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

Students are introduced to foundational and complex tasks involving binary trees. They will engage in constructing trees, manipulating tree structures, traversing trees, and understanding tree properties through a variety of exercises. This session aims to deepen students' understanding of tree algorithms, enhancing their ability to analyze and implement data structures efficiently.

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

📊 Part 1 : Instructor Led Session

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

👨‍💻 Part 2: Breakout Session

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

In breakout rooms you will:

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

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

▶ **Note on Expectations**

🔎 Problem Solving Approach

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

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

We will learn to:

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

▶ Comment on UPI

▶ UPI Example

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'
```

Breakout Problems Session 1

▾ Standard Problem Set Version 1

Problem 1: Grafting Apples

You are grafting different varieties of apple onto the same root tree can produce many different varieties of apples! Given the following `TreeNode` class, create the binary tree depicted below. The text representing each node should should be used as the `value`.

The `root`, or topmost node in the tree `TreeNode("Trunk")` has been provided for you.

```
            Trunk
          /       \
      Mcintosh   Granny Smith
      /     \       /     \
    Fuji   Opal   Crab   Gala
```

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

root = TreeNode("Trunk")
```

Example Usage:

```
# Using print_tree() included at the top of this page
print_tree(root)
```

Example Output:

```
['Trunk', 'Mcintosh', 'Granny Smith', 'Fuji', 'Opal', 'Crab', 'Gala']
```

▶ 💡 **Hint: Binary Trees**

Problem 2: Calculating Yield

You have a fruit tree represented as a binary tree with exactly three nodes: the `root` and its two children. Given the `root` of the tree, evaluate the amount of fruit your tree will yield this year. The tree has the following form:

- **Leaf nodes** have an integer value.
- The **root** has a string value of either `"+"`, `"-"`, `"*"`, or `"-"`.

The **yield** of a the tree is calculated by applying the mathematical operation to the two children.

Return the result of evaluating 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.

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

def calculate_yield(root):
  pass
```

Example Usage:

```
"""
    +
  /   \
 7     5
"""
apple_tree = TreeNode("+", TreeNode(7), TreeNode(5))

print(calculate_yield(apple_tree))
```

Example Output:

```
12
```

Problem 3: Ivy Cutting

You have a trailing ivy plant represented by a binary tree. You want to take a cutting to start a new plant using the rightmost vine in the plant. Given the `root` of the plant, return a list with the value of each node in the path from the `root` node to the rightmost 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 right_vine(root):
  pass
```

Example Usage:

```
"""
        Root
      /      \
    Node1    Node2
  /         /    \
Leaf1    Leaf2  Leaf3
"""
ivy1 = TreeNode("Root",
                TreeNode("Node1", TreeNode("Leaf1")),
                TreeNode("Node2", TreeNode("Leaf2"), TreeNode("Leaf3")))

"""
      Root
      /
    Node1
    /
  Leaf1
"""
ivy2 = TreeNode("Root", TreeNode("Node1", TreeNode("Leaf1")))

print(right_vine(ivy1))
print(right_vine(ivy2))
```

Example Output:

```
['Root', 'Node2', 'Leaf3']
['Root']
```

▶ 💡 **Hint: Balanced Trees**

Problem 4: Ivy Cutting II

If you implemented `right_vine()` iteratively in the previous problem, implement it recursively. If you implemented it recursively, implement it iteratively.

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 right_vine(root):
  pass
```

Example Usage:

```
"""
        Root
      /      \
    Node1    Node2
  /         /    \
Leaf1    Leaf2  Leaf3
"""
ivy1 = TreeNode("Root",
                TreeNode("Node1", TreeNode("Leaf1")),
                TreeNode("Node2", TreeNode("Leaf2"), TreeNode("Leaf3")))

"""
      Root
      /
    Node1
    /
  Leaf1
"""
ivy2 = TreeNode("Root", TreeNode("Node1", TreeNode("Leaf1")))

print(right_vine(ivy1))
print(right_vine(ivy2))
```

Example Output:

```
['Root', 'Node2', 'Leaf3']
['Root']
```

Problem 5: Count the Tree Leaves

You've grown an oak tree from a tiny little acorn and it's finally sprouting leaves! Given the `root` of a binary tree representing your oak tree, count the number of leaf nodes in the tree. A leaf node is a node that does not have any children.

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 count_leaves(root):
    pass
```

Example Usage:

```
"""
        Root
      /      \
    Node1    Node2
  /         /    \
Leaf1    Leaf2  Leaf3
"""

oak1 = TreeNode("Root",
                TreeNode("Node1", TreeNode("Leaf1")),
                TreeNode("Node2", TreeNode("Leaf2"), TreeNode("Leaf3")))

"""
      Root
      /
    Node1
    /
  Leaf1
"""
oak2 = TreeNode("Root", TreeNode("Node1", TreeNode("Leaf1")))


print(count_leaves(oak1))
print(count_leaves(oak2))
```

Example Output:

```
3
1
```

▶ 💡 **Hint: Traversing Trees**

Problem 6: Pruning Plans

You have a large overgrown Magnolia tree that's in desperate need of some pruning. Before you can prune the tree, you need to do a full survey of the tree to evaluate which sections need to be pruned.

Given the `root` of a binary tree representing the magnolia, return a list of the values of each node using a postorder traversal. In a postorder traversal, you explore the left subtree first, then the right subtree, and finally the root. Postorder traversals are often used when deleting nodes from a tree.

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 survey_tree(root):
    pass
```

Example Usage:

```
"""
        Root
      /      \
    Node1    Node2
  /         /    \
Leaf1    Leaf2  Leaf3
"""

magnolia = TreeNode("Root",
                TreeNode("Node1", TreeNode("Leaf1")),
                TreeNode("Node2", TreeNode("Leaf2"), TreeNode("Leaf3")))

print(survey_tree(magnolia))
```

Example Output:

```
["Leaf1", "Node1", "Leaf2", "Leaf3", "Node2", "Root"]
```

Problem 7: Foraging Berries

You've found a wild blueberry bush and want to do some foraging. However, you want to be conscious of the local ecosystem and leave some behind for local wildlife and regeneration. To do so, you plan to only harvest from branches where the number of berries is greater than `threshold`.

Given the `root` of a binary tree representing a berry bush where each node represents the number of berries on a branch of the bush, write a function `harvest_berries()`, that finds the number of berries you can harvest by returning the sum of all nodes with value greater than `threshold`.

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 harvest_berries(root, threshold):
  pass
```

Example Usage:

```
"""
       4
     /   \
   10     6
  /  \     \
 5    8    20
"""
bush = TreeNode(4, TreeNode(10, TreeNode(5), TreeNode(8)), TreeNode(6, None, TreeNod

print(harvest_berries(bush, 6))
print(harvest_berries(bush, 30))
```

Example Output:

```
38
Example 1 Explanation:
- Nodes greater than 6: 8, 10, 20
- 8 + 10 + 20 = 38

0
Example 2 Explanation: No nodes greater than 30
```

Problem 8: Flower Fields

You're looking for the perfect bloom to add to your bouquet of flowers. Given the `root` of a binary tree representing flower options, and a target flower `flower`, return `True` if the bloom you are looking for each exists in the tree and `False` otherwise.

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 find_flower(root, flower):
  pass
```

Example Usage:

```
"""
         Rose
        /    \
       /      \
     Lily     Daisy
    /    \         \
Orchid  Lilac    Dahlia
"""

flower_field = TreeNode("Rose",
                        TreeNode("Lily", TreeNode("Orchid"), TreeNode("Lilac")),
                                TreeNode("Daisy", None, TreeNode("Dahlia")))

print(find_flower(flower_field, "Lilac"))
print(find_flower(flower_field, "Hibiscus"))
```

Example Output:

```
True
False
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Building an Underwater Kingdom

Given the following `TreeNode` class, create the binary tree depicted below. The text representing each node should should be used as the `value`.

The `root`, or topmost node in the tree `TreeNode("Poseidon")` has been provided for you.

```
            Poseidon
          /         \
      Atlantis      Oceania
      /     \       /     \
  Coral     Pearl  Kelp    Reef
```

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

root = TreeNode("Poseidon")
# Add your code here
```

Example Usage:

```
# Using print_tree() included at the top of this page
print_tree(root)
```

Example Output:

```
['Poseidon', 'Atlantis', 'Oceania', 'Pearl', 'Kelp', 'Reef']
```

▶ 💡 **Hint: Binary Trees**

Problem 2: Are Twins?

Given the `root` of a binary tree that has at most three nodes: the `root`, its left child, and its right child.

Return `True` if the `root`'s children are twins (have equal value) and `False` otherwise. If the `root` has no children, return `False`.

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 mertwins(root):
    pass
```

Example Usage:

```
"""
      Mermother
       /    \
    Coral   Coral
"""
root1 = TreeNode("Mermother", TreeNode("Coral"), TreeNode("Coral"))

"""
      Merpapa
       /    \
   Calypso  Coral
"""
root2 = TreeNode("Merpapa", TreeNode("Calypso"), TreeNode("Coral"))

"""
      Merenby
           \
         Calypso
"""
root3 = TreeNode("Merenby", None, TreeNode("Calypso"))

print(mertwins(root1))
print(mertwins(root2))
print(mertwins(root3))
```

Example Output:

```
True
False
False
```

Problem 3: Poseidon's Decision

Poseidon has received advice on an important matter from his council of advisors. Help him evaluate the advice from his council to make a final decision. You are given the advice as the `root` of a binary tree representing a boolean expression that has at most three nodes. The `root` may have exactly 0 or 2 children.

- **Leaf nodes** have a boolean value of either `True` or `False`.
- **Non-leaf nodes** have a string value of either `AND` or `OR`.

The **evaluation** of a node is as follows:

- If the node is a leaf node, the evaluation is the **value** of the node, i.e. `True` or `False`.
- Otherwise evaluate the node's two children and apply the boolean operation of its value with the children's evaluations.

Return the boolean result of evaluating 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.

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

def get_decision(root):
  pass
```

Example Usage:

```
"""
        OR
      /    \
    True  False
"""
expression1 = TreeNode("OR", TreeNode(True), TreeNode(False))

"""
       False
"""
expression2 = TreeNode(False)

print(get_decision(expression1))
print(get_decision(expression2))
```

Example Output:

```
True
False
```

Problem 4: Escaping the Sea Caves

You are given the `root` of a binary tree representing possible route through a system of sea caves. You recall that so long as you take the leftmost branch at every fork in the route, you'll find your way back home. Write a function `leftmost_path()` that returns an array with the value of each node in the leftmost 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. 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 leftmost_path(root):
    pass
```

Example Usage:

```
"""
        CaveA
       /     \
    CaveB    CaveC
    /   \        \
CaveD CaveE     CaveF
"""
system_a = TreeNode("CaveA",
                    TreeNode("CaveB", TreeNode("CaveD"), TreeNode("CaveE")),
                            TreeNode("CaveC", None, TreeNode("CaveF")))

"""
  CaveA
      \
      CaveB
        \
        CaveC
"""
system_b = TreeNode("CaveA", None, TreeNode("CaveB", None, TreeNode("CaveC")))

print(leftmost_path(system_a))
print(leftmost_path(system_b))
```

Example Output:

```
['CaveA', 'CaveB', 'CaveD']
['CaveA']
```

▶ 💡 **Hint: Balanced Trees**

Problem 5: Escaping the Sea Caves II

If you implemented `leftmost_path()` iteratively in the previous problem, implement it recursively. If you implemented it recursively, implement it iteratively.

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 leftmost_path(root):
    pass
```

Example Usage:

```
"""
        CaveA
       /      \
    CaveB    CaveC
    /   \        \
CaveD CaveE     CaveF
"""
system_a = TreeNode("CaveA",
                  TreeNode("CaveB", TreeNode("CaveD"), TreeNode("CaveE")),
                          TreeNode("CaveC", None, TreeNode("CaveF")))

"""
  CaveA
      \
      CaveB
        \
        CaveC
"""
system_b = TreeNode("CaveA", None, TreeNode("CaveB", None, TreeNode("CaveC")))

print(leftmost_path(system_a))
print(leftmost_path(system_b))
```

Example Output:

```
['CaveA', 'CaveB', 'CaveD']
['CaveA']
```

Problem 6: Documenting Reefs

You are exploring a vast coral reef system. The reef is represented as a binary tree, where each node corresponds to a specific coral formation. You want to document the reef as you encounter it, starting from the `root` or main entrance of the reef.

Write a function `explore_reef()` that performs a preorder traversal of the reef and returns a list of the names of the coral formations in the order you visited them. In a preorder exploration, you explore the current node first, then the left subtree, and finally the right subtree.

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 explore_reef(root):
    pass
```

Example Usage:

```
"""
         CoralA
        /     \
     CoralB  CoralC
     /   \
 CoralD CoralE
"""

reef = TreeNode("CoralA",
                TreeNode("CoralB", TreeNode("CoralD"), TreeNode("CoralE")),
                          TreeNode("CoralC"))

print(explore_reef(reef))
```

Example Output:

```
['CoralA', 'CoralB', 'CoralD', 'CoralE', 'CoralC']
```

▶ 💡 **Hint: Traversing Trees**

Problem 7: Coral Count

Due to climate change, you have noticed that coral has been dying in the reef near Atlantis. You want to ensure there is still a healthy level of coral in the reef. Given the `root` of a binary tree where each node represents a coral in the reef, write a function `count_coral()` that returns the number of corals in the reef.

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 count_coral(root):
    pass
```

Example Usage:

```
"""
          Staghorn
         /        \
        /          \
    Sea Fan      Sea Whip
    /     \       /
 Bubble  Table  Star
  /
Fire
"""
reef1 = TreeNode("Staghorn",
                TreeNode("Sea Fan", TreeNode("Bubble", TreeNode("Fire")), TreeNode("
                        TreeNode("Sea Whip", TreeNode("Star")))

"""
     Fire
    /    \
   /      \
Black    Star
        /
     Lettuce
           \
        Sea Whip
"""
reef2 = TreeNode("Fire",
                TreeNode("Black"),
                        TreeNode("Star",
                                 TreeNode("Lettuce", None, TreeNode("Sea Whip"))))

print(count_coral(reef1))
print(count_coral(reef2))
```

Example Output:

```
7
5
```

Problem 8: Ocean Layers

Given the `root` of a binary tree that represents different sections of the ocean, write a function `count_ocean_layers()` that returns the depth of the ocean. The **depth** or **height** of the tree can be defined as the number of nodes on the longest path from the root node to a 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 ocean_depth(root):
    pass
```

Example Usage:

```
"""
                Sunlight
               /        \
              /          \
          Twilight      Squid
         /       \           \
      Abyss  Anglerfish    Giant Squid
      /
  Trenches
"""
ocean = TreeNode("Sunlight",
                TreeNode("Twilight",
                        TreeNode("Abyss",
                                TreeNode("Trenches")), TreeNode("Anglerfish")),
                                        TreeNode("Squid", TreeNode("Giant Squid")))

"""
    Spray Zone
    /         \
   /           \
Beach       High Tide
            /
      Middle Tide
              \
            Low Tide
"""
tidal_zones = TreeNode("Spray Zone",
                      TreeNode("Beach"),
                              TreeNode("High Tide",
                                      TreeNode("Middle Tide", None, TreeNode("Low Ti

print(ocean_depth(ocean))
print(ocean_depth(tidal_zones))
```

Example Output:

```
4
4
```

Close Section

▸ Advanced Problem Set Version 1

▸ Advanced Problem Set Version 2