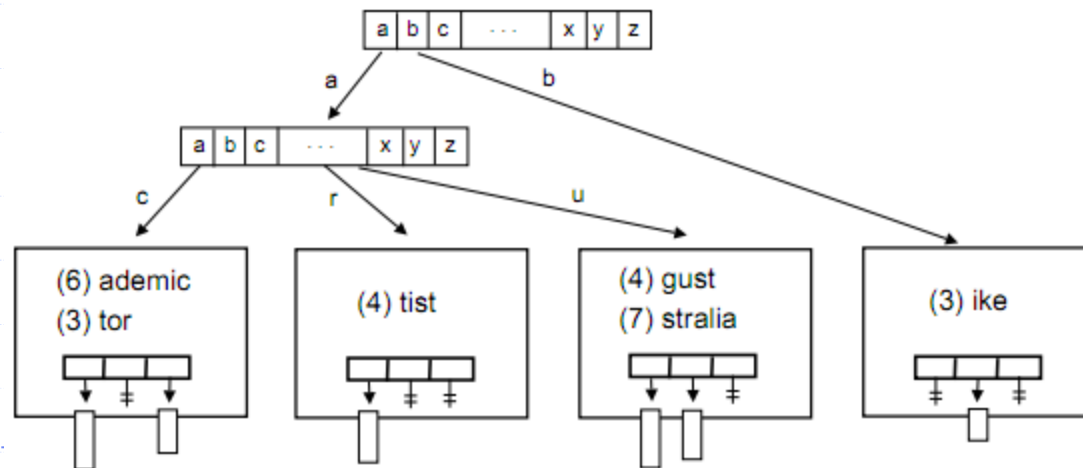


# Tries

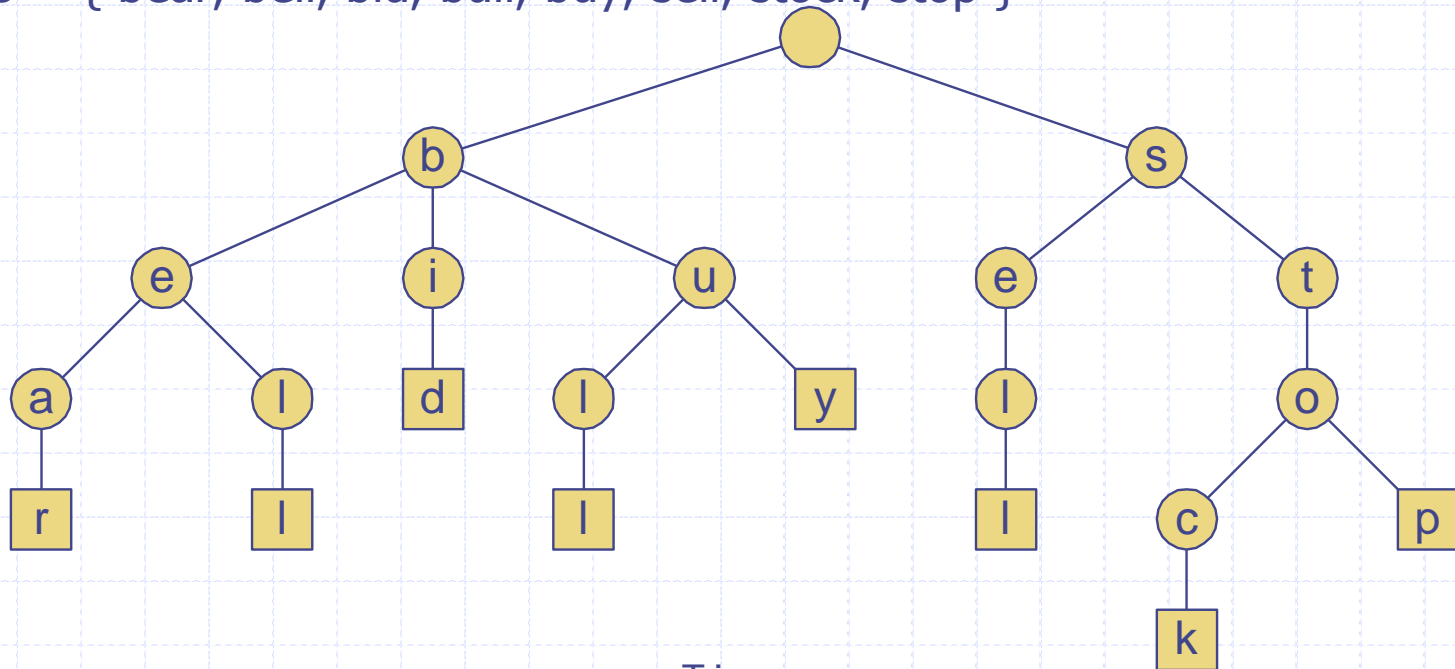


# Preprocessing Strings

- ◆ 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
- ◆ 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**

# Standard Tries

- ◆ 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  
 $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$



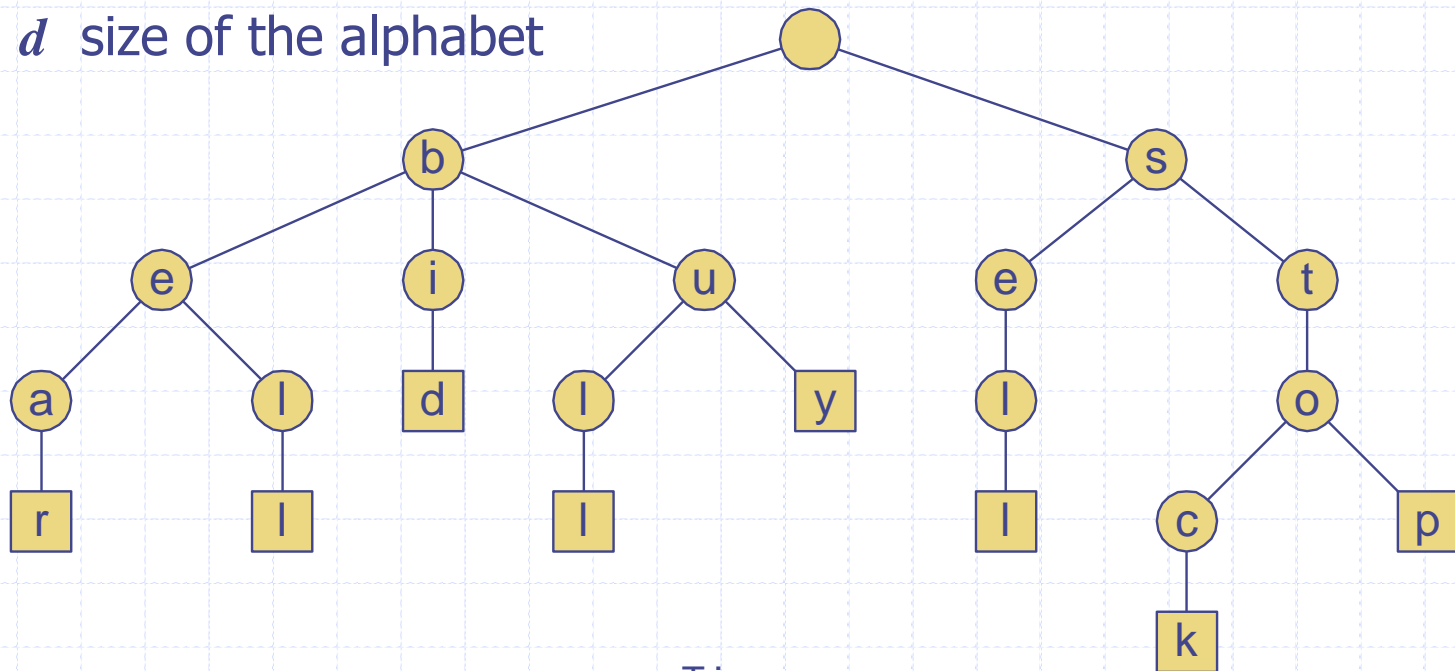
# Analysis of Standard Tries

- ◆ 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 string parameter of the operation

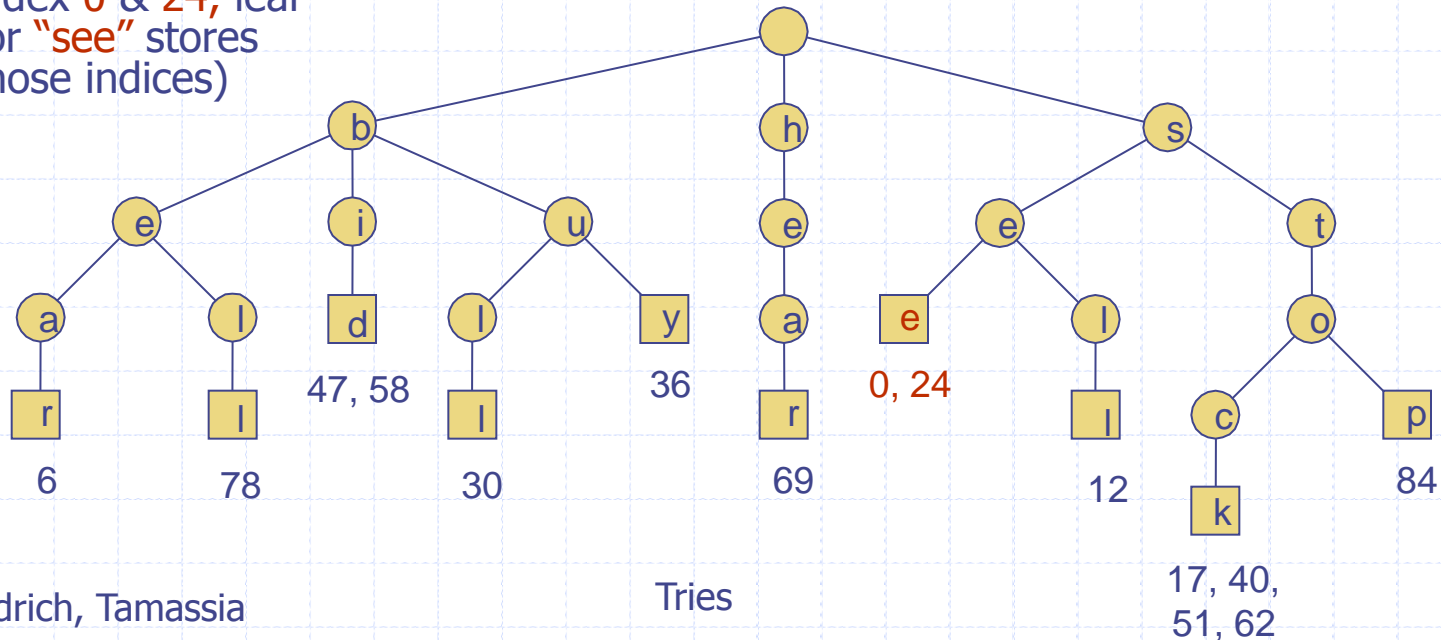
$d$  size of the alphabet



# Word Matching with a Trie

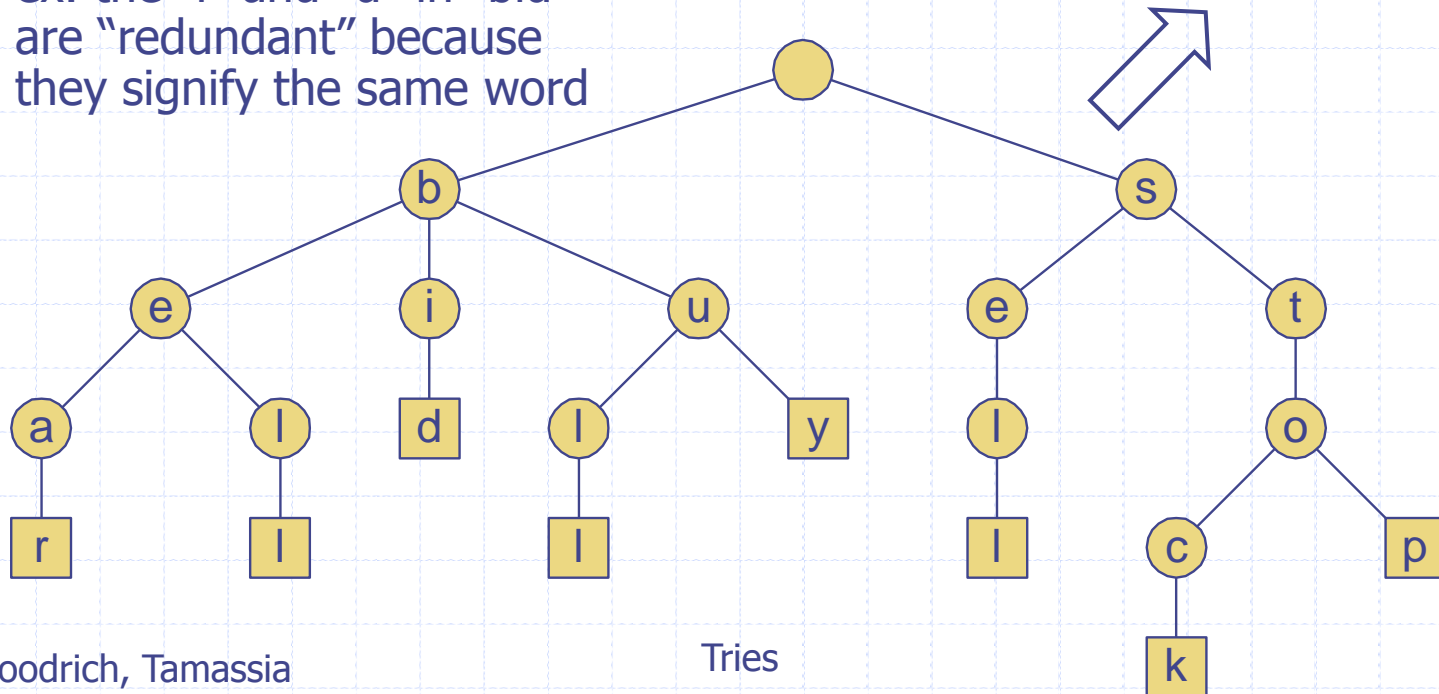
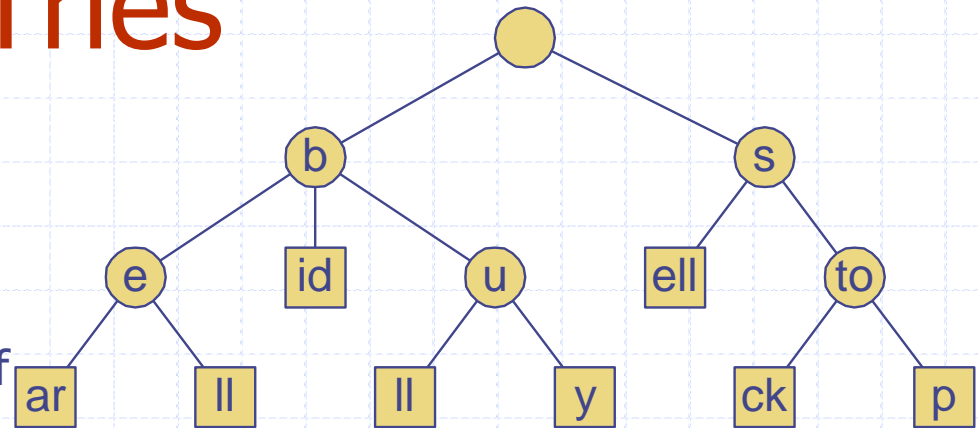
- ◆ insert the words of the text into trie
- ◆ Each leaf is associated w/ one particular word
- ◆ leaf stores indices where associated word begins  
("see" starts at index 0 & 24, leaf for "see" stores those indices)

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				



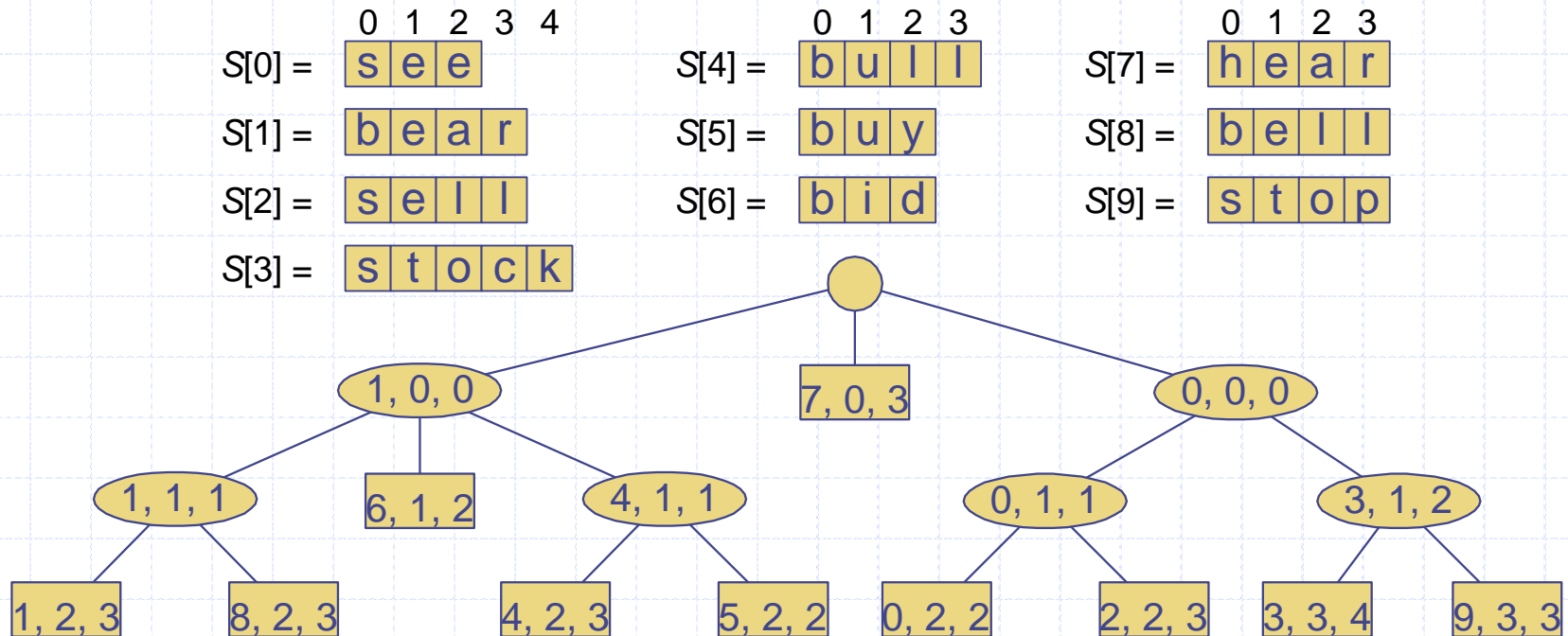
# Compressed Tries

- ◆ A compressed trie has internal nodes of degree at least two
- ◆ It is obtained from standard trie by compressing chains of "redundant" nodes
- ◆ ex. the "i" and "d" in "bid" are "redundant" because they signify the same word



# Compact Representation

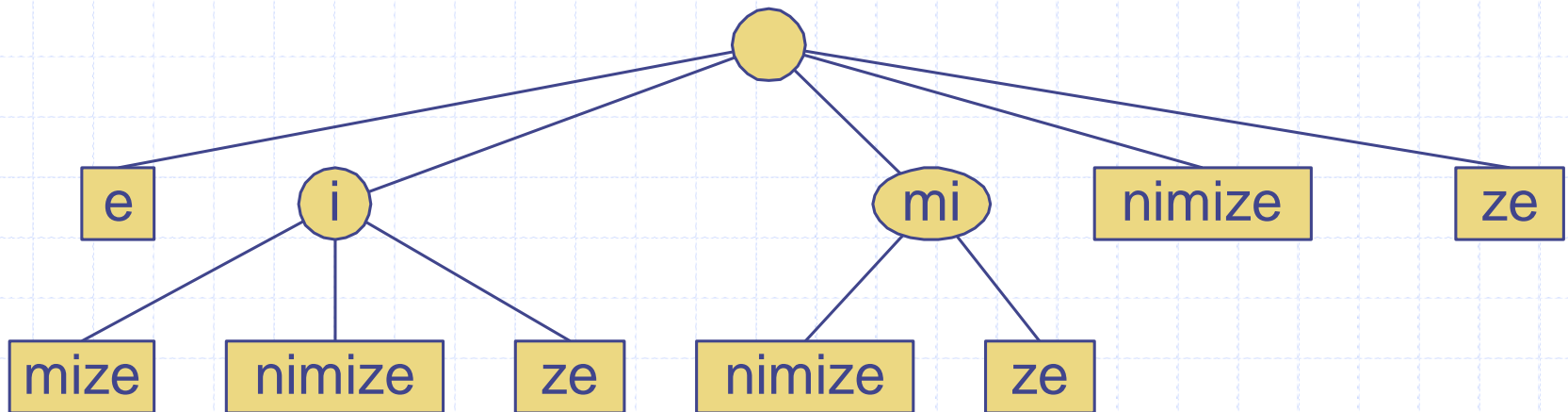
- ◆ Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses  $O(s)$  space, where  $s$  is the number of strings in the array
  - Serves as an auxiliary index structure



# Suffix Trie

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

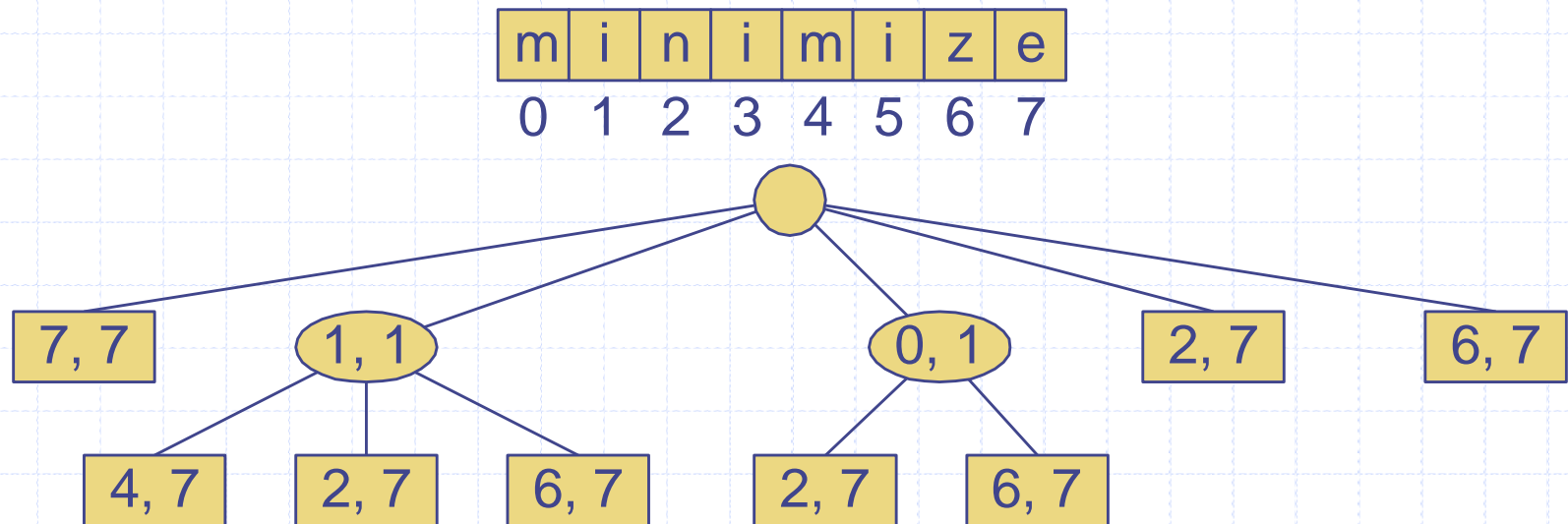
m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7





# Analysis of Suffix Tries

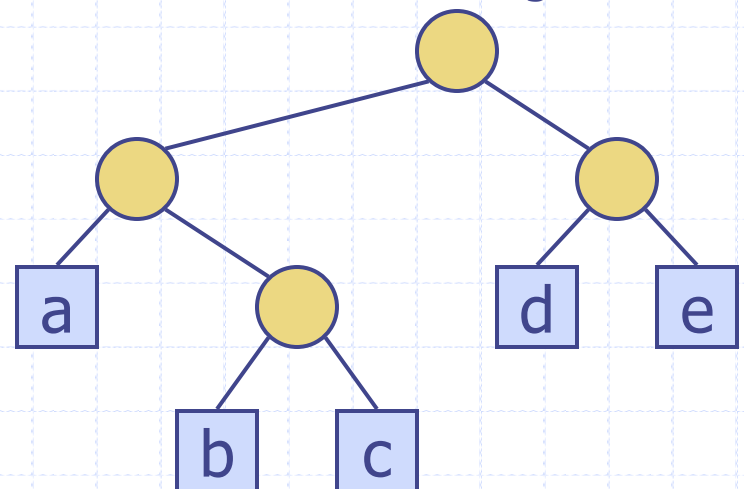
- ◆ Compact representation of the suffix trie for a string  $X$  of size  $n$  from an alphabet of size  $d$ 
  - Uses  $O(n)$  space
  - Supports arbitrary pattern matching queries in  $X$  in  $O(dm)$  time, where  $m$  is the size of the pattern
  - Can be constructed in  $O(n)$  time



# Encoding Trie (1)

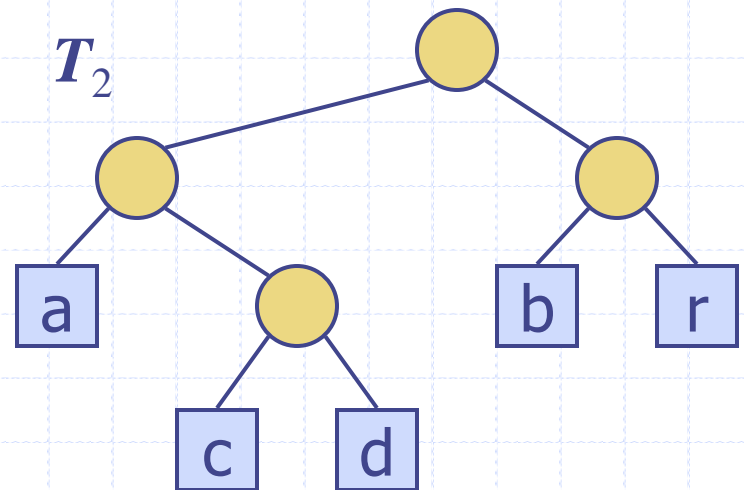
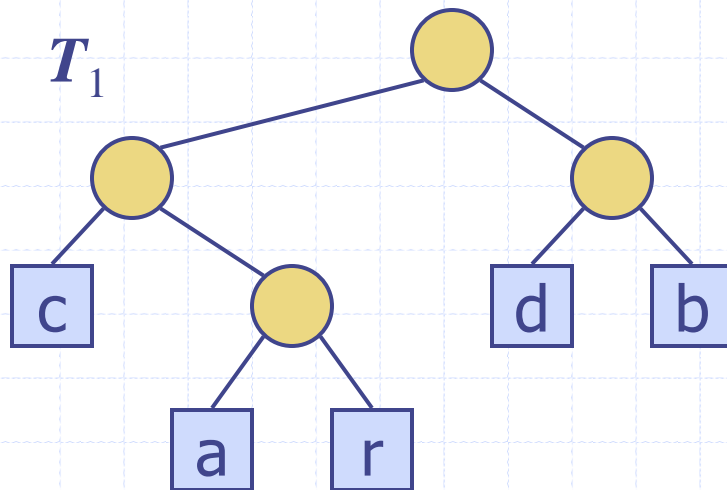
- ◆ A code is a mapping of each character of an alphabet to a binary code-word
- ◆ A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- ◆ An encoding trie represents a prefix code
  - Each leaf stores a character
  - The code word of a character is given by the 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 (2)

- ◆ 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
- ◆ Example
  - $X = \text{abracadabra}$
  - $T_1$  encodes  $X$  into 29 bits
  - $T_2$  encodes  $X$  into 24 bits



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

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

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

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

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

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

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

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

# Example

$X = \text{abracadabra}$   
Frequencies

a	b	c	d	r
5	2	1	1	2

