Diagonal Traversal of Binary Tree

Objective: Traverse a binary tree diagonally and return elements in a flattened 1D vector.

Algorithm Steps:

Step 1: Understand the Diagonal Concept - Diagonal traversal groups nodes by their diagonal level. - Right child stays on the same diagonal level. - Left child moves to the next diagonal level.

Step 2: Use DFS for Traversal - We can use a Depth First Search (DFS) approach. - Pass the current diagonal level as a parameter.

Step 3: Maintain a 2D Vector - Create a 2D vector `res` to store nodes at each diagonal. - Each `res[L]` represents the nodes on diagonal level `L`.

Step 4: Resize 2D Vector Dynamically - Before inserting a node at diagonal level `L`, check if `res` has enough rows. - If `res.size() <= L`, resize it to `L+1`.

Step 5: Push Node Data - Insert the current node's value into `res[L]`.

Step 6: Recur for Children - Recur for the left child with diagonal level `L+1` (next diagonal). - Recur for the right child with diagonal level `L` (same diagonal).

Step 7: Flatten the 2D Vector - After DFS, traverse each row of `res` and append elements into a single 1D vector `ans`.

Step 8: Return the Result - Return the flattened vector `ans` as the final diagonal traversal.

Pseudo Code:

```
function dfs(res, root, L):
    if root is NULL:
        return

    if size of res <= L:
        resize res to have L+1 rows

    append root.data to res[L]

    dfs(res, root.left, L+1)    # left child goes to next diagonal
    dfs(res, root.right, L)     # right child stays on same diagonal

function diagonal(root):
    initialize res as empty 2D vector
    dfs(res, root, 0)            # start from root at diagonal level 0
```

```
    initialize ans as empty 1D vector
    for each row in res:
        for each element in row:
            append element to ans

    return ans
```

Notes: - `res` dynamically grows to accommodate diagonal levels. - Right child stays in the same diagonal, left child moves to next diagonal. - Finally, 2D vector is flattened for output.

Time Complexity: O(n) # Each node visited once. **Space Complexity:** O(n) # For storing nodes in 2D vector and recursion stack.