My idea:

Idea1: Use vector

Idea2: Use 2 pointers (fast and slow)

World idea:

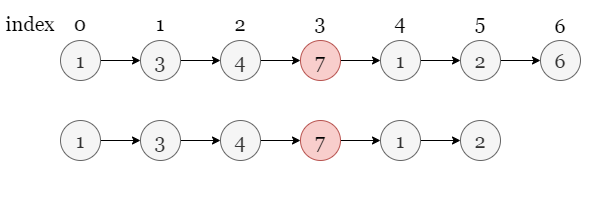
**Solution**

**Overview**

In this problem, we are given a linked list head and the task is to remove the **middle node**.

We are also given the definition of **middle node**. Suppose there are n nodes in the linked list, then the index of the middle node middleIndex is:  
middleIndex=floor(n/2)−1middleIndex = floor(n / 2) - 1*middleIndex*=*floor*(*n*/2)−1, if we use 0-base indexing.

In the figure below, the middle node is colored in red.



We have a [similar problem](https://leetcode.com/problems/middle-of-the-linked-list/) about finding the middle of a linked list, which is a simple version of this problem and you may like to practise on it as well!

**Approach 1: 2 Passes**

**Intuition**

Let's start with a simple approach.

We make the first iteration, starting from head, goint through the entire linked list and getting the total number of nodes (let's say count). According to the definition provided, the index of the middle node is middleIndex=floor(count / 2)−1middleIndex = floor(count\ /\ 2) - 1*middleIndex*=*floor*(*count* / 2)−1.

Now we make a second iteration to the predecessor node of the middle node, which means that we stop at index middleIndex−1middleIndex - 1*middleIndex*−1.

Once we reach the predecessor node of the middle node, we can remove the middle node from the linked list.

Take the slides below as an example:

1 / 11

**Algorithm**

1. If there is only one node, return null.
2. Otherwise, initialize two pointers p1 = head and p2 = head.
3. Iterate the linked list with p1 and count the total number of nodes it has (count).
4. Let p2 move forward by floor(count / 2) - 1 nodes, now it is the predecessor of the middle node, delete the middle node.
5. Return head.

**Implementation**

class Solution {

public:

    ListNode\* deleteMiddle(ListNode\* head) {

        // Edge case: return None if there is only one node

        if (head -> next == nullptr)

            return nullptr;

        int count = 0;

        ListNode \*p1 = head, \*p2 = head;

        // First pass, count the number of nodes in the linked list using 'p1'.

        while (p1 != nullptr) {

            count++;

            p1 = p1 -> next;

        }

        // Get the index of the node to be deleted.

        int middleIndex = count / 2;

        // Second pass, let 'p2' move toward the predecessor of the middle node.

        for (int i = 0; i < middleIndex - 1; ++i)

            p2 = p2 -> next;

        // Delete the middle node and return 'head'.

        p2 -> next = p2 -> next -> next;

        return head;

    }

};

**Complexity Analysis**

Let nn*n* be the length of the input linked list.

* Time complexity: O(n)O(n)*O*(*n*)
  + We iterate over the linked list twice, the first time traversing the entire linked list and the second traversing half of it. Hence there are a total of O(n)O(n)*O*(*n*) steps.
  + In each step, we move a pointer forward by one node, which takes constant time.
  + Remove the middle node takes a constant amount of time.
  + In summary, the overall time complexity is O(n)O(n)*O*(*n*).
* Space complexity: O(1)O(1)*O*(1)
  + We only need two pointers, thus the space complexity is O(1)O(1)*O*(1).

**Approach 2: Fast and Slow Pointers**

**Intuition**

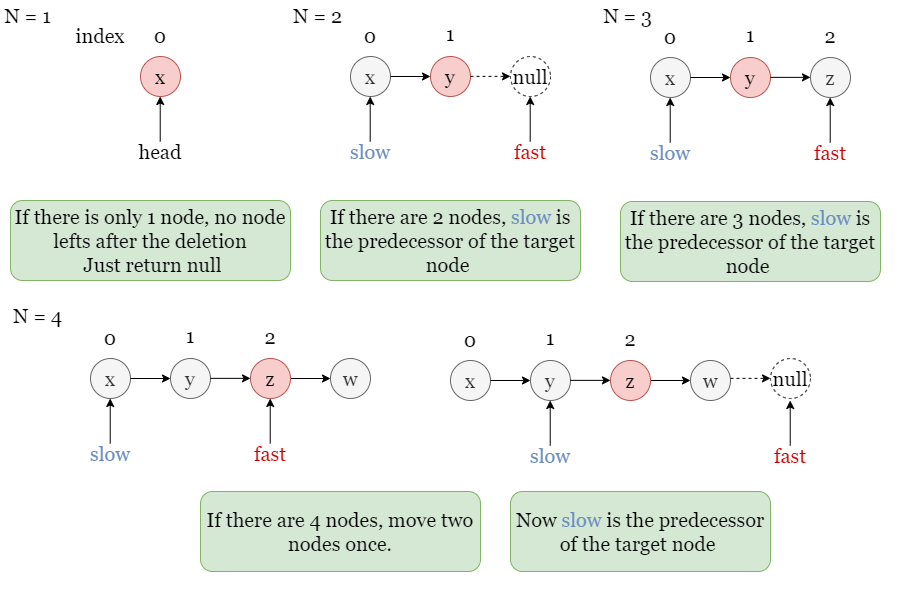
The key of this approach is that we have two pointers fast and slow traversing the linked list at the same time, and fast traverses twice as fast as slow. Therefore, when fast reaches the end of the linked list, slow is right in the middle! We only need one iteration to reach the middle node!

All that needs to be determined are a few lookup details. If there is only one node in the linked list, this node is also the one to be deleted and there are no nodes left after the deletion. Therefore, instead of initializing two pointers for the following procedure, we can just return null.

Why we initialize fast = head.next.next at the begining?

The reason for this is that we want to deleted the middle node instead of finding it. Therefore, we are actually looking for the predecessor of the middle node, not the middle node itself, or rather, this is like moving slow backward one node after the iteration stops.

Certainly, we can't move a pointer backward on a singly linked list, thus we can show this one less step on slow by letting fast moves forward one more step (by two nodes, of course). Hence, slow will also point to the predecessor node of the middle node (rather than the middle node) at the end of the iteration.



Take the slides below as an example:

1 / 7

**Algorithm**

1. If there is only one node, return null.
2. Otherwise, initialize two pointers slow and fast, with slow pointing to head and fast pointing to the second successor node of head.
3. While neither fast and fast.next is null:
   * we move fast forward by 2 nodes.
   * we move slow forward by 1 node.
4. Now slow is the predecessor of the middle node, delete the middle node.
5. Return head.

**Implementation**

class Solution {

public:

    ListNode\* deleteMiddle(ListNode\* head) {

        // Edge case: return nullptr if there is only one node.

        if (head -> next == nullptr)

            return nullptr;

        // Initialize two pointers, 'slow' and 'fast'.

        ListNode \*slow = head, \*fast = head -> next -> next;

        // Let 'fast' move forward by 2 nodes, 'slow' move forward by 1 node each step.

        while (fast != nullptr && fast -> next != nullptr) {

            slow = slow -> next;

            fast = fast -> next -> next;

        }

        // When 'fast' reaches the end, remove the next node of 'slow' and return 'head'.

        slow -> next = slow -> next -> next;

        return head;

    }

};

**Complexity Analysis**

Let nn*n* be the length of the input linked list.

* Time complexity: O(n)O(n)*O*(*n*)
  + We stop the iteration when the pointer fast reaches the end, fast moves forward 2 nodes per step, so there are at most n/2n/2*n*/2 steps.
  + In each step, we move both fast and slow, which takes a constant amount of time.
  + Removing the middle node also takes constant time.
  + In summary, the overall time complexity is O(n)O(n)*O*(*n*).
* Space complexity: O(1)O(1)*O*(1)
  + We only need two pointers, so the space complexity is O(1)O(1)*O*(1).