

CS375 Small Assignment 10

Due Nov 6th 10PM ET

Okay so I majorly screwed this assignment up. So let's try it again:

Let's revisit an old topic from back in the good old days of Data Structures: Binary Search Trees! As you hopefully recall:

- A **binary** tree is a system of nodes where 1 node is specifically marked as the root and every node has at most two children, where each child is designated as either the left or right child.
- A binary **search** tree is a binary tree where every node has a value and for every node v all nodes in the subtree rooted at the left child of v have value less than v 's value and all nodes in the subtree rooted at the right child of v have greater value.

For this assignment, we'll assume that Binary Search Trees are comprised of Nodes that have the following fields:

```
class Node {
    Node left , right ;
    int val ;
}
```

Recall the **add** method for Binary Search Trees:

```
def add(item):
    add(item, root, null)

def add(item, curNode, curParent):
    if curNode == null:
        curNode = new node with the given item

        if curParent == null:
            root = curNode
        else if item < curParent.val:
            curParent.left = curNode
        else:
            curParent.right = curNode

    else if item < curNode.val:
        add(item, curNode.left, curNode)

    else:
        add(item, curNode.right, curNode)
```

1. First, describe why the runtime of the above function is $O(H)$ where H is the height of the tree.
2. An interesting problem that comes up with Binary Search Trees is that they only work well when they are **balanced**: when $H = O(\log n)$. Without any self balancing techniques, many simple algorithms

like `add`, `get` or `contains` take $O(n)$ time instead of $O(\log n)$ time. So how can we guarantee log heights? The trick is to make sure that every node v satisfies the following rule: `balanceFactor(v) := height(v.left) - height(v.right) ≤ 1`. Such nodes that satisfy that rule we'll call *balanced*. So it seems like we'll want to compute `balanceFactor` quickly, which of course means we'd need to calculate `height` quickly. So let's talk about `height`:

```
def height(Node v):
    if v == None:
        return 0

    return 1 + max(height(v.left), height(v.right))
```

Argue the correctness of this algorithm and why the runtime of this algorithm is linear.

3. So now, imagine I have some function `fix(v)` that “fixes” an out-of-balance node v : it'll rearrange the subtree rooted at that tree so that all the nodes are “balanced”. I'll update my `add` method to make our tree balanced:

```
def add(item):
    add(item, root, null)

def add(item, curNode, curParent):
    if curNode == null:
        curNode = new node with the given item
        if curParent == null:
            root = curNode

        else if item < curParent.val:
            curParent.left = curNode

        else:
            curParent.right = curNode

    else if item < curNode.val:
        add(item, curNode.left, curNode)

    else:
        add(item, curNode.right, curNode)

    if |height(curNode.left) - height(curNode.right)| > 1:
        fix(curNode)
```

But there's a problem: even if `fix` takes constant time (which it can be made to do! e.g. AVL Trees), the runtime of our `add` method is no longer $O(H)$: why not?

4. By adding fields to the `Node` class, how do we fix the above method to take $O(H)$ time? Explain how this is basically the application of the ideas of dynamic programming to tree-like structures.