

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 continue to work with binary trees and are introduced to foundational algorithms for a common subcategory of trees: binary search trees (BSTs). The problems encourage exploring key operations such as inserting and removing nodes, checking tree balance, finding specific nodes, and traversing trees in various orders.

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

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.

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'

```

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

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

▼ Build Binary Tree Function

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

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

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

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

```

from collections import deque

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

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

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

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

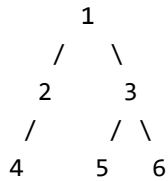
    while queue:
        node = queue.popleft()
        if index < len(values) and values[index] is not None:
            left_key, left_value = get_key_value(values[index])
            node.left = TreeNode(left_value, left_key)
            queue.append(node.left)
        index += 1
        if index < len(values)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

    return root

```

Example Usage:

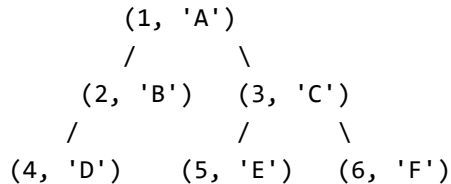
```
"""
```



```
"""
```

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

```
"""
```



```
"""
```

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

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

Example Output:

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

Breakout Problems Session 2

▼ Standard Problem Set Version 1

Problem 1: Monstera Madness

Given the `root` of a binary tree where each node represents the number of splits in a leaf of a Monstera plant, return the number of Monstera leaves that have an odd number of splits.

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_odd_splits(root):
    pass

```

Example Usage:

```

"""
    2
   / \
  /   \
 3     5
 / \   \
6  7   12
"""

# Using build_tree() function included at top of page
values = [2, 3, 5, 6, 7, None, 12]
monstera = build_tree(values)

print(count_odd_splits(monstera))
print(count_odd_splits(None))

```

Example Output:

```

3
0

```

▼ ✨ AI Hint: Traversing Trees

Key Skill: Use AI to explain code concepts

This problem requires you to traverse a binary tree. For a refresher on this topic, check out the Tree Traversal section of the Unit 8 Cheatsheet.

Still have questions? Try asking an AI tool like ChatGPT or GitHub Copilot to explain the different types of binary tree traversal. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please explain the different types of binary tree traversal, and show me how they would each work on an example tree."

Hint: Be sure to learn about "preorder", "postorder", and "inorder" traversals!

Problem 2: Flower Finding

You are looking to buy a new flower plant for your garden. The nursery you visit stores its inventory in a binary search tree (BST) where each node represents a plant in the store. The plants are organized according to their names (`vals`) in alphabetical order in the BST.

Given the root of the binary search tree `inventory` and a target flower `name` , write a function `find_flower()` that returns `True` if the flower is present in the garden 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(inventory, name):
    pass
```

Example Usage:

```
"""
    Rose
   /  \
Lilac  Tulip
 /  \      \
Daisy Lily  Violet
"""

# using build_tree() function at top of page
values = ["Rose", "Lilac", "Tulip", "Daisy", "Lily", None, "Violet"]
garden = build_tree(values)

print(find_flower(garden, "Lilac"))
print(find_flower(garden, "Sunflower"))
```

Example Output:

```
True
False
```

▼ ✨ AI Hint: Binary Search Trees

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with binary search trees (BSTs). This data structure is incredibly useful, and is often used in many coding interviews.

For a refresher on this topic, check out the Binary Search Tree section of the Unit 8 Cheatsheet.

To go deeper, you can ask an AI tool like ChatGPT or GitHub Copilot to explain the concept of binary search trees, how they work, and how to implement them in Python. You can also visit the VisuAlgo BST Visualizer to see how binary search trees work visually.

Problem 3: Flower Finding II

Consider the following function `non_bst_find_flower()` which accepts the root of a binary tree `inventory` and a flower `name`, and returns `True` if a flower (node) with `name` exists in the binary tree. Unlike the previous problem, this tree is **not** a binary search tree.

1. Compare your solution to `find_flower()` in Problem 2 to the following solution. Discuss with your group: How is the code different? Why?
2. What is the time complexity of `non_bst_find_flower()`? How does it compare to the time complexity of `find_flower()` in Problem 2?
3. How would the time complexity of `find_flower()` from Problem 2 change if the tree `inventory` was not balanced?

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

def non_bst_find_flower(root, name):
    if root is None:
        return False

    if root.val == name:
        return True

    return non_bst_find_flower(root.left, name) or non_bst_find_flower(root.right, name)
```

Example Usage:

```

"""
    Daisy
   /  \
  Lily  Tulip
 /  \   \
Rose Violet Lilac
"""

# using build_tree() function at top of page
values = ["Rose", "Lily", "Tulip", "Daisy", "Lilac", None, "Violet"]
garden = build_tree(values)

print(non_bst_find_flower(garden, "Lilac"))
print(non_bst_find_flower(garden, "Sunflower"))

```

Example Output:

```

True
False

```

Problem 4: Adding a New Plant to the Collection

You have just purchased a new houseplant and are excited to add it to your collection! Your collection is meticulously organized using a Binary Search Tree (BST) where each node in the tree represents a houseplant in your collection, and houseplants are organized alphabetically by name (`val`).

Given the root of your BST `collection` and a new houseplant `name`, insert a new node with value `name` into your collection. Return the root of your updated collection. If another plant with `name` already exists in the tree, add the new node in the existing node's 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 add_plant(collection, name):
    pass

```

Example Usage:

```

"""
        Money Tree
      /       \
Fiddle Leaf Fig  Snake Plant
"""

# Using build_tree() function at the top of page
values = ["Money Tree", "Fiddle Leaf Fig", "Snake Plant"]
collection = build_tree(values)

# Using print_tree() function at the top of page
print_tree(add_plant(collection, "Aloe"))

```

Example Output:

```
['Money Tree', 'Fiddle Leaf Fig', 'Snake Plant', 'Aloe']
```

Explanation:

Tree should have the following structure:

```

        Money Tree
      /       \
Fiddle Leaf Fig  Snake Plant
      /
    Aloe

```

▼ ✨ AI Hint: Binary Search Trees

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with binary search trees (BSTs). This data structure is incredibly useful, and is often used in many coding interviews.

For a refresher on this topic, check out the Binary Search Tree section of the Unit 8 Cheatsheet.

To go deeper, you can ask an AI tool like ChatGPT or GitHub Copilot to explain the concept of binary search trees, how they work, and how to implement them in Python. You can also visit the VisuAlgo BST Visualizer to see how binary search trees work visually.

Problem 5: Sorting Plants by Rarity

You are going to a plant swap where you can exchange cuttings of your plants for new plants from other plant enthusiasts. You want to bring a mix of cuttings from both common and rare plants in your collection. You track your plant collection in a BST where each node has a `key` and a `val`. The `val` contains the plant name, and the `key` is an integer representing the plant's rarity. Plants are organized in the BST by their `key`.

To help choose which plants to bring, write a function `sort_plants()` which takes in the BST root `collection` and returns an array of plant nodes as tuples in the form `(key, val)` sorted from least to most rare. Sorted order can be achieved by performing an **inorder traversal** of the BST.

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

def sort_plants(collection):
    pass
```

Example Usage:

```
"""
    (3, "Monstera")
   /      \
(1, "Pothos") (5, "Witchcraft Orchid")
  \      /
(2, "Spider Plant") (4, "Hoya Motoskei")
"""

# Using build_tree() function at the top of page
values = [(3, "Monstera"), (1, "Pothos"), (5, "Witchcraft Orchid"), None, (2, "Spider Plant")]
collection = build_tree(values)

print(sort_plants(collection))
```

Example Output:

```
[(1, 'Pothos'), (2, 'Spider Plant'), (3, 'Monstera'), (4, 'Hoya Motoskei'), (5, 'Witchcraft Orchid')]
```

Problem 6: Finding a New Plant Within Budget

You are looking for a new plant and have a max `budget`. The plant store that you are shopping at stores their inventory in a BST where each node has a `key` representing the price of the plant and `value` contains the plant's name. Plants are ordered by their prices. You want to find a plant that is close to but lower than your budget.

Given the root of the BST `inventory` and an integer `budget`, write a function `pick_plant()` that returns the plant with the highest price below `budget`. If no plant with a price strictly below `budget` exists, the function should return `None`.

```

class TreeNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key      # Plant price
        self.val = val      # Plant name
        self.left = left
        self.right = right

def pick_plant(root, budget):
    pass

```

Example Usage:

```

"""
        (50, "Fiddle Leaf Fig")
        /      \
    (25, "Monstera")  (70, "Snake Plant")
    /      \      /      \
(15, "Aloe") (40, "Pothos") (60, "Fern") (80, "ZZ Plant")
"""

# Using build_tree() function at the top of page
values = [(50, "Fiddle Leaf Fig"), (25, "Monstera"), (70, "Snake Plant"), (15, "Aloe"),
          (40, "Pothos"), (60, "Fern"), (80, "ZZ Plant")]
inventory = build_tree(values)

print(pick_plant(inventory, 50))
print(pick_plant(inventory, 25))
print(pick_plant(inventory, 15))

```

Example Output:

```

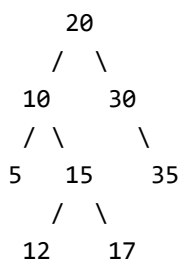
Pothos
Aloe
None

```

▼ 💡 Hint: Inorder Predecessor

The largest node smaller than a given node in a binary search tree is also called the **inorder predecessor**. The inorder predecessor of a node in a BST is the node that comes just before the given node in an inorder traversal of the tree.

Consider the following BST:



- The inorder predecessor 15 is 12.
- The inorder predecessor of 12 is 10.
- The inorder predecessor of 5 is `None` as it's the smallest node in the tree.

To find the inorder predecessor, we can follow the following steps:

1. If the node has a left subtree, the predecessor is the rightmost (largest) node in the left subtree.
2. If the node doesn't have a left subtree, you traverse upwards to find the deepest ancestor for which the given node lies in the right subtree.

Problem 7: Remove Plant

A plant in your houseplant collection has become infested with aphids, and unfortunately you need to throw it out. Given the root of a BST `collection` where each node represents a plant in your collection, and a plant `name`, remove the plant node with value `name` from the collection. Return the root of the modified collection. Plants are organized alphabetically in the tree by value.

If the node with `name` has two children in the tree, replace it with its **inorder predecessor** (rightmost node in its left subtree). You do not need to maintain a balanced tree.

Pseudocode has been provided for you.

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 remove_plant(collection, name):
    # Find the node to remove
    # If the node has no children
    #     # Remove the node by setting parent pointer to None
    # If the node has one child
    #     # Replace the node with its child
    # If the node has two children
    #     # Find the inorder predecessor
    #     # Replace the node's value with inorder predecessor value
    #     # Remove inorder predecessor
    # Return root of updated tree
    pass
```

Example Usage:

```

"""
    Money Tree
    /      \
  Hoya      Pilea
   \      /  \
   Ivy   Orchid ZZ Plant
"""

# Using build_tree() function at the top of page
values = ["Money Tree", "Hoya", "Pilea", None, "Ivy", "Orchid", "ZZ Plant"]
collection = build_tree(values)

# Using print_tree() function at the top of page
print_tree(remove_plant(collection, "Pilea"))

```

Example Output:

```
['Money Tree', 'Hoya', 'Orchid', None, 'Ivy', None, 'ZZ Plant']
```

Explanation:

The resulting tree structure:

```

    Money Tree
    /      \
  Hoya      Orchid
   \      \
   Ivy     ZZ Plant

```

[Close Section](#)

▼ Standard Problem Set Version 2

Problem 1: Find Lonely Cichlids

Sibling cichlid fish often form strong bonds after hatching, staying close to each other for protection. Given the `root` of a binary tree representing a family of cichlids where each node is a cichlid, return an array containing the values of all lonely cichlids in the family. A **lonely** cichlid is a fish (node) that is the only child of its parent. The matriarch (`root`) is not lonely because it does not have a parent. Return the array in any order.

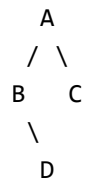
Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity. Assume the input tree is balanced when calculating time complexity.

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

def find_lonely_cichlids(root):
    pass
```

Example Input:


```
"""
```



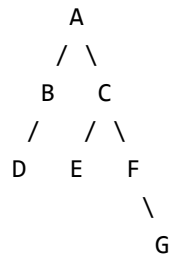
```
"""
```

Using build_tree() function at the top of page

```
values = ['A', 'B', 'C', None, 'D']
```

```
family_1 = build_tree(values)
```

```
"""
```

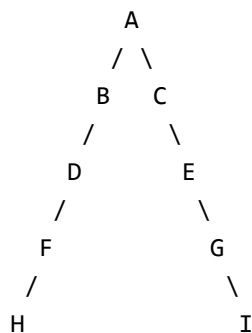


```
"""
```

```
values = ['A', 'B', 'C', None, 'D', None, 'E', 'F', None, None, None, None, None, 'G']
```

```
family_2 = build_tree(values)
```

```
"""
```



```
"""
```

```
values = ["A", "B", "C", "D", None, None, "E", "F", None, None, "G", "H", None, None, "I"]
```

```
family_3 = build_tree(values)
```

```
print(find_lonely_cichlids(family_1))
```

```
print(find_lonely_cichlids(family_2))
```

```
print(find_lonely_cichlids(family_3))
```

Example Output:

```
['D']
```

```
['D', 'G']
```

```
['D', 'F', 'H', 'E', 'G', 'I']
```

Note: The elements of the list may be returned in any order.

▼ ✨ AI Hint: Traversing Trees

Key Skill: Use AI to explain code concepts

This problem requires you to traverse a binary tree. For a refresher on this topic, check out the Tree Traversal section of the Unit 8 Cheatsheet.

Still have questions? Try asking an AI tool like ChatGPT or GitHub Copilot to explain the different types of binary tree traversal. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please explain the different types of binary tree traversal, and show me how they would each work on an example tree."

Hint: Be sure to learn about "preorder", "postorder", and "inorder" traversals!

Problem 2: Searching Ariel's Treasures

The mermaid princess Ariel is looking for a specific item in the grotto where she collects all the various objects from the human world she finds. Ariel's collection of human treasures is stored in a binary search tree (BST) where each node represents a different item in her collection. The items are organized according to their names (`val` s) in alphabetical order in the BST.

Given the root of the binary search tree `grotto` and a target object `treasure`, write a function `locate_treasure()` that returns `True` if `treasure` is present in the garden and `False` otherwise.

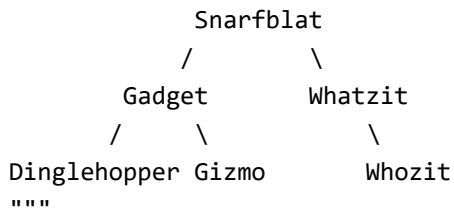
Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity. Assume the input tree is balanced when calculating time complexity.

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

def locate_treasure(grotto, treasure):
    pass
```

Example Usage:

```
"""
```



```
# Using build_tree() function at the top of page
values = ["Snarfblat", "Gadget", "Whatzit", "Dinglehopper", "Gizmo", None, "Whozit"]
grotto = build_tree(values)

print(locate_treasure(grotto, "Dinglehopper"))
print(locate_treasure(grotto, "Thingamabob"))
```

Example Output:

```
True
False
```

▼ ✨ AI Hint: Binary Search Trees

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with binary search trees (BSTs). This data structure is incredibly useful, and is often used in many coding interviews.

For a refresher on this topic, check out the Binary Search Tree section of the Unit 8 Cheatsheet.

To go deeper, you can ask an AI tool like ChatGPT or GitHub Copilot to explain the concept of binary search trees, how they work, and how to implement them in Python. You can also visit the VisuAlgo BST Visualizer to see how binary search trees work visually.

Problem 3: Add New Treasure to Collection

The mermaid princess Ariel and her pal Flounder visited a new shipwreck and found an exciting new human artifact to add to her collection. Ariel's collection of human treasures is stored in a binary search tree (BST) where each node represents a different item in her collection. Items are organized according to their names (`val`s) in alphabetical order in the BST.

Given the root of the binary search tree `grotto` and a string `new_item`, write a function `locate_treasure()` that adds a new node with value `new_item` to the collection and returns the `root` of the modified tree. If a node with value `new_item` already exists within the tree, return the original tree unmodified. You do not need to maintain balance in the 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 add_treasure(grotto, new_item):
        pass
```

Example Usage:

```
"""
        Snarfblat
       /      \
    Gadget    Whatzit
   /  \      \
Dinglehopper Gizmo Whozit
"""

# Using build_tree() function at the top of page
values = ["Snarfblat", "Gadget", "Whatzit", "Dinglehopper", "Gizmo", "Whozit"]
grotto = build_tree(values)

# Using print_tree() function included at top of page
print_tree(add_treasure(grotto, "Thingamabob"))
```

Example Output:

```
['Snarfblat', 'Gadget', 'Whatzit', 'Dinglehopper', 'Gizmo', None, 'Whozit']
Explanation:
Updated tree:
        Snarfblat
       /      \
    Gadget    Whatzit
   /  \      /  \
Dinglehopper Gizmo Thingamabob Whozit
```

Problem 4: Sorting Pearls by Size

You have a collection of pearls harvested from a local oyster bed. The pearls are organized by their size in a BST, where each node in the BST represents the size of a pearl.

A function `smallest_to_largest_recursive()` which takes in the BST root `pearls` and returns an array of pearl sizes sorted from smallest to largest has been provided for you.

Implement a new function `smallest_to_largest_iterative()` which provides a iterative solution, taking in the BST root `pearls` and returning an array of pearl sizes sorted from smallest to largest has been provided for you.

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 Pearl:
    def __init__(self, size, left=None, right=None):
        self.val = size
        self.left = left
        self.right = right

def smallest_to_largest_recursive(pearls):
    sorted_list = []

    def inorder_traversal(node):
        if node:
            inorder_traversal(node.left)
            sorted_list.append(node.val)
            inorder_traversal(node.right)

    inorder_traversal(pearls)
    return sorted_list

def smallest_to_largest_iterative(pearls):
    pass
```

Example Usage:

```
"""
    3
   / \
  /   \
 1     5
 \   / \
  2 4   8
"""

# Use build_tree() function at top of page
values = [3, 1, 5, None, 4, 8]
pearls = build_tree(values)

print(smallest_to_largest_recursive(pearls))
print(smallest_to_largest_iterative(pearls))
```

Example Output:

```
[1, 2, 3, 4, 5, 8]
[1, 2, 3, 4, 5, 8]
```

▼ 💡 **Hint: Recursive to Iterative Translations**

This problem can be solved using a stack! Often, when we want to translate recursive solutions to iterative solutions, the solution involves creating a stack. This is because behind the scenes, recursive solutions take advantage of a computer's in-built **call stack** which also uses a stack data structure. You can read more about the call stack in the Advanced Section of the Unit 7 cheatsheet if you are curious.

Problem 5: Smallest Pearl Above Minimum Size

You have a collection of pearls stored in a BST where each node represents a pearl with size `val`. You are looking for a pearl to gift the sea goddess, Yemaya. So as to not anger her, the pearl must be larger than `min_size`.

Given the root of a BST `pearls`, write a function `pick_pearl()` that returns the pearl with the smallest size above `min_size`. If no pearl with a size above `min_size` exists, the function should return `None`.

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 Pearl:
    def __init__(self, size, left=None, right=None):
        self.val = size
        self.left = left
        self.right = right

def pick_pearl(pearls, min_size):
    pass
```

```

"""
    3
   / \
  /   \
 1     5
  \   / \
   2 4   8
"""

# Use build_tree() function at top of page
values = [3, 1, 5, None, 4, 8]
pearls = build_tree(values)

print(pick_pearl(pearls, 3))
print(pick_pearl(pearls, 7))
print(pick_pearl(pearls, 8))

```

Example Output:

```

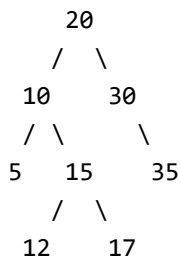
4
8
None

```

▼ 💡 Hint: Inorder Successor

The smallest node larger than a given node in a binary search tree is also called the **inorder successor**. The inorder successor of a node in a BST is the node that comes just after the given node in an inorder traversal of the tree.

Consider the following BST:



- The inorder successor 10 is 12.
- The inorder successor of 17 is 20.
- The inorder successor of 35 is as it's the largest node in the tree.

To find the inorder successor, we can follow the following steps:

1. If the node has a right subtree, the successor is the leftmost (smallest) node in the right subtree.
2. If the node doesn't have a right subtree, you traverse downwards to find the deepest ancestor for which the given node lies in the left subtree.

Problem 6: Remove Invasive Species

As a marine ecologist, you are worried about invasive species wreaking havoc on the local ecosystem. Given the root of a BST `ecosystem` where each node represents a species in a marine ecosystem, and an invasive species `name`, remove the species with value `name` from the ecosystem. Return the root of the modified ecosystem. Species are organized alphabetically in the tree by name (`val`).

If the node with `name` has two children in the tree, replace it with its **inorder successor** (leftmost node in its right subtree). You do not need to maintain a balanced tree.

Pseudocode has been provided for you.

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 remove_species(ecosystem, name):
    # Find the node to remove
    # If the node has no children
        # Remove the node by setting parent pointer to None
    # If the node has one child
        # Replace the node with its child
    # If the node has two children
        # Find the inorder successor
        # Replace the node's value with inorder successor value
        # Remove inorder successor
    # Return root of updated tree
    pass
```

Example Usage:


```

"""
        Dugong
      /   \
    Brain Coral  Lionfish
      \   /   \
        Clownfish Giant Clam  Seagrass
"""

# Use build_tree() function at top of page
values = ["Dugong", "Brain Coral", "Lionfish", None, "Clownfish", "Giant Clam", "Seagrass"]
ecosystem = build_tree(ecosystem)

# Using print_tree() function at top of page
print_tree(remove_species(ecosystem, "Lionfish"))

```

Example Output:

```
['Dugong', 'Brain Coral', 'Giant Clam', None, 'Clownfish', None, 'Seagrass']
```

Explanation:

The resulting tree structure:

```

      Dugong
    /   \
Brain Coral  Giant Clam
    \       \
  Clownfish  Seagrass

```

Problem 7: Minimum Difference in Pearl Size

You are analyzing your collection of pearls stored in a BST where each node represents a pearl with a specific size (`val`). You want to see if you have two pearls of similar size that you can make into a pair of earrings.

Write a function `min_diff_in_pearl_sizes()` that accepts the root of a BST `pearls`, and returns the minimum difference between the sizes of any two different pearls in the collection.

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 Pearl:
    def __init__(self, size=0, left=None, right=None):
        self.val = size
        self.left = left
        self.right = right

def min_diff_in_pearl_sizes(pearls):
    pass

```

Example Usage:

```
"""
    4
   / \
  2   6
 / \   \
1  3   8
"""

# Use build_tree() function at top of page
values = [4, 2, 6, 1, 3, None, 8]
pearls = build_tree(values)

print(min_diff_in_pearl_sizes(pearls))
```

Example Output:

```
1
Example Explanation: The difference between pearl sizes 3 and 4, or 2 and 3
```

[Close Section](#)

▼ Advanced Problem Set Version 1

Problem 1: Sorting Plants by Rarity

You are going to a plant swap where you can exchange cuttings of your plants for new plants from other plant enthusiasts. You want to bring a mix of cuttings from both common and rare plants in your collection. You track your plant collection in a binary search tree (BST) where each node has a `key` and a `val`. The `val` contains the plant name, and the `key` is an integer representing the plant's rarity. Plants are organized in the BST by their `key`.

To help choose which plants to bring, write a function `sort_plants()` which takes in the BST root `collection` and returns an array of plant nodes as tuples in the form `(key, val)` sorted from least to most rare. Sorted order can be achieved by performing an **inorder traversal** of the BST.

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, val, key, left=None, right=None):
        self.key = key      # Plant price
        self.val = val      # Plant name
        self.left = left
        self.right = right

def sort_plants(collection):
    pass
```

Example Usage:

```
"""
    (3, "Monstera")
   /  \
(1, "Pothos")  (5, "Witchcraft Orchid")
  \          /
(2, "Spider Plant") (4, "Hoya Motoskei")
"""

# Using build_tree() function at the top of page
values = [(3, "Monstera"), (1, "Pothos"), (5, "Witchcraft Orchid"), None, (2, "Spider Plant")]
collection = build_tree(values)

print(sort_plants(collection))
```

Example Output:

```
[(1, 'Pothos'), (2, 'Spider Plant'), (3, 'Monstera'), (4, 'Hoya Motoskei'), (5, 'Witchcraft Orchid')]
```

▼ ✨ AI Hint: Traversing Trees

Key Skill: Use AI to explain code concepts

This problem requires you to traverse a binary tree. For a refresher on this topic, check out the Tree Traversal section of the Unit 8 Cheatsheet.

Still have questions? Try asking an AI tool like ChatGPT or GitHub Copilot to explain the different types of binary tree traversal. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please explain the different types of binary tree traversal, and show me how they would each work on an example tree."

Hint: Be sure to learn about "preorder", "postorder", and "inorder" traversals!

Problem 2: Flower Finding

You are looking to buy a new flower plant for your garden. The nursery you visit stores its inventory in a binary search tree (BST) where each node represents a plant in the store. The plants are organized according to their names (`vals`) in alphabetical order in the BST.

Given the root of the binary search tree `inventory` and a target flower `name`, write a function `find_flower()` that returns `True` if the flower is present in the garden and `False` otherwise.

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

def find_flower(inventory, name):
    pass
```

Example Usage:

```
"""
    Rose
   /  \
  Lily Tulip
 /  \   \
Daisy Lilac Violet
"""

# using build_tree() function at top of page
values = ["Rose", "Lily", "Tulip", "Daisy", "Lilac", None, "Violet"]
garden = build_tree(values)

print(find_flower(garden, "Lilac"))
print(find_flower(garden, "Sunflower"))
```

Example Output:

```
True
False
```

▼ ✨ AI Hint: Binary Search Trees

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with binary search trees (BSTs). This data structure is incredibly useful, and is often used in many coding interviews.

For a refresher on this topic, check out the Binary Search Tree section of the Unit 8 Cheatsheet.

To go deeper, you can ask an AI tool like ChatGPT or GitHub Copilot to explain the concept of binary search trees, how they work, and how to implement them in Python. You can also visit the VisuAlgo BST Visualizer to see how binary search trees work visually.

Problem 3: Adding a New Plant to the Collection

You have just purchased a new houseplant and are excited to add it to your collection! Your collection is meticulously organized using a Binary Search Tree (BST) where each node in the tree represents a houseplant in your collection, and houseplants are organized alphabetically by name (`val`).

Given the root of your BST `collection` and a new houseplant `name`, insert a new node with value `name` into your collection. Return the root of your updated collection. If another plant with `name` already exists in the tree, add the new node in the existing node's right 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 and space complexity.

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

def add_plant(collection, name):
    pass
```

Example Usage:

```
"""
    Money Tree
    /      \
Fiddle Leaf Fig  Snake Plant
"""

# Using build_tree() function at the top of page
values = ["Money Tree", "Fiddle Leaf Fig", "Snake Plant"]
collection = build_tree(values)

# Using print_tree() function at the top of page
print_tree(add_plant(collection, "Aloe"))
```

Example Output:

```
['Money Tree', 'Fiddle Leaf Fig', 'Snake Plant', 'Aloe']
```

Explanation:

Tree should have the following structure:

```
      Money Tree
     /         \
Fiddle Leaf Fig Snake Plant
   /
  Aloe
```

Problem 4: Remove Plant

A plant in your houseplant collection has become infested with aphids, and unfortunately you need to throw it out. Given the root of a BST `collection` where each node represents a plant in your collection, and a plant `name`, remove the plant node with value `name` from the collection. Return the root of the modified collection. Plants are organized alphabetically in the tree by value.

If the node with `name` has two children in the tree, replace it with its **inorder predecessor** (rightmost node in its left subtree). You do not need to maintain a balanced tree.

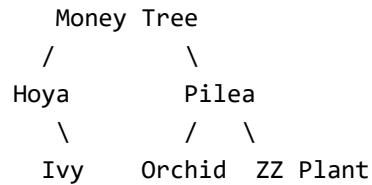
Evaluate the time and space complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity. Assume the input tree is balanced when calculating time and space complexity.

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

def remove_plant(collection, name):
    pass
```

Example Usage:

"""



"""

```
# Using build_tree() function at the top of page
values = ["Money Tree", "Hoya", "Pilea", None, "Ivy", "Orchid", "ZZ Plant"]
collection = build_tree(values)

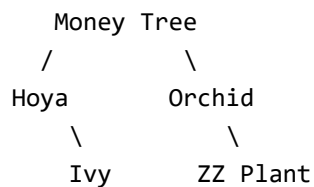
# Using print_tree() function at the top of page
print_tree(remove_plant(collection, "Pilea"))
```

Example Output:

```
['Money Tree', 'Hoya', 'Orchid', None, 'Ivy', None, 'ZZ Plant']
```

Explanation:

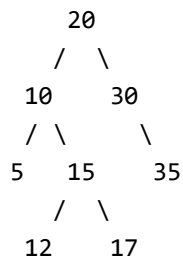
The resulting tree structure:



▼ 💡 Hint: Inorder Predecessor

The largest node smaller than a given node in a binary search tree is also called the **inorder predecessor**. The inorder predecessor of a node in a BST is the node that comes just before the given node in an inorder traversal of the tree.

Consider the following BST:



- The inorder predecessor 15 is 12.
- The inorder predecessor of 12 is 10.
- The inorder predecessor of 5 is as it's the smallest node in the tree.

To find the inorder predecessor, we can follow the following steps:

1. If the node has a left subtree, the predecessor is the rightmost (largest) node in the left subtree.
2. If the node doesn't have a left subtree, you traverse upwards to find the deepest ancestor for which the given node lies in the right subtree.

Problem 5: Find Most Common Plants in Collection

You have a vast plant collection and want to know which plants you own the most of. Given the `root` of a BST with duplicates where each node is a plant in your collection, return a list with the name(s) (`val`) of the most frequently occurring plant(s) in your collection. If multiple plants tie for the most frequently occurring plant, you may return them in any order.

Assume your BST organizes plants alphabetically by name and follows the following rules:

- The left subtree of a node contains only nodes with values **less than or equal** to the node's value
- The right subtree of a node contains only nodes with values **greater than or equal** to the node's value.
- Both the left and right subtrees must also be BSTs.

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

def find_most_common(root):
    pass
```

Example Usage:


```

"""
    Hoya
    \
    Pothos
    /
    Pothos
"""

# Using build_tree() function at top of page
values = ["Hoya", None, "Pothos", "Pothos"]
collection1 = build_tree(values)

"""
    Hoya
    /  \
  Aloe  Pothos
  /      \
 Aloe   Pothos
"""

values = ["Hoya", "Aloe", "Pothos", "Aloe", None, "Pothos"]
collection2 = build_tree(values)

print(find_most_common(collection1))
print(find_most_common(collection2))

```

Example Output:

```

['Pothos']
['Aloe', 'Pothos']

```

▼ 💡 Hint: Choosing a Traversal Order

Oftentimes when we solve a binary tree problem, the traversal order doesn't matter. But sometimes, it can help to specifically follow a preorder, inorder, or postorder traversal. This problem would most benefit from a inorder traversal because when traversing a BST, inorder traversal visits the nodes in sorted order from least to greatest. Since the plants are organized alphabetically, an inorder traversal will visit all occurrences of a plant consecutively. This makes it easy to count occurrences as we traverse the tree.

Problem 6: Split Collection

You've accumulated too many plants, and need to split up your collection. Given the root of a BST `collection` where each node represents a plant in your collection and a value `target`, split the tree into two subtrees where the first subtree has node values that are lexicographically

(alphabetically) smaller than or equal to `target` and the second subtree has all nodes that are greater than `target`. It is not necessarily the case that the collection contains a plant (node) with value `target`.

Additionally, most of the structure of the original tree should remain. Formally for any child plant `c` with parent `p` in the original collection, if they are both in the same subtree/subcollection after the split, then plant `c` should still have the parent `p`.

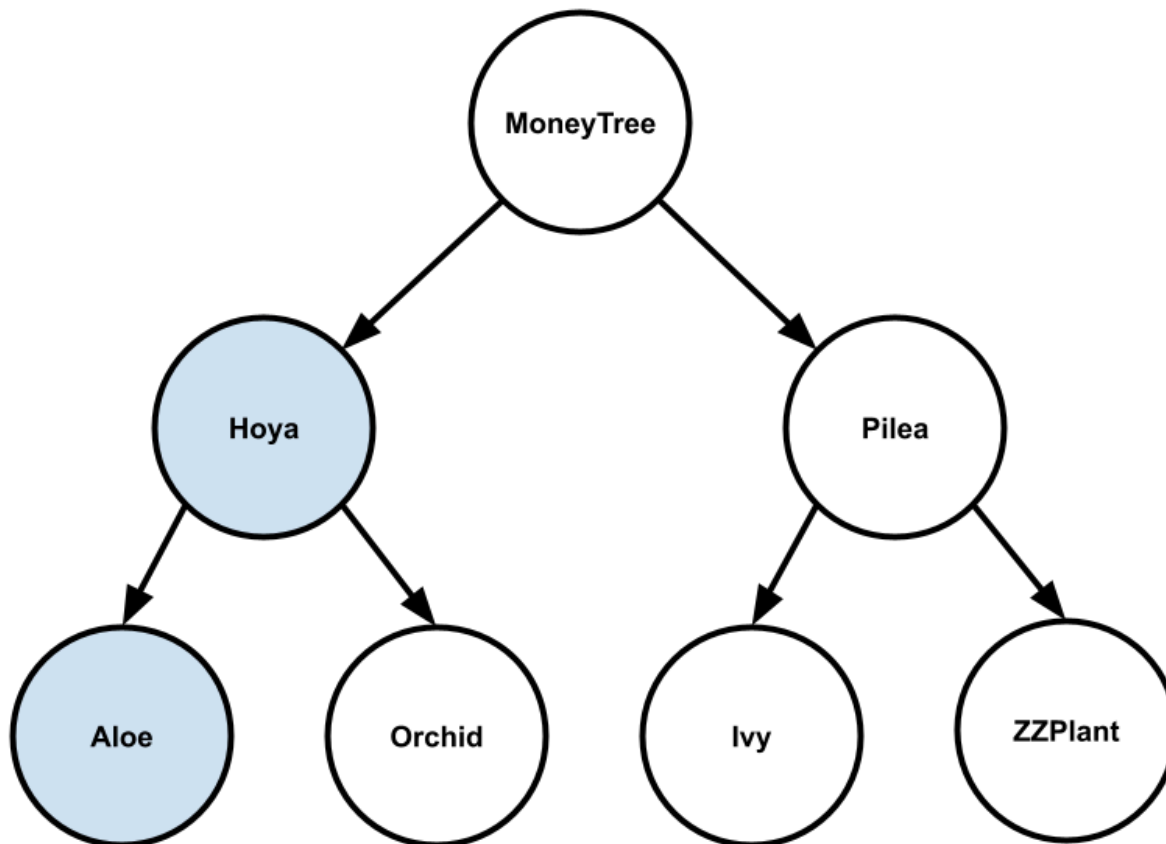
Return an array of the two root nodes of the two subtrees in order.

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

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

def split_collection(collection, target):
    pass
```

Example Usage:



```

"""
        Money Tree
      /      \
    Hoya      Pilea
   /  \    /  \
 Aloe  Ivy Orchid ZZ Plant
"""

# Using build_tree() function at the top of the page
values = ["Money Tree", "Hoya", "Pilea", "Aloe", "Ivy", "Orchid", "ZZ Plant"]
collection = build_tree(values)

# Using print_tree() function at the top of the page
left, right = split_collection(collection, "Hoya")
print_tree(left)
print_tree(right)

```

Example Output:

```

['Hoya', 'Aloe']
['Money Tree', 'Ivy', 'Pilea', None, None, 'Orchid', 'ZZ Plant']

Explanation:
Left Subtree:
  Hoya
  /
Aloe

Right Subtree:
  Money Tree
  /      \
Ivy      Pilea
   /      \
Orchid  ZZ Plant

```

Problem 7: Pruning Pothos

You have a Pothos plant represented as a binary tree, where each node in the tree represents a segment of the plant. Given the `root` of your pothos and a value `target`, you want to delete all **leaf nodes** with value `target`.

Note that once you delete a leaf node with value `target`, if its parent node becomes a leaf node and has the value `target`, it should also be deleted. You should continue deleting nodes until you cannot.

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.

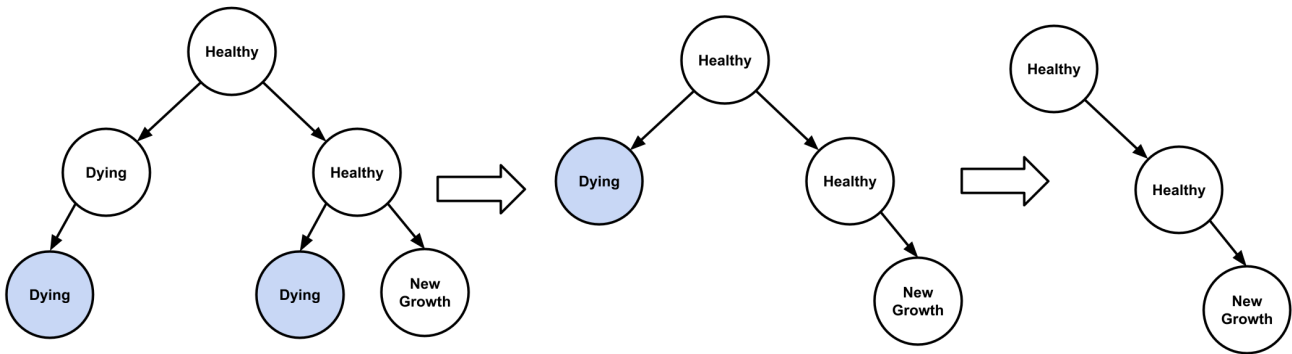
```

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

def prune(root, target):
    pass

```

Example Usage 1:



```

"""
    Healthy
   /    \
 Dying   Healthy
 /      /  \
Dying  Dying New Growth
"""

# Using build_tree() function at the top of the page
values = ["Healthy", "Dying", "Healthy", "Dying", None, "Dying", "New Growth"]
pothos1 = build_tree(values)

# Using print_tree() function at the top of the page
print_tree(prune(pothos1, "Dying"))

```

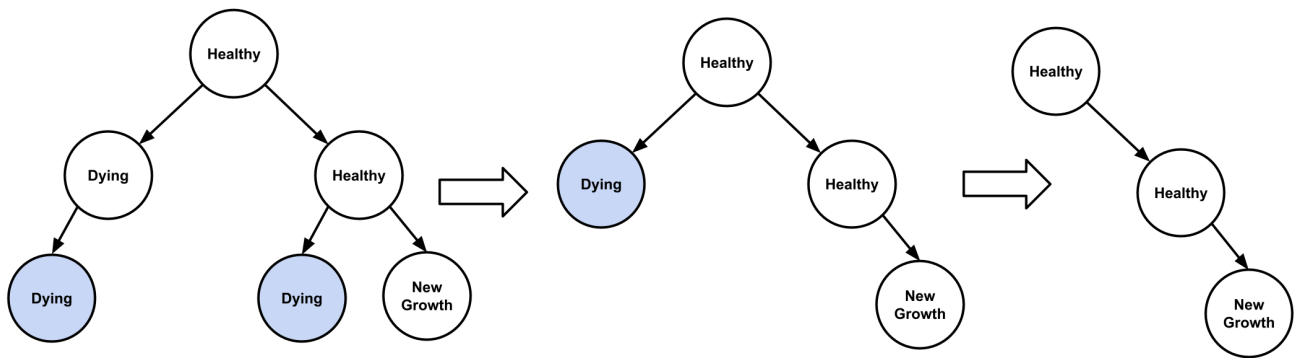
Example Output:

```

['Healthy', None, 'Healthy', None, 'New Growth']
Explanation:
Modified Tree:
Healthy
 \
  Healthy
   \
    New Growth

```

Example Usage 2:



```
"""
```

```
    Healthy
   /      \
  Aphids   Aphids
 /    \
Aphids New Growth
"""
```

```
values = ["Healthy", "Aphids", "Aphids", "Aphids", "New Growth"]
pothos2 = build_tree(values)

print_tree(prune(pothos2, "Aphids"))
```

Example Output 2:

```
['Healthy', 'Aphids', None, None, 'New Growth']
```

Explanation:

Modified Tree:

```
    Healthy
   /
  Aphids
   \
    New Growth
```

Problem 8: Find the Lowest Common Ancestor in a Plant Tree Based on Species Names

Given the `root` of a binary tree where each node represents a different plant species, return the value of the lowest common ancestor (LCA) of two given plants in the tree based on their species names. The species names are represented as strings, and the tree is structured according to lexicographical order (alphabetical order). The lowest common ancestor is defined between two species `p` and `q` as the lowest node in the tree that has both `p` and `q` as descendants (where we allow a node to be a descendant of itself).

Assume all plants are a unique species. Note that each `TreeNode` has a reference 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. Assume the input tree is balanced when calculating time and space complexity.

Note: the `build_tree()` function will not work for this problem because of the extra `parent` attribute. You must create your own tree manually for testing.

```
class TreeNode():
    def __init__(self, species, parent=None, left=None, right=None):
        self.val = species
        self.parent = parent # Parent of node
        self.left = left
        self.right = right

def lca(root, p, q):
    pass
```

Example Usage:

```
"""
      fern
     /  \
    /    \
   /      \
  cactus   rose
 /  \     /  \
bamboo dahlia lily oak
"""

fern = TreeNode("fern")
cactus = TreeNode("cactus", fern)
rose = TreeNode("rose", fern)
bamboo = TreeNode("bamboo", cactus)
dahlia = TreeNode("dahlia", cactus)
lily = TreeNode("lily", rose)
oak = TreeNode("oak", rose)

fern.left, fern.right = cactus, rose
cactus.left, cactus.right = bamboo, dahlia
rose.left, rose.right = lily, oak

print(lca(fern, "cactus", "rose"))
print(lca(fern, "bamboo", "oak"))
```

Example Output:

```
fern
Example 1 Explanation: The lowest common ancestor of "cactus" and "rose" is "fern" because "fern"
is the lowest node in the tree that has both "cactus" and "rose" as descendants.

cactus
Example 2 Explanation: The lowest common ancestor of "bamboo" and "oak" is "fern" because "fern"
is the lowest node in the tree that has both "bamboo" and "dahlia" as descendants.
```

▼ Advanced Problem Set Version 2

Problem 1: Sorting Pearls by Size

You have a collection of pearls harvested from a local oyster bed. The pearls are organized by their size in a BST, where each node in the BST represents the size of a pearl.

A function `smallest_to_largest_recursive()` which takes in the BST root `pearls` and returns an array of pearl sizes sorted from smallest to largest has been provided for you.

Implement a new function `smallest_to_largest_iterative()` which provides a iterative solution, taking in the BST root `pearls` and returning an array of pearl sizes sorted from smallest to largest has been provided for you.

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 Pearl:
    def __init__(self, size, left=None, right=None):
        self.val = size
        self.left = left
        self.right = right

def smallest_to_largest_recursive(pearls):
    sorted_list = []

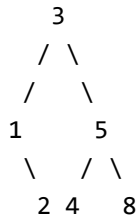
    def inorder_traversal(node):
        if node:
            inorder_traversal(node.left)
            sorted_list.append(node.val)
            inorder_traversal(node.right)

    inorder_traversal(pearls)
    return sorted_list

def smallest_to_largest_iterative(pearls):
    pass
```

Example Usage:

```
"""
```



```
"""
```

```
# Using build_tree() from the top of the page
values = [3, 1, 5, None, 2, 4, 8]
pearls = build_tree(values)

print(smallest_to_largest_recursive(pearls))
print(smallest_to_largest_iterative(pearls))
```

Example Output:

```
[1, 2, 3, 4, 5, 8]
[1, 2, 3, 4, 5, 8]
```

▼ 💡 Hint: Recursive to Iterative Translations

This problem can be solved using a stack! Often, when we want to translate recursive solutions to iterative solutions, the solution involves creating a stack. This is because behind the scenes, recursive solutions take advantage of a computer's in-built **call stack** which also uses a stack data structure. You can read more about the call stack in the Advanced Section of the Unit 7 cheatsheet if you are curious.

Problem 2: Searching Ariel's Treasures

The mermaid princess Ariel is looking for a specific item in the grotto where she collects all the various objects from the human world she finds. Ariel's collection of human treasures is stored in a binary search tree (BST) where each node represents a different item in her collection. The items are organized according to their names (`vals`) in alphabetical order in the BST.

Given the root of the binary search tree `grotto` and a target object `treasure`, write a function `locate_treasure()` that returns `True` if `treasure` is present in the garden and `False` otherwise.

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.

Hint: Intro to Binary Search Trees


```

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

def locate_treasure(grotto, treasure):
    pass

```

Example Usage:

```

"""
        Snarfblat
       /    \
    Gadget  Whatzit
   /  \    /  \
Dinglehopper Gizmo Whozit
"""

# Using build_tree() function at the top of page
values = ["Snarfblat", "Gadget", "Whatzit", "Dinglehopper", "Gizmo", "Whozit"]
grotto = build_tree(values)

print(locate_treasure(grotto, "Dinglehopper"))
print(locate_treasure(grotto, "Thingamabob"))

```

Example Output:

```

True
False

```

Problem 3: Add New Treasure to Collection

The mermaid princess Ariel and her pal Flounder visited a new shipwreck and found an exciting new human artifact to add to her collection. Ariel's collection of human treasures is stored in a binary search tree (BST) where each node represents a different item in her collection. Items are organized according to their names (`val`s) in alphabetical order in the BST.

Given the root of the binary search tree `grotto` and a string `new_item`, write a function `locate_treasure()` that adds a new node with value `new_item` to the collection and returns the `root` of the modified tree. If a node with value `new_item` already exists within the tree, return the original tree unmodified. You do not need to maintain balance in the 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. Assume the input tree is balanced when calculating time and space complexity.

```

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

def add_treasure(grotto, new_item):
    pass

```

Example Usage:

```

"""
        Snarfblat
       /      \
    Gadget    Whatzit
   /  \      \
Dinglehopper Gizmo  Whozit
"""

# Using build_tree() function at the top of page
values = ["Snarfblat", "Gadget", "Whatzit", "Dinglehopper", "Gizmo", "Whozit"]
grotto = build_tree(values)

# Using print_tree() function included at top of page
print_tree(add_treasure(grotto, "Thingamabob"))

```

Example Output:

```

['Snarfblat', 'Gadget', 'Whatzit', 'Dinglehopper', 'Gizmo', None, 'Whozit']
Explanation:
Updated tree:
        Snarfblat
       /      \
    Gadget    Whatzit
   /  \      /  \
Dinglehopper Gizmo Thingamabob Whozit

```

Problem 4: Remove Invasive Species

As a marine ecologist, you are worried about invasive species wreaking havoc on the local ecosystem. Given the root of a BST `ecosystem` where each node represents a species in a marine ecosystem, and an invasive species `name`, remove the species with value `name` from the ecosystem. Return the root of the modified ecosystem. Species are organized alphabetically in the tree by name (`val`).

If the node with `name` has two children in the tree, replace it with its **inorder successor** (leftmost node in its right subtree). You do not need to maintain a 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. Assume the input tree is balanced when calculating time and space complexity.

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

def remove_species(ecosystem, name):
    pass
```

Example Usage:

```
"""
        Dugong
      /   \
Brain Coral Lionfish
  \   /   \
Clownfish Giant Clam Seagrass
"""
# Use build_tree() function at top of page
values = ["Dugong", "Brain Coral", "Lionfish", None, "Clownfish", "Giant Clam", "Seagrass"]
ecosystem = build_tree(ecosystem)

# Using print_tree() function at top of page
print_tree(remove_species(ecosystem, "Lionfish"))
```

Example Output:

```
['Dugong', 'Brain Coral', 'Giant Clam', None, 'Clownfish', None, 'Seagrass']
```

Explanation:

The resulting tree structure:

```
      Dugong
    /   \
Brain Coral Giant Clam
  \       \
Clownfish  Seagrass
```

Problem 5: Minimum Difference in Pearl Size

You are analyzing your collection of pearls stored in a BST where each node represents a pearl with a specific size (`val`). You want to see if you have two pearls of similar size that you can make into a pair of earrings.

Write a function `min_diff_in_pearl_sizes()` that accepts the root of a BST `pearls`, and returns the minimum difference between the sizes of any two different pearls in the collection.

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 Pearl:
    def __init__(self, size=0, left=None, right=None):
        self.val = size
        self.left = left
        self.right = right

def min_diff_in_pearl_sizes(pearls):
    pass
```

Example Usage:

```
"""
    4
   / \
  2   6
 / \   \
1  3   8
"""

# Use build_tree() function at top of page
values = [4, 2, 6, 1, 3, None, 8]
pearls = build_tree(values)

print(min_diff_in_pearl_sizes(pearls))
```

Example Output:

```
1
Example Explanation: The difference between pearl sizes 3 and 4, or 2 and 3
```

Problem 6: Minimum Ocean Depth

You have just finished surveying a new, previously unexplored part of the ocean and want to find the shallowest part. Given the `root` of a binary tree representing this new part of the ocean, return its minimum depth. The minimum depth is the number of nodes along the shortest path from the `root` down to the nearest leaf 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. Assume the input tree is balanced when calculating time and space complexity.

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

def find_min_depth(root):
    pass
```

Example Usage:

```
"""
    Shipwreck
  /      \
Shallows  Reef
      /    \
    Cave    Trench
"""

# Using build_tree() function at top of page
values = ["Shipwreck", "Shallows", "Reef", None, None, "Cave", "Trench"]
ocean = build_tree(values)

print(find_min_depth(ocean))
```

Example Output:

2

Problem 7: Combining Shipwreck Loot

The mermaid princess Ariel and her friend Flounder have just finished exploring a new shipwreck and have each stored the items they found in a BST. Given the roots of two binary search trees, `root1` and `root2` where each node represents an item found in the shipwreck, return a list containing all the node values from **both trees** in lexographic (alphabetic) order. The tree nodes are organized in lexographic order within each 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. Assume the input tree is balanced when calculating time and space complexity.

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

def combine_loot(root1, root2):
    pass
```

Example Usage:

```
"""
    Fork          Coin
   /  \        /  \
Coin  Statue Anchor Mirror
"""

root1 = TreeNode("Fork", TreeNode("Coin"), TreeNode("Statue"))
root2 = TreeNode("Coin", TreeNode("Anchor"), TreeNode("Mirror"))

"""
    Fork          Necklace
     \          /
    Necklace  Fork
"""

root3 = TreeNode("Fork", None, TreeNode("Necklace"))
root4 = TreeNode("Necklace", TreeNode("Fork"))

print(combine_loot(root1, root2))
print(combine_loot(root3, root4))
```

Example Output:

```
['Anchor', 'Coin', 'Coin', 'Fork', 'Mirror', 'Statue']
['Fork', 'Fork', 'Necklace', 'Necklace']
```

Problem 8: Distributing Sunken Treasure

You and your friends have found a ship wreck full of gold pieces as part of a shipwreck and want to distribute the gold evenly amongst yourselves as efficiently as possible.

You are given the `root` of a binary tree with `n` nodes representing you and your friends where each friend currently has `node.val` coins. There are `n` coins in the whole tree (one for each of you!).

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

Return the minimum number of moves required to make every node have exactly one coin.

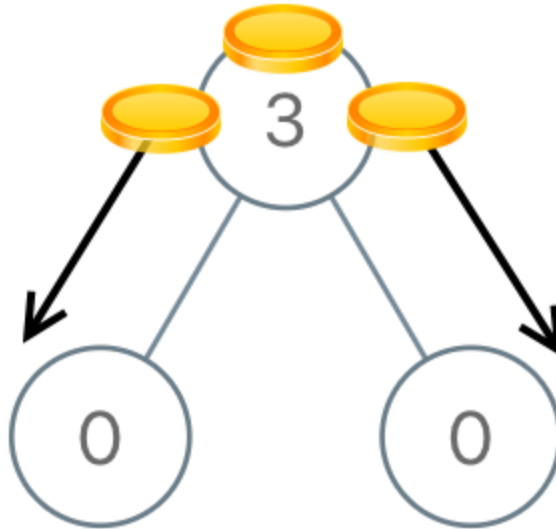
Evaluate the time and space complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity. Assume the input tree is balanced when calculating time and space complexity.

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

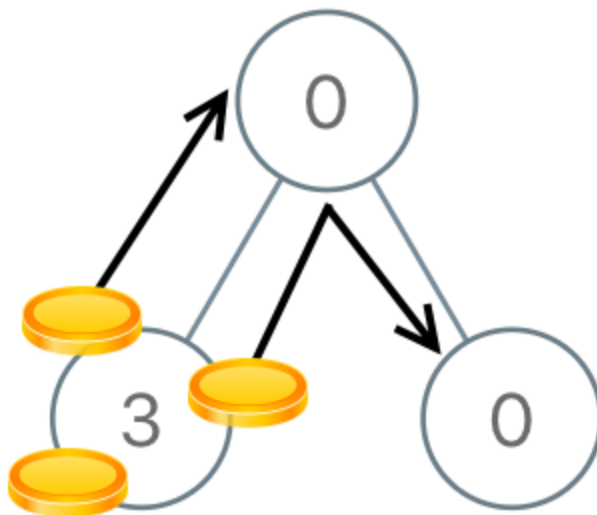
def distribute_coins(root):
    pass
```

Example Usage:

Example 1:



Example 2:



```

"""
    3
   / \
  0   0
"""
root1 = TreeNode(3, TreeNode(0), TreeNode(0))

"""
    0
   / \
  3   0
"""
root2 = TreeNode(0, TreeNode(3), TreeNode(0))

print(distribute_coins(root1))
print(distribute_coins(root2))

```

Example Output:

```

2
Example 1 Explanation: From the root of the tree, we move one coin to its left child,
and one coin to its right child.

3
Example 1 Explanation: From the left child of the root, we move two coins to the root
[taking two moves]. Then, we move one coin from the root of the tree to the right child.

```

▼ Hint: Choosing a Traversal Order

Oftentimes when we solve a binary tree problem, the traversal order doesn't matter. But sometimes, it can help to specifically follow a preorder, inorder, or postorder traversal. This problem would most benefit from a postorder traversal because postorder processes each subtree starting from the leaves and works its way up to the root. This is necessary because we want to know how many extra or deficient coins each subtree has before balancing the parent node.

Using a postorder approach, by the time we process a parent node, the coins in its children's subtrees will already been balanced, allowing us to decide how many coins should move between the parent and its children.

Close Section