Review Sheet 7a

CS 70: Data Structures and Program Development

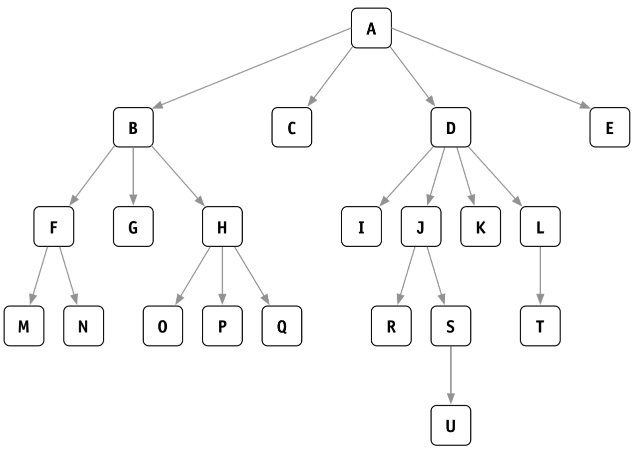
Tuesday, March 3, 2020

Learning Targets

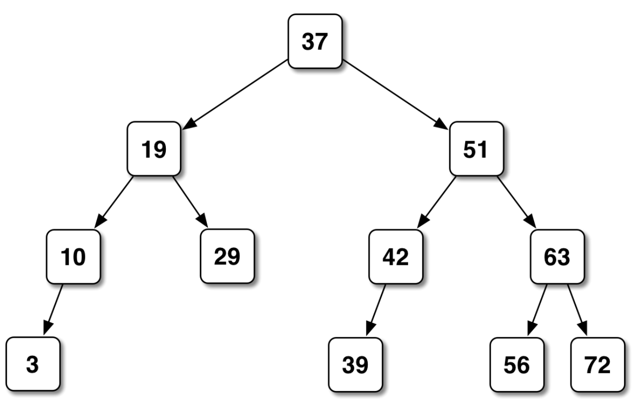
1. Given a tree, I can tell whether it's a valid BST.
2. I can simulate BST lookup, insert, and delete (on paper)
3. I can simulate left and right rotations (on paper)
4. I can simulate `insertAtRoot` (on paper)
5. I can simulate Randomized Binary Tree insertion.

Review

1. Define: node, edge; root, leaf, tree, subtree; parent, child, ancestor; height; balanced tree; binary tree; ordered binary tree (a.k.a. binary search tree)



2. Find 56; find 35; insert 47



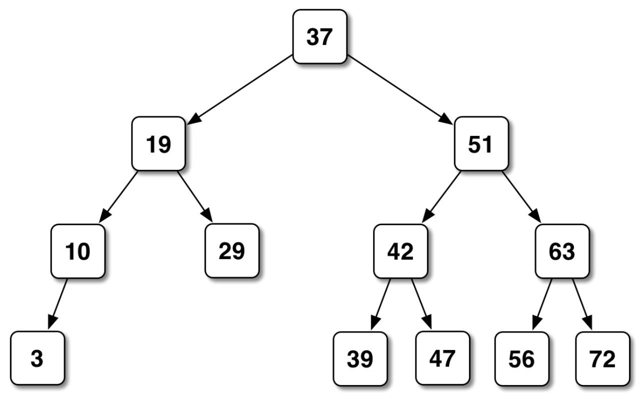
3. `insert` pseudocode {.shrink}

```
insert(tree, x):
    if tree is empty:
        make x its new root.

    else if x < tree's root:
        insert(left subtree, x)

    else if tree's root < x:
        insert(right subtree, x)
```

4. Delete 29, 10, 51



5. Exercise (continued)

   What tree results from the following sequences of inserts?

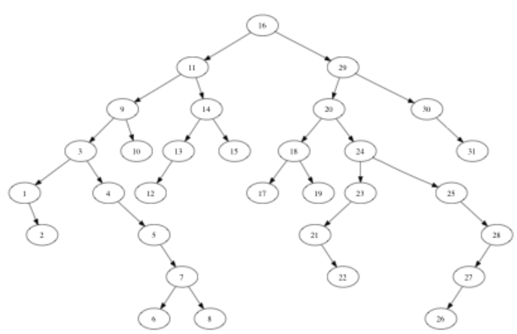
   - A, B, C, D, E, F, G
   - D, C, A, B, E, F, G

1. Suppose we have a BST with n nodes.

   What is the running time for `find` (and `insert`)

   - if we have a really terrible tree?
   - if we have a really nice tree?
   - if we have a "random" tree?

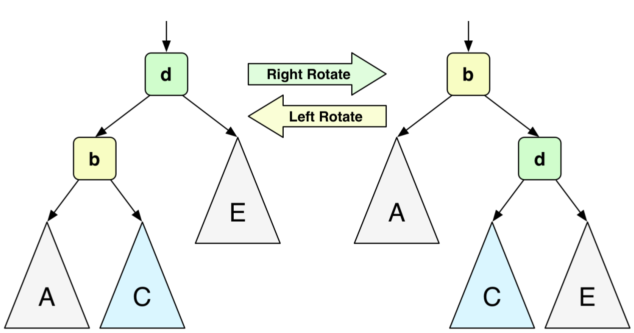
2. Random trees average 39% worse than perfect



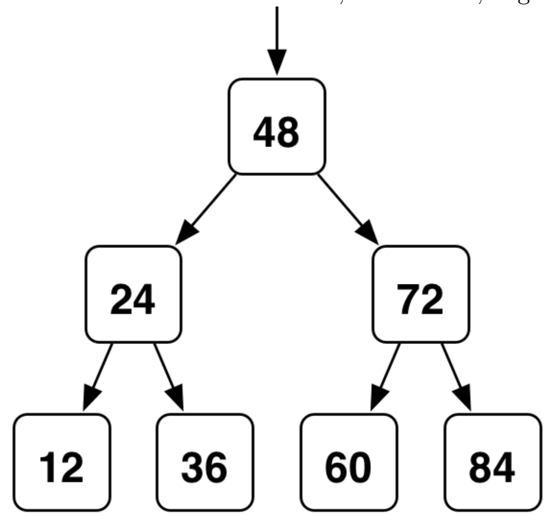
3. Building better trees: Off-line algorithm

   1. Take the inputs we want to put in the tree.
   2. Randomly shuffle them.
   3. Build tree by inserting in *shuffled* order.

4. Tree Rotations



5. Insert 40. Rotate left at 36, left at 24, right at 48.



6. The standard **`insert`** algorithm creates a new leaf to hold the new value. How could we modify it so that the new value ends up at the *root* of our tree?

7. Building better trees: Randomized Binary Trees Idea: insert each new key "randomly" into the tree-so-far

   - Maybe it should become the new root
   - Maybe put it somewhere below the existing root

   But how often to do each?

   Answer: If the tree has n nodes **before** the insert,

   - do insert-at-root with probability $1/(n+1)$
   - otherwise, insert randomly into the appropriate child.