**Assignment 11 Questions - Binary Search | DSA**

**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**

**------------**

**Solution :** class Solution():

**------------** def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:

if head.next == None:

return None

slow, fast = head, head.next.next

while fast and fast.next:

slow = slow.next

fast = fast.next.next

slow.next = slow.next.next

return head

**-------------------------**

**Complexity Analysis:**

**-------------------------**

Let n be the length of the input linked list.

* Time complexity : 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/2 steps. In each step, we move both fast and slow, which takes a constant amount of time.
* Space complexity: O(1). We only need two pointers, so the space complexity is O(1).

=====================================x===================================

**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

**------------**

**Solution :** class Node():

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

self.data = data

         self.ptr = None

def ispalindrome(head):

     slow = head

      stack = []

     ispalin = True

     while slow != None:

         stack.append(slow.data)

         slow = slow.ptr

     while head != None:

          i = stack.pop()

         if head.data == i:

             ispalin = True

         else:

             ispalin = False

             break

         head = head.ptr

     return ispalin

one = Node(1)

two = Node(2)

three = Node(3)

four = Node(4)

five = Node(3)

six = Node(2)

seven = Node(1)

one.ptr = two

two.ptr = three

three.ptr = four

four.ptr = five

five.ptr = six

six.ptr = seven

seven.ptr = None

result = ispalindrome(one)

print(result)

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time Complexity : O(N), Iterating over the linked list of size N.
* **Space Complexity** : O(N), Using an auxiliary stack

=====================================x===================================

**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**

**------------**

**Solution :** class Node():

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

self.data = data

         self.next = next

def printList(head):

     curr = head

     while curr:

         print(curr.data, end=' —> ')

         curr = curr.next

     print('None')

def removeCycle(head):

     prev = None        # previous pointer

     curr = head        # main pointer

     s = set()

  while curr:

         if curr in s:

             prev.next = None

             return

s.add(curr)

         prev = curr

         curr = curr.next

if \_\_name\_\_ == '\_\_main\_\_':

     n = 5

     head = None

     for i in reversed(range(1, n + 1)):

         head = Node(i, head)

     head.next.next.next.next.next = head.next

     removeCycle(head)

     printList(head)

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time Complexity : O(n)
* **Space Complexity** : O(N), Using an auxiliary stack

=====================================x===================================

**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.

**Example 1:** 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

**------------**

**Solution :** class Node():

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

         self.data = data

         self.next = None

class LinkedList:

       def \_\_init\_\_(self):

         self.head = None

def push(self, new\_data):

         new\_node = Node(new\_data)

         new\_node.next = self.head

         self.head = new\_node

     def printList(self):

         temp = self.head

         while(temp):

             print (temp.data,end=" ")

             temp = temp.next

     def skipMdeleteN(self, M, N):

         curr = self.head

         while(curr):

             for count in range(1, M):

                 if curr is None:

                     return

                 curr = curr.next

             if curr is None :

                 return

t = curr.next

             for count in range(1, N+1):

                 if t is None:

                     break

                 t = t.next

                   curr.next = t

             curr = t

llist = LinkedList()

M = 2

N = 3

llist.push(10)

llist.push(9)

llist.push(8)

llist.push(7)

llist.push(6)

llist.push(5)

llist.push(4)

llist.push(3)

llist.push(2)

llist.push(1)

print ("M = %d, N = %d\nGiven Linked List is:" %(M, N))

llist.printList()

llist.skipMdeleteN(M, N)

print ("\nLinked list after deletion is")

llist.printList()

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time Complexity : O(n) where n is number of nodes in linked list.
* Space Complexity : O(1).

=====================================x===================================

**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 :** class Node(object):

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

         self.data = data

         self.next = None

class LinkedList(object):

     def \_\_init\_\_(self):

         self.head = None

     def push(self, new\_data:int):

         new\_node = Node(new\_data)

         new\_node.next = self.head

         self.head = new\_node

def printList(self):

         temp = self.head

         while temp != None:

             print(temp.data)

             temp = temp.next

     def merge(self, p, q):

         p\_curr = p.head

         q\_curr = q.head

           while p\_curr != None and q\_curr != None:

               p\_next = p\_curr.next

             q\_next = q\_curr.next

               q\_curr.next = p\_next

             p\_curr.next = q\_curr

                         p\_curr = p\_next

             q\_curr = q\_next

             q.head = q\_curr

llist1 = LinkedList()

llist2 = LinkedList()

     llist1.push(3)

llist1.push(2)

llist1.push(1)

llist1.push(0)

for i in range(8, 3, -1):

     llist2.push(i)

print("First Linked List:")

llist1.printList()

print("Second Linked List:")

llist2.printList()

   llist1.merge(p=llist1, q=llist2)

print("Modified first linked list:")

llist1.printList()

print("Modified second linked list:")

llist2.printList()

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time Complexity : O(min(n1, n2)), where n1 and n2  represents the length of the given two linked lists.
* Space Complexity : O(1), no extra space is required, so it is a constant.

=====================================x===================================

**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 :** class Node(object):

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

         self.data = data  # Assign data

         self.next = None  # Initialize next as null

class LinkedList:

        def \_\_init\_\_(self):

         self.head = None

def Circular(head):

     if head == None:

         return True

node = head.next

     i = 0

       while((node is not None) and (node is not head)):

         i = i + 1

         node = node.next

     return(node == head)

if \_\_name\_\_ == '\_\_main\_\_':

     llist = LinkedList()

     llist.head = Node(1)

     second = Node(2)

     third = Node(3)

     fourth = Node(4)

     llist.head.next = second

     second.next = third

     third.next = fourth

     if (Circular(llist.head)):

         print('Yes')

     else:

         print('No')

     fourth.next = llist.head

     if (Circular(llist.head)):

         print('Yes')

     else:

         print('No')

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time Complexity : O(n).
* Space Complexity : O(1), no extra space is required, so it is a constant.

=====================================x===================================