Practice Interview

Objective

The partner assignment aims to provide participants 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.

Group Size

Each group should have 2 people. You will be assigned a partner

Part 1:

You and your partner must share each other's Assignment 1 submission.

Part 2:

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

- Paraphrase the problem in your own words.

The goal is to take a list of values representing a binary tree in level-order and build the tree. Then, find all possible paths from the root to the leaf nodes. Each path should be returned as a list of values, and the final output is a list of all such paths. If the input list is empty, the output should be an empty list.

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

Example

- **Input**: `root = [8, 4, 10, 2, 6, None, 20, 1]`
- **Tree Structure**:

In [9]:
```
      8
    /   \
   4      10
  / \       \
 2   6       20
```

```
  File <tokenize>:4
    / \       \
    ^
IndentationError: unindent does not match any outer indentation level
```

- **Output**: `[[8, 4, 2, 1], [8, 4, 6], [8, 10, 20]]`

- Copy the solution your partner wrote.

In [4]:
```
# 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 bt_path(values):
  """Finds all root-to-leaf paths in a binary tree, given as a list of va

  # Step 1: Build the binary tree from the list
  if not values:
      return []

  root = TreeNode(values[0])
  queue = [root]
  idx = 1
  while queue and idx < len(values):
      node = queue.pop(0)
      if values[idx] is not None:
          node.left = TreeNode(values[idx])
          queue.append(node.left)
      idx += 1
      if idx < len(values) and values[idx] is not None:
          node.right = TreeNode(values[idx])
          queue.append(node.right)
      idx += 1

  # Step 2: Perform DFS to find all root-to-leaf paths
  def dfs(node, path, paths):
      if not node:
          return
      path.append(node.val)
      if not node.left and not node.right:
          paths.append(path[:])  # Add a copy of the current path
      else:
          dfs(node.left, path, paths)
          dfs(node.right, path, paths)
      path.pop()

  paths = []
  dfs(root, [], paths)
  return paths

# Example 1
bt_path([4, 2, 6, 1, 3, 5, 7])
# Output: [[4, 2, 1], [4, 2, 3], [4, 6, 5], [4, 6, 7]]
# Example 2
bt_path([5, 1, 9, None, 3, 7, None, None, None, 4])
# Output: [[5, 1, 3, 4], [5, 9, 7]]
```

```
Out[4]: [[5, 1, 3], [5, 9, 7, 4]]
```

- Explain why their solution works in your own words.

The solution consists of two main parts:

Building the Tree:

- The function takes a list of values and creates a binary tree by processing each value in level-order.
- Each non- `None` value is added as a child of the current node, ensuring the tree is built correctly.

Finding Paths:

- Using depth-first search (DFS), the function explores all possible paths from the root to the leaf nodes.
  - For each node, its value is added to the current path.
  - When a leaf node (a node with no children) is reached, the path is saved as part of the result.
  - After exploring a path, the function backtracks to try other paths by removing the last added value.
- This process ensures that all root-to-leaf paths are found and returned.

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

Time Complexity

Building the Tree:

- The function processes each value in the input list exactly once to construct the binary tree. This step grows linearly with the size of the input list.

Finding Paths:

- The depth-first search visits every node in the tree one time. Since it doesn't revisit nodes, the time required is also proportional to the number of nodes in the tree.

Total:

- The overall time complexity is proportional to the total number of nodes, making it (O(n)).

Space Complexity

Tree Storage:

- The tree structure itself needs memory to store all the nodes created from the input. This takes space proportional to the number of nodes.

DFS Memory Usage:

- While finding paths, the function uses memory to keep track of the current path and the recursive call stack. The amount of memory used depends on the height of the tree (deeper trees require more memory).

Storing Results:

- The result list, which contains all paths from the root to the leaves, also requires memory. If there are many paths and each path is long, this can take up significant space.

Overall:

- The total memory used depends on the size of the tree, its depth, and the number of paths found.

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

Strengths:

- The solution is clear, logically organized, and meets the problem requirements.
- It uses an efficient approach for both tree construction and pathfinding.
- The recursive depth-first search is an appropriate and effective method for solving the problem.

Areas for Improvement:

1. **Input Validation**:
    - The code does not check for invalid or malformed inputs (e.g., non-list inputs or lists containing invalid values). Adding input validation would make the function more robust.
2. **Edge Case Handling**:
    - Scenarios like an empty input list (`[]`) or a tree with only `None` values (`[None]`) should be explicitly tested and handled to avoid unexpected behavior.
3. **Output Formatting**:
    - Sorting the paths before returning them could make the output more consistent, especially if the results are being compared with others.

Part 3:

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

Reflection

Working through Assignment 1 posed unique challenges, particularly in tackling a problem that required a deep understanding of binary tree structures and traversal algorithms. My assigned task involved identifying duplicate values in a binary tree, while my partner’s task focused on finding all root-to-leaf paths in a tree. Despite the difference in our questions, the underlying data structure was the same, which helped me draw parallels between our solutions.

Analyzing my partner’s work was both insightful and challenging. Their problem required constructing a binary tree from a list and then implementing depth-first search (DFS) to find paths, which differed significantly from my breadth-first search (BFS)-based solution for detecting duplicates. Breaking down their code to understand its logic and efficiency allowed me to see how they approached recursion and backtracking, two concepts I found less intuitive. I created additional test cases to validate their solution and found their implementation robust, though I suggested improvements for handling edge cases and invalid inputs.

Explaining their solution in my own words and critiquing it constructively pushed me to engage deeply with their logic. This process not only enhanced my understanding of binary tree traversal but also helped me appreciate the value of diverse problem-solving approaches. The assignment emphasized the importance of clear communication, peer learning, and analytical thinking, all of which are crucial for technical interviews.

Evaluation Criteria

We are looking for the similar points as Assignment 1

- Problem is accurately stated
- New example is correct and easily understandable
- Correctness, time, and space complexity of the coding solution
- Clarity in explaining why the solution works, its time and space complexity
- Quality of critique of your partner's assignment, if necessary

Submission Information

🚨 **Please review our Assignment Submission Guide** 🚨 for detailed instructions on how to format, branch, and submit your work. Following these guidelines is crucial for your submissions to be evaluated correctly.

Submission Parameters:

- Submission Due Date: `HH:MM AM/PM - DD/MM/YYYY`
- The branch name for your repo should be: `assignment-2`
- What to submit for this assignment:
  - This Jupyter Notebook (assignment_2.ipynb) should be populated and should be the only change in your pull request.
- What the pull request link should look like for this assignment: `https://github.com/<your_github_username>/algorithms_and_data_st`
  - Open a private window in your browser. Copy and paste the link to your pull request into the address bar. Make sure you can see your pull request properly. This helps the technical facilitator and learning support staff review your submission easily.

Checklist:

- ☐ Created a branch with the correct naming convention.
- ☐ Ensured that the repository is public.
- ☐ Reviewed the PR description guidelines and adhered to them.
- ☐ Verify that the link is accessible in a private browser window.

If you encounter any difficulties or have questions, please don't hesitate to reach out to our team via our Slack at `#cohort-3-help`. Our Technical Facilitators and Learning Support staff are here to help you navigate any challenges.