Documentation on Flatten a Multilevel Doubly Linked List

1. Problem Statement

Given a multilevel doubly linked list, where each node has:

- A next pointer to the next node in the same level.
- A prev pointer to the previous node in the same level.
- A child pointer, which may or may not point to a separate doubly linked list.

The task is to flatten this list into a single-level doubly linked list, ensuring:

- The order is maintained such that the child list appears immediately after its parent.
- All child pointers are set to None.

2. Intuition

The problem requires traversal of a hierarchical structure while maintaining order. The best approach is to use a **Depth-First Search (DFS)** approach with an iterative stack, allowing us to keep track of the nodes we need to return to after handling child nodes.

3. Key Observations

- The problem follows a tree-like structure, where each node can have a sublist (child).
- A recursive approach might lead to excessive stack memory usage.
- An **iterative approach using a stack** is more memory-efficient and avoids recursion depth limits.
- If a node has a child, it should be inserted before its next, ensuring order is preserved.

4. Approach

- **Initialize a stack** to keep track of next nodes when switching to a child node.
- **Traverse the list** using a while loop.
- **When encountering a child node**:
    - Push the next node onto the stack (if it exists).
    - Redirect the next pointer to the child.
    - Update the prev pointer and remove the child pointer.
- **When reaching the end of a level**, pop a node from the stack and continue traversal.
- **Continue until all nodes are processed**.

5. Edge Cases

- **Empty List**: If head is None, return None.
- **Single Node**: If there is only one node with no child, return it as is.
- **Deep Nesting**: The child nodes may contain further child nodes, requiring a recursive-like processing approach.
- **Multiple Child Nodes**: Ensure that all child nodes are inserted in order before returning to the next-level node.

6. Complexity Analysis

Time Complexity:

- **O(N)**, where N is the total number of nodes.
- Each node is visited exactly once, making the traversal linear.

Space Complexity:

- **O(D)**, where D is the maximum depth of child lists.
- At worst, all next pointers get pushed onto the stack if the list has deep nesting.

7. Alternative Approaches

Recursive Approach

- The function could be implemented recursively, flattening child lists first and connecting them afterward.
- However, this approach may lead to stack overflow for very deep nesting.

8. Test Cases

Example 1:

- Input:[1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
- Output:[1,2,3,7,8,11,12,9,10,4,5,6]

Example 2:

- Input:[1,2,null,3]
- Output:[1,3,2]

Example 3:

- Input:[]
- Output:[]

9. Final Thoughts

- The iterative approach using a stack is efficient and memory-friendly.
- It avoids recursion limits while maintaining order.
- The algorithm preserves doubly linked list properties by ensuring prev pointers are correctly assigned.