Q.Given a singly linked list, delete **middle** of the linked list. For example, if given linked list is 1->2->**3**->4->5 then linked list should be modified to 1->2->4->5.If there are **even** nodes, then there would be **two middle** nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.If the input linked list is NULL or has 1 node, then it should return NULL

**Example 1:**

Input:

LinkedList: 1->2->3->4->5

Output:1 2 4 5

Input:

LinkedList: 2->4->6->7->5->1

Output:2 4 6 5 1

Ans: To solve this problem, we can use the two-pointer technique. We'll have two pointers, a "slow" pointer and a "fast" pointer. The slow pointer moves one node at a time, while the fast pointer moves two nodes at a time. By the time the fast pointer reaches the end of the list, the slow pointer will be pointing to the middle node(s).

Here's the step-by-step algorithm to delete the middle node(s) of a singly linked list:

1. If the linked list is empty or has only one node, return NULL as there is no middle node to delete.

2. Initialize two pointers, slow and fast, to the head of the linked list.

3. Traverse the linked list using the fast pointer, moving two nodes at a time, and the slow pointer, moving one node at a time. Keep track of the previous node of the slow pointer.

4. When the fast pointer reaches the end of the list, the slow pointer will be pointing to the middle node(s).

5. Delete the middle node(s) by adjusting the previous node's next pointer to skip the middle node(s).

6. Return the modified linked list.

Here's the implementation in Python:

```python

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def delete\_middle\_node(head):

if head is None or head.next is None:

return None

slow = head

fast = head

prev = None

while fast is not None and fast.next is not None:

fast = fast.next.next

prev = slow

slow = slow.next

prev.next = slow.next # Skip the middle node(s)

return head

# Example 1:

# Input: 1->2->3->4->5

# Output: 1->2->4->5

head1 = ListNode(1)

head1.next = ListNode(2)

head1.next.next = ListNode(3)

head1.next.next.next = ListNode(4)

head1.next.next.next.next = ListNode(5)

new\_head1 = delete\_middle\_node(head1)

# Print the modified linked list

node = new\_head1

while node is not None:

print(node.val, end=" ")

node = node.next

# Output: 1 2 4 5

print()

# Example 2:

# Input: 2->4->6->7->5->1

# Output: 2->4->6->5->1

head2 = ListNode(2)

head2.next = ListNode(4)

head2.next.next = ListNode(6)

head2.next.next.next = ListNode(7)

head2.next.next.next.next = ListNode(5)

head2.next.next.next.next.next = ListNode(1)

new\_head2 = delete\_middle\_node(head2)

# Print the modified linked list

node = new\_head2

while node is not None:

print(node.val, end=" ")

node = node.next

# Output: 2 4 6 5 1

```

The above implementation will give you the desired output for the given examples.

Q. Given a linked list of **N** nodes. The task is to check if the linked list has a loop. Linked list can contain self loop.

Example 1:

Input:

N = 3

value[] = {1,3,4}

x(position at which tail is connected) = 2

Output:True

Explanation:In above test case N = 3.

The linked list with nodes N = 3 is

given. Then value of x=2 is given which

means last node is connected with xth

node of linked list. Therefore, there

exists a loop.

Example 2:

Input:

N = 4

value[] = {1,8,3,4}

x = 0

Output:False

Explanation:For N = 4 ,x = 0 means

then lastNode->next = NULL, then

the Linked list does not contains

any loop.

Ans: To check if a linked list has a loop, we can use the Floyd's Cycle-Finding Algorithm, also known as the Tortoise and Hare Algorithm. This algorithm involves using two pointers, a slow pointer and a fast pointer, to traverse the linked list.

Here's the step-by-step algorithm to check if a linked list has a loop:

1. Initialize two pointers, slow and fast, to the head of the linked list.

2. Move the slow pointer one step at a time and the fast pointer two steps at a time.

3. If the fast pointer reaches the end of the list (i.e., it becomes NULL), there is no loop in the linked list. Return False.

4. If the slow pointer and fast pointer meet (i.e., they point to the same node), there is a loop in the linked list. Return True.

Here's the implementation in Python:

```python

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def has\_loop(head):

slow = head

fast = head

while fast is not None and fast.next is not None:

slow = slow.next

fast = fast.next.next

if slow == fast:

return True

return False

# Example 1:

# Input: N = 3, value[] = {1, 3, 4}, x = 2

# Output: True

head1 = ListNode(1)

head1.next = ListNode(3)

head1.next.next = ListNode(4)

head1.next.next.next = head1.next # Creating a loop

result1 = has\_loop(head1)

print(result1)

# Output: True

# Example 2:

# Input: N = 4, value[] = {1, 8, 3, 4}, x = 0

# Output: False

head2 = ListNode(1)

head2.next = ListNode(8)

head2.next.next = ListNode(3)

head2.next.next.next = ListNode(4)

result2 = has\_loop(head2)

print(result2)

# Output: False

```

The above implementation uses the Floyd's Cycle-Finding Algorithm to detect a loop in the linked list. It returns True if a loop is found and False otherwise.

Q. Given a linked list consisting of **L** nodes and given a number **N**. The task is to find the **N**th node from the end of the linked list.

**Example 1:**

Input:

N = 2

LinkedList: 1->2->3->4->5->6->7->8->9

Output:8

Explanation:In the first example, there

are 9 nodes in linked list and we need

to find 2nd node from end. 2nd node

from end is 8.

**Example 2:**

**Input:**

**N = 5**

**LinkedList: 10->5->100->5**

**Output:-1**

**Explanation:In the second example, there**

**are 4 nodes in the linked list and we**

**need to find 5th from the end. Since 'n'**

**is more than the number of nodes in the**

**linked list, the output is -1.**

**Ans: To find the Nth node from the end of a linked list, we can use the two-pointer technique. This approach involves using two pointers, a main pointer and a reference pointer, with a distance of N nodes between them. When the main pointer reaches the end of the list, the reference pointer will be pointing to the Nth node from the end.**

**Here's the step-by-step algorithm to find the Nth node from the end of a linked list:**

**1. Initialize two pointers, mainPtr and refPtr, to the head of the linked list.**

**2. Move the refPtr N nodes ahead by traversing the linked list N times.**

**3. If the refPtr becomes NULL before reaching N nodes, it means the value of N is greater than the number of nodes in the linked list. Return -1 in this case.**

**4. Move both mainPtr and refPtr one node at a time until the refPtr reaches the end of the list.**

**5. At this point, the mainPtr will be pointing to the Nth node from the end of the list. Return the value of this node.**

**Here's the implementation in Python:**

**```python**

**class ListNode:**

**def \_\_init\_\_(self, val=0, next=None):**

**self.val = val**

**self.next = next**

**def find\_nth\_node\_from\_end(head, N):**

**mainPtr = head**

**refPtr = head**

**# Move the refPtr N nodes ahead**

**for \_ in range(N):**

**if refPtr is None:**

**return -1 # N is greater than the number of nodes in the list**

**refPtr = refPtr.next**

**# Move both mainPtr and refPtr until refPtr reaches the end**

**while refPtr is not None:**

**mainPtr = mainPtr.next**

**refPtr = refPtr.next**

**return mainPtr.val**

**# Example 1:**

**# Input: N = 2, LinkedList: 1->2->3->4->5->6->7->8->9**

**# Output: 8**

**head1 = ListNode(1)**

**head1.next = ListNode(2)**

**head1.next.next = ListNode(3)**

**head1.next.next.next = ListNode(4)**

**head1.next.next.next.next = ListNode(5)**

**head1.next.next.next.next.next = ListNode(6)**

**head1.next.next.next.next.next.next = ListNode(7)**

**head1.next.next.next.next.next.next.next = ListNode(8)**

**head1.next.next.next.next.next.next.next.next = ListNode(9)**

**result1 = find\_nth\_node\_from\_end(head1, 2)**

**print(result1)**

**# Output: 8**

**# Example 2:**

**# Input: N = 5, LinkedList: 10->5->100->5**

**# Output: -1**

**head2 = ListNode(10)**

**head2.next = ListNode(5)**

**head2.next.next = ListNode(100)**

**head2.next.next.next = ListNode(5)**

**result2 = find\_nth\_node\_from\_end(head2, 5)**

**print(result2)**

**# Output: -1**

**```**

**The above implementation finds the Nth node from the end of the linked list using the two-pointer technique. It returns the value of the Nth node if it exists, and -1 if N is greater than the number of nodes in the list.**

**Q.** Given a linked list of **N** nodes such that it may contain a loop.

A loop here means that the last node of the link list is connected to the node at position X(1-based index). If the link list does not have any loop, X=0.

Remove the loop from the linked list, if it is present, i.e. unlink the last node which is forming the loop.

**Input:**

**N = 3**

**value[] = {1,3,4}**

**X = 2**

**Output:1**

**Explanation:The link list looks like**

**1 -> 3 -> 4**

**^ |**

**|\_\_\_\_|**

**A loop is present. If you remove it**

**successfully, the answer will be 1.**

**Input:**

**N = 4**

**value[] = {1,8,3,4}**

**X = 0**

**Output:1**

**Explanation:The Linked list does not**

**contains any loop.**

**Input:**

**N = 4**

**value[] = {1,2,3,4}**

**X = 1**

**Output:1**

**Explanation:The link list looks like**

**1 -> 2 -> 3 -> 4**

**^ |**

**|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|**

**A loop is present.**

**If you remove it successfully,**

**the answer will be 1.**

**Ans: To remove a loop from a linked list, we can use the Floyd's Cycle-Finding Algorithm to detect the loop and then break the loop by adjusting the next pointer of the last node in the loop.**

**Here's the step-by-step algorithm to remove a loop from a linked list:**

**1. Use the Floyd's Cycle-Finding Algorithm to detect a loop in the linked list. Initialize two pointers, slow and fast, to the head of the linked list.**

**2. Move the slow pointer one step at a time and the fast pointer two steps at a time.**

**3. If the fast pointer reaches the end of the list (i.e., it becomes NULL), there is no loop in the linked list. Return the head of the linked list.**

**4. If the slow pointer and fast pointer meet (i.e., they point to the same node), there is a loop in the linked list. Break the loop by adjusting the next pointer of the last node in the loop.**

**4.1. Reset either the slow pointer or the fast pointer to the head of the linked list.**

**4.2. Move both pointers one step at a time until they meet again. The meeting point will be the last node in the loop.**

**4.3. Set the next pointer of the last node in the loop to NULL, breaking the loop.**

**5. Return the head of the modified linked list.**

**Here's the implementation in Python:**

**```python**

**class ListNode:**

**def \_\_init\_\_(self, val=0, next=None):**

**self.val = val**

**self.next = next**

**def detect\_and\_remove\_loop(head):**

**slow = head**

**fast = head**

**# Detect the loop using Floyd's Cycle-Finding Algorithm**

**while fast is not None and fast.next is not None:**

**slow = slow.next**

**fast = fast.next.next**

**if slow == fast:**

**break**

**# If no loop is present, return the head of the list**

**if fast is None or fast.next is None:**

**return head**

**# Reset either the slow pointer or the fast pointer to the head of the list**

**slow = head**

**# Move both pointers one step at a time until they meet again**

**while slow.next != fast.next:**

**slow = slow.next**

**fast = fast.next**

**# Set the next pointer of the last node in the loop to NULL, breaking the loop**

**fast.next = None**

**return head**

**# Example 1:**

**# Input: N = 3, value[] = {1, 3, 4}, X = 2**

**# Output: 1**

**head1 = ListNode(1)**

**head1.next = ListNode(3)**

**head1.next.next = ListNode(4)**

**head1.next.next.next = head1.next # Creating a loop**

**new\_head1 = detect\_and\_remove\_loop(head1)**

**# Print the modified linked list**

**node = new\_head1**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1**

**# Example 2:**

**# Input: N = 4, value[] = {1, 8, 3, 4}, X = 0**

**# Output: 1**

**head2 = ListNode(1)**

**head2.next = ListNode(8)**

**head2.next.next = ListNode(3)**

**head2.next.next.next = ListNode(4)**

**new\_head2 = detect\_and\_remove\_loop(head2)**

**# Print the modified linked list**

**node = new\_head2**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1**

**# Example 3:**

**# Input: N = 4, value[] = {1, 2, 3, 4}, X = 1**

**# Output: 1**

**head3 = ListNode(1)**

**head3.next = ListNode(2)**

**head3.next.next = ListNode(3)**

**head3.next.next.next = ListNode(4)**

**head3.next.next.next.next = head3 # Creating a loop**

**new\_head3 = detect\_and\_remove\_loop(head3)**

**# Print the modified linked list**

**node = new\_head3**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1**

**```**

**The above implementation uses the Floyd's Cycle-Finding Algorithm to detect and remove a loop from a linked list. It returns the head of the modified linked list without the loop.**

**Q.** Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

**Input:**

**M = 2, N = 2**

**Linked List: 1->2->3->4->5->6->7->8**

**Output:**

**Linked List: 1->2->5->6**

**Input:**

**M = 3, N = 2**

**Linked List: 1->2->3->4->5->6->7->8->9->10**

**Output:**

**Linked List: 1->2->3->6->7->8**

**Input:**

**M = 1, N = 1**

**Linked List: 1->2->3->4->5->6->7->8->9->10**

**Output:**

**Linked List: 1->3->5->7->9**

**Ans: To solve this problem, we can iterate through the linked list while maintaining two pointers: current and previous. We will traverse M nodes and then delete the next N nodes by adjusting the next pointers accordingly. We repeat this process until we reach the end of the linked list.**

**Here's the step-by-step algorithm to retain M nodes and delete N nodes in a linked list:**

**1. Initialize two pointers, current and previous, to the head of the linked list.**

**2. Traverse the linked list while the current pointer is not None.**

**3. Move the current pointer M nodes ahead. If the current pointer becomes None, we have reached the end of the list. Return the head of the modified linked list.**

**4. Move the previous pointer N nodes ahead from the current position. If the previous pointer becomes None, we have reached the end of the list. Return the head of the modified linked list.**

**5. Adjust the next pointer of the previous node to skip the N nodes after the M nodes.**

**6. Set the current pointer to the next node after the N nodes.**

**7. Repeat steps 3-6 until we reach the end of the linked list.**

**8. Return the head of the modified linked list.**

**Here's the implementation in Python:**

**```python**

**class ListNode:**

**def \_\_init\_\_(self, val=0, next=None):**

**self.val = val**

**self.next = next**

**def retain\_delete(head, M, N):**

**current = head**

**previous = None**

**while current:**

**# Traverse M nodes**

**for \_ in range(M):**

**if current is None:**

**return head**

**previous = current**

**current = current.next**

**# Delete N nodes**

**for \_ in range(N):**

**if current is None:**

**return head**

**current = current.next**

**# Adjust next pointer of previous node to skip N nodes**

**previous.next = current**

**return head**

**# Example 1:**

**# Input: M = 2, N = 2, Linked List: 1->2->3->4->5->6->7->8**

**# Output: Linked List: 1->2->5->6**

**head1 = ListNode(1)**

**head1.next = ListNode(2)**

**head1.next.next = ListNode(3)**

**head1.next.next.next = ListNode(4)**

**head1.next.next.next.next = ListNode(5)**

**head1.next.next.next.next.next = ListNode(6)**

**head1.next.next.next.next.next.next = ListNode(7)**

**head1.next.next.next.next.next.next.next = ListNode(8)**

**new\_head1 = retain\_delete(head1, 2, 2)**

**# Print the modified linked list**

**node = new\_head1**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1->2->5->6**

**print()**

**# Example 2:**

**# Input: M = 3, N = 2, Linked List: 1->2->3->4->5->6->7->8->9->10**

**# Output: Linked List: 1->2->3->6->7->8**

**head2 = ListNode(1)**

**head2.next = ListNode(2)**

**head2.next.next = ListNode(3)**

**head2.next.next.next = ListNode(4)**

**head2.next.next.next.next = ListNode(5)**

**head2.next.next.next.next.next = ListNode(6)**

**head2.next.next.next.next.next.next = ListNode(7)**

**head2.next.next.next.next.next.next.next = ListNode(8)**

**head2.next.next.next.next.next.next.next.next = ListNode(9)**

**head2.next.next.next.next.next.next.next.next.next = ListNode(10)**

**new\_head2 = retain\_delete(head2, 3, 2)**

**# Print the modified linked list**

**node = new\_head2**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1->2->3->6->7->8**

**print()**

**# Example 3:**

**# Input: M = 1, N = 1, Linked List: 1->2->3->4->5->6->7->8->9->10**

**# Output: Linked List: 1->3->5->7->9**

**head3 = ListNode(1)**

**head3.next = ListNode(2)**

**head3.next.next = ListNode(3)**

**head3.next.next.next = ListNode(4)**

**head3.next.next.next.next = ListNode(5)**

**head3.next.next.next.next.next = ListNode(6)**

**head3.next.next.next.next.next.next = ListNode(7)**

**head3.next.next.next.next.next.next.next = ListNode(8)**

**head3.next.next.next.next.next.next.next.next = ListNode(9)**

**head3.next.next.next.next.next.next.next.next.next = ListNode(10)**

**new\_head3 = retain\_delete(head3, 1, 1)**

**# Print the modified linked list**

**node = new\_head3**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 1->3->5->7->9**

**print()**

**```**

**The above implementation retains M nodes and deletes N nodes in the linked list as described. It returns the head of the modified linked list.**

**Q.** **To insert nodes of the second list into the first list at alternate positions, we can use a simple iterative approach. We'll traverse both lists simultaneously, and for each node in the second list, we'll insert it into the first list at the appropriate position.**

**Here's the step-by-step algorithm to insert nodes of the second list into the first list at alternate positions:**

**1. If the second list is empty, return the modified first list.**

**2. Initialize two pointers, firstPtr and secondPtr, to the heads of the first and second lists, respectively.**

**3. While both firstPtr and secondPtr are not None, do the following:**

**3.1. Keep track of the next nodes in both lists, i.e., nextFirst and nextSecond.**

**3.2. Set the next pointer of the current node in the second list (secondPtr) to the next node of the first list (nextFirst).**

**3.3. Set the next pointer of the current node in the first list (firstPtr) to the current node in the second list (secondPtr).**

**3.4. Move the firstPtr to the nextFirst node and the secondPtr to the nextSecond node.**

**4. If there are remaining nodes in the second list, append them to the end of the first list.**

**5. Set the head of the second list to None (empty the second list).**

**6. Return the modified first list.**

**Here's the implementation in Python:**

**```python**

**class ListNode:**

**def \_\_init\_\_(self, val=0, next=None):**

**self.val = val**

**self.next = next**

**def insert\_at\_alternate\_positions(first, second):**

**if second is None:**

**return first**

**firstPtr = first**

**secondPtr = second**

**while firstPtr is not None and secondPtr is not None:**

**nextFirst = firstPtr.next**

**nextSecond = secondPtr.next**

**secondPtr.next = nextFirst**

**firstPtr.next = secondPtr**

**firstPtr = nextFirst**

**secondPtr = nextSecond**

**# Append remaining nodes of second list to the end of first list**

**if secondPtr is not None:**

**firstPtr.next = secondPtr**

**# Empty the second list**

**second = None**

**return first**

**# Example:**

**# First List: 5->7->17->13->11**

**# Second List: 12->10->2->4->6**

**# Expected Output:**

**# First List: 5->12->7->10->17->2->13->4->11->6**

**# Second List: None**

**head1 = ListNode(5)**

**head1.next = ListNode(7)**

**head1.next.next = ListNode(17)**

**head1.next.next.next = ListNode(13)**

**head1.next.next.next.next = ListNode(11)**

**head2 = ListNode(12)**

**head2.next = ListNode(10)**

**head2.next.next = ListNode(2)**

**head2.next.next.next = ListNode(4)**

**head2.next.next.next.next = ListNode(6)**

**new\_head1 = insert\_at\_alternate\_positions(head1, head2)**

**# Print the modified first list**

**node = new\_head1**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: 5->12->7->10->17->2->13->4->11->6**

**print()**

**# Print the modified second list**

**node = head2**

**while node is not None:**

**print(node.val, end=" ")**

**node = node.next**

**# Output: None**

**print()**

**```**

**The above implementation inserts nodes of the second list into the first list at alternate positions, without using any extra space. It returns the modified first list and sets the second list to None to empty it.**