TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: **117667**

Session 1: Binary Trees

Session Overview

One paragraph explanation of what students will learn in this lesson.

> 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 wn 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

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`.

```python
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, 'I
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 1

▸ Standard Problem Set Version 1

▸ Standard Problem Set Version 2

▾ Advanced Problem Set Version 1

Problem 1: Croquembouche II

You are designing a delicious croquembouche (a French dessert composed of a cone-shaped tower of cream puffs 😋), for a couple's wedding. They want the cream puffs to have a variety of flavors. You've finished your design and want to send it to the couple for review.

Given a root of a binary tree `design` where each node in the tree represents a cream puff in the croquembouche, traverse the croquembouche in tier order (i.e., level by level, left to right).

You should return a list of lists where each inner list represents a tier (level) of the croquembouche and the elements of each inner list contain the flavors of each cream puff on that tier (node `val` s from left to right).

Note: The `build_tree()` and `print_tree()` functions both use variations of a level order traversal. To get the most out of this problem, we recommend that you reference these functions as little as possible while implementing your solution.

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.

Hint: Level order traversal, BST

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

def listify_design(design):
    pass
```

Example Usage:

```
"""
            Vanilla
           /       \
      Chocolate   Strawberry
      /     \
  Vanilla   Matcha
"""
croquembouche = Puff("Vanilla",
                    Puff("Chocolate", Puff("Vanilla"), Puff("Matcha")),
                    Puff("Strawberry"))
print(listify_design(croquembouche))
```

Example Output:

```
[['Vanilla'], ['Chocolate', 'Strawberry'], ['Vanilla', 'Matcha']]
```

▶ 💡 **Hint: Breadth First Search Traversal**

Problem 2: Icing Cupcakes in Zigzag Order

You have rows of cupcakes represented as a binary tree `cupcakes` where each node in the tree represents a cupcake. To ice them efficiently, you are icing cupcakes one row (level) at a time, in zig zag order (i.e., from left to right, then right to left for the next row and alternate between).

Return a list of the cupcake values in the order you iced them.

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, flavor, left=None, right=None):
        self.val = flavor
        self.left = left
        self.right = right

def zigzag_icing_order(cupcakes):
    pass
```

Example Usage:

```
"""
            Chocolate
           /         \
        Vanilla       Lemon
       /              /    \
    Strawberry   Hazelnut   Red Velvet
"""

# Using build_tree() function included at top of page
flavors = ["Chocolate", "Vanilla", "Lemon", "Strawberry", None, "Hazelnut", "Red Vel
cupcakes = build_tree(flavors)
```

Example Output:

```
['Chocolate', 'Lemon', 'Vanilla', 'Strawberry', 'Hazelnut', 'Red Velvet']
```

▶ 💡 **Hint: Choosing your Traversal Method**

Problem 3: 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:

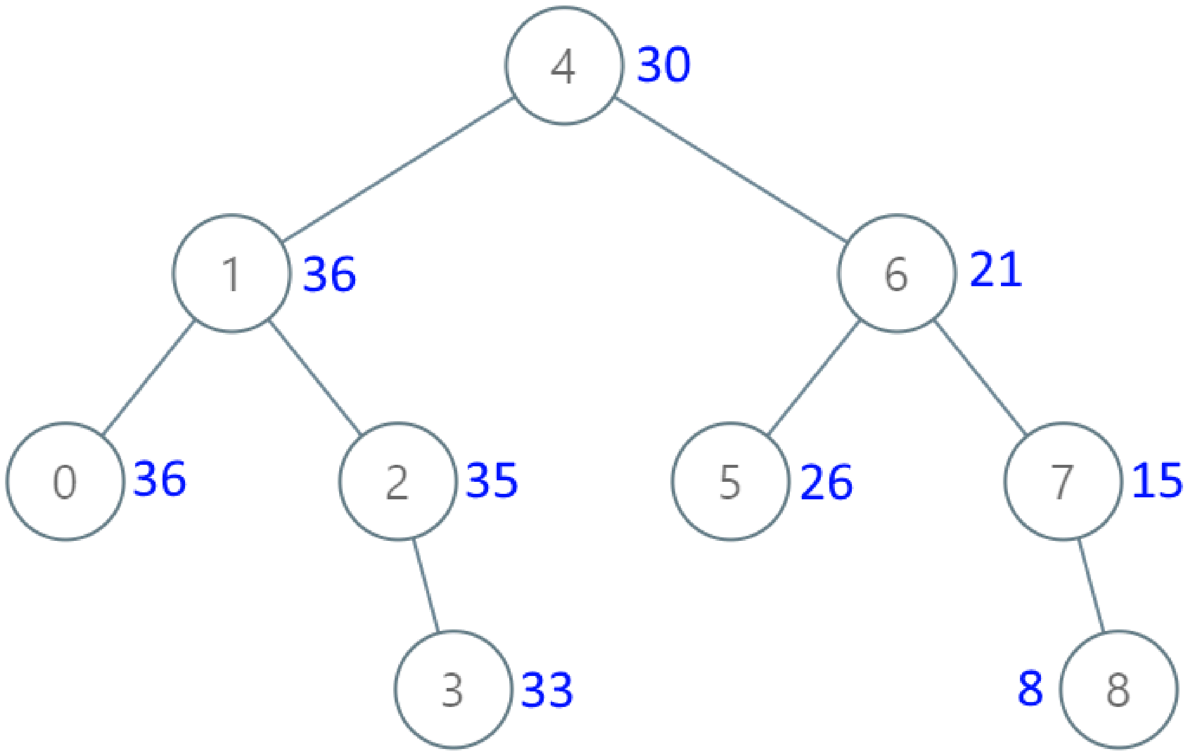
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

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
```

▶ 💡 **Hint: Choosing your Traversal Method**

Problem 4: 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 larger_order_tree(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 = larger_order_tree(cupcakes, cake)
next_order2 = larger_order_tree(cupcakes, cookies)
print(next_order1.val)
print(next_order2.val)
```

Example Output:

```
Eclair
None
```

Problem 5: Add Row of Cupcakes to Display

You have a cupcake display represented by a binary tree where each node represents a different cupcake in the display and each node value represents the flavor of the cupcake. Given the root of the binary tree `display` a string `flavor` and an integer `depth`, add a row of nodes with value `flavor` at the given depth `depth`.

Note that the root node has depth `1`.

The adding rule is:

- Given the integer `depth`, for each not `None` tree node `cur` at the depth `depth - 1`, create two cupcakes with value `flavor` as `cur`'s left subtree root and right subtree root.
- `cur`'s original left subtree should be the left subtree of the new left subtree root.
- `cur`'s original right subtree should be the right subtree of the new right subtree root.
- If `depth == 1` that means there is no depth `depth - 1` at all, then create a cupcake with value `flavor` as the new root of the whole original tree, and the original tree is the new root's left subtree.

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.

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

def add_row(display, flavor, depth):
    pass
```

```python
"""
        Chocolate
        /        \
    Vanilla    Strawberry
                /     \
         Chocolate    Red Velvet
"""
# Using build_tree() function included at top of page
cupcake_flavors = ["Chocolate", "Vanilla", "Strawberry", None, None, "Chocolate", "R
display = build_tree(cupcake_flavors)

# Using print_tree() function included at top of page
print_tree(add_row(display, "Mocha", 3))
```

Example Output:

```
['Chocolate', 'Vanilla', 'Strawberry', 'Mocha', 'Mocha', 'Mocha', 'Mocha', None, Non
Explanation:
Tree with added row:
                   Chocolate
                   /        \
             Vanilla        Strawberry
             /    \          /       \
          Mocha   Mocha  Mocha      Mocha
                          /             \
                      Chocolate       Red Velvet
```

Problem 6: Maximum Icing Difference

In your bakery, you're planning a display of cupcakes where each cupcake is represented by a node in a binary tree. The sweetness level of the icing on each cupcake is stored in the node's value. You want to identify the maximum icing difference between any two cupcakes where one cupcake is an ancestor of the other in the display.

Given the `root` of a binary tree representing the cupcake display, find the maximum value `v` for which there exist different cupcakes `a` and `b` where `v = |a.val - b.val|` and `a` is an ancestor of `b`.

A cupcake `a` is an ancestor of `b` if either any child of `a` is equal to `b`, or any child of `a` is an ancestor of `b`.

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, sweetness, left=None, right=None):
        self.val = sweetness
        self.left = left
        self.right = right

def max_icing_difference(root):
    pass
```

Example Usage:

```
"""
            8
           /  \
         3     10
        / \      \
       1   6      14
          /  \    /
         4    7  13
"""

# Using build_tree() function included at top of page
sweetness_levels = [8, 3, 10, 1, 6, None, 14, None, None, 4, 7, 13]
display = build_tree(sweetness_levels)

print(max_icing_difference(display))
```

Example Output:

```
13
Explanation: The maximum icing difference is between the root cupcake (8) and a desc
sweetness level 1, yielding a difference of |8 - 1| = 7.
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Mapping a Haunted Hotel II

You have been working the night shift at a haunted hotel and guests have been coming to check out of rooms that you're pretty sure don't exist in the hotel... or are you imagining things? To make sure, you want to explore the entire hotel and make your own map.

Given the root of a binary tree `hotel` where each node represents a room in the hotel, write a function `map_hotel()` that returns a dictionary mapping each level of the hotel to a list with the level's room values in the order they appear on that level from left to right.

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.

Note: The `build_tree()` and `print_tree()` functions both use variations of a level order traversal. To get the most out of this problem, we recommend that you reference these functions as little as possible while implementing your solution.

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

def map_hotel(hotel):
    pass
```

Example Usage:

```
"""
         Lobby
        /     \
       /       \
      101      102
     /   \    /   \
   201  202  203  204
   /                \
 301                302
"""

hotel = Room("Lobby",
                Room(101, Room(201, Room(301)), Room(202)),
                Room(102, Room(203), Room(204, None, Room(302))))

print(map_hotel(hotel))
```

Example Output:

```
{
    0: ['Lobby'],
    1: [101, 102],
    2: [201, 202, 203, 204],
    3: [301, 302]
}
```

▶ 💡 **Hint: Breadth First Search Traversal**

Problem 2: Reverse Odd Levels of the Hotel

A poltergeist has been causing mischief and reversed the order of rooms on odd level floors. Given the root of a binary tree `hotel` where each node represents a room in the hotel and the root, restore order by reversing the node values at each odd level in the tree.

For example, suppose the rooms on level 3 have values `[308, 307, 306, 305, 304, 303, 302, 301]`. It should become `[301, 302, 303, 304, 305, 306, 307, 308]`.

Return the root of the altered tree.

A binary tree is perfect if all parent nodes have two children and all leaves are on the same level.

The level of a node is the number of edges along the path between it and the root node.

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. Assume the input tree is balanced when calculating time complexity.

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

def reverse_odd_levels(hotel):
    pass
```

Example Usage:

```
"""
        Lobby
      /      \
     102     101
     / \     /  \
   201 202 203 204
"""
hotel = Room("Lobby",
            Room(102, Room(201), Room (202)),
                Room(101, Room(203), Room(204)))

# Using print_tree() function included at the top
print_tree(reverse_odd_levels(hotel))
```

Example Output:

```
['Lobby', 101, 102, 201, 202, 203, 204]

Explanation:
Updated Tree Structure:
        Lobby
      /      \
     101     102
     / \     /  \
   201 202 203 204
```

▶ 💡 **Hint: Choosing your Traversal Method**

Problem 3: Purging Unwanted Guests

There are unwanted visitors lurking in the rooms of your haunteds 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:

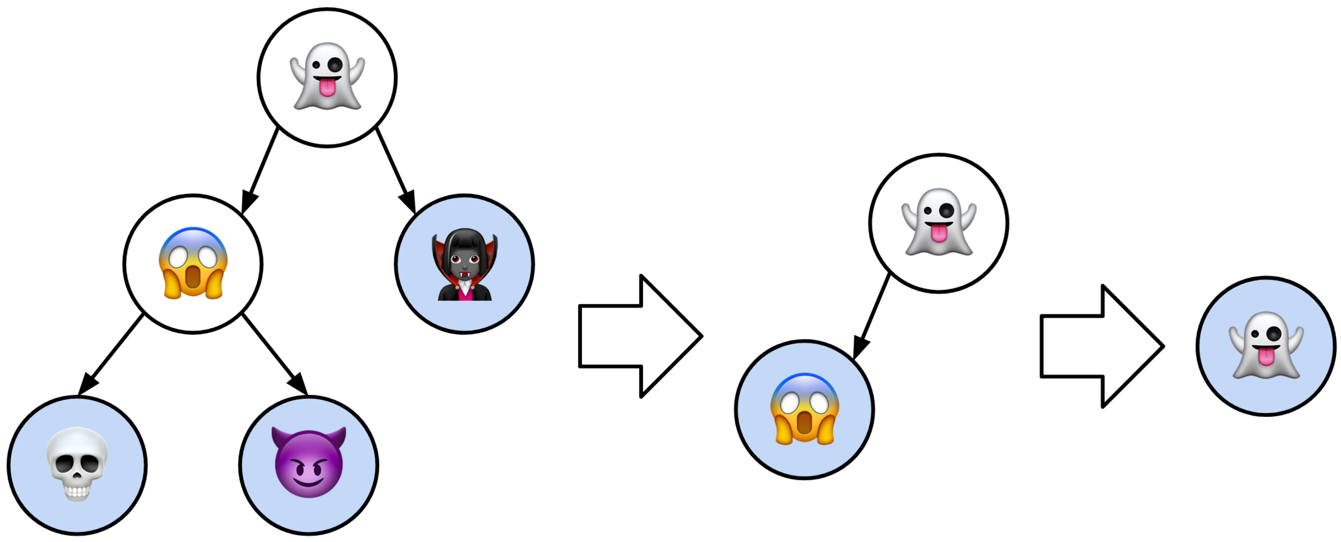
- Collect the guests (values) of all leaf nodes and store them in a list. The leaf nodes may be stored in any order.
- Remove all the leaf nodes.
- Repeat until the hotel (tree) is empty.

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

```python
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:



```python
"""
       👻
      /  \
    😱    🧛🏿‍♀️
   /  \
  💀   😈
"""

# 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 al
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.
```

▶ 💡 **Hint: Choosing your Traversal Method**

Problem 4: Kth Spookiest Room in the Hotel

Over time, your hotel has gained a reputation for being haunted, and you now have customers coming specifically for a spooky experience. You are given the `root` of a binary search tree (BST) with `n` nodes where each node represents a room in the hotel and each node has an integer `key` representing the spookiness of the room (`1` being most spooky and `n` being least spooky) and `val` representing the room number. The tree is organized according to its keys.

Given the `root` of a BST and an integer `k` write a function `kth_spookiest()` that returns the **value** of the `kth` spookiest room (smallest `key`, 1-indexed) of all the rooms in the hotel.

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 and space complexity.

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

def kth_spookiest(root, k):
    pass
```

Example Usage:

```
"""
       (3, Lobby)
     /          \
(1, 101)    (4, 102)
      \
       (2, 201)
"""

# Using build_tree() function at the top of the page
rooms = [(3, "Lobby"), (1, 101), (4, 102), None, (2, 201)]
hotel1 = build_tree(rooms)


"""
            (5, Lobby)
            /         \
        (3, 101)    (6, 102)
        /      \
    (2, 201)  (4, 202)
    /
(1, 301)
"""
rooms = [(5, 'Lobby'), (3, 101), (6, 102), (2, 201), (4, 202), None, None, (1, 301)]
hotel2 = build_tree(rooms)

print(kth_spookiest(hotel1, 1))
print(kth_spookiest(hotel2, 3))
```

Example Markdown:

```
101
101
```

Problem 5: Lowest Common Ancestor of Youngest Children

There's a tapestry hanging up on the wall with the family tree of the cursed family who owns the hotel. Given the `root` of the binary tree where each node represents a member in the family, return the value of the lowest common ancestor of the youngest children in the family. The youngest children in the family are the deepest leaves in the tree.

Recall that:

- The node of a binary tree is a leaf if and only if it has no children
- The depth of the root of the tree is `0`. If the depth of a node is `d`, the depth of each of its children is `d + 1`.
- The lowest common ancestor of a set `S` of nodes, is the node `A` with the largest depth such that every node in `S` is in the subtree with root `A`.

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

def lca_youngest_children(root):
    pass
```

Example Usage:

```
"""
                Isadora the Hexed
                /                \
            Thorne               Raven
           /      \             /      \
      Dracula     Doom      Hecate    Wraith
                 /    \
             Gloom   Mortis
"""
# Using build_tree() function included at top of the page
members = ["Isadora the Hexed", "Thorne", "Raven", "Dracula", "Doom", "Hecate", "Wra
family1 = build_tree(members)

"""
              Grandmama Addams
              /              \
        Gomez Addams        Uncle Fester
                \
            Wednesday Addams
"""
members = ["Grandmama Addams", "Gomez Addams", "Uncle Fester", None, "Wednesday Adda
family2 = build_tree(members)

print(lca_youngest_children(family1))
print(lca_youngest_children(family2))
```

Example Output:

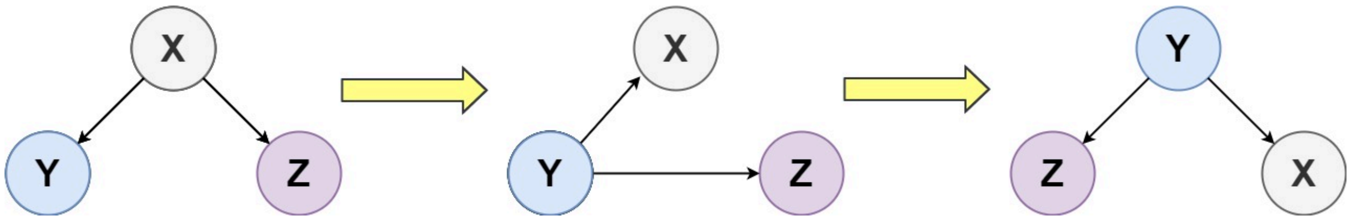
```
Doom
Example 1 Explanation: Gloom and Mortis are the youngest children (deepest leaves) i
Doom in their lowest common ancestor.

Wednesday Addams
Example 2 Explanation: The youngest child in the tree is Wednesday Addams and the lo
of one node is itself
```

Problem 6: Topsy Turvy

You're walking down the hotel hallway one night and something strange begins to happen - the entire hotel flips upside down. The room sand their connections were flipped in a peculiar way and now you need to restore order. Given the root of a binary tree `hotel` where each node represents a room in the hotel, write a function `upside_down_hotel()` that flips the hotel right side up according to the following rules:

1. The original left child becomes the new root
2. The original root becomes the new right child
3. The original right child becomes the new left child.



The above steps are done level by level. It is **guaranteed** each right node has a sibling (a left node with the same parent) and has no children.

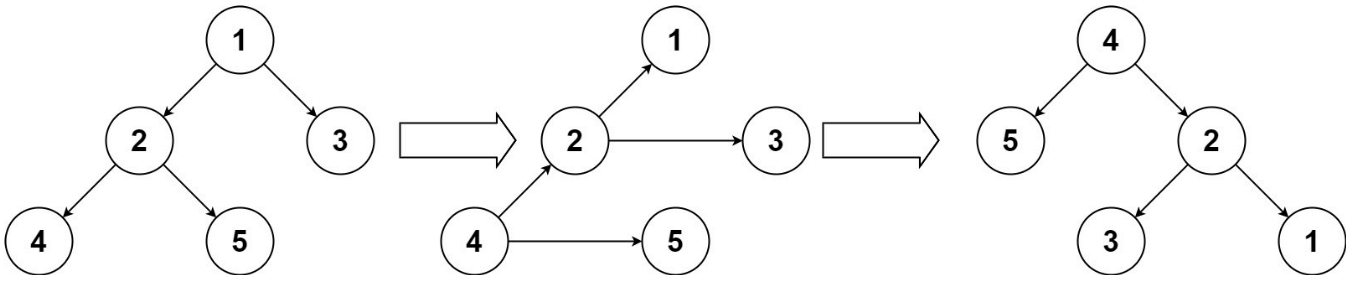
Return the root of the flipped hotel.

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 and space complexity.

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

def flip_hotel(hotel):
    pass
```

Example Usage:



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

# Using build_tree() function included at top of page
rooms = [1, 2, 3, 4, 5]
hotel = build_tree(rooms)

# Using print_tree() function included at top of page
print_tree(flip_hotel(hotel))
```

Example Output:

```
[4, 5, 2, None, None, 3, 1]
Explanation:
Flipped hotel structure:
      4
    /   \
   5     2
        / \
       3   1
```

Close Section