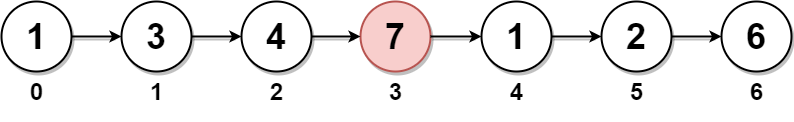
<https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list/description/>

You are given the head of a linked list. Delete the middle node, and return the head of the modified linked list.

The middle node of a linked list of size n is the ⌊n / 2⌋th node from the start using 0-based indexing, where ⌊x⌋ denotes the largest integer less than or equal to x.

For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.

**Example 1:**



**Input:** head = [1,3,4,7,1,2,6]

**Output:** [1,3,4,1,2,6]

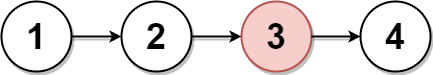
**Explanation:**

The above figure represents the given linked list. The indices of the nodes are written below.

Since n = 7, node 3 with value 7 is the middle node, which is marked in red.

We return the new list after removing this node.

**Example 2:**



**Input:** head = [1,2,3,4]

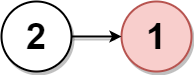
**Output:** [1,2,4]

**Explanation:**

The above figure represents the given linked list.

For n = 4, node 2 with value 3 is the middle node, which is marked in red.

**Example 3:**



**Input:** head = [2,1]

**Output:** [2]

**Explanation:**

The above figure represents the given linked list.

For n = 2, node 1 with value 1 is the middle node, which is marked in red.

Node 0 with value 2 is the only node remaining after removing node 1.

**Constraints:**

The number of nodes in the list is in the range [1, 10^5].

1 <= Node.val <= 10^5

**Attempt 1: 2024-03-28**

**Solution 1: Fast and Slow Pointers (30 min)**

**Style 1: Create dummy node and slow, fast node both start at dummy node**

/\*\*

 \* Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode() {}

 \*     ListNode(int val) { this.val = val; }

 \*     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

 \* }

 \*/

class Solution {

    public ListNode deleteMiddle(ListNode head) {

        ListNode dummy = new ListNode(-1);

        ListNode slow = dummy;

        ListNode fast = dummy;

        dummy.next = head;

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

            slow = slow.next;

            fast = fast.next.next;

        }

        slow.next = slow.next.next;

        return dummy.next;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Step by Step**

**We can test with two major inputs: odd number of nodes list and even number of nodes list**

public class LinkedListSolution {

private class ListNode {

int val;

ListNode next;

ListNode() {}

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

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

}

public ListNode deleteMiddle(ListNode head) {

ListNode dummy = new ListNode(-1), slow = dummy, fast = dummy;

dummy.next = head;

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

slow = slow.next;

fast = fast.next.next;

}

slow.next = slow.next.next;

return dummy.next;

}

public static void main(String[] args) {

LinkedListSolution l = new LinkedListSolution();

ListNode node1 = l.new ListNode(1);

ListNode node2 = l.new ListNode(2);

ListNode node3 = l.new ListNode(3);

ListNode node4 = l.new ListNode(4);

//ListNode node5 = l.new ListNode(5); -> switch to odd number of list

node1.next = node2;

node2.next = node3;

node3.next = node4;

//node4.next = node5;

ListNode result = l.deleteMiddle(node1);

System.out.println(result);

}

}

**Refer to**

**[Java/Python3] Slow and fast pointers w/ brief explanation and analysis.**

<https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list/solutions/1611982/java-python3-slow-and-fast-pointers-w-brief-explanation-and-analysis/>

**Q & A:**

Q1: Why do you use a prev and dummy variable for this as it's taking extra space? The following code is good enough without them:

public ListNode deleteMiddle(ListNode head) {

if (head == null || head.next == null) return null;

ListNode slow = head, fast = head.next.next;

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

fast = fast.next.next;

slow = slow.next;

}

slow.next = slow.next.next;

return head;

}

A1: For this specific problem we are given a constraint **The number of nodes in the list is at least 1**. Therefore the above code is perfect without using prev or dummy.

My code is implemented for more general cases including empty list, in which case a dummy head is necessary.

Q2: Most of the solutions just skip the node to be deleted.

To properly delete the node :

ListNode y = slow.next.next; //delete\_node\_after\_slow\_node

slow.next.next = null;

slow.next = y;

A2: Excellent critical comment. Actually there is an ambiguous point about the delete: after deleting from the list, terminate the life of the object or not?

You use an explicit way to terminate the object, and most others just overwrite the corresponding reference, and GC will take care of the object if there is no reference to it, otherwise the object will be still alive out of the list.

**End of Q & A**

**Method 1: Two pointers**

Create a dummy head, and initialize slow and fast pointers as dummy;

Traverse the ListNodes starting from dummy by the afore-mentioned two pointers, slow forwards 1 step and fast forwards 2 steps per iteration;

Terminate the traversal till fast.next or fast.next.next is null, and now slow points to the previous node of the middle node; remove the middle node;

Return dummy.next as the result.

public ListNode deleteMiddle(ListNode head) {

ListNode dummy = new ListNode(-1), slow = dummy, fast = dummy;

dummy.next = head;

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

slow = slow.next;

fast = fast.next.next;

}

slow.next = slow.next.next;

return dummy.next;

}

**Method2: Three pointers** - inspired by **@Zudas**.

Add a prev pointers following slow one.

public ListNode deleteMiddle(ListNode head) {

ListNode dummy = new ListNode(-1), prev = dummy, slow = head, fast = head;

prev.next = head;

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

prev = slow;

slow = slow.next;

fast = fast.next.next;

}

prev.next = slow.next;

return dummy.next;

}

**Analysis:**

Time: O(n), space: O(1), where n = # of the nodes.

**Style 2: Create dummy node and slow node start at dummy node, fast node start at head node**

/\*\*

 \* Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode() {}

 \*     ListNode(int val) { this.val = val; }

 \*     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

 \* }

 \*/

class Solution {

    public ListNode deleteMiddle(ListNode head) {

        ListNode dummy = new ListNode(-1);

        ListNode slow = dummy;

        ListNode fast = head;

        dummy.next = head;

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

            slow = slow.next;

            fast = fast.next.next;

        }

        slow.next = slow.next.next;

        return dummy.next;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/2095>

**Problem Description**

You have a singly linked list. 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.

3 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**

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:

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 list.

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.

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).

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 dummy, which is before the head).

Adjust the next pointer of the slow node so that it skips over the middle node, effectively removing it from the list.

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.

**Solution Approach**

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 head of the list.

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.

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

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 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.

**Example Walkthrough**

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 ⌊5 / 2⌋ = 2nd node from the beginning, which in this case is node [ C ].

Following the solution approach:

**Step 1: Initialization**

We create a dummy node [ X ] and point it to the head of the list [ A ].

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

**Step 2: Two-Pointers**

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

**Step 3: Traversal**

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

First iteration:

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

↑ ↑

slow fast

Second iteration:

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

↑ ↑

slow fast

Third iteration (fast.next is null):

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

↑ ↑

slow fast (end of list)

**Step 4: Deletion**

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

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

Before deletion:

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

↑ ↑

slow slow.next (to be deleted)

After deletion:

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

↑ ↑

slow slow.next

**Step 5: Return Updated List**

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

The final updated list looks like this:

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

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

**Java Solution**

// Definition of a singly-linked list node.

class ListNode {

    int value;

    ListNode next;

    ListNode() {}

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

    ListNode(int value, ListNode next) {

        this.value = value;

        this.next = next;

    }

}

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);

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

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

        }

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

        slow.next = slow.next.next;

        // Return the modified list. The dummy's next points to the new list's head.

        return dummy.next;

    }

}

**Time and Space Complexity**

**Time Complexity**

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 these variables does not increase. Therefore, the space complexity of the algorithm is O(1), indicating that it uses a constant amount of additional space beyond the input list.

**Refer to**

[L203.Remove Linked List Elements](note://636C112756564A3985BB8B72FF6310AB)