Tree-based Data Structures and Algorithms

BinaryTree::FindSmall()

How would you search for the smallest element in a *generic* binary tree?

BinarySearchTree::FindSmall()

How would you search for the smallest element in a binary *search* tree?

BST::FindNextLargest()

How would you find the next largest element in a BST based on the element at which you were currently positioned?

BST::Add(val) and Del(val)

How would you add something to a BST?

How would you delete something from a BST?

Question

Will the following algorithm work to determine whether a binary tree is a binary search tree if you pass in its root?

```
boolean testBST(Node root) {
    boolean answer = true;
    if (root.left != null) {
        answer = answer &&
            root.val >= root.left.val &&
            testBST(root.left);
    }
    if (root.right != null) {
        answer = answer &&
            root.val <= root.right.val &&
            testBST(root.right);
    }
    return answer;
}
```

How to write testBST?

This is left in part as a thought exercise for those who might find this question interesting.

It might be tempting to say that ***for each node*** you should check to see if the largest value in left subtree is smaller than the value in the node and then check to see if the minimum value in right subtree greater than it, but does that have a good runtime cost.

If you work out (or find) a solution that is correct, you should then analyze the runtime in terms of data comparisons. It should be ***n*** time really… It should also not use too much extra space… We've actually talked about something similar that will work…

Balanced Trees

In terms of height, the best binary tree is a complete binary tree.

Problem: It will be costly to maintain this property as new data is added to a complete binary search tree.

Typical Solution: Allow for a certain (very small) degree of imbalance.

AVL Trees

The AVL Tree (named for Adelson-Velskii and Landis) is an example of a "height-balanced" binary search tree.

- Any two subtrees of a node have heights that differ by at most one.

Is this constraint enough to guarantee that the height of the tree will be O(log*n*)?

AVL Tree Run-times

Search

- This is O(log*n*) since we have proven that the height of the tree is O(log*n*).

Insert

- This is also O(log*n*) but requires a bit of thought.
- Always insert at the leaf level and then rebalance. There are only a few cases that need to be handled.

Delete

- This can also be done in O(log*n*) time and also requires a bit of thought.
- Always delete at the leaf level. If the value to be deleted isn't at the leaf, find a value at the leaf level that can take the place of the one that you want to delete.

Other BSTs…

Red-Black Trees (balanced)

- The height is at most $2\log(n+1)$.
- Insertion and deletion is $O(\log n)$.

Splay Trees (not balanced)

- The height can get as bad as $O(n)$ in the worst case (but searching the tree actually helps rebalance the tree).
- Insertion, Deletion, and Searching have *amortized* runtimes of $O(\log n)$.
- In this data structure, the search target is moved to the root - this means that if you search for the same subset of things repeatedly, you get even better performance.