**DSA – ASSIGNMENT 12**

💡 **Question 1** 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

**Example 2:**

Input:

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

Output:2 4 6 5 1

**Solution. :-**

* Initialize two pointers, slow and fast, pointing to the head of the linked list.
* 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.
* When the fast pointer reaches the end of the linked list, the slow pointer will be pointing to the middle node(s).
* Delete the middle node(s) by updating the next pointer of the previous node to skip the slow pointer.
* Return the modified linked list.

**class Node:**

**def \_\_init\_\_(self, data=None):**

**self.data = data**

**self.next = None**

**def delete\_middle(head):**

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

**return head**

**slow = head**

**fast = head**

**prev = None**

**while fast and fast.next:**

**fast = fast.next.next**

**prev = slow**

**slow = slow.next**

**# Delete the middle node(s)**

**prev.next = slow.next**

**return head**

**# Function to print the linked list**

**def print\_list(head):**

**curr = head**

**while curr:**

**print(curr.data, end=" ")**

**curr = curr.next**

**print()**

**# Create the linked list: 1->2->3->4->5**

**head = Node(1)**

**head.next = Node(2)**

**head.next.next = Node(3)**

**head.next.next.next = Node(4)**

**head.next.next.next.next = Node(5)**

**print("Original Linked List:")**

**print\_list(head)**

**head = delete\_middle(head)**

**print("Modified Linked List:")**

**print\_list(head)**

💡 **Question 2** 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.

**Solution. :-**

* Initialize two pointers, slow and fast, pointing to the head of the linked list.
* Move the slow pointer one step at a time and the fast pointer two steps at a time.
* If the fast pointer reaches the end of the linked list (i.e., it becomes NULL), there is no loop in the linked list. Return False.
* If the slow and fast pointers meet at any point, it indicates the presence of a loop in the linked list. Return True.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def detectLoop(head):**

**slow = head**

**fast = head**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**if slow == fast:**

**return True**

**return False**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(3)**

**node3 = Node(4)**

**head.next = node2**

**node2.next = node3**

**# Create a loop by connecting the last node to the second node**

**node3.next = node2**

**# Check if the linked list has a loop**

**hasLoop = detectLoop(head)**

**print(hasLoop)**

💡 **Question 3** 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.

**Solution. :-**

* Initialize two pointers, main\_ptr, and ref\_ptr, pointing to the head of the linked list.
* Move the ref\_ptr N nodes ahead in the linked list.
* Now, move both the main\_ptr and ref\_ptr one node at a time until ref\_ptr reaches the end of the linked list (i.e., ref\_ptr becomes NULL).
* At this point, the main\_ptr will be pointing to the Nth node from the end of the linked list.
* Return the value stored in the main\_ptr.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def findNthFromEnd(head, N):**

**main\_ptr = head**

**ref\_ptr = head**

**# Move ref\_ptr N nodes ahead**

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

**if ref\_ptr is None:**

**return None**

**ref\_ptr = ref\_ptr.next**

**# Move both pointers until ref\_ptr reaches the end**

**while ref\_ptr:**

**main\_ptr = main\_ptr.next**

**ref\_ptr = ref\_ptr.next**

**# main\_ptr will be pointing to the Nth node from the end**

**return main\_ptr.data**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(4)**

**node5 = Node(5)**

**node6 = Node(6)**

**node7 = Node(7)**

**node8 = Node(8)**

**node9 = Node(9)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**node6.next = node7**

**node7.next = node8**

**node8.next = node9**

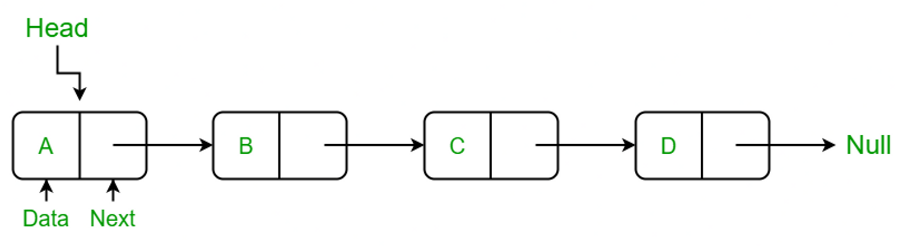
**# Find the Nth node from the end**

**N = 2**

**result = findNthFromEnd(head, N)**

**print(result)**

💡 **Question 4** Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.



**Example 1:**

**Input:** R->A->D->A->R->NULL

**Output:** Yes

**Example 2:**

**Input:** C->O->D->E->NULL

**Output:** No

**Solution. :-**

* Traverse the linked list to determine its length and store the characters in a list or array.
* Initialize two pointers, one at the beginning of the list and the other at the end.
* Compare the characters at the corresponding positions of the two pointers while moving towards the center of the list.
* If at any point the characters are not equal, return False.
* If the pointers meet or cross each other, return True as it indicates that the linked list is a palindrome.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def isPalindrome(head):**

**# Step 1: Traverse the linked list and store characters in a list**

**chars = []**

**current = head**

**while current:**

**chars.append(current.data)**

**current = current.next**

**# Step 2: Initialize pointers**

**start = 0**

**end = len(chars) - 1**

**# Step 3: Compare characters while moving towards the center**

**while start < end:**

**if chars[start] != chars[end]:**

**return False**

**start += 1**

**end -= 1**

**return True**

**# Create the linked list**

**head = Node('R')**

**node2 = Node('A')**

**node3 = Node('D')**

**node4 = Node('A')**

**node5 = Node('R')**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**# Check if the linked list is a palindrome**

**isPalindrome = isPalindrome(head)**

**if isPalindrome:**

**print("Yes")**

**else:**

**print("No")**

💡 **Question 5** 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.

**Example 1:**

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.

**Example 2:**

Input:

N = 4

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

X = 0

Output:1

Explanation:

The Linked list does not contains any loop.

**Example 3:**

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.

**Solution. :-**

* Initialize two pointers, slow and fast, pointing to the head of the linked list.
* Move the slow pointer one step at a time and the fast pointer two steps at a time until they meet or the fast pointer reaches the end of the linked list.
* If the fast pointer reaches the end of the linked list (i.e., it becomes NULL), there is no loop in the linked list. Return the linked list as it is.
* If the slow and fast pointers meet at any point, it indicates the presence of a loop in the linked list.
* Move the slow pointer back to the head of the linked list and keep the fast pointer at the meeting point.
* Move both pointers one step at a time until they meet again. This time, both pointers should move at the same pace.
* When the pointers meet again, it indicates the start of the loop.
* Set the next pointer of the node where the pointers meet to NULL, breaking the loop.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def detectAndRemoveLoop(head):**

**slow = head**

**fast = head**

**# Step 1: Detect the loop using Floyd's cycle-finding algorithm**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**if slow == fast:**

**break**

**# Step 2: If there is no loop, return the linked list as it is**

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

**return head**

**# Step 3: Move slow pointer to the head and keep fast pointer at the meeting point**

**slow = head**

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

**slow = slow.next**

**fast = fast.next**

**# Step 4: Break the loop by setting the next pointer of the meeting point to NULL**

**fast.next = None**

**return head**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(3)**

**node3 = Node(4)**

**head.next = node2**

**node2.next = node3**

**# Create a loop by connecting the last node to the second node**

**node3.next = node2**

**# Remove the loop from the linked list**

**head = detectAndRemoveLoop(head)**

**# Print the modified linked list**

**current = head**

**while current:**

**print(current.data, end=" ")**

**current = current.next**

💡 **Question 6** 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

**Example 1**:

Input:

M = 2, N = 2

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

Output:

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

**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

**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

**Solution. :-**

* Traverse the linked list using a pointer, and maintain a count of nodes visited.
* When the count reaches M, keep a reference to the current node as the previous node and move to the next node N times, deleting each node along the way.
* Once N nodes are deleted, update the next pointer of the previous node to point to the current node.
* Continue this process until the end of the linked list is reached.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def deleteNAfterM(head, M, N):**

**current = head**

**previous = None**

**while current:**

**# Traverse M nodes**

**count = 0**

**while current and count < M:**

**previous = current**

**current = current.next**

**count += 1**

**# Delete N nodes**

**count = 0**

**while current and count < N:**

**temp = current**

**current = current.next**

**del temp**

**count += 1**

**# Update the next pointer of the previous node**

**previous.next = current**

**return head**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(4)**

**node5 = Node(5)**

**node6 = Node(6)**

**node7 = Node(7)**

**node8 = Node(8)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**node6.next = node7**

**node7.next = node8**

**# Define M and N values**

**M = 2**

**N = 2**

**# Delete N nodes after M nodes**

**head = deleteNAfterM(head, M, N)**

**# Print the modified linked list**

**current = head**

**while current:**

**print(current.data, end=" ")**

**current = current.next**

💡 **Question 7** Given two linked lists, insert nodes of second list into first list at alternate positions of first list. For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

**Solution. :-**

* Initialize two pointers, current1 and current2, pointing to the heads of the first and second linked lists, respectively.
* Traverse both linked lists simultaneously.
* For each pair of nodes, insert the second linked list's node after the first linked list's node.
* Update the current pointers accordingly to proceed to the next pair of nodes.
* Continue this process until either the first or second linked list reaches the end.
* If the second linked list still has remaining nodes after reaching its end, append them to the end of the first linked list.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def insertAlternatePositions(first, second):**

**if not second: # If the second linked list is empty, return the first linked list**

**return first**

**current1 = first**

**current2 = second**

**while current1 and current2:**

**next1 = current1.next**

**next2 = current2.next**

**# Insert the second linked list's node after the first linked list's node**

**current1.next = current2**

**current2.next = next1**

**# Update the current pointers**

**current1 = next1**

**current2 = next2**

**if current2: # Append the remaining nodes of the second linked list to the end of the first linked list**

**current1.next = current2**

**# Clear the second linked list**

**second = None**

**return first**

**# Create the first linked list**

**first = Node(5)**

**first.next = Node(7)**

**first.next.next = Node(17)**

**first.next.next.next = Node(13)**

**first.next.next.next.next = Node(11)**

**# Create the second linked list**

**second = Node(12)**

**second.next = Node(10)**

**second.next.next = Node(2)**

**second.next.next.next = Node(4)**

**second.next.next.next.next = Node(6)**

**# Insert nodes of the second linked list into the first linked list at alternate positions**

**first = insertAlternatePositions(first, second)**

**# Print the modified first linked list**

**current = first**

**while current:**

**print(current.data, end=" ")**

**current = current.next**

**# Print the modified second linked list (should be empty)**

**current = second**

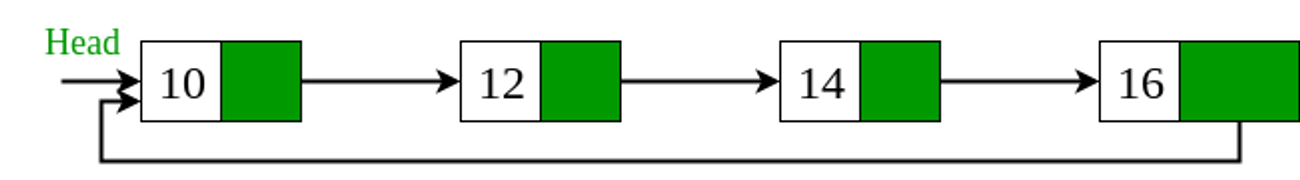
**while current:**

**print(current.data, end=" ")**

**current = current.next**

💡 **Question 8** Given a singly linked list, find if the linked list is [circular](https://www.geeksforgeeks.org/circular-linked-list/amp/) or not.

A linked list is called circular if it is not NULL-terminated and all nodes are connected in the form of a cycle. Below is an example of a circular linked list.



**Solution. :-**

* Initialize two pointers, slow and fast, to the head of the linked list.
* Traverse the linked list with the fast pointer moving two steps at a time and the slow pointer moving one step at a time.
* If the fast pointer reaches the end of the list (i.e., it becomes NULL), the linked list is not circular.
* If, at any point, the fast and slow pointers meet (i.e., they point to the same node), the linked list is circular.

**class Node:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.next = None**

**def isCircular(head):**

**if not head: # If the linked list is empty, it is not circular**

**return False**

**slow = head**

**fast = head.next**

**while fast and fast.next:**

**if slow == fast: # The fast pointer caught up to the slow pointer, indicating a cycle**

**return True**

**slow = slow.next**

**fast = fast.next.next**

**return False # The fast pointer reached the end of the list, indicating no cycle**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(4)**

**node5 = Node(5)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**# Create a cycle by connecting the last node to the second node**

**node5.next = node2**

**# Check if the linked list is circular**

**isCircular = isCircular(head)**

**if isCircular:**

**print("The linked list is circular.")**

**else:**

**print("The linked list is not circular.")**