NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 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

ANS –

To delete the middle node(s) of a singly linked list, we can use the two-pointer approach. We'll use two pointers, slow and fast, where slow moves one node at a time and fast moves two nodes at a time. By the time fast reaches the end of the list, slow will be pointing to the middle node.

Here's the step-by-step process:

If the linked list is empty or has only one node, return None.

Initialize prev to None and slow and fast to the head of the linked list.

Traverse the linked list using the two-pointer approach:

Move fast two nodes at a time.

Update prev to slow.

Move slow one node at a time.

After the traversal, slow will be pointing to the middle node(s).

To delete the middle node(s), update the next pointer of prev to skip the middle node(s).

Return the modified linked list.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def deleteMiddleNode(head):

if not head or not head.next:

return None

prev = None

slow = fast = head

while fast and fast.next:

fast = fast.next.next

prev = slow

slow = slow.next

prev.next = slow.next

return head

Let's create the linked lists and test the function with the provided examples:

# 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)

head1 = deleteMiddleNode(head1)

while head1:

print(head1.val, end=' ')

head1 = head1.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)

head2 = deleteMiddleNode(head2)

while head2:

print(head2.val, end=' ')

head2 = head2.next

# Output: 2 4 6 5 1

The function produces the expected results for the given examples.

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.

Ans –

To check if a linked list contains a loop, we can use the Floyd's Cycle-Finding Algorithm, also known as the "Tortoise and Hare" algorithm. This algorithm uses two pointers, one moving at a slower pace (tortoise) and the other moving at a faster pace (hare). If the linked list contains a loop, eventually the hare will catch up to the tortoise.

Here's the step-by-step process:

Initialize the tortoise and hare pointers to the head of the linked list.

Move the tortoise one node at a time and the hare two nodes at a time.

If the hare reaches the end of the linked list (i.e., encounters a None node), there is no loop in the linked list, so return False.

If the tortoise and hare pointers meet (i.e., their references are the same), there is a loop in the linked list, so return True.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def hasCycle(head):

if not head or not head.next:

return False

tortoise = hare = head

while hare and hare.next:

tortoise = tortoise.next

hare = hare.next.next

if tortoise == hare:

return True

return False

Let's create the linked lists and test the function with the provided examples:

# 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

print(hasCycle(head1))

# 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)

print(hasCycle(head2))

# Output: False

The function produces the expected results for the given examples.

Question 3

Given a linked list consisting of L nodes and given a number N. The task is to find the Nth 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 approach. We maintain two pointers, first and second, with a distance of N between them. We move both pointers simultaneously until the first pointer reaches the end of the linked list. At this point, the second pointer will be pointing to the Nth node from the end.

Here's the step-by-step process:

Initialize the first and second pointers to the head of the linked list.

Move the first pointer N steps ahead.

If the first pointer becomes None before reaching N steps (i.e., the number of nodes in the linked list is less than N), return -1.

Move both the first and second pointers one step at a time until the first pointer reaches the end of the linked list.

At this point, the second pointer will be pointing to the Nth node from the end. Return the value of the node.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def findNthFromEnd(head, N):

first = second = head

# Move the first pointer N steps ahead

for \_ in range(N):

if not first:

return -1

first = first.next

# Move both pointers until the first pointer reaches the end

while first:

first = first.next

second = second.next

return second.val

Let's create the linked list and test the function with the provided examples:

# 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)

print(findNthFromEnd(head1, 2))

# 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)

print(findNthFromEnd(head2, 5))

# Output: -1

The function produces the expected results for the given examples.

Question 4

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

Examples:

Input: R->A->D->A->R->NULL

Output: Yes

Input: C->O->D->E->NULL

Output: No

ANS –

To determine whether a singly linked list is a palindrome, we can use a two-pointer approach along with a stack.

Here's the step-by-step process:

Traverse the linked list using two pointers, a slow pointer and a fast pointer. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. This allows the slow pointer to reach the middle of the linked list when the fast pointer reaches the end.

Push the elements encountered by the slow pointer onto a stack.

When the fast pointer reaches the end of the linked list, the slow pointer will be at the middle (or the middle-left) of the linked list.

If the length of the linked list is odd, move the slow pointer one step ahead to skip the middle element.

Compare the remaining elements of the linked list with the elements popped from the stack. If any of the elements don't match, return false.

If all the elements match, return true.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def isPalindrome(head):

slow = fast = head

stack = []

# Traverse the linked list and push elements onto the stack

while fast and fast.next:

stack.append(slow.val)

slow = slow.next

fast = fast.next.next

# Skip the middle element if the length of the linked list is odd

if fast:

slow = slow.next

# Compare the remaining elements with the elements popped from the stack

while slow:

if slow.val != stack.pop():

return False

slow = slow.next

return True

Let's create the linked list and test the function with the provided examples:

# Example 1

# Input: R->A->D->A->R->NULL

# Output: Yes

head1 = ListNode('R')

head1.next = ListNode('A')

head1.next.next = ListNode('D')

head1.next.next.next = ListNode('A')

head1.next.next.next.next = ListNode('R')

print(isPalindrome(head1))

# Output: True

# Example 2

# Input: C->O->D->E->NULL

# Output: No

head2 = ListNode('C')

head2.next = ListNode('O')

head2.next.next = ListNode('D')

head2.next.next.next = ListNode('E')

print(isPalindrome(head2))

# Output: False

The function produces the expected results for the given examples.

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.

ANS –

To remove a loop from a linked list, we can use Floyd's cycle detection algorithm, also known as the "tortoise and hare" algorithm. This algorithm helps us detect if a loop exists in the linked list and find the node where the loop starts.

Here's the step-by-step process to remove the loop:

Initialize two pointers, slow and fast, both 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 or reaches the last node), it means there is no loop. Return the linked list as it is.

If the slow and fast pointers meet at some point, it confirms the presence of a loop in the linked list.

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

Move both pointers one step at a time until they meet again. This time, both pointers will move at the same pace.

The point at which they meet again will be the starting point of the loop.

To remove the loop, move the pointer from the meeting point (starting point of the loop) and another pointer from the head of the linked list, both moving one step at a time, until they meet at the previous node of the meeting point.

Set the next pointer of the previous node to NULL, effectively removing the loop.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def detectAndRemoveLoop(head):

slow = fast = head

# Detect the loop using Floyd's cycle detection algorithm

while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

break

# If there is no loop, return the linked list as it is

if not fast or not fast.next:

return head

# Move one pointer back to the head and find the starting point of the loop

slow = head

while slow.next != fast.next:

slow = slow.next

fast = fast.next

# Remove the loop by setting the next pointer of the previous node to NULL

fast.next = None

return head

Let's create the linked list and test the function with the provided examples:

# 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

head1 = detectAndRemoveLoop(head1)

# Print the modified linked list

while head1:

print(head1.val, end=' ')

head1 = head1.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)

head2 = detectAndRemoveLoop(head2)

# Print the modified linked list

while head2:

print(head2.val, end=' ')

head2 = head2.next

# Output: 1 8 3 4

# 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

head3 = detectAndRemoveLoop(head3)

# Print the modified linked list

while head3:

print(head3.val, end=' ')

head3 = head3.next

# Output: 1 2 3 4

The function produces the expected results for the given examples, removing the loops from the linked lists.

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

Examples:

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 traverse the linked list and keep track of two pointers: current and previous. The current pointer is used to traverse the linked list, and the previous pointer is used to keep track of the previous node.

The steps to retain M nodes and delete N nodes in a linked list are as follows:

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

Traverse the linked list using the current pointer.

For every M nodes, move the current pointer M-1 times.

If the current pointer becomes NULL before moving M-1 times, break the loop as we have reached the end of the linked list.

If the current pointer is not NULL, move the current pointer N times to reach the node after the N nodes to be deleted.

If the current pointer becomes NULL before moving N times, break the loop as we have reached the end of the linked list.

Update the next pointer of the previous node to the node pointed by the current pointer, effectively deleting the N nodes.

Set the previous pointer to the node pointed by the current pointer.

Repeat steps 3-8 until the end of the linked list is reached.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def retainAndDelete(head, M, N):

if not head:

return head

current = head

previous = None

while current:

# Retain M nodes

for \_ in range(M):

if current:

previous = current

current = current.next

else:

break

# Delete N nodes

for \_ in range(N):

if current:

current = current.next

else:

break

# Connect the previous node to the next node after N nodes

previous.next = current

return head

Let's create the linked list and test the function with the provided examples:

# 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)

head1 = retainAndDelete(head1, 2, 2)

# Print the modified linked list

while head1:

print(head1.val, end=' ')

head1 = head1.next

# 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

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)

head2 = retainAndDelete(head2, 3, 2)

# Print the modified linked list

while head2:

print(head2.val, end=' ')

head2 = head2.next

# 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

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)

head3 = retainAndDelete(head3, 1, 1)

# Print the modified linked list

while head3:

print(head3.val, end=' ')

head3 = head3.next

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

The function produces the expected results for the given examples, retaining M nodes and deleting N nodes in the linked list.

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.

ANS –

To insert nodes of the second list into the first list at alternate positions, we can follow these steps:

Check if either of the lists is empty. If one of them is empty, return the other list as the result.

Initialize two pointers, current1 and current2, to the heads of the first and second lists, respectively.

While both current1 and current2 are not NULL, do the following:

Store the next pointers of current1 and current2 in temporary variables, next1 and next2, respectively.

Point the next pointer of current2 to the node pointed by next1.

Point the next pointer of current1 to current2.

Update current1 to next1 and current2 to next2.

If the second list still has nodes remaining, append the remaining nodes of the second list to the end of the first list.

Return the modified first list.

Here's the implementation of the above algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def mergeLists(first, second):

if not first:

return second

if not second:

return first

current1 = first

current2 = second

while current1 and current2:

next1 = current1.next

next2 = current2.next

current1.next = current2

current2.next = next1

current1 = next1

current2 = next2

if current2:

current1.next = current2

return first

Let's create the first and second lists and test the function with the provided example:

# Example

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

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

# After merging: 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)

head1 = mergeLists(head1, head2)

# Print the modified first list

while head1:

print(head1.val, end=' ')

head1 = head1.next

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

# Print the modified second list

print()

print(head2)

# Output: None

The function correctly merges the nodes of the second list into the first list at alternate positions, satisfying the given requirements.

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.

ANS –

To determine if a singly linked list is circular, we can use the "Floyd's Cycle-Finding Algorithm" also known as the "Tortoise and Hare Algorithm". This algorithm uses two pointers, often called slow and fast pointers, to traverse the linked list. If the linked list is circular, the fast pointer will eventually catch up to the slow pointer, indicating the presence of a cycle. If the linked list is not circular, the fast pointer will reach the end of the list (i.e., become NULL) before catching up to the slow pointer.

Here's the implementation of the algorithm in Python:

class ListNode:

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

self.val = val

self.next = next

def isCircular(head):

if not head or not head.next:

return False

slow = head

fast = head.next

while fast and fast.next:

if slow == fast:

return True

slow = slow.next

fast = fast.next.next

return False

Let's create a circular linked list and a non-circular linked list to test the function:

# Circular Linked List: 1->2->3->4->5->2 (5 is connected to 2, forming a cycle)

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 = head1.next

# Non-Circular Linked List: 1->2->3->4->5

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)

print(isCircular(head1)) # Output: True

print(isCircular(head2)) # Output: False

The isCircular function correctly identifies whether a linked list is circular or not based on the presence of a cycle in the linked list.