1265. Print Immutable Linked List in Reverse

Medium Stack Recursion Linked List Two Pointers

Leetcode Link

Problem Description

In this problem, we are provided with an immutable linked list through an interface named `ImmutableListNode`. We cannot directly access or modify the nodes of the linked list. We are only allowed to use two provided API functions:

- `ImmutableListNode.printValue()`: This method, when called on a node, prints the value of that node.
- `ImmutableListNode.getNext()`: This method, when called on a node, returns the next node in the linked list.

Given the head of the list, our task is to print the values of all the nodes in the linked list in reverse order, adhering strictly to using the given interface without altering the list.

Intuition

The solution to printing the values of nodes in reverse without modifying the list lies in recursion. The recursive approach allows us to visit each node once and "unwind" the calls such that we print the nodes when the recursion stack starts to collapse, effectively printing them in reverse order.

Here's the thinking process behind the recursive solution:

1. Base Case: If the current node (`head`) is `None`, it signifies that we have reached the end of the list, and we don't need to process further.
2. Recursive Case: Assuming the rest of the list can be handled and printed in reverse, we first make a recursive call with the next node `head.getNext()`.
3. Printing Step: Once the recursive call is made, and the recursion starts unwinding, we print the current node's value using the `head.printValue()` method.

We recursively go deeper into the list until we hit the base case. As we return from each recursive call, we're going back from the end of the list towards the front, allowing us to print the values in reverse order by printing the value of the node after the recursive call to the rest of the list.

Solution Approach

The solution provided uses a simple recursive strategy to print the values of an immutable linked list in reverse. Let's delve into the implementation details:

1. **Recursion**: The primary algorithm employed here is recursion, which allows us to "delay" the printing of a node's value until all of its subsequent nodes have been processed (hence printing in reverse).
2. **No Additional Data Structures**: Since the list is immutable, we are not using any extra data structures like stacks or arrays to store values. We are simply leveraging the call stack to hold our place as we recurse through the list.

We start with:

```
if head:
    self.printLinkedListInReverse(head.getNext())
    head.printValue()
```

As seen in the code, there's an `if` condition to check for the base case where `head` is `None`. In linked lists, a `None` node typically signifies the end of the list.

When the `if` condition is true, indicating there are more nodes to explore, we perform two steps:

- **Recursive Call**: We call `self.printLinkedListInReverse(head.getNext())`, which moves one node forward in the list. This line is the crux of the recursive step, causing the function to dive one level deeper until it reaches the end of the list.
- **Print Value**: After the recursive call returns (for each call stack), the subsequent statement `head.printValue()` is executed. Here is where the actual printing happens. The nature of the recursion ensures that this line is called in the reverse order of the node traversal, since the deepest node (the last one) will return from the recursion first, printing its value before its predecessors.

The simplicity of the recursion obviates the need for explicitly handling complex iterative control flows or additional data manipulations, making the code both clean and efficient.

It's important to note that the recursion depth could potentially be a problem if the list is very long; a stack overflow might occur with sufficiently large input. However, for the scope of this exercise, we assume that input sizes won't exceed the limits of the system's call stack.

Example Walkthrough

Imagine we have an immutable linked list with the following values: [1, 2, 3, 4]. We need to print these values in reverse order using the recursive approach detailed above. Let's walk through the steps of the solution using this example:

Step 1: Start at the head of the list which is value 1.

- We call `self.printLinkedListInReverse` on the next node of `1`, which is the node with value `2`.

Step 2: Now our current node is `2`.

- We call `self.printLinkedListInReverse` on the next node which is the node with value `3`.

Step 3: Our current node is now `3`.

- We call `self.printLinkedListInReverse` on the next node which is the node with value `4`.

Step 4: We have reached the final node, `4`.

- We call `self.printLinkedListInReverse` on the next node of `4`, but there is no next node (it's `None`), so we hit our base case and do not make any further recursive calls.

Step 5: As the recursion starts to unwind, we are now back at node `4`.

- We execute `head.printValue()` which prints `4`.

Step 6: The recursion unwinds one step further, and we are back at node `3`.

- We execute `head.printValue()` which prints `3`.

Step 7: The recursion continues to unwind, and we are back at node `2`.

- We execute `head.printValue()` which prints `2`.

Step 8: Finally, the recursion completely unwinds to the first node, `1`.

- We execute `head.printValue()` which prints `1`.

Through this recursion, we've printed the nodes in the reverse order (4, 3, 2, 1) by leveraging the call stack to delay the print operations until we've encountered all nodes. This approach elegantly uses the system stack in place of an explicit stack data structure to reverse the print order.

Python Solution

```python
class Solution:
    def printLinkedListInReverse(self, head: 'ImmutableListNode') -> None:
        """
        Recursively prints the values of a linked list in reverse.

        :param head: The head node of the immutable linked list.
        :return: None
        """
        # Base case check: if head is not None, proceed with recursion
        if head:
            # Recursively call the function for the next node in the list
            self.printLinkedListInReverse(head.getNext())
            # After the recursion unfolds, print the value of the current node
            head.printValue()

```

Java Solution

```java
// This solution is for printing an immutable linked list in reverse.
// The ImmutableListNode's API interface is predefined and must not be implemented here.
class Solution {

    // This method prints the linked list in reverse order by using recursion.
    public void printLinkedListInReverse(ImmutableListNode head) {
        // Base case: if the current node is not null, proceed.
        if (head != null) {
            // Recursive call: move to the next node in the list.
            printLinkedListInReverse(head.getNext());

            // After reaching the end of the list, or the recursive call for the last node returns,
            // print the value of the current node.
            // Due to recursion, this will happen in reverse order,
            // since the last node's value will be printed first.
            head.printValue();
        }
        // When the head is null (which means the starting node was null),
        // or we have reached the beginning of the list), the method will do nothing and return,
        // effectively ending the recursive chain.
    }
}

```

C++ Solution

```cpp
/**
 * // Given is the API interface for an ImmutableListNode.
 * // The implementation of this interface should not be modified or assumed.
 * class ImmutableListNode {
 * public:
 *     void printValue(); // Prints the value of the node.
 *     ImmutableListNode* getNext(); // Returns the next node in the list, if any.
 * };
 */

class Solution {
public:
    // This function prints the values of the linked list in reverse.
    void printLinkedListInReverse(ImmutableListNode* head) {
        // Base case: if the current node is not null, proceed.
        if (head) {
            // Recursive call to process the next node in the list.
            printLinkedListInReverse(head->getNext());

            // After the recursion unwinds (i.e., after reaching the end of the list),
            // print the value of the current node.
            head->printValue();
        }
        // When the 'head' is null, the function does nothing and returns,
        // effectively working as a stopping condition for the recursion.
    }
};

```

Typescript Solution

```typescript
/**
 * Function to print the values of an immutable linked list in reverse order.
 * The function uses recursion to traverse to the end of the list before printing values on the call stack's unwind.
 * @param {ImmutableListNode} head - The head node of the immutable linked list.
 */
function printLinkedListInReverse(head: ImmutableListNode): void {
    // Base case: if the current node is not null, recurse to the next node
    if (head) {
        // recursive call with the next node in the list
        const nextNode: ImmutableListNode = head.getNext();
        printLinkedListInReverse(nextNode);

        // After reaching the end and as the recursion stack unwinds, print the value of the current node
        head.printValue();
    }
    // Implicit return of undefined when the if condition is not met (i.e., when head is null)
}

```

Time and Space Complexity

The given Python code defines a recursive function `printLinkedListInReverse` that prints the values of nodes in an immutable linked list in reverse order. Here's an analysis of its time complexity and space complexity:

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the linked list. This is because the function must visit each node exactly once to print its value, and the recursive calls do not overlap in terms of the number of nodes they process.

Space Complexity

The space complexity of the code is also `O(n)`, due to the recursive nature of the solution. Each recursive call to `printLinkedListInReverse` adds a new frame to the call stack. Since the function recurses to the end of the list before printing and backtracking, there will be `n` recursive calls stacked in the call stack, with `n` being the number of nodes in the list.