Topic Introduction

Today's prep letter is all about **Breadth-First Search (BFS) in Binary Trees**. If you've ever had to traverse a tree *by levels*, BFS is your go-to tool. BFS explores a tree one layer (or level) at a time, visiting all the nodes at the current depth before moving to the next one. This is perfect for problems where you want to process nodes by how far they are from the root or need to gather information from each level.

What is BFS?
Breadth-First Search is a graph and tree traversal technique that uses a queue to explore nodes in the order they're encountered. In a binary tree, this means you visit the root, then all its children, then all their children, and so on.

How does it work?
Start with the root node in a queue. Repeatedly take a node from the front, process it, and add its non-null children to the back of the queue. Continue until the queue is empty.

When and why is it useful in interviews?
BFS is especially useful when you need to work with trees level by level, such as finding the shortest path, determining levels, or collecting data in a layered fashion. Interviewers love BFS tree questions because they test your understanding of queues, traversal order, and problem decomposition.

Quick Example (not one of our target problems):
Suppose you want to find the sum of all nodes at each level of a binary tree. BFS lets you process each level in order, summing as you go, and storing the result for each level.

Let's apply this concept to three classic problems:

- **Binary Tree Level Order Traversal**
- **Binary Tree Zigzag Level Order Traversal**
- **Binary Tree Right Side View**

Why are these grouped together?
All three require you to traverse a binary tree by levels using BFS, but each asks for a different output:

- The first returns the nodes at each level in order.
- The second alternates the direction at each level (left-to-right, then right-to-left).
- The third only wants the *rightmost* node at each level (the one you'd see from the side).

Each is a twist on BFS tree traversal, so practicing these will make you a BFS pro!

Problem 1: Binary Tree Level Order Traversal

LeetCode 102: Binary Tree Level Order Traversal

Rephrased Problem Statement:
Given the root of a binary tree, return a list of lists, where each inner list contains the values of the nodes at that level, from top to bottom.

Example:

Input:

```
    3
   / \
  9  20
     / \
    15  7
```

Output:

`[[3], [9, 20], [15, 7]]`

Walkthrough:

- Level 0: [3]
- Level 1: [9, 20]
- Level 2: [15, 7]

So, we return `[[3], [9, 20], [15, 7]]`.

Another test case to try:

Input:

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

Expected Output:

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

How to Solve

Brute-force approach:

You could traverse the tree multiple times, once for each level, using recursion to collect nodes at each depth. This is inefficient, as you might visit the same nodes repeatedly.

- **Time complexity:** O(N^2) in the worst case (for a skewed tree).

Optimal approach: BFS with a Queue

- Use a queue to process nodes level by level.
- At each level, process all nodes currently in the queue (these form one level).
- Append their values to a result list, and add their children to the queue.

Step-by-step logic:

- Create an empty result list and a queue initialized with the root node.
- While the queue is not empty:
    - For each node currently in the queue (i.e., for the current level):
        - Pop it, record its value, add its left and right children (if any) to the queue.

- Add the list of values for this level to the result.

Python Solution:

```
from collections import deque

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

def levelOrder(root):
    if not root:
        return []
    result = []
    queue = deque([root])  # Use deque for efficient pops from the left
    while queue:
        level_size = len(queue)  # Number of nodes at current level
        level = []
        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

Time complexity: O(N), where N is the number of nodes (each node is visited once).

Space complexity: O(N) for the queue and result.

What does each part do?

- `queue = deque([root])`: Initializes a queue with the root node.
- `while queue:`: Keeps looping until all nodes are processed.
- `level_size = len(queue)`: The number of nodes at the current level.
- `for _ in range(level_size):`: Ensures you only process nodes at this level before moving to the next.
- `queue.popleft()`: Removes and returns the node at the front of the queue.
- `level.append(node.val)`: Collects the node's value for the current level.
- `queue.append(node.left/right)`: Adds children to the queue for the next level.
- `result.append(level)`: Adds the collected values for the current level to the result.

Trace of a test case:

Input:

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

- Start: queue = [1]
- Level 0: Pop 1, add [1], queue = [2, 3]
- Level 1: Pop 2 (add 4 to queue), Pop 3 (add 5 to queue), add [2, 3], queue = [4, 5]
- Level 2: Pop 4, Pop 5, add [4, 5], queue = []

Result: `[[1], [2, 3], [4, 5]]`

Try it yourself:

Input:

```
    1
     \
      2
       \
        3
       /
      4
```

Expected output? Dry-run the steps and see!

Take a moment to solve this on your own before looking back at the code.

Problem 2: Binary Tree Zigzag Level Order Traversal

LeetCode 103: Binary Tree Zigzag Level Order Traversal

Rephrased Problem Statement:

Given a binary tree, return its level order traversal as a list of lists, but alternate the direction of traversal at each level. For example, the first level is left-to-right, the next is right-to-left, and so on.

How is this different from Problem 1?

It's the same BFS structure, but at every *other* level, you reverse the order of values collected.

Example:

Input:

```
    1
   / \
  2   3
 / \   \
```

```
4   5   6
```

Output:

```
[[1], [3, 2], [4, 5, 6]]
```

Walkthrough:

- Level 0: [1] (left-to-right)
- Level 1: [2, 3] (right-to-left: [3, 2])
- Level 2: [4, 5, 6] (left-to-right)

Try this test case:

Input:

```
    10
   /  \
  5    15
 /      \
3        20
```

Expected Output:

```
[[10], [15, 5], [3, 20]]
```

How to Solve

Brute-force approach:

Use the regular BFS to collect each level, then at the end, reverse every other list. This is inefficient because you might reverse large lists unnecessarily.

Optimal approach: BFS + Direction Flag

- Use BFS as before, but keep a flag indicating whether you're on a left-to-right or right-to-left level.
- When collecting node values for a level, reverse them if needed, or insert at the beginning of the list for right-to-left.

Step-by-step pseudocode:

```
queue = [root]
result = []
left_to_right = True

while queue is not empty:
    level_size = length of queue
    level = []
    for i in 0..level_size-1:
        node = dequeue from queue
        if left_to_right:
            append node.val to level
        else:
```

```
            insert node.val at start of level
        enqueue node.left and node.right if they exist
    append level to result
    left_to_right = not left_to_right
```

Time complexity: O(N)

Space complexity: O(N)

Example trace:

Input:

```
    1
   / \
  2   3
 / \   \
4   5   6
```

- Start: queue = [1], left_to_right = True
- Level 0: Pop 1, add [1], queue = [2, 3], left_to_right = False
- Level 1: Pop 2, Pop 3, add [3, 2] (right-to-left), queue = [4, 5, 6], left_to_right = True
- Level 2: Pop 4, 5, 6, add [4, 5, 6], queue = []

Result: `[[1], [3, 2], [4, 5, 6]]`

Try this input:

```
    1
   / \
  2   3
   \
    4
```

Expected output? Dry-run and check!

Problem 3: Binary Tree Right Side View

LeetCode 199: Binary Tree Right Side View

Rephrased Problem Statement:

Given the root of a binary tree, return the list of node values that are visible when looking at the tree from the right side (one node per level: the rightmost node at each level).

How is this different?

Instead of collecting all the nodes at a level, you only want the last node processed at each level (i.e., the rightmost node).

Example:

Input:

```
    1
```

```
  / \
 2   3
  \   \
   5   4
```

Output:

`[1, 3, 4]`

How to Solve

Brute-force approach:

You could collect all nodes at each level (like in Problem 1), then at the end, pick the last value from each level's list.

Optimal approach: BFS + Last Node per Level

- Use BFS as before.
- For each level, process all nodes, but after finishing the level, record the last node's value.

Step-by-step pseudocode:

```
queue = [root]
result = []

while queue is not empty:
    level_size = length of queue
    for i in 0..level_size-1:
        node = dequeue from queue
        if node.left: enqueue node.left
        if node.right: enqueue node.right
        if i == level_size - 1:
            append node.val to result
```

Time complexity: O(N)

Space complexity: O(N)

Example trace:

Input:

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

- Level 0: [1] => rightmost: 1
- Level 1: [2, 3] => rightmost: 3
- Level 2: [5, 4] => rightmost: 4

Result: `[1, 3, 4]`

Try this test case:

Input:

```
    1
   /
  2
 /
3
```

Expected output? Walk through the steps!

Reflect:

How else could you solve this? Could you use DFS? What would you need to keep track of?

Summary and Next Steps

Today, you practiced three classic BFS-based binary tree problems:

- **Level Order Traversal:** Collecting all nodes at each level.
- **Zigzag Level Order Traversal:** Alternating direction at each level.
- **Right Side View:** Finding the rightmost node at each level.

Key takeaways:

- Use a queue for BFS to process nodes level by level.
- Track the number of nodes at each level to know when to switch levels.
- For variations, tweak how you collect or process nodes at each level.
- Common traps: forgetting to process only nodes at the current level, not managing the direction for zigzag, or not picking the correct node for the right side view.

Action List:

- Solve all three problems on your own, even the one with code provided.
- Try solving Problem 2 and 3 using DFS (preorder traversal with level tracking).
- Explore other tree problems that use BFS, like finding minimum depth or populating next right pointers.
- Compare your solution code with others, paying attention to clean handling of edge cases.
- Don’t get discouraged if you get stuck! Each problem deepens your understanding.

Happy coding, and may your BFS adventures be ever fruitful!