CS 240: Data Structures and Data Management
Module 6 Study Guide

Taylor J. Smith — Spring 2017

Key Concepts

- **Tries** are binary trees that store data based on bitwise comparisons.
- As we follow edges in a trie, we read symbols that eventually lead to marked nodes (stored data).
- Operations:
  - Search — from the root, follow the path corresponding to the current symbol until you have no more symbols to read
  - Delete — perform a search, then unmark the node and delete all empty (useless) parent nodes
  - Insert — perform a search, then expand the trie from the last node by adding nodes to represent extra symbols
  - Time complexity of all operations is $\Theta(|x|)$, where $|x|$ denotes the number of symbols in x
- **Compressed tries** reduce space complexity by eliminating unmarked nodes with one child.
- Nodes in a compressed trie store more information: the next index position to read in the string and the string stored at that node (if any).
- Operations:
  - Search — from the root, follow the path as usual, then verify that the key in the final node matches the search key
  - Delete — perform a search, then delete the key and adjust the remaining nodes based on the structure of the trie
  - Insert — perform a search, then determine the position where the key belongs relative to the existing nodes (. . . it's complicated!)
- We can generalize tries from binary alphabets to arbitrary-size alphabets with **multiway tries**.
- We can remove the need to mark nodes by introducing an end-of-word character, usually $, and appending it to the end of each word in our trie.

Suggested Readings

- **Sedgewick:** 15.2 (Tries), 15.3 (Patricia Tries), 15.4 (Multiway Tries and TSTs)
- **Goodrich/Tamassia:** 9.2 (Tries)

Practice Questions

Sedgewick

15.13. Draw the trie that results when you insert items with the keys 01010011, 00000111, 00100001, 01010001, 11101100, 00100001, 10010101, 01001010 into an initially empty trie.

15.19. Write a program that prints out all keys in a trie that have the same initial t bits as a given search key.

15.35. Draw the patricia trie that results when you insert the keys 01010011, 00000111, 00100001, 01010001, 11101100, 00100001, 10010101, 01001010 into an initially empty trie.

15.36. Draw the patricia trie that results when you insert the keys 01001010, 10010101, 00100001, 11101100, 01010001, 00100001, 00000111, 01010011 in that order into an initially empty trie.

15.42. Write a program that prints out all keys in a patricia trie that have the same initial t bits as a given search key.

CLRS

12-2. Given two strings $a = a_0 a_1 \dots a_p$ and $b = b_0 b_1 \dots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is **lexicographically less than** string b if either

(a) there exists an integer j, where $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \dots, j-1$ and $a_j < b_j$, or

(b) $p < q$ and $a_i = b_i$ for all $i = 0, 1, \dots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The **radix tree** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0 a_1 \dots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

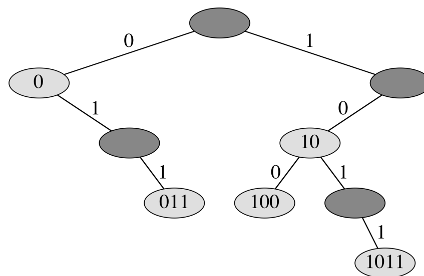

Figure 12.5: A radix tree storing the bit strings 1011, 10, 011, 100, and 0. We can determine each node's key by traversing the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.