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.

Imagine a tree that starts with one main point at the top (called the "root").

From there, it splits into two branches at each step, like a family tree.

The very bottom points (called "leaves") are the ones that don’t split anymore—they’re the endpoints.

The task is to find all the possible paths from the top (root) to each of these bottom points (leaves).

For each path, we write down the sequence of points you pass through from the top to the bottom.

Finally, we collect all these sequences into a list

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

Answers for this part will be commented.

In [1]:
```
# root_1 = [5, 4, 8, 11, 13, 4, 7, 2, 1]

# The tree looks like this:

#       5
#      / \
#     4   8
#    /   / \
#   11  13  4
#  /  \    / \
# 7    2  5   1
# The paths are:

# [[5, 4, 11, 7], [5, 4, 11, 2], [5, 8, 13], [5, 8, 4, 5], [5, 8, 4, 1]]
# The first path is [5, 4, 11, 7], the second is [5, 4, 11, 2], and so on.
```

In [2]:
```
# root_2 = [3, 9, 20, 15, 7]

# The tree looks like this:

#       3
#      / \
#     9  20
#       /  \
#      15   7
# The paths are:

# [[3, 9], [3, 20, 15], [3, 20, 7]]
# The first path is [3, 9], the second is [3, 20, 15], and so on.
```

- Copy the solution your partner wrote.

In [2]:
```
from typing import List
```

In [4]:
```
class TreeNode(object):
    def __init__(self, val = 0, left = None, right = None):
        self.val = val
        self.left = left
        self.right = right

# This will fix the that intput is pointing to the hexadecimal representation of th
    # def __repr__(self):
    #     return f"TreeNode({self.val})"
```

In [5]:
```
def bt_path(root: TreeNode) -> List[List[int]]:
    # List to store all the paths
    paths = []

    # function to perform DFS traversal
```

```
    def dfs(node, current_path):
        if node is None:
            return

        # Add the current node's value to the path
        current_path.append(node.val)

        # If it's a leaf node, store the current path
        if node.left is None and node.right is None:
            paths.append(list(current_path))

        # Recursively visit the left and right child nodes
        dfs(node.left, current_path)
        dfs(node.right, current_path)

        # Backtrack: remove the current node from the path before going back
        current_path.pop()

    # Start DFS traversal from the root node
    dfs(root, [])

    return paths


# Example 1:
# Input:
root = TreeNode(5)
root.left = TreeNode(4)
root.right = TreeNode(8)
root.left.left = TreeNode(11)
root.left.left.left = TreeNode(7)
root.left.left.right = TreeNode(2)
root.right.left = TreeNode(13)
root.right.right = TreeNode(4)
root.right.right.left = TreeNode(5)
root.right.right.right = TreeNode(1)

#         5
#        / \
#       4   8
#      /   /  \
#     11  13   4
#    /  \     /  \
#   7    2   5    1


print(f"Input: {root}")

print(f"Output: {bt_path(root)}")
```

```
Input: <__main__.TreeNode object at 0x000002185E35C440>
Output: [[5, 4, 11, 7], [5, 4, 11, 2], [5, 8, 13], [5, 8, 4, 5], [5, 8, 4, 1]]
```

- Explain why their solution works in your own words.

The solution imports the List module from the typing library and defines a TreeNode class to represent the nodes of a binary tree.

The bt_path function takes the root node of a binary tree as input and returns a list of all root-to-leaf paths in the tree.

This solution employs Depth-First Search (DFS) to find all root-to-leaf paths in the binary tree. The algorithm follows these steps:

Starts at the root and traverses down the tree.

Keeps track of the current path.

When it reaches a leaf node (a node without children), it saves the path.

Uses backtracking to remove the last node before moving to the next path.

Returns all paths from root to leaf in a list.

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

Time Complexity: O(N) because each node is visited once.

Space Complexity: O(H + N), where H is the height of the tree and N is the total number of paths.

The space complexity is dominated by the recursive call stack and the paths list.

The recursive call stack can go as deep as the height of the tree, and the paths list can store all possible paths from root to leaf.

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

Alternative Solution

Instead of DFS, we could use Breadth-First Search (BFS):

Use a queue to traverse level by level.

Start with the root node and an empty path.

When reaching a leaf, store the path.

Continue adding child nodes to the queue with their respective paths.

This method iterates through nodes iteratively instead of recursively.

The time complexity remains O(N), but the space complexity is O(N) due to the queue storing all nodes.

The BFS approach is more suitable for large trees with a high branching factor.

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.

Part 4:

Implementation using Breadth-First Search (BFS) implementation for finding root-to-leaf paths in a binary tree:

In [1]:
```python
from collections import deque
from typing import List

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

    def __repr__(self):
        return f"TreeNode({self.val})"

def bt_path_bfs(root: TreeNode) -> List[List[int]]:
    if not root:
        return []

    paths = []
    queue = deque([(root, [root.val])])  # Queue stores (node, path_so_far)

    while queue:
        node, path = queue.popleft()

        # If it's a leaf node, add the path to the result
        if not node.left and not node.right:
            paths.append(path)

        # If there are children, add them to the queue with updated paths
        if node.left:
            queue.append((node.left, path + [node.left.val]))
        if node.right:
            queue.append((node.right, path + [node.right.val]))

    return paths
```

```
# Example usage
root = TreeNode(5)
root.left = TreeNode(4)
root.right = TreeNode(8)
root.left.left = TreeNode(11)
root.left.left.left = TreeNode(7)
root.left.left.right = TreeNode(2)
root.right.left = TreeNode(13)
root.right.right = TreeNode(4)
root.right.right.left = TreeNode(5)
root.right.right.right = TreeNode(1)

print(f"Output (BFS): {bt_path_bfs(root)}")
```

```
Output (BFS): [[5, 8, 13], [5, 4, 11, 7], [5, 4, 11, 2], [5, 8, 4, 5], [5, 8, 4, 1]]
```

In [3]:

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

    def __repr__(self):
        return f"TreeNode({self.val})"

def bt_path_bfs(root):
    if not root:
        return []

    paths = []
    queue = [(root, [root.val])]  # Using a list as a queue

    while queue:
        node, path = queue.pop(0)  # Remove the first element (O(N) operation)

        # If it's a leaf node, add the path to the result
        if not node.left and not node.right:
            paths.append(path)

        # If there are children, add them to the queue with updated paths
        if node.left:
            queue.append((node.left, path + [node.left.val]))
        if node.right:
            queue.append((node.right, path + [node.right.val]))

    return paths

# Example usage
root = TreeNode(5)
root.left = TreeNode(4)
root.right = TreeNode(8)
root.left.left = TreeNode(11)
root.left.left.left = TreeNode(7)
root.left.left.right = TreeNode(2)
root.right.left = TreeNode(13)
root.right.right = TreeNode(4)
```

```
root.right.right.left = TreeNode(5)
root.right.right.right = TreeNode(1)

print(f"Output (BFS no imports): {bt_path_bfs(root)}")
```

```
Output (BFS no imports): [[5, 8, 13], [5, 4, 11, 7], [5, 4, 11, 2], [5, 8, 4, 5],
[5, 8, 4, 1]]
```

How This Works:

Queue-based traversal: We use a queue (deque) to store nodes along with the path that led to them. Processing nodes level by level: Nodes are processed in the order they are encountered.

Leaf check: If a node has no children, its path is stored in the result list.

Children are enqueued: Left and right children are added to the queue with their respective paths.

Time & Space Complexity:

Time Complexity: O(N) (every node is visited once)

Space Complexity: O(N) (storing all root-to-leaf paths)

This BFS approach provides an iterative alternative to the DFS recursion

Approach using deque (short for double-ended queue) is a data structure in Python provided by the collections module.

It allows fast insertions and deletions from both ends (front and back), making it more efficient than Python's built-in list for queue operations.

Why Use deque Instead of a List for BFS?

In Breadth-First Search (BFS), we frequently:

Enqueue (append) nodes at the back of the queue.

Dequeue (remove) nodes from the front of the queue.

Using a list, removing elements from the front (list.pop(0)) requires O(N) time because it shifts all elements.

Using deque, popleft() takes O(1) time, making it much faster.

How deque Works in BFS

We initialize a queue with deque().

We add nodes to the back using append().

We remove nodes from the front using popleft().

We process each node level by level.

Key Advantages of deque

✅ Faster than list for queue operations (O(1) instead of O(N)).

✅ Supports appending/removing from both ends efficiently.

✅ Ideal for BFS and sliding window problems.

Reflection

This assignment was a valuable learning experience. At first, understanding how to traverse a binary tree was challenging, but breaking the problem into steps made it easier. Using Depth First Search (DFS) felt natural because it allowed me to explore each path step by step before going back. Writing examples helped me think about different cases and tree structures.

The peer review process was just as helpful. Looking at my partner's solution gave me a new perspective and showed different coding styles. Explaining time and space complexity during the review also deepened my understanding of these concepts. Discussing alternative approaches, like Bread First Search (BFS), helped me see another way to explore a tree. While DFS works well for recursion, BFS is useful for looking at each level of the tree before moving to the next.

This experience showed me the value of both writing and reviewing code. It helped me improve problem-solving, communication, and coding skills, which are all important for technical interviews and real-world programming. I feel more confident in analyzing problems and explaining my thinking. I am excited to take on more coding challenges and peer reviews to keep improving. Thank you for this opportunity!

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_structures/`
  - 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.