430. Flatten a Multilevel Doubly Linked List

Medium Depth-First Search Linked List **Doubly-Linked List** Leetcode Link

Problem Description

You are provided with a doubly linked list where each node contains the following pointers:

- `next` pointing to the next node in the same level.
- `prev` pointing to the previous node in the same level.
- `child` potentially pointing to the head of another doubly linked list which represents a lower level.

These `child` pointers can create a multilevel doubly linked list structure. Different nodes at the same level may have child pointers leading to separate doubly linked lists, and this can continue across multiple levels.

The objective is to transform this multilevel doubly linked list into a single-level doubly linked list. To do so, we have to `flatten` the list, ensuring that any node's children (and their subsequent descendants) will be inserted in the list directly after the node and before the node's next sibling.

After the list is flattened, no node should have a child pointer; all child pointers must be set to `null`. The function should return the `head` of the flattened list.

Intuition

The intuition behind the solution lies in traversing the multilevel doubly linked list and restructuring it without using additional space (except for recursion stack). This can be achieved using a depth-first search-like approach. Here's the intuition step by step:

1. Use a depth-first search (DFS) approach that starts from the `head` and explores as far as possible along each branch before backtracking.
2. Create a nested function (like `preorder`) to use for recursion, maintaining the current node and its previous node as parameters to wire the pointers correctly.
3. As you traverse the list, when encountering a child node, recursively process the child list first and then continue with the next nodes, thereby `flattening` the list in place.
4. Sever the link between the current node and its child after flattening the child list, ensuring all child pointers are set to `null`.
5. Utilize the fact that in a doubly linked list, each node has a reference to both its previous and next nodes. Make sure that, after flattening, nodes still maintain proper references to their next and prev nodes.
6. The recursion continues until there are no children nor next nodes to process, at which point the list is fully flattened.
7. Use a dummy node to handle corner cases such as when the head node has a child; this simplifies the process of connecting the nodes.
8. After flattening, disconnect the dummy head from the actual head and return the flattened list without the dummy.

Using this process, we incrementally build the flattened list by connecting sibling nodes and inserting child nodes appropriately, maintaining all the necessary connections both to preceding and following nodes.

Solution Approach

The solution provided uses a recursive approach to flatten the multilevel doubly linked list in place. Here is an explanation of the algorithm, which details how it operates on the doubly linked list structure:

1. **Recursive Preorder Traversal**: The solution approach uses a depth-first search strategy reminiscent of the preorder traversal in tree data structures. A preorder traversal visits a node first, then its children, and then its siblings. The function `preorder(pre, cur)` where `pre` is the previous node, and `cur` is the current node, is designed to leverage this.

2. **Initialization with a Dummy Node**: A dummy node is created with value "`0`," and it precedes the head of the list. This is useful to easily return the new head of the flattened list and to handle corner cases easily, such as when the head node itself has a child.

3. **Recursive Connection of Nodes**: The `preorder` function recursively connects the nodes as follows:
   - If the current node (`cur`) is `None`, the end of a branch has been reached, and the function returns `pre` as the tail.
   - The current node `cur` is connected to the previous node `pre` by updating `cur.prev` and `pre.next`.
   - The recursion saves the next node of `cur` in a temporary variable `t` because the next pointer might change when processing the child nodes.
   - Recursively call `preorder(cur, cur.child)` to process and flatten any child list starting from the current node.
   - The connection to the child node is severed by setting `cur.child` to `None` after it's processed, thus flattening this current section.
   - Recursively call `preorder(tail, t)` using the last processed node (`tail`) from the child list and the saved next node (`t`). This continues flattening the rest of the list after the child list.
4. **Flatten the Whole List**: The initial call `preorder(dummy, head)` starts the recursive processing of the list. The dummy node temporarily holds the place before the head node of the original list.

5. **Finishing Touches**: After the entire list is flattened, the next pointer of the dummy node points to the new head of the flattened list. The previous pointer of the new head node (which is `dummy.next`), however, needs to be set to `None` because the dummy is not part of the actual list.

6. **Return the Flattened List**: Finally, the new head of the flattened list is returned by `dummy.next`, excluding the dummy node from the final result.

Through this recursive process, the list is transformed in place. End-to-end, this means that child lists are integrated between their parent and the subsequent nodes, resulting in a properly flattened list that respects the original order and respects the doubly linked list property that each node should reference its previous and next siblings appropriately.

Example Walkthrough

Consider the following multilevel doubly linked list as an example:

```
1 <-> 2 <-> 3 <-> 4
      |
      5 <-> 6 <-> 7
            |
            8
```

In this example, node 2 has a child doubly linked list starting with node 5, and node 6 also has a child doubly linked list starting with node 8.

Now let's flatten this list using the recursive approach described:

1. Initialize the dummy node as the `prev` to the head of the list. Start flattening with `preorder(dummy, head)` where `head` is node 1. At this step, `dummy.next` will point to node 1.

2. Using the `preorder` function, start the recursion with node 1 (head). Since node 1 has no children, it connects directly to node 2. The function proceeds to node 2.

3. At node 2, a child list starts with node 5. Call `preorder` recursively to process this child list:
   - Node 5 is connected to node 2 before the preorder of the child list is initiated.
   - Node 5's child list (5 → 6 → 7) is processed fully. Node 6 is checked and found to have a child node 8.
   - Recursively process node 8 and connect it after node 6, flattening the child list further. Now the child list is (5 → 6 → 8 → 7), which is connected to node 2.
4. After flattening the list starting at node 5, node 2's child pointer is set to `null`, and we are left with a partial flat list:

```
1 <-> 2 <-> 5 <-> 6 <-> 8 <-> 7
            |           |
           null        null
```

5. The recursion process returns to node 2 and continues with node 3, connecting node 7 to node 3, and so on until all nodes are processed linearly.

6. After recursive calls have been executed for all nodes, we have a completely flattened list:

```
1 <-> 2 <-> 5 <-> 6 <-> 8 <-> 7 <-> 3 <-> 4
```

7. Now, as per the final steps of the algorithm, adjust the linkage so that the dummy node is removed from the list, ensuring the new head's previous pointer (`head.prev`) is set to `None`.

8. Return the new head of the flattened list, which is at first node 1 in our example.

Through these steps, we flatten the multilevel doubly linked list into a single-level doubly linked list in place without the use of additional space (other than the recursion stack).

Python Solution

```python
# Definition for a Node.
class Node:
    def __init__(self, val, prev=None, next=None, child=None):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child


class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        # Helper function to perform a preorder traversal and flatten the list.
        def flatten_list(prev_node, current_node):
            # Base case: if the current node is None, return the previous node.
            if current_node is None:
                return prev_node

            # Connect the current node to the previous node in the flattened list.
            current_node.prev = prev_node
            prev_node.next = current_node

            # Save the next node to continue traversal after the child nodes.
            next_node = current_node.next

            # Recursively flatten the child nodes.
            tail = flatten_list(current_node, current_node.child)
            # After flattening child nodes, the current node's child is set to None.
            current_node.child = None

            # Continue flattening with the saved next node.
            return flatten_list(tail, next_node)

        # If the head is None, there is nothing to flatten.
        if head is None:
            return None

        # Create a dummy node to act as the starting point of the flattened list.
        dummy = Node(0)
        # Call the helper function to start flattening with the head node.
        flatten_list(dummy, head)
        # Disconnect the dummy node from the resulting flattened list and return it.
        dummy.next.prev = None
        return dummy.next
```

Java Solution

```java
class Solution {
    // Method to flatten a multilevel doubly linked list.
    public Node flatten(Node head) {
        // If the list is empty, return null.
        if (head == null) {
            return null;
        }

        Node dummyHead = new Node();  // Dummy head to simplify edge case management.
        dummyHead.next = head;
        flattenList(dummyHead, head); // Flatten the list starting from 'head'.
        dummyHead.next.prev = null;   // Detach the dummy head from the real head of the list.

        return dummyHead.next;        // Return the flattened list without the dummy head.
    }

    // Helper method to recursively flatten the list.
    // It connects the previous node ('prev') with the current node ('curr') and flattens the child lists.
    private Node flattenList(Node prev, Node curr) {
        if (curr == null) {
            // If we reach the end of a list segment, return the last node that was encountered (prev).
            return prev;
        }

        curr.prev = prev;  // Connect the current node to the previous one.
        prev.next = curr;

        Node tempNext = curr.next;  // Temporarily store the next node.
        Node tail = flattenList(curr, curr.child); // Flatten the child list.
        curr.child = null;  // Clear the child pointer since it's now included in the flattened list.

        // Continue flattening from the end of the flattened child list and the next node after 'curr'.
        return flattenList(tail, tempNext);
    }
}
```

C++ Solution

```cpp
// The Node structure definition remains the same with the 'val', 'prev', 'next', and 'child' pointers.
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node* child;
};

class Solution {
public:
    // The 'flatten' function is the entry point to flatten the multilevel doubly linked list.
    Node* flatten(Node* head) {
        // Start the flatten process and ignore the tail returned by helper function.
        flattenAndGetTail(head);
        return head; // Return the modified list with all levels flattened.
    }

    // Helper function which flattens the list and returns the tail node of the flattened list.
    Node* flattenAndGetTail(Node* head) {
        Node* current = head; // Pointer to traverse the list.
        Node* tail = nullptr; // Pointer to keep track of the tail node.

        while (current) {
            Node* nextNode = current->next; // Store the next node of the current node.

            // If the current node has a child, we need to process it.
            if (current->child) {
                Node* childNode = current->child; // The child node which needs to be flattened.
                Node* childTail = flattenAndGetTail(current->child); // Flatten the child list and get its tail.

                current->child = nullptr; // Unlink the child from the current node.
                current->next = childNode; // Make the current node point to the child node.
                childNode->prev = current; // The child node's previous should now point back to the current node.
                childTail->next = nextNode; // Connect the tail of the child list to the next node.

                // If the next node is not null, adjust its previous pointer accordingly.
                if (nextNode)
                    nextNode->prev = childTail;

                tail = childTail; // The new tail is the tail of the flattened child list.
            } else {
                tail = current; // If there's no child, the current node is the new tail.
            }

            current = nextNode; // Move to the next node in the list.
        }

        return tail; // Return the tail node of the flattened list.
    }
};
```

Typescript Solution

```typescript
// Node type definition with 'val', 'prev', 'next', and 'child' properties.
type Node = {
    val: number;
    prev: Node | null;
    next: Node | null;
    child: Node | null;
};

// Entry function to flatten the multilevel doubly linked list.
function flatten(head: Node | null): Node | null {
    // Start the flatten process and ignore the tail returned by the helper function.
    flattenAndGetTail(head);
    return head; // Return the modified list with all levels flattened.
}

// Helper function which flattens the list and returns the tail node of the flattened list.
function flattenAndGetTail(head: Node | null): Node | null {
    let current: Node | null = head; // Pointer to traverse the list.
    let tail: Node | null = null; // Pointer to keep track of the tail node.

    while (current) {
        let nextNode: Node | null = current.next; // Store the next node of the current node.

        // If the current node has a child, we need to process it.
        if (current.child) {
            let childNode: Node = current.child; // The child node which needs to be flattened.
            let childTail: Node | null = flattenAndGetTail(current.child); // Flatten the child list and get its tail.

            current.child = null; // Unlink the child from the current node.
            current.next = childNode; // Make the current node point to the child node.
            childNode.prev = current; // The child node's previous should now point back to the current node.

            if (childTail) {
                childTail.next = nextNode; // Connect the tail of the child list to the next node.
            }

            // If the next node is not null, adjust its previous pointer accordingly.
            if (nextNode) {
                nextNode.prev = childTail;
            }

            tail = childTail; // The new tail is the tail of the flattened child list.
        } else {
            tail = current; // If there's no child, the current node is the new tail.
        }

        current = nextNode; // Move to the next node in the list.
    }

    return tail; // Return the tail node of the flattened list.
}
```

Time and Space Complexity

The given code performs a flattening operation on a doubly linked list with child pointers, which may point to separate doubly linked list nodes, effectively creating a multilevel doubly linked list.

Time Complexity:

The `preorder()` function used in the code traverses each node exactly once. The traversal involves visiting each node's `next` and `child` pointer if present. No node is visited more than once because as we visit a node with a `child`, we nullify the `child` pointer after processing.

Hence, if there are `n` total nodes in the multilevel linked list, each node will be visited only once. This means that the time complexity of the preorder traversal is `O(n)`.

Space Complexity:

The space complexity is determined by the amount of stack space used by the recursive calls. In the worst case, the recursion goes as deep as the maximum depth of the multilevel linked list, which is `O(n)` if every node has a `child` and no `next`. However, in the best case, where the list is already a flat doubly linked list with no `child` pointers, the space complexity is `O(1)` since the recursive function `preorder()` is not called.

The average case would depend on the structure of the multilevel doubly linked list but is typically less than `O(n)` since not all nodes will have a `child`. Regardless, the worst-case space complexity can be stated as `O(n)` since this represents the maximum space that could be required in the presence of a deep nesting structure of child nodes.