EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
HUMANITIES
Department of General and Computational Linguistics

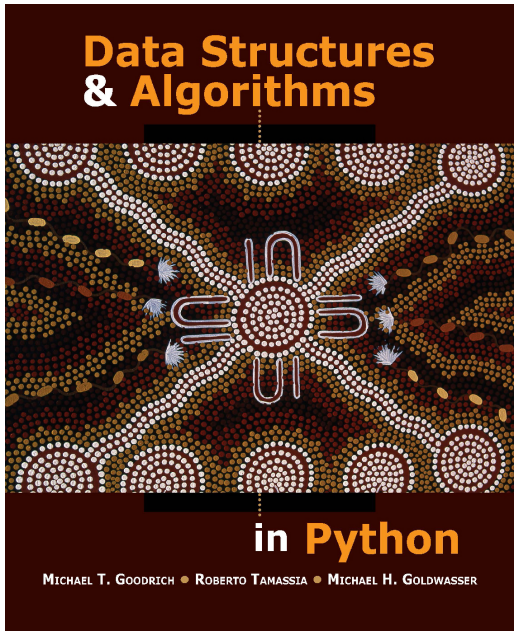
Tries

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Data Structures & Algorithms in Python

MICHAEL GOODRICH
ROBERTO TAMASSIA
MICHAEL GOLDWASSER



13.5 Tries

- Standard Tries
- Compressed Tries
- Suffix Tries

Standard Tries

Standard Tries

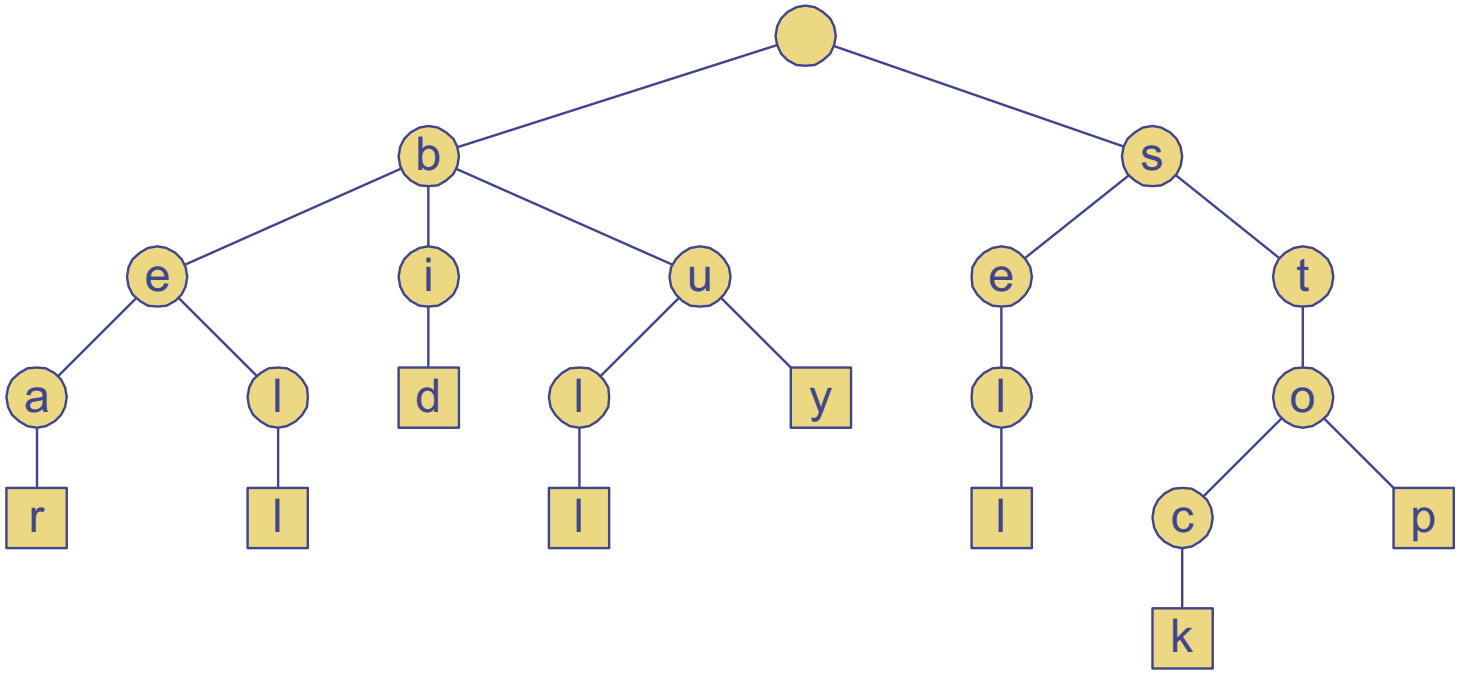
- a trie (pronounced „try“) is a tree-based data structure for storing strings in order to support fast pattern matching
- Main application: information retrieval
- Primary query operations supported by tries: pattern matching, prefix matching
- Approach suitable for applications where a series of queries is performed on a fixed text, such that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query
- Example:
  - Website that offers pattern matching in works by Shakespeare
  - Text is large, immutable and often searched for
- Trie: compact data structure for representing a set of strings, e.g. all the words in a text
  - Supports pattern-matching queries in time proportional to the pattern size

Standard Tries – Formal Definition

- Let S be a set of s strings from alphabet Σ such that no string in S is a prefix for another string
- A standard trie for the set of strings S is an ordered tree T such that:
  - Each node of T, except the root, is labeled with a character from Σ
  - The children of an internal node of T have distinct labels and are alphabetically ordered
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v

Standard Tries - Example

- Standard trie for the set of strings $S = \{bear, bell, bid, bull, buy, sell, stock, stop\}$



Standard Tries - Properties

- An internal node can have anywhere between 1 and |Σ| children
  - In practice the average degree of internal nodes is small
  - On larger datasets, the average degree of nodes decreases with the depth of the tree (fewer strings sharing a common prefix)
  - In many languages there are character combinations that are unlikely to occur
- There is an edge connecting the root node to a child node for every character from Σ that is the first character of a string from S
- A path connecting the root node to an internal node v at depth k corresponds to a k-character prefix $X[0:k]$ of a string X of S
  - A trie stores the common prefixes in a set of strings

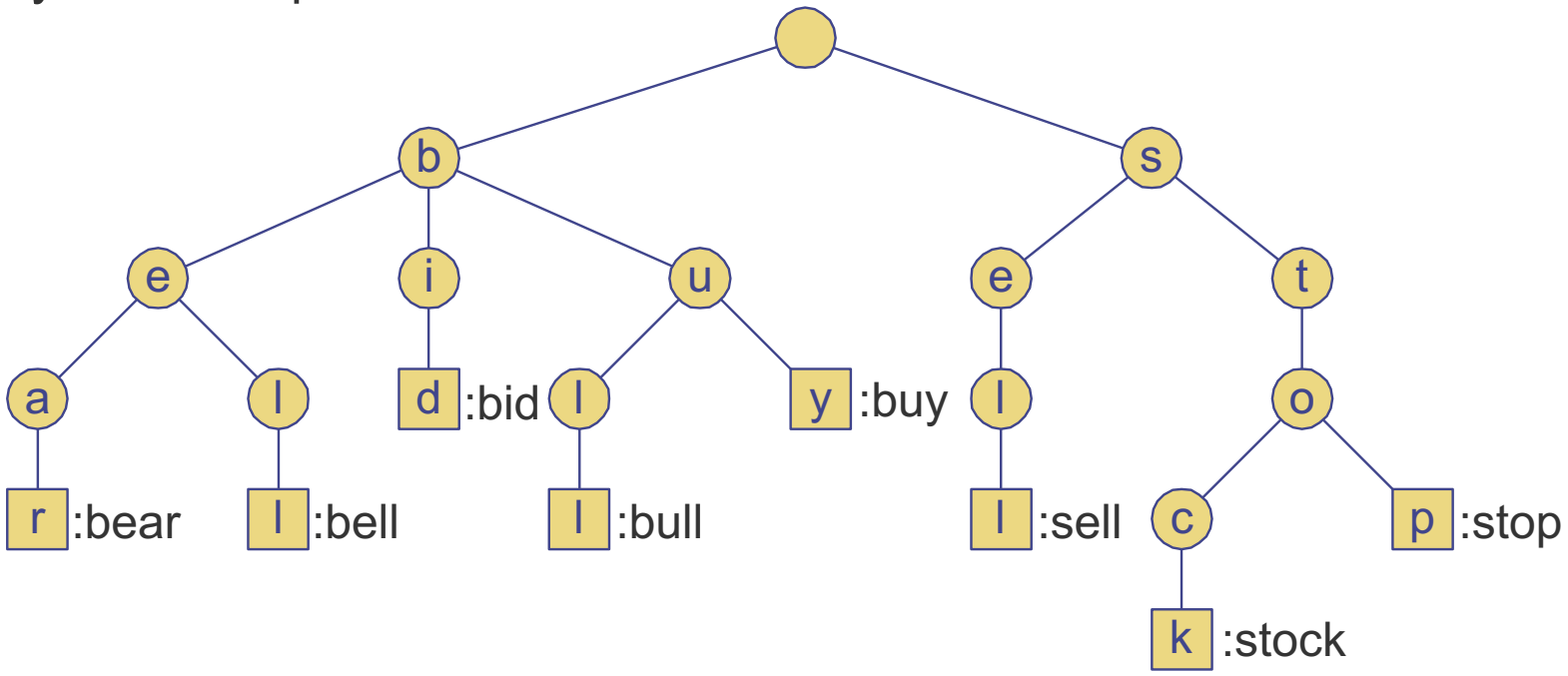
Standard Tries – Properties (cont'd)

- The following properties hold for a standard trie T storing a collection S containing s strings of total length n from an alphabet Σ:
  - The height of the trie T is equal to the length of the longest string in S
  - Every internal node of T has at most $|\Sigma|$ children
  - T has s leaves
  - The number of nodes of T is at most $n + 1$
    - Worst case: no two strings share a common, non-empty prefix – i.e. except for the root, all internal nodes have only one child

Trie Application: Map with String Keys

- A search in a trie T for the string X can be performed by tracing down from the root the path indicated by the characters of X
  - If the path can be traced and terminates in a leaf node - X is a key in the map
  - If the path cannot be traced, or it can be traced but terminates at an internal node – X is not a key in the map

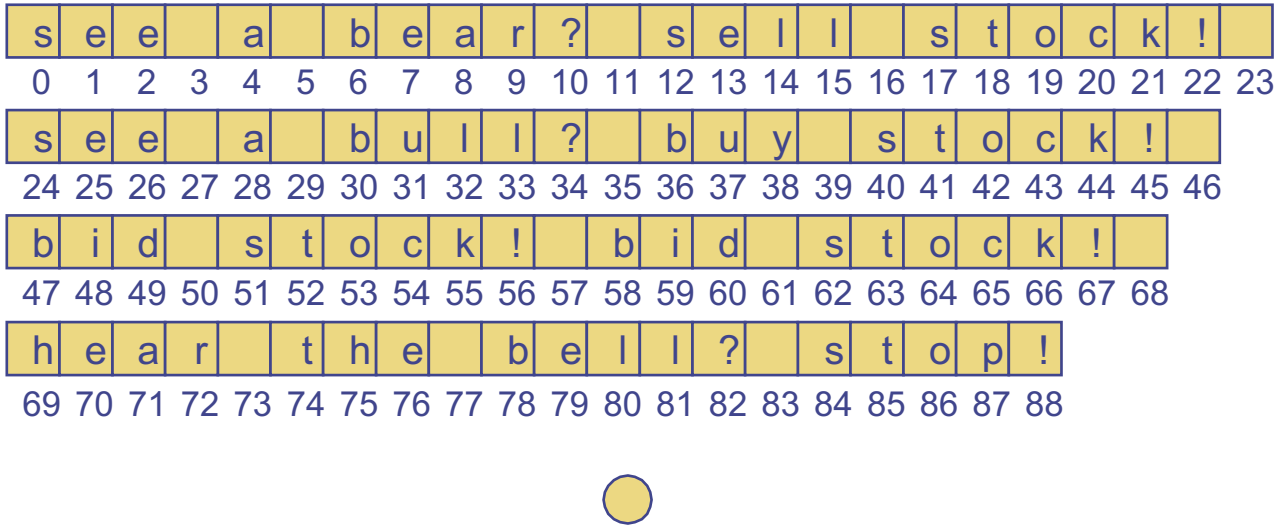
- bear
- big
- be



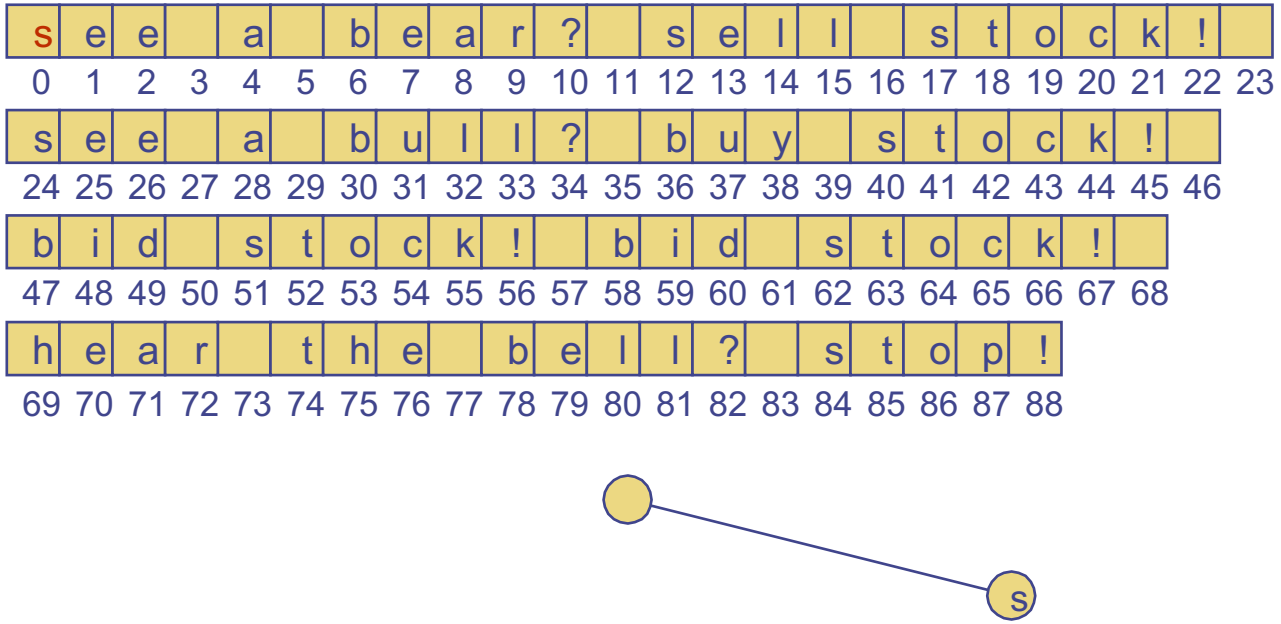
Trie Application: Map with String Keys (cont'd)

- Running time for searching for a string X of length m
  - At most $m + 1$ nodes of T are visited (the root + each of the characters)
    - At each node we spend at most $O(|\Sigma|)$ time determining what edge to follow next – that is – finding the child node which has the next character as its label
    - $O(|\Sigma|)$ is achievable even if the children are unordered – each node has at most $|\Sigma|$ children
  - Time can be improved by mapping characters to children by using at each node:
    - a secondary search table - $O(\log|\Sigma|)$
    - a hash table - $O(1)$
    - a direct lookup table of size $|\Sigma|$, if $|\Sigma|$ is small enough - $O(1)$
  - Typically, the search for a string of length m runs in $O(m)$ time

Word Matching with a Trie



Word Matching with a Trie (cont'd)



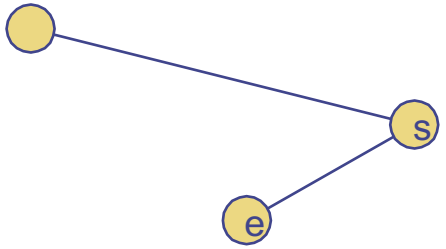
Word Matching with a Trie (cont'd)

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

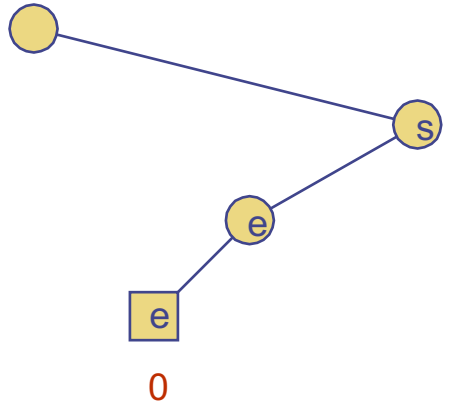
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

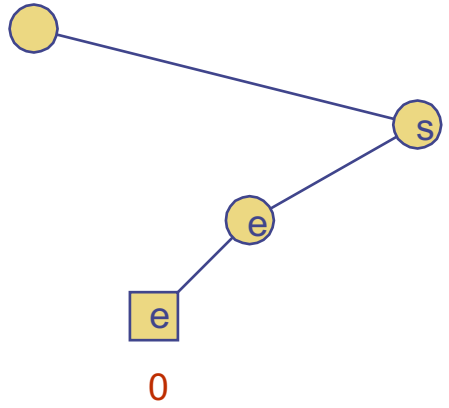
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



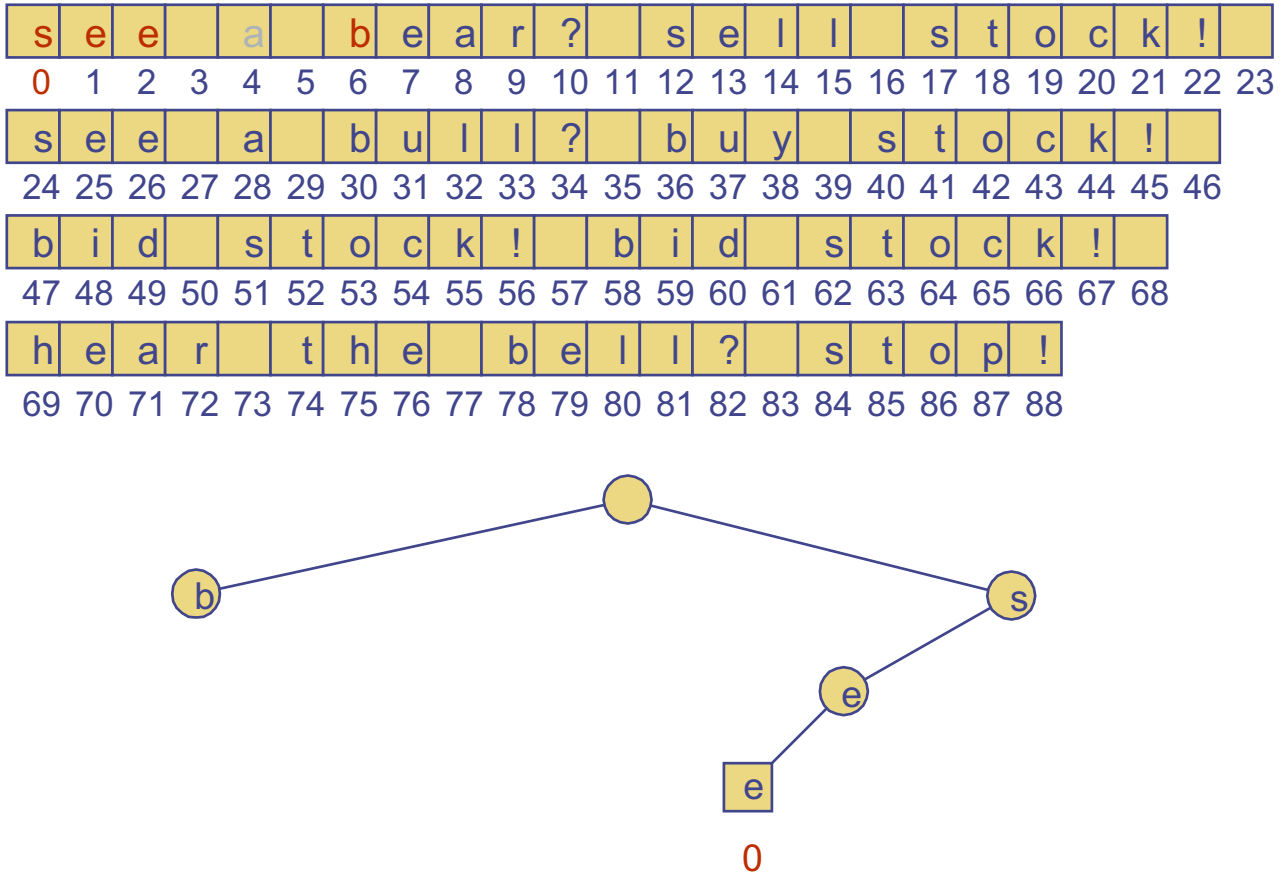
Word Matching with a Trie (cont'd)



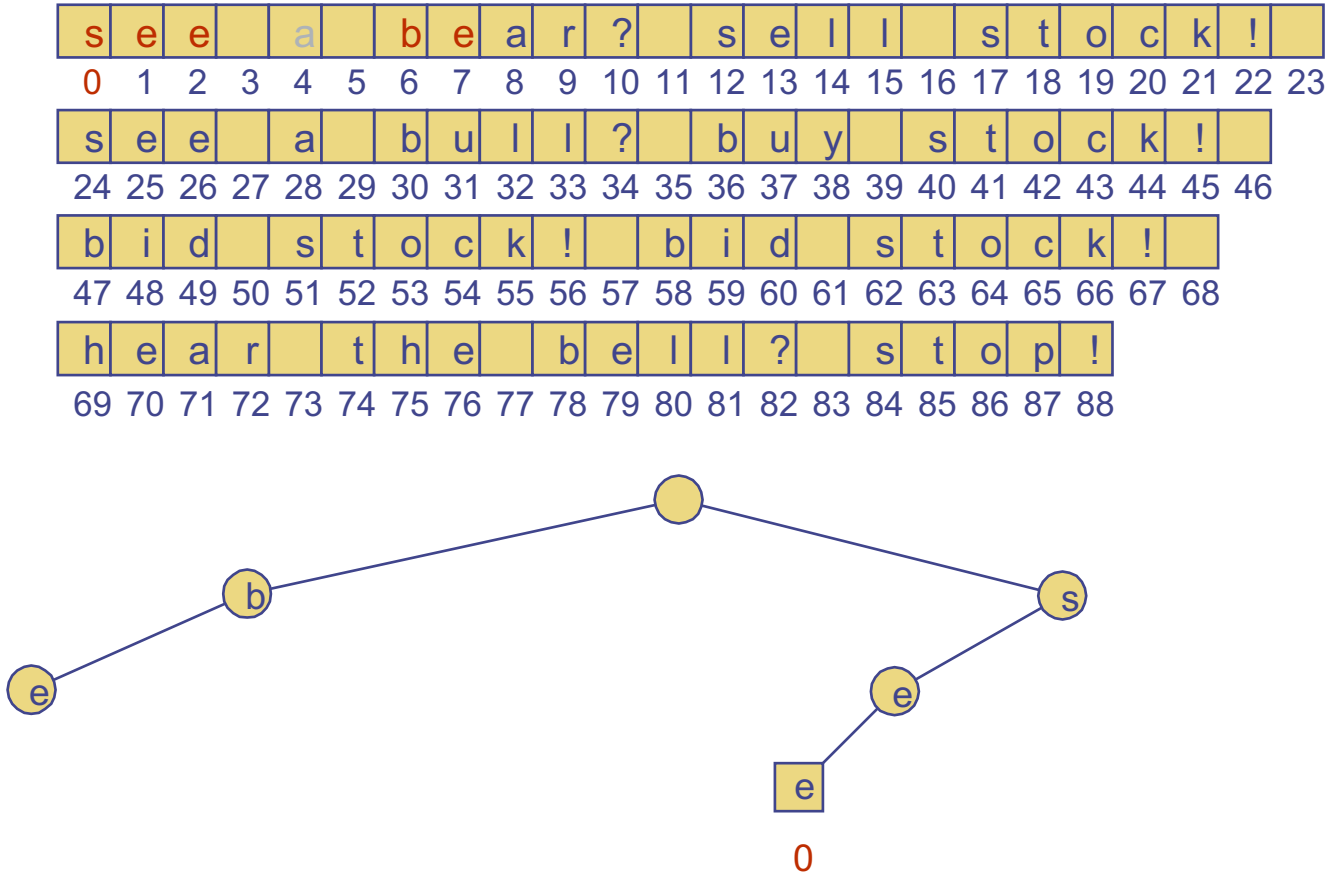
Word Matching with a Trie (cont'd)



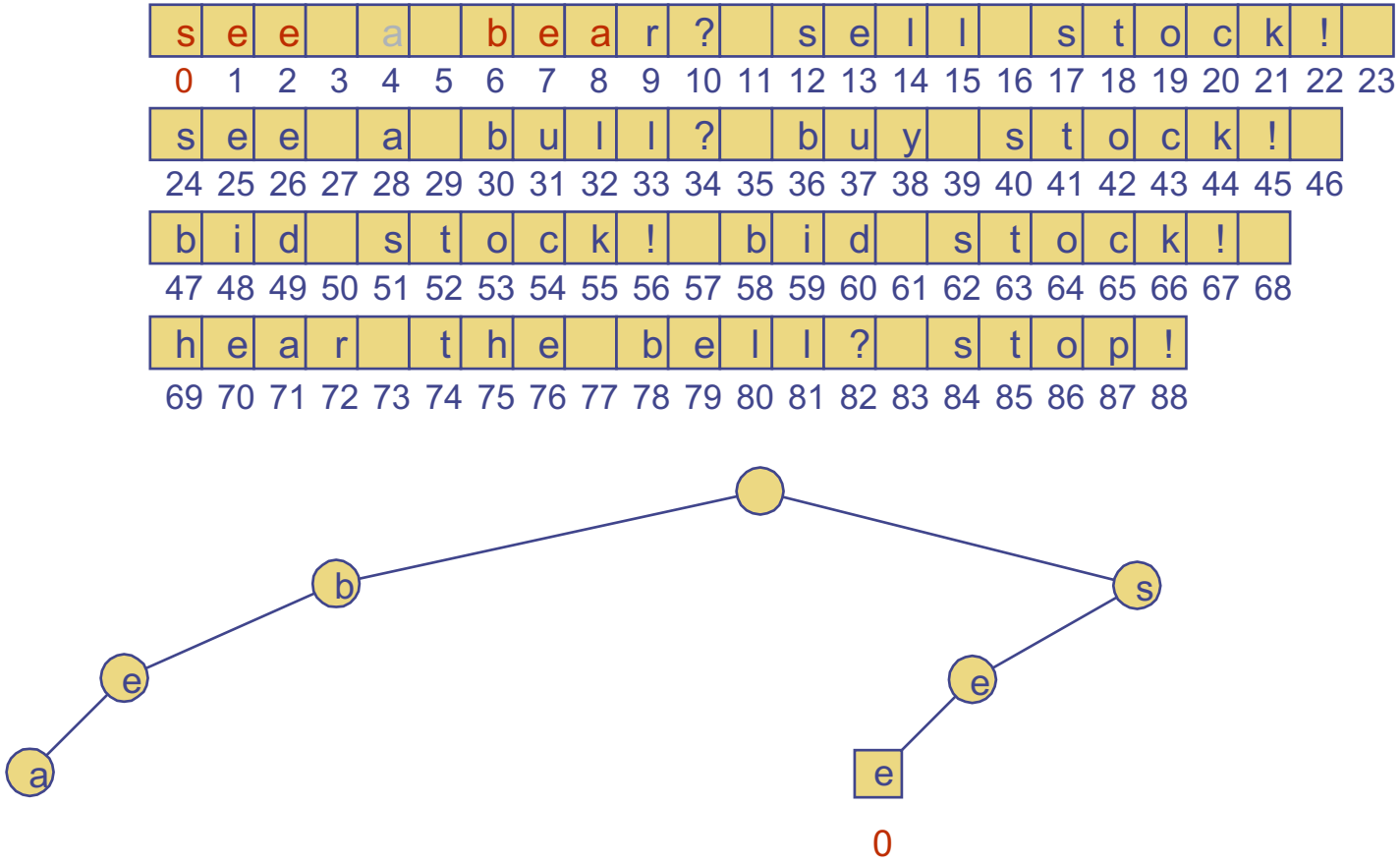
Word Matching with a Trie (cont'd)



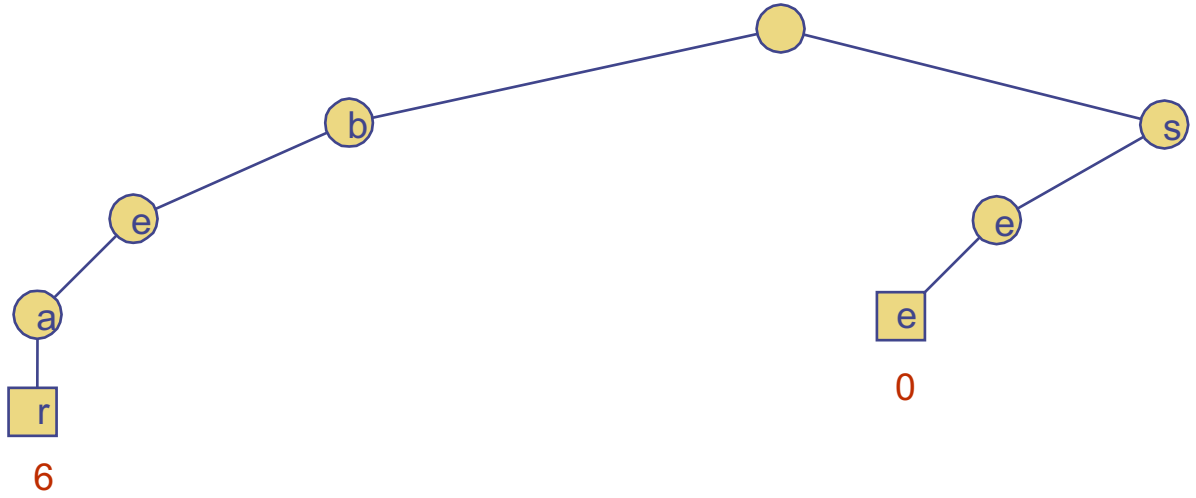
Word Matching with a Trie (cont'd)



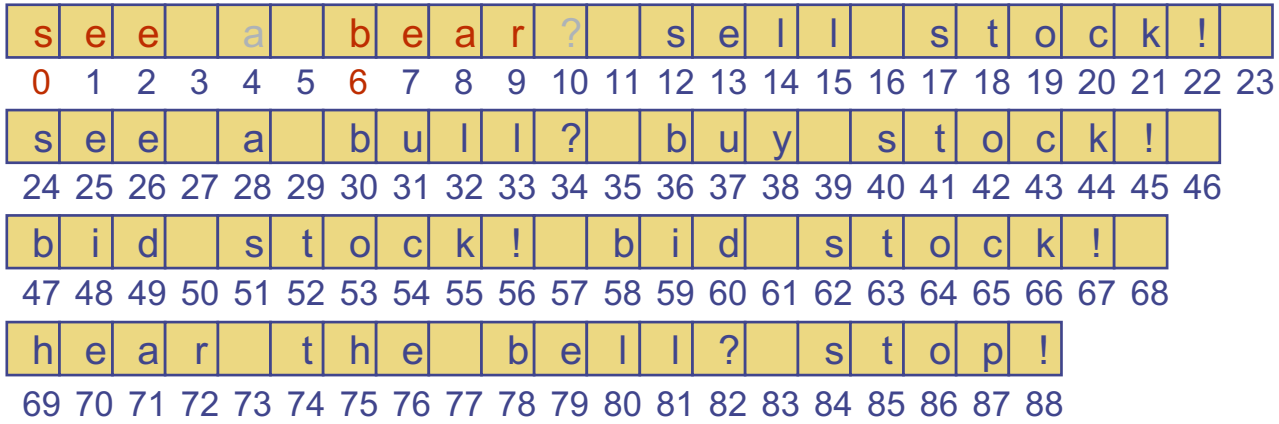
Word Matching with a Trie (cont'd)

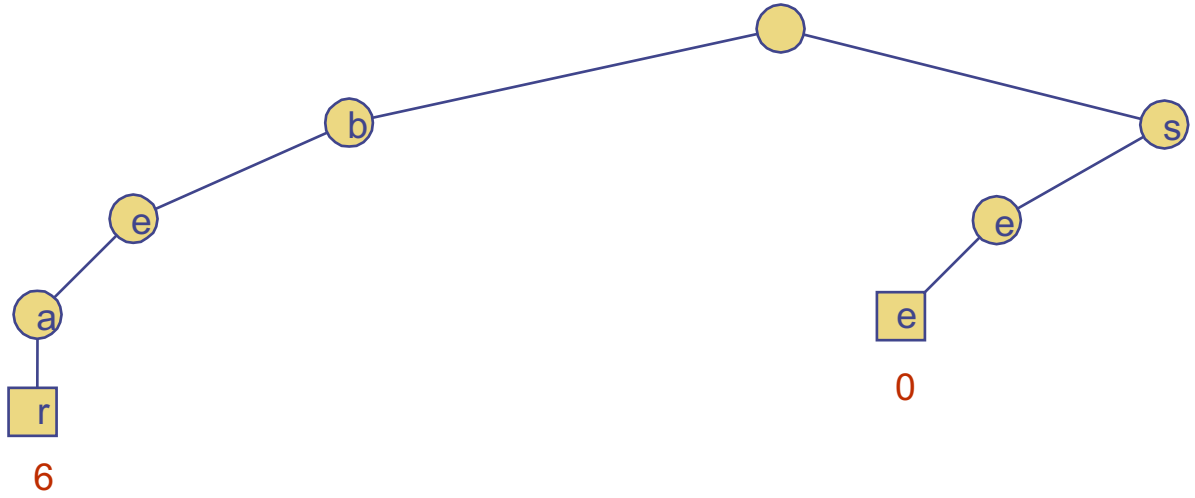


Word Matching with a Trie (cont'd)

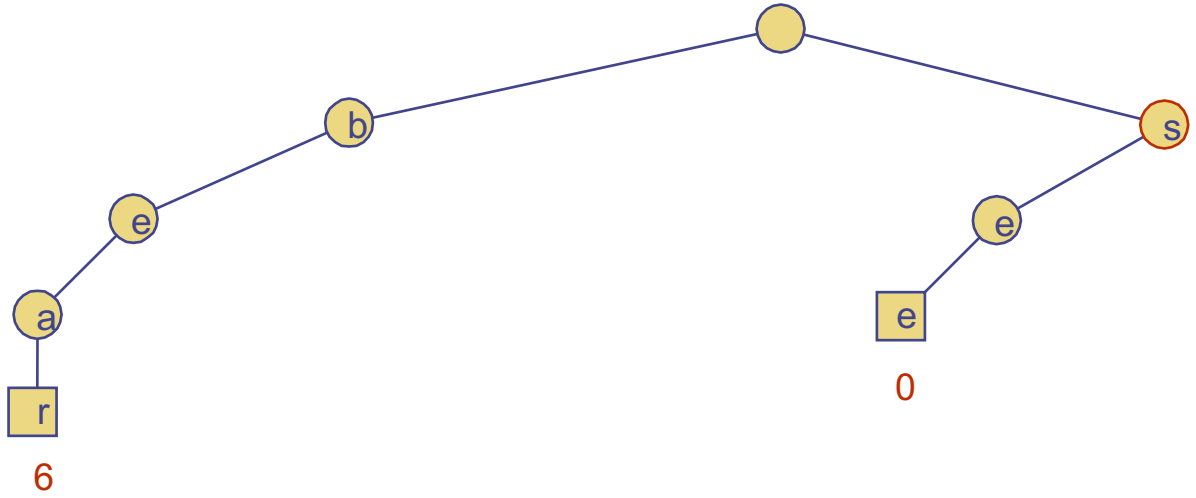


Word Matching with a Trie (cont'd)

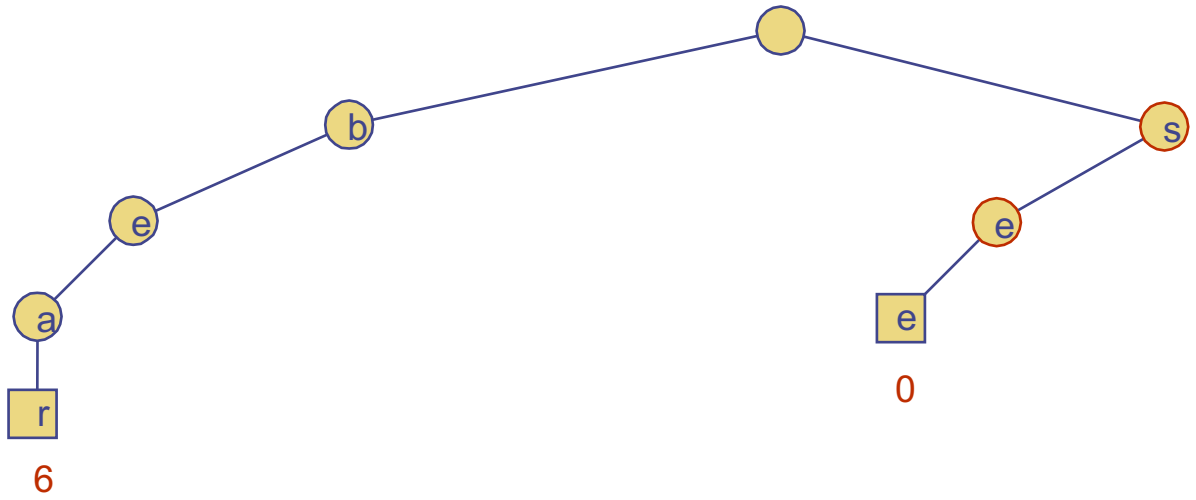




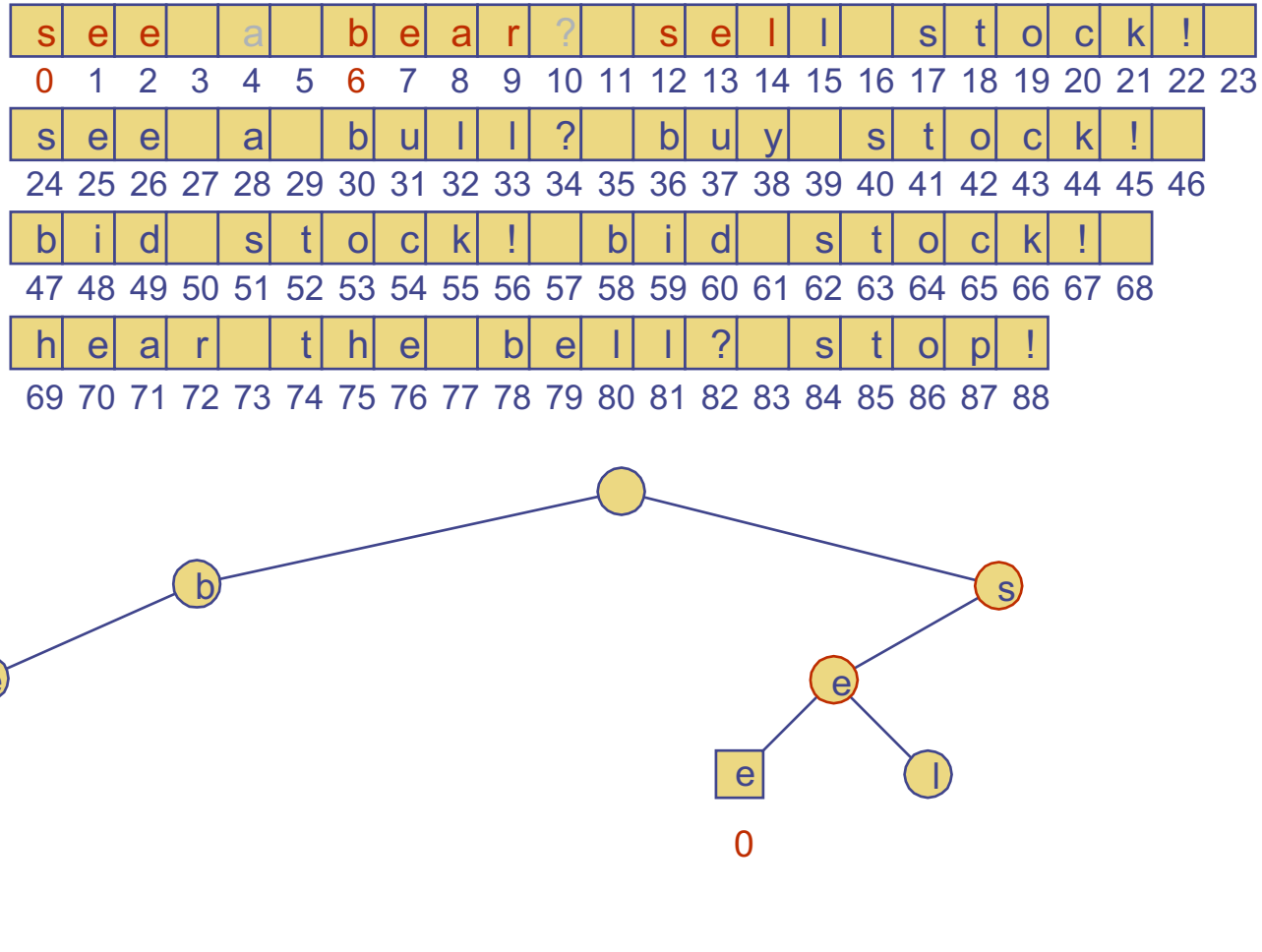
Word Matching with a Trie (cont'd)



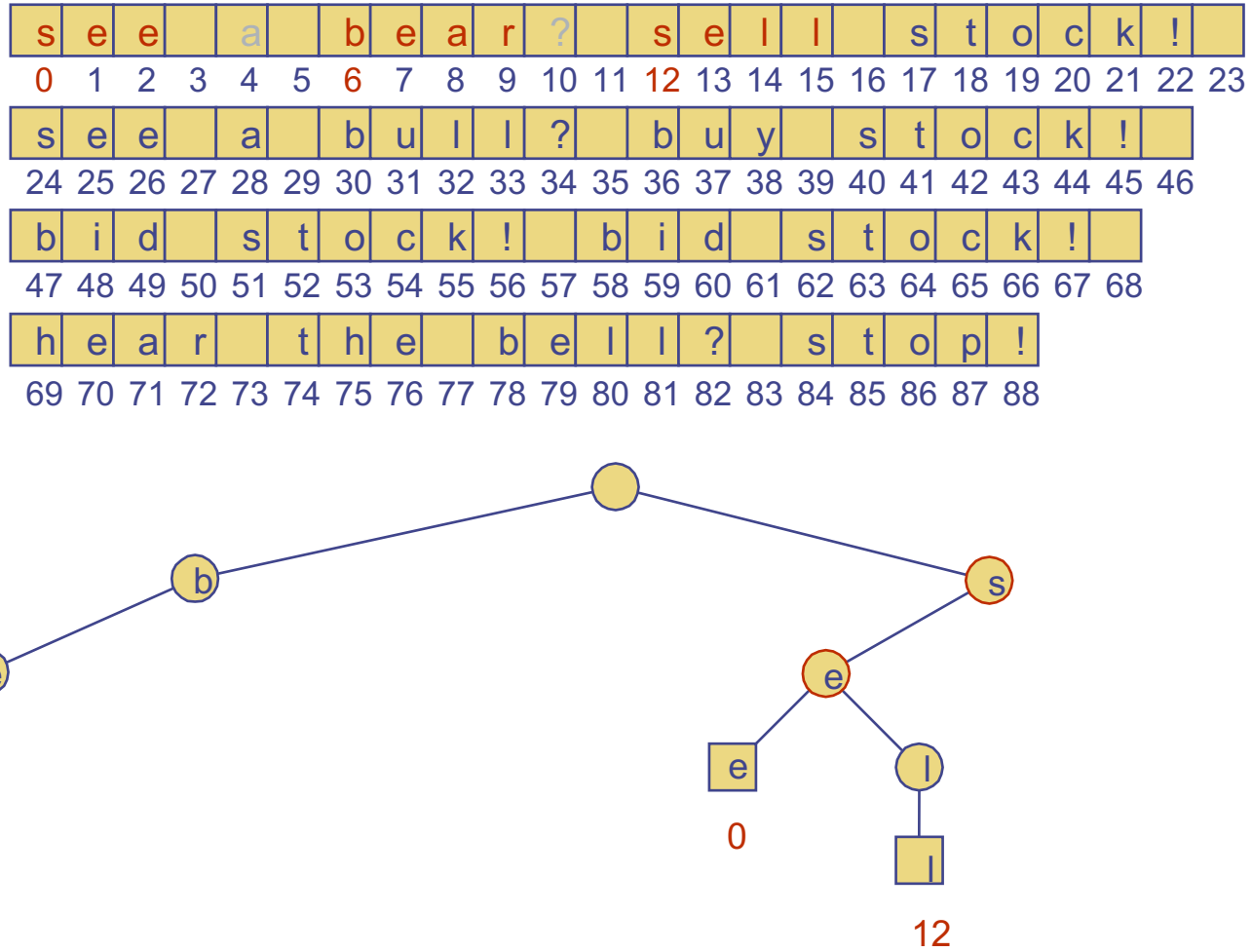
Word Matching with a Trie (cont'd)



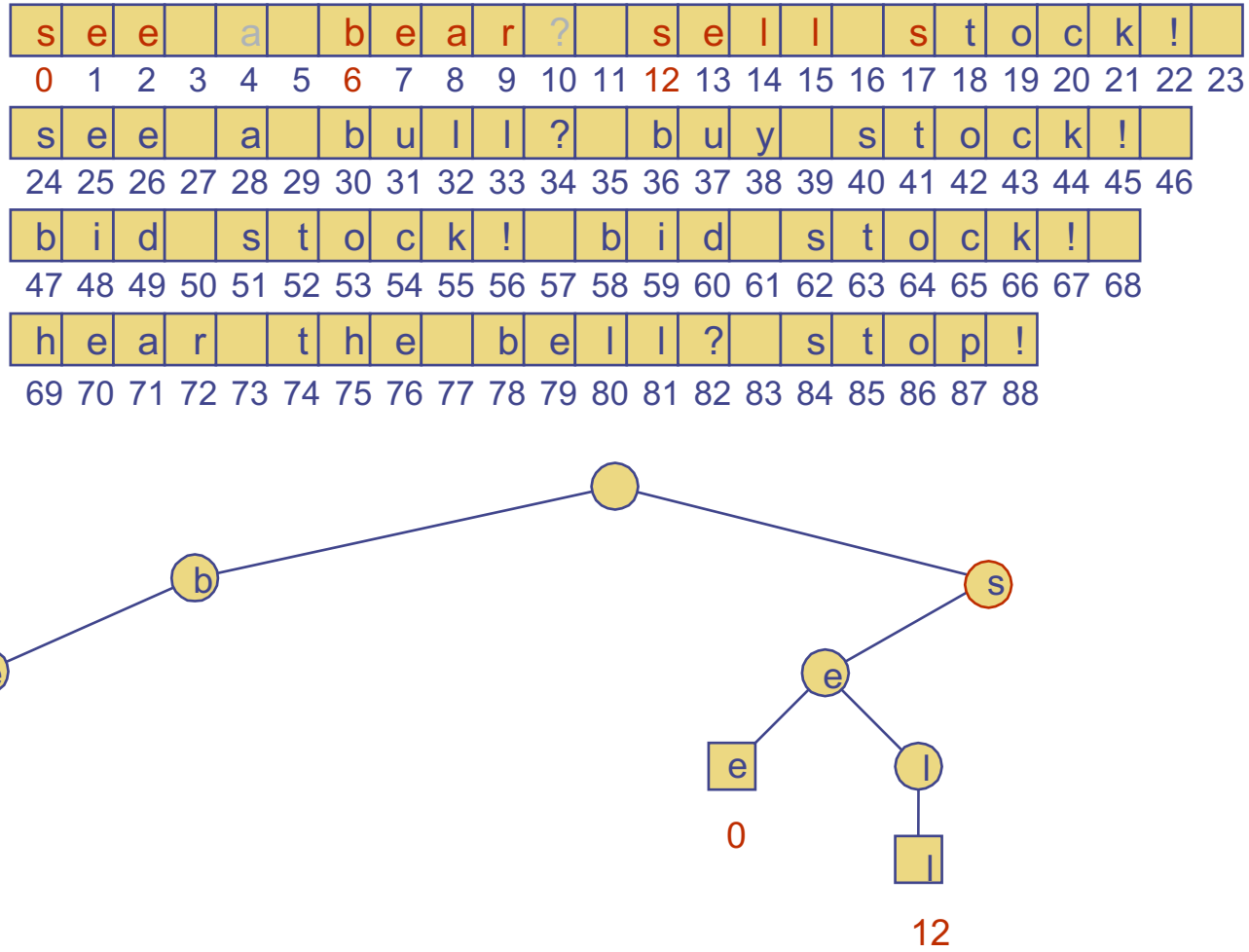
Word Matching with a Trie (cont'd)



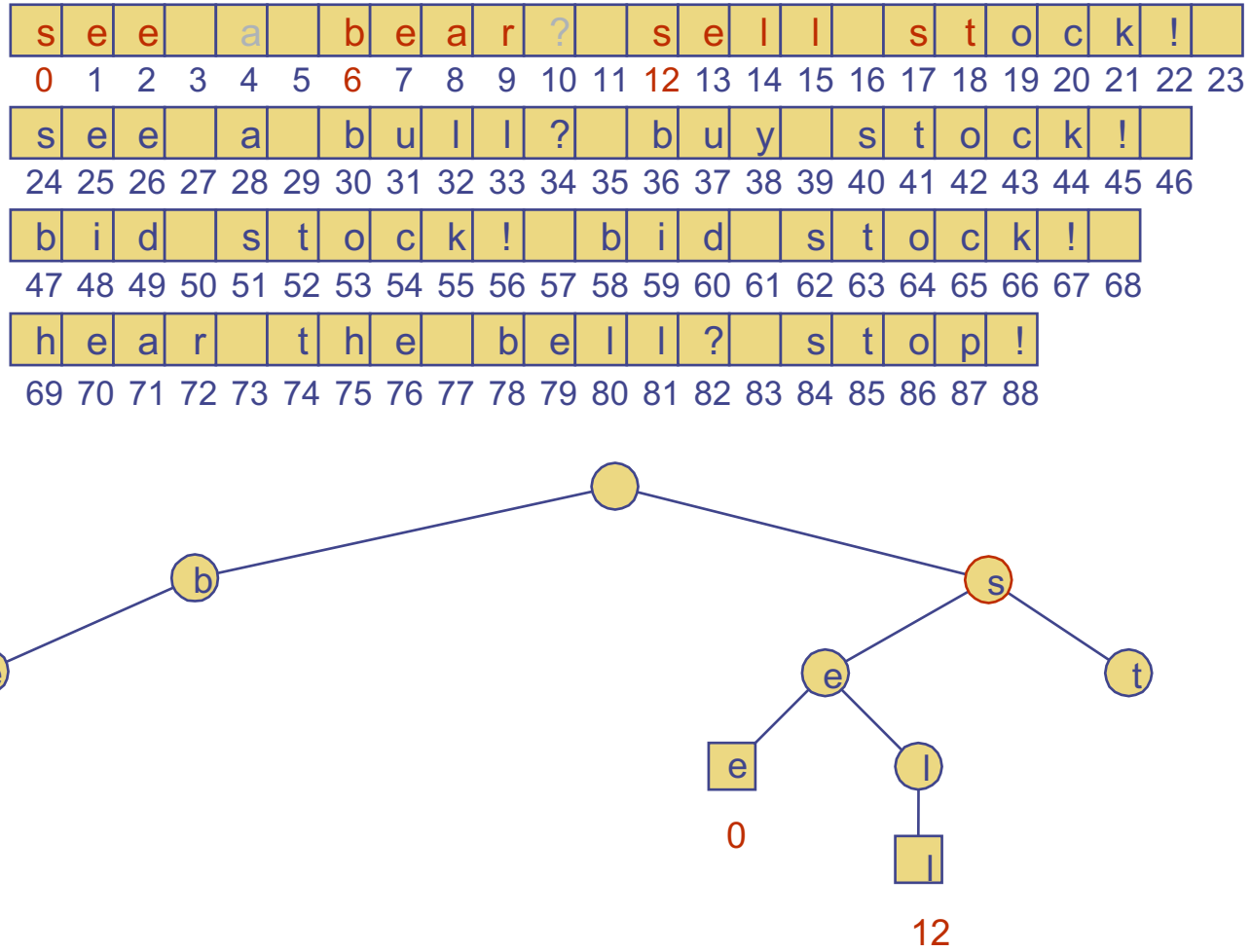
Word Matching with a Trie (cont'd)



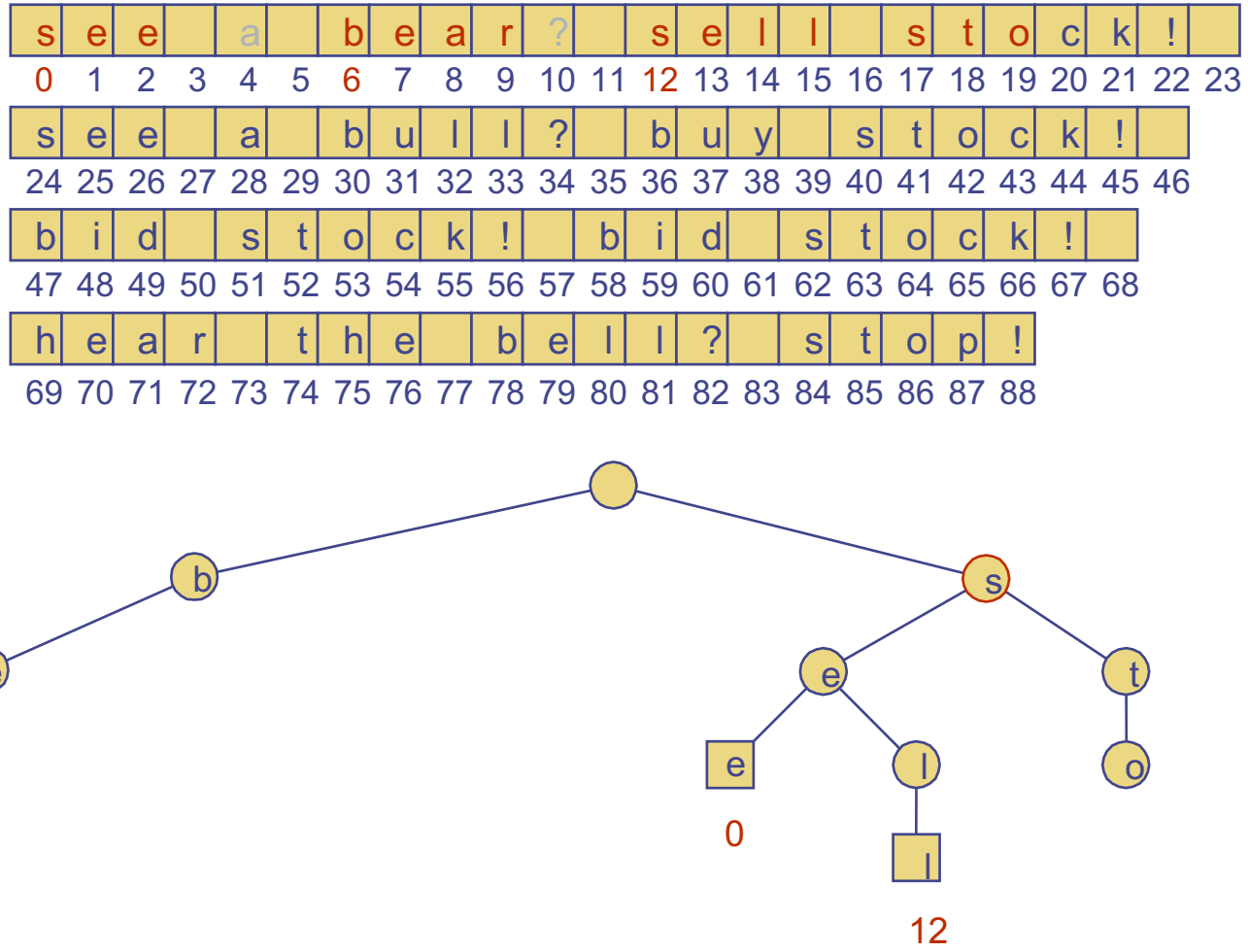
Word Matching with a Trie (cont'd)



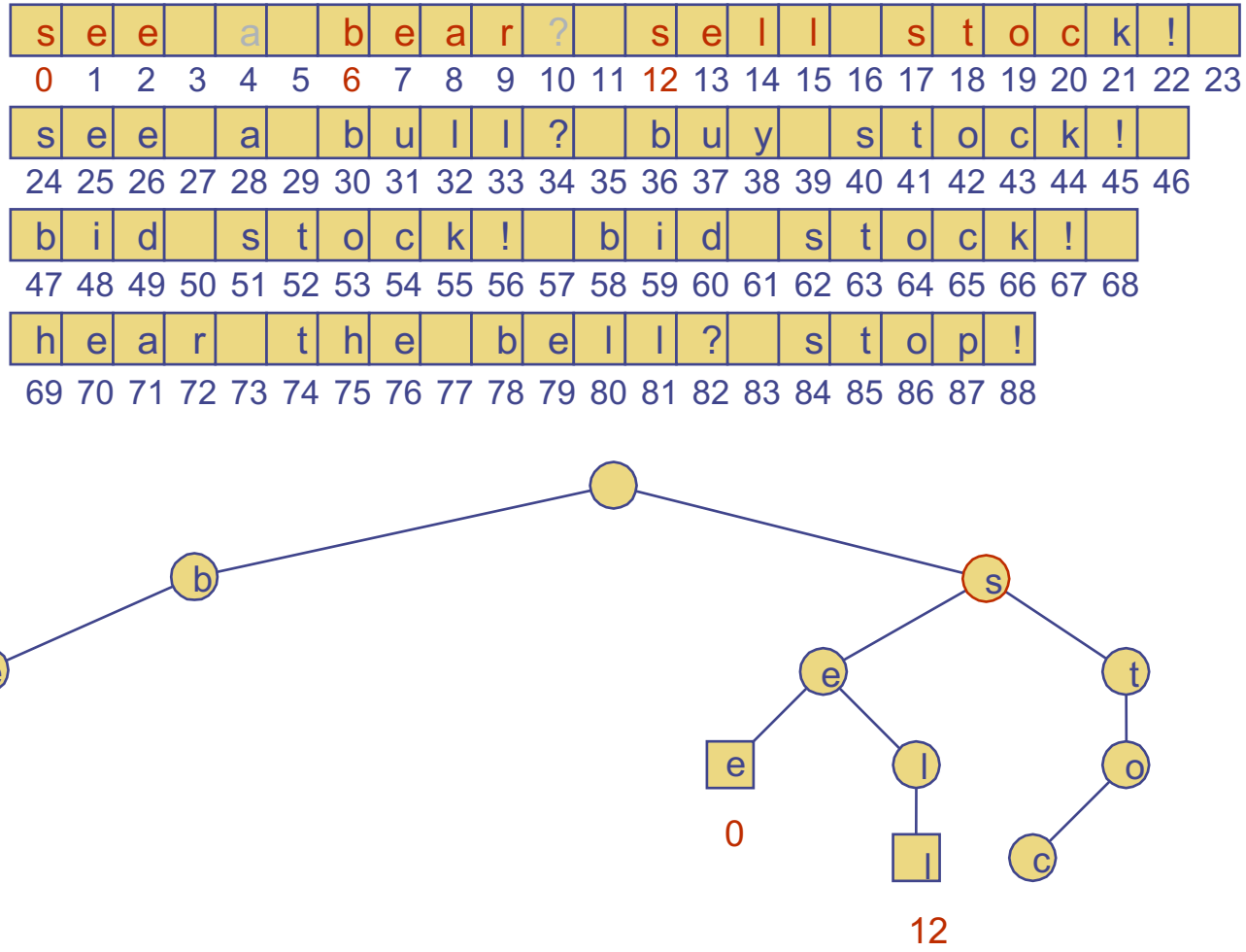
Word Matching with a Trie (cont'd)



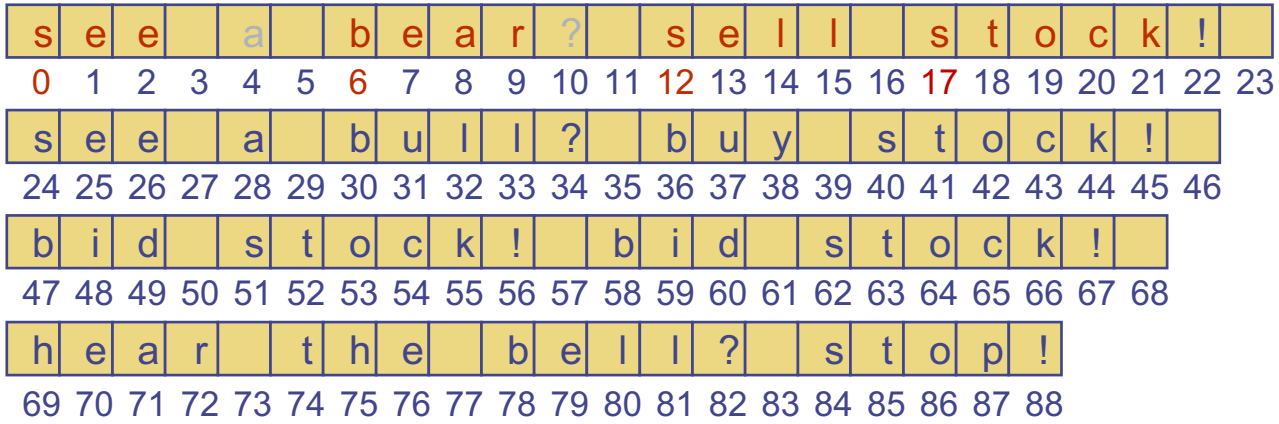
Word Matching with a Trie (cont'd)

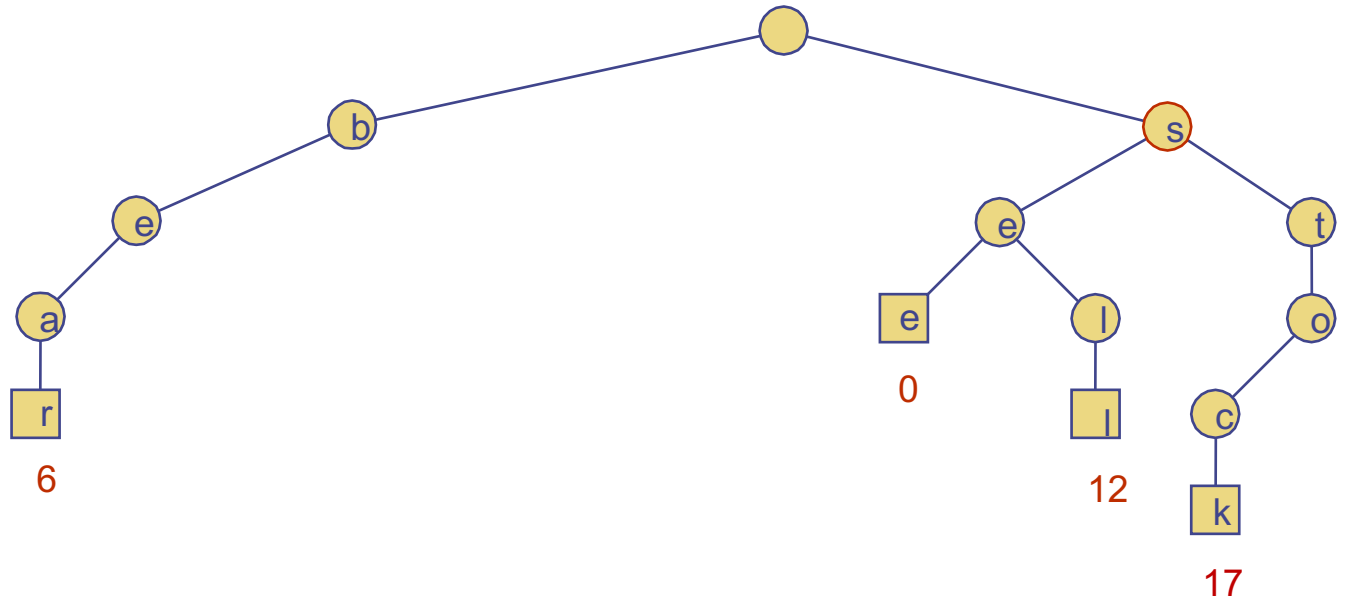


Word Matching with a Trie (cont'd)

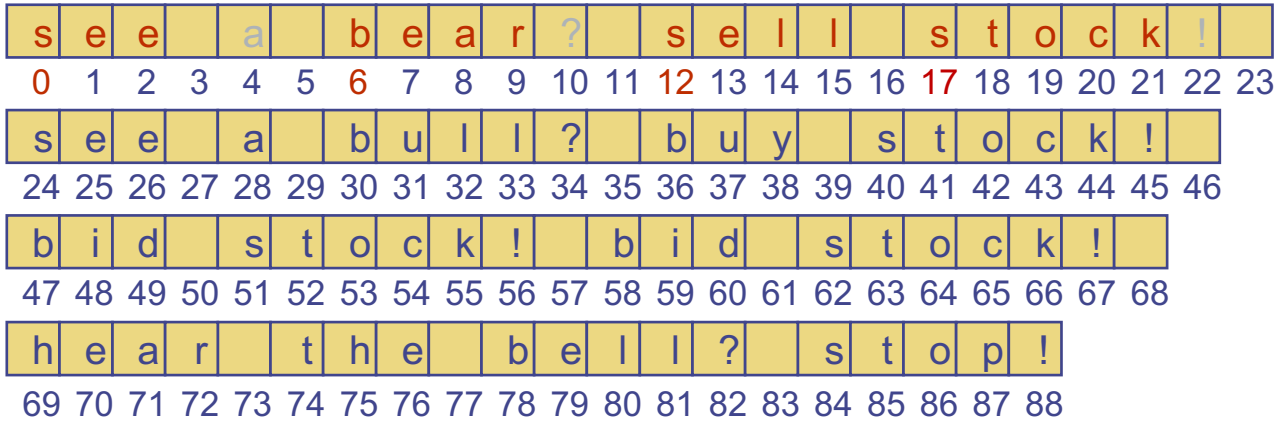


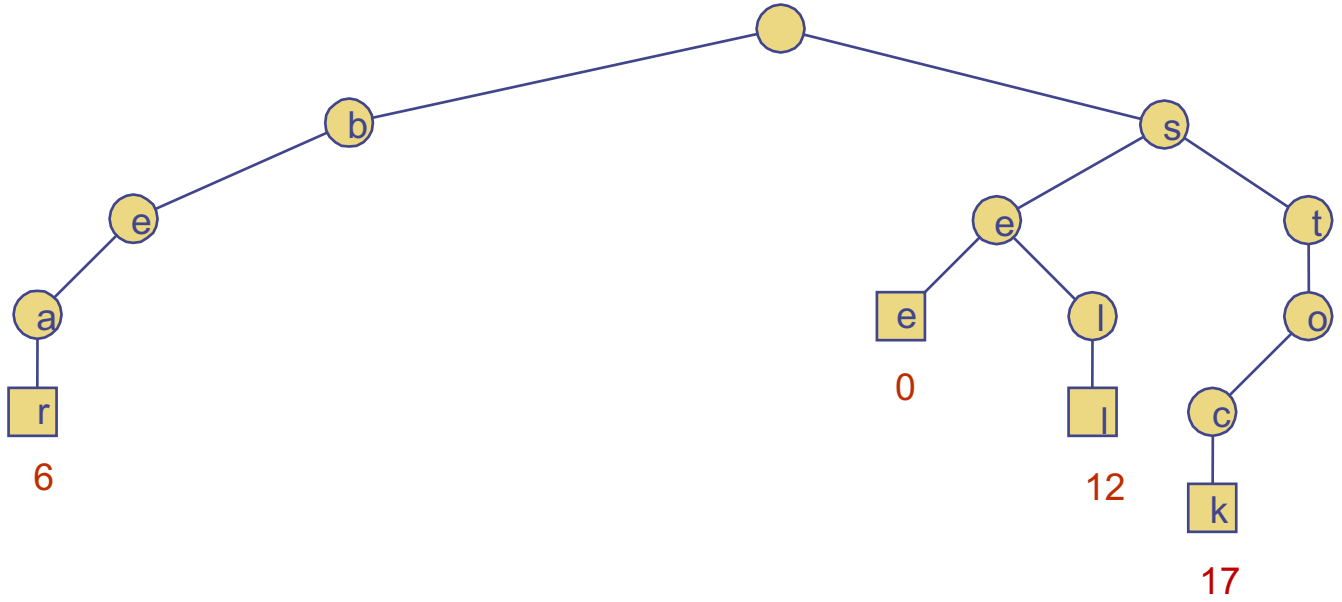
Word Matching with a Trie (cont'd)



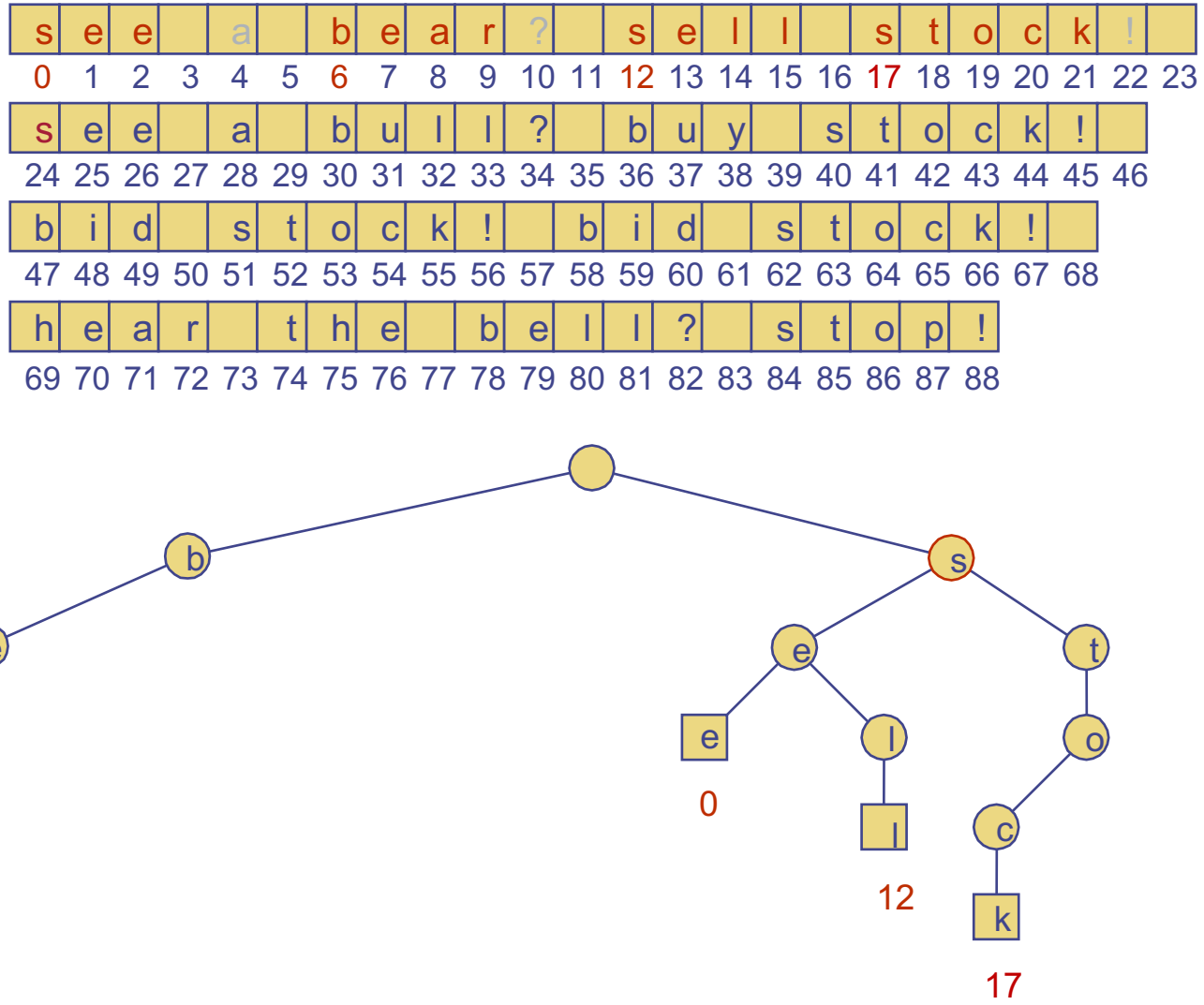


Word Matching with a Trie (cont'd)

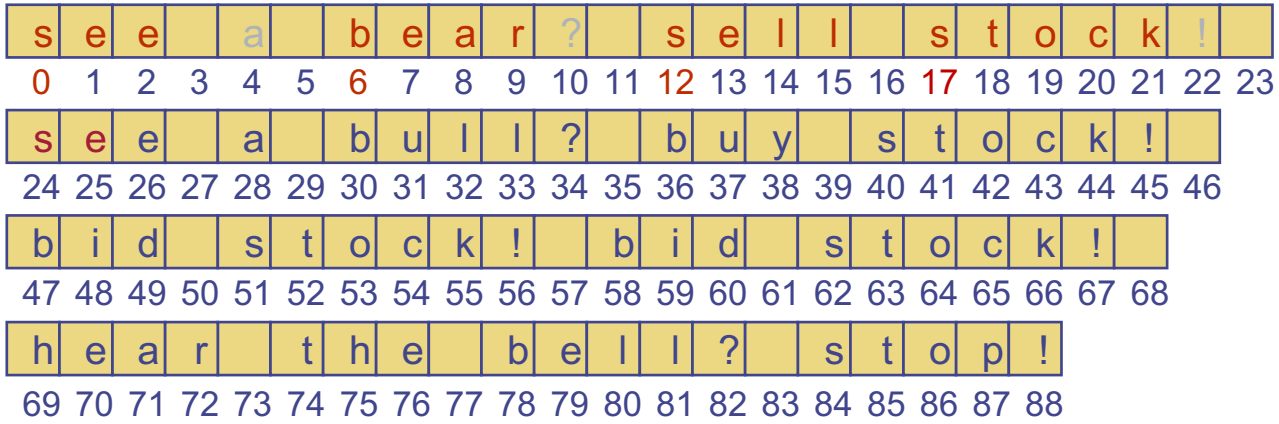


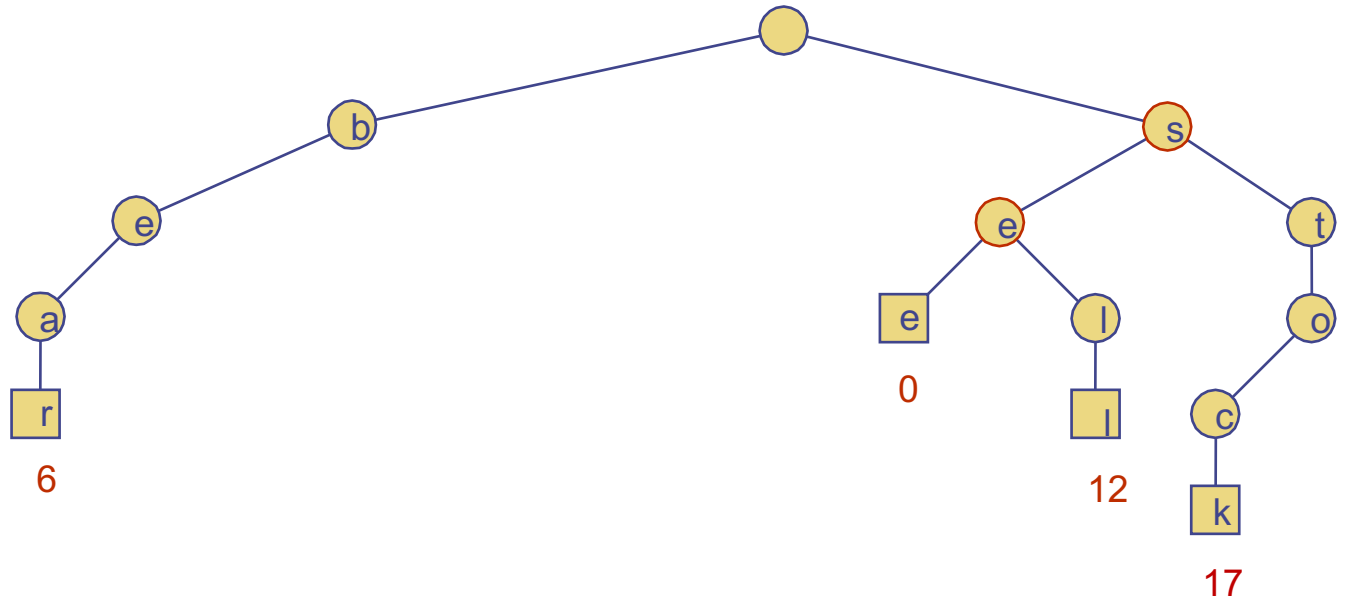


Word Matching with a Trie (cont'd)

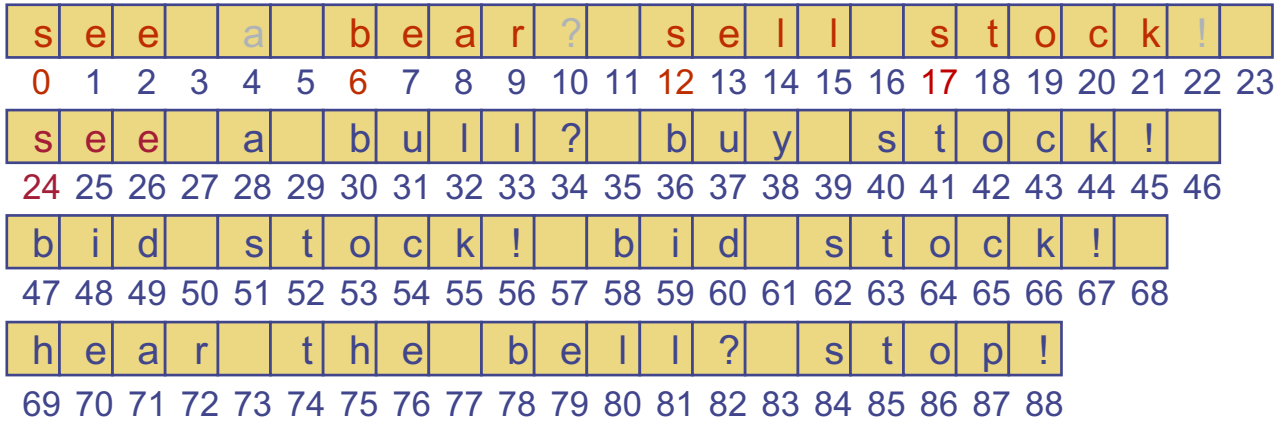


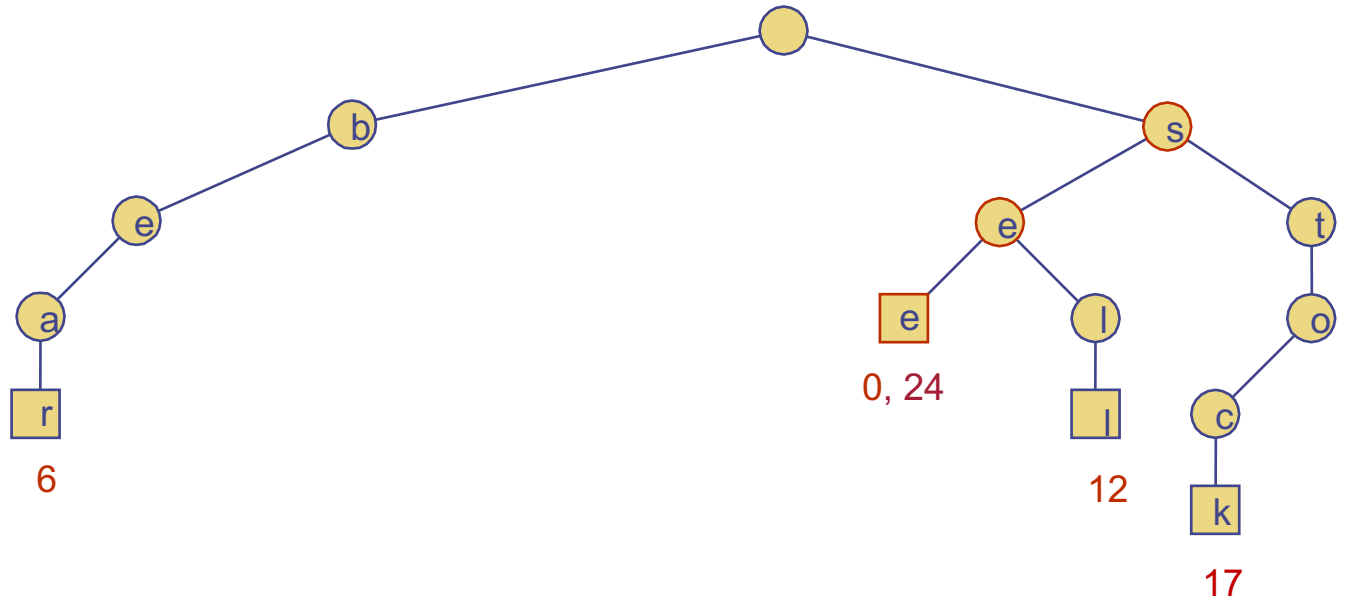
Word Matching with a Trie (cont'd)



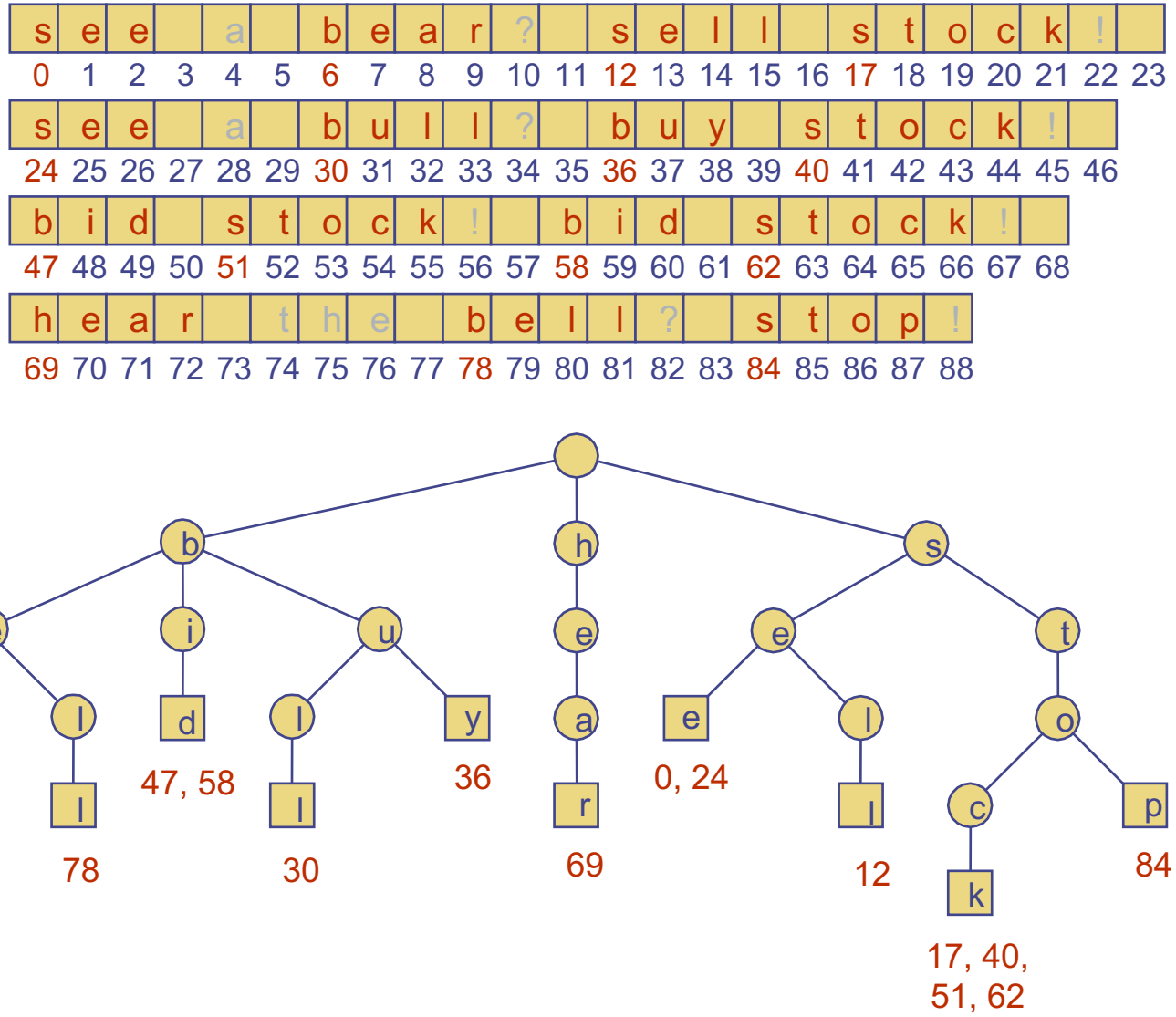


Word Matching with a Trie (cont'd)





Word Matching with a Trie (cont'd)

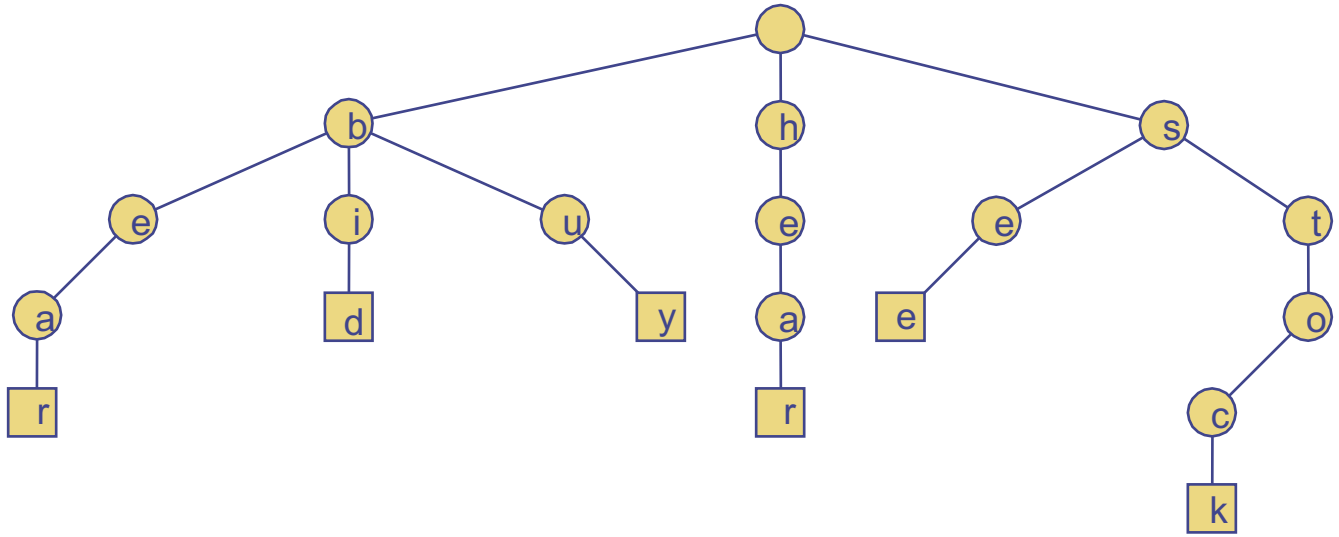


Standard Trie construction – Running Time

- construct a standard trie T for a set S of strings of total length n over the alphabet Σ
- Incremental algorithm
  - Insert strings one at a time
    - no string of S is a substring of another string
    - append \$ at the end of each string, where \$ $\notin |\Sigma|$, if there are prefixes
  - To insert the string X of size m into T: trace the path associated with X in T, creating new nodes if stuck
  - Running time for each insertion is similar to search: $O(m \cdot |\Sigma|)$ worst-case performance, $O(m)$ when using a secondary hash table at each node
  - Total expected running time: $O(n)$ - n is the total length of the strings in S

Better Tries?

- Standard tries have a potential space inefficiency
  - There can be many nodes with only one child, which is wasteful
- Solution
  - compressed tries (aka Patricia tries)

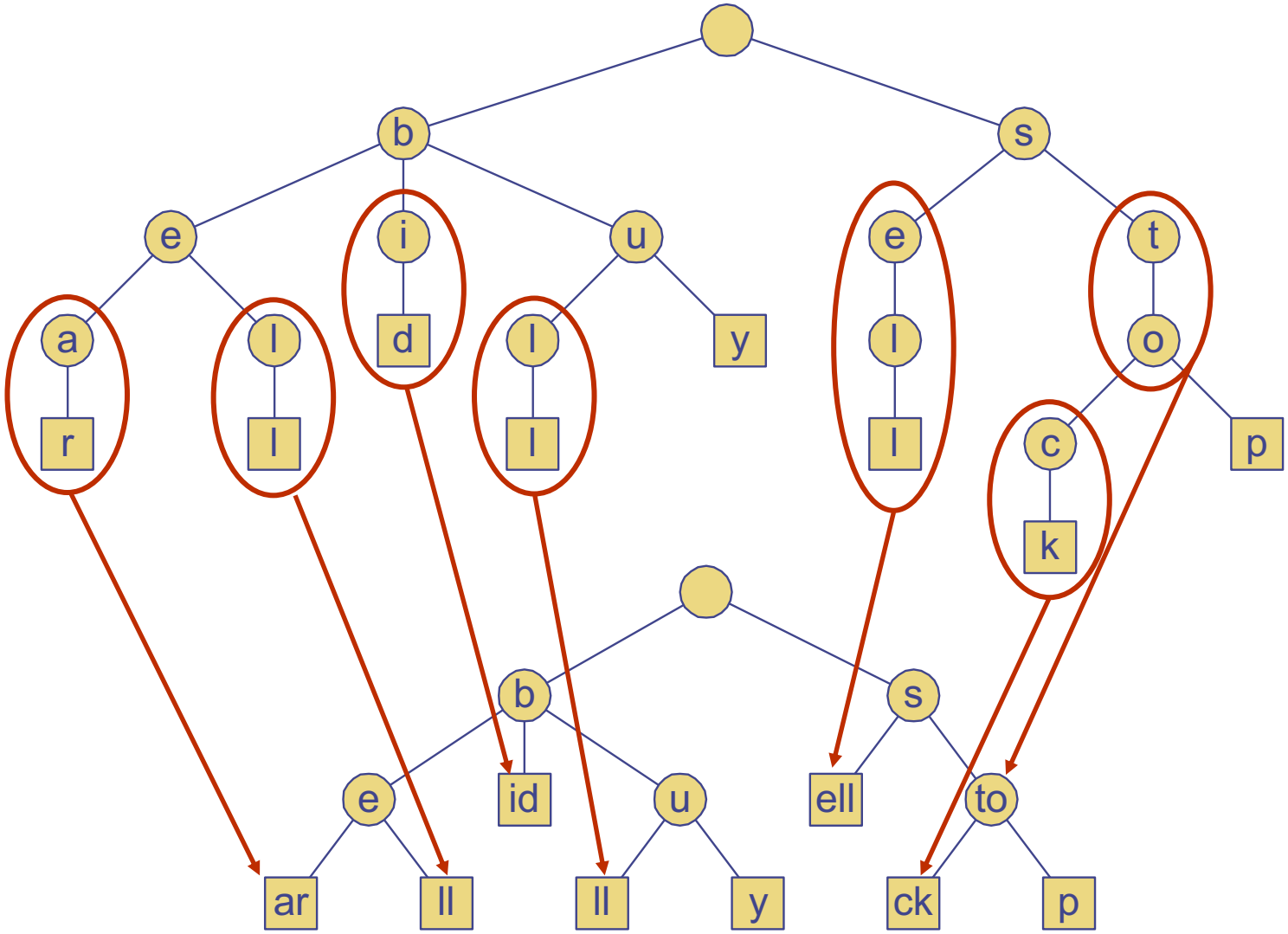


Compressed Tries

Compressed Tries

- Similar to a standard trie, but each internal node must have at least two children
- Rule enforced by compressing chains of single-child nodes into individual edges
- consider T, a standard trie; an internal node v of T is redundant if v has only one child and it is not the root
- A chain of $k \geq 2$ edges, $(v_0, v_1)(v_1, v_2) \dots (v_{k-1}, v_k)$ is redundant if:
  - v_i is redundant for $i = 1, \dots, k-1$
  - v_0 and v_k are not redundant
- A standard trie can be transformed into a compressed trie by replacing each redundant chain of $k \geq 2$ edges, $(v_0, v_1)(v_1, v_2) \dots (v_{k-1}, v_k)$, into a single edge (v_0, v_k) and relabeling v_k with the concatenation of the labels of the nodes $v_1, \dots, v_k$

Standard Trie → Compressed Trie Example



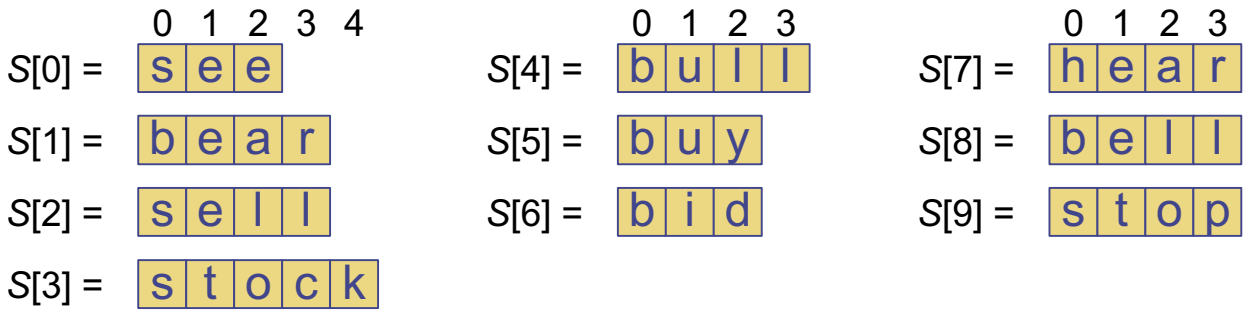
Properties of Compressed Tries

- in a compressed trie T storing a collection S of s strings from an alphabet of size $|\Sigma|$
  - Every internal node of T has at least two children and at most $|\Sigma|$ children
  - T has s leaves
  - The number of nodes in T is $O(s)$
    - Remember from the standard trie: the number of nodes of T was at most $n + 1$
    - Worst case: no two strings share a common, non-empty prefix – i.e. except for the root, all internal nodes have only one child
    - The worst-case scenario corresponds then to a compressed trie where all the chains from the root are redundant - and are therefore compressed – leading to $O(s)$ nodes

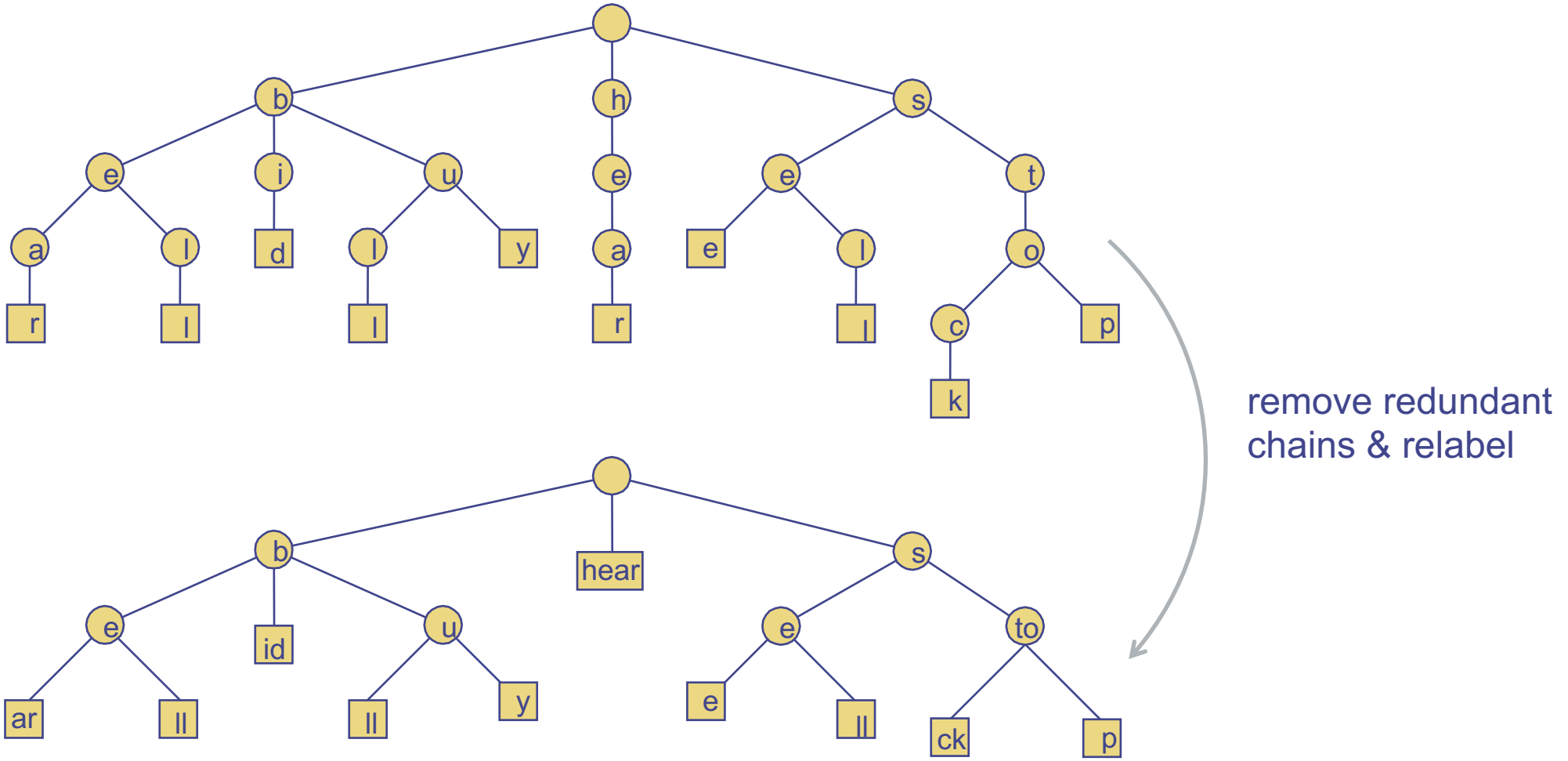
Compact Representation

- A compressed trie stores the node labels, which can be large (and have redundancy)
- Does the compression of the paths really provide an advantage?
- Yes, when using the compressed trie as an auxiliary index structure – not storing the actual characters, only indices into an already stored collection of strings

Compact Representation - Example

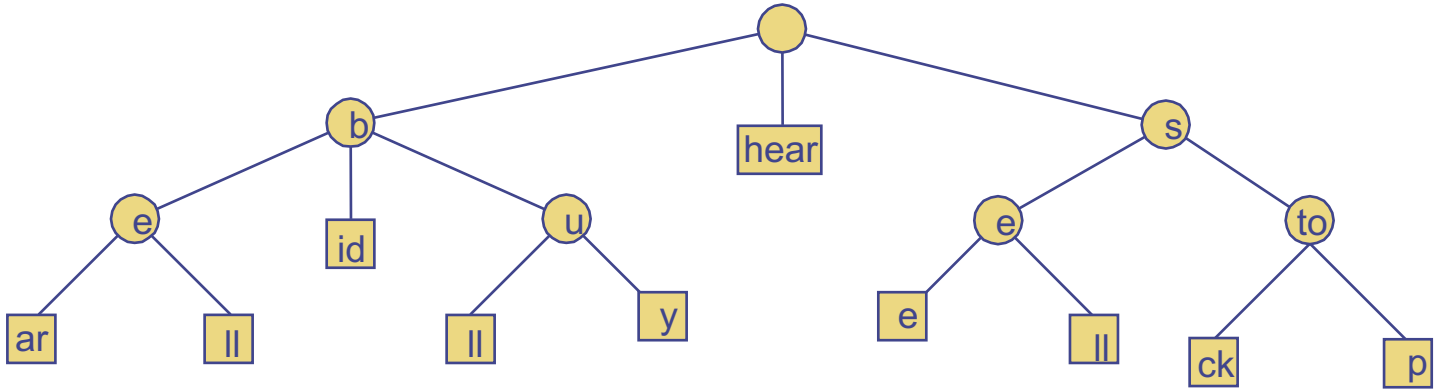


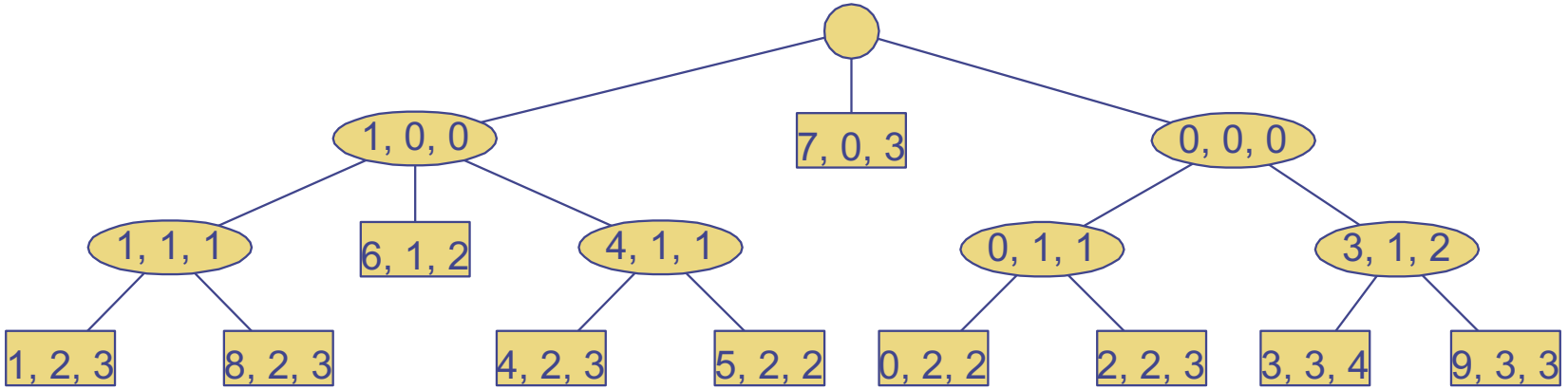
Compact Representation – Example (cont'd)



Compact Representation – Example (cont'd)

	0	1	2	3	4
$S[0]$ =	s	e	e		
$S[1]$ =	b	e	a	r	
$S[2]$ =	s	e	l	l	
$S[3]$ =	s	t	o	c	k
$S[4]$ =	b	u	l	l	
$S[5]$ =	b	u	y		
$S[6]$ =	b	i	d		
$S[7]$ =	h	e	a	r	
$S[8]$ =	b	e	l	l	
$S[9]$ =	s	t	o	p	





Compact representation gains

- Using the indexing method the total space required for storing the trie is reduced from $O(n)$ – where n is the total length of the strings in S - to $O(s)$ – where s is the number of strings in S
- The strings themselves must be stored in an additional structure, but we reduce the space required for the trie - which we need for searching
- Searching in a compressed trie is not necessarily faster than in a standard trie
  - Every character of the searched string must be compared with the character labels while traversing the trie
  - Labels can be multi-character

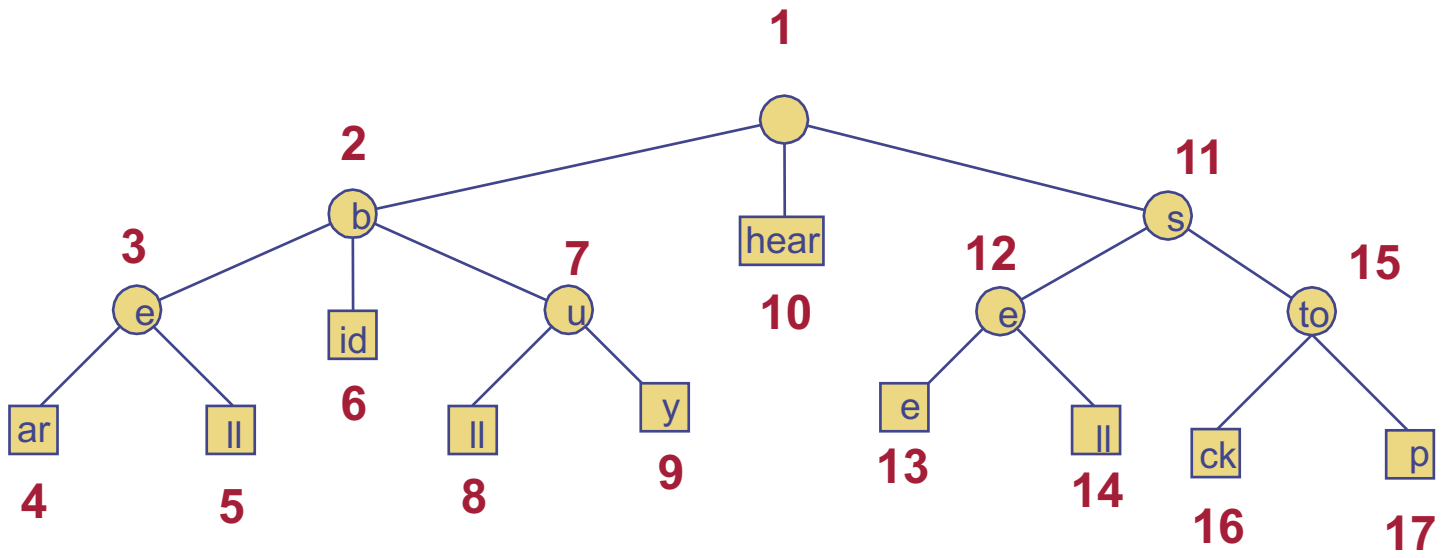
Tree Traversal

- to compress the nodes, one would need to traverse the trie (which is a tree)
- a traversal visits the nodes of a tree in a systematic manner
- types of traversals for n-ary trees
  - preorder traveral
  - postorder traversal

Preorder Traversal

- in preorder traversal, a node is visited before its descendants

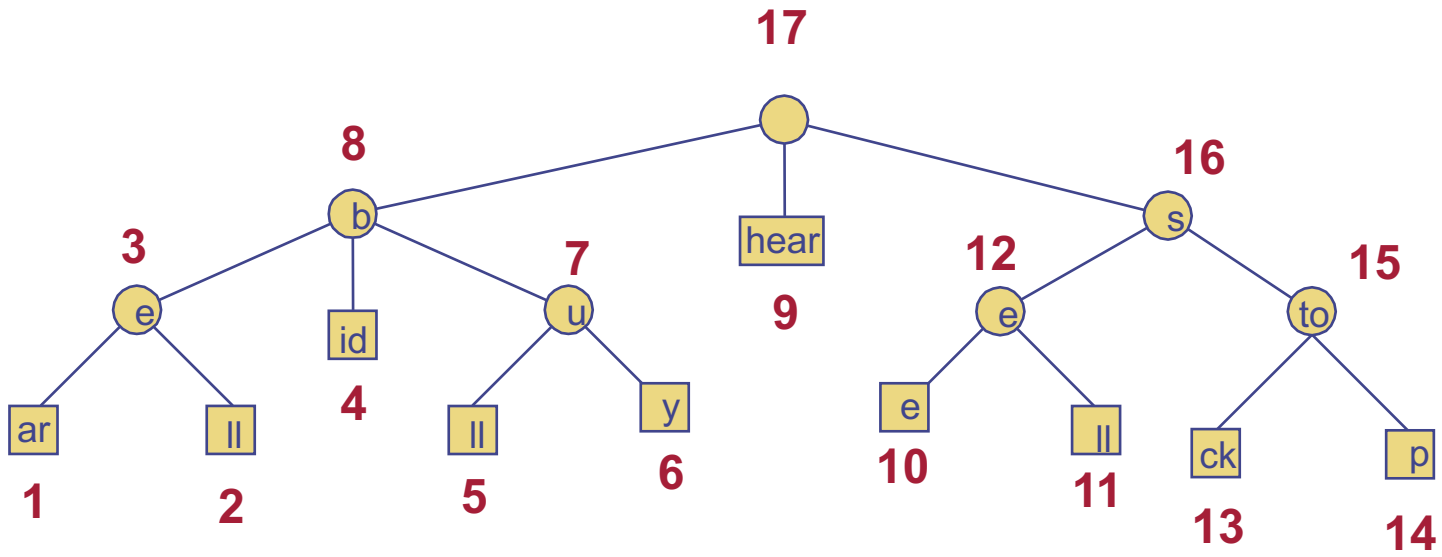
Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



Postorder Traversal

- in postorder traversal, a node is visited after its descendants

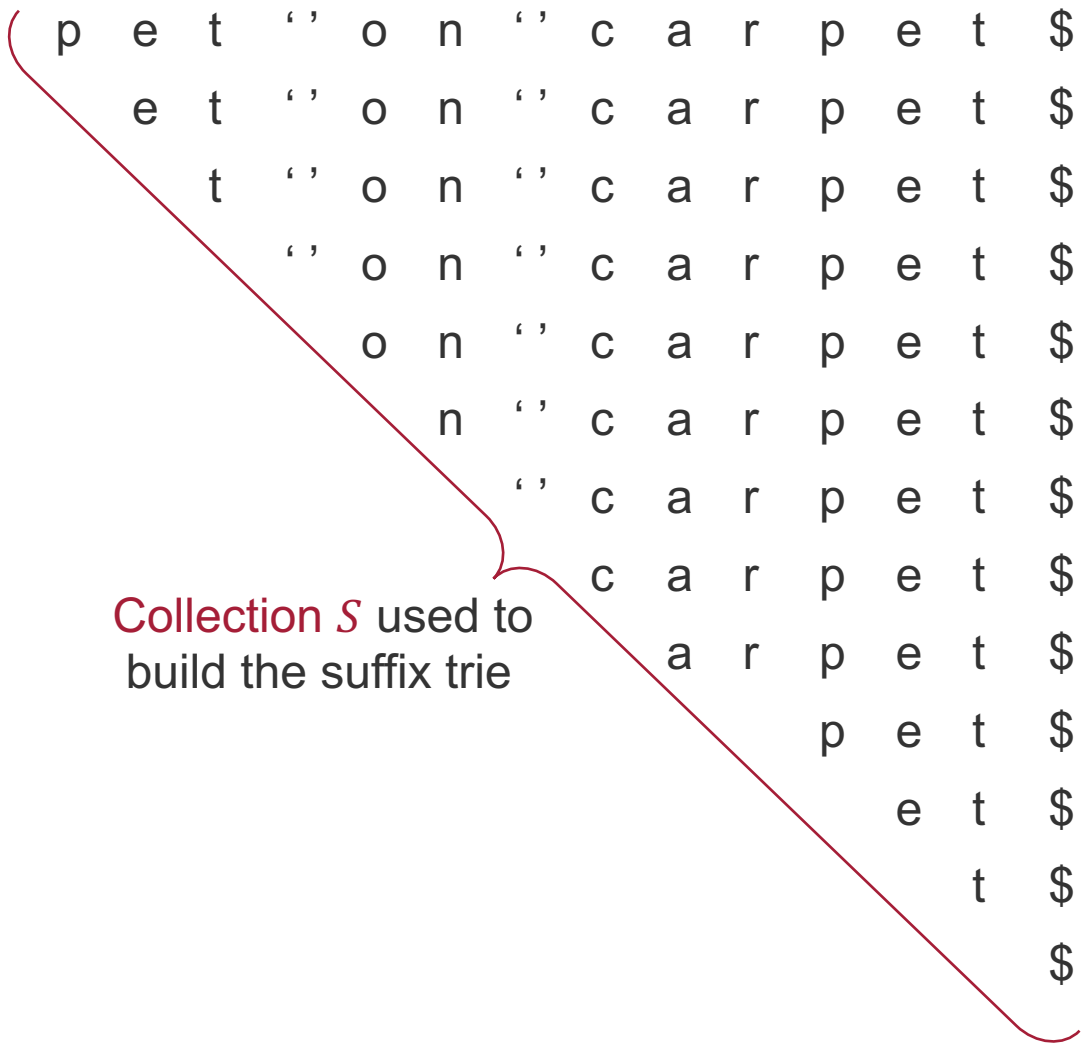
Algorithm *postOrder(v)*
  for each child *w* of *v*
    postOrder (*w*)
  visit(*v*)



Suffix Tries

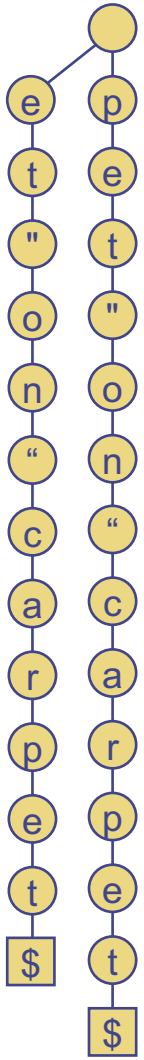
Suffix Tries

- The strings in the collection S are all suffixes of a string X
- Called a *suffix trie*, *suffix tree* or *position tree*
- Example X = *pet on carpet*
- S was supposed to be a set of s strings from alphabet Σ such that no string in S is a prefix for another string
- Add an artificial character \$, which is not part of Σ, at the end of each suffix
- For X of length n, we build a trie using the set of n strings $X[j:n]$, $j = 0 \ldots n-1$

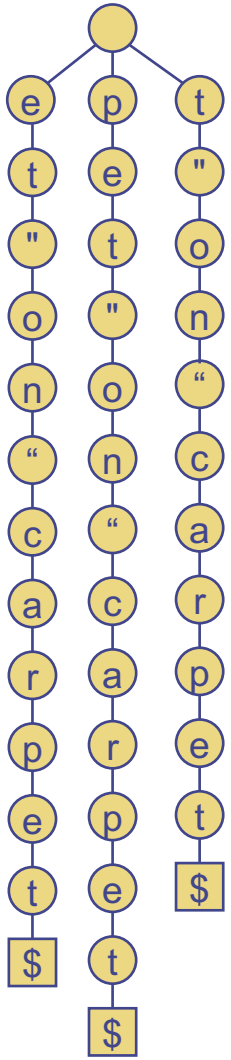


Suffix Trie Construction

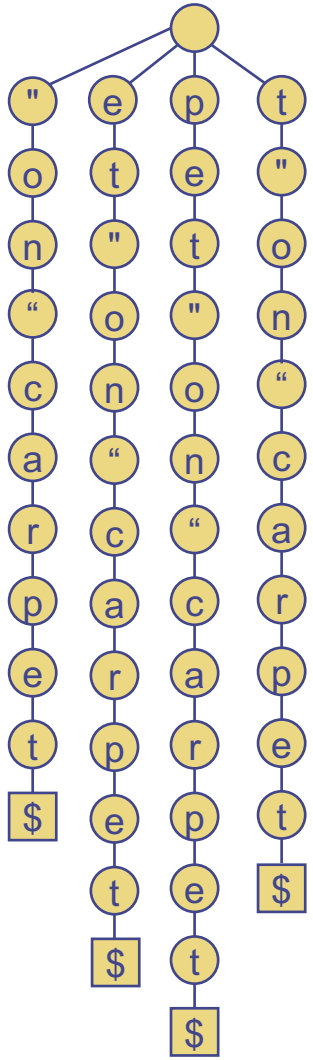
Suffix Trie Construction



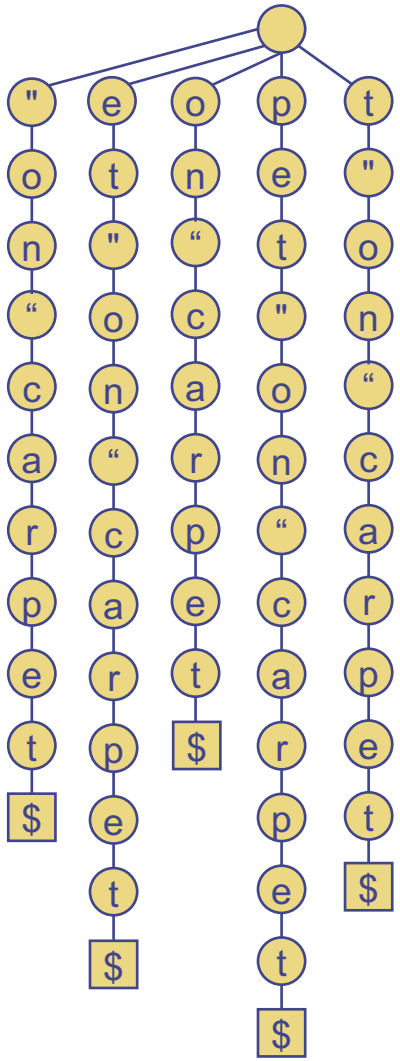
Suffix Trie Construction



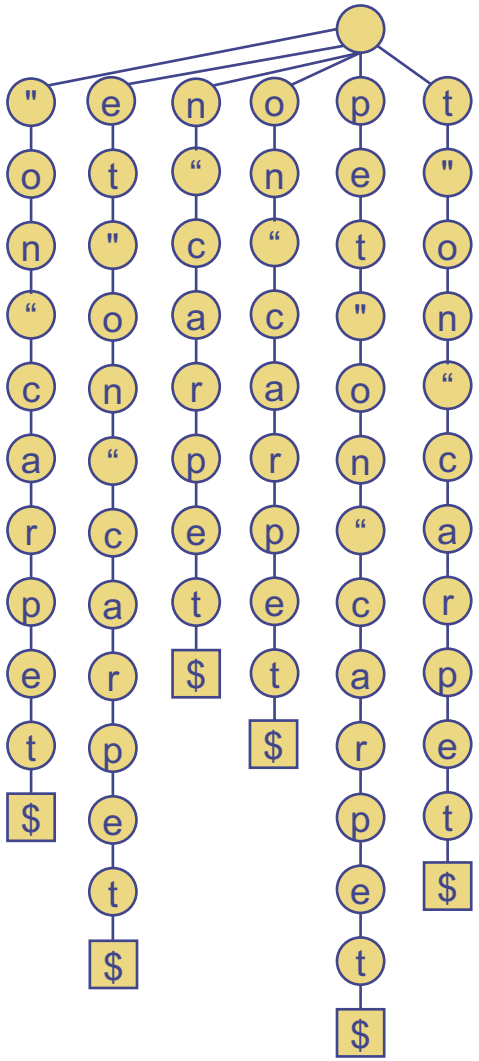
Suffix Trie Construction



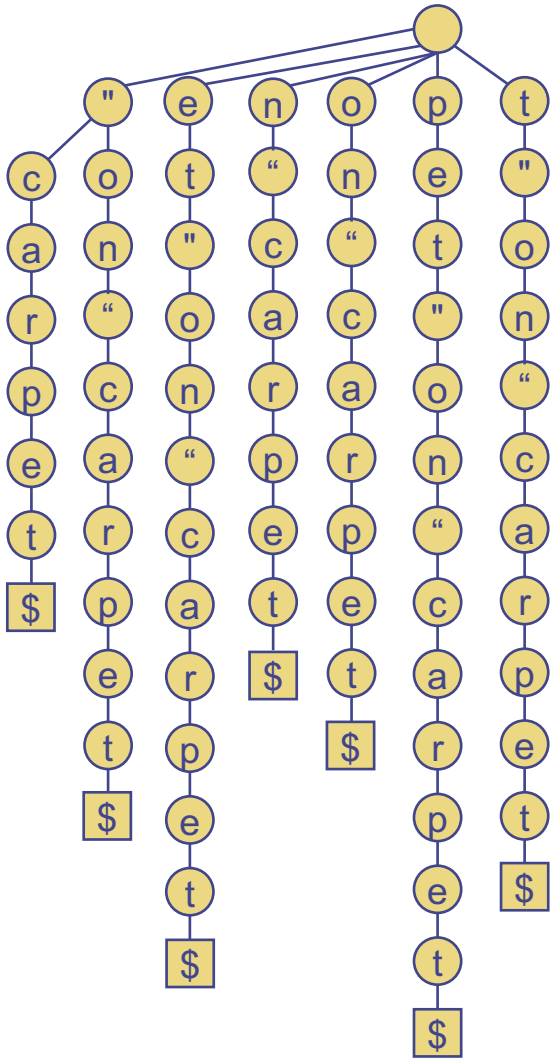
Suffix Trie Construction



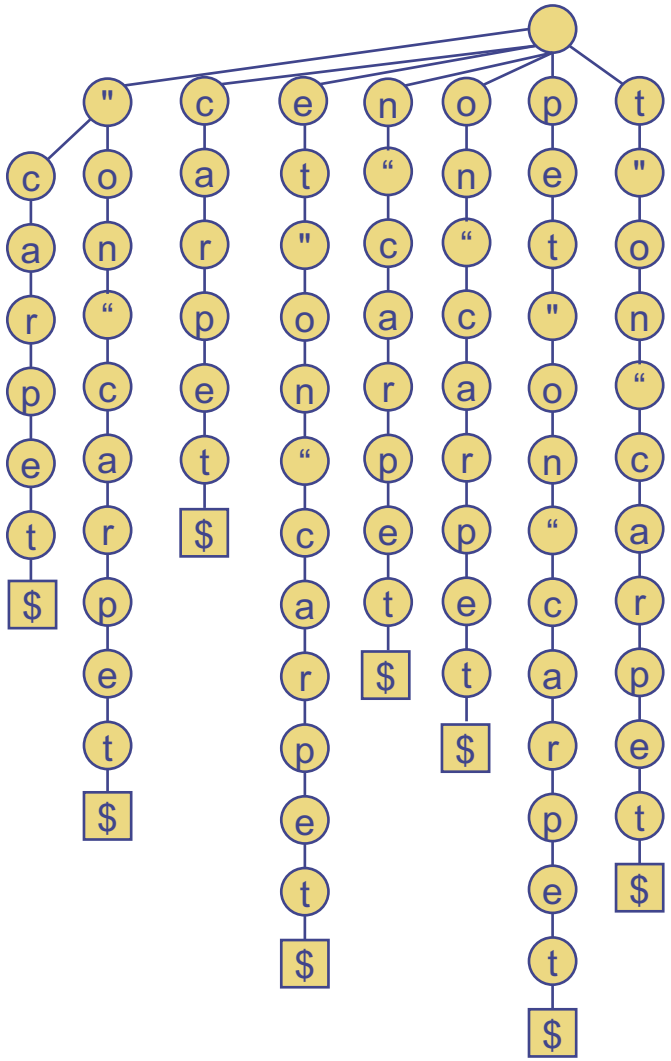
Suffix Trie Construction



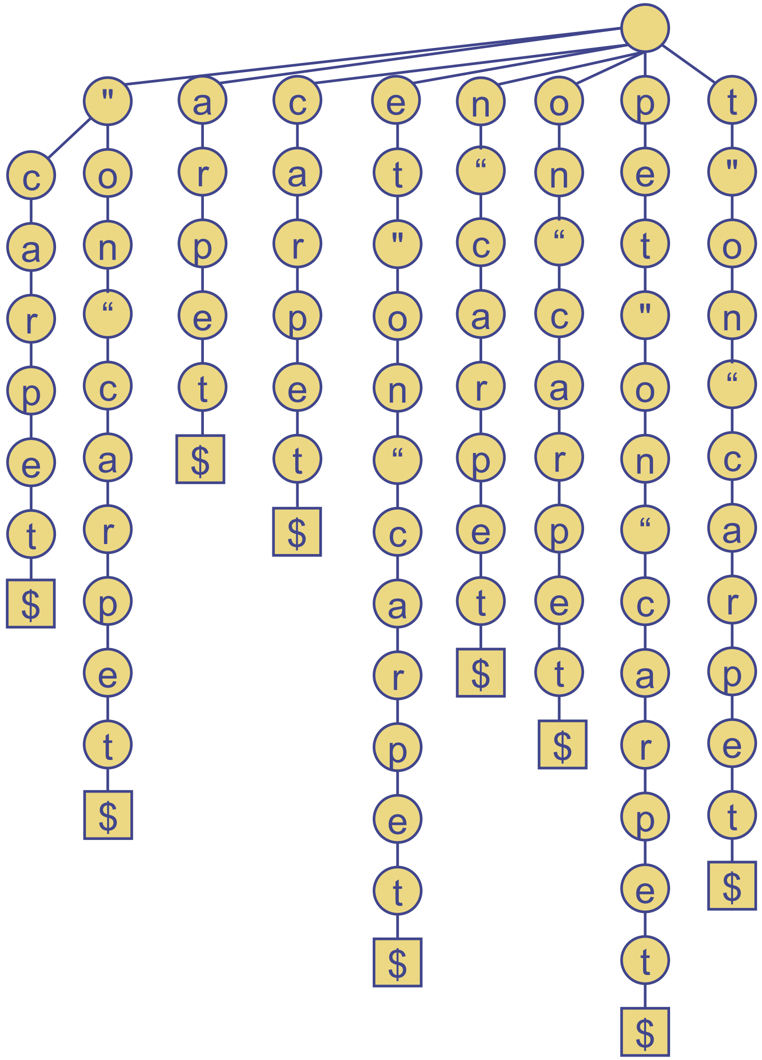
Suffix Trie Construction



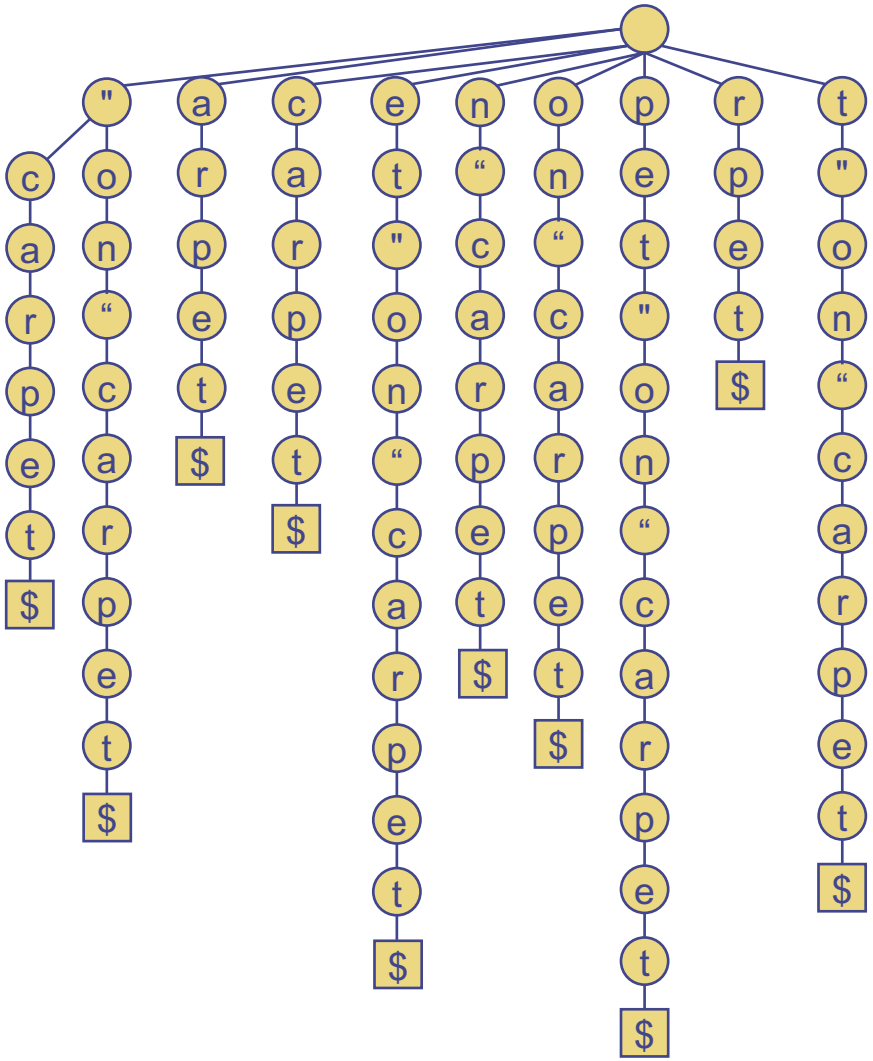
Suffix Trie Construction



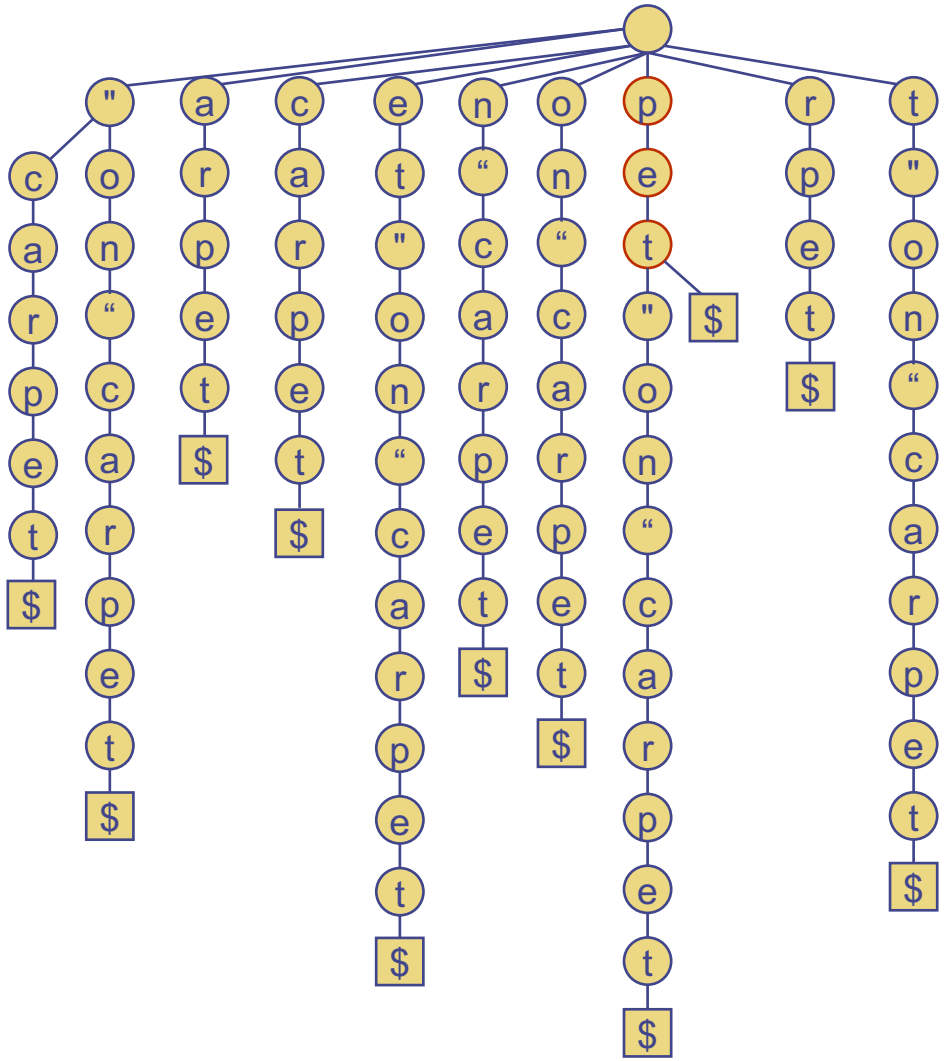
Suffix Trie Construction



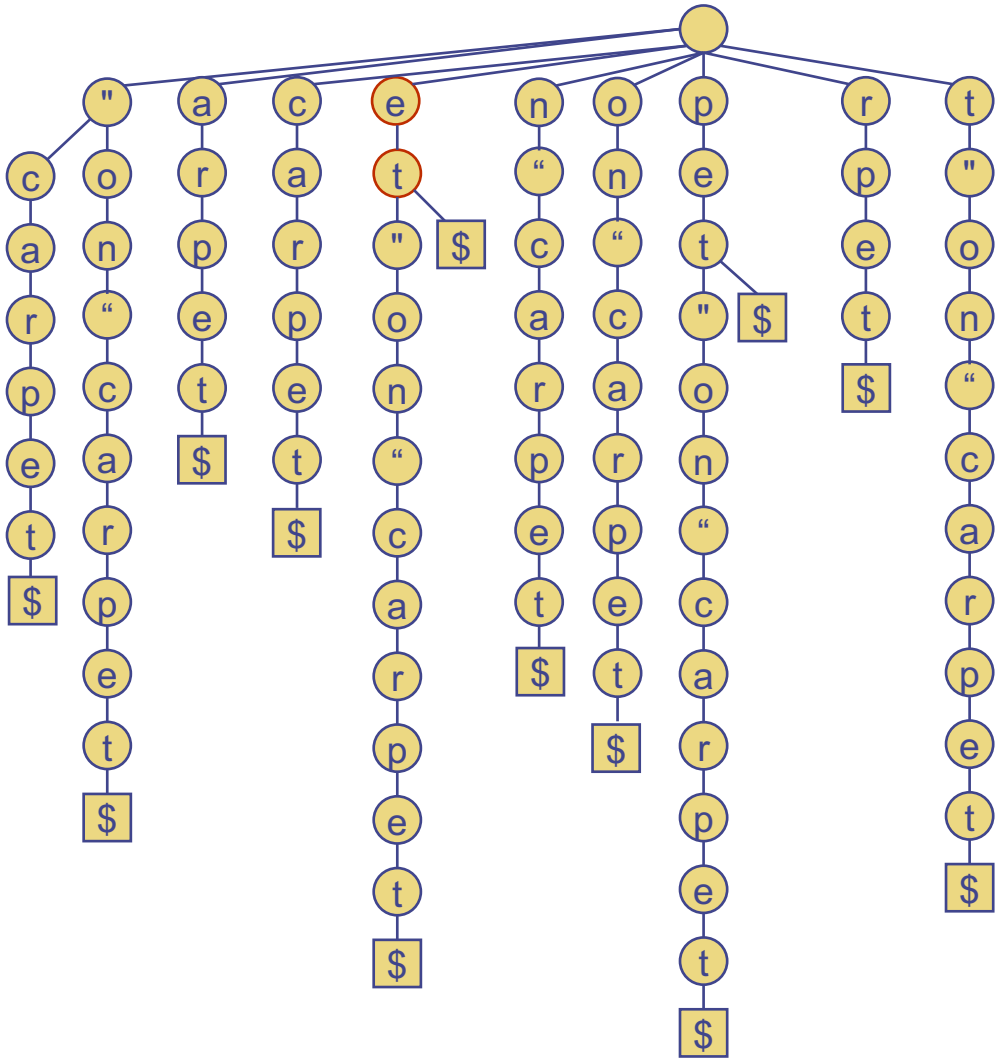
Suffix Trie Construction



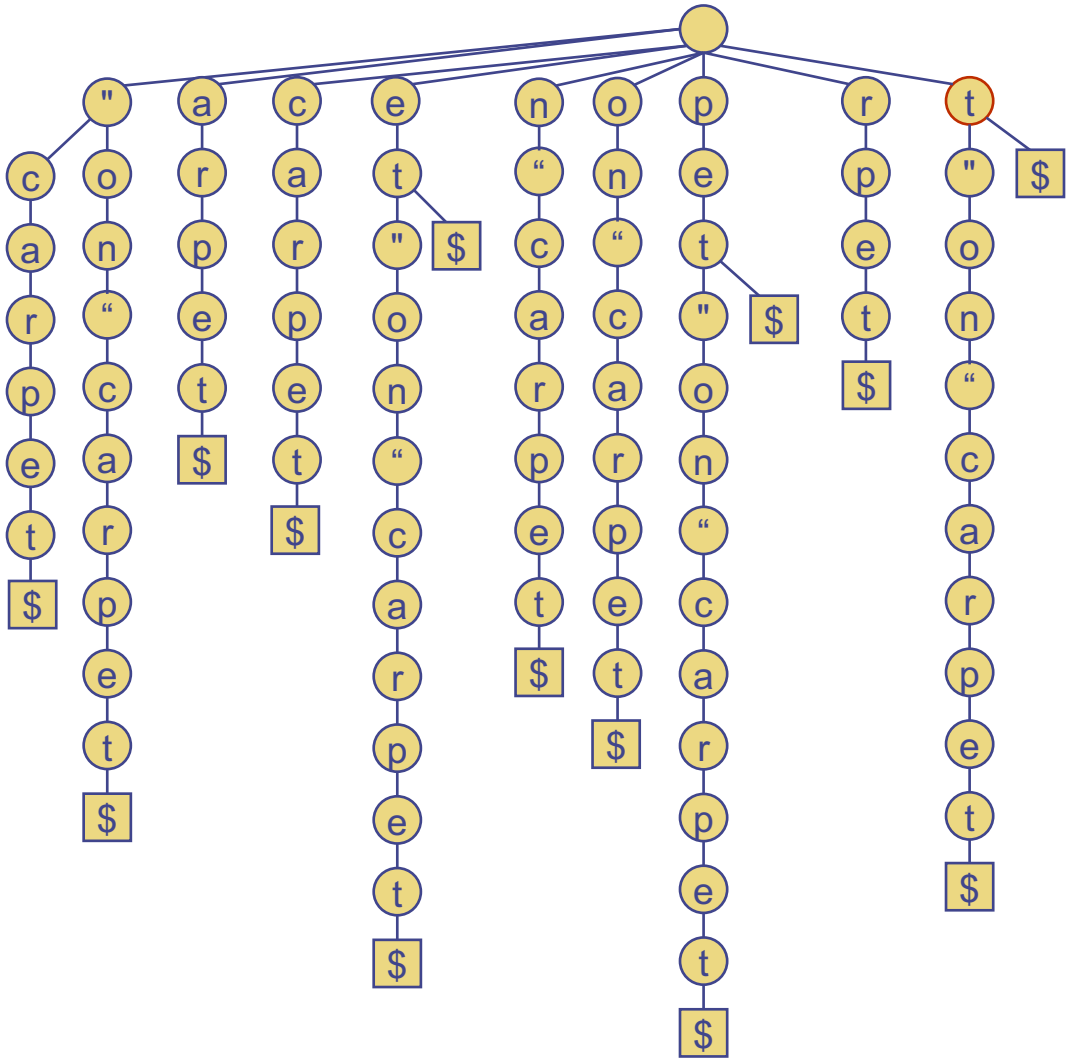
Suffix Trie Construction



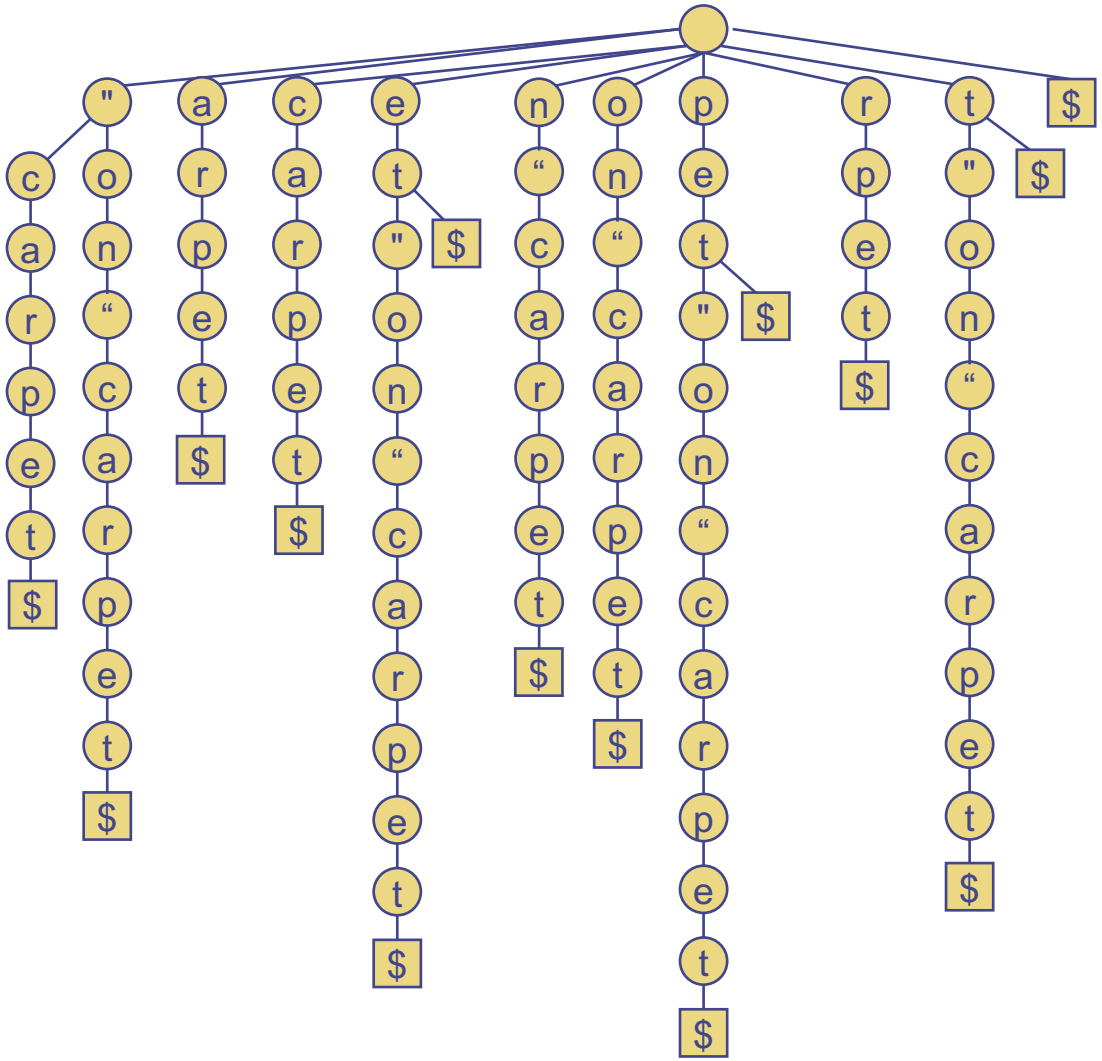
Suffix Trie Construction



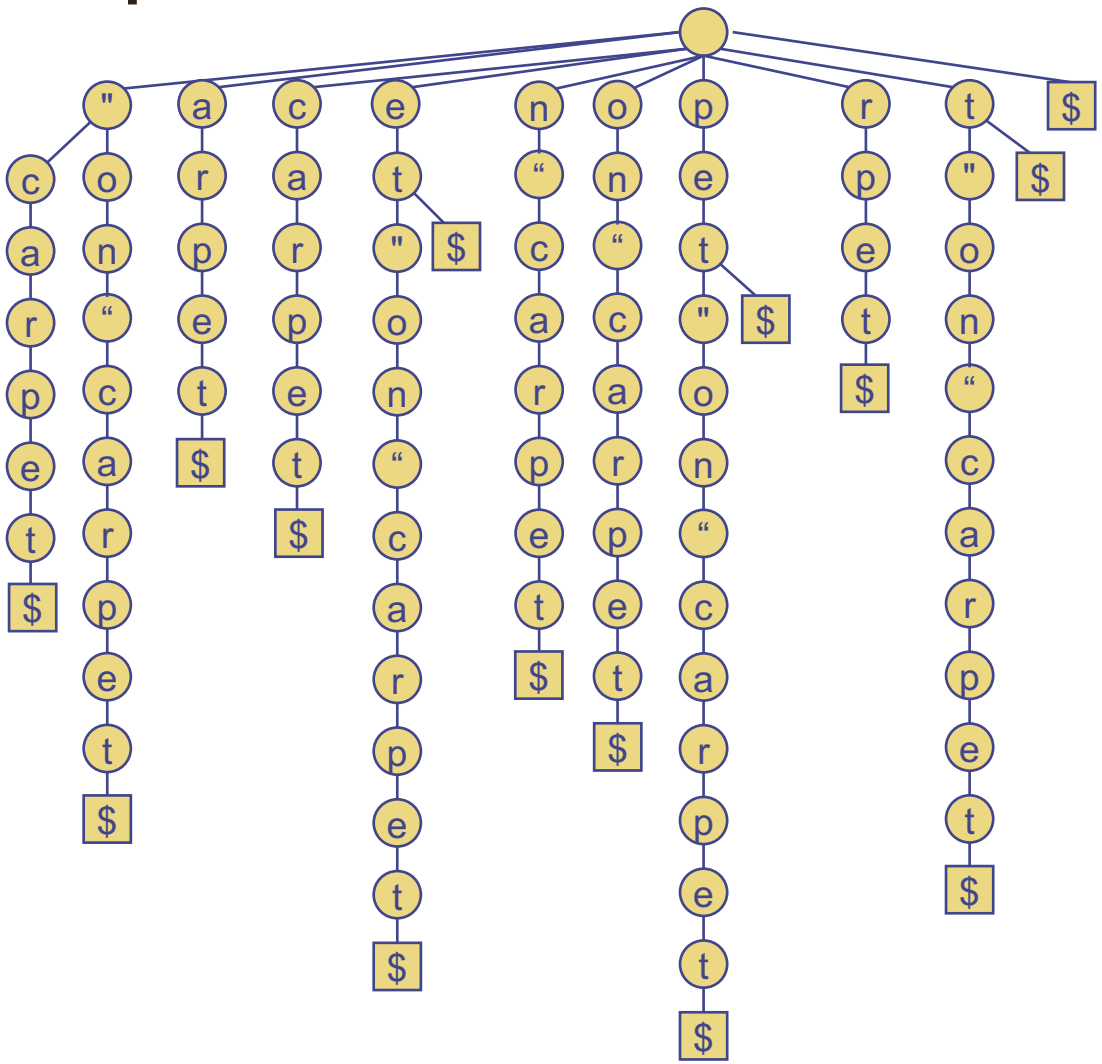
Suffix Trie Construction



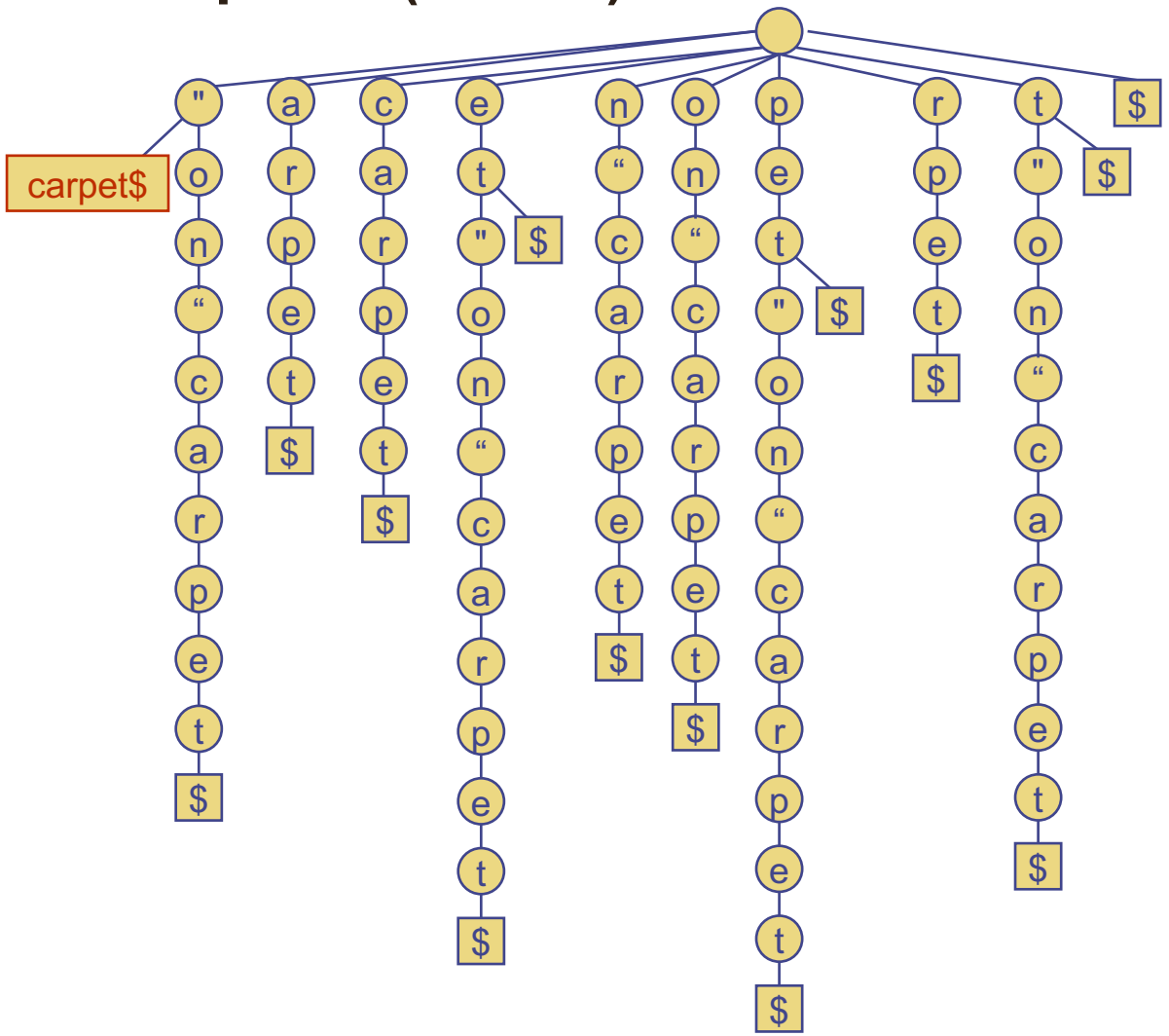
Suffix Trie Construction



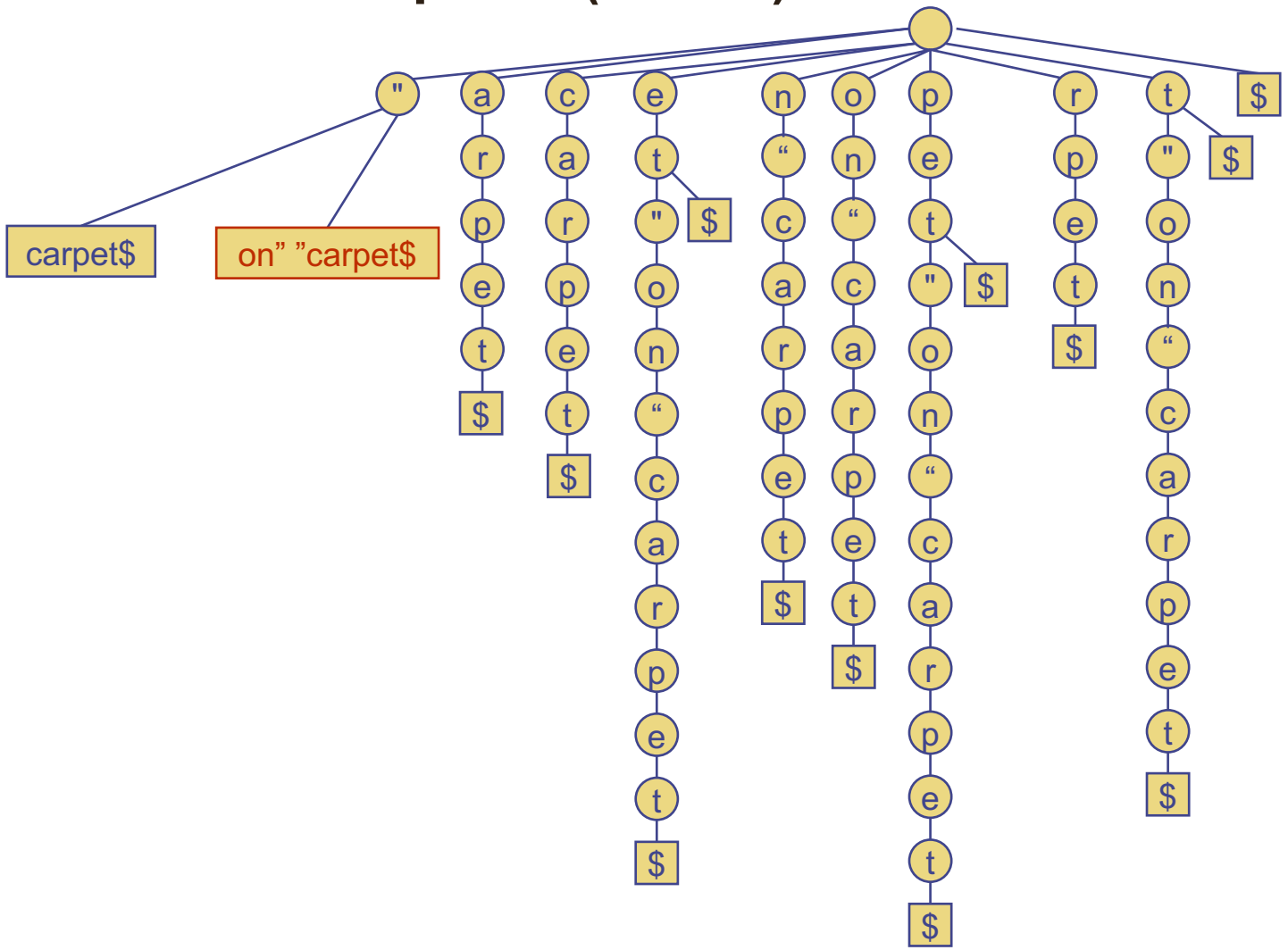
Suffix Trie: Compress



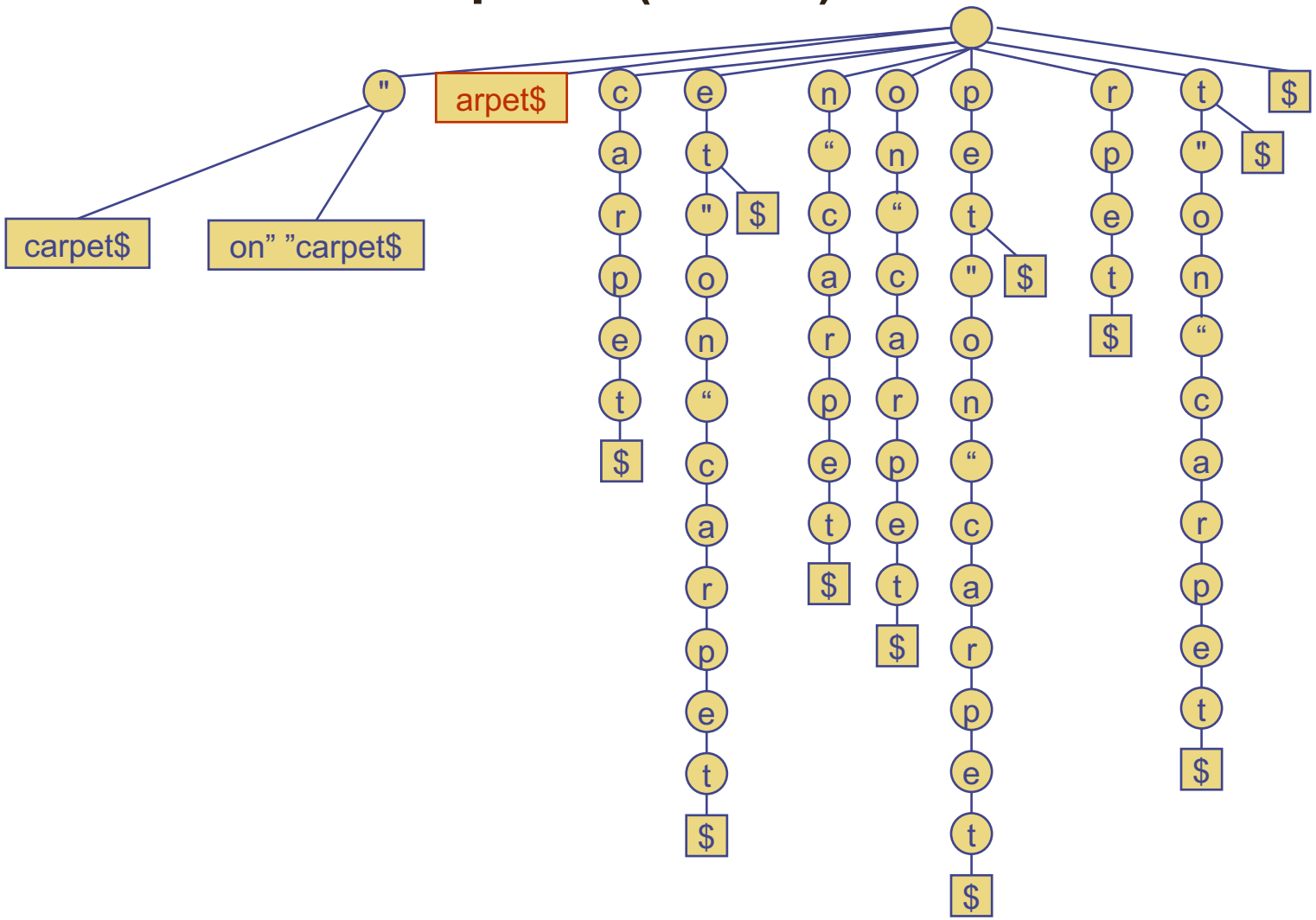
Suffix Trie: Compress (cont'd)



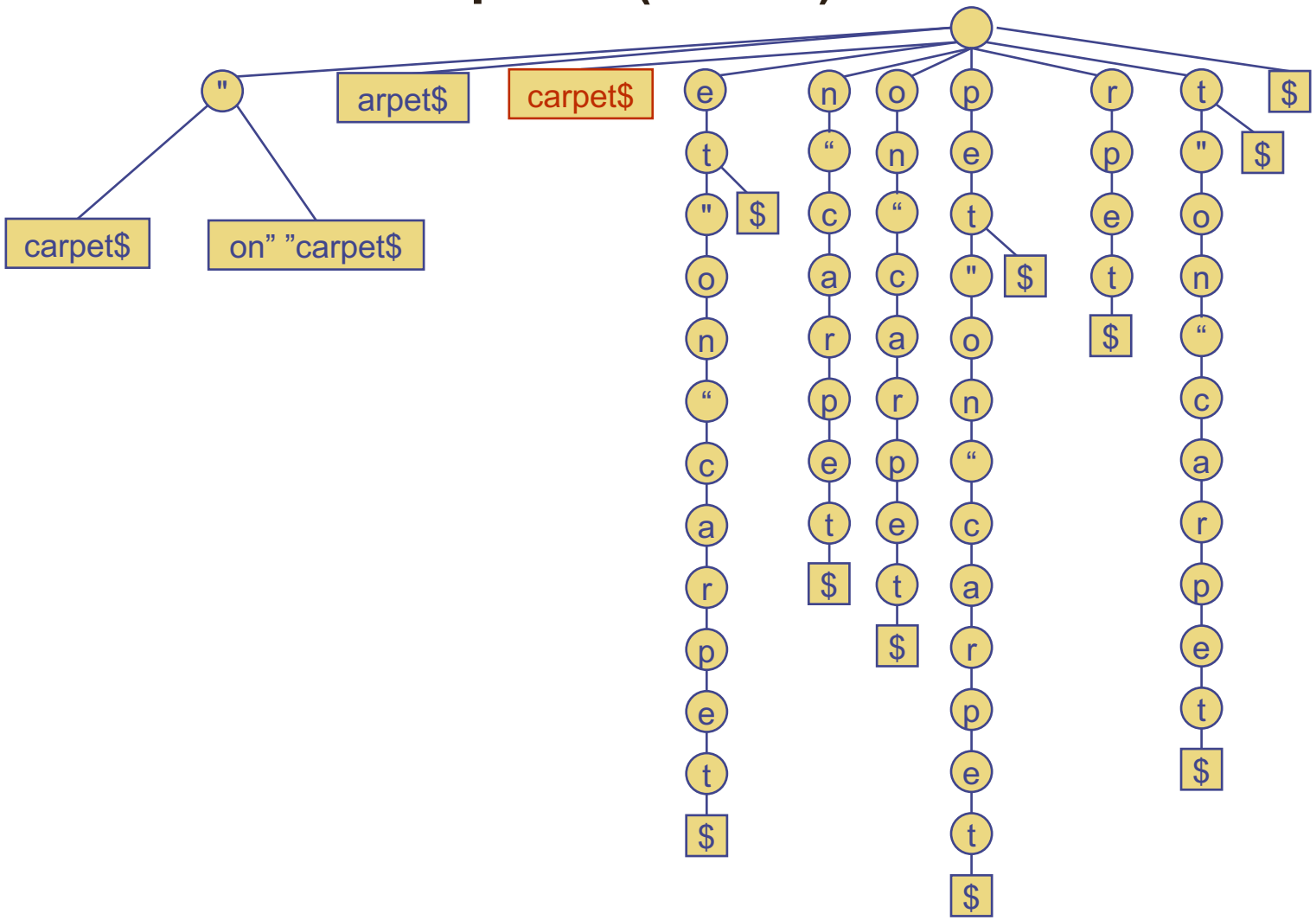
Suffix Trie: Compress (cont'd)



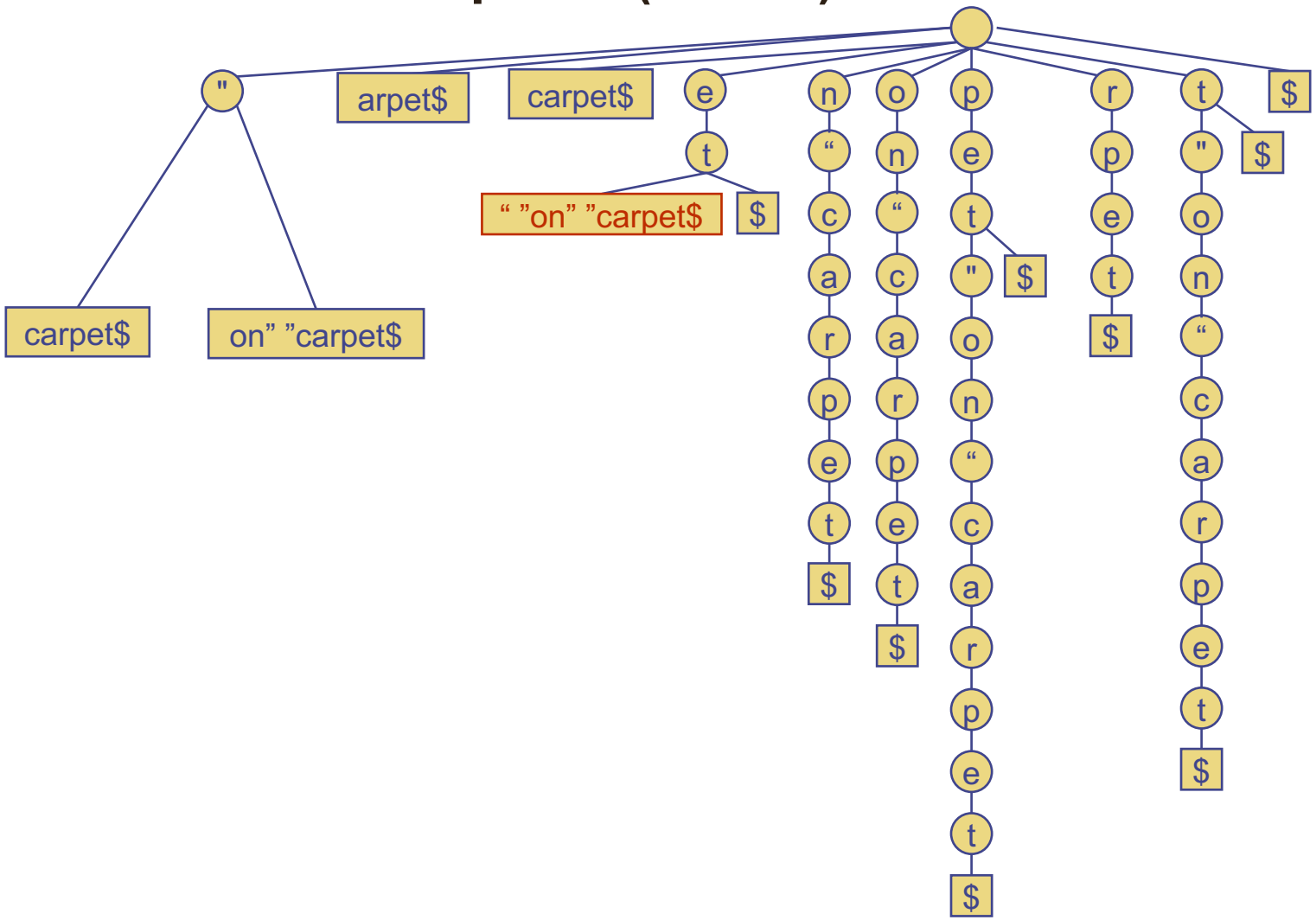
Suffix Trie: Compress (cont'd)



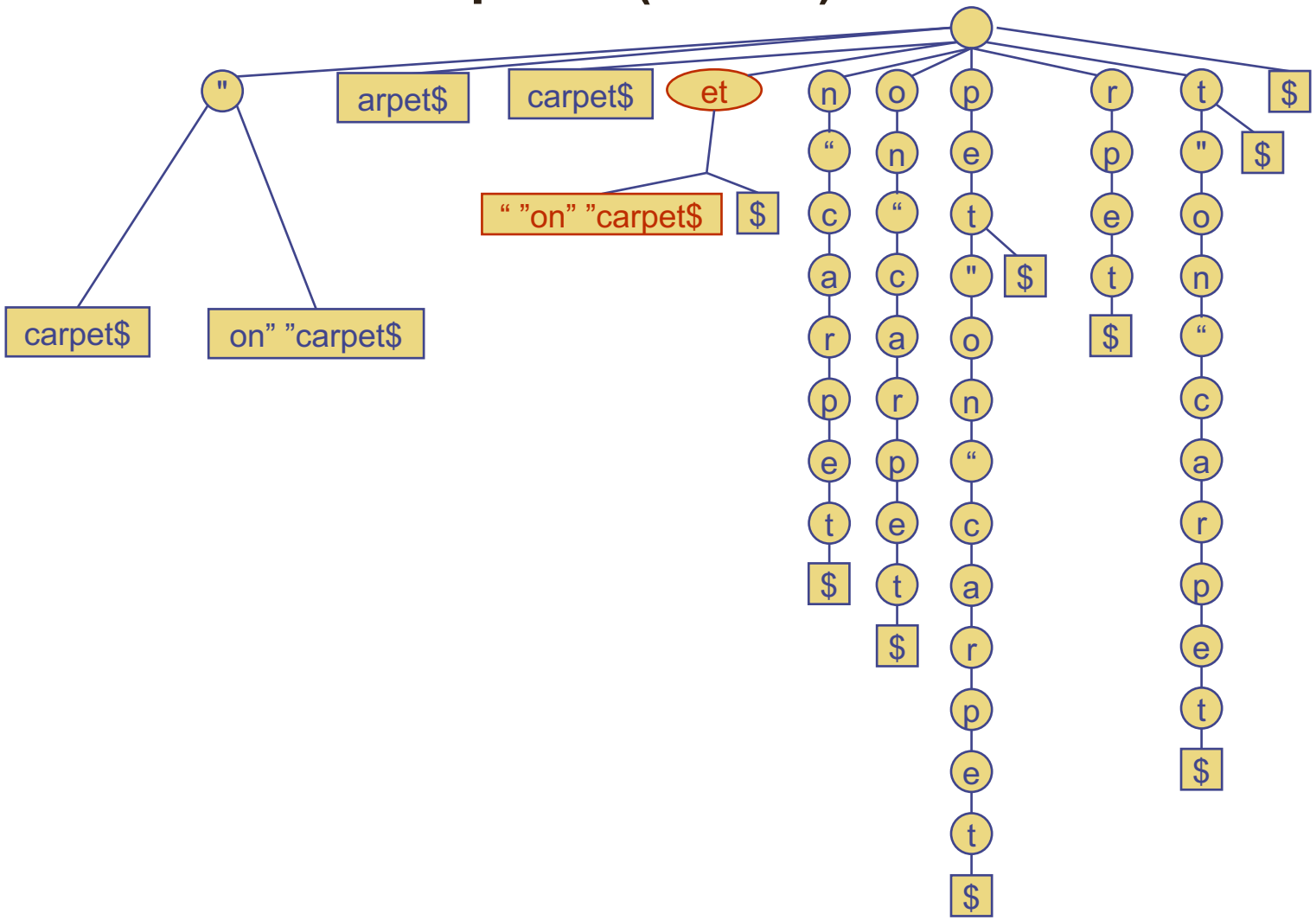
Suffix Trie: Compress (cont'd)



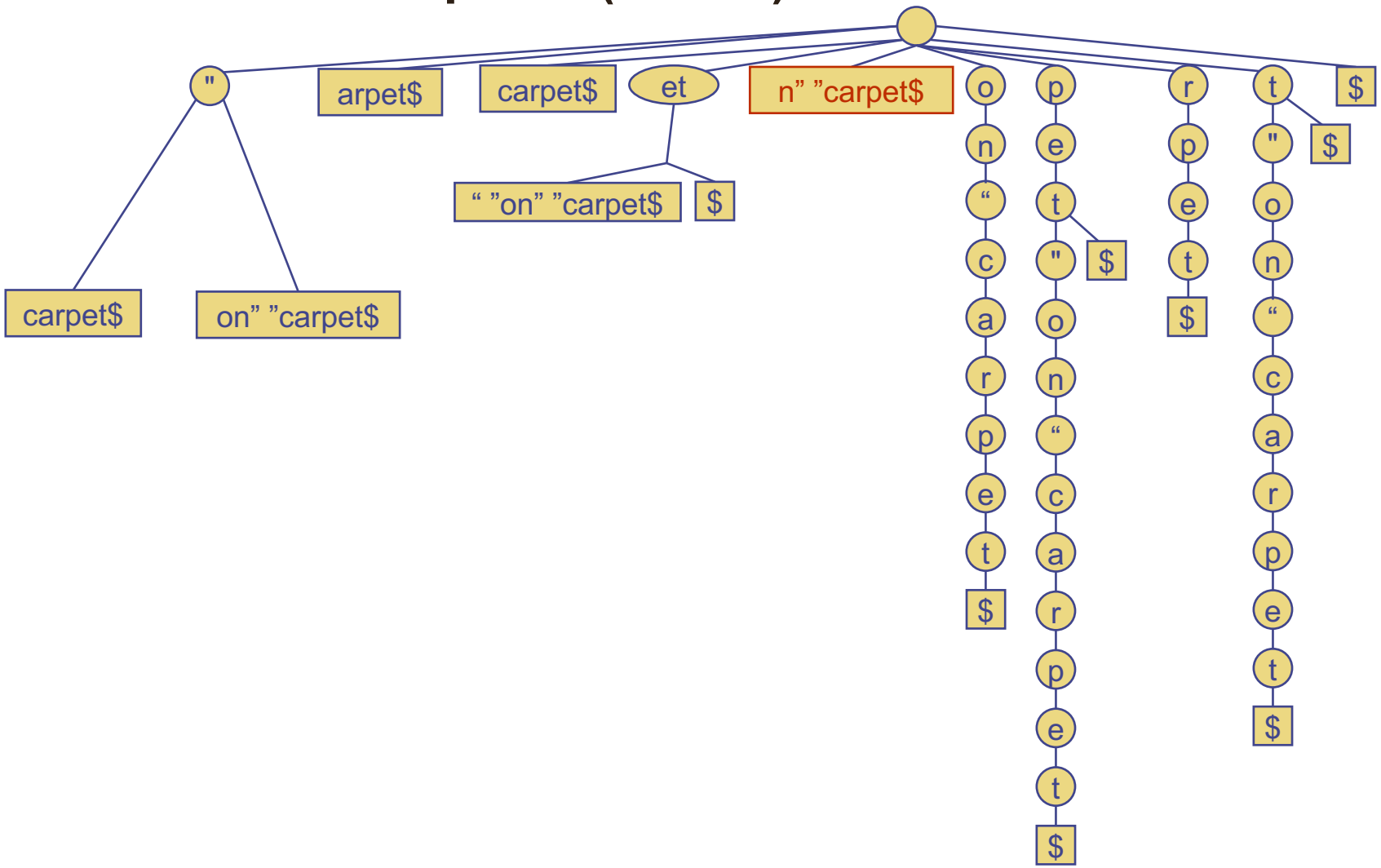
Suffix Trie: Compress (cont'd)



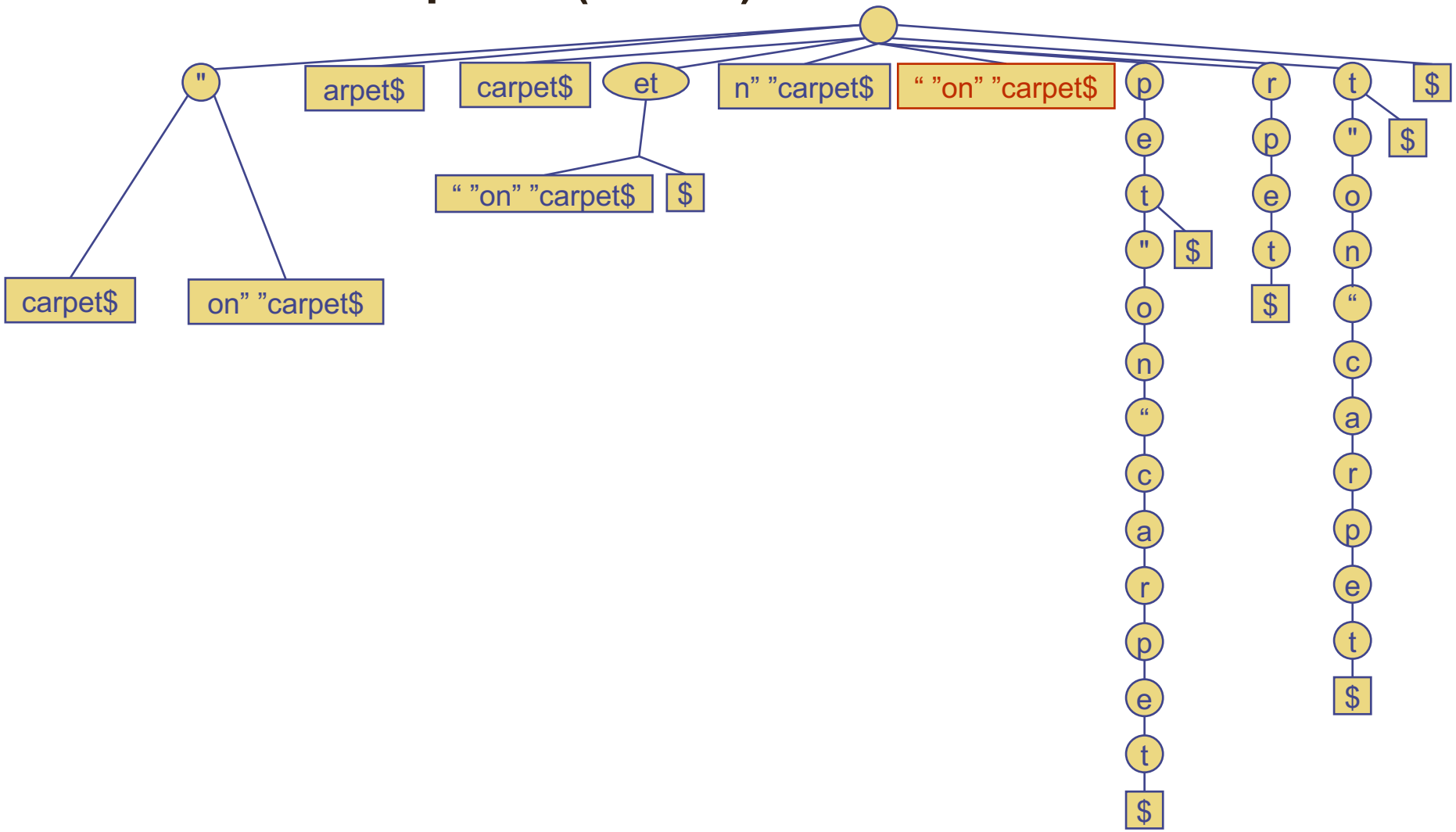
Suffix Trie: Compress (cont'd)



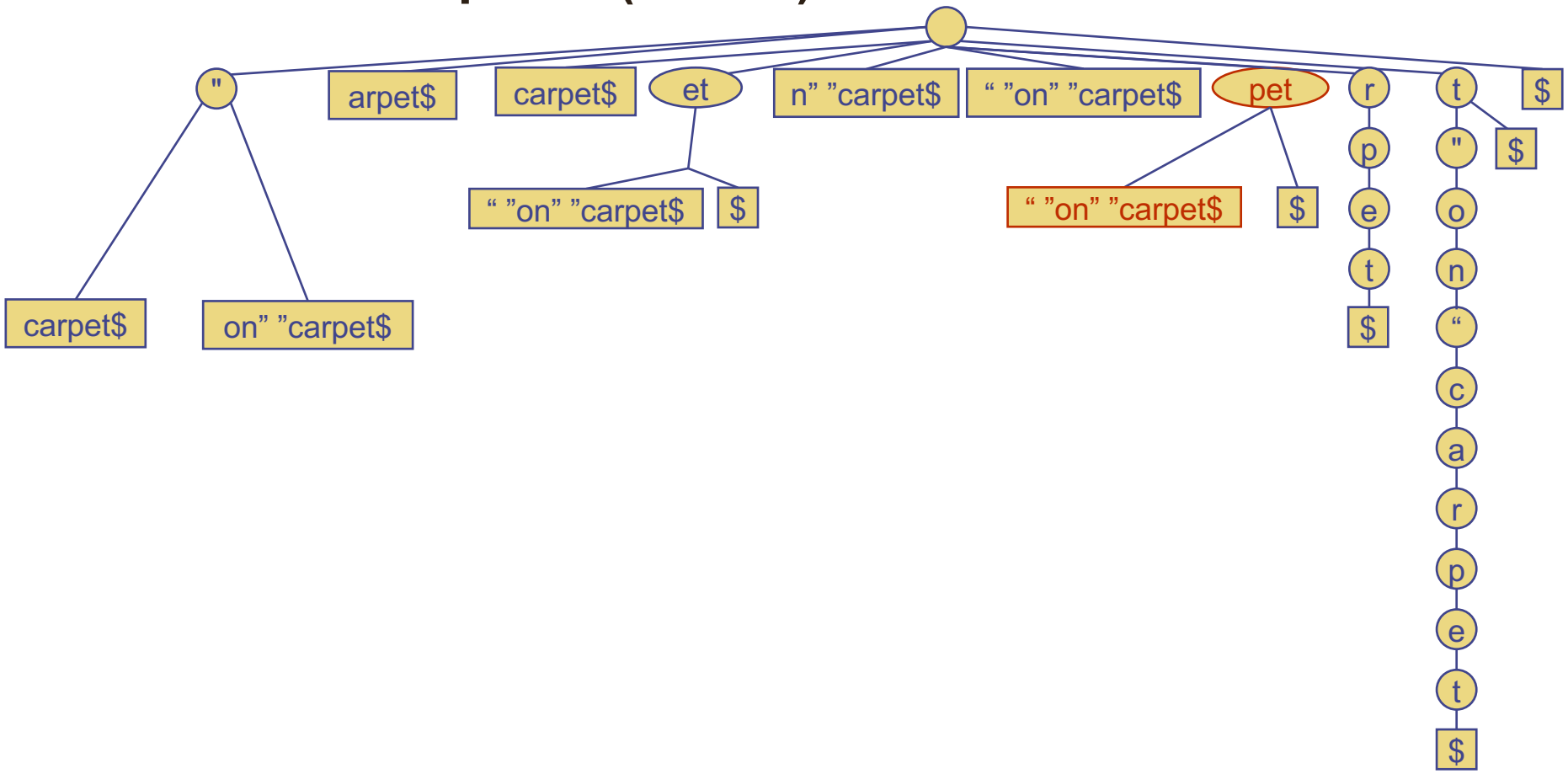
Suffix Trie: Compress (cont'd)



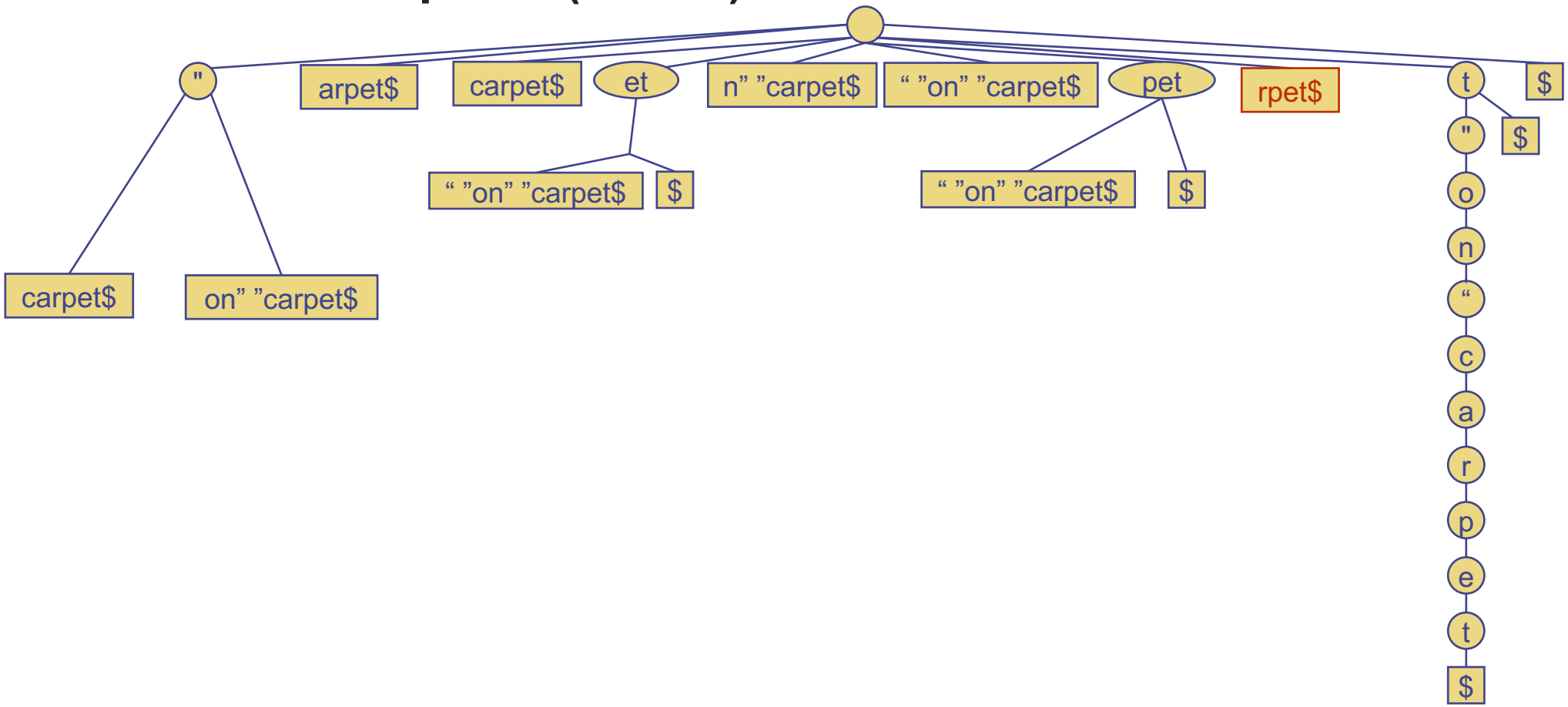
Suffix Trie: Compress (cont'd)



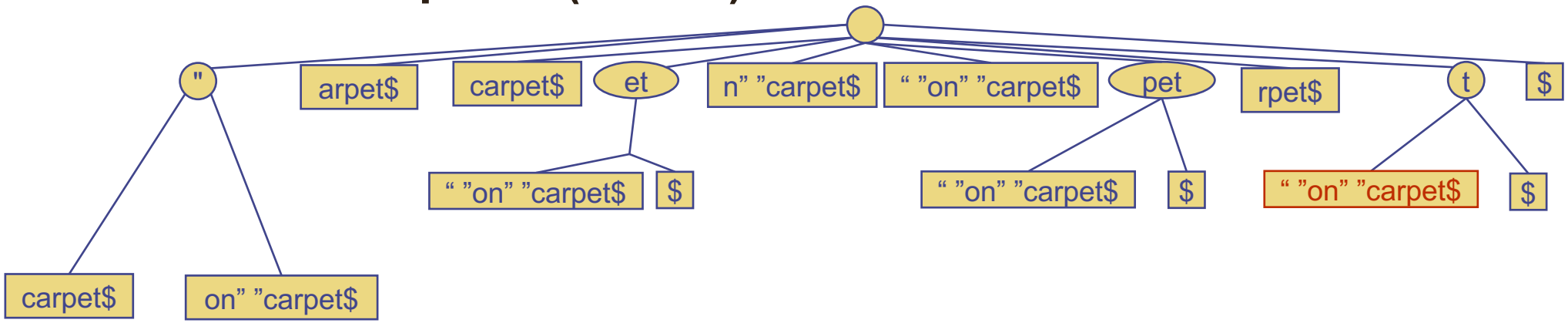
Suffix Trie: Compress (cont'd)



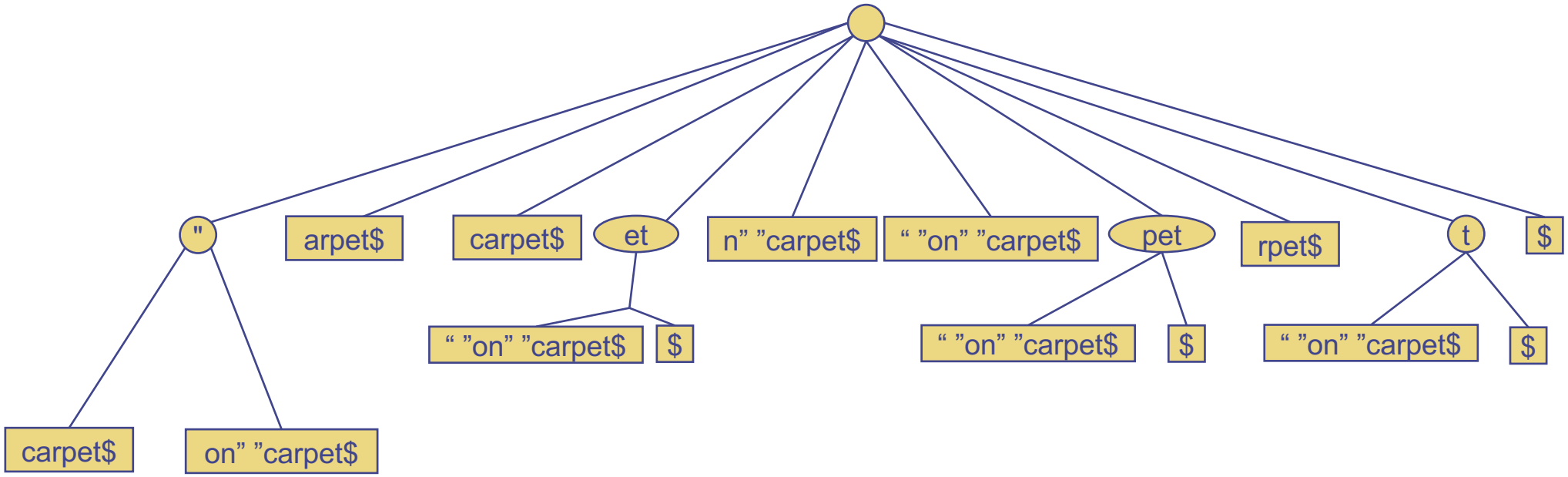
Suffix Trie: Compress (cont'd)



Suffix Trie: Compress (cont'd)

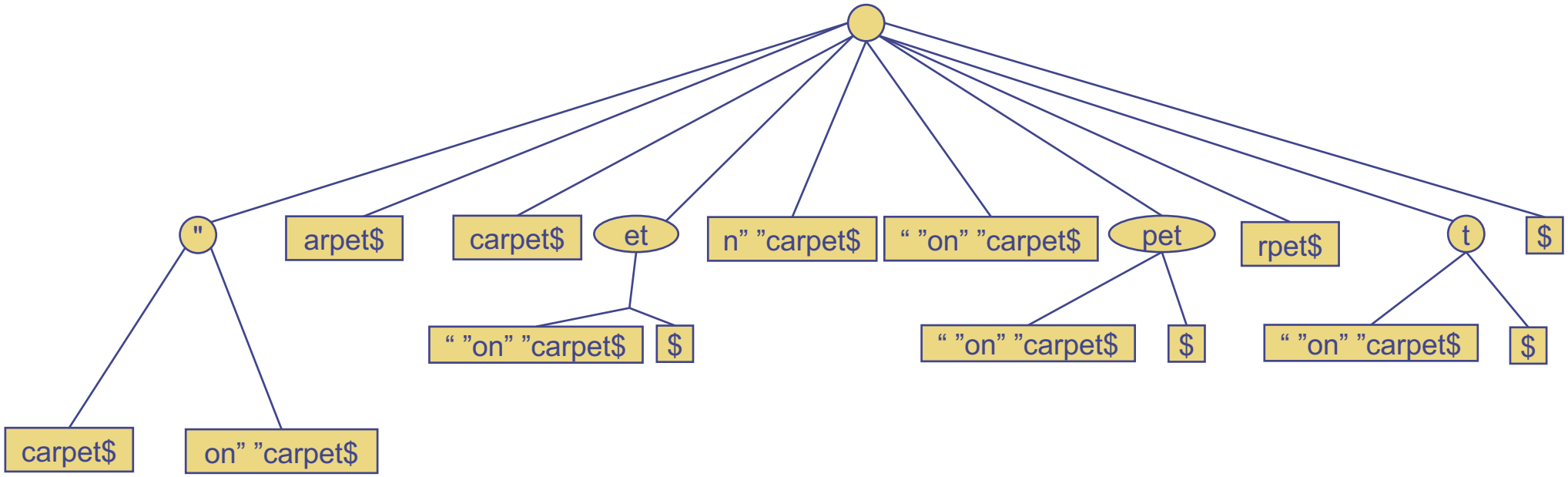


Suffix Trie: Index



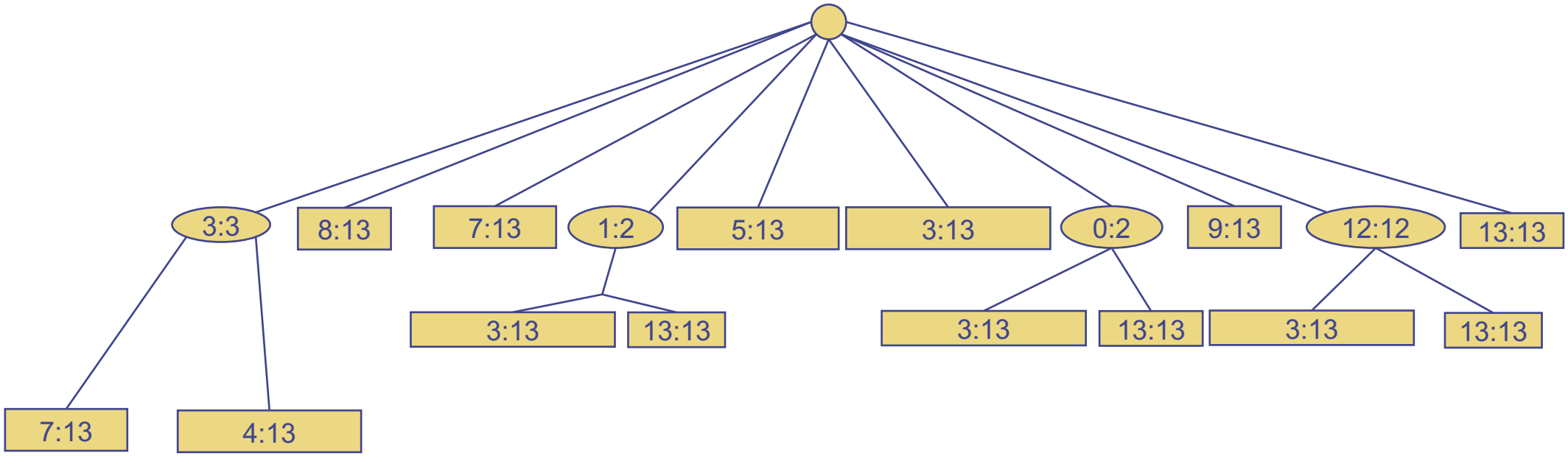
- Many substrings appear over and over again – impractical to store each label directly
- Instead, store only the indices in an array

Suffix Trie: Index (cont'd)



p	e	t	' '	o	n	' '	c	a	r	p	e	t	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Suffix Trie: Index (cont'd)



p	e	t	‘ ’	o	n	‘ ’	c	a	r	p	e	t	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Suffix Tries: Properties

- A suffix trie saves space compared to a standard trie, because of using compression
- The length of all the suffixes of a string X of length n is

$$n + (n-1) + (n-2) + \ \ldots + 2 + 1 = \frac{n(n+1)}{2}$$

- A standard trie uses $O(n^2)$ space to store all the suffixes of X
- A suffix trie (which is compressed) uses only $O(n)$ space
- A suffix trie can be constructed using the simple procedure described in $O(|\Sigma|n^2)$ time because the total length of the suffixes is quadratic in n
- For production use, there is are linear-time algorithms - $O(n)$ - for constructing suffix tries
  - P. Weiner. 1973. *Linear Pattern Matching Algorithms*.
  - E. Ukkonen. 1995. *On-line Construction of Suffix Trees*.

Thank you.