1474. Delete N Nodes After M Nodes of a Linked List

Problem Description

Linked List

Easy

through it and alternating between keeping and removing nodes. To be precise, you have to keep the first m nodes, then remove the next n nodes, and continue this pattern until the end of the list. The outcome should be the head of the linked list after these operations have been performed.

In this problem, you are given a singly linked list and two integers m and n. The task is to modify the linked list by iterating

Intuition The intuition behind the solution is to effectively use two pointers to traverse the linked list: one to mark the boundary of the

nodes that will be retained (pre) and the other to find the boundary where the removal will stop (cur). The general approach is:

1. Keep a pointer to the current node we're looking at, starting with the head of the list. 2. Move this pointer m-1 times forward (since we want to keep m nodes, we move m-1 times to stay on the last node that we want to keep). 3. Establish another pointer at the same position and move it n times forward to find the last node that we want to remove.

- 4. Connect the mth node to the node right after the last removed node (effectively skipping over n nodes). 5. Continue this process until we reach the end of the list.
- This algorithm is an in-place transformation which means we modify the original linked list without using extra space for another list.
- Solution Approach

The solution to this problem follows a straightforward iterative approach, using a simple while loop to traverse the linked list, removing unwanted nodes as it goes. Below are the steps involved in the implementation: Initialize a pointer named pre to point to the head of the list. This pointer will be used to track the node after which node

Use a while loop that continues until pre is not None (meaning we have not reached the end of the list).

Example Walkthrough

two nodes, 1 and 2).

Here are the steps in detail:

Skip n nodes: Connect 2 to 6.

•

•

deletion will start.

the count of nodes to be deleted.

Within the while loop, process keeping m nodes intact by iterating m-1 times with a for loop. The -1 is used because we are starting from the node pre which is already considered the first of the m nodes. If during this process pre becomes None,

- we exit the loop because we've reached the end of the list. Now, we need a second pointer called cur which will start at the same position as pre and move n times forward to mark
- cur is None at the end of the for loop, it means we reached the end of the list, and thus, we can safely set pre.next to None. Finally, move pre to pre.next to process the next batch of nodes, beginning the m-n cycle again.

After the for loop to remove n nodes, set pre.next to cur.next, this effectively skips over n nodes that are to be deleted. If

The pattern used is the two-pointer technique, where one pointer (pre) is used to keep track of the node at the border between kept and removed nodes, and the other (cur) used to find the next node that pre should point to after removing n nodes.

A side note is that we have to manage the case when we reach the end of the list correctly. If cur becomes None after the

Let's consider a linked list and use a small example to illustrate the solution approach. Suppose our linked list is 1 -> 2 -> 3 ->

4 -> 5 -> 6 -> 7 -> 8 and the integers m and n are given as m = 2 and n = 3. This means we want to keep 2 nodes and then

The pointer pre starts at the head node 1. We iterate m-1 times forward, which in this case is 1 time (since we are keeping

complexity. The time complexity is O(L) where L is the number of nodes in the linked list, as each node is visited at most once.

This solution employs no additional data structures, effectively making it an in-place operation with 0(1) additional space

removal loop, it means that we do not have any more nodes to process, and we should break out of the loop.

Start with the head of the list. The list is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$.

Now pre is at node 2. Next, we set up cur to point to the same node as pre.

after pre, we do not continue and our final output is the list $1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 8$.

Initialization: pre points to 1, and the list is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$.

remove the next 3 nodes, repeating this process until the end of the list.

Keep m nodes: pre traverses 1 node and still points to 2.

Remove n nodes: cur starts at 2 and traverses 3 nodes, ending up at 5.

The final list, after the in-place modifications, is $1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 8$.

def deleteNodes(self. head: ListNode, m: int, n: int) -> ListNode:

current node = current node.next

We then move cur n times forward. Since n is 3, we move cur to node 5. The list now looks like this, where the brackets indicate the nodes that will remain: [1 -> 2] 3 -> 4 -> 5 -> 6 -> 7 -> 8. • We connect the node at pre (which is node 2) to cur.next (which is node 6). Our list is now 1 -> 2 -> 6 -> 7 -> 8.

Now, we move pre to pre.next which points to node 6. Then we repeat our process. Since there are fewer than m nodes left

Iteration 1:

Since there are fewer than m nodes left after pre, and we cannot remove more nodes, the process stops.

- The list is now [1 → 2] → 6 → 7 → 8. Move pre to pre.next: pre now points to 6.
- **Python** # Definition for singly-linked list. # class ListNode:

class Solution:

Java

class Solution {

Solution Implementation

self.val = val

self.next = next

current_node = head

while current node:

def init (self, val=0, next=None):

Iterate over the entire linked list

more nodes to delete

return head

if current node is None:

to_delete = current_node.next

current_node.next = to_delete

current_node = to_delete

ListNode currentNode = head;

while (currentNode != null) {

if (currentNode == null) {

return head;

Move to the next set of nodes

return head # Return the modified list

public ListNode deleteNodes(ListNode head, int m, int n) {

currentNode = currentNode.next;

ListNode nodeToBeDeleted = currentNode;

currentNode = currentNode.next;

Iteration 2:

Skip m nodes, these nodes will be retained for in range(m - 1): if current node:

If we reach the end of the list, return the head as we don't have

Now current node points to the last node before the deletion begins

Connect the current node to the node following the last deleted node

Skip n nodes to find the last node which needs to be deleted

for in range(n): if to delete: to_delete = to_delete.next

```
/**
* Given a linked list, its head, and two integers m and n,
* this function deletes every n nodes in the list after keeping m nodes.
* @param head The head of the singly-linked list.
* @param m The number of nodes to keep before deletion starts.
             The number of nodes to delete.
* @param n
* @return The head of the modified list.
*/
```

// 'currentNode' is used to traverse the linked list starting from the head.

// Continue loop until 'currentNode' is null, which means end of the list.

// If 'currentNode' is null after the for loop, we return the head

// as we reached the end of the list and cannot delete further nodes.

// 'nodeToBeDeleted' points to the node from where we start deletion.

// Advance 'nodeToBeDeleted' 'n' times or until the end of the list is reached.

// If 'nodeToBeDeleted' is null, we've reached the end and thus set next to null.

// Connect the 'currentNode.next' to the node after the last deleted node.

currentNode.next = (nodeToBeDeleted == null) ? null : nodeToBeDeleted.next;

// Skip 'm' nodes but stop if the end of the list is reached.

for (int i = 0; i < m - 1 && currentNode != null; ++i) {

for (int i = 0; i < n && nodeToBeDeleted != null; ++i) {</pre>

// Move 'currentNode' ahead to process the next chunk of nodes.

nodeToBeDeleted = nodeToBeDeleted.next;

```
// Return the head of the modified list.
        return head;
C++
 * Definition for singly-linked list.
 * struct ListNode {
       int val;
       ListNode *next:
       ListNode(): val(0), next(nullptr) {}
       ListNode(int x) : val(x), next(nullptr) {}
       ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* deleteNodes(ListNode* head, int m, int n) {
        // previous node will point to the last node before the sequence to be deleted
        ListNode* previous_node = head;
        // Continue iterating through the linked list until we reach the end
        while (previous node) {
            // Skip m nodes, but retain the last one before deletion begins
            for (int i = 0; i < m - 1 \&\& previous node; ++i) {
                previous_node = previous_node->next;
            // If we've reached the end, return the head as no deletion is needed
            if (!previous node) {
                return head;
            // Skip n nodes starting from previous_node->next for deletion
            ListNode* current = previous node:
            for (int i = 0; i < n && current; ++i) {</pre>
                current = current->next;
            // Connect the previous node to the node after the n nodes to be deleted
            // If current is not nullptr, link to the node after the deleted sequence
            // If current is nullptr, it means we've reached the end of the list, so we set it to nullptr
            previous_node->next = (current ? current->next : nullptr);
            // Move the previous node forward to start a new sequence
            previous_node = previous_node->next;
        // Return the head of the modified list
        return head;
};
```

* Deletes nodes in a linked list following a pattern: keep 'm' nodes then delete 'n' nodes, repeating this process.

TypeScript

/**

class ListNode {

val: number:

next: ListNode | null;

this.val = val:

while (previousNode) {

if (!previousNode) {

return head;

this.next = next;

// Definition for singly—linked list node

* @param head - The head of the singly-linked list.

* @returns The head of the modified linked list.

let previousNode: ListNode | null = head;

* @param m - The number of nodes to keep.

* @param n - The number of nodes to delete.

constructor(val: number = 0, next: ListNode | null = null) {

function deleteNodes(head: ListNode | null, m: number, n: number): ListNode | null {

// previousNode will point to the last node before the sequence to be deleted

// Continue iterating through the linked list until the end is reached

// Skip 'n' nodes starting from previousNode.next for deletion

for (let i = 0; i < m - 1 && previousNode; <math>i++) {

let currentNode: ListNode | null = previousNode;

for (let i = 0; i < n && currentNode != null; i++) {</pre>

// If the end is reached, no further deletion is needed

previousNode = previousNode.next;

// Skip 'm' nodes, but retain the last one before deletion begins

currentNode = currentNode.next; // Connect previousNode to the node after the 'n' nodes to delete // If currentNode is not null, link to the node after the deleted sequence // If currentNode is null, set the next of previousNode to null (end of list) previousNode.next = currentNode ? currentNode.next : null; // Move previousNode forward to start a new sequence previousNode = previousNode.next; // Return the head of the modified list return head; # Definition for singly-linked list. # class ListNode: def init (self, val=0, next=None): self.val = val self.next = nextclass Solution: def deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode: current_node = head # Iterate over the entire linked list while current node: # Skip m nodes, these nodes will be retained for in range(m - 1): if current node: current node = current node.next # If we reach the end of the list, return the head as we don't have # more nodes to delete if current node is None: return head # Now current node points to the last node before the deletion begins to_delete = current_node.next # Skip n nodes to find the last node which needs to be deleted for in range(n): if to delete: to_delete = to_delete.next # Connect the current node to the node following the last deleted node current_node.next = to_delete # Move to the next set of nodes current_node = to_delete

Time Complexity

return head # Return the modified list

Time and Space Complexity

m and n. The outer while loop goes through each node, but the iteration is controlled by the logic that deletes every n following m nodes. Each node is visited at most once due to the nature of the single pass through the linked list. The for loop running m times and the for loop running n times are sequential and do not multiply their number of operations since they operate on different sets of nodes. Therefore, the time complexity of this function is 0(m + n), for each group of m+n nodes, with a total of O(T/m+n * (m + n)) = O(T) where T is the total number of nodes in the linked list. **Space Complexity**

As there are no additional data structures that grow with the input size, and the given code only uses a fixed number of variables

to keep track of the current and previous nodes, the space complexity is constant. Therefore, the space complexity is 0(1).

The time complexity of the given code involves iterating through each node of the linked list using two nested loops controlled by