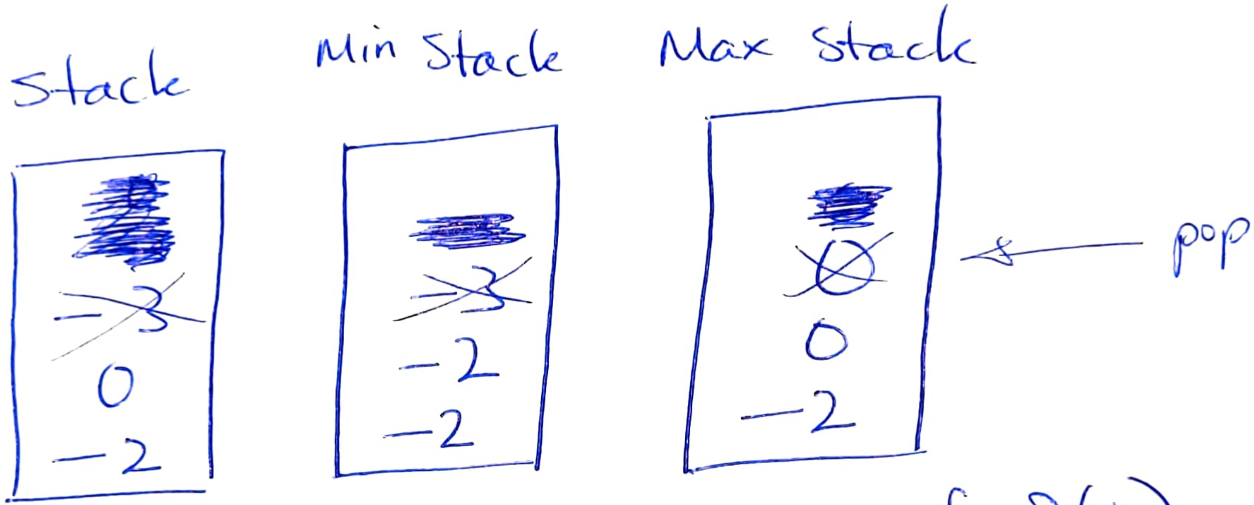
Implement a stack w/ Max API

8.1 7/8/22

- For this problem, we need to define another stack w/ the minimum/maximum in the order of the values pushed/popped



- This way, we can have runtime of $O(1)$ since we are comparing the pushed value w/ the ~~previous~~ min/max value at the cost of $O(n)$ space.

- Here, we create a parallel data structure.

- This problem is less algorithmic and has more to do w/ using OOP and designing data structure.

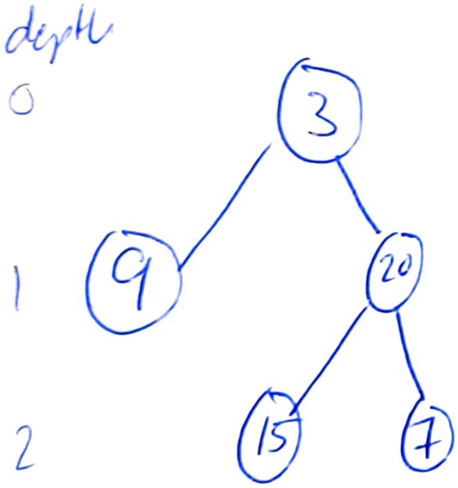
Alternative Approaches

- Iterate through entire stack, pop every element and push them back in to check the values and return the min/max. $\longrightarrow O(n), O(1)$

- $O(\log n), O(n)$ can be achieved using heap or BST, and a hash table.

```
Class MaxStack(object):
    def __init__(self):
        self.stack = []
        self.maxes = []
    def push(self, val):
        self.stack.append(val)
        # IF the max value from the maxes stack
          is greater, add that to stack otherwise,
          push that value that is greater than the max
        if self.maxes and self.maxes[-1] > val:
            self.maxes.append(self.maxes[-1])
        else:
            self.maxes.append(val)
    def pop(self):
        if self.maxes:              # Need to make
            self.maxes.pop()        sure we also pop val
                                    from maxes stack
        return self.stack.pop()
    def max(self):
        return self.maxes[-1]
```

First set an empty list to return. Store the first node. Then for each depth, store the children node to a queue, ~~and~~; store the elements in the queue to a list and append that to the result. Do this until all nodes are processed.



queue	result
[9,20]	[[3]]
[15,7]	[[3],[9,20]]
[ ]	[[3],[9,20],[15,7]]

```
def binary_tree_depth_order(tree: BinaryTreeNode)
    -> List[List[int]]:

    result: List[List[int]] = []
    if not tree:
        return result

    curr_depth_nodes = [tree]
    while curr_depth_nodes:
        nodes_at_depth = []
        for node in curr_depth_nodes:
            nodes_at_depth.append(node)
        result.append(nodes_at_depth)

        next_depth = []
        for node in curr_depth_nodes:
            for child in (node.left, node.right):
                if child:
                    next_depth.append(child)
        curr_depth_nodes = []   #empty the queue
```

Queue → curr_depth_nodes

~~# Nodes @ current depth~~

append nodes in the depth to the result

Process the nodes in the next depth

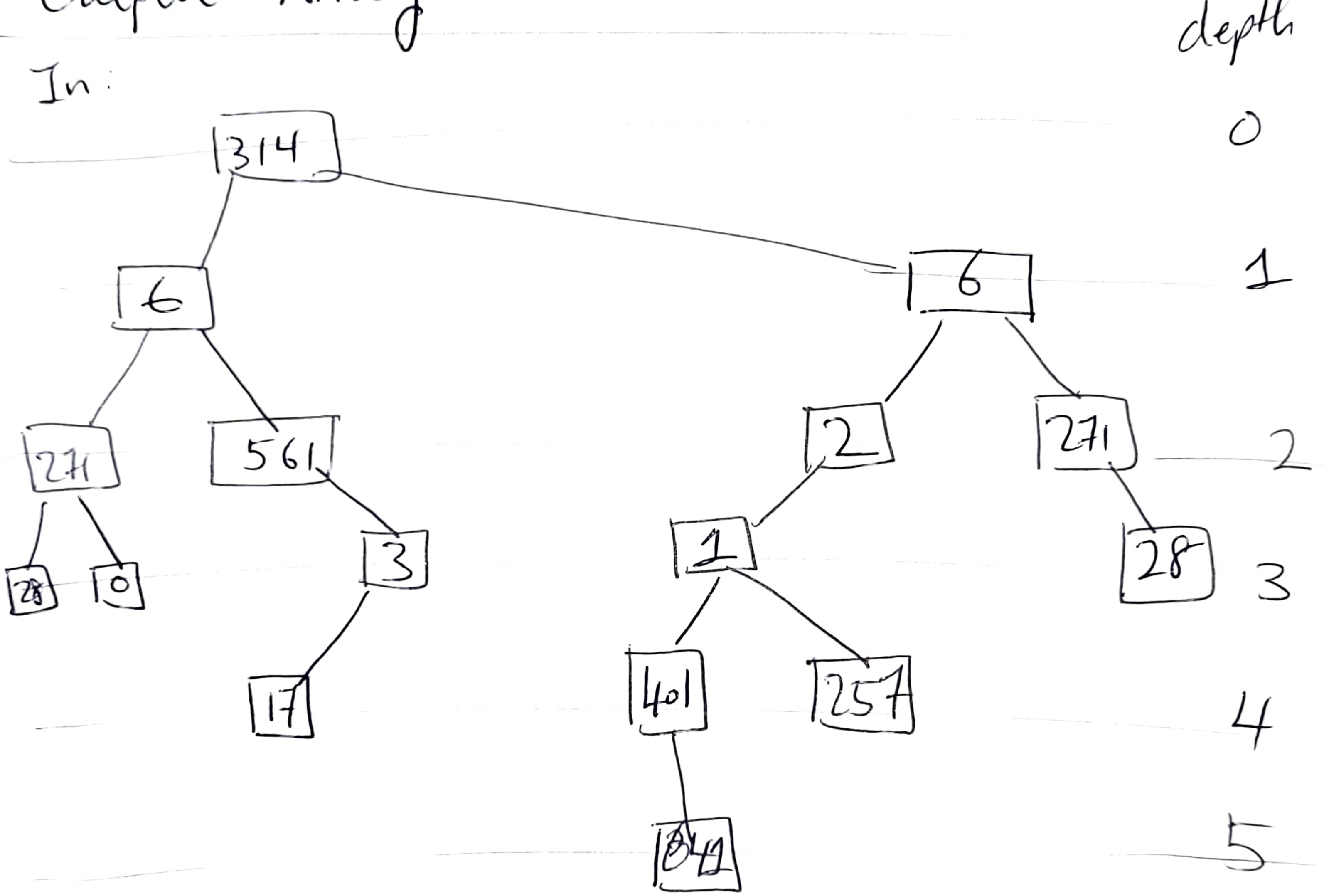
```
    # prepare the next queue
    for node in next_depth:
        curr_depth_nodes.append(node)

return result
```

Input: Binary Tree

Output: Array

In:



Should return:

$\langle \langle 314 \rangle, \langle 6, 6 \rangle, \langle 271, 561, 2, 271 \rangle \ldots \rangle$

(d0) (d1) (d2)

· So we are return a List[List[int]] ~~where the elements are entries~~ for each depth.

· I don't know how the bin. tree input looks like.

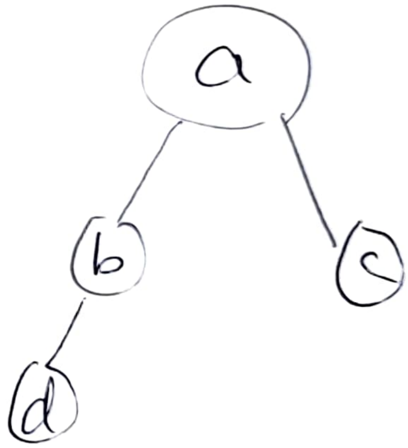
Bin Tree will look like..

None

class Node(object):
    def __init__(self, ~~val~~ data, left=None, right=None)
        self.data = data
        self.left = left
        self.right = right

and the ~~i~~ inputs:

root = Node('a')
root.left = Node('b')
root.left.left = Node('d')
root.right = Node('c')



(13min)

- So the problem is asking for a class that has a method for getting the depth and for each depth, append all the values into a list and append to another list to return the list of values. ~~upto~~ ($\forall$ depths)

~~Since~~

- <u>Boundary/edge cases</u>:
  ↳ Since this a binary tree, at each depth the max amount of elem is 2^{depth}.

(*) How can I use queue in this?

(*) What is the Naive Approach?

→ Maybe I can append to a list the max amount of elem. checking each branch, where node.left ≠ null; ~~result.app~~ temp_list.append(node.left)

and then result.append(temp_list)

→ Since we are check all branches → $O(2^{max(depth)})$ time, $O(depth)$ space

26min Quit

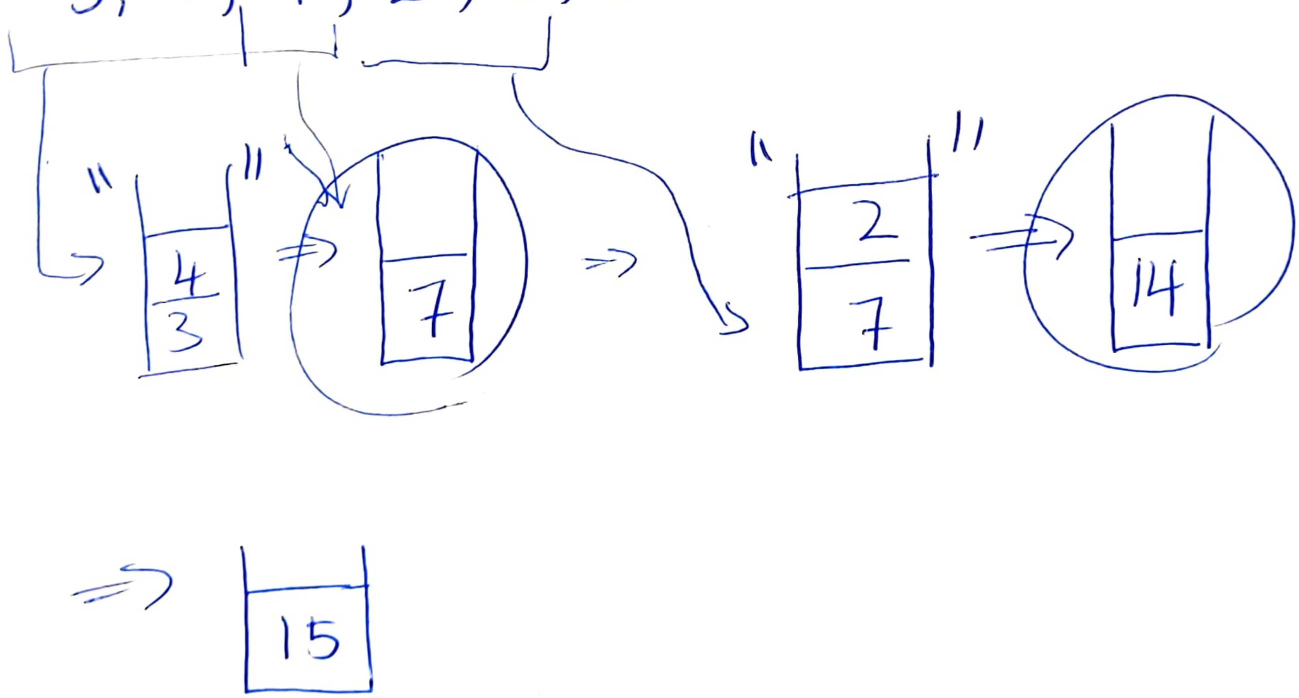
3/3

$\boxed{8.2}$ 7/8/22

Evaluation

↳ The design was similar but not correct. When doing the computation, I had to observe the A and B by popping them and do the computation. Then push the value back into the list.

"3, 4, +, 2, x, 1, +"



(*) For each number value (not operators) we convert and push/append as integer.

1/1

$\boxed{8.2}$

Input : RPN ; Output : ~~int~~ ~~float~~ Number

$$\text{"}\underbrace{\underbrace{\underbrace{3,\ 4,\ +}_{=7},\ 2,\ \times}_{=14},\ 1,\ +}_{=15}\text{"}$$

$$\text{"}\underbrace{\underbrace{-641,\ 6,\ /}_{=106.8\overline{33}},\ 28,\ /}_{\approx 3.82}\text{"}$$

(*float)

Must be an integer, round up or down?

- Are we returning an $\mathbb{Z}$? or float?

The Naive Approach would be to iterate through the string for a math symbol, convert the numbers before the symbol and compute the calculation. We can take that number value and traverse the string until next math symbol or (the) end. We can do this recursively.

$\boxed{\cancel{O(n)},\ O(1)}$

string ?

1/2

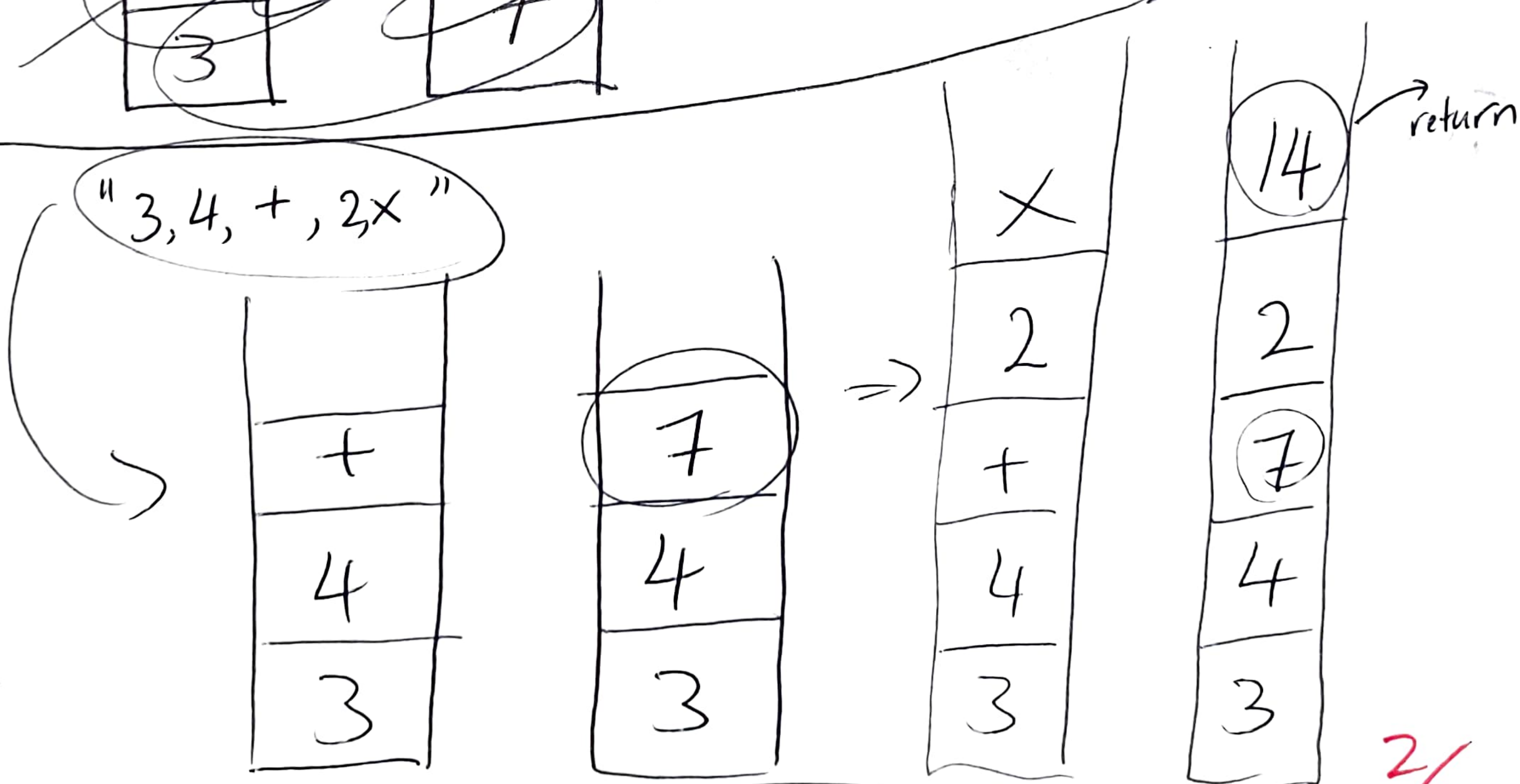
13min

We might be able to use stacks to store the computed value and compute again when there is a symbol.

To compute in accordance to the symbol, we might have to use a dictionary.

LIFO

This can reduce the time but use extra memory ($O(1)$, $O(n)$)



20min Quit

2/2