QUESTION 1. Nested List Recursion (7 marks)

PART A. Example Calls (2 marks)

Write down the expected results of calling:

get_element_counts('A')

{'A': 1}

get_element_counts(['A', 'B'])

{'A': 1, 'B': 1}

get_element_counts([[['B', 'C'], 'A'], 'A'])

{'B': 1, 'C': 1, 'A': 2}

Briefly describe how we would use the results of the calls above to get the result of calling:

get_element_counts(['A', ['A', 'B'], [['B', 'C'], 'A'], 'A'])

"We would combine the dictionaries by adding their keys and values together."

(Or something that indicates that they know that they’re supposed to merge the dictionaries.)

GRADING SCHEME

- 0.5 marks per part (If they leave it blank, they get 20% of the marks = 0.1 marks)
  - Ignore the order of the keys in the dictionary; as long as the right keys and values are there, it's fine.
PART B. get_element_counts (5 marks)

```python
def get_element_counts(x: Any) -> dict:
    """
    Return a dictionary containing the counts of each non-list item
    that appears in x.
    >>> get_element_counts('C')
    {'C': 1}
    >>> get_element_counts(['D', 'C', 'C'])
    {'D': 1, 'C': 2}
    >>> get_element_counts([10, [20, 'A'], [['B', 'A'], 20]])
    {10: 1, 20: 2, 'A': 2, 'B': 1}
    """
    if not isinstance(x, list):
        return {x: 1}
    new_dict = {}
    for item in x:
        d = get_element_counts(item)
        for key in d:
            if key in new_dict:
                new_dict[key] += d[key]
            else:
                new_dict[key] = d[key]
    return new_dict
```

GRADING SCHEME (20% = 1; if they leave it blank/cannot answer)

(Letter): Marks
Write the letters along the side and the mark they got for each letter.

- **F**: 1 mark for having the correct if-statement for the base case
  - They should be using if not isinstance(x, list) or if type(x) != list
- **B**: 1 mark for returning the correct dictionary in the base case
  - This should always be {x: 1}
- **R**: 1 mark for making the recursive calls correctly
- **U**: 1 mark for updating the dictionary correctly (adding the count from the recursive calls to it)
  - 0.5 marks if they didn't check if the key was in already or not and just did:
    new_dict[key] = d[key] (or += d[key])
  - 0.5 marks if they checked whether a key was in the dictionary but added 1
    instead of d[key]. If they didn't check whether it was in the dictionary, they get
    0 for this part.
- **C**: 1 mark for returning the correct dictionary in the end.
  - This is contingent on the other categories all being correct (the only errors
    being ones that don't fall into the above categories; i.e. minor coding errors).
- -0.5 marks for other minor coding errors.
QUESTION 2. Trees (10 marks)

PART A. Example Calls (3 marks)

Suppose we have the following Tree t:

![Tree Diagram]

Draw a box around each of the subtrees of t (do not draw boxes around the subtrees of the subtrees) and give it a label (e.g. "A").

For each of the subtrees of t, what should calling the `get_level_items()` method return? Write this below in relation to the label you gave the subtree.
(e.g. A: `[whatever A.get_level_items() returns]`)

A: `[[3], [4, 1, 6]]`

B: `[[5]]`

C: `[[7], [8, 2], [0]]`

What should calling `t.get_level_items()` return?

`[[9], [3, 5, 7], [4, 1, 6, 8, 2], [0]]`

GRADING SCHEME (20% = 0.6; if they leave it blank/cannot answer)

- 1 mark for drawing the right subtrees
  - 0.5 marks for each incorrect subtree
- 1 mark for getting the right results for calling `get_level_items()` on the subtrees they labelled
- 1 mark for correctly writing what `t.get_level_items()` should return
PART B. get_level_items (7 marks)

There are 3 approaches students may have taken; each of them have their own grading scheme.

Solution 1: Using only recursion

def get_level_items(self) -> List[List[Any]]:
    levels = []

    recursive_calls = [item.get_level_items() for item in self.children]
    for lst in recursive_calls:
        for i in range(len(lst)):
            if i >= len(levels):
                levels.append(lst[i])
            else:
                levels[i].extend(lst[i])

    return [[self.value]] + levels

GRADING SCHEME (20% = 1.4; if they leave it blank/cannot answer)

[Letter]: Marks
Write the letters along the side and the mark they got for each letter.

- R: 1 mark for making recursive calls on each of the children
- C: 1 mark for going through each of the children's recursive call's results
- A: 1 mark for adding new levels in properly
  - I.e. If they don't have a list for a certain level/depth yet, they add that level in.
- E: 1 mark for extending existing levels properly
  - I.e. If they have a list for a certain level/depth already, they extend that list.
- S: 1 mark for adding [self.value] to the front of the list they're returning.
- L: 1 mark for making sure the levels are a list of lists.
- C: 1 mark for returning the correct list in the end
  - This is contingent on the other categories being correct aside from minor errors.
- -0.5 marks if they treat get_level_items() as a function instead of a method.
Solution 2: Using a Queue

def get_level_items(self) -> List[List[Any]]:
    levels = []
    current_level = []
    q = [self, "END"]

    while q:
        item = q.pop(0)
        if item == "END":
            levels.append(current_level)
            current_level = []
            if q:
                q.append("END")
            else:
                current_level.append(item.value)
                for child in item.children:
                    q.append(child)

    return levels

GRADING SCHEME (20% = 1.4; if they leave it blank/cannot answer)

[Letter]: Marks
Write the letters along the side and the mark they got for each letter.

- **Q**: 1 mark for starting the Queue with self and "END" (or some delimiter)
- **L**: 1 mark for keeping track of the items in the current level (i.e. in a list)
- **V**: 1 mark for adding self.value to the current level whenever a Tree is popped
- **C**: 1 mark for adding each of the children of the popped item to the Queue
  - If they use self.children instead of something like item.children, they don't get this mark (but can get the others)
- **E**: 1 mark for adding "END" (or some delimiter) after they're done with each level/whenever they see "END"
  - -0.5 if they don't check whether the Queue is empty or not before adding "END" back in.
- **R**: 1 mark for resetting the list of items in the current level whenever they see "END"
- **D**: 1 mark for adding the list of items at the current level to the list of levels whenever they see "END"
Solution 3: Using Helper Methods

def get_level_items(self) -> List[Any]:
    height = self.get_height()

    levels = []
    for i in range(height):
        levels.append(self.get_items_at_depth(i))
    return levels

def get_height(self):
    if self.children == []: return 1
    return 1 + max([child.get_height() for child in self.children])

def get_items_at_depth(self, depth):
    if depth == 0:
        return [self.value]
    return sum([child.get_items_at_depth(depth - 1) for child in self.children], [])

GRADING SCHEME (20% = 1.4; if they leave it blank/cannot answer)

[Letter]: Marks

Write the letters along the side and the mark they got for each letter.

- **For the 'items at depth' helper:**
  - **F:** 1 mark for having the correct base case condition (depth == 0)
  - **B:** 1 mark for returning the correct thing at the base case ([self.value])
  - **D:** 1 mark for decreasing depth by 1 in their recursive calls
  - **R:** 1 mark for returning the correct list in the end (i.e. they get the sum of the results of their recursive calls)

- **For the height/max depth helper:**
  - Not needed if they check whether the 'items at depth' helper returns an empty list or not. They can get both of these marks.
  - **H:** 1 mark for returning the height (or max depth) correctly in the base case.
  - **M:** 1 mark for getting the max of the recursive calls and returning that + 1.

- **In get_level_items:**
  - **C:** 1 mark for returning the correct list in the end.
    - This is contingent on their helper function(s) working correctly.
    - If they use max depth, they must loop for range(max_depth + 1) to be inclusive of the last depth.
QUESTION 3. Binary Trees (8 marks)

PART A. Traversals (3 marks)

Suppose we have the following Binary Tree:

Write down the values of the nodes if the tree is visited using a **pre-order** traversal.

1, 2, 4, 8, 9, 5, 10, 3, 6, 11, 7

Write down the values of the nodes if the tree is visited using a **post-order** traversal.

8, 9, 4, 10, 5, 2, 11, 6, 7, 3, 1

Write down the values of the nodes if the tree is visited using an **in-order** traversal.

8, 4, 9, 2, 10, 5, 1, 6, 11, 3, 7

**GRADING SCHEME** (20% = 0.2; if they leave it blank/cannot answer)

- 1 mark for each traversal (all or nothing)
def add_in_order(t: Union['BinaryTree', None]) -> str:
    if t is None:
        return ''
    return add_in_order(t.left) + t.value + add_in_order(t.right)

GRADING SCHEME (20% = 1; if they leave it blank/cannot answer)

- **N**: 1 mark for handling the case where t is None (and returning '')
- **B**: 1 mark for returning the correct string in the base case
  - If they used 't is None' as the base case, they get this mark.
  - This is mostly for the case where they have a non-None base case.
- **R**: 1 mark for making the correct recursive calls to the subtrees
  - 0.5 marks for doing add_in_order(t.left.value) instead of add_in_order(t.left),
    but they also lose the mark for S.
- **I**: 1 mark for having the right order (left, value, right)
- **S**: 1 mark for returning the right string.
- -0.5 marks for treating add_in_order as a method instead of a function.
- -0.5 marks for each minor coding error