TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

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**

Please Note: **It is not required or expected** that you complete all of the practice problems! In some sessions you may only complete 1 problem and that's okay.

Strengthening your **approach** to problems, and your **ability to speak and engage through the process** are key skills most often underdeveloped for engineers at this stage - focus on those in our small groups for your long term success!

You can always return to problems independently, after class time, to embrace the technical concepts and gain additional practice.

Close Section

🔎 Problem Solving Approach

We will approach problems using the six steps in the UMPIRE approach.

UMPIRE: Understand, Match, Plan, Implement, Review, Evaluate.

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

We will learn to:

- **Understand** the problem
- **Match** identifies common approaches you've seen/used before
- **Plan** a solution step-by-step, and
- **Implement** the solution
- **Review** your solution
- **Evaluate** your solution's time and space complexity and think critically about the advantages and disadvantages of your chosen approach.

ℹ️ 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'
```

ℹ️ 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:

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

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

"""
          (1, 'A')
          /       \
       (2, 'B')   (3, 'C')
      /           /      \
  (4, 'D')    (5, 'E')   (6, 'F')
"""

tree_with_keys_and_values = [(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), None, (5, 'E'), (6,
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(chocolatebox1))
print(sweet_difference(chocolatebox2))
```

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

# 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", "Smore", No
order1 = build_tree(flavors1)
order2 = build_tree(flavors2)

can_rearrange_orders(order1, order2)
```

Example Output:

```
True
Explanation:
              Red Velvet                              Red Velvet
             /          \                            /           \
        Vanilla         Lemon          ->        Lemon            Vanilla
        /      \        /   \                    /     \          /      \      ->
      Ube    Almond  Chai   Carrot           Chai   Carrot      Ube    Almond
                    /   \        \          /    \       \
                Chai   Maple   Smore     Chai   Maple   Smore


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

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:

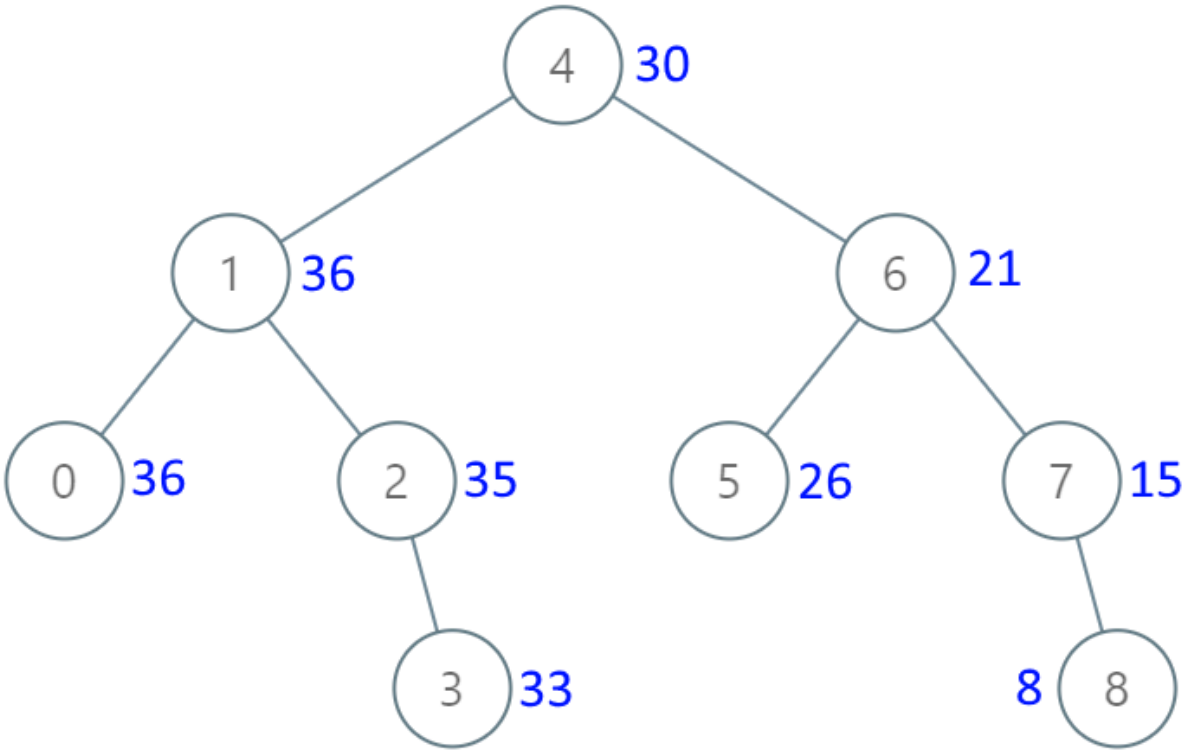
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

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:




```
"""
         4
       /   \
      /     \
     1       6
    / \     / \
   0   2   5   7
        \       \
         3       8
"""
# 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:
        30
       /   \
      /     \
     36     21
    / \     / \
   36  35  26  15
         \       \
         33       8
```

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 thngs.

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 = ["🧛", "💪", "🤳", "🪶", None, None, "👞"]
vampire = build_tree(body_parts)


"""
      🎃
     /   \
    😈
        /  \
       🧟   👻
"""
spooky_objects = ["🎃", "😈", " ", None, None, "🧟", "👻"]
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:

```
"""
    1
   /  \
  3    2
 / \    \
5   3    9
"""
# 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(themes))
```

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.

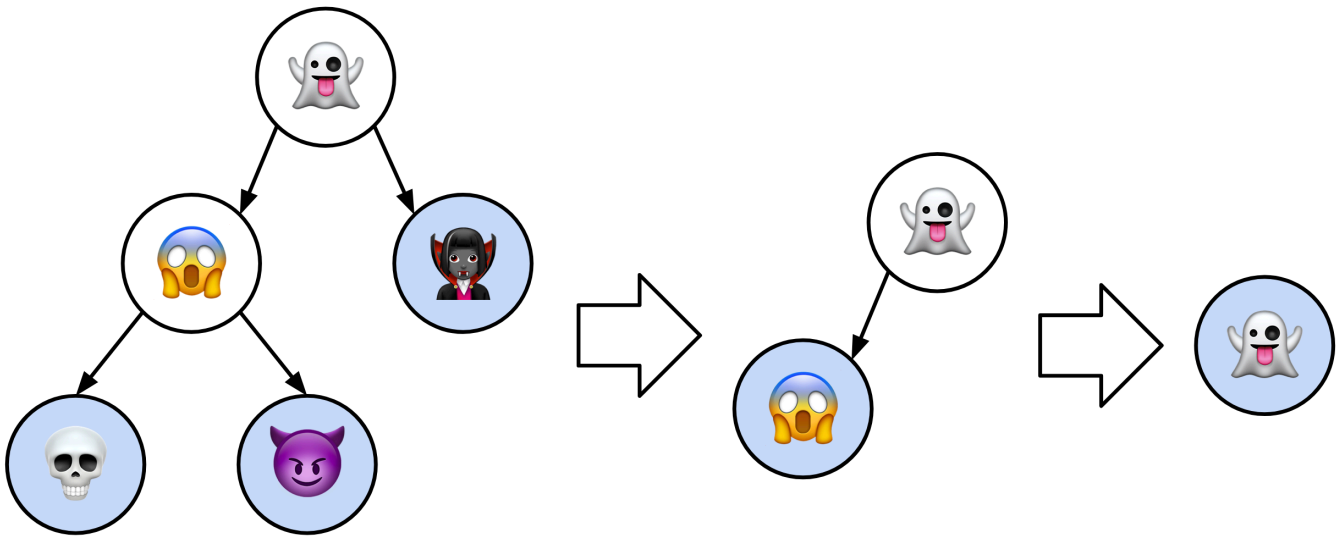
Return a list of lists, where each inner list represents a collection of leaf nodes.

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 also
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 deepesnt 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 the
nodes in the tree. Notice that subtrees with roots `101` and `Lobby` also contain the deepes
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 valid s
have roots `💀`, `101`, and `Lobby`, but `💀` is the smallest.
```

▾ Advanced Problem Set Version 1

Problem 1: Creating Cookie Orders from Descriptions

In your bakery, customer cookie orders are organized in a binary tree, where each node represents a different flavor of cookie ordered by the customers. You are given a 2D integer array `descriptions` where `descriptions[i] = [parent_i, child_i, is_left_i]` indicates that `parent_i` is the parent of `child_i` in a binary tree of unique flavors.

- If `is_left_i == 1`, then `child_i` is the left child of `parent_i`.
- If `is_left_i == 0`, then `child_i` is the right child of `parent_i`.

Construct the binary tree described by descriptions and return its root.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def build_cookie_tree(descriptions):
    pass
```

Example Usage:

```
descriptions1 = [
    ["Chocolate Chip", "Peanut Butter", 1],
    ["Chocolate Chip", "Oatmeal Raisin", 0],
    ["Peanut Butter", "Sugar", 1]
]

descriptions2 = [
    ["Ginger Snap", "Snickerdoodle", 0],
    ["Ginger Snap", "Shortbread", 1]
]

# Using print_tree() function included at top of page
print_tree(build_cookie_tree(descriptions1))
print_tree(build_cookie_tree(descriptions2))
```

Example Output:

```
['Chocolate Chip', 'Peanut Butter', 'Oatmeal Raisin', 'Sugar']
Example 1 Explanation:
The tree structure:
      Chocolate Chip
     /              \
Peanut Butter     Oatmeal Raisin
    /
 Sugar

['Ginger Snap', 'Shortbread', 'Snickerdoodle']
Example 2 Explanation:
The tree structure:
      Ginger Snap
     /           \
Shortbread   Snickerdoodle
```

Problem 2: Cookie Sum

Given the `root` of a binary tree where each node represents a certain number of cookies, return the number of unique paths from the `root` to a leaf node where the total number of cookies equals a given `target_sum`.

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. Evaluate the complexities for both a balanced and unbalanced tree.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def count_cookie_paths(root, target_sum):
    pass
```

```
"""
    10
   /  \
  5     8
 / \   / \
3   7 12  4
"""
# Using build_tree() function included at the top of the page
cookie_nums = [10, 5, 8, 3, 7, 12, 4]
cookies1 = build_tree(cookie_nums)

"""
    8
   / \
  4   12
 / \    \
2   6    10
"""
cookie_nums = [8, 4, 12, 2, 6, None, 10]
cookies2 = build_tree(cookie_nums)

print(count_cookie_paths(cookies1, 22))
print(count_cookie_paths(cookies2, 14))
```

Example Output:

```
2
1
```

Problem 3: Most Popular Cookie Combo

In your bakery, each cookie order is represented by a binary tree where each node contains the number of cookies of a particular type. The cookie combo for any node is defined as the total number of cookies in the entire subtree rooted at that node (including that node itself).

Given the `root` of a cookie order tree, return an array of the most frequent cookie combo in your bakery's orders. If there is a tie, return all the most frequent combos in any order.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def most_popular_cookie_combo(root):
    pass
```

Example Usage:

```
"""
       5
      / \
     2  -3
"""
cookies1 = TreeNode(5, TreeNode(2), TreeNode(-3))

"""
       5
      / \
     2  -5
"""
cookies2 = TreeNode(5, TreeNode(2), TreeNode(-5))

print(most_popular_cookie_combo(cookies1))
print(most_popular_cookie_combo(cookies2))
```

Example Output:

```
[2, 4, -3]
[2]
```

Problem 4: Convert Binary Tree of Bakery Orders to Linked List

You've been storing your bakery's orders in a binary tree where each node represents an order for a while now, but are wondering whether a new system would work better for you. You want to try storing orders in a linked list instead.

Given the root of a binary tree `orders`, flatten the tree into a 'linked list'.

- The 'linked list' should use the same `TreeNode` class where the `right` child points to the next node in the list and the `left` child pointer is always `None`.
- The 'linked list' should be in the same order as a preorder traversal of the binary tree.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def flatten_orders(orders):
    pass
```

Example Usage:

```
"""
        Croissant
       /         \
    Cupcake      Bagel
   /      \           \
Cake     Pie         Blondies
"""
# Using build_tree() function included at the top of page
items = ["Croissant", "Cupcake", "Bagel", "Cake", "Pie", None, "Blondies"]
orders = build_tree(items)

# Using print_tree() function included at the top of page
print_tree(flatten_orders(orders))
```

Example Output:

```
['Croissant', None, 'Cupcake', None, 'Cake', None, 'Pie', None, 'Bagel', None, 'Blondies']
Explanation:
'Linked List':
Croissant
    \
   Cupcake
       \
       Cake
         \
         Pie
           \
           Bagel
             \
            Blondies
```

Problem 5: Check Bakery Order Completeness

You have a customer order you are currently making stored in a binary tree where each node represents a different item in the order. Given the `root` of the order you are fulfilling, return `True` if the order is complete and `False` otherwise.

An order is complete if every level of the tree, except possibly the last, is completely filled with items (nodes), and all items in the last level are as far left as possible. It can have between `1` and `2^h` items inclusive at the last level `h` where levels are 0-indexed.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def is_complete(root):
    pass
```

Example Usage:

```
"""
        Croissant
       /         \
    Cupcake      Bagel
   /      \      /
Cake     Pie  Blondies
"""
# Using build_tree() function included at the top of page
items = ["Croissant", "Cupcake", "Bagel", "Cake", "Pie", "Blondies"]
order1 = build_tree(items)

"""
        Croissant
       /         \
    Cupcake      Bagel
   /      \           \
Cake     Pie         Blondies
"""
items = ["Croissant", "Cupcake", "Bagel", "Cake", "Pie", None, "Blondies"]
order2 = build_tree(items)

print(is_complete(order1))
print(is_complete(order2))
```

Example Output:

```
True
False
```

Problem 6: Vertical Bakery Display

Your bakery's inventory is organized in a binary tree where each node represents a different bakery item. To make it easier for staff to locate items, you want to create a vertical display of the inventory. The vertical order traversal should be organized column by column, from left to right.

If two items are in the same row and column, they should be listed from left to right, just as they appear in the inventory.

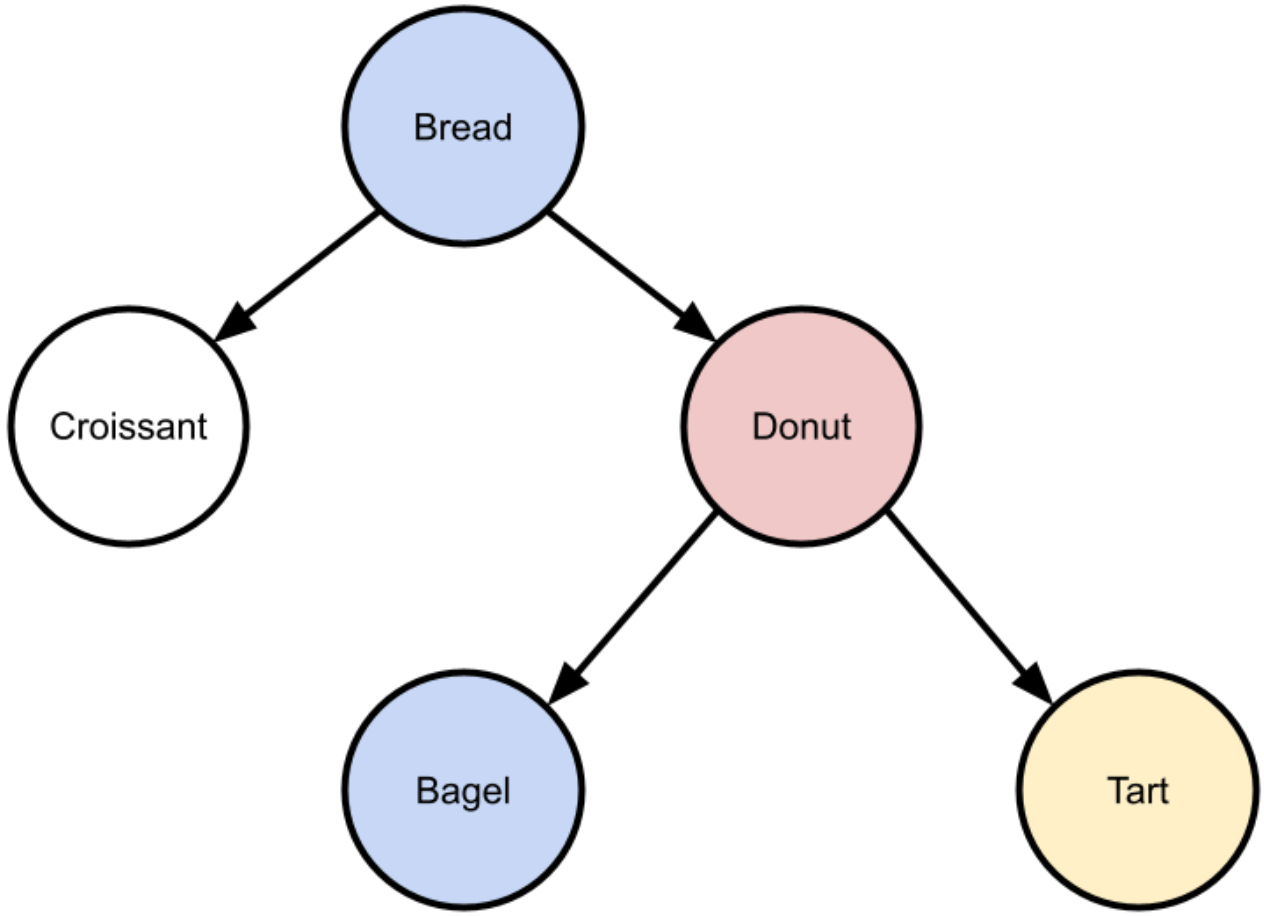
Given the `root` of the binary tree representing the inventory, return a list of lists with the vertical order traversal of the bakery items. Each inner list should represent the `ith` column in the inventory tree, and each inner list's elements should include the values of each bakery item in that column.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def vertical_inventoyr_display(root):
    pass
```

Example Usage 1:



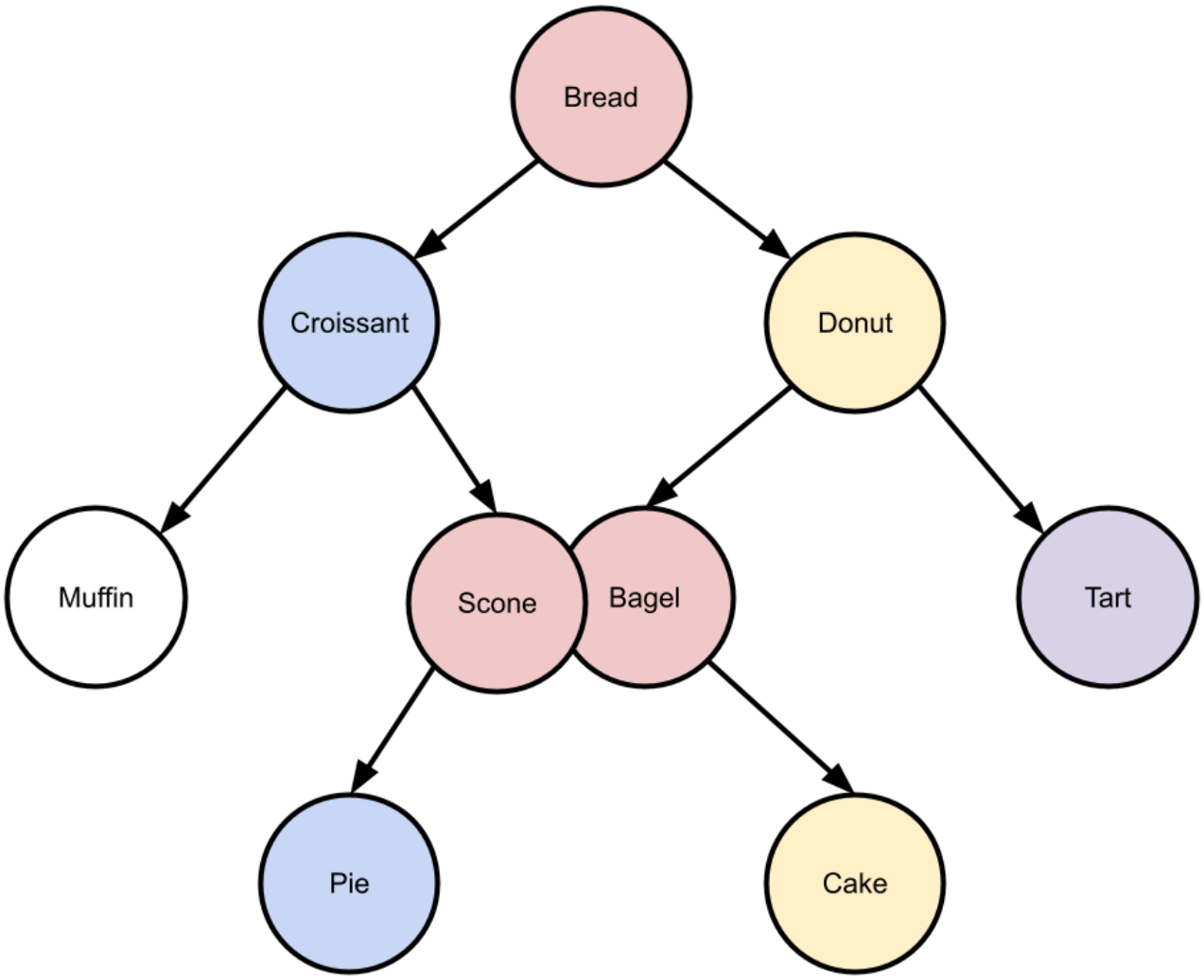

```
"""
         Bread
       /       \
   Croissant    Donut
                /   \
            Bagel Tart
"""
# Using build_tree() function included at the top of the page
inventory_items = ["Bread", "Croissant", "Donut", None, None, "Bagel", "Tart"]
inventory1 = build_tree(inventory_items)

print(vertical_inventory_display(inventory1))
```

Example Output 1:

```
[['Croissant'], ['Bread', 'Bagel'], ['Donut'], ['Tart']]
```

Example Usage 2:




```
"""
         Bread
       /       \
   Croissant    Donut
   /    \        /   \
Muffin  Scone Bagel Tart
        /       \
      Pie     Cake
"""
inventory_items = ["Bread", "Croissant", "Donut", "Muffin", "Scone", "Bagel", "Tart", None,
inventory2 = build_tree(inventory_items)

print(vertical_inventory_display(inventory2))
```

Example Output 2:

```
[['Muffin'], ['Croissant', 'Pie'], ['Bread', 'Scone', 'Bagel'], ['Donut', 'Cake'], ['Tart']]
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: 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 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. Evaluate the complexities for both a balanced and unbalanced tree.

```
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:

```
"""
    1
   /  \
  3    2
 / \    \
5   3    9
"""
# 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 2: 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 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. Evaluate the complexities for both a balanced and unbalanced tree.

```
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(themes))
```

Example Output:

```
3
```

Problem 3: Duplicate Sections of the Hotel

On one of your shifts at the haunted hotel, you find that you keep stumbling upon the same rooms in different halls. It's almost as if some parts of the hotel are being duplicated...

Given the root of a binary tree `hotel` where each node represents a room in the hotel, return a list of the roots of all duplicate subtrees. For each kind of duplicate subtree, you only need to return the root node of any **one** of them. Two trees are duplicate if they have the same structure and the same node values.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def find_duplicate_subtrees(hotel):
    pass
```

Example Usage:

```
"""
        Lobby
      /       \
    101      123
    /        /  \
  201      101  201
          /
         201
"""
# Using build_tree() function included at top of page
rooms = ["Lobby", 101, 123, 201, None, 101, 201, None, None, 201]
hotel = build_tree(rooms)

# Using print_tree() function included at top of page
subtree_lst = find_duplicate_subtrees(hotel)
for subtree in subtree_lst:
    print_tree(subtree)
```

Example Output:

```
[2, 4]
[4]
Explanation:
Subtrees:
   Subtree 1   Subtree 2
     101         201
     /
   201
```

Problem 4: Organizing Haunted Hallways

The haunted hotel is expanding, and the management wants to add new hallways filled with rooms that must be carefully arranged to maintain a spooky atmosphere. Given an integer array `rooms` sorted in ascending order where each element represents a unique room number, write a function that converts the array into a height-balanced binary search tree (BST) and returns the root of the balanced tree.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def array_to_bst(rooms):
    pass
```

Example Usage:

```
rooms = [4, 7, 13, 666, 1313]

# Using print_tree() function included at top of page
print_tree(array_to_bst(rooms))
```

Example Output:

```
[13, 7, 1313, 4, None, 666]
Explanation:
Balanced Tree:
       13
      /  \
    7    1313
  /       /
 4       666

[13, 4, 666, None, 7, 1313] is also an acceptable answer.

        13
      /  \
     4   666
      \     \
       7    1313
```

Problem 5: Count Cursed Hallways

The haunted hotel is known for its mysterious hallways, where guests often lose their way. Some hallways are said to be cursed, leading travelers to strange places when they follow a certain sequence of rooms. A hallway is said to be cursed if the sum of its room numbers adds up to a `target_sum`.

Given the root of a binary tree `hotel` where each node represents a room number in the hotel and an integer `target_sum` that represents the cursed sum, return the number of distinct paths in the hotel where the sum of the room numbers along the path equals `target_sum`

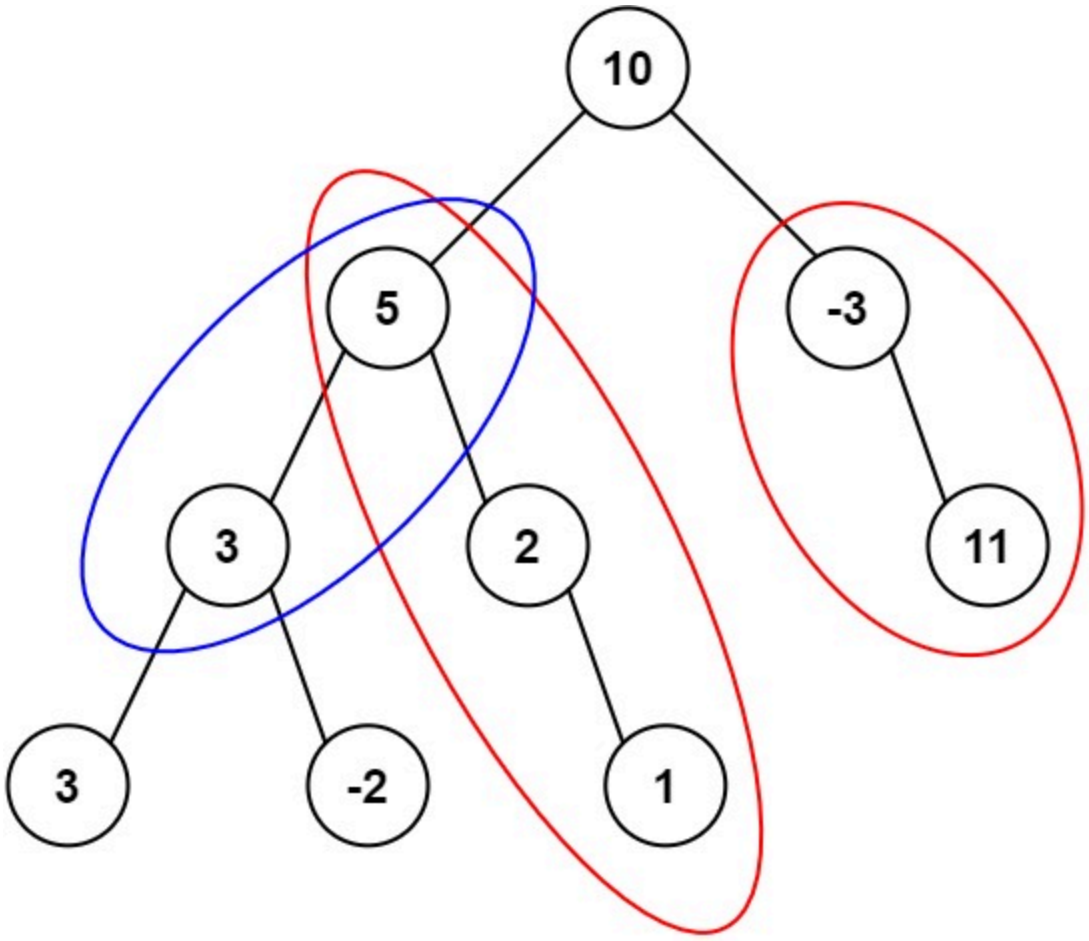
The path can start and end at any room but must follow the direction from parent rooms to child rooms. Your task is to count all such cursed paths that yield the exact `target_sum`.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def count_cursed_hallways(hotel, target_sum):
    pass
```

Example Usage:



```
"""
        10
       /  \
      /    \
     5      -3
    / \       \
   3   2      11
  / \   \
 3  -2   1
"""
# Using build_tree() function included at top of page
room_numbers = [10,5,-3,3,2,None,11,3,-2,None,1]
hotel = build_tree(room_numbers)

print(count_cursed_hallways(hotel, 8))
```

Example Output:

```
3
```

Problem 6: Step by Step Directions to Hotel Room

You have a lost guest who needs step by step directions to their hotel room. The hotel is stored in a binary tree where each node represents a room in the hotel. Each room in the hotel is uniquely assigned a value from `1` to `n`. You have the `root` of the hotel with `n` rooms, an integer `current_location` representing the value of the start node `s` and an integer `room_number` representing the value of the destination node `t`.

Find the shortest path starting from node `s` and ending at node `t`. Return step by step directions for the guest of this path as a string consisting of only uppercase letters `'L'`, `'R'`, and `'U'`. Each letter indicates a specific direction:

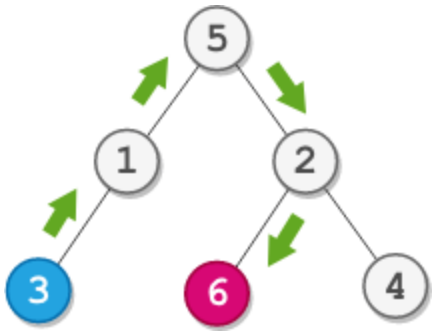
- `'L'` means to go from a node to its left child node.
- `'R'` means to go from a node to its right child node.
- `'U'` means to go from a node to its parent node.

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. Evaluate the complexities for both a balanced and unbalanced tree.

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

def count_cursed_hallways(hotel, current_location, room_number):
    pass
```

Example Usage 1:



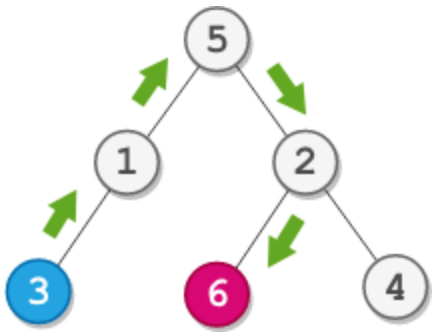

```
"""
       5
      / \
     1   2
    /   / \
   3   6   4
"""
# Using build_tree() function included at top of page
room_nums = [5,1,2,3,None,6,4]
hotel1 = build_tree(hotel1)

print(count_cursed_hallways(hotel1))
```

Example Output 2:

```
UURL
Explanation: The shortest path is: 3 -> 1 -> 5 -> 2 -> 6
```

Example Usage 2:



```
"""
  2
 /
1
"""
hotel2 = TreeNode(1, TreeNode(2))

print(count_cursed_hallways(hotel2))
```

Example Output 2:

```
L
Explanation: The shortest path is: 2 -> 1
```

Close Section