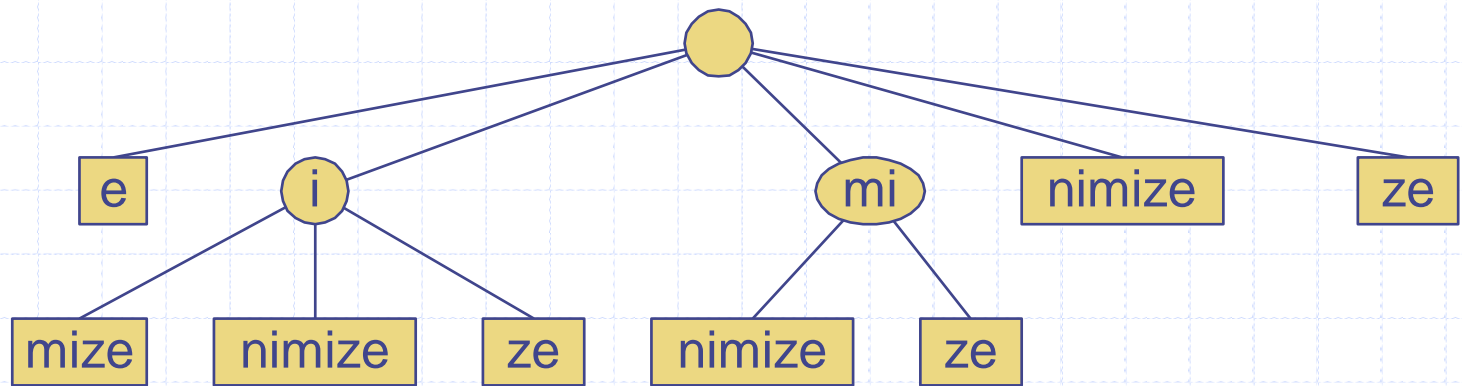


# Tries



# Outline

- ◆ Standard tries
- ◆ Compressed tries
- ◆ Suffix tries
- ◆ Huffman encoding tries

# Where does “trie” come from?

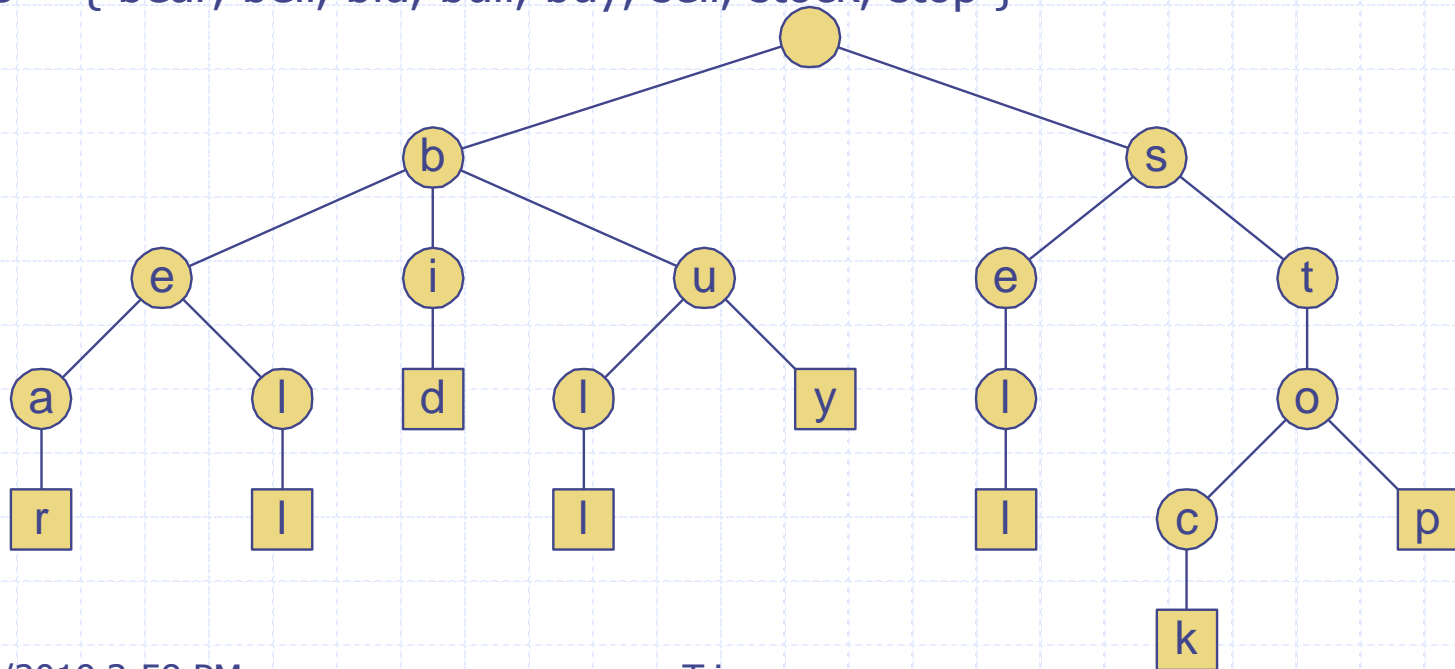
- ◆ From the word retrieval
- ◆ Introduced in the 1960's

# Preprocessing Strings

- ◆ Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
- ◆ If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
  - Thus do better than  $O(n+m)$  for text of size  $n$  and pattern of size  $m$
- ◆ A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A trie supports pattern matching queries in time proportional to the pattern size ( $\sim O(m)$ )!

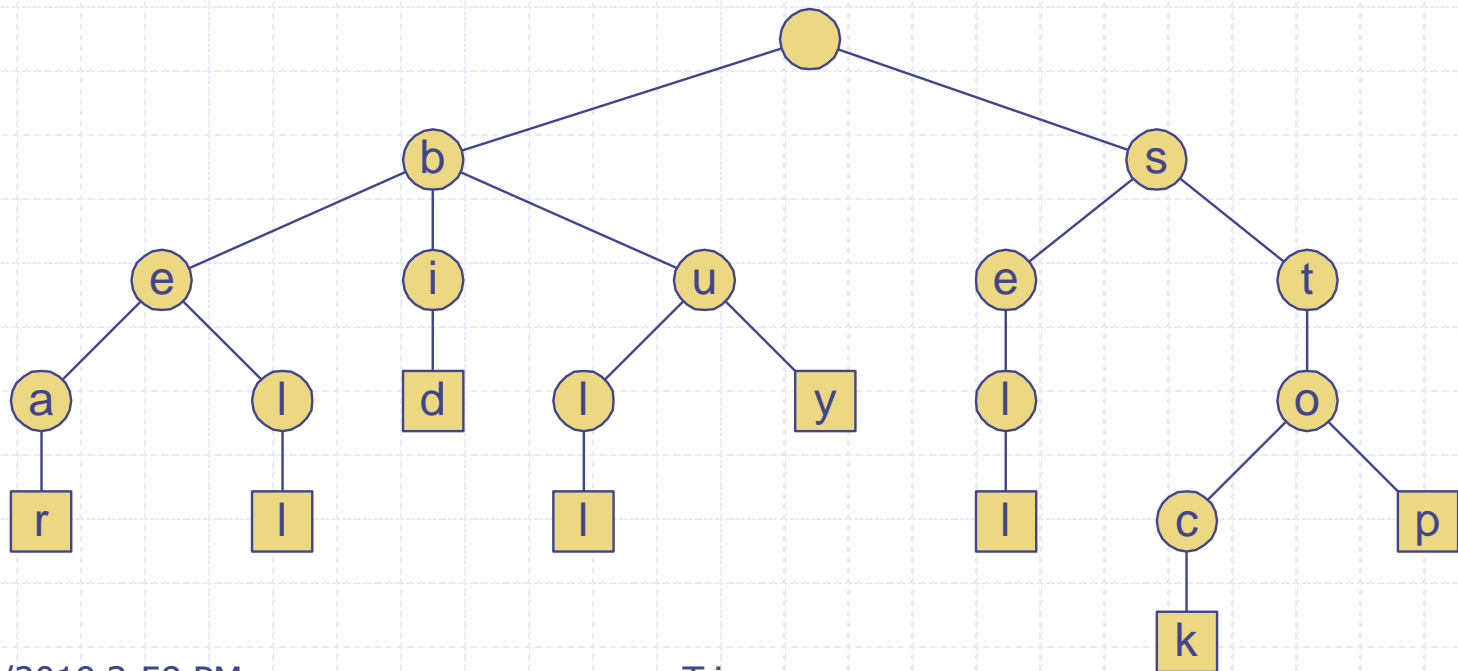
# Standard Trie

- ◆ The standard trie for a set of strings  $S$  is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of  $S$
- ◆ Example: standard trie for the set of strings  
 $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$



# Standard Trie

- ◆ What space does the trie use?
- ◆ What is the maximum height of the tree?



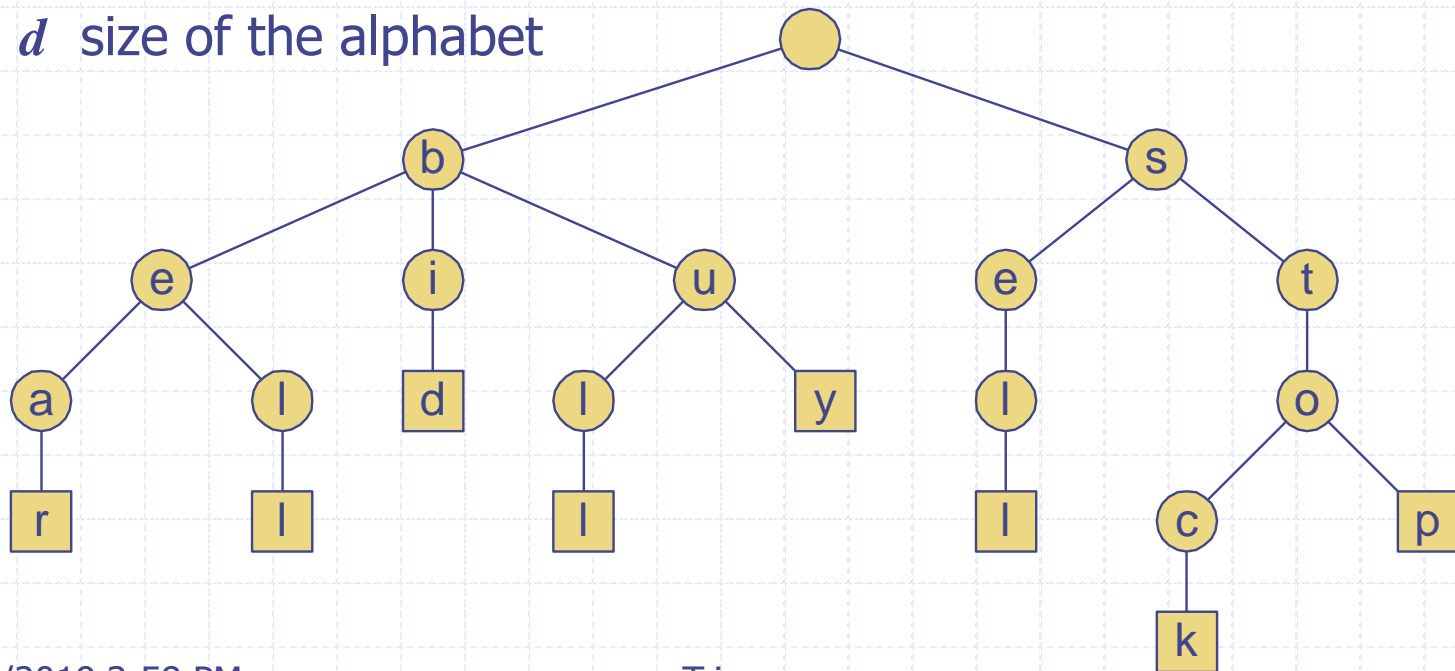
# Standard Trie

- ◆ A standard trie uses  $O(n)$  space and supports searches, insertions and deletions in time  $O(dm)$ , where:

$n$  total size of the strings in  $S$

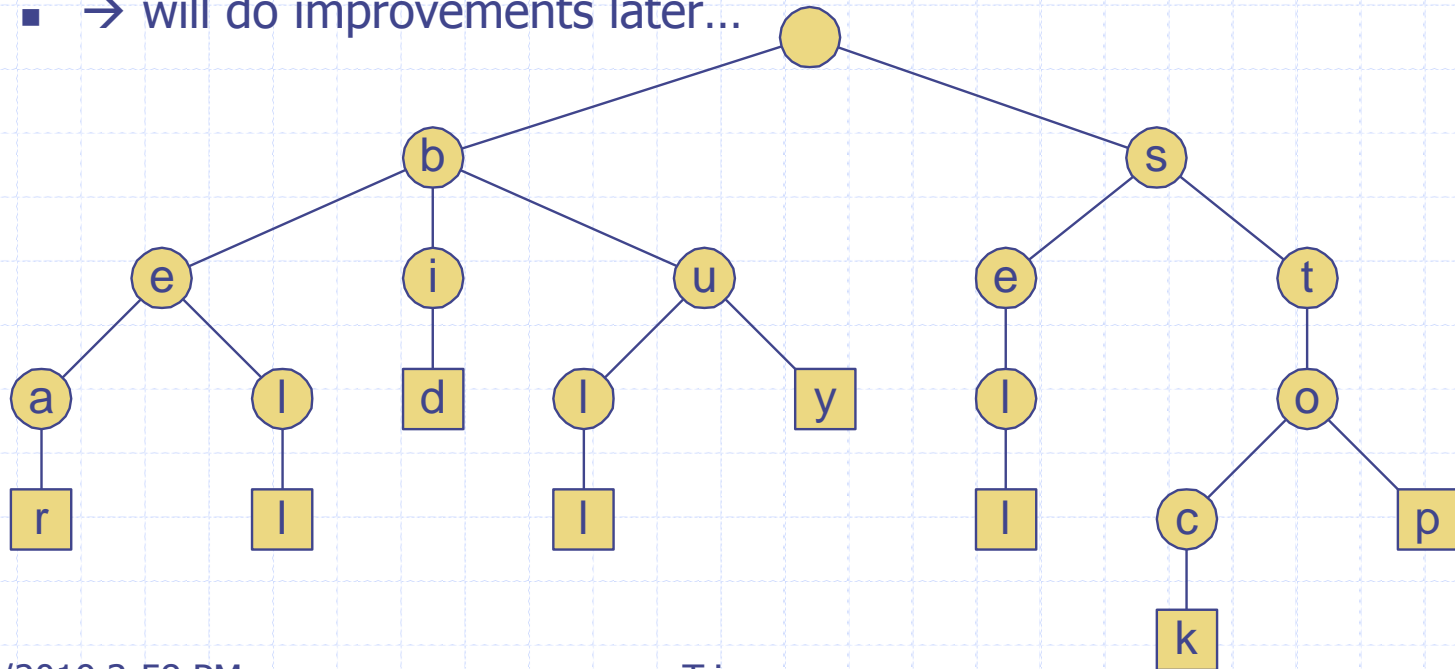
$m$  size of the (maximum) string parameter of the operation

$d$  size of the alphabet



# Standard Trie

- ◆ When is  $n$ , and thus space, maximum?
  - When  $S$  consists of mutually unique words with no letters in common
- ◆ What type of word(s) produces the largest search time?
  - Short word? Long word?
  - Answer: long words, especially those whose prefix is very common
  - → will do improvements later...



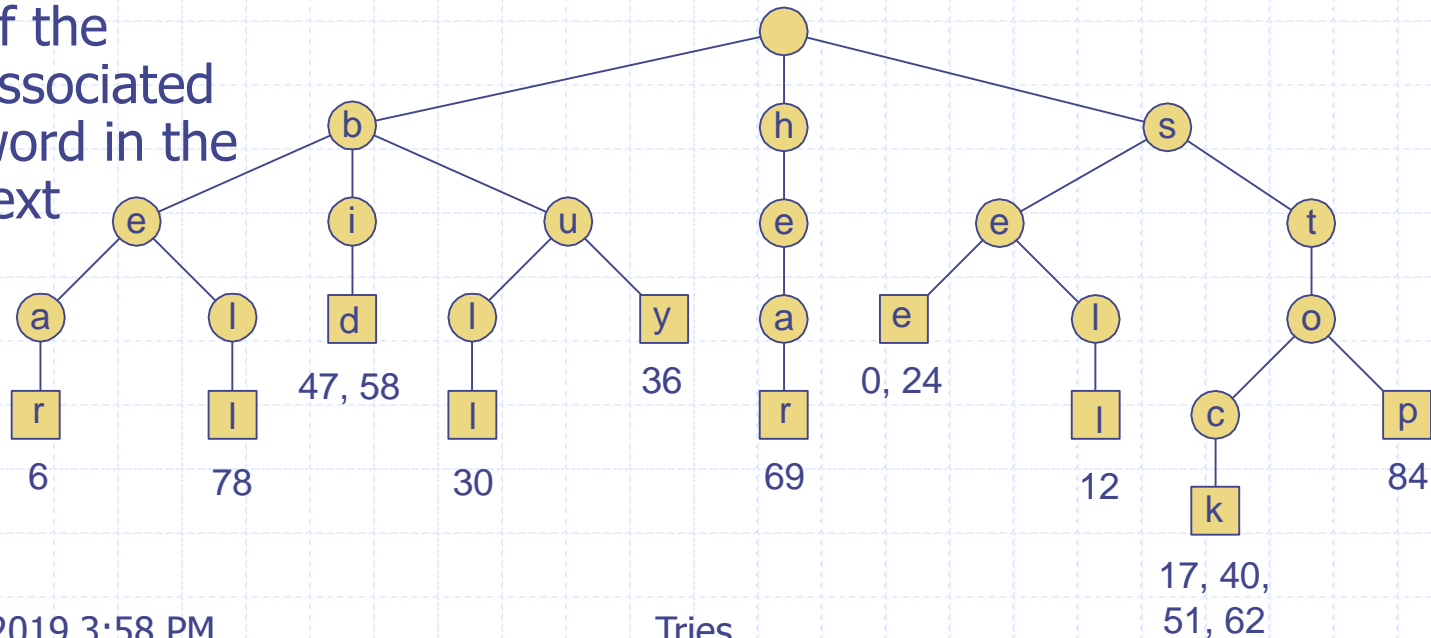


# Word Matching with a Trie

- ◆ We insert the words of the text into a trie

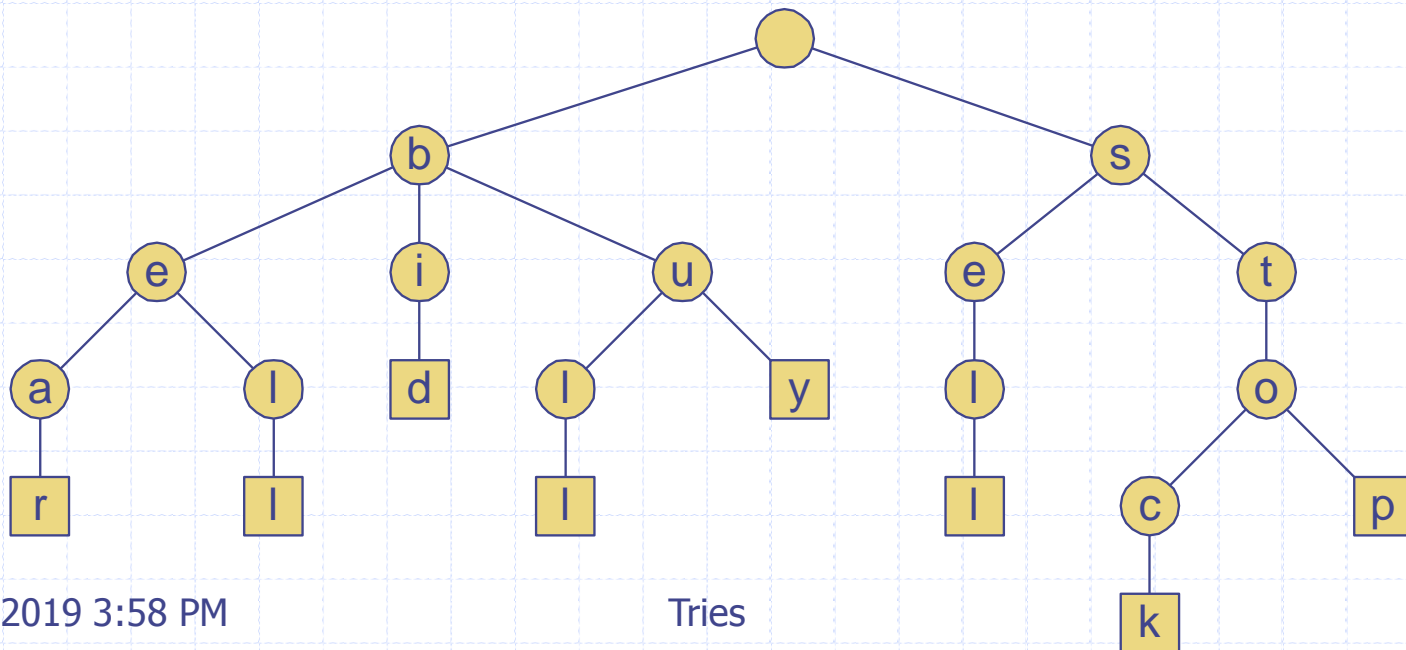
- ◆ Each leaf stores the occurrences of the associated word in the text

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					



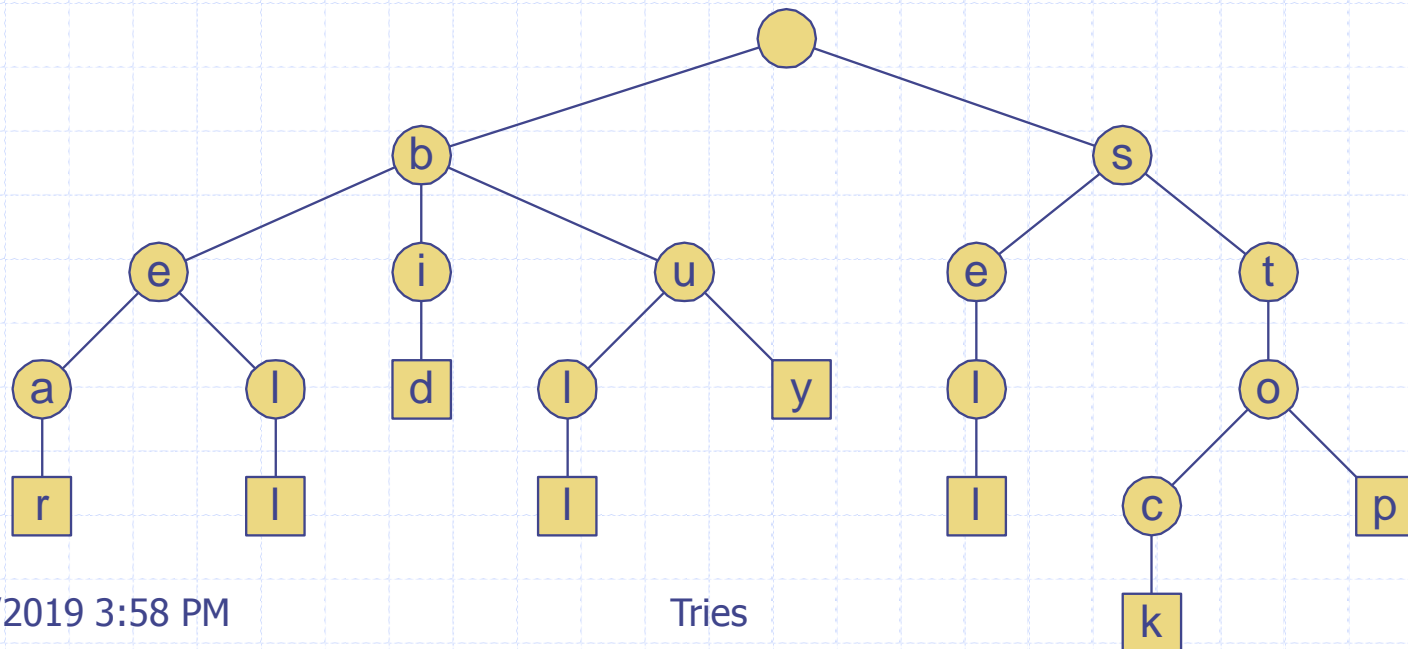
# Standard Trie Construction

◆ How do you build it?



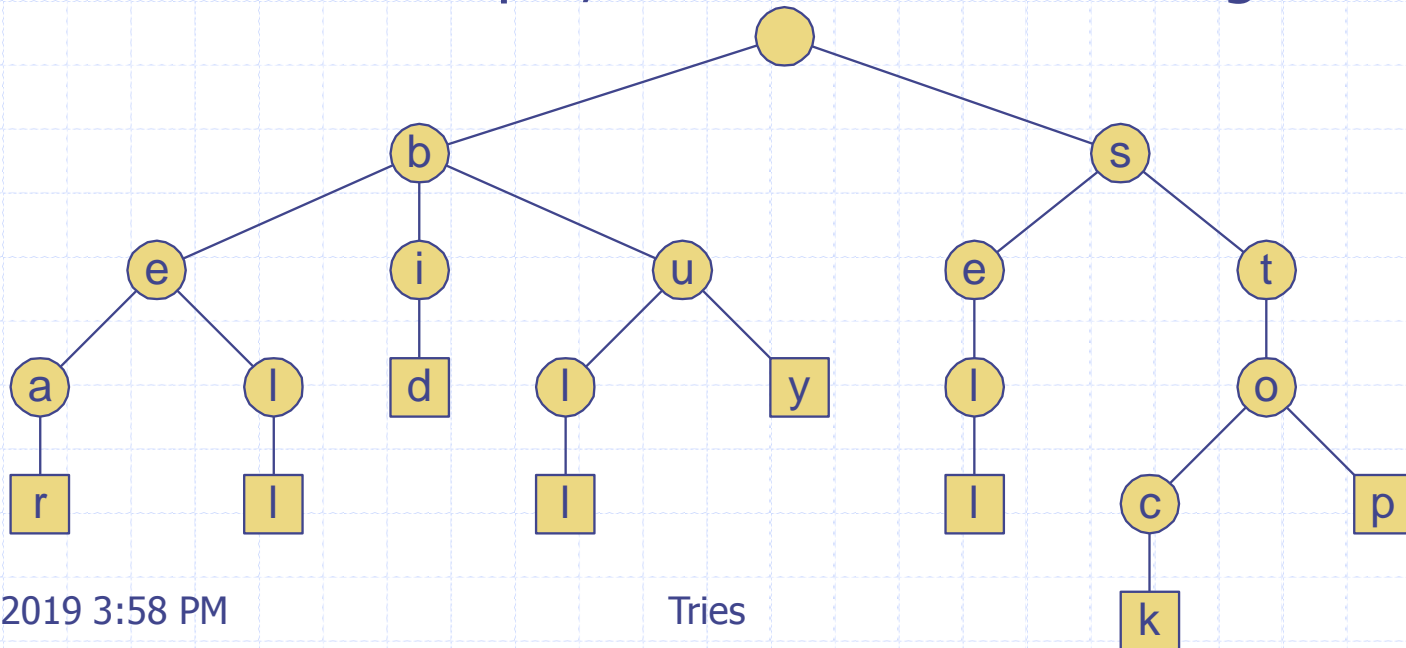
# Standard Trie Construction

- ◆ Assuming the input strings are words in the English language, how many children does the root node have?



# Standard Trie Construction

- ◆ The number of children of the root node equals to the maximum number of distinct first letters all the words in the input string
  - 2 in this example, maximum of 26 in English



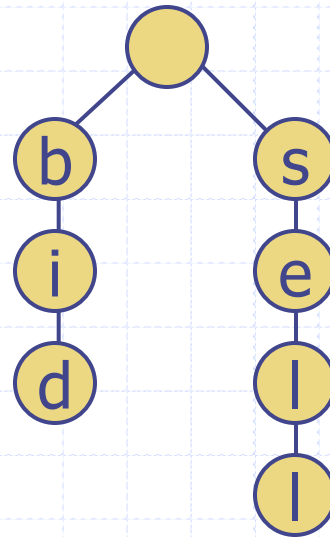
# Standard Trie Construction

◆ What does the tree for “bid” look like?



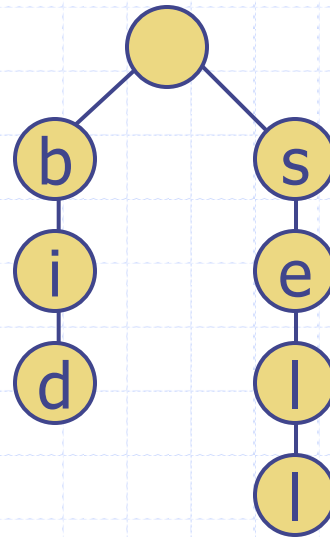
# Standard Trie Construction

- ◆ What does the tree for “bid” and “sell” look like?



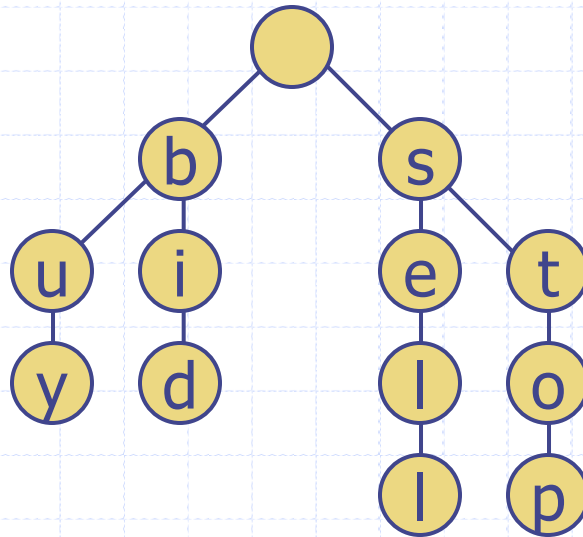
# Standard Trie Construction

◆ What is the tree after adding “buy” and “stop”?



# Standard Trie Construction

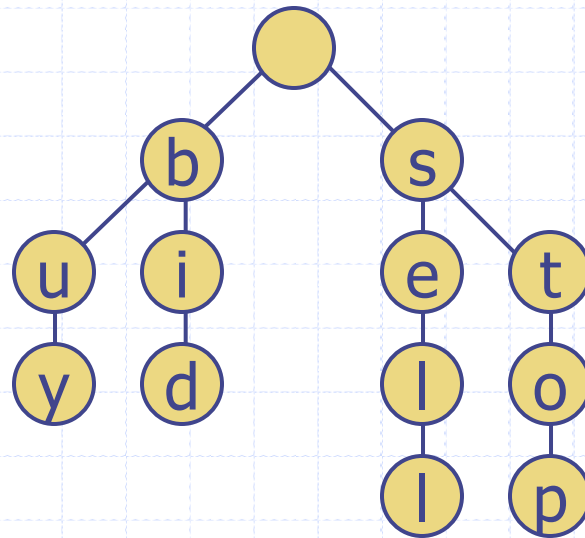
◆ What is the tree after adding “buy” and “stop”?





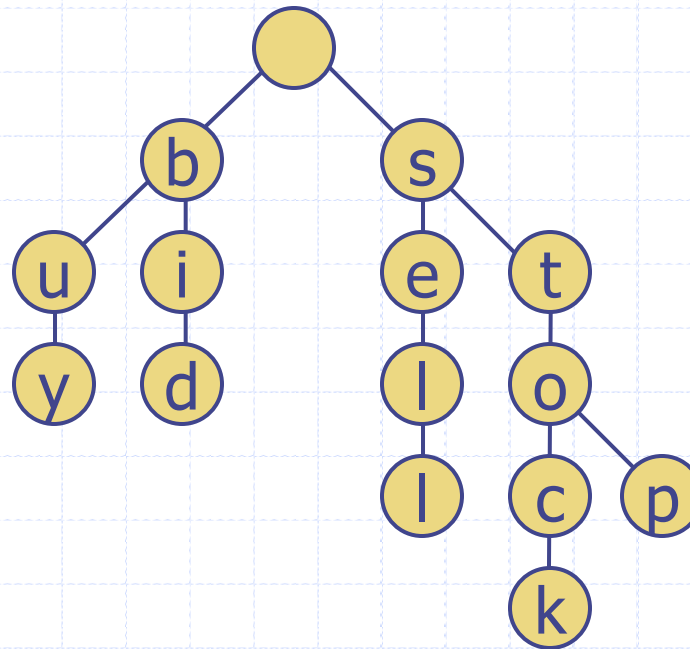
# Standard Trie Construction

◆ What is the tree after adding “stock”?



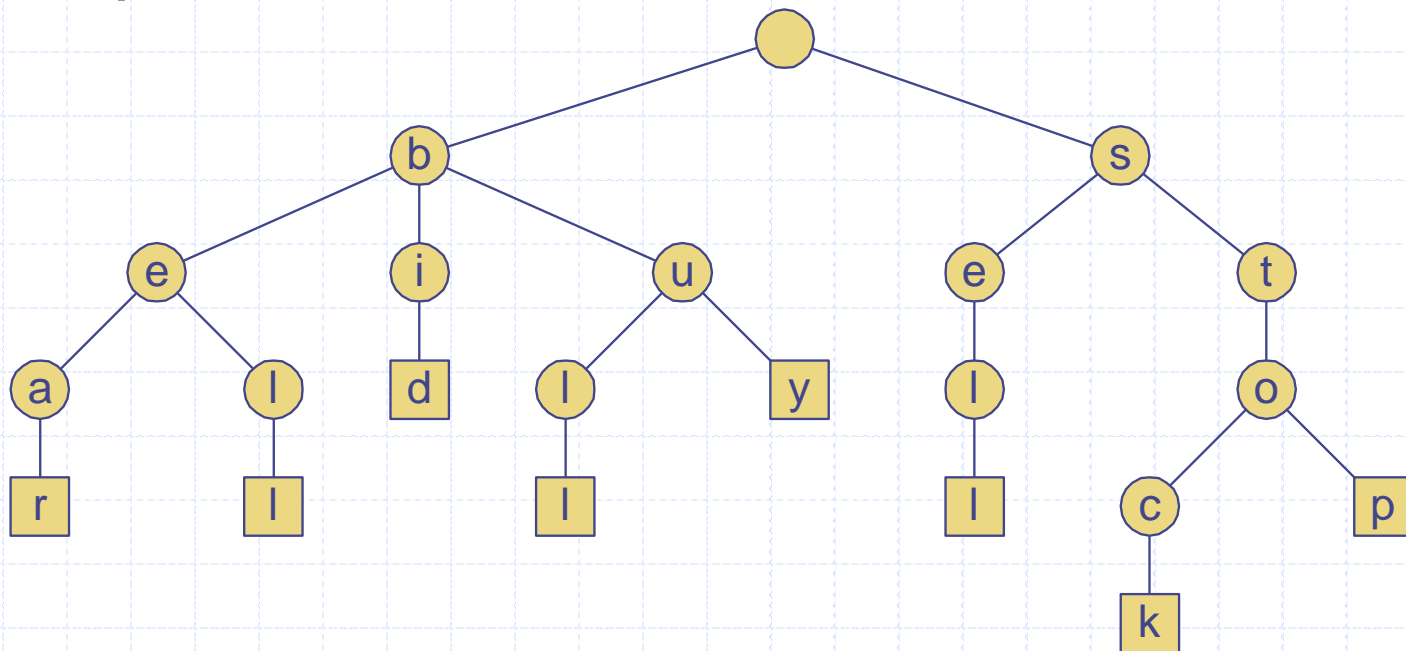
# Standard Trie Construction

◆ What is the tree after adding "stock"?



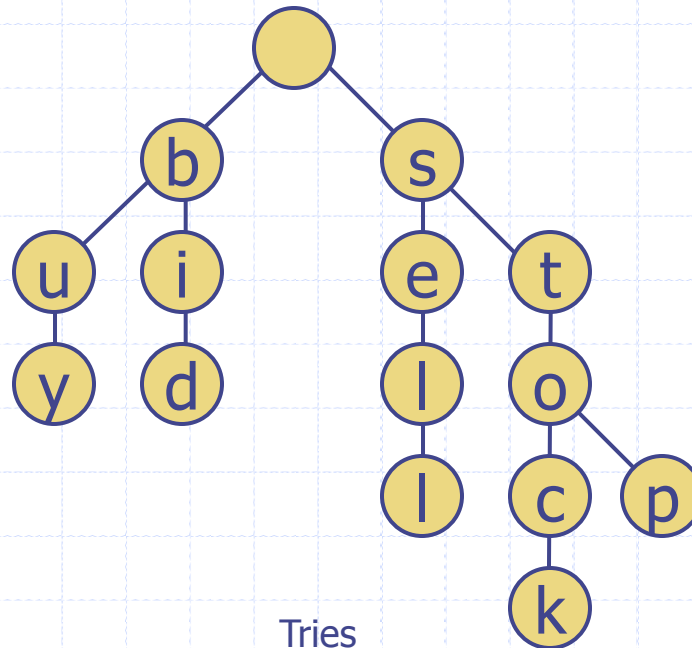
# Standard Trie Construction

◆ After "bear, bell, bid, bull, buy, sell, stock, stop"...



# Improvements

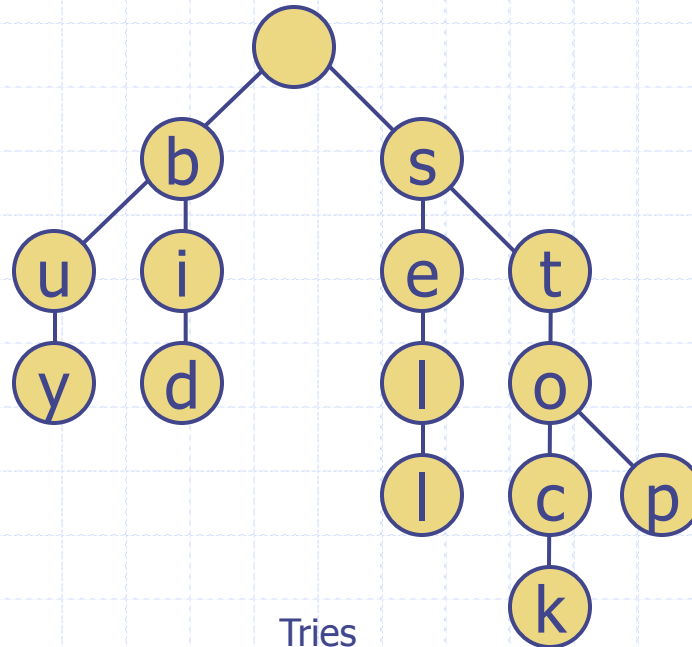
- ◆ What comes to mind for tries?
  - e.g., is this entire tree really necessary?



# Improvements

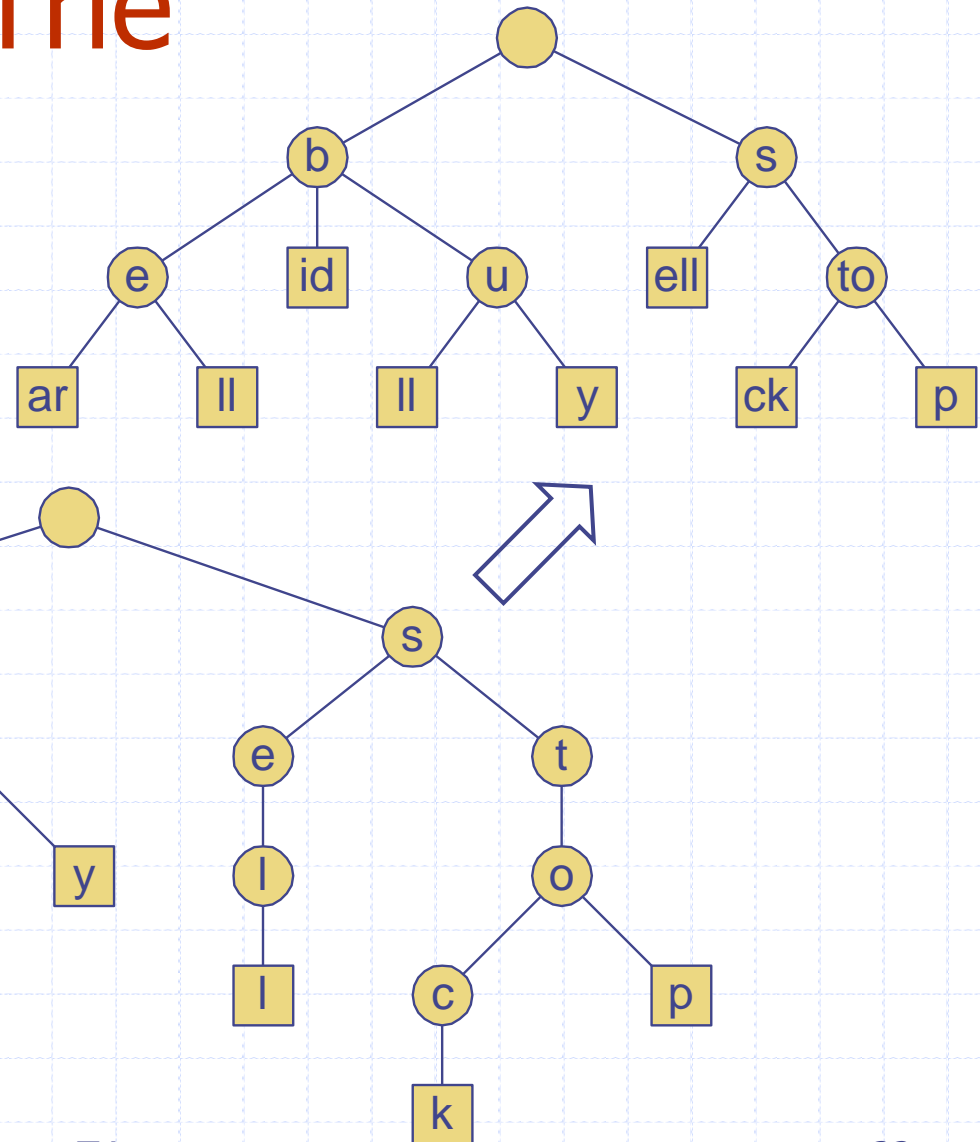
## ◆ Two types of compression

- Compress internal single-children node sequences
  - ◆ Also called “PATRICIA Tries” – Why?
  - ◆ = Practical AlgoriThm to Retrieve Information Coded In Alphanumeric (also called a “radix tree”)
- Compress external single-children leaf-node sequences



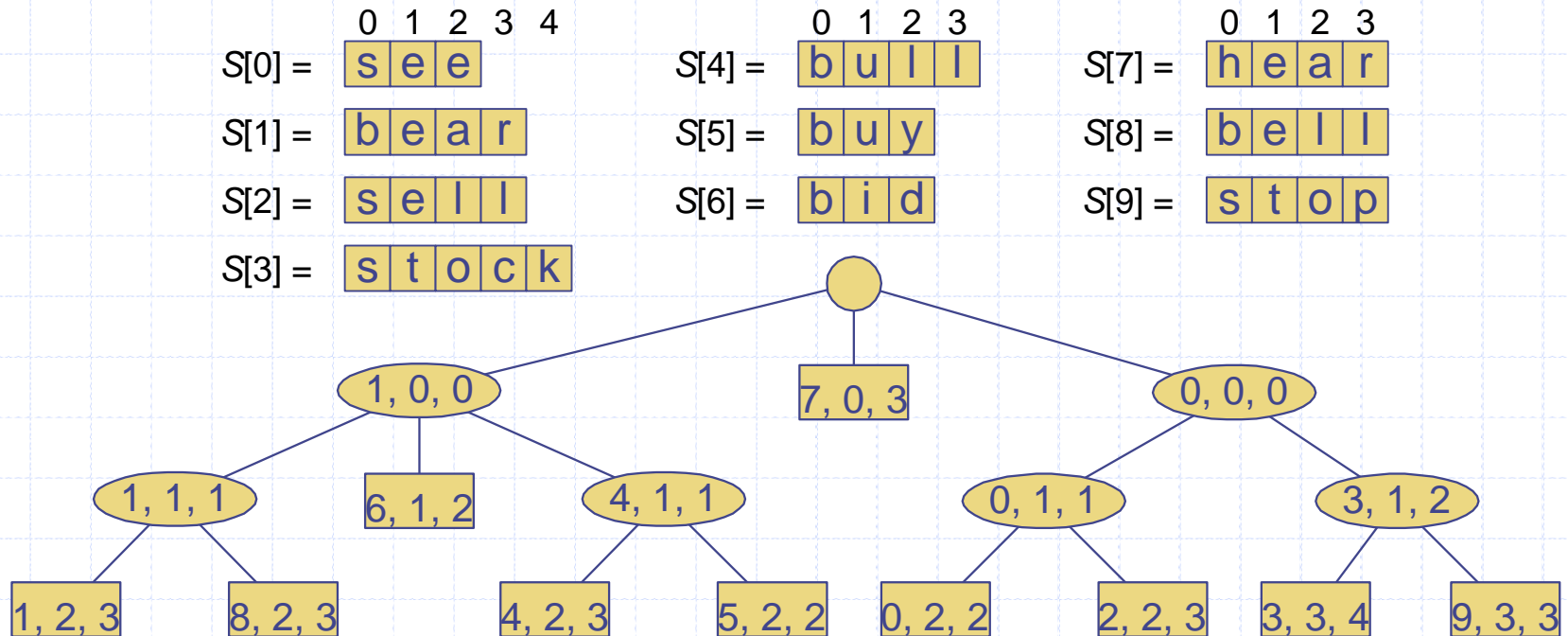
# Compressed Trie

- ◆ A compressed trie has internal nodes of degree at least two
- ◆ It is obtained from standard trie by compressing chains of “redundant” nodes



# Compact Representation

- ◆ Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Serves as an auxiliary index structure



# More Improvements

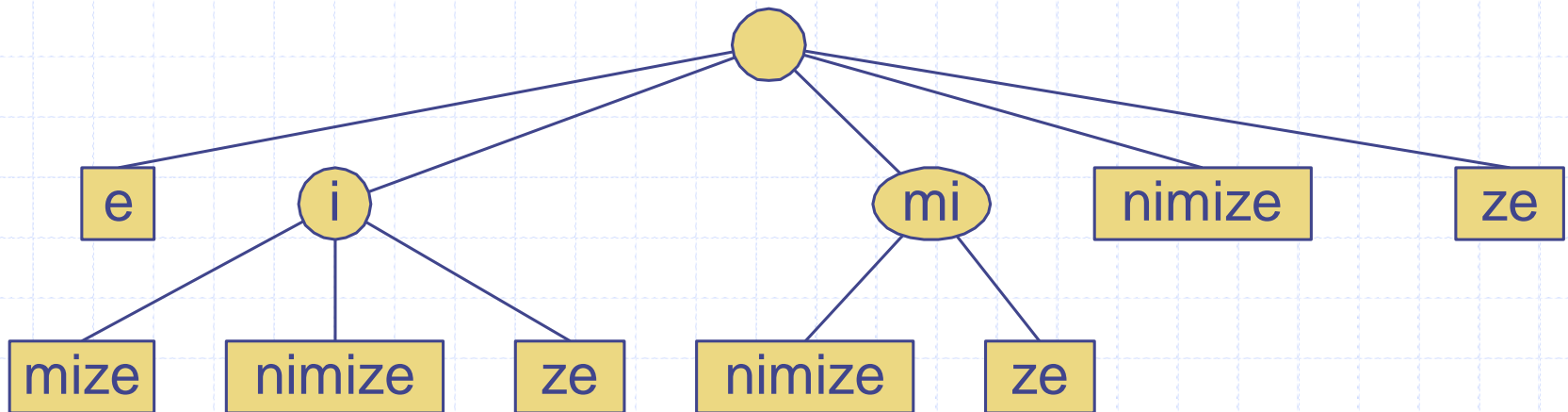
- ◆ We can build a standard trie and then compress it
- ◆ But, can we build some sort of compressed trie directly?
- ◆ Ideas?



# Suffix Trie

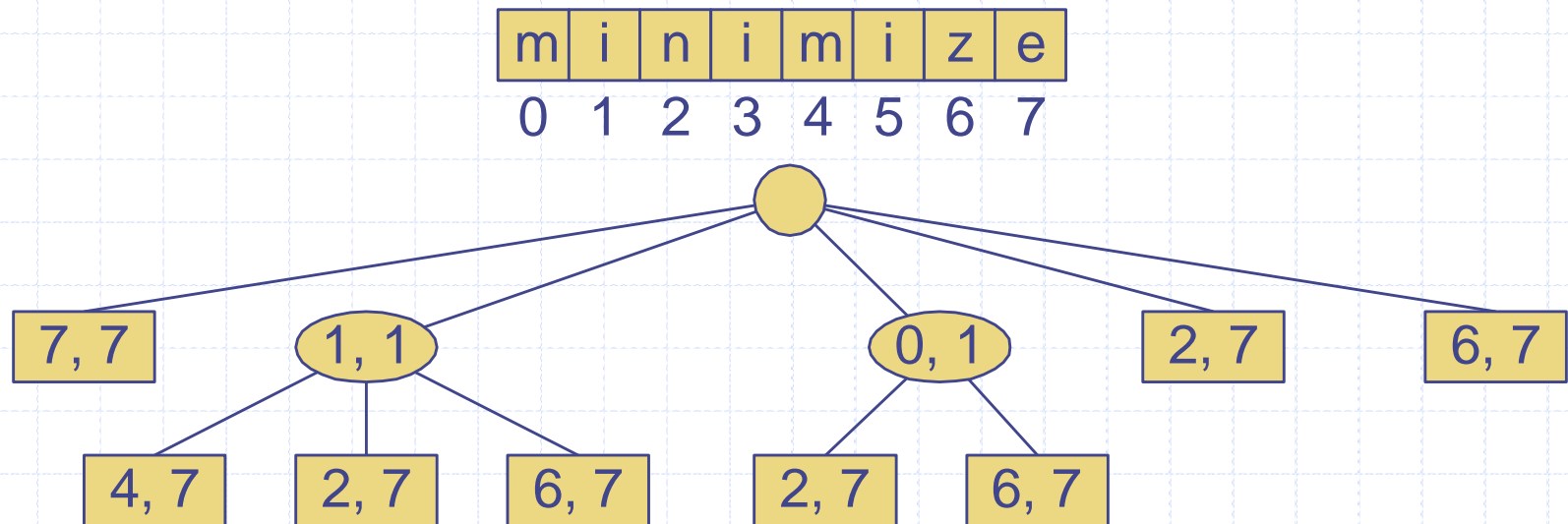
- ◆ The suffix trie of a string  $X$  is the compressed trie of all the suffixes of  $X$

m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7



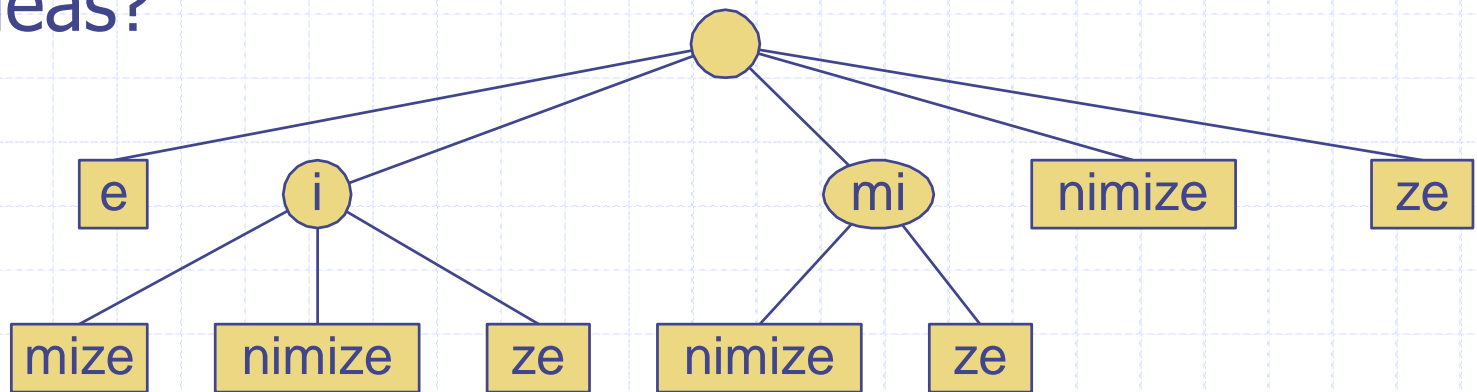
# Suffix Trie

- ◆ Compact representation of the suffix trie for a string  $X$  of size  $n$  from an alphabet of size  $d$ 
  - Uses  $O(n)$  space
  - Supports arbitrary pattern matching queries in  $X$  in  $O(dm)$  time, where  $m$  is the size of the pattern
  - Repetitive words not stored repetitively



# More Improvements

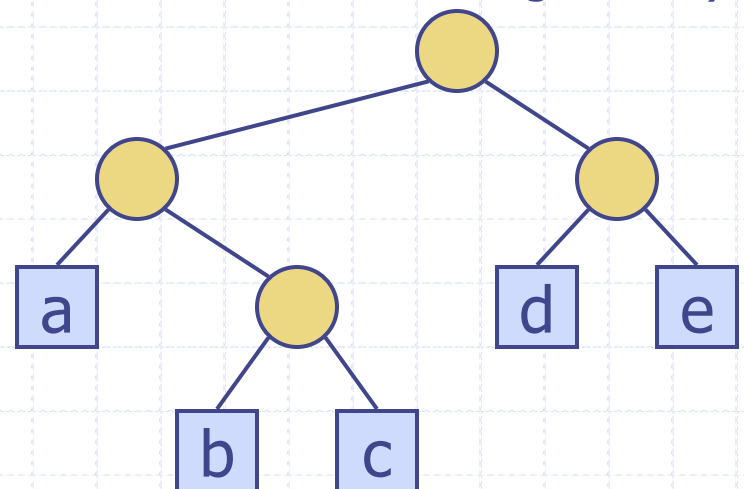
- ◆ There is still some repetition in this tree
  - e.g., “mize” appears several times
- ◆ How can we further compress the trie, thus reducing space and improving query time?
- ◆ Ideas?



# Encoding Trie

- ◆ A code is a mapping of each character of an alphabet to a binary code-word
- ◆ A prefix code is a binary code such that no code-word is the prefix of another code-word
- ◆ An encoding trie represents a prefix code
  - Each leaf stores a character
  - The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child)

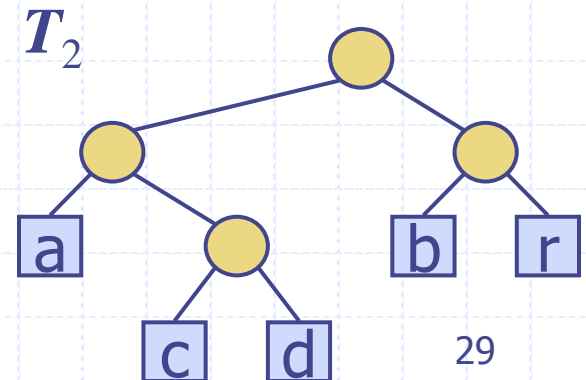
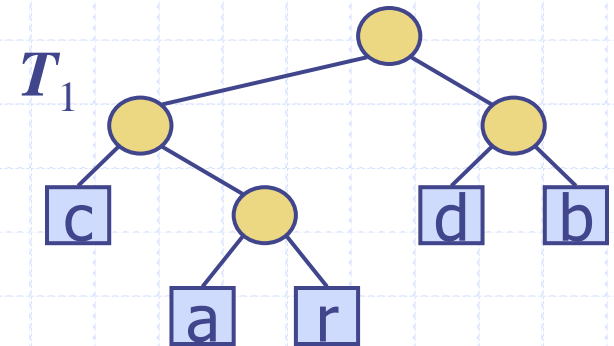
00	010	011	10	11
a	b	c	d	e



# Encoding Trie

- ◆ Given a text string  $X$ , we want to find a prefix code for the characters of  $X$  that yields a small encoding for  $X$ 
  - Frequent characters should have short code-words
  - Rare characters should have long code-words
  - Why?
- ◆ Example
  - $X = \text{abracadabra}$
  - $T_1$  encodes  $X$  into 29 bits
    - ◆  $29 = 3 + 2 + 3 + 3 + 2 + 3 + 2 + 3 + 2 + 3 + 3$
  - $T_2$  encodes  $X$  into how many bits?
    - ◆  $24 = 2 + 2 + 2 + 2 + 3 + 2 + 3 + 2 + 2 + 2 + 2$

How can we build a good encoding trie?



# Huffman's Algorithm

- ◆ Given a string  $X$ , Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of  $X$
- ◆ It runs in time  $O(n + d \log d)$ , where  $n$  is the size of  $X$  and  $d$  is the number of distinct characters of  $X$
- ◆ A heap-based priority queue is used as an auxiliary structure

## Algorithm *HuffmanEncoding*( $X$ )

**Input** string  $X$  of size  $n$

**Output** optimal encoding trie for  $X$

$C \leftarrow \text{distinctCharacters}(X)$

$\text{computeFrequencies}(C, X)$

$Q \leftarrow$  new empty heap

**for all**  $c \in C$

$T \leftarrow$  new single-node tree storing  $c$

$Q.\text{insert}(\text{getFrequency}(c), T)$

**while**  $Q.\text{size}() > 1$

$f_1 \leftarrow Q.\text{minKey}()$

$T_1 \leftarrow Q.\text{removeMin}()$

$f_2 \leftarrow Q.\text{minKey}()$

$T_2 \leftarrow Q.\text{removeMin}()$

$T \leftarrow \text{join}(T_1, T_2)$

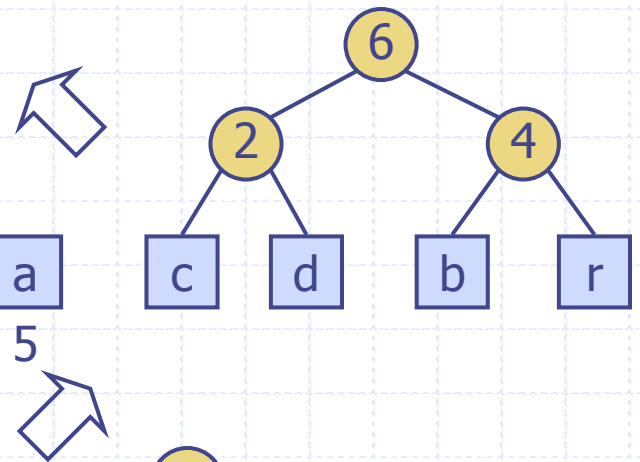
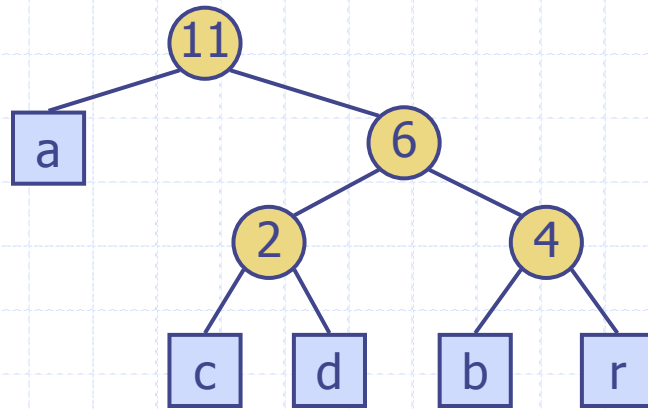
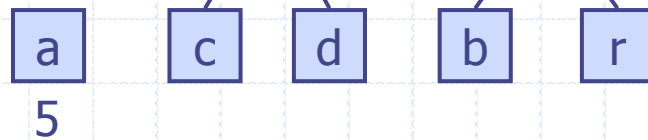
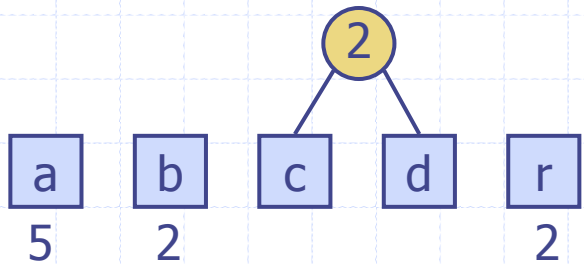
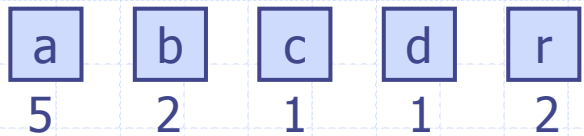
$Q.\text{insert}(f_1 + f_2, T)$

**return**  $Q.\text{removeMin}()$

# Example

$X = \text{abracadabra}$   
Frequencies

a	b	c	d	r
5	2	1	1	2



# Summary of Pattern Matching

Algorithm	Search Time	Notes
Brute force	$O(nm)$	<ul style="list-style-type: none"><li>◆ simple, no preprocessing</li><li>◆ slow (good for small inputs)</li></ul>
Boyer-Moore	$O(nm+s)$	<ul style="list-style-type: none"><li>◆ <math>O(m)</math> preprocessing</li><li>◆ significantly faster than previous in practice</li></ul>
KMP	$O(n+m)$	<ul style="list-style-type: none"><li>◆ <math>O(m+s)</math> preprocessing</li><li>◆ more complex, but ideal very fast</li></ul>
Standard Trie	$O(dm)$	<ul style="list-style-type: none"><li>◆ <math>O(n)</math> preprocessing, <math>d</math> = size of alphabet</li><li>◆ fast</li></ul>
Suffix Trie	$O(dm)$	<ul style="list-style-type: none"><li>◆ <math>O(n)</math> preprocessing</li><li>◆ faster in practice because "compressed"</li></ul>
Huffman-Encoding Trie	$O(dm)$	<ul style="list-style-type: none"><li>◆ <math>O(n+d\log d)</math> preprocessing</li><li>◆ fastest and smallest in practice</li><li>◆ leads to lossless compression: ZIP</li></ul>