CS 151

Tries
Announcements

Structural Induction Bonus Session now on Friday in class
  • bring your algebra worksheet, completed. Bring prelab 7, if you can find it
  • I will send out some of the induction questions we may attempt via email

Test #2 is next Wednesday (right before Thanksgiving break)
  • If you want to take the test on Tuesday, let me know so we can arrange it.
  • Topics: BSTs (incl. balanced BSTs), PQs, heaps, struct. ind., hash tables.
  • For each data structure, you should know:
    • the supported operations
    • any assumptions about the data (e.g. comparable?)
    • RT of various implementations (our “ideas” from slides)
    • advantages and disadvantages of various implementations
  • From the labs, you should know
    • AVL rotations
    • heaps percolateUp, percolateDown, heapify
    • Comparators, Iterators
    • Markov model (applied to text generation)
A set is a collection of items without duplicates. Set operations include:

- `add( element )` — add the element to the set
- `contains( element )` — return whether the element is in the set
- `remove( element )` — return and remove the element
- `size()`
- `isEmpty()`
- `iterate()`, `list()`
- `(union(set T), intersection(set T), complement(), possibly.)`

What are some different ways we could implement a set?

**idea #1:** use a BST if the elements have `<`, `>` defined

- `add`, `contains`, `remove` would be $O(\log n)$, listing $O(n)$

**idea #2:** use a HT where the value is ignored (make it a Boolean or something)

- `add`, `contains`, `remove` are amortized $O(1)$ if you have good hash function

We'll implement the operations in $O(|e|)$, where $e$ is the element you are adding, removing, etc. Usually $|e| \in O(1)$ so this would be fantastic!
A trie (pronounced “try”) is used to store elements that happen to be (bounded) sequences of characters or digits. (Short for “reTRIEval”)
Ex. a set of strings, or a set of integers, but not a set of students (unless you use a numerical student ID as a key, in which case, fine.)
Trie’s are also known as “radix trees” and “digital search trees.”
idea: a trie is a tree, but instead of each node storing the key / element in its entirety, we use the path to a node to store some information about the element
We’ll do some examples first, and then come up with a definition:
A trie is a:
• boolean flag (isWord), along with
• an array of pointers to child Tries — one entry per radix value
• if a child is null, indicates no valid elements beginning with root-child prefix

contains( x=x[0]x[1]x[2]...x[s-1] ) // s = length(x)
• start at the root
• if root is null, return false // base case #1
• if string is empty, return isWord flag // base case #2
• otherwise, x has at least one character...
  • recursively search for x[1]x[2]...x[s-1] in Trie rooted at child x[0]

RT is: $O(|x|) = O(s)$

add( x=x[0]x[1]...x[s-1] )
• follow the same path as contains
• if hit null pointer, replace the pointer with a new Trie (with isWord false)
• when get to the end of the string, set isWord to true

RT is: $O(|x|) = O(s)$
list: (in lexicographical (alphabetical) order)
• write a subroutine list( prefix ) that list all words in Trie with prefix as prefix
  • if isWord is true, list prefix
  • for each non-null child i
    • call child i’s list( prefix + i’s radix )
  • to list all words in trie, from the root, call list(“”)

Note: this is a preorder traversal of the trie

Tries seem fantastically fast! Why don’t we use them all the time??
• space inefficiency. Use large array of infrequently used letters / radixes.
• can alleviate this by using some “tree tricks”, but can’t get rid of it completely.