2095. Delete the Middle Node of a Linked List

**Problem Description** 

**Linked List** 

Two Pointers

Medium

# You have a singly <u>linked list</u>. Your task is to remove the node that is in the middle of this list and return the head of the updated

list. The middle node is defined as the ⌊ n / 2⌋ <sup>th</sup> node from the beginning of the list, where n is the total number of nodes in the list and ⌊x⌋ signifies the greatest integer less than or equal to x. This means that you're not counting from 1, but from 0. So, in a linked list with: 1 node, the middle is the 0th node.

• 2 nodes, the middle is the 1st node.

- 4 nodes, the middle is the 2nd node. 5 nodes, the middle is the 2nd node. Your goal is to efficiently find and remove this middle node and ensure the integrity of the linked list is maintained after the
- removal. Intuition

3 nodes, the middle is the 1st node.

- To solve this problem, the two-pointer technique is a perfect fit. The idea is to have two pointers, slow and fast. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. By the time the fast pointer reaches the end of the list, the slow pointer will be at the middle node.

## Here's the step-by-step intuition:

list. 2. Start both slow and fast pointers. The slow pointer will start from the dummy node, while fast will start from the head node of the list. 3. Move slow one step and fast two steps until fast reaches the end of the list or has no next node (this is for cases when the number of nodes is even).

4. When the fast pointer reaches the end of the list, the slow pointer will be on the node just before the middle node (since it started from

1. Initialize a dummy node that points to the head. This dummy node will help us easily handle edge cases like when there's only one node in the

dummy, which is before the head). 5. Adjust the next pointer of the slow node so that it skips over the middle node, effectively removing it from the list.

Solution Approach

head of the list.

**Example Walkthrough** 

**Step 2: Two-Pointers** 

Step 3: Traversal

First iteration:

Second iteration:

Before deletion:

**Step 5: Return Updated List** 

The final updated list looks like this:

[ A ] -> [ B ] -> [ D ] -> [ E ]

Solution Implementation

# Definition for singly-linked list.

return dummy\_node.next

// Definition of a singly-linked list node.

ListNode(int value, ListNode next) {

this.value = value;

this.next = next;

ListNode(int value) { this.value = value; }

self.val = val

self.next = next

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

slow\_pointer, fast\_pointer = dummy\_node, head

# Iterate through the list to find the middle

slow pointer = slow pointer.next # Move slow pointer one step

# Return the head of the updated list, by skipping over the dummy node

fast\_pointer = fast\_pointer.next.next # Move fast pointer two steps

# Now, slow pointer is at the node just before the middle one. Delete the middle node

while fast pointer and fast pointer.next:

slow\_pointer.next = slow\_pointer.next.next

slow

slow

- 6. Return the new head of the list, which is pointed to by dummy next, since the dummy node was added before the original head. By utilizing this approach, we can identify and remove the middle node in a singly linked list in a single pass and O(n) time complexity, where n is the number of nodes in the list.
- The solution utilizes a two-pointer approach, which is a common technique for problems involving linked lists or arrays where elements need to be compared or modified based on their positions. Here's a detailed walk-through:

Initialization: A dummy node is created and set to point to the head of the list. The dummy node, not present in the original

list, serves as a starting point for the slow pointer, and helps in case the list has only one node, or if we need to delete the

Traversal: The traversal begins with a while loop that continues until fast is neither null nor pointing to a node with a null

**Deletion**: After the loop, slow is at the node just before the middle node. To delete the middle node, slow next is updated to

point to slow.next.next. This effectively removes the middle node from the list by "skipping" it, as it is no longer referenced

### Two-Pointers: Two pointers are defined: slow starting at the dummy node and fast at the head node. This offset will allow slow to reach the node just before the middle node by the time fast reaches the end.

next pointer. Inside the loop: The slow pointer is moved one node forward with slow = slow.next. The fast pointer is moved two nodes forward with fast = fast.next.next.

- by the previous node. Return Updated List: Finally, the head of the updated list is returned, which is dummy next. This is because dummy is a pseudo-head pointing to the actual head of the list and its next pointer reflects the head of the updated list post-deletion.
- The key data structure used is the singly-linked list, which is manipulated using pointer operations. This solution ensures that the middle node is deleted in a single pass, with a time complexity of O(n), where n is the number of nodes in the list. There is a constant space complexity of O(1), as the number of new variables used does not scale with the input size.
- Let's illustrate the solution approach using a small example. Suppose we have the following linked list: [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]

We want to remove the  $\{lightimes lightimes lightimes flows; lightimes lightimes flows; li$ 

Following the solution approach: Step 1: Initialization • We create a dummy node [ X ] and point it to the head of the list [ A ].

fast (end of list)

### Third iteration (fast.next is null): [X] -> [A] -> [B] -> [C] -> [D] -> [E]

**Step 4: Deletion** Now that fast has reached the end of the list, slow is just before the node we want to delete ([ C ]).

• The list now looks like this: [ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ].

• We start moving both pointers through the list with the loop condition in mind.

[X] -> [A] -> [B] -> [C] -> [D] -> [E]

[X] -> [A] -> [B] -> [C] -> [D] -> [E]

slow

fast

• We set the slow pointer to [ X ] (dummy node) and the fast pointer to [ A ] (head node).

fast

• We perform the deletion by updating the next pointer of the slow node to skip [ C ] and point to [ slow.next.next ].

slow.next

slow slow.next (to be deleted) After deletion: [X] -> [A] -> [B] -----> [D] -> [E]

Node [ C ] has been removed, and the integrity of the list is maintained.

[X] -> [A] -> [B] -> [C] -> [D] -> [E]

slow

We return the head of the updated list, which is dummy.next.

#### def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]: # Create a dummy node that points to the head of the list, to handle edge cases smoothly dummy\_node = ListNode(next=head) # Initialize two pointers, slow will move one step at a time, fast will move two steps at a time

Java

class ListNode {

int value;

ListNode next;

ListNode() {}

class Solution:

class ListNode:

**Python** 

class Solution { public ListNode deleteMiddle(ListNode head) { // Create a dummy node that acts as a predecessor of the head node. ListNode dummy = new ListNode(0, head);

slow = slow.next; // Move slow pointer one step.

ListNode(int value = 0) : value(value), next(nullptr) {}

// Function to delete the middle node of the linked list.

ListNode\* dummyNode = new ListNode(0, head);

// until fast reaches the end of the list.

while (fastPointer && fastPointer->next) {

slowPointer = slowPointer->next;

ListNode\* newHead = dummyNode->next;

\* Deletes the middle node of a singly linked list.

\* If the list is empty or has only one node, returns null.

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

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

slow = slow.next; // Move 'slow' one step.

fast = fast.next.next; // Move 'fast' two steps.

ListNode\* deleteMiddle(ListNode\* head) {

ListNode\* slowPointer = dummyNode;

ListNode\* fastPointer = head;

ListNode(int value, ListNode \*next) : value(value), next(next) {}

// Create a dummy node that points to the head of the list.

// Initialize slow and fast pointers for the runner technique.

// The slow pointer now points at the node before the middle node.

// The head of the new modified list is the next node of dummyNode.

delete dummyNode; // Delete the dummyNode to prevent memory leak

\* @returns The head node of the modified list with the middle node removed.

let slow: ListNode = head; // This pointer will move one step at a time.

def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:

delete toDelete; // Free memory of the node to be deleted

return newHead; // Return the new head of the list

slowPointer->next = slowPointer->next->next; // Remove the middle node

\* If the list has an even number of nodes, it removes the second of the two middle nodes.

// If the list is empty or has only one node, it means there is no middle to delete.

// Move through the list until 'fast' reaches the last or the second to last node.

// 'slow' is now behind the middle node, so set its 'next' to skip the middle node.

let fast: ListNode | null = head.next; // This pointer will move at twice the speed of 'slow'.

# Create a dummy node that points to the head of the list, to handle edge cases smoothly

# Initialize two pointers, slow will move one step at a time, fast will move two steps at a time

these variables does not increase. Therefore, the space complexity of the algorithm is 0(1), indicating that it uses a constant

// Advance the fast pointer by two steps and the slow pointer by one step

fastPointer = fastPointer->next->next; // Move fast pointer by two

ListNode\* toDelete = slowPointer->next: // Store the middle node to delete

// Move slow pointer by one

fast = fast.next.next; // Move fast pointer two steps.

// Skip the middle node. Slow pointer now points to the node before the middle node.

// Initialize two pointers, slow and fast. Slow moves 1 node at a time, while fast moves 2 nodes. ListNode slow = dummy, fast = head; // Iterate through the list with the fast pointer advancing twice as fast as the slow pointer // so that when the fast pointer reaches the end, the slow pointer will be at the middle.

while (fast != null && fast.next != null) {

slow.next = slow.next.next: // Return the modified list. The dummy's next points to the new list's head. return dummy.next; C++

\* Definition for singly-linked list.

\* struct ListNode {

class Solution {

public:

int value;

ListNode \*next;

**}**;

\*/

**TypeScript** 

/\*\*

// Return the modified list with the middle node removed. return head; # Definition for singly-linked list. class ListNode: def init (self, val=0, next=None):

self.val = val

class Solution:

self.next = next

dummy\_node = ListNode(next=head)

slow\_pointer, fast\_pointer = dummy\_node, head

# Iterate through the list to find the middle

while fast pointer and fast pointer.next:

slow.next = slow.next.next;

if (!head || !head.next) {

while (fast.next && fast.next.next) {

return null;

# Now, slow pointer is at the node just before the middle one. Delete the middle node slow\_pointer.next = slow\_pointer.next.next return dummy\_node.next

# Return the head of the updated list, by skipping over the dummy node Time and Space Complexity **Time Complexity** 

slow pointer = slow pointer.next # Move slow pointer one step fast\_pointer = fast\_pointer.next.next # Move fast pointer two steps

The provided code implements an algorithm to delete the middle node of a singly-linked list. The while loop iterates through the list with two pointers: slow moves one step at a time, and fast moves two steps at a time. This loop will continue until fast (or its successor fast next) reaches the end of the list. This means that we traverse the list only once, which leads to a time complexity of O(N), where N is the number of nodes in the singly-linked list.

**Space Complexity** 

The algorithm uses a few constant extra variables: dummy, slow, and fast. Regardless of the size of the list, the space used by

amount of additional space beyond the input list.