

TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)

Personal Member ID#: 117667

Session 2: Binary Trees

Session Overview

Students will apply advanced techniques in tree traversal and restructuring to tackle challenges involving balanced trees, pathfinding, and tree transformations. Key topics covered include tree mirroring, root-to-leaf path sums, subtree identification, and handling tree-based inventory data.

You can find all resources from today including session slide decks, session recordings, and more on the resources tab



Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.



Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
 - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

► Note on Expectations

Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

► Comment on UPI

► UPI Example

Note: Testing your Binary Tree (Printing)

To keep the amount of starter code manageable, we have chosen not to include a function to print a binary tree as part of each relevant problem statement. You may instead copy the function in the drop-down below `print_tree()` and use it as needed while you complete the problem sets.

▼ Print Binary Tree Function

Accepts the root of a binary tree and prints out the values of each node level by level from left to right. Values of `None` are used to indicate a null child node between non-null children on the same level. Prints `"Empty"` for an empty tree.

```

from collections import deque

# Tree Node class
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def print_tree(root):
    if not root:
        return "Empty"
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)
    while result and result[-1] is None:
        result.pop()
    print(result)

```

Example Usage:

```

"""
      1
     / \
    2   3
   /   / \
  4   5  6
"""

root = Node(1, Node(2, Node(4)), Node(3, Node(5), Node(6)))

print_tree(root)
print_tree(None)

```

Example Output:

```

[1, 2, 3, 4, None, 5, 6]
'Empty'

```

 **Note: Testing your Binary Tree (Generating a Tree)**

Now that you have practice manually building trees for testing in previous sessions, we are providing a function that builds binary trees based off of a list of values to speed up the testing process. We have chosen not to include this function in the starter code for each problem to keep the length of problems manageable. You may instead copy the function in the drop-down below `build_tree()` and use it as needed while you complete the problem sets.

▼ Build Binary Tree Function

Takes in a list `values` where each element in the list corresponds to a node in the binary tree you would like to build. The values should be in level order (from top to bottom, left to right). Use `None` to indicate a null child between non-null children on the same level.

Some problems may ask you to build a tree where nodes have both keys and values. This function may be used to build trees with just values *and* trees with both keys and values:

- If building a tree with only values, `values` should be given in the form:
`[value1, value2, value3, ...]`.
- If building a tree with both keys and values `values` should be given in the form
`[(key1, value1), (key2, value2), (key3, value3), ...]`.

Returns the `root` of the binary tree made from `values`.

```

from collections import deque

# Tree Node class
class TreeNode:
    def __init__(self, value, key=None, left=None, right=None):
        self.key = key
        self.val = value
        self.left = left
        self.right = right

def build_tree(values):
    if not values:
        return None

    def get_key_value(item):
        if isinstance(item, tuple):
            return item[0], item[1]
        else:
            return None, item

    key, value = get_key_value(values[0])
    root = TreeNode(value, key)
    queue = deque([root])
    index = 1

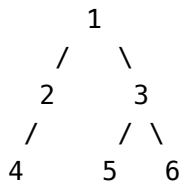
    while queue:
        node = queue.popleft()
        if index < len(values) and values[index] is not None:
            left_key, left_value = get_key_value(values[index])
            node.left = TreeNode(left_value, left_key)
            queue.append(node.left)
        index += 1
        if index < len(values) and values[index] is not None:
            right_key, right_value = get_key_value(values[index])
            node.right = TreeNode(right_value, right_key)
            queue.append(node.right)
        index += 1

    return root

```

Example Usage:

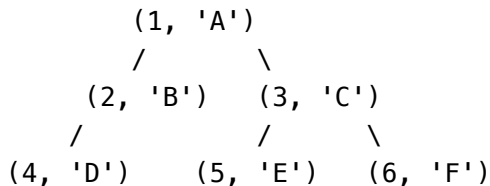
"""



"""

```
tree_with_just_values = [1, 2, 3, 4, None, 5, 6]
val_tree = build_tree(tree_with_just_values)
```

"""



"""

```
tree_with_keys_and_values = [(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), None, (5, 'E'), (6, 'F')]
key_val_tree = build_tree(tree_with_keys_and_values)
```

```
# Using print_tree() function included above
print_tree(val_tree)
print_tree(key_val_tree) # Only values will be printed
```

Example Output:

```
[1, 2, 3, 4, None, 5, 6]
['A', 'B', 'C', 'D', None, 'E', 'F']
```

Breakout Problems Session 2

▼ Standard Problem Set Version 1

Problem 1: Balanced Baked Goods Display

Given the root of a binary tree `display` representing the baked goods on display at your store, return `True` if the tree is balanced and `False` otherwise.

A balanced display is a binary tree in which the difference in the height of the two subtrees of every node never exceeds one.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```


class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def is_balanced(display):
    pass

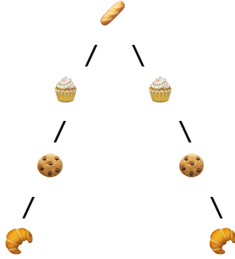
```

Example Usage:

```

"""

"""

# Using build_tree() function included at top of page
baked_goods = ["🍰", "🍪", "🍩", "🍞", "🧁"]
display1 = build_tree(baked_goods)

"""

"""

baked_goods = ["🍞", "🧁", "🧁", "🍪", None, None, "🍪", "🍪", None, None, "🍪"]
display2 = build_tree(baked_goods)

print(is_balanced(display1))
print(is_balanced(display2))

```

Example Output:

```

True
False

```

Problem 2: Sum of Cookies Sold Each Day

Your bakery stores each customer order in a binary tree, where each node represents a different customer's order and each node value represents the number of cookies ordered. Each level of the tree represents the orders for a given day.

Given the root of a binary tree `orders`, return a list of the sums of all cookies ordered in each day (level) of the tree.

Evaluate the time complexity of your solution. Define your variables and give a rationale as to why you believe your solution has the stated time complexity.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def sum_each_days_orders(orders):
    pass
```

Example Usage:

```
"""
    4
   / \
  2   6
 / \
1   3
"""

# Using build_tree() function included at top of page
order_sizes = [4, 2, 6, 1, 3]
orders = build_tree(order_sizes)

print(sum_each_days_orders(orders))
```

Example Output:

```
[4, 8, 4]
```

Problem 3: Sweetness Difference

You are given the root of a binary tree `chocolates` where each node represents a chocolate in a box of chocolates and each node value represents the sweetness level of the chocolate. Write a function that returns a list of the **absolute differences** between the highest and lowest sweetness levels in each row of the chocolate box.

The sweetness difference in a row with only one chocolate is `0`.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def sweet_difference(chocolates):
    pass
```

Example Usage:

```
"""
    3
   / \
  9  20
   / \
  15  7
"""

# Using build_tree() function included at top of page
sweetness_levels1 = [3, 9, 20, None, None, 15, 7]
chocolate_box1 = build_tree(sweetness_levels)

"""
    1
   / \
  2   3
 / \  \
4  5  6
"""

sweetness_levels2 = [1, 2, 3, 4, 5, None, 6]
chocolate_box2 = build_tree(sweetness_levels)

print(sweet_difference(chocolate_box1))
print(sweet_difference(chocolate_box2))
```

Example Output:

```
[0, 11, 8]
[0, 1, 2]
```

Problem 4: Transformable Bakery Orders

In your bakery, customer orders are each represented by a binary tree. The value of each node in the tree represents a type of cupcake, and the tree structure represents how the order is organized in the delivery box. Sometimes, orders don't get picked up.

Given two orders, you want to see if you can rearrange the first order that didn't get picked up into the second order so as not to waste any cupcakes. You can swap the left and right subtrees of any cupcake (node) in the order.

Given the roots of two binary trees `order1` and `order2`, write a function `can_rearrange_orders()` that returns `True` if the tree represented by `order1` can be rearranged to match the tree represented by `order2` by doing any number of swaps of `order1`'s left and right branches.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
class TreeNode():
    def __init__(self, flavor, left=None, right=None):
        self.val = flavor
        self.left = left
        self.right = right

def can_rearrange_orders(order1, order2):
    pass
```

Example Usage:

```
"""
      Red Velvet
     /      \
  Vanilla    Lemon
   /  \      /  \
 Ube  Almond Chai Carrot
           /  \      \
           Chai Maple Smore
"""

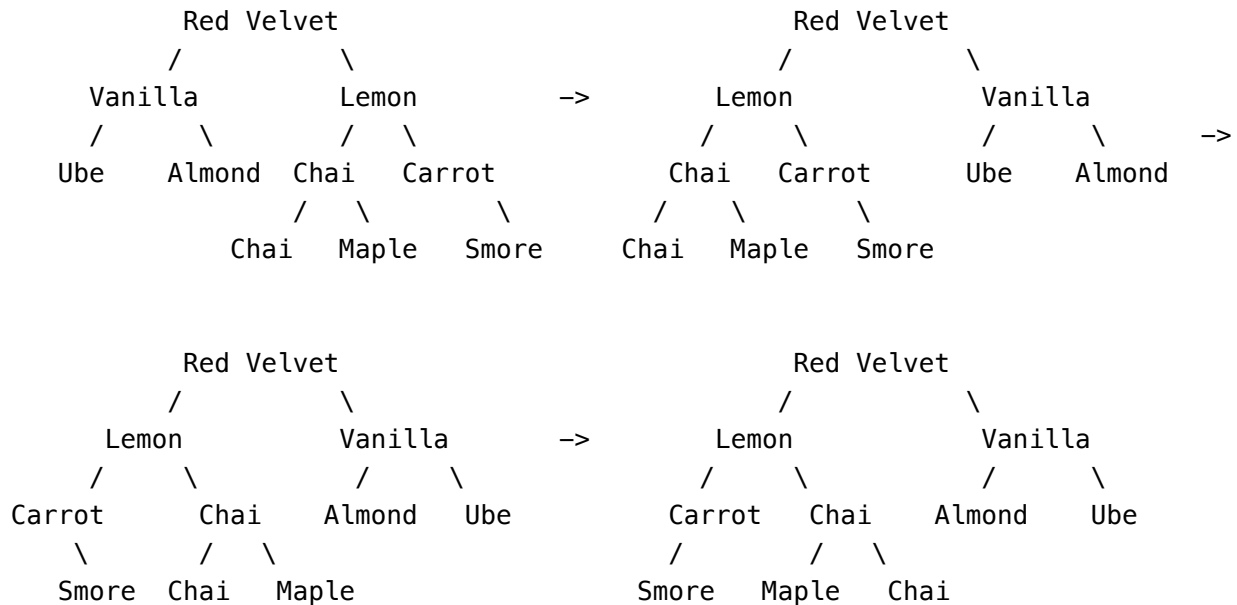
# Using build_tree() function included at top of page
flavors1 = ["Red Velvet", "Vanilla", "Lemon", "Ube", "Almond", "Chai", "Carrot",
            None, None, None, None, "Chai", "Maple", None, "Smore"]
flavors2 = ["Red Velvet", "Lemon", "Vanilla", "Carrot", "Chai", "Almond", "Ube", "Sm
order1 = build_tree(flavors1)
order2 = build_tree(flavors2)

can_rearrange_orders(order1, order2)
```

Example Output:

True

Explanation:



Problem 5: Larger Order Tree

You have the root of a binary search tree `orders`, where each node in the tree represents an order and each node's value represents the number of cupcakes the customer ordered. Convert the tree to a 'larger order tree' such that the value of each node in tree is equal to its original value plus the sum of all node values greater than it.

As a reminder a BST satisfies the following constraints:

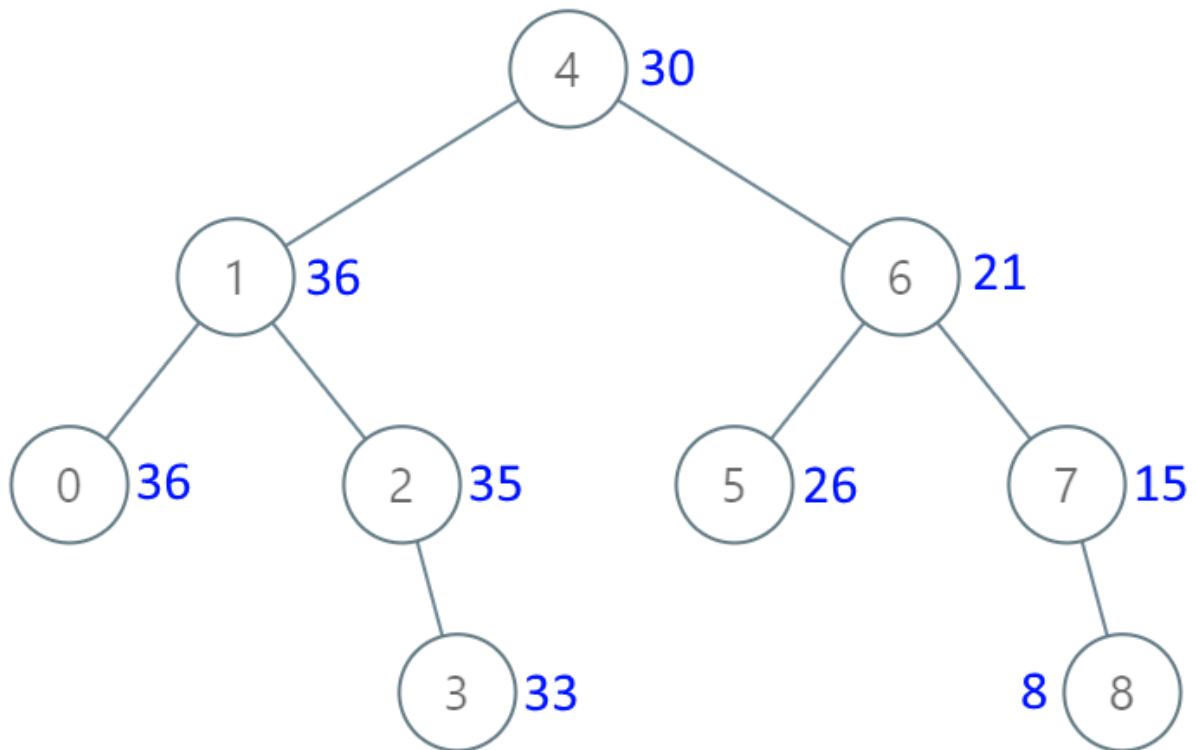
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Evaluate the time and space complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity. Assume the input tree is balanced when calculating time complexity.

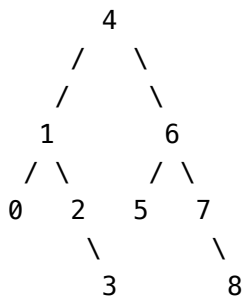
```
class TreeNode():
    def __init__(self, order_size, left=None, right=None):
        self.val = order_size
        self.left = left
        self.right = right

def larger_order_tree(orders):
    pass
```

Examples Usage:



.....



.....

```

# using build_tree() function included at top of page
order_sizes = [4,1,6,0,2,5,7,None,None,None,3,None,None,None,8]
orders = build_tree(order_sizes)

# using print_tree() function included at top of page
print_tree(larger_order_tree(orders))

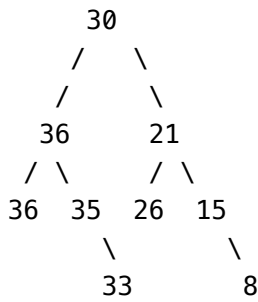
```

Example Output:

```
[30,36,21,36,35,26,15,None,None,None,33,None,None,None,8]
```

Explanation:

Larger Order Tree:



Problem 6: Find Next Order to Fulfill Today

You store each customer order at your bakery in a binary tree where each node represents a different order. Each level of the tree represents a different day's orders. Given the root of a binary tree `order_tree` and an `Treenode` object `order` representing the order you are currently fulfilling, return the next order to fulfill that day. The next order to fulfill is the nearest node on the same level. Return `None` if `order` is the last order of the day (rightmost node of the level).

Note: Because we must pass in a reference to a node in the tree, you cannot use the `build_tree()` function for testing. You must manually create the tree.

```
class Treenode():
    def __init__(self, order, left=None, right=None):
        self.val = order
        self.left = left
        self.right = right

def find_next_order(order_tree, order):
    pass
```

Example Usage:

```

      """"
          Cupcakes
        /      \
    Macaron    Cookies
      \      /  \
      Cake   Eclair  Croissant
      """"

cupcakes = TreeNode("Cupcakes")
macaron = TreeNode("Macaron")
cookies = TreeNode("Cookies")
cake = TreeNode("Cake")
eclair = TreeNode("Eclair")
croissant = TreeNode("Croissant")

cupcakes.left, cupcakes.right = macaron, cookies
macaron.right = cake
cookies.left, cookies.right = eclair, croissant

next_order1 = find_next_order(cupcakes, cake)
next_order2 = find_next_order(cupcakes, cookies)
print(next_order1.val)
print(next_order2.val)

```

Example Output:

```

Eclair
None

```

[Close Section](#)

▼ Standard Problem Set Version 2

Problem 1: Haunted Mirror

A vampire has come to stay at the haunted hotel, but he can't see his reflection! What's more, he doesn't seem to be able to see the reflection of anything in the mirror! He's asked you to come to his aid and help him see the reflections of different things.

Given the root of a binary tree `vampire`, return the mirror image of the tree. The mirror image of a tree is obtained by flipping the tree along its vertical axis, meaning that the left and right children of all non-leaf nodes are swapped.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

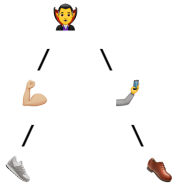
```

class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def mirror_tree(root):
    pass

```

"""



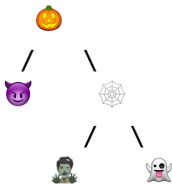
"""

```

# Using build_tree() function included at the top of the page
body_parts = ["Person", "Bicep", "Foot", "Shoe", None, None, "Boot"]
vampire = build_tree(body_parts)

```

"""



"""

```

spooky_objects = ["Jack-o'-lantern", "Cat Face", "Spider Web", None, None, "Ghost", "Ghost"]
spooky_tree = build_tree(spooky_objects)

# Using print_tree() function included at the top of the page
print_tree(mirror_tree(vampire))
print_tree(mirror_tree(spooky_tree))

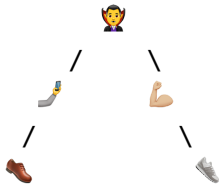
```

Example Output:

```
['👤', '👤', '👤', '👤', '👤', None, None, '👤']
```

Example 1 Explanation:

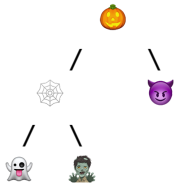
Mirrored Tree:



```
['🍂', '🍂', '🍂', '🍂', '🍂', '🍂', '🍂',]
```

Example 2 Explanation:

Mirrored Tree:



Problem 2: Pumpkin Patch Path

Leaning into the haunted hotel aesthetic, you've begun growing a pumpkin patch behind the hotel for the upcoming Halloween season. Given the `root` of a binary tree where each node represents a section of a pumpkin patch with a certain number of pumpkins, find the root-to-leaf path that yields the largest number of pumpkins. Return a list of the node values along the maximum pumpkin path.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def max_pumpkins_path(root):
    pass
```

Example Usage:


```

      7
     / \
    3   10
   / \  \
  1  5   15
=====
# Using build_tree() function includedd at the top of the page
pumpkin_quantities = [7, 3, 10, 1, None, 5, 15]
root1 = build_tree(pumpkin_quantities)

      12
     / \
    3   8
   / \  \
  4  50  10
=====

pumpkin_quantities = [12,3, 8, 4, 50, None, 10]
root2 = build_tree(pumpkin_quantities)

print(max_pumpkins_path(root1))
print(max_pumpkins_path(root2))

```

Example Output:

```

[7, 10, 15]
[12, 3, 50]

```

Problem 3: Largest Pumpkin in each Row

Given the root of a binary tree `pumpkin_patch` where each node represents a pumpkin in the patch and each node value represents the pumpkin's size, return an array of the largest pumpkin in each row of the pumpkin patch. Each level in the tree represents a row of pumpkins.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```

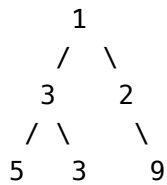
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def largest_pumpkins(pumpkin_patch):
    pass

```

Example Usage:

```
"""
```



```
"""
```

```
# Using build_tree() function included at the top of the page
pumpkin_sizes = [1, 3, 2, 5, 3, None, 9]
pumpkin_patch = build_tree(pumpkin_sizes)

print(largest_pumpkins(pumpkin_patch))
```

Example Output:

```
[1, 3, 9]
```

Problem 4: Counting Room Clusters

Given the root of a binary tree `hotel` where each node represents a room in the hotel and each node value represents the theme of the room, return the number of **distinct clusters** in the hotel. A distinct cluster is defined as a group of connected rooms (connected by edges) where each room has the same theme (`val`).

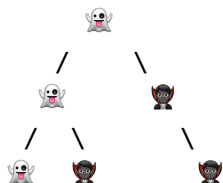
Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def count_clusters(hotel):
    pass
```

Example Usage:

```
"""
```



```
"""
```

```
# Using build_tree() function included at the top of the page
themes = ["👻", "👻", "🤖", "👻", "🤖", None, "🤖"]
hotel = build_tree(themes)

print(count_clusters(hotel))
```

Example Output:

3

Problem 5: Purging Unwanted Guests

There are unwanted visitors lurking in the rooms of your haunted hotel, and it's time for a clear out. Given the root of a binary tree `hotel` where each node represents a room in the hotel and each node value represents the guest staying in that room. You want to systematically remove visitors in the following order:

- Collect the guests (values) of all leaf nodes and store them in a list. The leaf nodes may be stored in any order.
- Remove all the leaf nodes.
- Repeat until the hotel (tree) is empty.

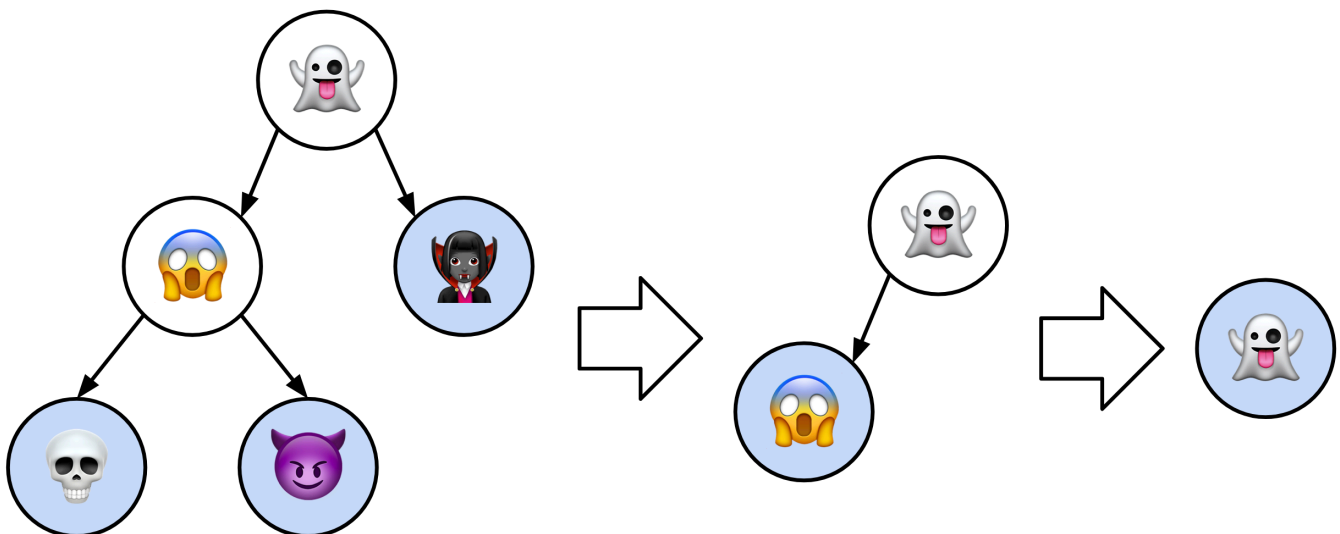
Return a list of lists, where each inner list represents a collection of leaf nodes.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

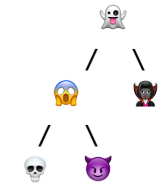
```
class TreeNode():
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def purge_hotel(hotel):
    pass
```

Example Usage:



|||||



|||||

```
# Using build_tree() function included at the top of the page
guests = ["👻", "👻", "👻", "💀", "🐱"]
hotel = build_tree(guests)

# Using print_tree() function included at the top of the page
print_tree(hotel)
print(purge_hotel(hotel))
```

Example Output:

```
Empty
[[['💀', '🐱', '👻'], ['👻'], ['👻']]
Explanation:
[[['💀', '👻', '🐱'], ['👻'], ['👻']] and [['👻', '🐱', '💀'], ['👻'], ['👻']] are all
answers since it doesn't matter which order the leaves in a given level are returned
The tree should always be empty once `purge_hotel()` has been executed.
```

Problem 6: Sectioning Off Cursed Zones

You've been hearing mysterious wailing and other haunting noises emanating from the deepest depths of the hotel. To keep guests safe, you want to section off the deepest parts of the hotel but keep as much of the hotel open as possible.

Given the root of a binary tree `hotel` where each node represents a room in the hotel, return the root of the smallest subtree in the hotel such that it contains all the deepest nodes of the original tree.

The depth of a room (node) is the shortest distance from it to the root. A room is called **the deepest** if it has the largest depth possible among any rooms in the entire hotel.

The subtree of a room is a tree consisting of that room, plus the set of all its descendants.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```

class TreeNode():
    def __init__(self, value, left=None, right=None):
        self.val = value
        self.left = left
        self.right = right

def smallest_subtree_with_deepest_rooms(hotel):
    pass

```

Example Usage:

```

"""
    Lobby
   /  \
  /    \
 101    102
 /  \  /  \
201 202 203 204
   /  \
 🤖 🧟
"""

# Using build_tree() included at top of page
rooms = ["Lobby", 101, 102, 201, 202, 203, 204, None, None, "🤖", "🧟"]
hotel1 = build_tree(rooms)

"""
    Lobby
   /    \
  101    102
   \
  🧟
"""

rooms = ["Lobby", 101, 102, None, "🧟"]
hotel2 = build_tree(rooms)

# Using print_tree() function included at top of page
print_tree(smallest_subtree_with_deepest_rooms(hotel1))
print_tree(smallest_subtree_with_deepest_rooms(hotel2))

```

Example Output:

```

[202, '🤖', '🧟']

```

Example 1 Explanation: Return node with value `202`. The emoji nodes `🤖` and `🧟` are nodes in the tree. Notice that subtrees with roots `101` and `Lobby` also contain the nodes, but the subtree with root `202` is the smallest.

```

['🧟']

```

Example 2 Explanation: The deepest node in the tree is the node with value `🧟`. The nodes with roots `🧟`, `101`, and `Lobby`, but `🧟` is the smallest.

- ▶ **Advanced Problem Set Version 1**
- ▶ **Advanced Problem Set Version 2**