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'
```

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, 'E
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

Problem 1: Merging Cookie Orders

You run a local bakery and are given the roots of two binary trees `order1` and `order2` where each node in the binary tree represents the number of a certain cookie type the customer has ordered. To maximize efficiency, you want to bake enough of each type of cookie for both orders together.

Given `order1` and `order2`, merge the order together into one tree and return the root of the merged tree. To merge the orders, imagine that when place one tree on top of the other, some nodes of the two trees are overlapped while others are not. If two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the **not** `None` node will be used as the node of the new tree.

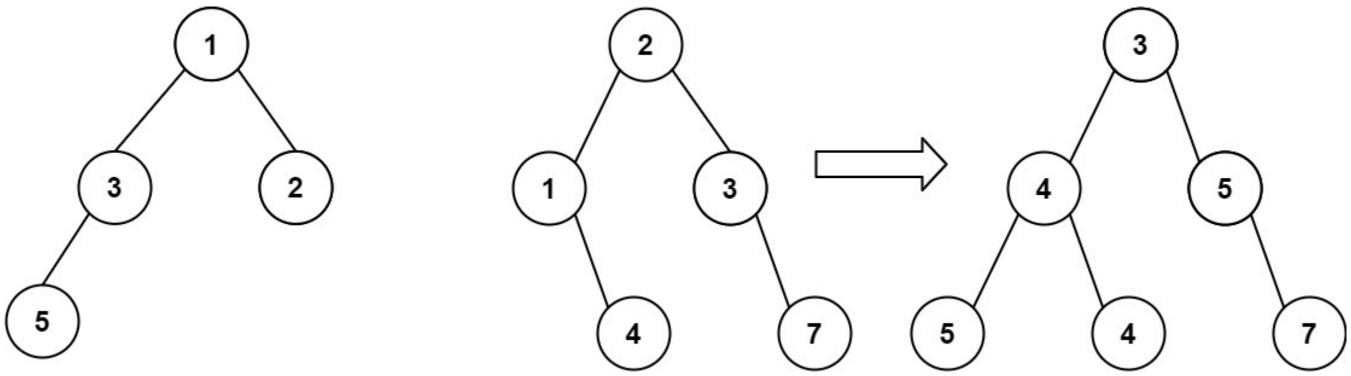
Start the merging process from the root of both orders.

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

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

Example Usage:



```
"""
     1             2
    / \          /   \
   3    2       1      3
 /               \      \
5                 4      7
"""

# Using build_tree() function included at top of page
cookies1 = [1, 3, 2, 5]
cookies2 = [2, 1, 3, None, 4, None, 7]
order1 = build_tree(cookies1)
order2 = build_tree(cookies2)

# Using print_tree() function included at top of page
print_tree(merge_orders(order1, order2))
```

Example Usage:

```
[3, 4, 5, 5, 4, None, 7]
Explanation:
Merged Tree:
     3
    /  \
  4     5
 / \      \
5   4      7
```

Problem 2: Croquembouche

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, that **prints** a list of the flavors (`val`s) of each cream puff in level order (i.e., from left to right, level by level).

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 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.

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

def print_design(design):
    pass
```

Example Usage:

```python
"""
            Vanilla
           /       \
      Chocolate   Strawberry
      /     \
  Vanilla   Matcha
"""
croquembouche = Puff("Vanilla",
                    Puff("Chocolate", Puff("Vanilla"), Puff("Matcha")),
                    Puff("Strawberry"))
print_design(croquembouche)
```

Example Output:

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

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

Problem 3: Maximum Tiers in Cake

You have entered your bakery into a cake baking competition and for your entry have decided build a complicated pyramid shape cake, where different sections have different numbers of tiers. Given the root of a binary tree `cake` where each node represents a different section of your cake, return the maximum number of tiers in your cake.

The maximum number of tiers is the number of nodes along the longest path from the root node down to the farthest leaf 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 max_tiers(cake):
    pass
```

Example Usage:

```
"""
        Chocolate
        /        \
    Vanilla    Strawberry
                /      \
         Chocolate    Coffee
"""
# Using build_tree() function included at top of page
cake_sections = ["Chocolate", "Vanilla", "Strawberry", None, None, "Chocolate", "Cof
cake = build_tree(cake_sections)

print(max_tiers(cake))
```

Example Output:

```
3
```

Problem 4: Maximum Tiers in Cake II

If you solved `max_tiers()` in the previous problem using a depth first search approach, reimplement your solution using a breadth first search approach. If you implemented it using a breadth first search approach, use a depth first search approach.

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 max_tiers(cake):
    pass
```

Example Usage:

```
"""
        Chocolate
        /        \
    Vanilla    Strawberry
                /     \
         Chocolate    Coffee
"""
# Using build_tree() function included at top of page
cake_sections = ["Chocolate", "Vanilla", "Strawberry", None, None, "Chocolate", "Cof
cake = build_tree(cake_sections)

print(max_tiers(cake))
```

Example Output:

```
3
```

Problem 5: Can Fulfill Order

At your bakery, you organize your current stock of baked goods in a binary tree with root `inventory` where each node represents the quantity of a baked good in your bakery. A customer comes in wanting a random assortment of baked goods of quantity `order_size`. Given the root `inventory` and integer `order_size`, return `True` if you can fulfill the order and `False` otherwise. You can fulfill the order if the tree has a root-to-leaf path such that adding up all the values along the path equals `order_size`.

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 can_fulfill_order(inventory, order_size):
    pass
```

Example Usage:

```
"""
             5
           /   \
          4     8
        /      /  \
       11     13   4
      /  \          \
     7    2          1
"""

# Using build_tree() function included at top of the page
quantities = [5,4,8,11,None,13,4,7,2,None,None,None,1]
baked_goods = build_tree(quantities)

print(can_fulfill_order(baked_goods, 22))
print(can_fulfill_order(baked_goods, 2))
```

Example Output:

```
True
Example 1 Explanation: 5 + 4 + 11 + 2 = 22

False
```

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

Problem 6: 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 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, 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)
print(zigzag_icing_order(cupcakes))
```

Example Output:

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

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

▶ 💡 **Hint: Available `deque()` methods**

Close Section

▼ Standard Problem Set Version 2

Problem 1: Clone Detection

You have just started a new job working the night shift at a local hotel, but strange things have been happening and you're starting to think it might be haunted. Lately, you think you've been seeing double of some of the guests.

Given the roots of two binary trees `guest1` and `guest2` each representing a guest at the hotel, write a function that returns `True` if they are clones of each other and `False` otherwise.

Two binary trees are considered clones if they are structurally identical, and the nodes have the same values.

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.

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

def is_clone(guest1, guest2):
    pass
```

Example Usage:

```python
"""
     John Doe               John Doe
     /      \              /       \
  6 ft    Brown Eyes      6ft      Brown Eyes
"""
guest1 = TreeNode("John Doe", TreeNode("6 ft"), TreeNode("Brown Eyes"))
guest2 = TreeNode("John Doe", TreeNode("6 ft"), TreeNode("Brown Eyes"))

"""
     John Doe         John Doe
     /                      \
   6 ft                     6 ft
"""
guest3 = TreeNode("John Doe", TreeNode("6 ft"))
guest4 = TreeNode("John Doe", None, TreeNode("6 ft"))

print(is_clone(guest1, guest2))
print(is_clone(guest3, guest4))
```

Example Output:

```
True
False
```

Problem 2: Mapping a Haunted Hotel

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 list of each room value in the hotel. You should explore the hotel level by level from left to right.

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.

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:

```
['Lobby', 101, 102, 201, 202, 203, 204, 301, 302]
```

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

Problem 3: Minimum Depth of Secret Path

You've found a strange door in the hotel and aren't sure where it leads. Given the root of a binary tree `door` where each node represents a destination along a path behind the door, return the minimum depth of the tree.

The minimum depth is the number of nodes along the shortest path from from the root node down to the nearest leaf 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 min_depth(door):
    pass
```

Example Usage:

```
"""
     Door
    /    \
 Attic    Cursed Room
         /       \
      Crypt     Haunted Cellar
"""

door = Room("Door", Room("Attic"), Room("Cursed Room", Room("Crypt"), Room("Haunted 

print(min_depth(attic))
```

Example Output:

```
2
```

Problem 4: Minimum Depth of Secret Path II

If you used a breadth first search approach to solve the previous problem, reimplement your solution using a depth first search approach. If you used a depth first search approach, try using a breadth first search approach.

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 min_depth(door):
    pass
```

Example Usage:

```
"""
     Door
    /    \
 Attic    Cursed Room
         /       \
      Crypt     Haunted Cellar
"""

door = Room("Door", Room("Attic"), Room("Cursed Room", Room("Crypt"), Room("Haunted 

print(min_depth(attic))
```

Example Output:

```
2
```

Problem 5: 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 6: 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 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, 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
```

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

Close Section

▶ Advanced Problem Set Version 1

▶ Advanced Problem Set Version 2