2074. Reverse Nodes in Even Length Groups Medium Linked List

Leetcode Link

The problem presents a scenario where we have a singly linked list, a data structure consisting of nodes, each with a value and a

Problem Description

forms the first group, the next two nodes form the second group, the following three nodes form the third group, and so on. These groups' sizes increment by one, creating a naturally incremental grouping pattern across the linked list. However, the issue involves a twist: we're asked to reverse the nodes only in those groups that have an even number of nodes. After executing these reversals, we need to return the new head of the modified linked list. It's crucial to notice that the last group's size

reference to the next node. The nodes in the list must be grouped according to the sequence of natural numbers i.e., the first node

sequence. Intuition

may be smaller than or equal to the penultimate group, implying that the list may end before completing the size pattern of the

2. Reverse the nodes in these even-length groups. First, we note that the even-length groups are every other group, starting with the second one (sizes 2, 4, 6, etc.). To manage this,

we can iterate through the list while keeping track of the current group size and the total nodes covered so far. Once we reach the end of a group, we reverse it if the group size is even, and then proceed to the next group.

The solution requires us to perform two primary operations:

Identify the groups within the linked list that are of even length.

- For reversing a group of given size 'I' when present in the list, we temporarily detach this portion from the list, perform a standard
- linked list reversal, and then reconnect the reversed segment back to the main list.

group, and we apply the even-length condition to decide if a reversal is needed. The provided Python function reverseEvenLengthGroups follows this approach, utilizing a nested reverse helper function that specifically handles the reversal of 'I' nodes in the sublist it is given.

To prevent issues with the final group (which may not conform to the natural number sequence if the list's length doesn't perfectly

align with these group sizes), we check if there are enough nodes left for another group. If not, the remaining nodes form the last

Lastly, to simplify handling edge cases such as the start and end of the list, a dummy node is employed, allowing for easy assignment and reference changes without additional null checks.

This approach requires updating node connections and occasionally counting nodes, both operations being O(1), making the entire algorithm run in O(n) time where n is the number of nodes in the list. Solution Approach

manipulation techniques. To perform the list reversal operation, consider the reverse helper function defined within the Solution class. It takes two arguments: the head of the sublist to be reversed and the length 1 of the sublist. The function follows the classical linked list reversal

1. Initialize three pointers: prev (set to None), cur (pointing to the head of the sublist), and tail (also pointing to the head). These

2. Iterate i from 0 to 1 - 1. During each iteration, reverse the links by pointing the current node's next to its previous node (prev).

pointers help in reversing the links of the sublist while keeping track of its tail.

Then, update prev to the current node and move cur to the next node in the original list.

The implementation employs a two-fold strategy—group detection and conditional reversal—based on simple list traversal and

3. Ensure the reversed sublist is reconnected to the main list by setting the next property of the tail to the current node cur, which now points to the sublist's successor in the main list.

iteration:

The main part of the solution is structured as follows:

the list and increments a counter for each node encountered.

equivalent to skipping over the current group.

Let's illustrate the solution approach with a small example:

Group 2: 2 -> 3 (even-length, should be reversed)

this: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7.

our linked list becomes: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$.

them. The linked list remains: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$.

Suppose we have the following linked list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

Group 1: 1

Increment the group size 1 by 1 for the next iteration.

approach:

edge case handling in linked list problems, particularly when modifications near the head are required. The actual traversal starts from the dummy node, using a pointer prev that keeps track of the end of the last processed group. The main while-loop executes as long as the starting index of the current group is within the bounds of the list. For each group

Then, update previnext with the returned node, which is the new head of the reversed group.

of nodes also needs to be reversed. The same reverse function is called for this part.

According to the pattern described in the problem, the nodes should be grouped like so:

The initial step involves determining the total number of nodes n in the original list. This is done via a while-loop that traverses

A dummy node is created with a value of 0 and its next reference pointing to the head of the list. This is a common trick to simplify

o If 1 is even, indicating an even-length group, call the reverse function with prevenent as the starting node of the group.

o Independently of whether a group is reversed, a nested while-loop advances the prev pointer 1 times forward, which is

Special attention is given for the last group, denoted by left. If left is even and greater than zero, it implies that the last group

number of nodes in the linked list. Example Walkthrough

Since the operations performed for each node are constant time, the overall time complexity of the solution is O(n), where n is the

 Group 3: 4 -> 5 -> 6 Group 4: 7 (End of the list; this would be the part of the next group if we had more nodes) Following the steps outlined:

1. We start by determining the group sizes. The first group has a size of 1, so it remains unchanged. The linked list still looks like

2. Next, we look at the second group which contains nodes 2 and 3. Since it's even-length, we need to reverse it. After the reversal,

4. Finally, we have the incomplete fourth group which would have had four members if the list were longer. Since it only contains 7,

3. We move on to the third group, which contains 4, 5, 6. Since this group has an odd number of nodes (3), we do not reverse

and it's an odd-length group, we leave it as is.

Python Solution

2 class Listnode:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

42

43

44

45

46

47

48

49

50

51

52

53

54

54

55

56

57

58

59

60

61

62

63

64

65

67

68

70

69 }

9 };

10

13

14

15

16

17

19

20

21

22

23

24

25

26

27

28

29

32

33

34

35

36

37

38

39

40

41

43

44

45

46

9

11

12

13

14

15

16

int count = 0;

// Perform reversal for groupSize nodes

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

groupTail.next = current;

// Definition for singly-linked list.

std::vector<int> nodeValues;

ListNode* currentNode = head;

while (currentNode != nullptr) {

ListNode(int x) : val(x), next(nullptr) {}

12 ListNode* reverseEvenLengthGroups(ListNode* head) -

currentNode = currentNode->next;

int totalNodes = nodeValues.size();

if (groupSize % 2 == 0)

for (int value : nodeValues) {

if (currentNode != nullptr) {

// Return the modified linked list head.

// Node values are stored in this array.

let currentNode: ListNode | null = head;

nodeValues.push(currentNode.val);

const totalNodes: number = nodeValues.length;

currentNode = currentNode.next;

let nodeValues: number[] = [];

while (currentNode) {

// Start processing groups.

// Start processing groups.

currentNode = head;

// A vector to store the node values.

// Function to reverse even length groups in a singly linked list.

for (int startIndex = 0, groupSize = 1; startIndex < totalNodes; startIndex += groupSize, groupSize++) {

std::reverse(nodeValues.begin() + startIndex, nodeValues.begin() + startIndex + groupSize);

for (let startIndex = 0, groupSize = 1; startIndex < totalNodes; startIndex += groupSize, groupSize++) {</pre>

// For the last group, adjust the size if it's shorter than expected.

currentNode->val = value; // Set value to the current node.

currentNode = currentNode->next; // Move to the next node.

groupSize = std::min(totalNodes - startIndex, groupSize);

// Check if the current group is of even length.

// Reverse the even length group.

// Reset the current node to the head of the list.

// Update the list nodes with the reordered values.

// Function to reverse even length groups in a singly linked list.

// Traverse the linked list and collect node values.

function reverseEvenLengthGroups(head: ListNode | null): ListNode | null {

// For the last group, adjust the size if it's shorter than expected.

groupSize = Math.min(totalNodes - startIndex, groupSize);

// Traverse the linked list and collect node values.

nodeValues.push_back(currentNode->val);

count++;

return prev;

C++ Solution

1 #include <vector>

#include <algorithm>

ListNode *next;

struct ListNode {

int val;

while (current != null && count < groupSize) {</pre>

// Link the end of the reversed group to the rest of the list

1 -> 3 -> 2 -> 4 -> 5 -> 6 -> 7

1 # Definition for singly-linked list.

count = 0

self.next = next_node

self.val = val

def __init__(self, val=0, next_node=None):

while current and count < length:

Calculate the total number of nodes in the linked list

while (1 + group_size) * group_size // 2 <= node_count and previous_group_end:</pre>

Check if there are leftover nodes in the last group that form an even-length group

previous_group_end.next = reverse(previous_group_end.next, remaining_nodes)

If the current group size is an even number, reverse the group

previous_group_end = previous_group_end.next

remaining_nodes = node_count - group_size * (group_size - 1) // 2

Increment the group size for the next iteration

if remaining_nodes > 0 and remaining_nodes % 2 == 0:

Return the new head of the modified linked list

current.next = previous

temp = current.next

previous = current

temp_head = temp_head.next

Go through the list group by group

current = temp

count += 1

return previous

node_count += 1

group_size += 1

return dummy.next

dummy = ListNode(0, head)

previous_group_end = dummy

node_count = 0

group_size = 1

temp_head = head

while temp_head:

start.next = current

From this example, we can see how the algorithm applies the group size pattern to decide where reversals are required and performs the reversal only on even-length groups using the reverse helper function. The use of pointers like prev, cur, and tail allows for the necessary list manipulations while traversing through the linked list only once, yielding an efficient O(n) time complexity.

Therefore, after applying the steps of the algorithm to the small example list, the modified linked list is:

class Solution: def reverseEvenLengthGroups(self, head: Optional[ListNode]) -> Optional[ListNode]: # Helper function to reverse the linked list of a given length 'length' 9 10 def reverse(start, length): previous, current = None, start 11

if group_size % 2 == 0: 37 previous_group_end.next = reverse(previous_group_end.next, group_size) 38 39 # Move the pointer to the end of the current group 40 for _ in range(group_size): if previous_group_end: 41

```
Java Solution
   class Solution {
       public ListNode reverseEvenLengthGroups(ListNode head) {
            int n = 0;
           // Calculate the total number of nodes in the linked list
            for (ListNode node = head; node != null; node = node.next) {
                n++;
 8
           // Dummy node to simplify reversal operations
 9
           ListNode dummy = new ListNode(0, head);
10
           ListNode groupPrev = dummy;
11
12
13
           // Group size starts from 1
14
           int currentGroupSize = 1;
15
           // Continue grouping and reversing while there are enough nodes remaining
           while (((currentGroupSize + 1) * currentGroupSize) / 2 <= n && groupPrev != null) {</pre>
16
               // Reverse the group if the size is even
17
               if (currentGroupSize % 2 == 0) {
18
19
                    ListNode nextGroupHead = groupPrev.next;
                    groupPrev.next = reverse(nextGroupHead, currentGroupSize);
20
21
22
                // Move the groupPrev pointer to the end of the current group
24
                for (int i = 0; i < currentGroupSize && groupPrev != null; i++) {</pre>
25
                    groupPrev = groupPrev.next;
26
27
28
               // Proceed to the next group
29
                currentGroupSize++;
30
31
32
           // Calculate the number of nodes left for the last group
33
           int nodesLeft = n - currentGroupSize * (currentGroupSize - 1) / 2;
34
           // Perform group reversal if the number of nodes left is even
35
           if (nodesLeft > 0 && nodesLeft % 2 == 0) {
36
                ListNode nextGroupHead = groupPrev.next;
37
                groupPrev.next = reverse(nextGroupHead, nodesLeft);
38
39
40
           // Return the head of the reversed list
           return dummy.next;
41
42
43
44
       /**
        * Reverses a group of nodes in the linked list.
45
46
        * @param head the head of the group to reverse
47
        * @param groupSize the number of nodes to reverse
        * @return new head of the reversed group
48
49
         */
50
       private ListNode reverse(ListNode head, int groupSize) {
51
           ListNode prev = null;
52
           ListNode current = head;
53
           ListNode groupTail = current;
```

47 } 48 Typescript Solution

return head;

```
// Check if the current group is of even length.
19
           if (groupSize % 2 === 0) {
20
               // Extract the subgroup of even length.
               let subGroup: number[] = nodeValues.splice(startIndex, groupSize);
21
               subGroup.reverse(); // Reverse the subgroup.
22
23
               // Insert the reversed subgroup back into the node values array.
24
               nodeValues.splice(startIndex, 0, ...subGroup);
25
26
27
28
       // Reset the current node to the head of the list.
       currentNode = head;
       // Update the list nodes with the reordered values.
       for (let value of nodeValues) {
31
32
           if (currentNode) {
33
               currentNode.val = value; // Set value to the current node.
34
               currentNode = currentNode.next; // Move to the next node.
35
36
37
       // Return the modified linked list head.
38
39
       return head;
40 }
41
Time and Space Complexity
Time Complexity
The given code consists of two main parts: first, finding the total number of nodes in the linked list, and second, reversing even-
length groups.
  1. Counting the nodes in the linked list has a time complexity of O(n), where n is the total number of nodes. This is done with a
    simple while loop iterating through each node exactly once.
 2. The second part involves iterating over the list again and for each group potentially reversing the nodes. Each node is involved in
    at most one reverse operation. The worst-case scenario is when all groups but the last are of even length. In this case, each
```

Space Complexity The space complexity is determined by the amount of extra space used by the algorithm, not counting the input linked list.

Combining these parts, the overall time complexity is O(n + n), which simplifies to O(n).

complexity of O(2n) for the worst case, simplified to O(n).

directly, but on the count of groups.

1. We have a fixed number of variables (prev, cur, t, tail, n, l, dummy, and left) and a fixed-size stack for our recursive call to reverse the list. However, since the reversing is done iteratively within a while loop and not recursively, the space complexity is not directly affected by the size of the input list.

2. The reverse function uses a few temporary variables, but it does not allocate scaling additional space other than function call

node is touched twice: once for the actual reversal and once more when iterating through the groups. This implies a time

3. The arithmetic calculation of (1 + 1) * 1 // 2 is constant time for each group, since it doesn't depend on the list's length

Thus, the space complexity is 0(1) since it does not scale with the size of the input. Overall, the code efficiently processes the list in linear time with constant extra space.

overhead, which is constant for each call since we are not using recursive calls.