**Linked List Questions**

Table of Contents

[**Theory** 2](#_Toc208243057)

[Linked List 2](#_Toc208243058)

[Doubly Linked List 2](#_Toc208243059)

[What are uses of Linked List? 2](#_Toc208243060)

[LEVEL 1: **EASY** 3](#_Toc208243061)

[1. Implement basic functionalities of Linked List in python. 3](#_Toc208243062)

[2. Reverse Linked List 3](#_Toc208243063)

[3. Linked List Cycle 3](#_Toc208243064)

[4. Middle of the Linked List 3](#_Toc208243065)

[5. Merge Two Sorted Lists 3](#_Toc208243066)

[6. Intersection of Two Linked Lists 3](#_Toc208243067)

[7. Palindrome Linked List 3](#_Toc208243068)

[8. Remove Duplicates from Sorted List 3](#_Toc208243069)

[LEVEL 2: **Medium** 4](#_Toc208243070)

[LEVEL 3: **Difficult** 5](#_Toc208243071)

[**SOLUTIONS:** 6](#_Toc208243072)

[**LEVEL 1:** 6](#_Toc208243073)

[**LEVEL 2:** 11](#_Toc208243074)

[**LEVEL 3:** 12](#_Toc208243075)

# Categories: LC problem number is also given, like 206, 203 etc

These build the fundamentals of traversing, inserting, deleting, or modifying nodes.

* 206. Reverse Linked List
* 203. Remove Linked List Elements
* 21. Merge Two Sorted Lists
* 83. Remove Duplicates from Sorted List
* 141. Linked List Cycle (easy version of cycle detection)
* 160. Intersection of Two Linked Lists
* 234. Palindrome Linked List (basic version)

**2. Slow & Fast Pointers (Two Pointers)**

Used for finding cycles, middle node, detecting intersection, etc.

* 141. Linked List Cycle
* 142. Linked List Cycle II
* 876. Middle of the Linked List
* 19. Remove Nth Node From End of List
* 160. Intersection of Two Linked Lists
* 234. Palindrome Linked List
* 2095. Delete the Middle Node of a Linked List

**3. Cycle / Loop Based Problems**

Directly focus on cycle detection and cycle-related tricks.

* 141. Linked List Cycle
* 142. Linked List Cycle II
* 287. Find the Duplicate Number (array, but solved with linked list cycle detection)
* 457. Circular Array Loop
* 202. Happy Number (linked list cycle logic applied on numbers)

**4. Reversal Based Problems**

Classic interview set where you reverse entire or parts of linked list.

* 206. Reverse Linked List
* 92. Reverse Linked List II
* 25. Reverse Nodes in k-Group
* 61. Rotate List
* 2130. Maximum Twin Sum of a Linked List

**5. Dummy Node Trick (Simplifies Edge Cases)**

Dummy node helps when handling head deletions or merges.

* 21. Merge Two Sorted Lists
* 23. Merge k Sorted Lists
* 82. Remove Duplicates from Sorted List II
* 19. Remove Nth Node From End of List
* 86. Partition List
* 328. Odd Even Linked List

**6. Recursive Problems**

These require recursion on linked lists (good for interviews).

* 206. Reverse Linked List (recursive solution)
* 234. Palindrome Linked List (recursive check)
* 25. Reverse Nodes in k-Group (recursive version)
* 24. Swap Nodes in Pairs
* 445. Add Two Numbers II

**7. Math / Arithmetic on Linked Lists**

Problems where lists represent numbers.

* 2. Add Two Numbers
* 445. Add Two Numbers II
* 369. Plus One Linked List
* 1019. Next Greater Node in Linked List
* 1290. Convert Binary Number in a Linked List to Integer

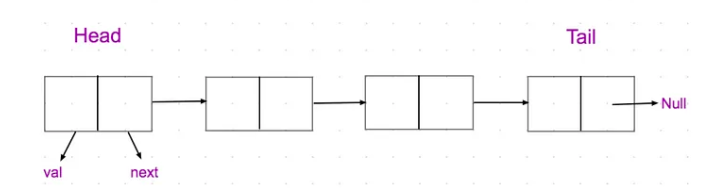
**8. Complex / Advanced Problems**

These are challenging and combine multiple concepts.

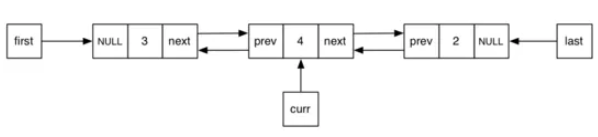
* 23. Merge k Sorted Lists (in striver video too)
* 25. Reverse Nodes in k-Group
* 138. Copy List with Random Pointer
* 430. Flatten a Multilevel Doubly Linked List
* 708. Insert into a Sorted Circular Linked List
* 147. Insertion Sort List
* 148. Sort List (merge sort in striver video L26,27th video)

# **Theory**

## Linked List

  
A linked list is a linear data structure where elements are stored in nodes, each containing a reference to the next node in the sequence.

## Doubly Linked List



## What are uses of Linked List?

Linked lists are used in various scenarios in computer systems and system design due to their flexibility and efficiency in certain operations:

1. **Dynamic Memory Allocation**: Linked lists are used in dynamic memory allocation systems such as malloc() and free() in C programming language to manage memory efficiently.
2. **Implementing Data Structures**: Linked lists are fundamental for implementing other data structures like stacks, queues, and hash tables. Eg Python deque is implemented using doublly linked list.
3. **File Systems**: Linked lists can be used to represent directories and files in a file system, where each node represents a file or directory, and each node contains a reference to the next node in the directory structure.
4. **LRU Cache**: Linked lists are used in the implementation of Least Recently Used (LRU) cache, where the least recently used item is removed from the cache when the cache is full. Each node in the linked list represents an item in the cache.
5. **Undo Functionality**: Linked lists can be used to implement undo functionality in applications where each node represents a state, and traversing the linked list allows users to undo actions.
6. **Sparse Matrix Representation**: Linked lists can be used to represent sparse matrices efficiently, where each node represents a non-zero element in the matrix.
7. **Polynomial Representation**: Linked lists can be used to represent polynomials efficiently, where each node represents a term in the polynomial.

# LEVEL 1: **EASY ( do categorization)**

Add some easier questions, this are little medium ty pe

### Implement basic functionalities of Linked List in python.

### Reverse a Doubly Linked List

Link: <https://www.geeksforgeeks.org/problems/reverse-a-doubly-linked-list/1>

### Sort a linked list of 0s, 1s and 2s

Link: <https://www.geeksforgeeks.org/problems/given-a-linked-list-of-0s-1s-and-2s-sort-it/1>

### Reverse Linked List

Link: <https://leetcode.com/problems/reverse-linked-list/>

### Linked List Cycle

Link: <https://leetcode.com/problems/linked-list-cycle/>

### Middle of the Linked List (put in medium)

Link: <https://leetcode.com/problems/middle-of-the-linked-list/>

### Merge Two Sorted Lists

Link: <https://leetcode.com/problems/merge-two-sorted-lists/>

### Intersection of Two Linked Lists

Link: <https://leetcode.com/problems/intersection-of-two-linked-lists/>

### Palindrome Linked List

Link: <https://leetcode.com/problems/palindrome-linked-list/>

### Remove Duplicates from Sorted List

Link: <https://leetcode.com/problems/remove-duplicates-from-sorted-list/>

### Find Pairs with given sum in DLL

Link: <https://www.geeksforgeeks.org/problems/find-pairs-with-given-sum-in-doubly-linked-list/1>

Categories:

1. Linked List Traversal

2. Fast and Slow

3. Loops in linked List

4. Linked List with Recursion

# LEVEL 2: **Medium**

### Remove Nth Node from End of List

Link: <https://leetcode.com/problems/remove-nth-node-from-end-of-list/>

### Add 1 to a number represented as Linked List

Link: <https://www.geeksforgeeks.org/problems/add-1-to-a-number-represented-as-linked-list/1>

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **LEVEL 1:**

**\*\*Add diagrams and explanation in notes later\*\***

1. Linked List implementation

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def append(self, data):

        new\_node = Node(data)

        if not self.head:

            self.head = new\_node

            return

        last\_node = self.head

        while last\_node.next:

            last\_node = last\_node.next

        last\_node.next = new\_node

    def delete\_node(self, key):

        current\_node = self.head

        if current\_node and current\_node.data == key:

            self.head = current\_node.next

            current\_node = None

            return

        prev = None

        while current\_node and current\_node.data != key:

            prev = current\_node

            current\_node = current\_node.next

        if current\_node is None:

            return

        prev.next = current\_node.next

        current\_node = None

    def print\_list(self):

        current\_node = self.head

        while current\_node:

            print(current\_node.data, end=" ")

            current\_node = current\_node.next

        print()

# Example usage:

# Create a linked list

linked\_list = LinkedList()

# Append elements

linked\_list.append(1)

linked\_list.append(2)

linked\_list.append(3)

# Print the linked list

print("Linked list:")

linked\_list.print\_list()  # Output: 1 2 3

# Delete an element

linked\_list.delete\_node(2)

# Print the linked list after deletion

print("Linked list after deleting 2:")

linked\_list.print\_list()  # Output: 1 3

1. Reverse a Doubly Linked List

Explain thought process, check for one node, apply same for all nodes

#Add comment, execution and dry run later

class Solution:

    def reverse(self, head):

        # code here

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

            return head

        curr = head

        prev = None

        while curr:

            prev = curr.prev

            curr.prev = curr.next   #now it holds next where curr should move

            curr.next = prev

            curr = curr.prev

        return prev.prev

Version 2:

class Solution:

    def reverse(self, head):

        # code here

        prev = None

        curr = head

        while curr:

            nxt = curr.next       #holds next Node

            curr.next = curr.prev  #swaps

            curr.prev = nxt

            prev = curr

            curr = nxt

        return prev

1. Sort a linked list of 0s, 1s and 2s

**Approach Explanation**

1. **Divide into buckets:**  
   Create three separate linked lists — one for all 0s, one for all 1s, and one for all 2s.
2. **Traverse input:**  
   As you iterate through the original list, append each node to the respective list based on its value.
3. **Merge lists:**  
   Connect the 0s list → 1s list → 2s list to form a single segregated list.
4. **Return new head:**  
   The head of the 0s list (skipping dummy) becomes the head of the final sorted linked list.

class Solution:

    def segregate(self, head):

        # Create 3 dummy heads for 3 separate linked lists: 0s, 1s, and 2s

        head\_z = Node(-1)   # dummy head for 0s list

        head\_o = Node(-1)   # dummy head for 1s list

        head\_t = Node(-1)   # dummy head for 2s list

        # Pointers to build each list

        temp\_z, temp\_o, temp\_t = head\_z, head\_o, head\_t

        # Traverse original list and distribute nodes into 0s, 1s, 2s

        temp = head

        while temp:

            if temp.data == 0:

                temp\_z.next = Node(0)     # attach new node with value 0

                temp\_z = temp\_z.next

            elif temp.data == 1:

                temp\_o.next = Node(1)     # attach new node with value 1

                temp\_o = temp\_o.next

            else:

                temp\_t.next = Node(2)     # attach new node with value 2

                temp\_t = temp\_t.next

            temp = temp.next

        # Merge the three lists together: 0s → 1s → 2s

        temp\_o.next = head\_t.next    # 1s connect to 2s

        temp\_z.next = head\_o.next    # 0s connect to 1s

        # Return the head of the new sorted list (skipping dummy)

        return head\_z.next

1. Reverse a Linked List

# class ListNode:

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

#         self.val = val

#         self.next = next

class Solution:

    def reverseList(self, head: ListNode) -> ListNode:

        curr,prev = head, None

        while(curr):

            nxt = curr.next

            curr.next = prev

            prev = curr

            curr = nxt

        return prev

1. Linked List Cycle

Approach 1: Store nodes in set, and check if a node is already visited

class Solution:

    def hasCycle(self, head: Optional[ListNode]) -> bool:

        s = set()

        temp = head

        while temp:

            if temp in s:

                return True

            s.add(temp)

            temp = temp.next

        return False

class Solution:

    def hasCycle(self, head: Optional[ListNode]) -> bool:

        slow, fast = head, head

        while(fast and fast.next):

            slow = slow.next

            fast = fast.next.next

            if slow == fast:

                return True

Add explanation: <https://www.youtube.com/watch?v=wiOo4DC5GGA&list=PLgUwDviBIf0rAuz8tVcM0AymmhTRsfaLU&index=15>

1. Middle of the Linked List

# Definition for singly-linked list.

# class ListNode:

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

#         self.val = val

#         self.next = next

class Solution:

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

        #Here we use Slow and fast pointer algorithm

        slow = head

        fast = head

        while(fast and fast.next):

            slow = slow.next

            fast = fast.next.next

        return slow

1. Merge Two Sorted Lists

[Detailed Solution](https://leetcode.com/problems/merge-two-sorted-lists/solutions/2212406/python-fastest-explanation/)

class Solution:

    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:

        head=ListNode()

        t=head

        while(l1 and l2):

            if l1.val < l2.val:

                t.next = ListNode(l1.val)

                l1 = l1.next

            elif l1.val >= l2.val:

                t.next = ListNode(l2.val)

                l2 = l2.next

            t = t.next

        if l1 is not None:

            t.next=l1

        if l2 is not None:

            t.next=l2

        return head.next

1. Intersection of Two Linked List

# class ListNode:

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.next = None

class Solution:

    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:

        t1=headA

        t2=headB

        l1=l2=0

        while(t1):

            l1+=1

            t1=t1.next

        while(t2):

            t2=t2.next

            l2+=1

        if(l1<l2):

            large,small = headB,headA

        if(l1>=l2):

            large,small = headA,headB

        for \_ in range(abs(l1-l2)):

            large = large.next

        while large!=small:

            large=large.next

            small=small.next

        return small

1. Palindrome Linked List

class Solution:

    def isPalindrome(self, head: Optional[ListNode]) -> bool:

        stack = []

        #First push all elements of linked list in stack

        t = head

        while(t):

            stack.append(t.val)

            t = t.next

        #Traverse Linked List again, and also pop from stack to get elements in reverse order

        #if value don't match, it is not palindrome

        t = head

        while(t):

            if t.val != stack.pop():

                return False

            t = t.next

        return True

Other approach too: <https://www.youtube.com/watch?v=lRY_G-u_8jk&list=PLgUwDviBIf0rAuz8tVcM0AymmhTRsfaLU&index=11>

1. Remove Duplicates from Sorted List

class Solution:

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

        temp = head

        if not head: return head

        while(temp.next):

            if temp.next.val == temp.val:

                temp.next = temp.next.next

            else:

                temp = temp.next

        return head

1. Find Pairs with given sum in doubly linked list

class Solution:

    def findPairsWithGivenSum(self, target : int, head : Optional['Node']) -> List[List[int]]:

        # using concept like binary search since sorted

        left = head           # start pointer (leftmost node of DLL)

        right = head          # end pointer (rightmost node of DLL)

        ans = []              # stores valid pairs

        # Step 1: Move `right` to the last node

        while right.next:

            right = right.next

        # Step 2: Use two-pointer technique

        while left.data < right.data:   # stop when pointers meet or cross

            nodes\_sum = left.data + right.data   # sum of current pair

            if nodes\_sum == target:

                # Found a valid pair

                ans.append([left.data, right.data])

                # Move both pointers inward

                left = left.next

                right = right.prev

            elif nodes\_sum > target:

                # If sum is too big, decrease it by moving `right` leftward

                right = right.prev

            else:

                # If sum is too small, increase it by moving `left` rightward

                left = left.next

        return ans

## **LEVEL 2:**

1. Remove Nth node from End of List

<https://www.youtube.com/watch?v=3kMKYQ2wNIU&list=PLgUwDviBIf0rAuz8tVcM0AymmhTRsfaLU&index=9>

go through both approach, code for both and add in notes

create diagrams too, use it for final youtube video creation

class Solution:

    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

        slow, fast = head, head

        k = n

        while k:

            fast = fast.next

            k-=1

        #case when n=len(linked list), need to remove head

        if fast is None:

            return head.next

        #move fast and slow together now

        #if took just fast, fast move to none and slow move to 1 ahead position

        while fast.next:

            fast = fast.next

            slow = slow.next

        slow.next = slow.next.next

        return head

1. Add 1 to a number represented as Linked List

Approach 1: reverse the linked list, add 1 and again reverse the linked list. This works

Approach 2: using recursion and backtracking

**Backtracking Approach:**

1. The problem is to add 1 to a number represented by a linked list (each node = digit).
2. We recursively go to the **last node** (rightmost digit) using the helper function.
3. On returning (backtracking), we add the carry to the current node’s value (node.data).
4. If the sum becomes 10, we set it to 0 and return carry 1 to the previous node.
5. If not 10, return carry 0 to stop further propagation.
6. Finally, if a carry remains after processing the head, we create a new node with 1 at the front.

class Solution:

# This method returns the carry for given node

    def helper(self, node):

        if node is None: # Base case: if we reach end of list, we need to add 1

            return 1

        carry = self.helper(node.next)

        node.data = node.data + carry # Add carry to current node's value

        if node.data == 10: #If sum becomes 10, set current digit to 0 and return carry=1

            node.data = 0

            return 1

        return 0

    def addOne(self, head):

        # Start recursion from head

        carry = self.helper(head)

        # If carry remains even after head is processed,

        # we need a new node in front (e.g., 999 + 1 = 1000)

        if carry == 1:

            parent = Node(1)

            parent.next = head

            return parent

        # Otherwise, return updated list

        return head

Categories:

1. Linked List Traversal

2. Fast and Slow

3. Loops in linked List

4. Linked List with Recursion

## **LEVEL 3:**