UofT DSI Algorithms

Assignment 2

Submitted by Suzanne M Chalambalacis 1-Mar-2024

Objective

The partner assignment aims to provide students with the opportunity to practice coding in an interview context. You will analyze your partner's Assignment 1. Moreover, code reviews are common practice in a software development team. This assignment should give you a taste of the code review process.

Part 1:

You and your partner should send to each other your Assignment 1 submission.

Partner is Leila Samar and her Assignment 1 can be found here: https://github.com/LeilaSamar/Algorithms.git

Leila completed Question 1, which is outlined below.

```
(hash('Leila')%3)+1
```

```
1
```

Question One: Check Duplicates in Tree

Given the root of a binary tree, check whether it is contains a duplicate value. If a duplicate exists, return the duplicate value. If there are multiple duplicates, return the one with the closest distance to the root. If no duplicate exists, return -1.

Examples

Example 1

Input: `root = [1, 2, 2, 3, 5, 6, 7]` *What traversal method is this?*

Output: 2

Example 2

Input: `root = [1, 10, 2, 3, 10, 12, 12]`

Output: 10

Example 3

Input: `root = [10, 9, 8, 7]`

Output: -1

Starter Code

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val = 0, left = None, right = None):
#         self.val = val
#         self.left = left
#         self.right = right
def is_symmetric(root: TreeNode) -> int:
    # TODO
```

Part 2:

Create a Jupyter Notebook, create 6 of the following headings, and complete the following about the your partner's assignment 1:

Paraphrase the problem in your own words

Answer:

Given the root of a binary tree, the task is to determine whether the tree contains any duplicate values. If duplicates exist, the goal is to find and return the duplicate value that is closest to the root by distance. If there are multiple duplicates, the one with the minimum distance should be returned. If no duplicates exist, the function should return -1.

Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it.

Answer:

My example

Input: `root = [5, 3, 8, 2, 6, 3, 9, 7]`

Output: 3

In my example the example, the binary tree has a duplicate value of 3.

Leila's Example 1

Input: `root = [8, 8, 10, 15, 20, 10]`

Output: 8

In Leila's Example 1, the binary tree can be visualized as:

```
    8
   / \
  8   10
     / \
    15  20
       /
      10
```

In Leila's Example 1, the binary tree can be traced and explained as follows:

1. Starting with the root node `8`, we can move down the left child `8` which is a duplicate value at a depth of `0`. This is the end of this branch with a duplicate value `8` at a depth of `0`.
2. Starting with the root node `8`, we can move down the right child `10` which is not a duplicate, then move to it's left child `15` which is also not a duplicate. This is the end of this branch with no duplicates.
3. Returning to the child node `10`, we can move down the left child `20` which is not a duplicate, then move to it's left child `10` which is a duplicate value at a depth of `1`. This is the end of this branch with a duplicate value of `10` at a depth of `1`.
4. The duplicate value of `10` at a depth of `1` has a greater distance then the previous duplicate value of `8` at a depth of `0`, therefore, the output is `8` as this is the closest duplicate value to the root.

Copy the solution your partner wrote.

Answer:

```
## Full code in Python to solve the Check Duplicates in Tree problem using the provided s
# Definition for a binary tree node.
# class TreeNode(object):
#   def __init__(self, val=0, left=None, right=None):
#     self.val = val
#     self.left = left
#     self.right = right

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

def is_symmetric(root: TreeNode) -> int:
  def dfs(node, depth, seen):
    if not node:
      return -1

    if node.val in seen:
      return depth

    seen.add(node.val)

    left = dfs(node.left, depth + 1, seen.copy())
    right = dfs(node.right, depth + 1, seen)

    if left != -1 and right != -1:
      return min(left, right)
    return max(left, right)

  return dfs(root, 0, set())

# Test the function
# Sample tree creation for testing
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print(is_symmetric(root)) # Output: 2
```

```
    -1
```

Explain why their solution works in your own words.

Answer:

For the Check Duplicates in Tree problem. Leila's code uses a depth-first search (DFS) approach to traverse the binary tree and keeps track of the seen values at each depth. If a duplicate value is encountered, it compares the depths and returns the one with the minimum depth. If there are no duplicates, it returns -1.

Leila has also included a test case with a sample tree to verify the function's correctness.

Explain the problem's time and space complexity in your own words.

Answer:

Time Complexity:

The time complexity of the solution is O(n), where `n` is the number of nodes in the binary tree. This is because the solution performs a depth-first traversal of the entire tree, visiting each node once.

Space Complexity:

The space complexity is O(h), where `h` is the height of the binary tree. In the worst case, the space required for the set of seen values is proportional to the height of the tree. This is because the depth-first traversal involves recursive calls, and the depth of the recursion is determined by the height of the tree.

Critique your partner's solution, including explanation, if there is anything should be adjusted.

Answer:

The solution overall is well-structured and correctly addresses the problem. It uses a depth-first search (DFS) approach to traverse the binary tree, keeping track of the seen values and their depths. The function then returns the closest duplicate value.

Part 3: Reflection

Please write a 200 word reflection documenting your studying process from assignment 1, and your presentation and reviewing experience with your partner at the bottom of the Juypter Notebook under a new heading "Reflection." Again, export this Notebook as pdf.

Answer:

Solving the "Check Duplicates in Tree" problem was interesting because it required a systematic approach to understand the problem and design an efficient solution. After reviewing the problem statement, I identified the key requirements such as finding the closest duplicate value and handling multiple occurrences. I think Leila's approach with the depth-first traversal was the most suitable choice for tree exploration.

The problem/exercise involved breaking down the problem into manageable components and designing an algorithm that could handle the occurrences of multiple duplicates. I found it to be an interesting exercise that allowed me to further understand the binary tree structures and the principles of recursion, especially since the recursive nature of the algorithm required keeping track of seen values at different depths.

Reviewing a partner's code also emphasised the importance of clarity and simplicity in code presentation, and the importance of including notes along the way. This has been discussed in multiple instructors for this course and this experience helps underscores why we should do that with our codes and why it's important when collaborating with others.