**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
    - properties and properties of the data (e.g. compressed)

- From the labs, you should know:
  - AVL rotations
  - heaps percolateUp, percolateDown, heapify
  - from the labs, you should know
  - if you have good hash function

**Test #2** is next Wednesday (right before Thanksgiving break)

- I will send out some of the induction questions we may attempt via email.
- Bring your algebra worksheet, completed. Bring prelab 7 if you can find it.

---

**Sets**

- **add()** - add an element to the set
- **contains()** - check if an element is in the set
- **remove()** - remove an element from the set
- **size()** - return the size of the set

**Tries**

- A **trie** (pronounced "try") is used to store elements that happen to be
  - a bounded subset of characters or digits (short for "retrieval")
- Each node of the trie stores the path to the node and a pointer to the node
- A **trie** is a tree, but instead of each node storing the key / element
  - the path to a node stores some information about the element

**CS 151**

---
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]

`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 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 the trie, from the root, call list("")

Tries seem fantastically fast! Why don't we use them all the time?!
• space inefficiency. Use large array of infrequently used letters / radices
• can alleviate this by using some "free indices", but can't get rid of it
• Trie seem remarkably fast. Why don't we use them all the time??

Note: this is a preorder traversal of the trie
  • to list all words in the trie, from the root, call list("")
    • for each non-null child i
      • call child i's list( prefix + i's radix )
  • write a subroutine list( prefix ) that list all words in the trie with prefix as prefix

Isn't in lexicographical (alphabetical) order?

A trie is:

```c
RT: O(|x||0| = O(6)
```