1019. Next Greater Node In Linked List

Medium Stack Array Linked List Monotonic Stack

Leetcode Link

Problem Description

In this problem, we are provided with the head of a singly linked list, which contains `n` nodes. Each node in the list contains a value, and we are tasked with finding the value of the *next greater node* for each node in the list. This means for each node in the list, we need to find the first node that appears after it whose value is strictly larger than the value of the current node.

The output of the problem should be an array of integers, where each entry corresponds to the value of the next greater node. Specifically, `answer[i]` would contain the value of the next greater node for the `i-th` node in the list (the list is 1-indexed). If a node does not have a next greater node, we should set `answer[i]` to `0`.

To illustrate with an example, let's say we have a linked list `2 -> 1 -> 5`. The next greater node for the first node (`2`) is `5`, since `5` is the next node with a value greater than `2`. The second node (`1`) has a next greater node (`5`) as well. However, the third node (`5`) doesn't have a next greater node, so its corresponding output would be `0`. Hence, the answer would be `[5, 5, 0]`.

Intuition

To arrive at the solution, we analyze the requirements. We need to find the next greater value for each node, and this gives us a hint that we should look at each node from a reverse perspective. If we traverse from the end of the list to the beginning, we can keep track of the greater values we have seen so far, which can help us determine if there's a greater next node.

The solution uses a "stack" data structure to maintain a collection of the greater values we've encountered as we iterate in reverse. At each node, we examine the stack:

1. We remove from the stack all the elements that are less than or equal to the current node's value, because they will not be the "next greater node" for any of the following elements.
2. If the stack still has elements after this operation, the top of the stack represents the next greater value for the current node.
3. We record this value in the answer array.
4. We then add the current node's value onto the stack to be a potential "next greater node" for the preceding nodes.
5. Finally, we proceed to the next node (which is actually the previous list node, as we are going in reverse).

By performing these steps for each node, we leverage the stack to keep track of potential "next greater nodes" for each element and efficiently compute the result in a single reverse pass through the list.

Solution Approach

The solution involves a technique known as *Monotonic Stack*. The stack is used to keep the elements in a sorted manner so that for any given element, we can quickly find the next greater element.

Let's break down the implementation step-by-step:

1. First, we convert the linked list into an array `nums` to enable easy reverse traversal and to avoid dealing with list node pointers.
   This is done using a simple while loop that iterates through the linked list and appends each value to `nums`.

```
nums = []
while head:
    nums.append(head.val)
    head = head.next
```

2. We initialize a stack `stk` that will store values from the list in a decreasing order. This is crucial for the monotonic stack pattern to work. Additionally, we initialize the answer array `ans` with the same length as `nums` and fill it with zeros.

```
stk = []
ans = [0] * len(nums)
```

3. We iterate through the `nums` list in reverse order using a for loop, starting from the last element down to the first. This enables us to consider the "next" elements before the current element, which makes determining the next greater node possible.

```
for i in range(len(nums) - 1, -1, -1):
    ...
```

4. Inside the loop, while the stack is not empty and the value at the top of the stack is less than or equal to `nums[i]`, we pop elements from the stack. This is done to maintain the monotonic decreasing order of the stack.

```
while stk and stk[-1] <= nums[i]:
    stk.pop()
```

5. After clearing the top of the stack, if the stack is not empty, the new top is guaranteed to be the next greater element for `nums[i]` because of the property of the monotonic stack. We record this value in the `ans` array at the corresponding index.

```
if stk:
    ans[i] = stk[-1]
```

6. Finally, we push `nums[i]` onto the stack. This is because `nums[i]` could be the next greater element for the previous nodes in the original list.

```
stk.append(nums[i])
```

7. After the loop completes, `ans` contains all the next greater values for each node.

The overall complexity of the solution is O(N), where N is the number of nodes in the list because each element is pushed onto and popped from the stack at most once.

Example Walkthrough

Let's illustrate the solution approach using a small example of a linked list: `3 -> 2 -> 1 -> 5`.

1. Convert Linked List to Array:
   - Initialize `nums` as an empty list.
   - Traverse the linked list: `nums = [3, 2, 1, 5]`.
2. Initializations:
   - Initialize the stack `stk` as an empty list and the answer list `ans` as `[0, 0, 0, 0]` (equal in length to `nums`).
3. Iterate in Reverse:
   - Start from the end of `nums`: index `i = 3` (value = `5`).
   - Stack `stk` is empty, so add `nums[3]` to `stk`: `stk = [5]`.
4. Reverse Iteration to `i = 2` (value = `1`):
   - Stack `stk` is `[5]`. Top (5) is greater than `nums[2]` (1), so `ans[2] = 5`.
   - Push `nums[2]` onto the stack: `stk = [5, 1]`.
5. Reverse Iteration to `i = 1` (value = `2`):
   - Stack `stk` is `[5, 1]`. Pop top (1) since it's not greater than `nums[1]` (2).
   - Now `stk = [5]`. Top (5) is greater than `nums[1]` (2), so `ans[1] = 5`.
   - Push `nums[1]` onto the stack: `stk = [5, 2]`.
6. Reverse Iteration to `i = 0` (value = `3`):
   - Stack `stk` is `[5, 2]`. Pop top (2) since it's not greater than `nums[0]` (3).
   - Now `stk = [5]`. Top (5) is greater than `nums[0]` (3), so `ans[0] = 5`.
   - Push `nums[0]` onto the stack: `stk = [5, 3]`.
7. With the iteration complete, the `ans` list is `[5, 5, 5, 0]`, which corresponds to the next greater values for each node in the list: `[3, 2, 1, 5]`.

Hence, for the linked list `3 -> 2 -> 1 -> 5`, the output array indicating the next greater node values is `[5, 5, 5, 0]`.

Python Solution

```
# Class definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def nextLargerNodes(self, head: Optional[ListNode]) -> List[int]:
        # Convert the linked list into an array to work with indices.
        valuesList = []
        while head:
            valuesList.append(head.val)  # Add node's value into the valuesList
            head = head.next             # Move to the next node

        # Stack to keep track of the next larger element
        stack = []

        # Length of the linked list or the valuesList
        n = len(valuesList)

        # Initialize an answer list of size 'n' with default value of 0
        answer = [0] * n

        # Iterate over the values list in reverse
        for i in range(n - 1, -1, -1):
            # Pop elements from the stack if they are smaller or equal
            # to the current element since we are looking for the next larger element
            while stack and stack[-1] <= valuesList[i]:
                stack.pop()

            # If stack is not empty, assign the next larger element to answer
            if stack:
                answer[i] = stack[-1]

            # Push current value onto stack
            stack.append(valuesList[i])

        # Return the filled answer list
        return answer
```

Java Solution

```
class Solution {

    // Method to find the next larger node values for each node in the linked list
    public int[] nextLargerNodes(ListNode head) {
        // Use an ArrayList to store the values of the nodes for easier access
        List<Integer> nodeValues = new ArrayList<>();
        // Traverse the linked list and add values to the list
        while (head != null) {
            nodeValues.add(head.val);
            head = head.next;
        }

        // Use a Deque as a stack to keep track of next larger element we have seen so far
        Deque<Integer> stack = new ArrayDeque<>();
        // Find the size of the linked list
        int listSize = nodeValues.size();
        // Create an array to store the result
        int[] result = new int[listSize];

        // Traverse the list in reverse using the values ArrayList
        for (int i = listSize - 1; i >= 0; --i) {
            // Pop elements from the stack if they are less than or equal
            // to the current node's value, since we are only interested
            // in greater values from now on
            while (!stack.isEmpty() && stack.peek() <= nodeValues.get(i)) {
                stack.pop();
            }
            // If the stack is not empty after popping, the current value at the top
            // of the stack is the next greater value, so we add it to the result
            if (!stack.isEmpty()) {
                result[i] = stack.peek();
            }
            // Push the current node's value onto stack
            stack.push(nodeValues.get(i));
        }
        // Return the populated result array containing next larger values
        return result;
    }
}
```

C++ Solution

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(), next(nullptr) {}
 *     ListNode(int x) : ListNode *next() : val(x), next(next) {}
 * };
 */

class Solution {
public:
    // Function to find the next greater node for each element in a linked list.
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> values; // This vector will hold the values of the nodes in the list

        // Traverse the linked list and populate the values vector.
        while (head != nullptr) {
            values.push_back(head->val);
            head = head->next;
        }

        // Initialize a stack to keep track of the next larger elements.
        stack<int> stack;
        // Determine the size of the values vector.
        int size = values.size();
        // Create a vector to store the answers (next larger elements).
        vector<int> answer(size, 0); // Initialize with zeros.

        // Loop through the values vector in reverse to find next larger elements.
        for (int i = size - 1; i >= 0; --i) {
            // Pop elements from the stack that are smaller or equal to the current element,
            // since we want to find the next larger one.
            while (!stack.empty() && stack.top() <= values[i]) {
                stack.pop();
            }
            // If the stack is not empty, the top element is the next larger element.
            if (!stack.empty()) {
                answer[i] = stack.top();
            }
            // Push the current element onto the stack.
            stack.push(values[i]);
        }

        // Return the vector with the next larger elements.
        return answer;
    }
};
```

Typescript Solution

```
/**
 * Definition for singly-linked list.
 */
interface ListNode {
    val: number
    next: ListNode | null
}

/**
 * Function to find the next larger value for every element of a linked list.
 * @param head - The head of the linked list.
 * @returns An array of next larger values.
 */
function nextLargerNodes(head: ListNode | null): number[] {
    // Initialize an array to hold the list node values.
    const valuesArray: number[] = [];

    // Traverse the linked list and populate the valuesArray with node values.
    while (head !== null) {
        valuesArray.push(head.val);
        head = head.next;
    }

    // Initialize a stack to keep track of the larger values.
    const stack: number[] = [];
    const length = valuesArray.length;
    // Initialize an array to hold the answer.
    const nextLargerValues: number[] = new Array(length).fill(0);

    // Iterate the valuesArray from the end to the beginning.
    for (let i = length - 1; i >= 0; --i) {
        // Pop elements from the stack that are less than or equal to the current value.
        while (stack.length > 0 && stack[stack.length - 1] <= valuesArray[i]) {
            stack.pop();
        }

        // If stack is not empty, the top will be the next larger value.
        nextLargerValues[i] = stack.length > 0 ? stack[stack.length - 1] : 0;

        // Push the current value onto the stack.
        stack.push(valuesArray[i]);
    }

    // Return the array of next larger values.
    return nextLargerValues;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code involves iterating over all the elements of the linked list, followed by a single reverse iteration over the list of node values while maintaining a stack to keep track of the next larger node values. Specifically:

- **Converting the linked list to an array**: We iterate over the linked list once, which has `n` elements, so this portion of the algorithm is $O(n)$.
- **Next Larger Elements using a Stack**: In the worst case, every element is pushed to and popped from the stack once. Because each element is handled twice (once for push and once for pop) in this process, and these are constant-time operations (assuming the stack's `append` and `pop` operations are $O(1)$), the complexity is $O(2n)$, which simplifies to $O(n)$.

Combined, since both operations are sequential and not nested, the overall time complexity is $O(n) + O(n)$ which simplifies to $O(n)$.

Space Complexity

The space complexity of the algorithm arises from the space needed to store the list of values and the stack, in addition to the list used for the output:

- **Array of node values**: The array to store node values has the same size as the number of nodes in the linked list, therefore the space complexity is $O(n)$.
- **Stack**: In the worst case, the stack might contain all `n` elements at the same time if the sequence is monotonically decreasing, which gives us a space complexity of $O(n)$ for the stack.
- **Output list**: An output list of size `n` is used to store the answer, resulting in a space complexity of $O(n)$.

When we sum these up, we get $O(n) + O(n) + O(n)$ which simplifies to $O(n)$ overall space complexity as constants are dropped in big O notation.