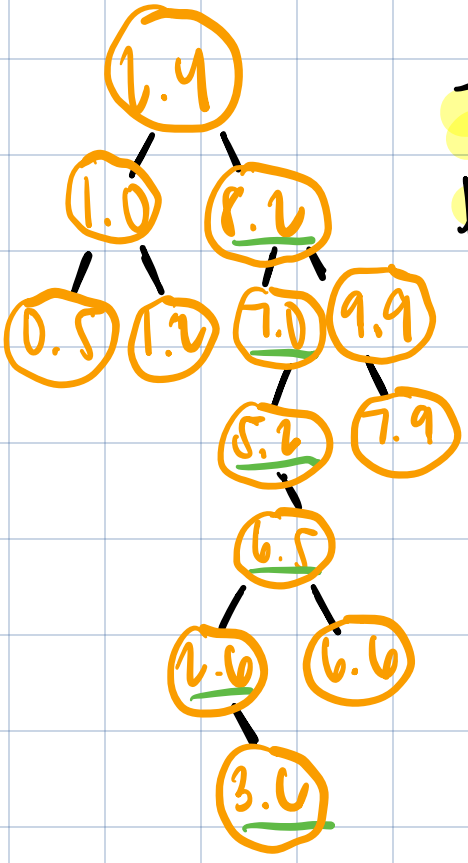
Problems:

1 (text) Type of Tree [10 points] Given each of the following arrays, create a BST by adding each element of the array in the order it appears in the array. Afterwards, indicate what is the height of the resulting tree.

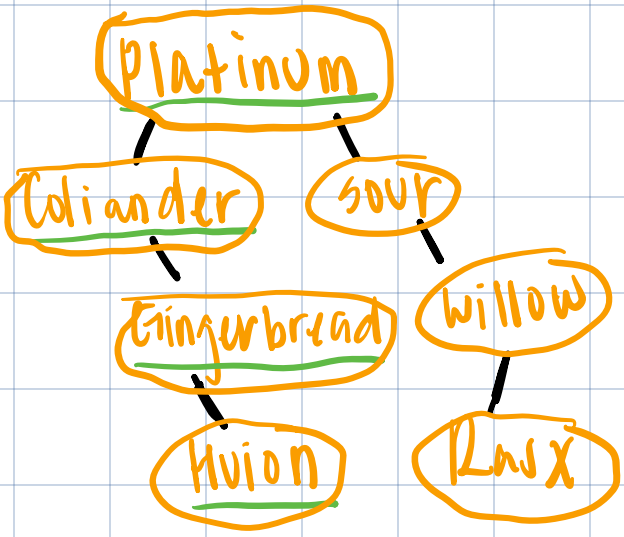
a) [2.4, 8.2, 7.0, 5.2, 1.0, 6.5, 2.6, 3.6, 7.9, 0.5, 6.6, 9.9, 1.2]
b) ["Platinum", "Coliander", "Sour", "Willow", "Rasx", "Gingerbread", "Hiuon"]
c) [1, 37, 11, 71, 62, 4, 61, 1, 11, 4]
d) [78, 76, 56, 43, 55, 77, 10, 93, 46, 79, 33, 8, 89, 100, 98,19]

(a.) [2.4, 8.2, 7.0, 1.0, 6.5, 2.6, 3.6, 7.9, 0.5, 6.6, 9.9, 1.2]



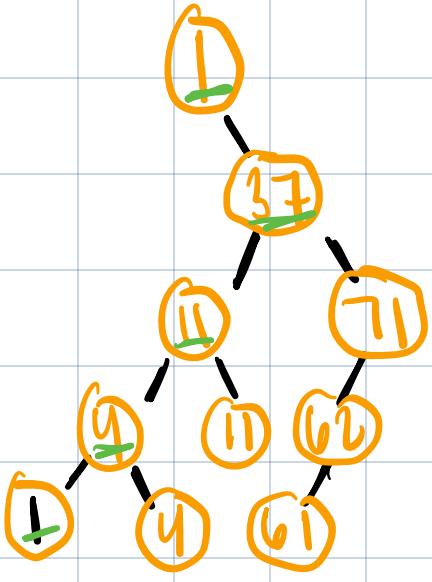
From root 2.4 down to 3.6 is 6 nodes → height = 5

(b.) ["Platinum", "Coliander", "Sour", "Willow", "Rasx", "Gingerbread", "Hivon"]



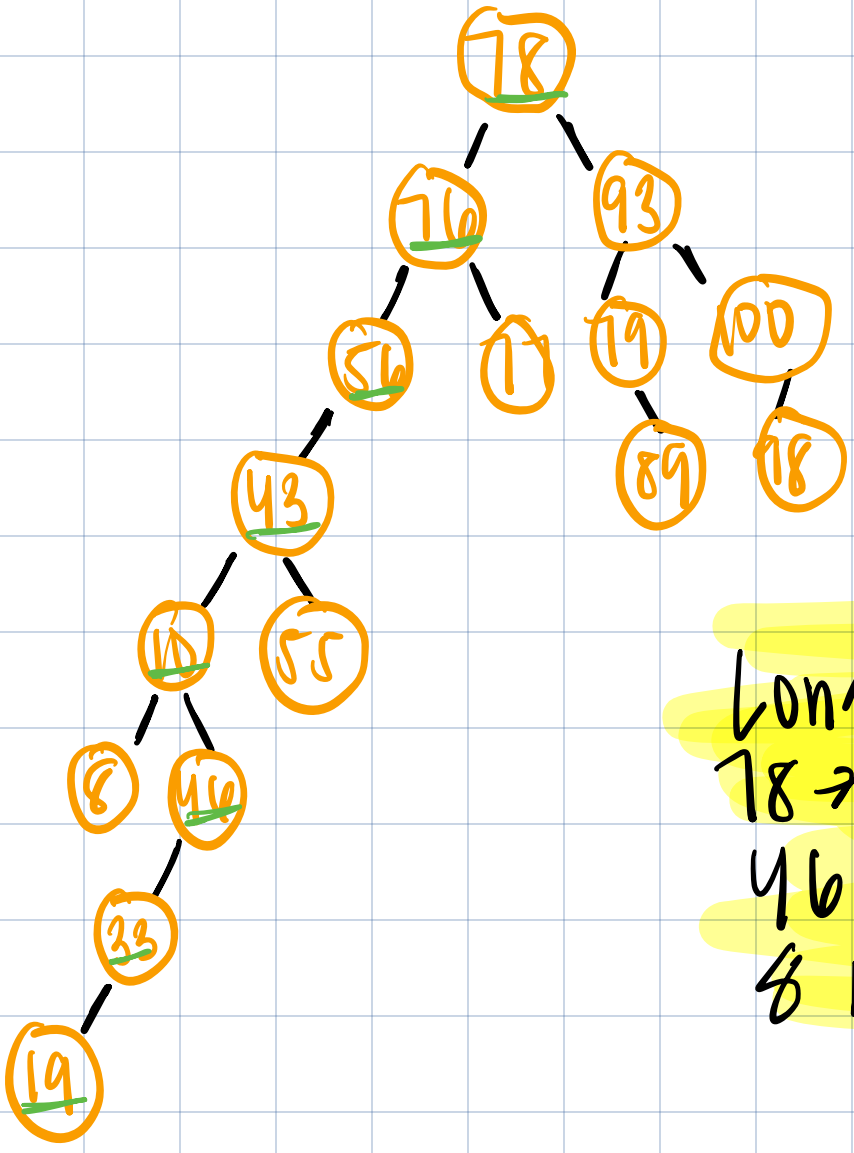
Longest path:
Platinum → Coliander → Gingerbread → Hvion.
Height = 3

(c.) [1, 37, 11, 71, 62, 4, 1, 11, 4]



Longest path: 1 → 37 → 11 → 4 → 1
5 nodes → height 4

(d) [78, 76, 56, 43, 55, 77, 10, 93, 46, 79, 33, 8, 89, 100, 98, 19]

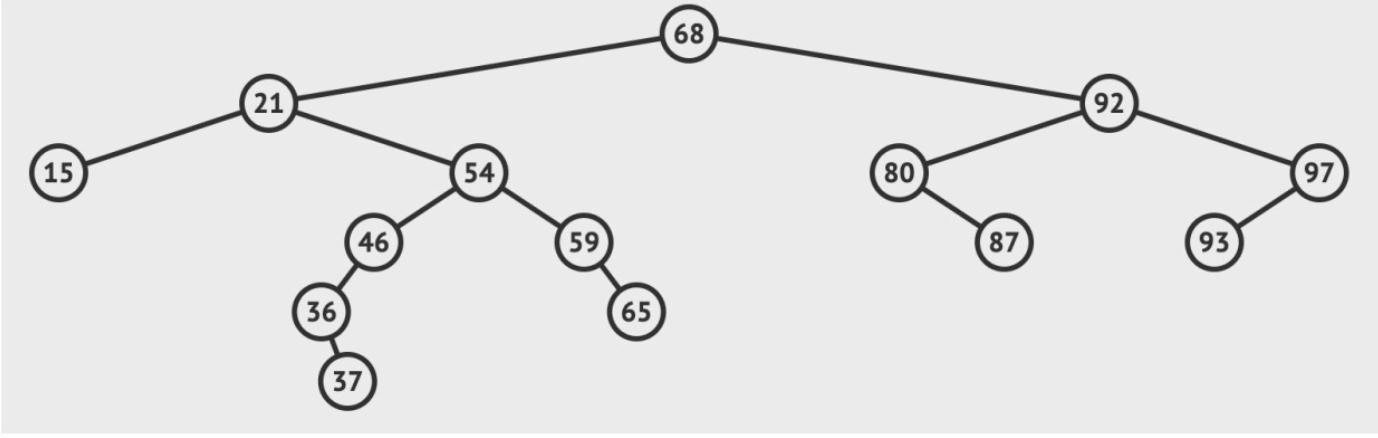


Longest path:
78 → 56 → 43 → 10 → 46 → 33 → 19
8 nodes → height=7

2 (text) BST Traversal [10 points] Given the following BST, in which the data on an empty node is 0.

a. **[6 points]** What will be the resulting tree after doing preorder and inorder traversal and applying the following operation on each node. Note round up the node values.

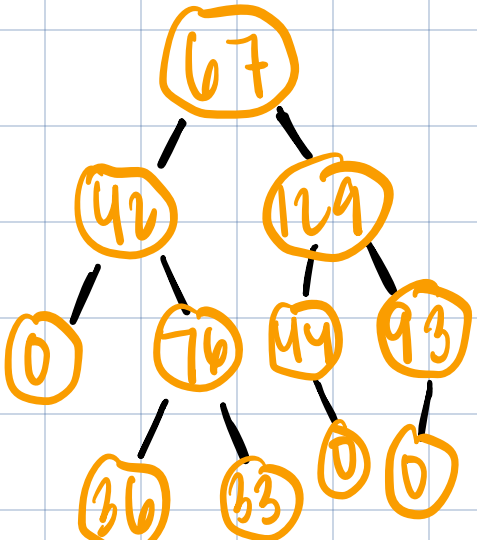
node.data = left.data + (right.data/2)



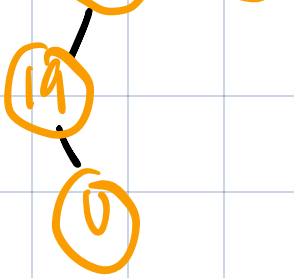
b. **[2 points]** Are the resulting trees BSTs?
c. **[2 points]** Are the resulting trees AVLs?

(a.)

node 68: 21 + (92 / 2) = 67
node 21: 15 + (54 / 2) = 42
node 15: 0 + 0 = 0
node 54: 46 + (59 / 2) = 76
node 46: 36 + (0 / 2) = 36
node 36: 0 + (37 / 2) = 19
node 37: 0 + (0 / 2) = 0
node 59: 0 + (65 / 2) = 33
node 65: 0 + (0 / 0) = 0
node 92: 80 + (97 / 2) = 129
node 80: 0 + (87 / 2) = 44
node 87: 0 + (0 / 0) = 0
node 97: 93 + (0 / 2) = 93
node 93: 0



(b.) No, the trees are not BST's. Some nodes end up having values that break the rules. Like, a child on the



right might be smaller than its parent, which shouldn't happen in BST's

(C.) I don't think they are AVL trees either. AVL's have to be balanced, meaning the left and right sides of every node can't differ too much in height. In this case, some parts of the tree are way deeper than others.

7 (text) Algorithm Analysis [20 points]
For each of the code you wrote for problems 3-6, explain its time complexity and space complexity for each problem using Big-O and Big-Ω notation. Explain how you arrived at your answer.

Problem 4:

The time complexity of my code is O(n) in the worst case because I might need to check every spot in the array to see if the number I am looking for

is there. I am using recursion to move through the tree, so each spot could lead to more work. The best case is $\Omega(1)$ if the target is found right at the beginning.

The space complexity is also $O(n)$ in the worst case because each recursive call takes up space, and I think if the tree is really deep, it could use alot of it. In the best case, if I find the target right away, the space used is just $\Omega(1)$ since I dont need to any further.

Problem 5:

The time complexity is $O(n)$ because I always have to go through every string in the array to check how long it is. Even if none of them are added to the result, I still have to look at each one,

so the best case is also $\Omega(n)$.

the space complexity is $O(n)$ in the worst case if all strings are short enough and get added to the list. In the best case, if no strings are short enough, the list stays empty, so the space used is only $\Omega(1)$.

problem 6:

The time complexity of my deleteInRange method is $O(n)$ in the worst case because I might need to look at every node in the tree to check if its value is inside the range. Even if I don't delete many nodes, I still have to check them all so the best case is $\Omega(n)$.

the space complexity is $O(n)$ in the worst case because the method uses recursion, and if the tree is really

unbalanced (like all nodes going in one direction), it keeps calling itself over and over without stopping, using more memory each time. In the best case, if the tree is balanced, the calls don't go as deep, so it only uses a little memory - that's $\Omega(\log n)$.