Tries & Suffix Trees
Storing Collections of Strings

• All our dictionary data structures can be used to store strings, of course.

• But strings have several properties that might make you want a different data structure:
  - Comparing strings can be slow (not constant time)
  - Strings can “overlap” or be similar: e.g. cat, car, cars, cart
  - Special kinds of queries are important: e.g. “Does string S contain “marmoset” as a substring?”
Tries

- A trie, pronounced “try”, is a tree that exploits some structure in the keys
  - e.g. looking at their individual characters
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Saving Space #1

- Store nodes in a 2-d table.
- table size = \(|\sum| = size of the alphabet by number of nodes m\)
- Each entry contains the index of the node it “points to”
- Uses \(O(\log m)\) space instead of the size of the pointer (e.g. 32 or 64 bits)
- General trick: you ensure nodes are contained in some region of memory of size \(M\).
Saving Space #2

• Replace

  A  C  D  E  $

  with

  A  \rightarrow  D  \rightarrow  $  \rightarrow  NULL

• at each node

Array at each node becomes a linked list.

Saves space when the branching factor is low
don’t need to store an entry for each character in
the alphabet)

Also imagine a hybrid method, using arrays at
nodes with high branching factors

de la Brandais tree
Saving Space #3

A lot of nodes were non-discriminatory: they didn’t discriminate between two keys:

Idea: only store the discriminatory nodes
**Saving Space #3 – Patricia Tries**

Same tree, but only storing the discriminatory nodes.

**BUT:** now have to store the index of the character position the node is testing before, a node at depth d tested position d, but now that isn’t true: we can skip over positions)

Also: now you must CHECK whether a leaf you reach matches the query.

E.g. what if we searched for **cedar**.
Suffix Tree Variant

suffix $i$ is the part of the string from $i$ to the end

suffix identifier = the prefix of the suffix that is unique within the string

Every suffix has a identifier because of the $\$ character

Suffix tree variant: store the n+1 suffix identifiers in a trie
Store the suffix number (start position of the suffix) at each leaf
Search for abbr:
follow the path for abbr leaves under end node correspond to the locations of abbr.
Space usage:

- Use Patricia trees to store the suffix tree
- Then: \# leaves = \( O(n) \) [one leaf for each position in the string]
- Every internal node is at least a binary split.
- Therefore, \# number of internal nodes is about equal to the number of leaves
- Hence, linear space.
Topics for Final

- **Pre-midterm:**
  - C++, Stacks, queues, decks, graphs, BFS, DFS, sparse matrices, self-organizing lists, trees, threaded trees, tree traversals, bitvectors for sets, full binary tree thm, array implementation of trees, general trees as binary trees
  - B-trees, AVL-trees, Splay trees, heaps (leftist, skew), skip lists

- **Post-midterm:**
  - PR, MX, Point quadtrees
  - kd-trees, range searching
  - incremental nearest neighbor
  - 2-d range trees, “1-d range trees”, fractional cascading
  - Interval trees
  - Segment trees
  - Binary Space Partitions (painters algorithm, BSP trees)
  - Doubly-connected edge lists
  - Grid files
  - tries, patricia trees, de la Brandais tres, O(log m) pointer space, suffix trees