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 tries supports pattern matching queries in time proportional to the pattern size
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
$S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$
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
We insert the words of the text into a trie.

Each leaf stores the occurrences of the associated word in the text.
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.
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

$S[0] = \text{see}$  $S[4] = \text{bull}$  $S[7] = \text{hear}$

© 2004 Goodrich, Tamassia
The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$. 

Prefixes include:
- mize
- nimize
- ze
- nimize
- ze

The trie structure:

- M: minimized
- i
- m: minimize
- z: ze
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