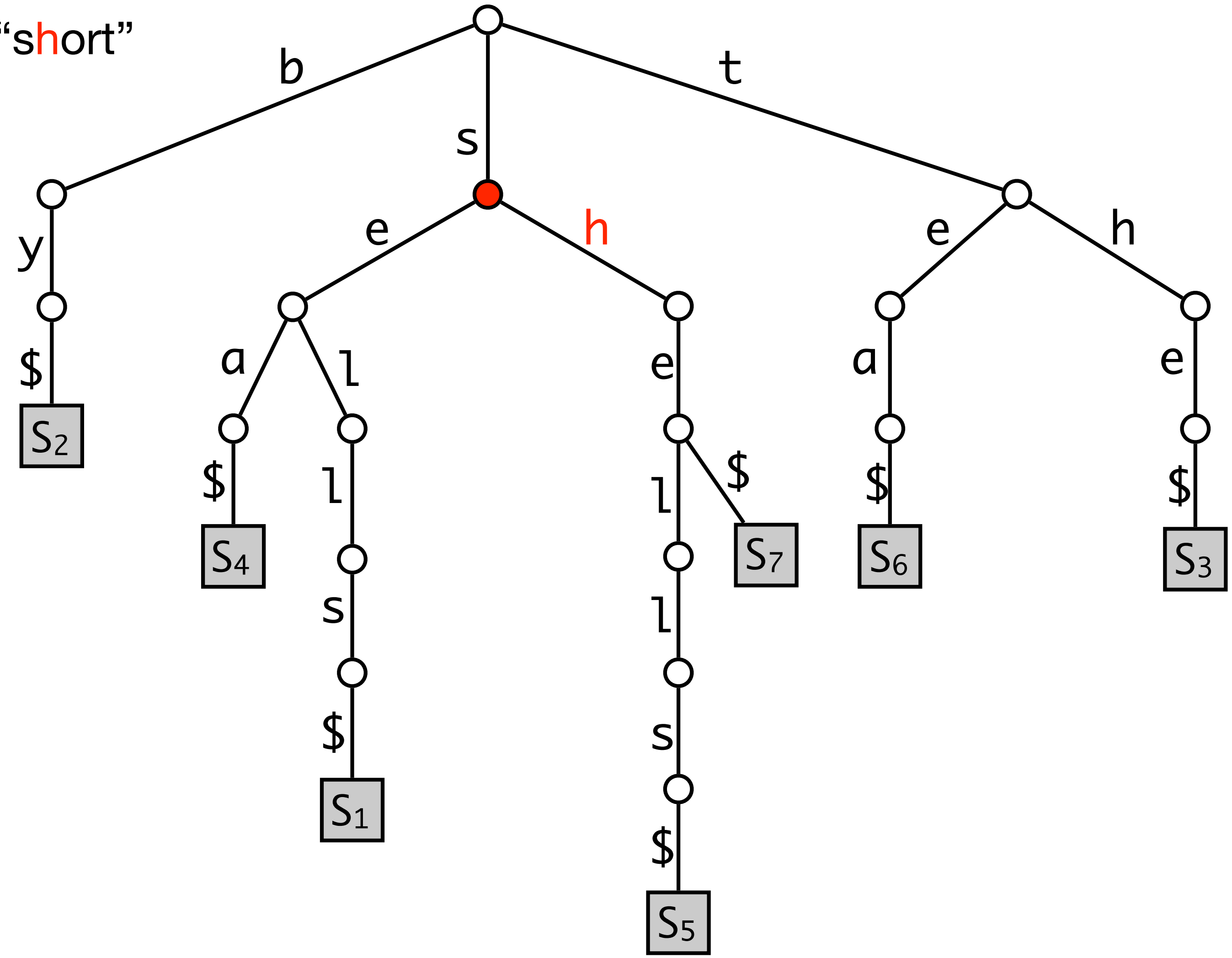
- Text re**trie**val
- Search for "**s**hort"



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

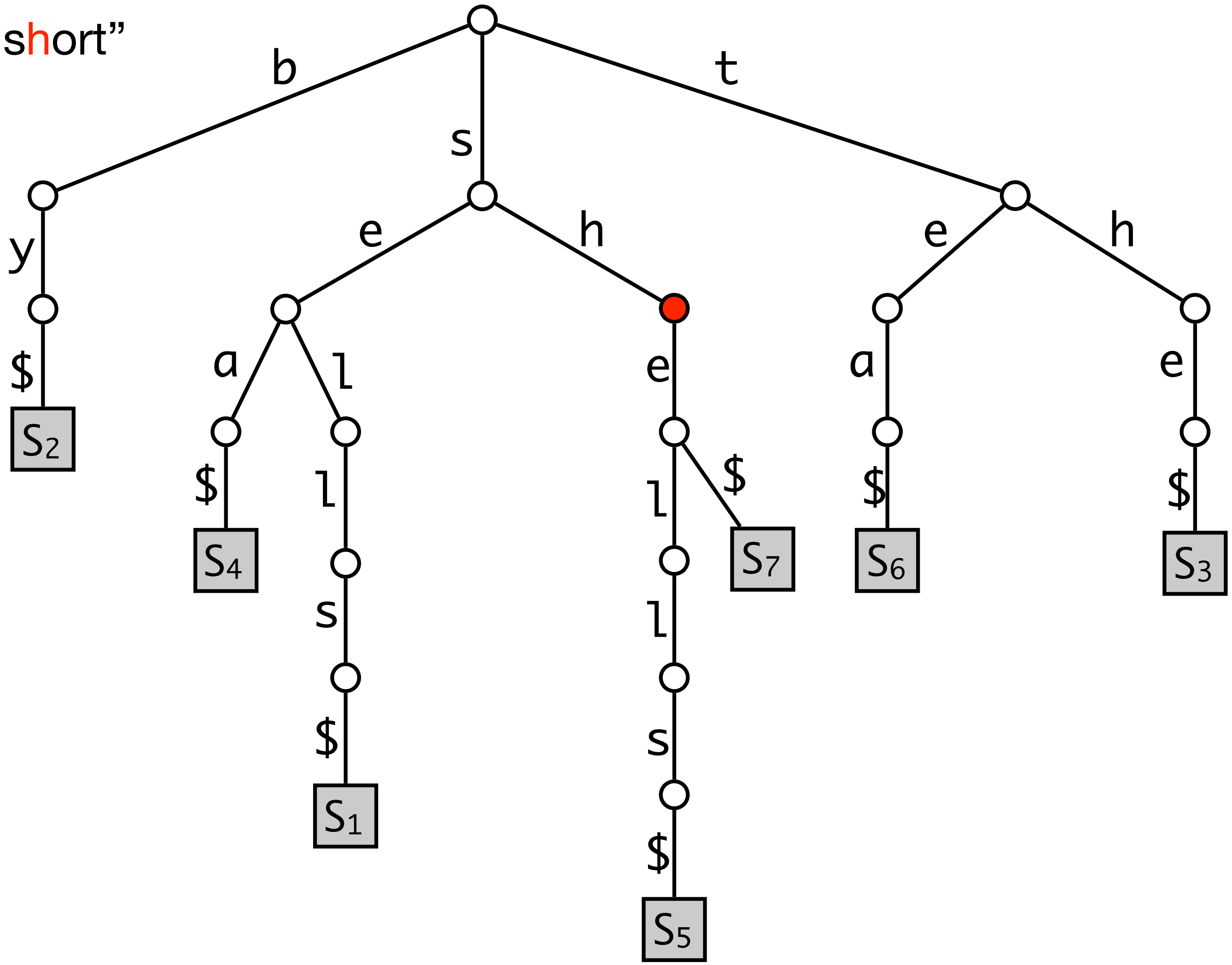
- Text retrieval
- Search for “short”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

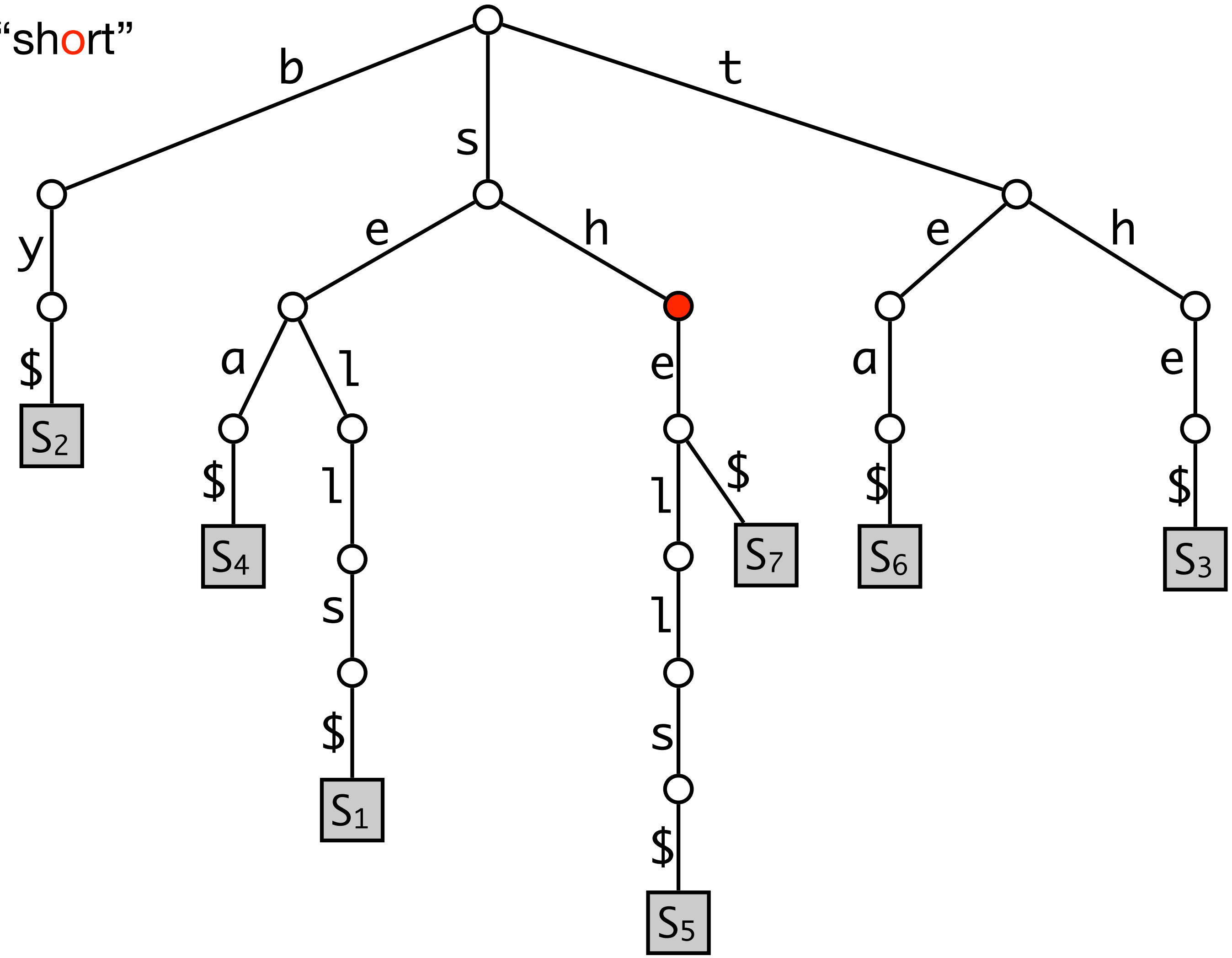
- Text retrieval
- Search for “short”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

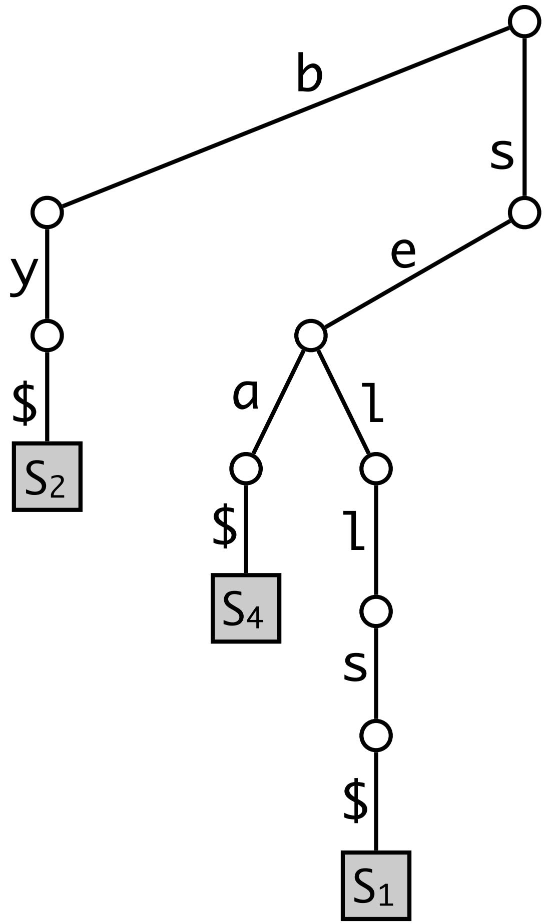
- Text retrieval
- Search for “short”



- Trie over the strings: sells$, by$, the$, sea$, shells$, tea$, she$.

Tries

- Build a trie over the strings: by$, sells$, sea$.



Trie

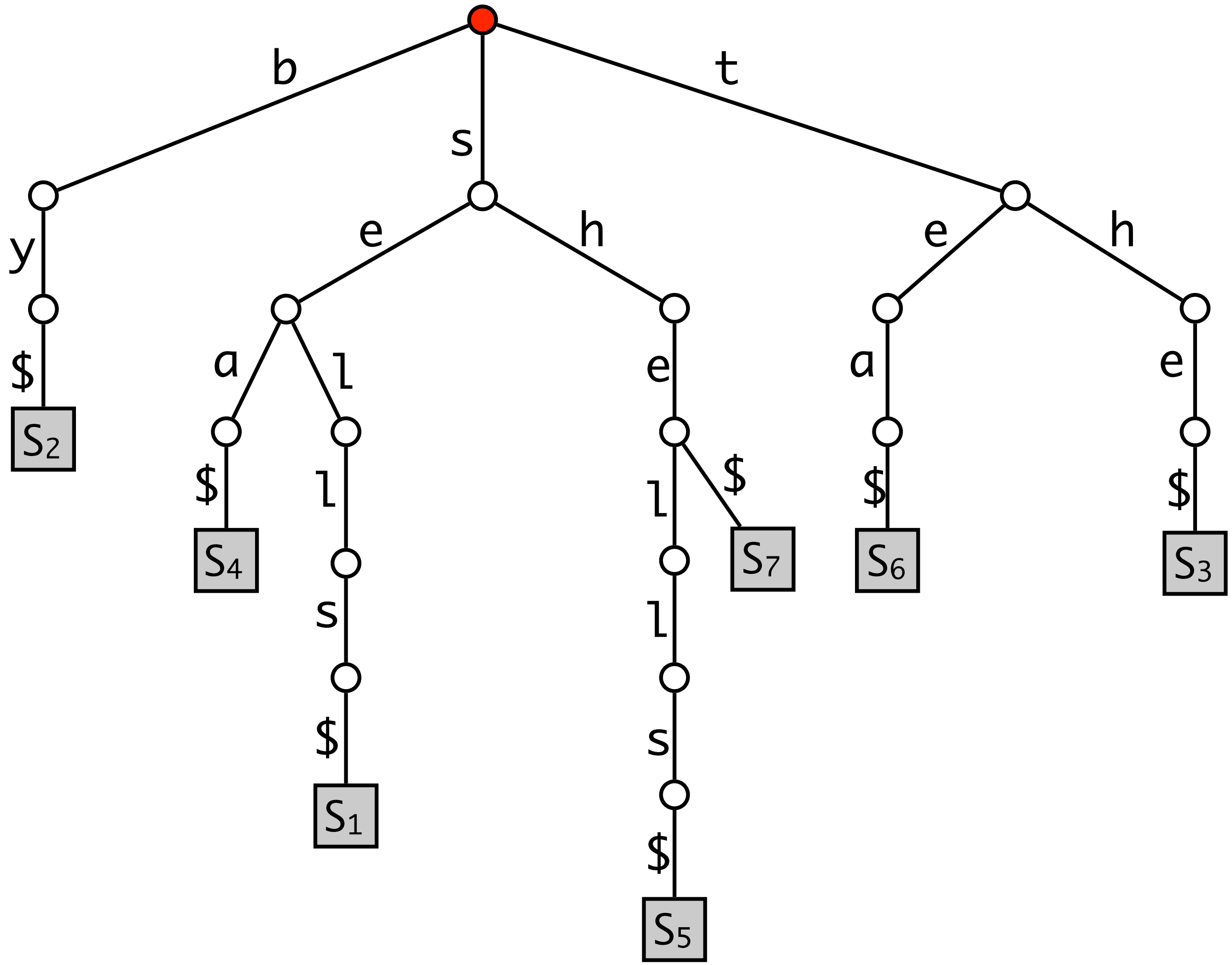
- Properties of the trie. A trie T storing a collection S of s strings of total length *n* from an alphabet of size *d* has the following properties:
    - How many children can a node have?
    - How many leaves does T have?
    - What is the height of T?
    - What is the number of nodes in T?

Trie

- Search time: O(d) in each node => O(dm).
  - O(m) if d constant.
  - d not constant: use dictionary
    - Hashing O(1)
    - Balanced BST: O(log d)
- Time and space for a trie (for small/constant d):
  - O(m) for searching for a string of length m.
  - O(n) space.
  - Preprocessing: O(n)

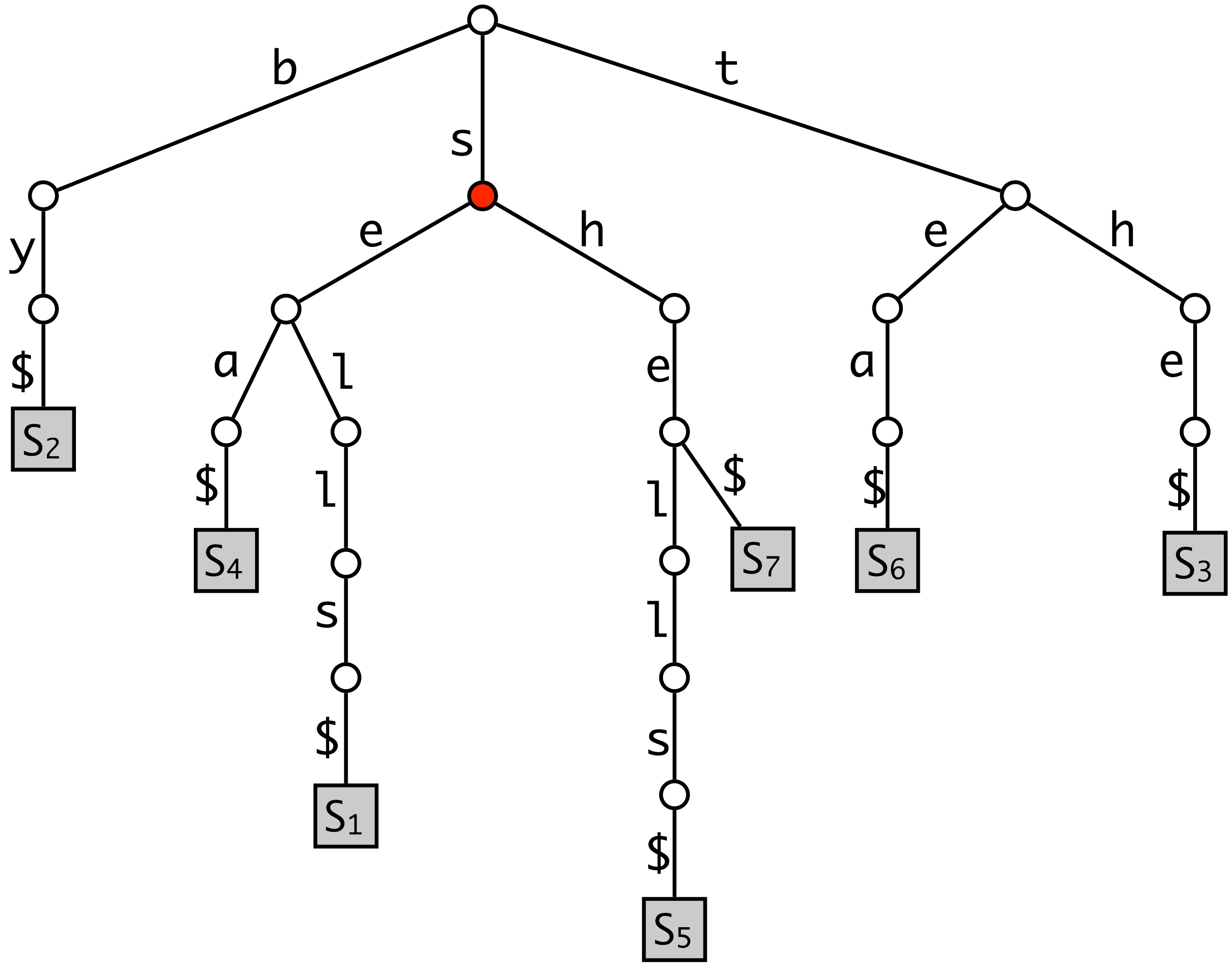
Tries

- Prefix search: return all words in the trie starting with "se"



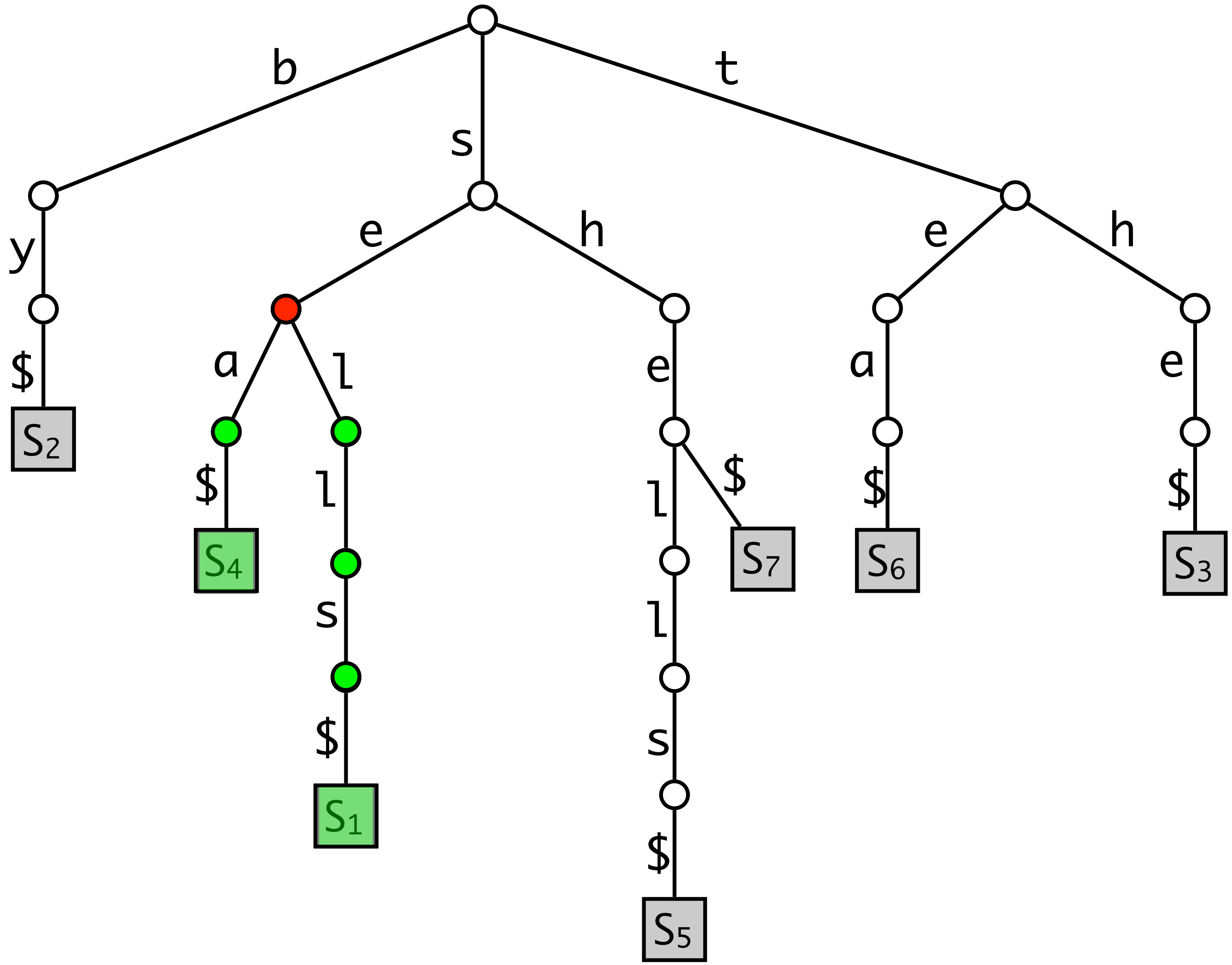
Tries

- Prefix search: return all words in the trie starting with “se”



Tries

- Prefix search: return all words in the trie starting with “se”



Trie

- Time for prefix search: O(m) + time to report all occurrences.

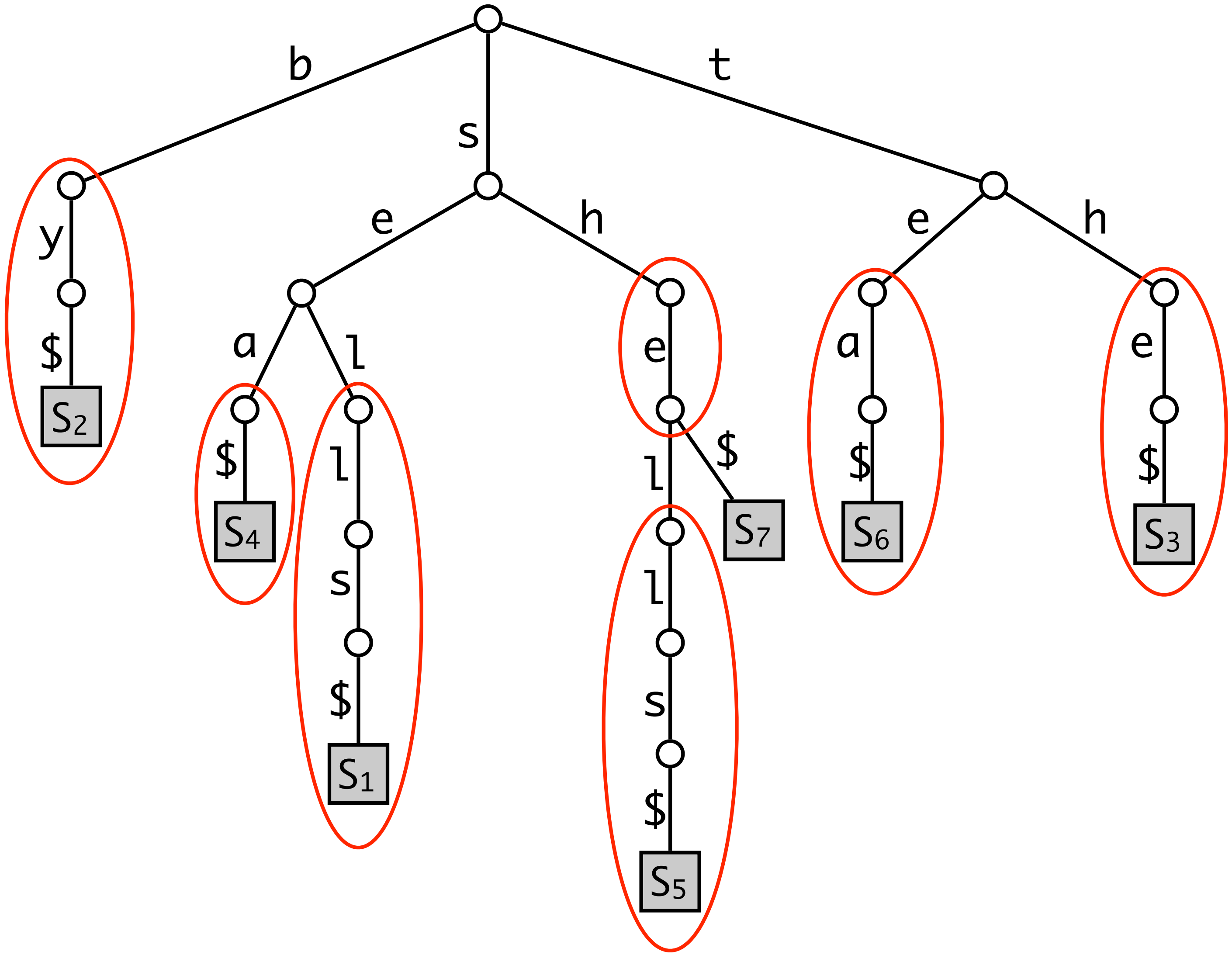
Could be large!!

- Solution: compact tries.

Compact tries

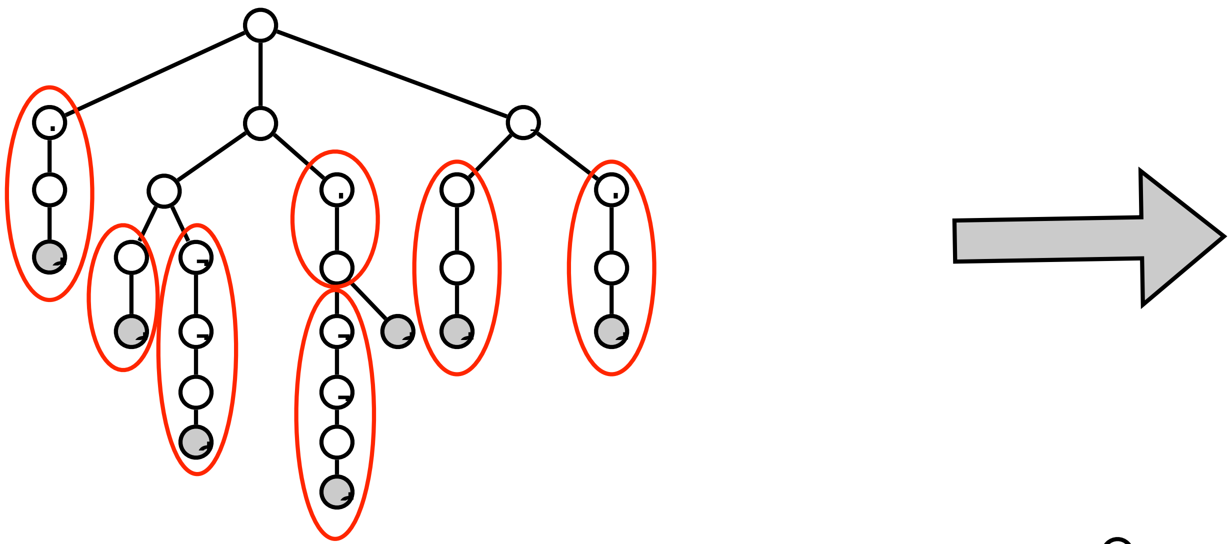
Tries

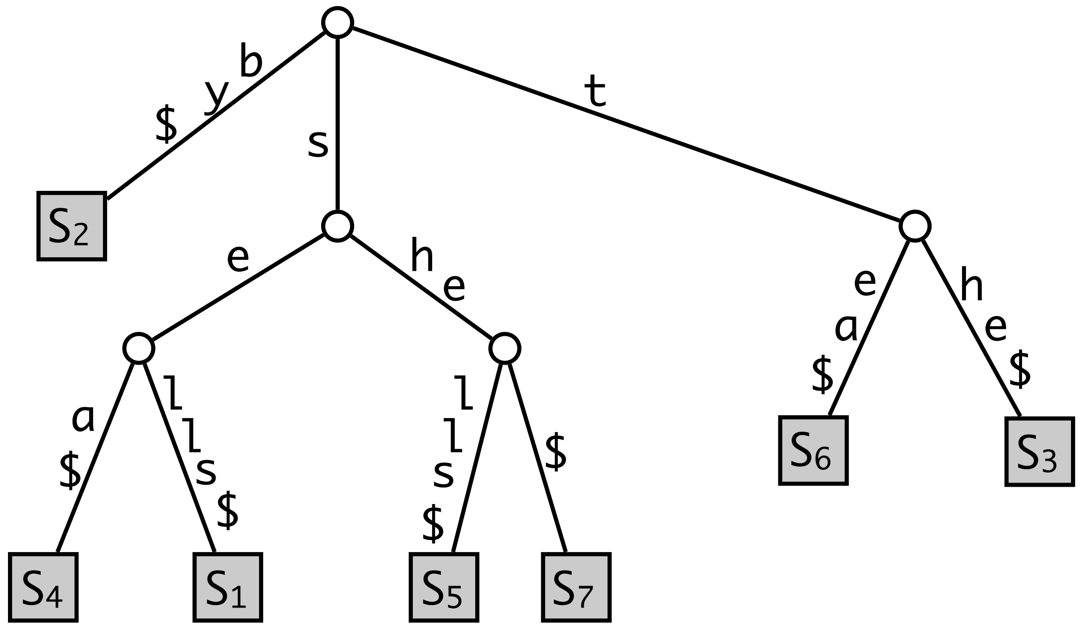
- Compact trie: Chains of nodes with a single child is merged into a single node.



Tries

- Compact trie: Chains of nodes with a single child is merged into a single node.





Trie

- Properties of the compact trie. A compact trie T storing a collection S of s strings of total length *n* from an alphabet of size *d* has the following properties:
  - Every internal node of T has at least 2 and at most *d* children.
  - T has *s* leaves
  - The number of nodes in T is $< 2s$.

- Time and space for a compact trie (constant d):
  - O(m) for searching for a string of length m.
  - O(m + occ) for prefix search, where occ = #occurrences
  - O(s) space.
  - Preprocessing: O(n)

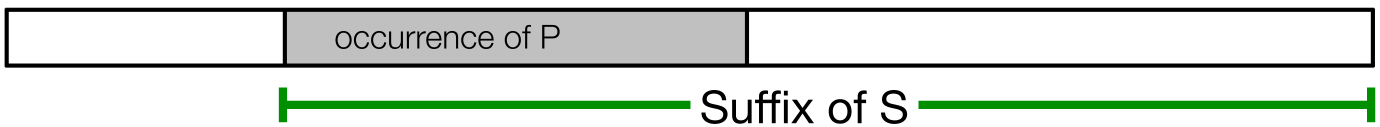
Suffix trees

Suffix tree

- String indexing problem. Given a string S of characters from an alphabet Σ. Preprocess S into a data structure to support
  - Search(P): Return starting position of all occurrences of P in S.

- Build a compressed trie over all suffixes of S (suffix tree). Label leaves with index of suffix.

- Observation: An occurrence of P is a prefix of a suffix of S.

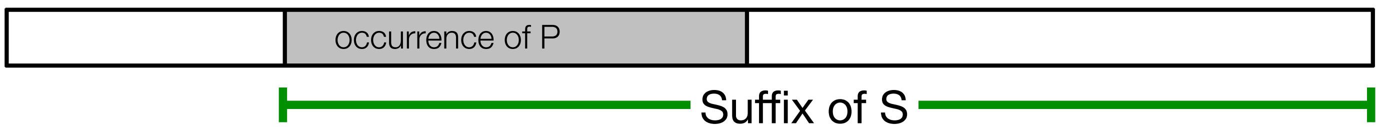


Suffix tree

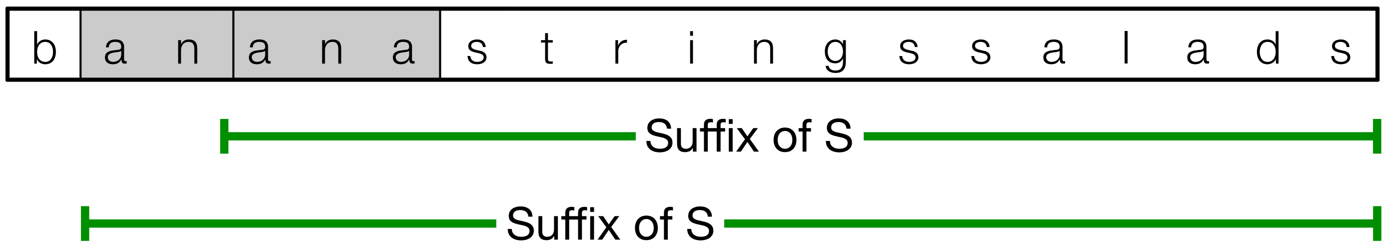
- String indexing problem. Given a string S of characters from an alphabet Σ. Preprocess S into a data structure to support
  - Search(P): Return starting position of all occurrences of P in S.

- Build a compressed trie over all suffixes of S (suffix tree). Label leaves with index of suffix.

- Observation: An occurrence of P is a prefix of a suffix of S.

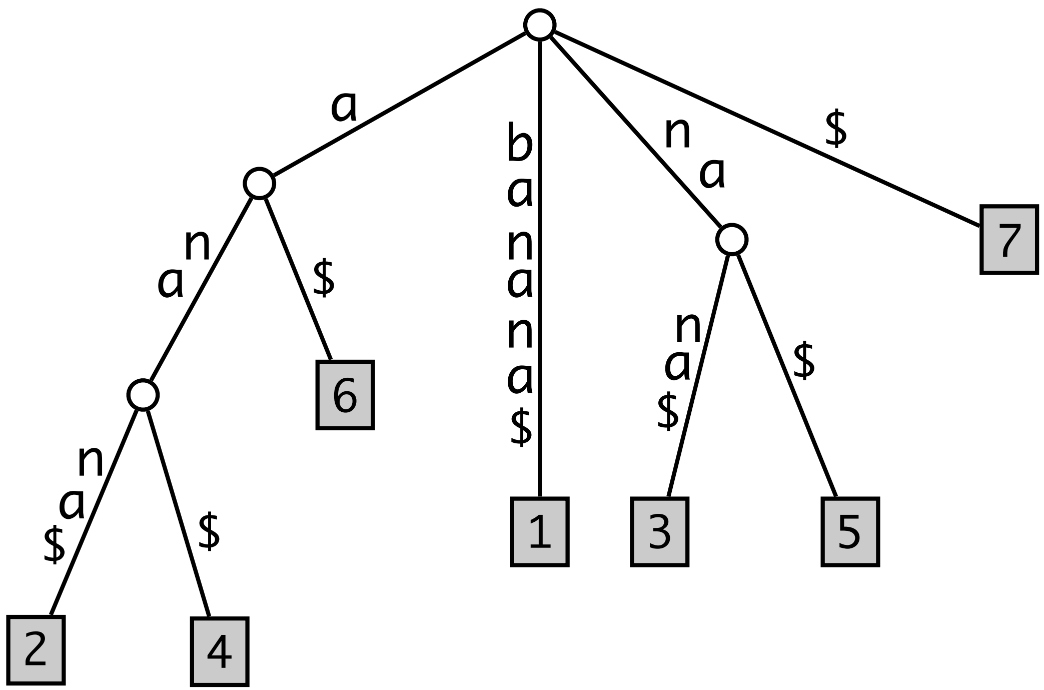


- Example: P = ana.



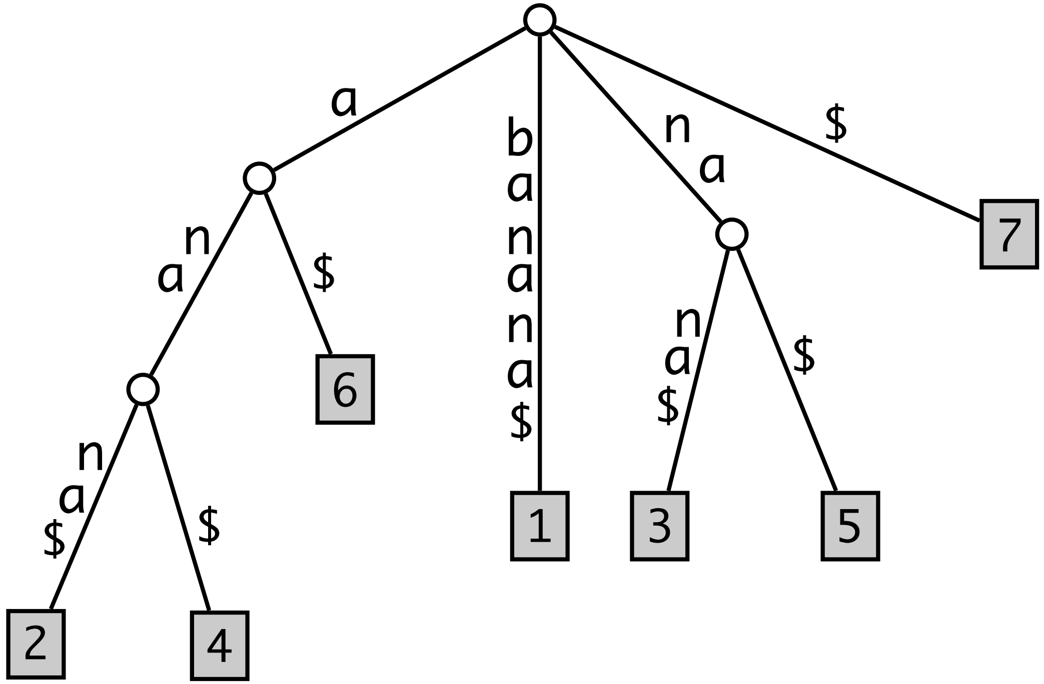
Suffix Tree

- Suffix tree: over the string banana$



Suffix Tree

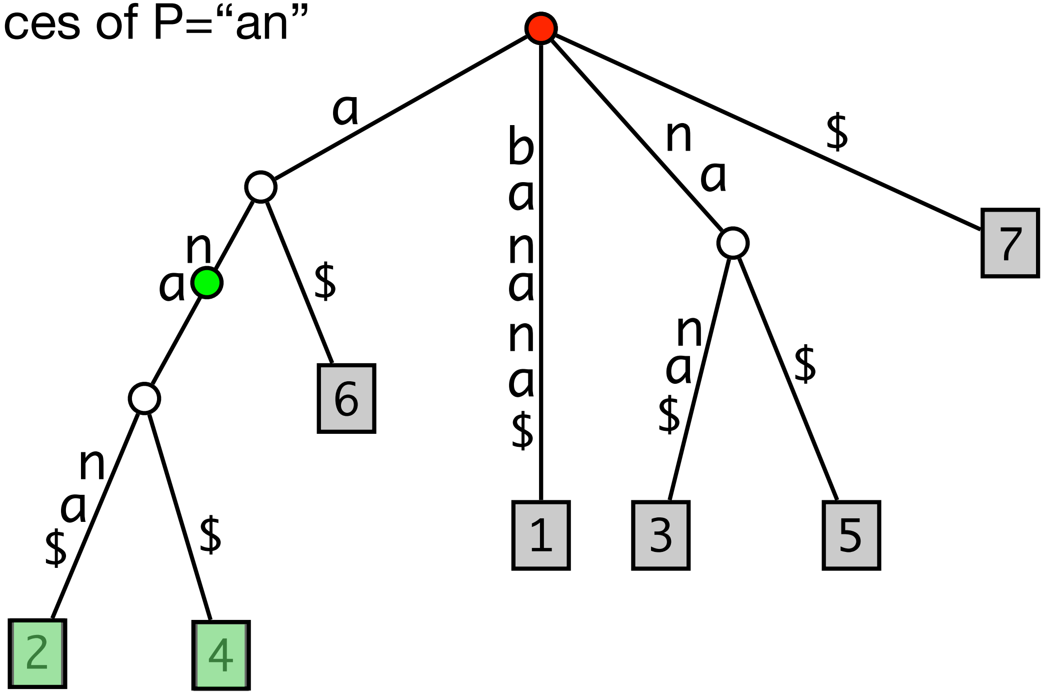
- Suffix tree: over the string banana$



- Search for P.
- Report labels of all leaves below final node

Suffix Tree

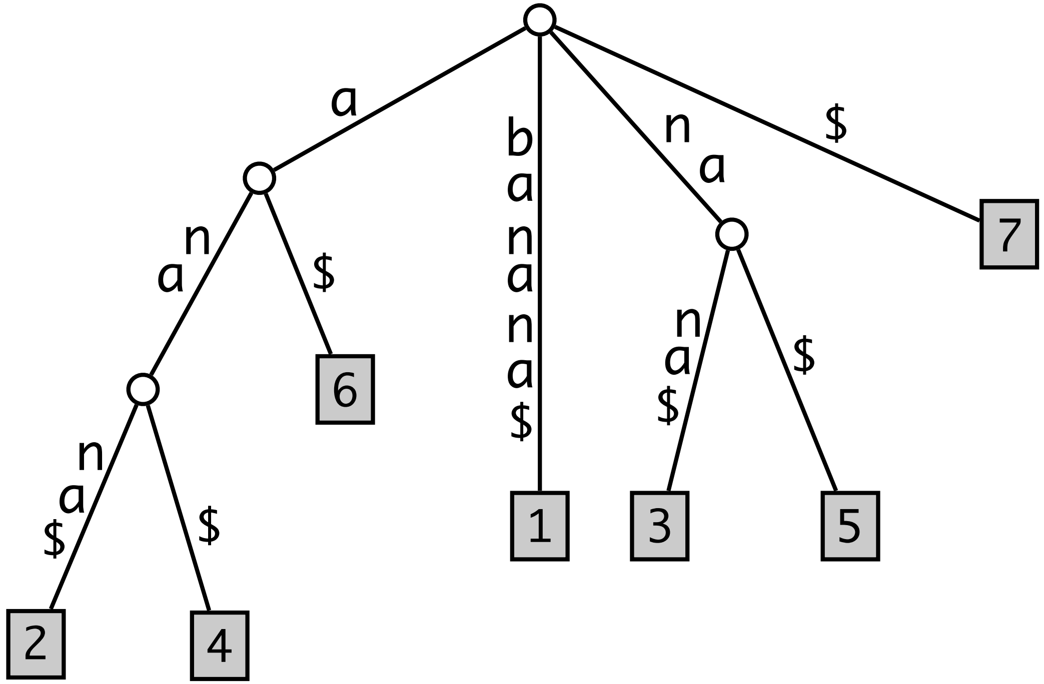
- Suffix tree: over the string banana$
- Find all occurrences of P="an"



- Search for P.
- Report labels of all leaves below final node

Suffix Tree

- Suffix tree: over the string banana$

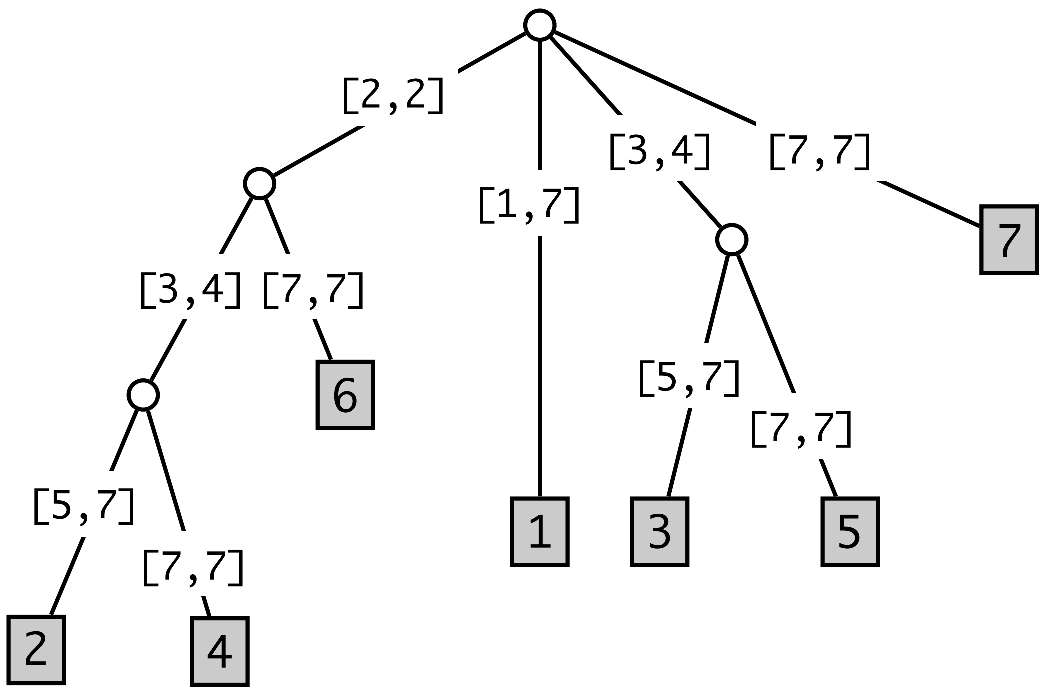


- Store S and store node labels by reference to S.

1	2	3	4	5	6	7
b	a	n	a	n	a	$

Suffix Tree

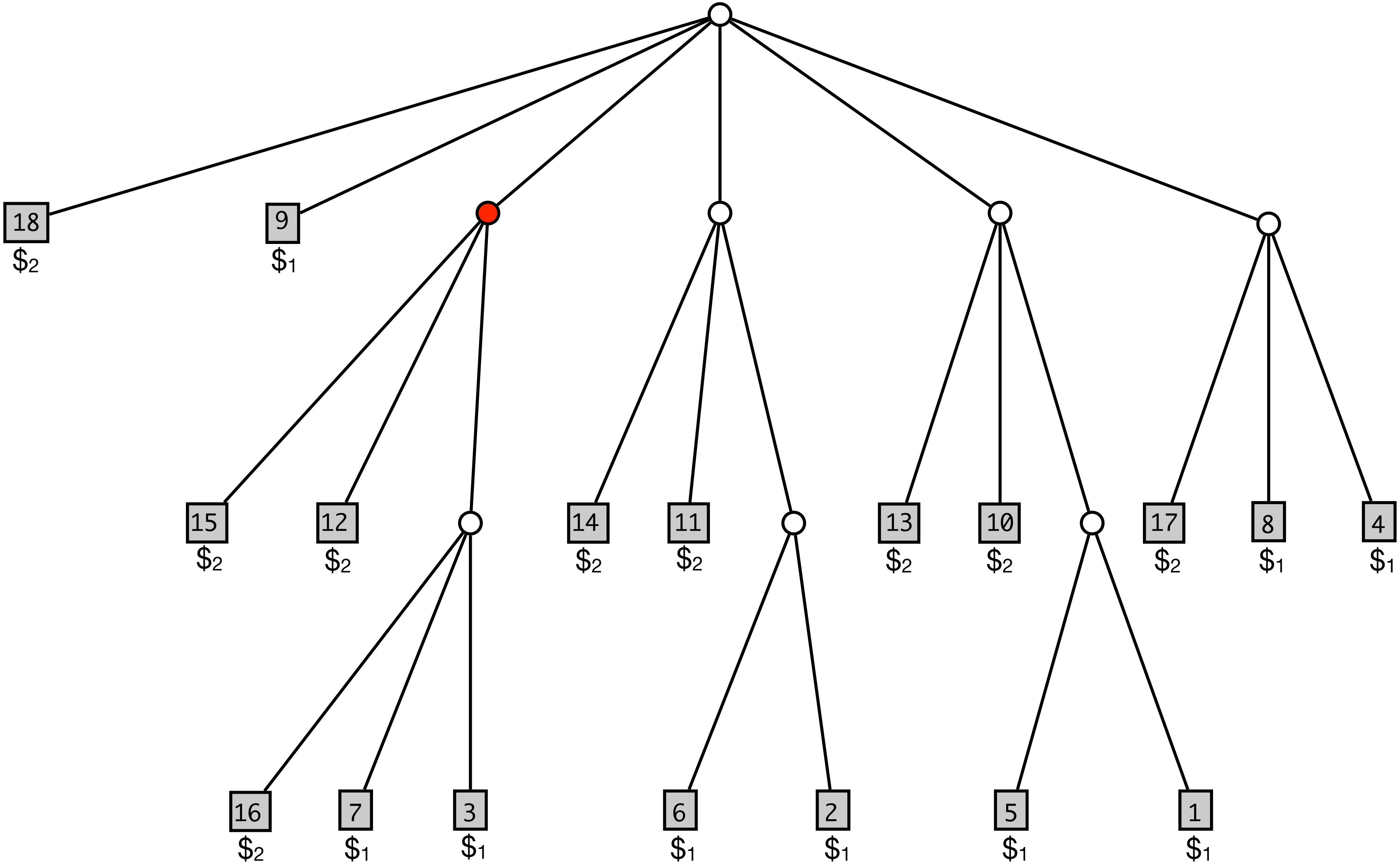
- Suffix tree: over the string banana$



- Store S and store node labels by reference to S.

1	2	3	4	5	6	7
b	a	n	a	n	a	$

Suffix trees and common substrings



Suffix tree

- Suffix tree of a string S: Compact trie over all suffixes of S.

- Space and time:
    - Space: O(n)
    - Search time: O(m) + time to report all occurrences = O(m+occ)
    - Preprocessing: Can be done in O(sort(n,|Σ|)) time, where sort(n,|Σ|) is the time it takes to sort n characters from an alphabet Σ.

- Suffix trees can be used to solve the String indexing problem in:
    - Space: O(n)
    - Search time: O(m+occ)
    - Preprocessing: O(sort(n,|Σ|)) time

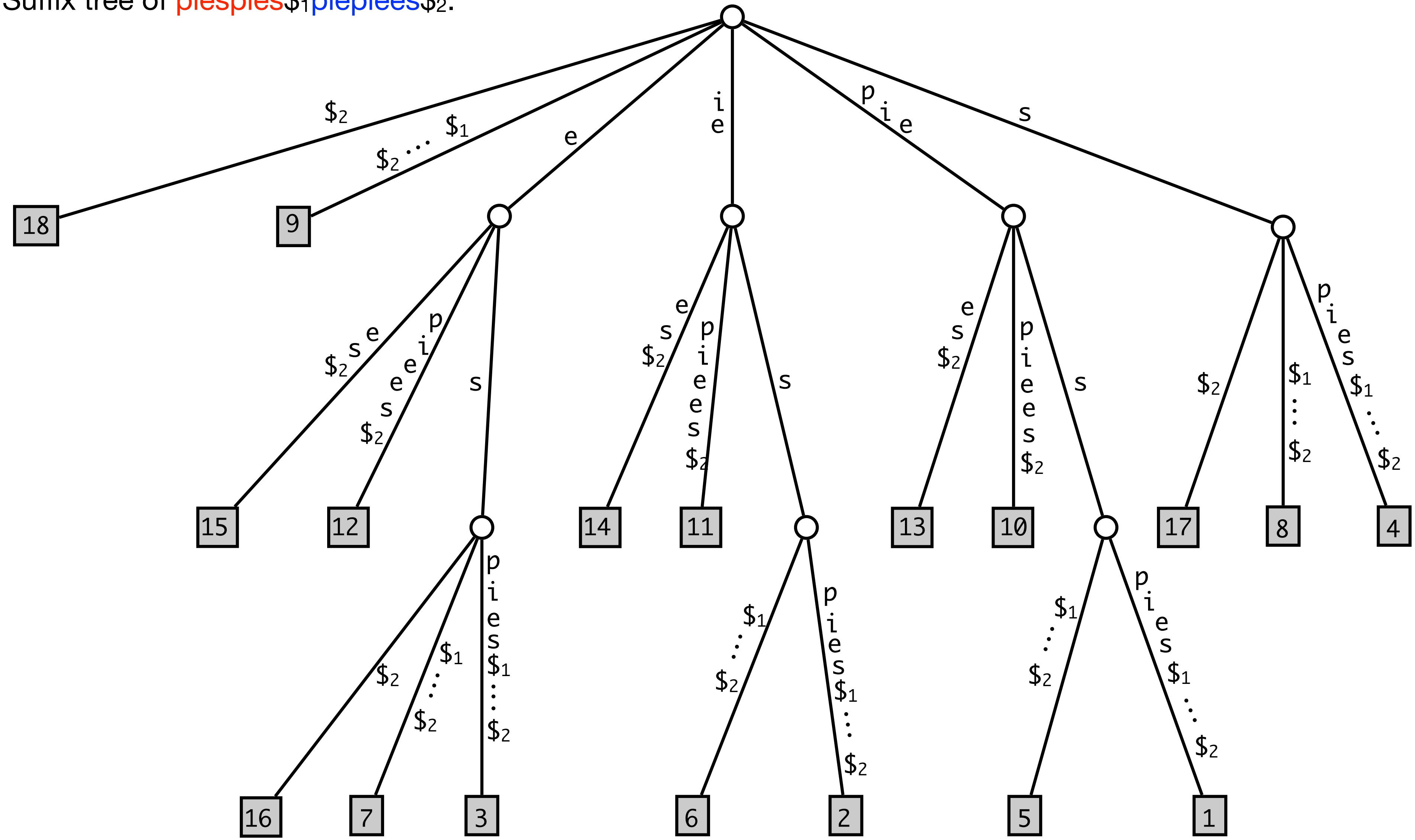
Applications of suffix trees

Longest common substring

- Find longest common substring of strings S_1 and S_2.
- Construct the suffix tree over $S_1\$_1S_2\$_2$.
- Example: Find longest common substring of piespies and piepiees:
    - Construct suffix tree of piespies$\$_1$piepiees$\$_2$.

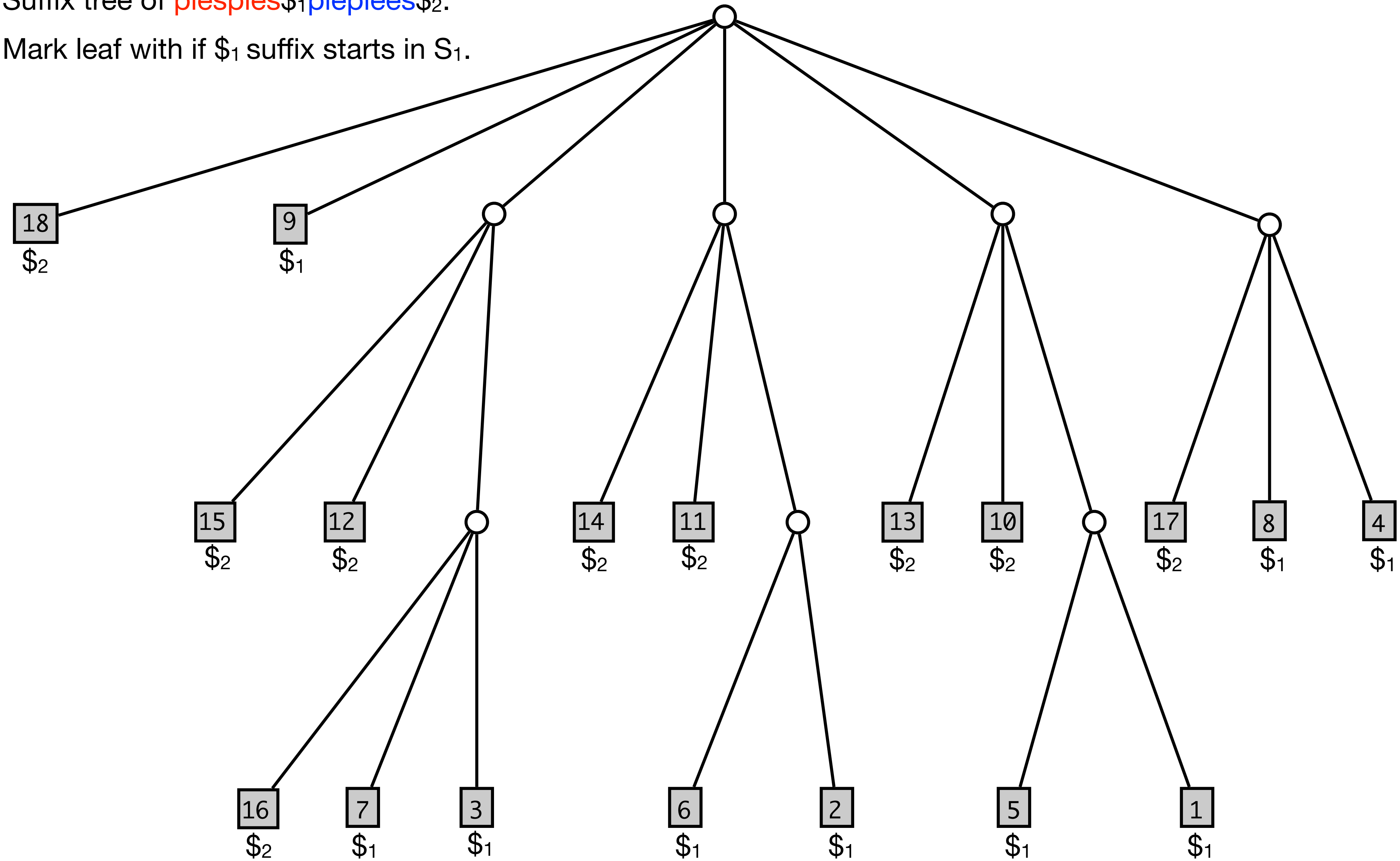
Generalized suffix tree

- Suffix tree of piespies$\$_1$piepiees$\$_2$.



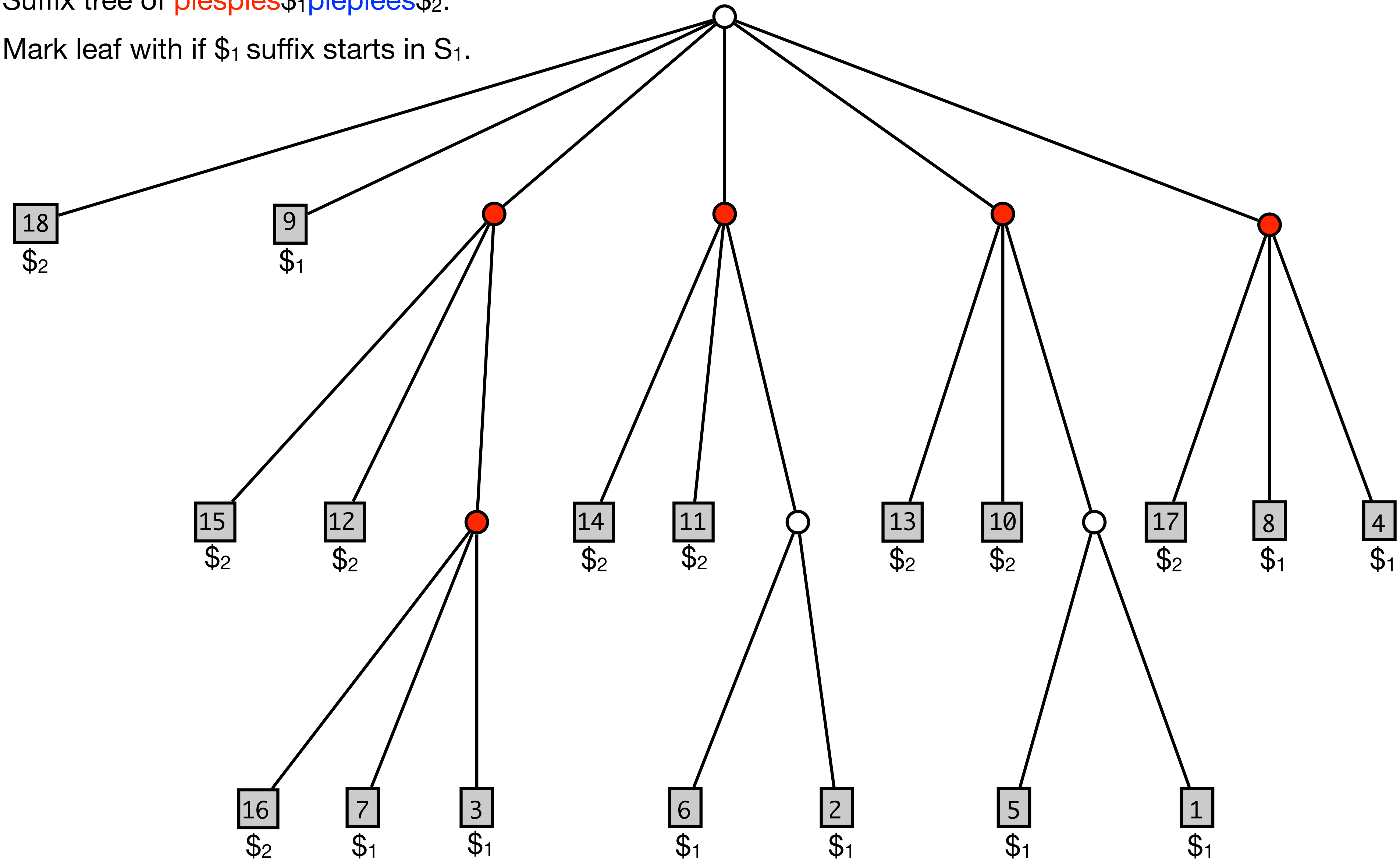
Generalized suffix tree

- Suffix tree of piespies$\$_1$piepiees$\$_2$.
- Mark leaf with if $\$_1$ suffix starts in S_1.



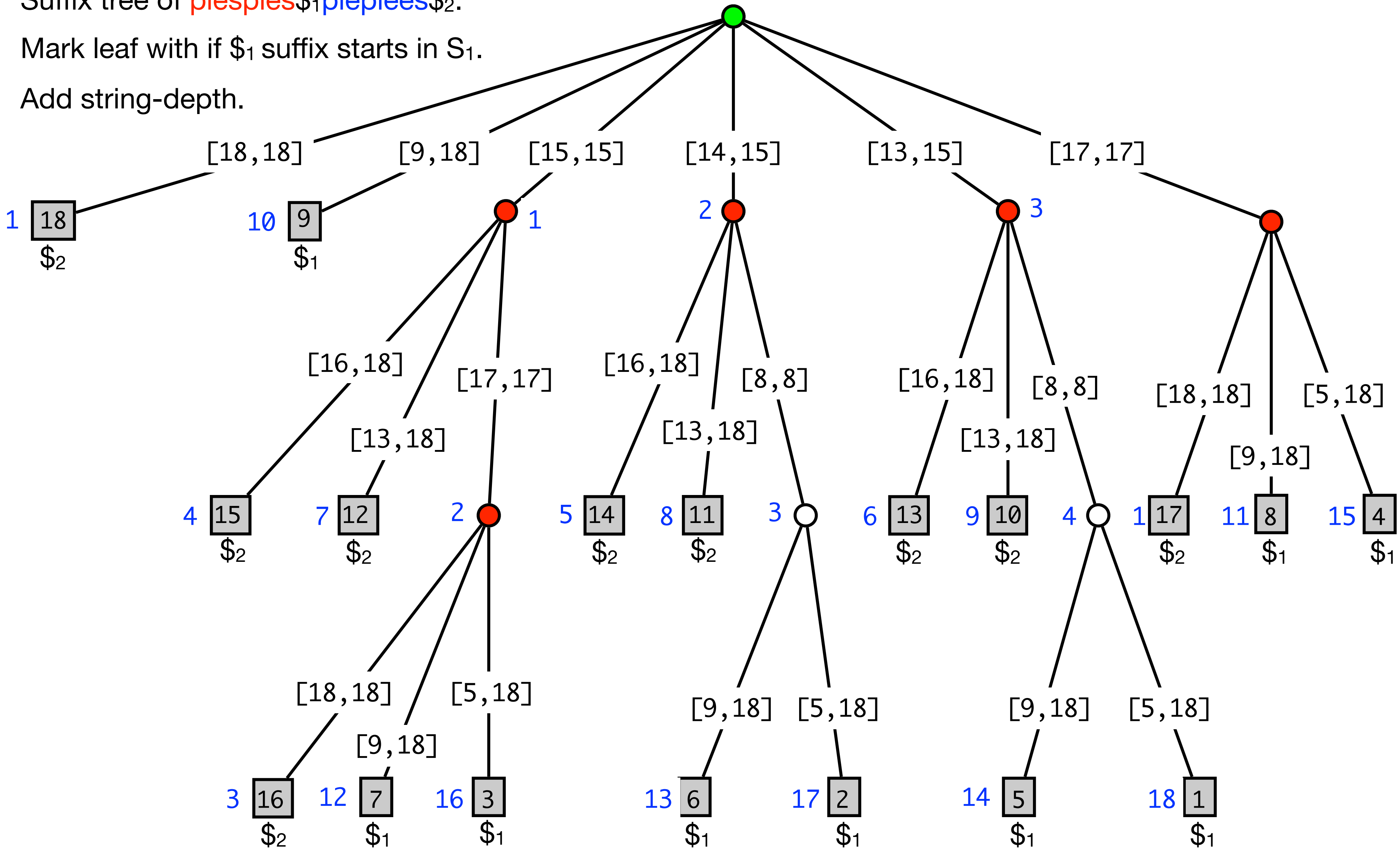
Generalized suffix tree

- Suffix tree of piespies$\$_1$piepiees$\$_2$.
- Mark leaf with if $\$_1$ suffix starts in S_1.



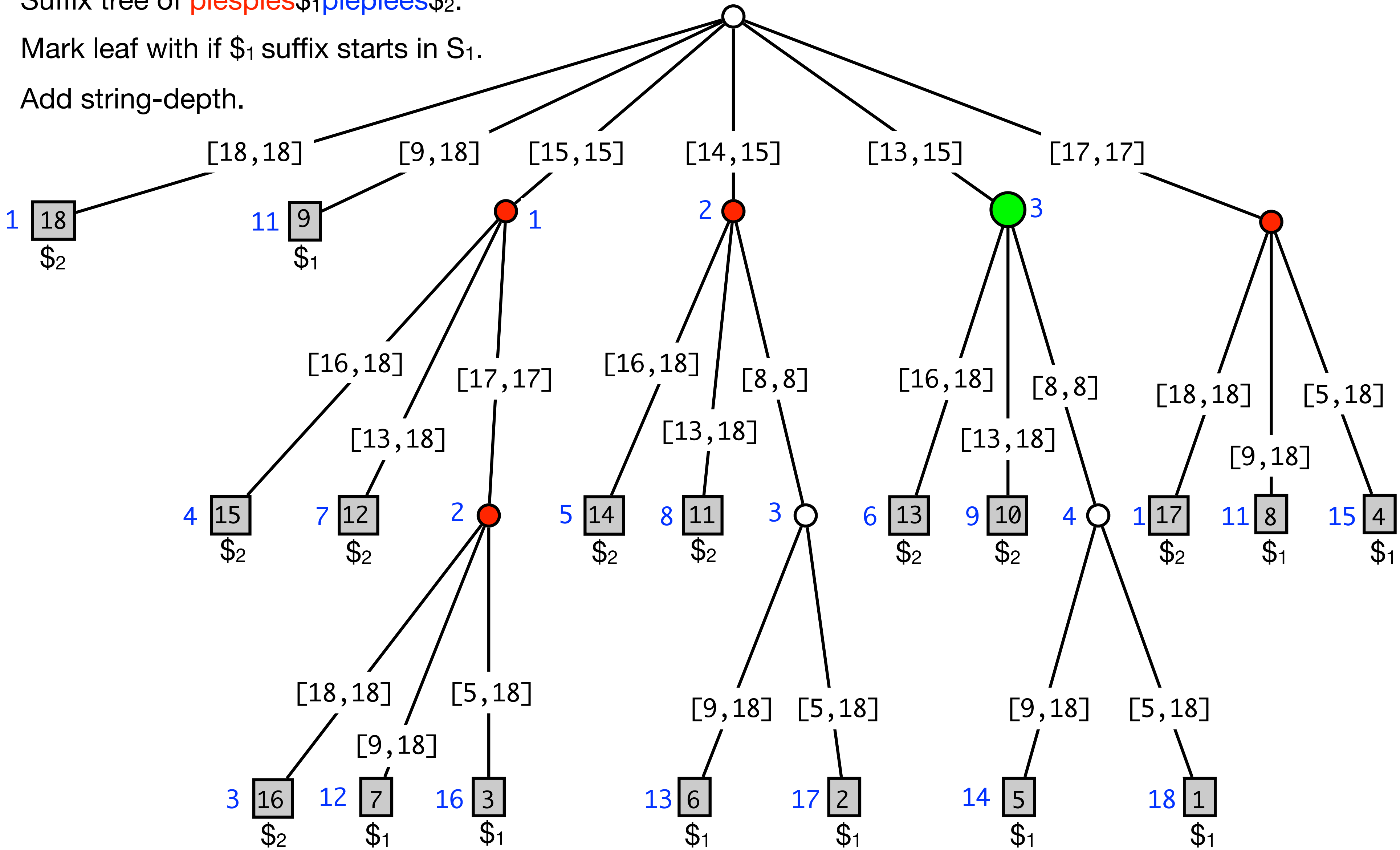
Generalized suffix tree

- Suffix tree of piespies$\$_1$piepiees$\$_2$.
- Mark leaf with if $\$_1$ suffix starts in S_1.
- Add string-depth.



Generalized suffix tree

- Suffix tree of piespies$\$_1$piepiees$\$_2$.
- Mark leaf with if $\$_1$ suffix starts in S_1.
- Add string-depth.



S[13,15] = "pie" is the longest common substring.

Longest common substring

- Using a suffix tree we can solve the longest common substring problem in linear time (for a constant size alphabet).