**6.3 Medium Problems on Linked List:**

1. **Reverse a Doubly Linked List**

**Problem Statement:** Reverse a Doubly Linked List.

**Illustrate the working of function by doing dry run on above linked list.**

**Program:**

*public* *class* ReverseDLL {

*static* *Node* head;

*static* *class* **Node**{

*int* data;

*Node* prev;

*Node* next;

**Node**(*int* *value*){

            data = value;

        }

    }

*static* *void* **insert**(*int* *value*){

*Node* new\_node = new Node(value);

        if(head == null){

            head = new\_node;

            return;

        }

*Node* temp = head;

        while(temp.next != null){

            temp = temp.next;

        }

        new\_node.prev = temp;

        temp.next = new\_node;

    }

*static* *void* **printList**(){

*Node* temp = head;

        while(temp!=null){

            System.out.print("" + temp.data + "->");

            temp=temp.next;

        }

        System.out.println("null");

    }

*public* *static* *void* main(*String*[] *args*) {

        insert(10); insert(20); insert(30);

        System.out.print(" Original LinkedList: ");

        printList();

        System.out.print(" Reversed LinkedList: ");

        reverseDLL();

        printList();

    }

}

* **Solution1: Naive approach (Reverse the data using stack in double pass)**

**Step1:** Traverse the whole Linked List and keep pushing the node’s data into the stack.

**Step2:** Then keep popping the elements out of the stack and updating the doubly Linked List.

**Code:**

*static* *void* **reverseDLL1()**{

*Stack*<*Integer*> stack= new *Stack*<*Integer*>();

*//Step1:*

*Node* temp = head;

        while(temp!=null){

            stack.push(temp.data);

            temp = temp.next;

        }

*//Step2:*

        temp=head;

        while(temp!=null){

            temp.data = stack.pop();

            temp = temp.next;

        }

    }

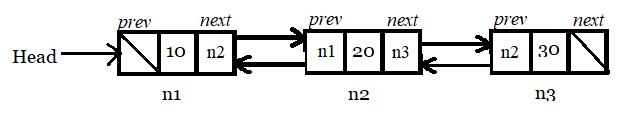
**Time complexity: O(2N),** because we have to traversing linked list twice for pushing & popping.

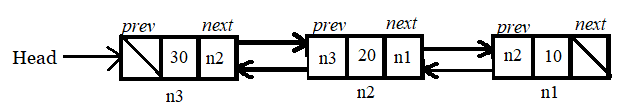
**Space Complexity: O(N),** Stack space.

* **Solution 2: [Efficient] (Reverse the pointers in a single pass)**

Illustrate the working of code by doing dry run on above linked list.

1. Check edge case: If linkedlist is empty i.e. head==null than return.
2. There are two pointers temp and curr pointing to head.
3. temp is used to iterate over the elements of linked list and curr is used to access current node and exchange its next and prev fields. After exchange of fields is done curr will point to next node i.e.temp.
4. Repeat step-2 till second last node i.e. temp.next !=null.
5. et prev field of last node to null and its next field to curr which will be the second last node. Make the last node as head node.





**Code:**

static void **reverseDLL**(){

        // Ede case: If linkedlist is empty or contains only one

        if(head==null || head.next==null){

            return;

        }

        Node temp = null;

        Node curr = head;

        // swap next and prev fields for all nodes of doubly linked list

        while(curr != null){

            temp = curr.prev;

            curr.prev = curr.next;

            curr.next = temp;

            curr = curr.prev;

        }

        if(temp!=null){

            head = temp.prev;

        }

    }

**Output:**

Original LinkedList: 10->20->30->null

Reversed LinkedList: 30->20->10->null

**Time complexity: O(N),** in printing, reversing as well as insertion because we have to traverse all the nodes of linked list.

**Space Complexity: O(1),** we haven’t used any extra space.

1. **Find middle element in a Linked List**

**Problem Statement:** Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

**Program:**

public class MiddleNode {

    static class Node{

        int data;

        Node next;

        Node(int value){

            data = value;

        }

    }

    static Node **insertAtEnd**(Node head, int value){

        Node new\_node = new Node(value);

        //Edge case: If linkedlist is empty, make new\_node as head node

        if(head == null){

            head = new\_node;

            return head;

        }

        Node temp = head;

        while(temp.next!=null){

            temp = temp.next;

        }

        temp.next = new\_node;

        return head;

    }

    static void **printList**(Node head){

        Node temp = head;

        System.out.print(" List contains: ");

        while(temp!=null){

            System.out.print(temp.data + "->");

            temp=temp.next;

        }

        System.out.print("null\n");

    }

    public static void main(String[] args) {

        Node head = null;

        head = insertAtEnd(head, 10); head = insertAtEnd(head, 20);

        head = insertAtEnd(head, 30); head = insertAtEnd(head, 40);

        head = insertAtEnd(head, 50);         head = insertAtEnd(head, 60);

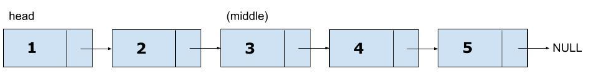
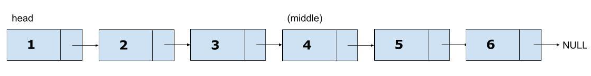
        printList(head);

        Node middle = middleNode2(head);

        System.out.println(" The data of middle node: " + middle.data);

    }

}

****

* **Solution1: Naive approach**

**Intuition:** We can traverse through the Linked List while maintaining a count of nodes let’s say in variable n, and then traversing for 2nd time for n/2 nodes to get to the middle of the list.

**Code:**

*static* *Node* **middleNode1**(*Node* *head*){

*Node* temp = head;

*int* count=0;

        while(temp!=null){

            count++;

            temp = temp.next;

        }

temp=head;

        System.out.println("(count/2): " + (count/2));

        for(*int* i=1; i<=(count/2); i++){

            temp = temp.next;

        }

        return temp;

    }

**Output:**

List contains: 10->20->30->40->50->60->null List contains: 10->20->30->40->50->null

(count/2): (6/2) = 3 (count/2): (5/2) = 2

The data of middle node: 40 The data of middle node: 30

**Time Complexity: O(n) + O(n/2),** 0(n) to count the total no. of nodes and O(n/2) to traverse linkedlist till middle.

**Space Complexity: O(1),** as no extra space is used.

* **Solution 2: [Efficient] Tortoise-Hare-Approach**

**Intuition:** In the Tortoise-Hare approach, we increment slow ptr by 1 and fast ptr by 2, so if take a close look fast ptr will travel double that of the slow pointer. So when the fast ptr will be at the end of the Linked List, slow ptr would have covered half of the Linked List till then. So slow ptr will be pointing towards the middle of Linked List. (ptr means pointer). [DO illustration by hand]

**Code: (*If no. of nodes is even, fast ptr will be pointing to null at the end. Else if no. of nodes is odd fast ptr will be pointing to last node at the end.*)**

static Node **middleNode2**(Node head){

        Node slow = head, fast = head;

        while(fast!=null && fast.next!=null){

            slow = slow.next; // Move one step at a time

            fast = fast.next.next; // Move two steps at a time

        }

        return slow;

    }

**Output:**

List contains: 10->20->30->40->50->60->null

The data of middle node: 40

**Time Complexity: O(n/2)** as when slow will reach at middle i.e.(n/2) the fast has already reached at the end i.e.(n).

**Space Complexity: O(1),** as no extra space is used.

1. **Reverse a Singly Linked List (Refer: gfg for illustration)**

**Problem Statement:** Given the head of a singly linked list, write a program to reverse the linked list, and return the head pointer to the reversed list.

* **Solution1: Naive approach (Reverse the data using stack in double pass)**

**Step1:** Traverse the whole Linked List and keep pushing the node’s data into the stack.

**Step2:** Then keep popping the elements out of the stack and updating the singly Linked List.

**Code:**

static Node **reverseSLL1**(Node head){

Stack<Integer> stack= new Stack<Integer>();

//Step1:

Node temp = head;

while(temp!=null){

stack.push(temp.data);

temp = temp.next;

}

//Step2

temp=head;

while(temp!=null){

temp.data = stack.pop();

temp = temp.next;

}

return head;

}

**Time complexity: O(2N),** because we have to traversing linked list twice for pushing & popping.

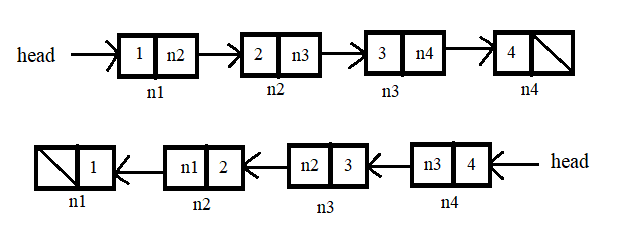
**Space Complexity: O(N),** Stack space.

* **Solution2: Iterative approach**

**Approach:** The idea is to use three pointers curr, prev, and next to keep track of nodes to update reverse links.

***Follow the steps below to solve the problem:***

* Initialize three pointers prev as NULL, curr as head, and next as NULL.
* Iterate through the linked list. In a loop, do the following till **curr!=null**:
  + Before changing the next of curr, store it in next node in order to preserve it
    - **next = curr -> next**
  + Now update the next pointer of curr to the prev in order to reverse the link
    - **curr -> next = prev**
  + Update prev as curr and curr as next in order to move towards the next node
    - **prev = curr**
    - **curr = next**



**Code:**

static Node **reverseSLL**(Node head){

Node prev = null, next=null;

Node curr = head;

while(curr != null){

next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

return prev; // returns new head which will be the last node

}

**Time Complexity: O(N),** Traversing over the linked list of size N**.**

**Space Complexity: O(1)**

* **Solution3: Recursive approach (See Dry run of the code)**

**Approach:** The idea is to reach the last node of the linked list using recursion then start reversing the linked list.

* We traverse till the end of the linked list recursively.
* When we reach at the of the linked list, we’ll make the last node the head.

**Code:**

static Node **reverseSLL3**(Node head){

if(head==null || head.next==null){

return head;

}

Node newHead = **reverseSLL3**(head.next);

Node front = head.next;

front.next = head;

head.next = null;

return newHead;

}

**Output:** Original LinkedList: 10->20->30->40->null

Reversed LinkedList: 40->30->20->10->null

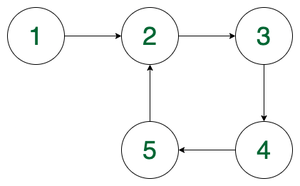
**Time Complexity: O(N)**

**Space Complexity: O(N),** Recursive stack space for function call made.

1. **Detect a loop in Singly Linked List**

**Problem Statement:** Given the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.

Return true if there is a cycle in the linked list. Otherwise, return false.



Here, we can see that we can reach node at position 2 again by following the next pointer. Thus, we return true for this case.

static Node **createCycle**(Node head, int pos){

Node temp = head, curr = head;

// Traverse till last node

while(temp.next != null){

temp = temp.next;

}

// Traverse till node you want to create the starting point of the loop

while(pos>1){

curr = curr.next;

pos--;

}

temp.next = curr;

return head;

}

* **Solution1: Using Hashing (***The idea is to insert the nodes in the hashset and whenever a node is encountered that is already present in the hashset then return true****.*)**

**Follow the steps below to solve the problem:**

* Traverse the list individually and keep putting the node addresses in a Hash Table.
* If the current node is present in the hash table already, this indicates the cycle is present in the linked list and returns true.
* Else move insert the node in the hash table and move ahead.
* If we reach the end of the list, which is NULL, then we can say that the given list does not have a cycle in it and hence we return false.

**Code:**

static boolean **detectLoop1**(Node head){

Node temp = head;

HashSet<Node> set = new HashSet<Node>();

while(temp != null){

if(set.contains(temp)){

return true;

}

set.add(temp);

temp = temp.next;

}

return false;

}

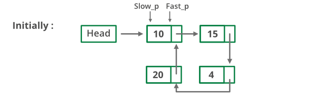
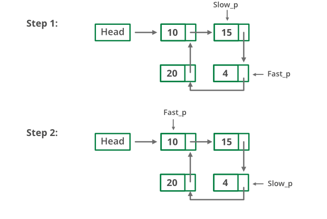
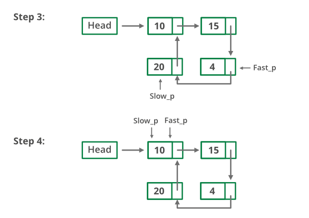
**Time complexity: O(N),** Only one traversal of the loop is needed.

**Auxiliary Space: O(N**), N is the space required to store the value in the hashmap.

* **Solution2: Floyd’s Cycle-Finding Algorithm or Hare-Tortoise approach (***This algorithm is used to find a loop in a linked list. It uses two pointers one moving twice as fast as the other one. The faster one is called the fast pointer and the other one is called the slow pointer.***)**

**Follow the steps below to solve the problem:**

* Traverse linked list using two pointers.
* Move one pointer(slow) by one and another pointer(fast) by two.
* If these pointers meet at the same node then there is a loop. If pointers do not meet then the linked list doesn’t have a loop.

****

**Code:**

static boolean **detectLoop2**(Node head){

Node slow = head, fast = head;

while(fast!=null && fast.next!=null){

slow = slow.next;

fast = fast.next.next;

if(slow == fast)

return true;

}

return false;

}

**Time Complexity:** O(N) *Reason*: In the worst case, all the nodes of the list are visited.

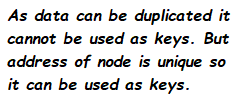
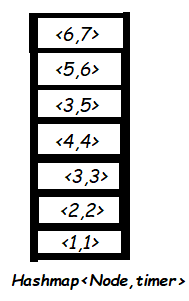
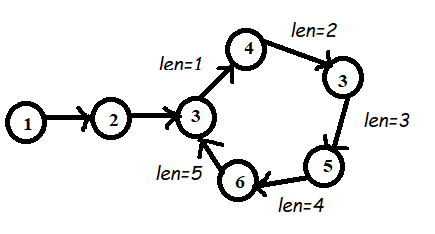
**Space Complexity:** O(1) *Reason*: No extra data structure is used.

1. **Find the length of a loop in Singly Linked List**

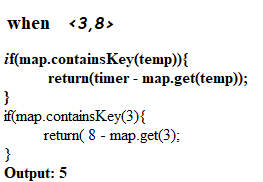
**Problem Statement:** Given the head of a linked list. If a loop exists in the linked list then return **the length of the loop** else return 0.

* **Solution1: Using Hashing**

**Approach***: The idea is to insert the node and timer associated with it in the hashmap and whenever a node is encountered that is already present in the hashmap then return timer-Node’s timer else return 0****.***

****

**Code:**static int lengthOfLoop1(Node head){

 Node temp = head;

int timer=1;

HashMap<Node,Integer> map = new HashMap<>();

while(temp != null){

if(map.containsKey(temp)){

return(timer - map.get(temp));

}

map.put(temp,timer);

timer++;

temp = temp.next;

}

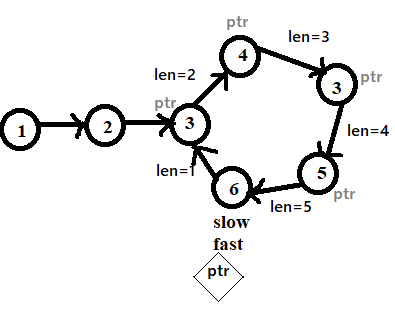
return 0;

}

**Time complexity: O(N),** Only one traversal of the loop is needed.

**Auxiliary Space: O(N**), N is the space required to store the value in the hashmap.

* **Solution2: Floyd’s Cycle-Finding Algorithm or Hare-Tortoise approach (***This algorithm is used to find a loop in a linked list. It uses two pointers one moving twice as fast as the other one. The faster one is called the fast pointer and the other one is called the slow pointer.***)**
* Floyd’s Cycle detection algorithm terminates when fast and slow pointers meet at a common point. It is also known that this common point is one of the loop nodes.
* Store the address of this common point in a pointer variable ptr. Then initialize a counter with 1 and start from the common point and keeps on visiting the next node and increasing the counter till the common pointer is reached again. At that point, the value of the counter will be equal to the length of the loop.



**Code:**static int **findLength**(Node slow){

int len=1;

Node ptr = slow.next;

while(ptr != slow){

len++;

ptr=ptr.next;

}

return len;

}

static int **lengthOfLoop2**(Node head){

Node slow = head, fast = head;

while(fast!=null && fast.next!=null){

slow = slow.next;

fast = fast.next.next;

if(slow == fast)

return findLength(slow);

}

return 0;

}

**Output:**

List contains: 1->2->3->4->3->5->6->null

List containing cycle: 1->2->3->4->3->5->6->3->4->3->5->6->

Length of list: 5

**Time complexity: O(N),** Only one traversal of the loop is needed.

**Auxiliary Space: O(1**) as no extra space is used.

1. **Find the starting point of the cycle in the linked list.**

**Problem Statement:**Given the head of a [linked list](https://takeuforward.org/linked-list/linked-list-introduction/), return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. (Internally, pos is used to denote the index of the node that the tail’s next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.)

* **Solution 1: Brute Force (Using Hashmap)**

**Approach:**

We can store nodes in a hash table so that, if a loop exists, the temp will encounter the same node again. This node will be present in the table and hence, we can detect the loop. (*Do not store data. Instead store the entire node. As there can be duplicate data. But duplicate nodes can’t be possible as address are always unique. Node can be a key as it is unique but data can*)

**The steps are:**

* Iterate the given list.
* For each node visited by the temp pointer, check if the node is present in the hash table.
* If yes, the loop detected and the node already present is the starting point of the cycle.
* If not, insert the node in the hash table and move the temp pointer ahead.
* If the temp reaches null, then the given list does not have a cycle in it.

**Code:**

static Node startingPoint(Node head){

Node temp = head;

HashMap<Node,Integer> map = new HashMap<>();

while(temp!=null){

if(map.containsKey(temp)){

return temp;

}

map.put(temp,1);

temp=temp.next;

}

return null;

}

**Output:**

List contains: 1->2->3->4->3->5->6->null

No cycle detected.

List containing cycle: 1->2->3->4->3->5->6->3->4->3->5->6->

Starting point of the cycle: 3

**Time complexity: O(N\*2\*fun),** Only one traversal of the loop is needed.

* **fun:** Time complexity of containsKey() & put() i.e. finding and inserting function depends on programming language used.

**Auxiliary Space: O(N**), N is the space required to store the value in the hashmap.

* **Solution2: Floyd’s Cycle-Finding Algorithm or Hare-Tortoise approach**

**Approach** (see Dry run in strive a to z sheet)

The following steps are required:

* Initially take two pointers, fast and slow. The fast pointer takes two steps ahead while the slow pointer will take a single step ahead for each iteration.
* We know that if a cycle exists, fast and slow pointers will collide.
* If the cycle does not exist, the fast pointer will move to NULL
* Else, when both slow and fast pointer collides, it detects a cycle exists.
* Take another pointer, say entry. Point to the very first of the linked list.
* Move the slow and the entry pointer ahead by single steps until they collide.
* Once they collide, we get the starting node of the linked list.

**Code:**

static Node **startingPoint2**(Node head){

Node fast=head, slow=head;

while(fast!=null && fast.next!=null){

slow = slow.next;

fast = fast.next.next;

if(fast==slow){

Node entry = head;

while(slow!=entry){

slow = slow.next;

entry = entry.next;

}

return entry;

}

}

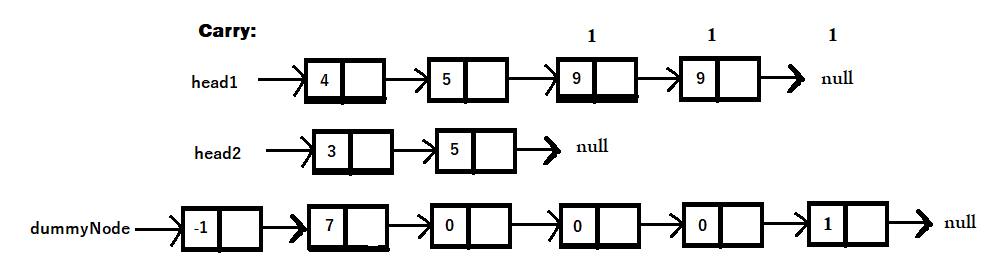
return null;

}

**Time complexity: O(N),** Only one traversal of the loop is needed.

**Auxiliary Space: O(1**) as no extra space is used.

1. **Add two numbers represented as Linked Lists**

**Problem Statement:**  Given the**heads** of two non-empty linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the **sum** as a linked list.

* **Solution1: Elementary Math**
* Each node contains a single digit and the digits are stored in reverse order.
* Just like how you would sum two numbers on a piece of paper, we begin by summing the least significant digits, which is the head of l1 and l2.
* Since each digit is in the range of 0…9, summing two digits may “overflow”. For example, 5 + 7 = 12. In this case, we set the current digit to 2 and bring over the carry=1 to the next iteration.
* carry must be either 0 or 1 because the largest possible sum of two digits (including the carry) is 9 + 9 + 1 = 19.
* ***Note****: that we use a dummy head to simplify the code. Without a dummy head, you would have to write extra conditional statements to initialize the head’s value. Whenever you need to create a new list where you need to store result, always prefer the concept of dummy node.*
* Take extra caution in the following cases:

**Test case** **Explanation**

l1=[0,1], l2=[0,1,2] When one list is longer than the other.

l1=[], l2=[0,1] When one list is null, which means an empty list.

l1=[9,9], l2=[1] The sum could have an extra carry of 1 at the end, which is easy to forget.

**Code:**

*static* *Node* **addNumbers**(*Node* *head1*, *Node* *head2*){

*Node* temp1=head1, temp2=head2;

*Node* dummyNode = new Node(-1);

*Node* curr=dummyNode;

*int* carry = 0;

        while(temp1!=null || temp2!=null){

*int* sum=carry;

            if(temp1!=null)

                sum += temp1.data;

            if(temp2!=null)

                sum += temp2.data;

*Node* newNode = new Node(sum % 10); *//Last digit*

            carry = sum / 10; *//First digit*

            curr.next = newNode;

            curr = curr.next;

            if(temp1!=null) temp1 = temp1.next;

            if(temp2!=null) temp2 = temp2.next;

        }

        if(carry!=0){

*Node* newNode = new Node(carry);

            curr.next = newNode;

        }

        return dummyNode.next;

    }

**Output:**

**List1:** 4->5->9->9->null

**List2:** 3->5->null

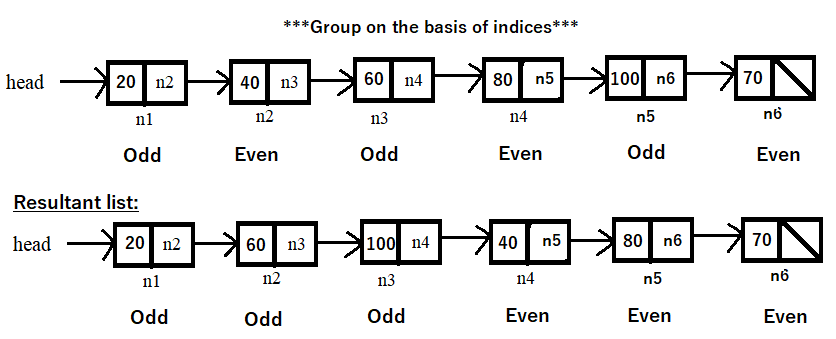
**Sum List:** 7->0->0->0->1->null

**Time Complexity: O(max(m,n)).** Assume that m and n represent the length of l1 and l2 respectively, the algorithm above iterates at most max(m,n) times.

**Space Complexity: O(max(m,n)).** The length of the new list is at most max(m,n)+1. (Extra 1 because of carry.)

1. **Segregate Odd and even nodes in linked list**

**Problem Statement:** Given the head of the linked list, the task is to group all the odd indexed nodes together and group all the even indexed nodes together. Connect the group of Odd indexed nodes come with even indexed nodes. Modify the original linked list.



* **Solution1: Using array data structure**
* Traverse the linked list from head (first node), put odd-indexed node’s data in the array and jump it by two places.
* Traverse the linked list from head.next (second node), put even-indexed node’s data in the array and jump it by two places.
* **Replace the data** in the original linked list with the data from the array.

**Code:**

*static* *Node* **segregateOddEven**(*Node* *head*){

*/\*Edge case: If list is empty or has single node \*/*

            if(head==null || head.next==null){

                return head;

            }

*ArrayList*<*Integer*> arr= new *ArrayList*<>();

*/\* Step1: Segregating odd-indexed node* ***-* O(n/2) *\*/***

*Node* temp = head;

            while(temp!=null && temp.next!=null){

                arr.add(temp.data);

                temp = temp.next.next;

            }

*// If array is odd-length, the condition temp.next!=null tends to skip the last node*

            if(temp!=null){

                arr.add(temp.data);

            }

*/\* Step2: Segregating even-indexed node* ***-* O(n/2)** *\*/*

            temp = head.next;

            while(temp!=null && temp.next!=null){

                arr.add(temp.data);

                temp = temp.next.next;

            }

*// If array is even-length, the condition temp.next!=null tends to skip the last node*

            if(temp!=null){

                arr.add(temp.data);

            }

*/\* Step3: Replacing data* ***-* O(n)** *\*/*

            temp=head;

*int* i=0;

            while(temp!=null){

                temp.data = arr.get(i);

                temp = temp.next;

                i+=1;

            }

            return head;

        }

**Output:**

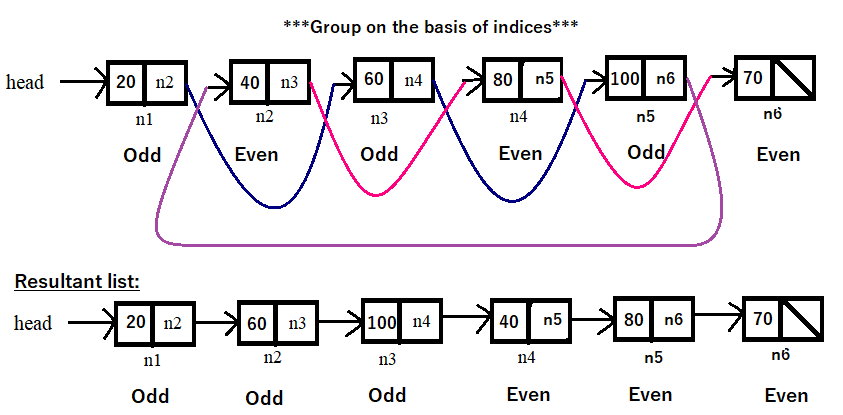
Original list: 20->40->60->80->100->70->null

Segregated list: 20->60->100->40->80->70->null

**Time Complexity: O(2n),** O(n/2)+O(n/2)+O(n) =O( 2n)

**Space Complexity: O(n),** using array to store the n elements.

* **Solution2: By changing links**
* Starting with the head(first node), connect the links for odd-indexed nodes by changing the links.
* Starting with the head.next(second node), connect the links for even-indexed nodes by changing the links.
* Connect the last odd-indexed node to the first even-indexed node.

****

**Note:**

* The **even pointer** will always be ahead of **odd pointer**.
* Before starting the link changes, memorize head.next in evenHead in order to preserve the heads of odd and even group nodes. So that you can connect the last odd-indexed node to the first even-indexed node i.e.evenHead. Return the head of modified segregated list.

**Code:**

static Node **segregateOddEven2**(Node head){

/\*Edge case: If list is empty or has single node \*/

if(head==null || head.next==null){

return head;

}

Node odd=head, even=head.next, evenHead=head.next;

/\*Condition: we need to check for even pointer because even ptr will be ahead of odd ptr. So if even ptr has not reached the end odd will also have not reached the end. \*/

while(even!=null && even.next!=null){

odd.next = odd.next.next;

even.next = even.next.next;

odd=odd.next;

even=even.next;

}

odd.next = evenHead;

return head;

}

**Output: (**same as before**)**

**Time Complexity: O(n),** we are doing two link changing operations which takes O(n/2) times. So total time taken 2 \* O(n/2) = O(n). If we do the two operations in separate while loops there’s no greater difference in time. But doing it in single while loop is slightly faster than doing it in two while loops.

**Space Complexity: O(1),** as we are not using any extra space.

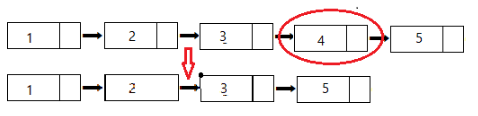
1. **Remove Nth node from the linked list**

**Problem Statement:** Given a linked list, and a number N. Find the Nth node from the end of this linked list and delete it. Return the head of the new modified linked list.

**Example:**

**Input**: head = [1,2,3,4,5], n = 2

**Output**: [1,2,3,5]



* **Solution1: Naïve Approach (Traverse 2 times)**

**Intuition:** We can [traverse through the Linked List](https://takeuforward.org/data-structure/linked-list-traversal/) while maintaining a count of nodes, let’s say in the variable count, and then traversing for the 2nd time for (N – ***count***) nodes to get to the nth node of the list.

**Code:**

static int **countNode**(Node head){

int count=0;

Node temp=head;

while(temp!=null){

count++;

temp = temp.next;

}

return count;

}

static Node **removeNode**(Node head, int N){

if(head==null || head.next==null){

return null;

}

int count = countNode(head);

if(count==N){

return head.next;

}

int res = count-N;

Node temp = head;

while(res>1){

temp = temp.next;

res--;

}

temp.next = temp.next.next;

return head;

}

**main() function:**

int N=2;

// valid value of n:

**if(N>=1 && N<=countNode(head))** {

System.out.print("Original List: ");

printList1(head);

System.out.println("After deleting " + N + "th node from the back");

System.out.print("Modified List: ");

head=removeNode(head,n);

printList1(head);

**}** else **{**

System.out.print("Invalid value of N");

**}**

**Output:**

Original List: 1->2->3->4->5->null

After deleting 2th node from the back

Modified List: 1->2->3->5->null

**Time Complexity: O(count)+O(count-N),**

* **O(count)** for counting the length of the list.
* **O(count-N)** in order to reach the previous node of the Nth node- the node to be deleted.

**Space Complexity: O(1),** no extra space.

**Worst case Time complexity:** O(2\*count). When N=1 O(count-N) is equivalent to O(count). This is the reason, you need an optimise solution.

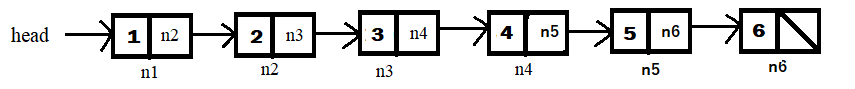
* **Solution2: Two Pointer Approach**

**Intuition:**

What if we had to modify the same above approach to work in just one traversal? Let’s see what all information we have here:

* We have the flexibility of using two-pointers now.
* We know, that the n-th node from the end is the same as (total nodes – n)th node from start.
* But we are not allowed to calculate total nodes, as we can do only one traversal.
* What if, one out of the two-pointers is at the nth node from start instead of the end? Will it make anything easier for us?

Yes, with two pointers in hand out of which one is at the n-th node from start, we can just advance both of them till the end simultaneously, once the faster reaches the end, the slower will stand at the nth node from the end.



**Code: (Perform dry run of code in above LL.)**

static Node **removeNode2**(Node head, int n){

/\*Edge case: If list is empty or has single node \*/

if(head==null || head.next==null)

return null;

Node fast=head, slow=head;

/\* Step1: Move fast pointer to (n+1)th node \*/

for(int i=1; i<=n; i++){

fast=fast.next;

}

/\* when n = length of LL \*/

if(fast==null){

return head.next;

}

/\* Step2: Move fast and slow pointer on step till fast reaches the last node. At the end of this step slow pointer will be at node before the node to be deleted. \*/

while(fast.next!=null){

fast=fast.next;

slow=slow.next;

}

/\* Step3: Delete the node by updating links \*/

slow.next = slow.next.next;

return head;

}

**Output:**

Original List: 1->2->3->4->5->6->null

After deleting 2th node from the back

1->2->3->4->6->null

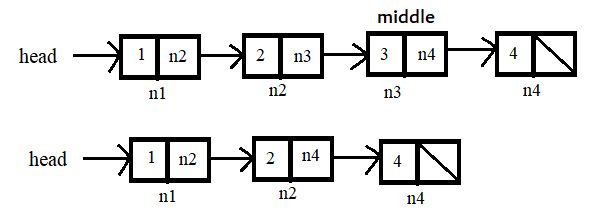
**Time Complexity: O(n),** combining both the loops, the fast pointer is traversing the entire list once.

**Space Complexity: O(1),** as no extra space is used.

1. **Delete the middle node of Linked list**

**Problem Statement:** Given the head of linked list, your task is to delete the middle node of this list and return the head of the modified list.

**Example: Input:** 1->2->3->4 **Input:** 1->2->3->4->5

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

* **Solution1: Brute Force approach**

**Code:**

static Node **deleteMiddleNode**(Node head){

/\* Edge case: If list is empty or has only single node \*/

if(head==null || head.next==null){

return null;

}

Node temp = head;

int count=0;

/\*Step1: Count the nodes – TC: O(n) \*/

while(temp!=null){

count++;

temp=temp.next;

}

/\*Step2: Traverse till node before the middle node – TC: O(n/2) \*/

temp=head;

for(int i=1; i<(count/2); i++){

temp=temp.next;

}

/\*Step3: Link the previous node to the node after the middle node and delete the midddle node \*/

temp.next = temp.next.next;

return head;

}

**Output:** Original list: 1->2->3->4->null

Modified list: 1->2->4->null

**Time Complexity: O(n)+O(n/2)**

**Space Complexity: O(1),** as no extra space is used.

* **Solution2: [Efficient] Tortoise-Hare-Approach with slight modification**

**Intuition:** In the Tortoise-Hare approach, we increment slow ptr by 1 and fast ptr by 2, so if take a close look fast ptr will travel double that of the slow pointer. So when the fast ptr will be at the end of the Linked List, slow ptr would have covered half of the Linked List till then. So slow ptr will be pointing towards the middle of Linked List. (ptr means pointer). [DO illustration by hand]

**Code: (*If no. of nodes is even, fast ptr will be pointing to null at the end. Else if no. of nodes is odd fast ptr will be pointing to last node at the end.*)**

static Node deleteMiddleNode2(Node head){

/\* Edge case: If list is empty or has only single node \*/

if(head==null || head.next==null){

return null;

}

/\*As we want the node before the middle node, we will skip one step of slow ptr by initializing fast ptr with third node ahead instead of head node.\*/

Node slow=head, fast=head.next.next;

while(fast!=null && fast.next!=null){

slow=slow.next;

fast=fast.next.next;

}

/\*Step3: Link the previous node to the node after the middle node \*/

slow.next=slow.next.next;

return head;

}

**Output:** Original list: 1->2->3->4->null

Modified list: 1->2->4->null

**Time Complexity: O(n/2),** as we traversing only till the half of the list.

**Space Complexity: O(1),** as no extra space is used.

1. **Sort Linked List**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

1. **Sort a Linked List of 0’s, 1’s and 2’**

**Problem Statement:** Given a linked list of ‘N’ nodes, where each node has an integer value that can be 0, 1 or 2. You need to sort the linked list in non-decreasing order and return the head of the sorted list.

**Input**: 1->0->2->1->0->2->1

**Output:** 0->0->1->1->1->2->2

**Explanation:** In this example, the original linked list contains two 0s, three 1s and two 2s. The sorted linked list has all the 0s at the beginning, followed by all the 1s, and finally, all the 2s at the end.

* **Solution1: Brute Force Approach – Data Replacement (Two travsersal)**

**Step1:** Take 3 count variables and store the no. of 0s, 1s and 2s in it by traversing the entire linked list. **O(n)**

**Step2:** Traverse the linked list and replace the data with 0s, 1s and 2s in ascending order till count variables become 0. **O(n)**

**Code:**

static Node **sortNodes1**(Node head){

Node temp=head;

/\*Step1: storing the no. of 0s, 1s and 2s\*/

int cnt0=0, cnt1=0, cnt2=0;

while(temp!=null){

if(temp.data==0)

cnt0++;

else if(temp.data==1)

cnt1++;

else

cnt2++;

temp=temp.next;

}

/\*Step2: Replacing data \*/

temp=head;

while(temp!=null){

if(cnt0!=0){

temp.data=0;

cnt0--;

}else if(cnt1!=0){

temp.data=1;

cnt1--;

}else{

temp.data=2;

cnt2--;

}

temp=temp.next;

}

return head;

}

**Output:**

Original List: 1->0->1->2->0->2->1->null

Sorted List: 0->0->1->1->1->2->2->null

**Time Complexity: O(2n),** as we aretraversing the list twice.

**Space Complexity: O(1),** as no extra space is used.

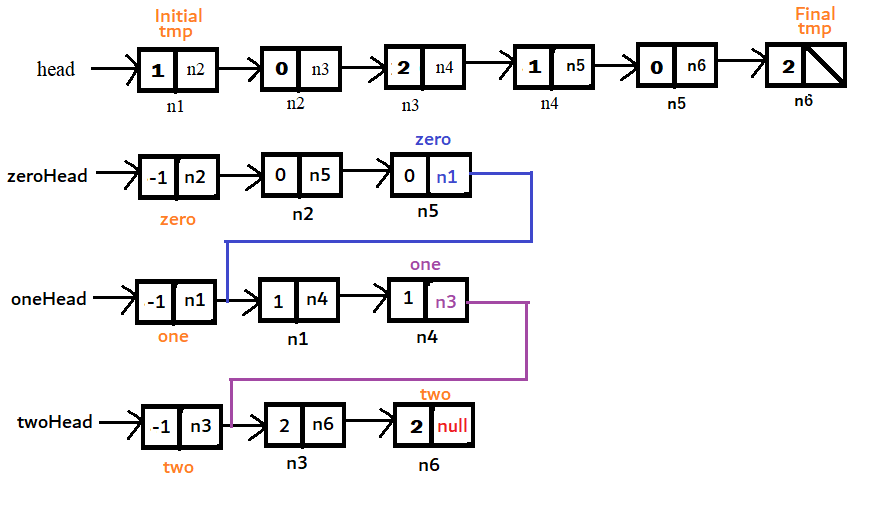
* **Solution2: Optimal Approach –Changing links (one traversal)**

**Step1:** Take 3 dummyNodes: zeroHead, oneHead and twoHead in order to preserve the beginning of each group which is required for connecting list. Take 3 nodes zero, one and two. Let these three nodes point to the dummyNodes.

**Step2:** Traverse the list starting from the head. Change the links according to data 0s, 1s and 2s. Let zero ptr points to the node with data 0. Let one ptr points to the node with data 1. Let two ptr points to the node with data 2.

**Note:** **We are not creating new nodes, but changing the links such that the pointers zero, one and two points to the nodes of original list on the basis of data 0s, 1s and 2s.**

**Step3:** Connect the last node of each group with the beginning of the next group such that the sorted linked list has all the 0s at the beginning, followed by all the 1s, and finally, all the 2s at the end.

****

**Code:**

static Node sortNodes2(Node head){

/\* Edge case: If list is empty or has one node \*/

if(head==null || head.next==null){

return head;

}

/\* Three Dummy Nodes \*/

Node zeroHead = new Node(-1);

Node oneHead = new Node(-1);

Node twoHead = new Node(-1);

Node zero = zeroHead;

Node one = oneHead;

Node two = twoHead;

Node temp=head;

while(temp!=null){

if(temp.data==0){

zero.next=temp;

zero=temp;

}else if(temp.data==1){

one.next=temp;

one=temp;

}else{

two.next=temp;

two=temp;

}

temp=temp.next;

}

zero.next = (oneHead.next!=null) ? oneHead.next : twoHead.next;

one.next = twoHead.next;

two.next = null;

Node newHead = zeroHead.next;

return newHead;

}

**Output:** Original List: 1->0->1->2->0->2->1->null

Sorted List: 0->0->1->1->1->2->2->null

**Time Complexity: O(n),** as only one traversal.

**Space Complexity: O(1),** as no extra space is used

1. **Check if Linked List is palindrome or not**

**Problem Statement:** Given the head of a singly linked list, return true if it is a palindrome.

**Example 1:**

**Input:** head = [1,2,3,3,2,1]

**Output:** true

**Explanation:** If we read elements from left to right, we get [1,2,3,3,2,1]. When we read elements from right to left, we get [1,2,3,3,2,1]. Both ways list remains same and hence, the given linked list is palindrome.

* **Solution1: Brute Force Approach (Using Stack data structure)**

**Step1: Storing:** Traversethe linked list starting from head and store the data in stack. As we know that stack based on LIFO principle. The item we stored the last is the first one to be accessed i.e.**in reverse manner**.

**Step2:** **Comparing:** Traverse the linked list from starting head and compare the data stored in stack with the linked list’s data. If data does not match return false. If the stack data matches for the entire linked list then return true.

**Code:**

static boolean **isPalindrome**(Node head){

if(head==null || head.next==null){

return true;

}

Stack<Integer> stack = new Stack<>();

Node temp = head;

while(temp!=null){

stack.push(temp.data);

temp = temp.next;

}

temp=head;

while(temp!=null){

if(temp.data!=stack.peek()){

return false;

}

stack.pop();

temp=temp.next;

}

return true;

}

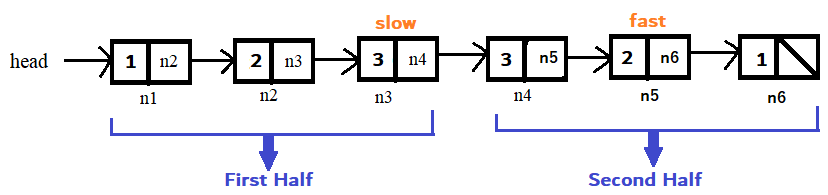
**Output:** Original List: 1->2->3->2->1->null

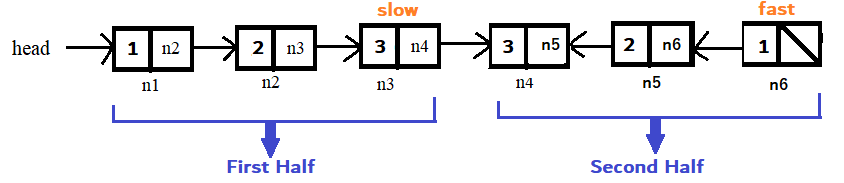
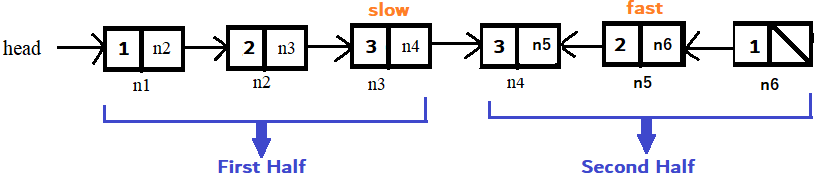
Yes, Linked list is palindrome!

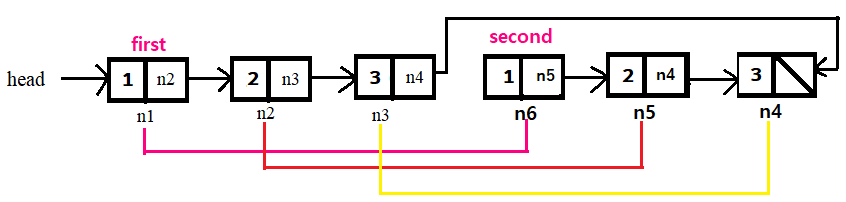
**Time Complexity: O(2n),** for storing and comparing.

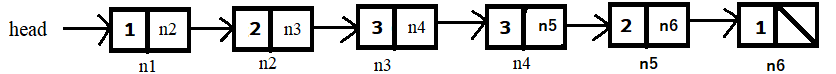
**Space Complexity: O(n),** storing data in stack data structure.

* **Solution2: Efficient Approach**

**Step1:** Find out the middle node of the linked list (m1 in case of even length list) using hare-tortoise algorithm. **TC: O(n/2)**

**Step2:** Reverse the second half of the linked list. **TC: O(n/2)**

**Step3:** Comparing the nodes of first and second half. **TC: O(n/2)**

**Step4:** Re-reversing the second half because we are not supposed to alter the input original linked list. So, if we are altering the data make sure to restore the original data back. **TC: O(n/2)**

**Code:**

static Node **reverse**(Node head)**{**

if(head==null || head.next==null){

return head;

}

Node newHead = reverse(head.next);

Node front = head.next;

front.next = head;

head.next = null;

return newHead;

**}**

static boolean **isPalindrome2**(Node head)**{**

Node slow=head, fast=head;

while(fast.next.next!=null && fast.next!=null){

slow=slow.next;

fast=fast.next;

}

Node newHead = reverse(slow.next);

Node first = head;

Node second = newHead;

while(second!=null){

if(first.data!=second.data){

reverse(newHead);

return false;

}

first = first.next;

second = second.next;

}

reverse(newHead);

return true;

**}**

**Output:** Original List: 1->2->3->2->1->null

Yes, Linked list is palindrome!

**Time Complexity: O(2n)**

* **O(n/2):** For finding the middle node of the linked list.
* **O(n/2) + O(n/2) = O(n):** For reversing the second half of the list before and after comparing the nodes in steps 2 and 4.
* **O(n/2):** For comparing the data of nodes in the fast half with the second half.

**Space Complexity: O(n),** as no extra space is used.

1. **Find the intersection point of the Linked List**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

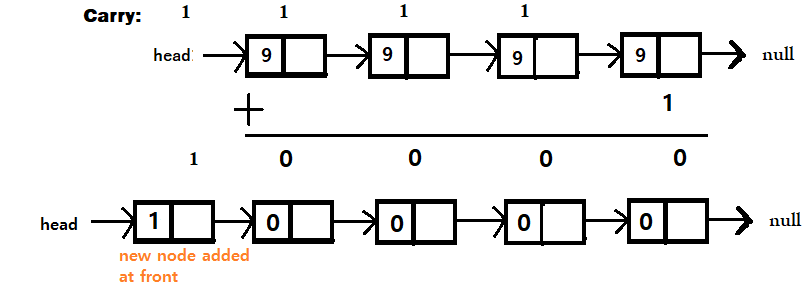
**Output:**

**Time Complexity:**

**Space Complexity:**

1. **Add 1 to a number represented by Linked List**

**Problem Statement:** Given a positive integer represented in the form of singly linked list of digits. The length of the numbers is ‘n’. Add 1 to the number, i.e., increment the given number by one. The digits are stored such that the most significant digit is at the head of the linked list and the least significant digit is at the tail of the linked list.

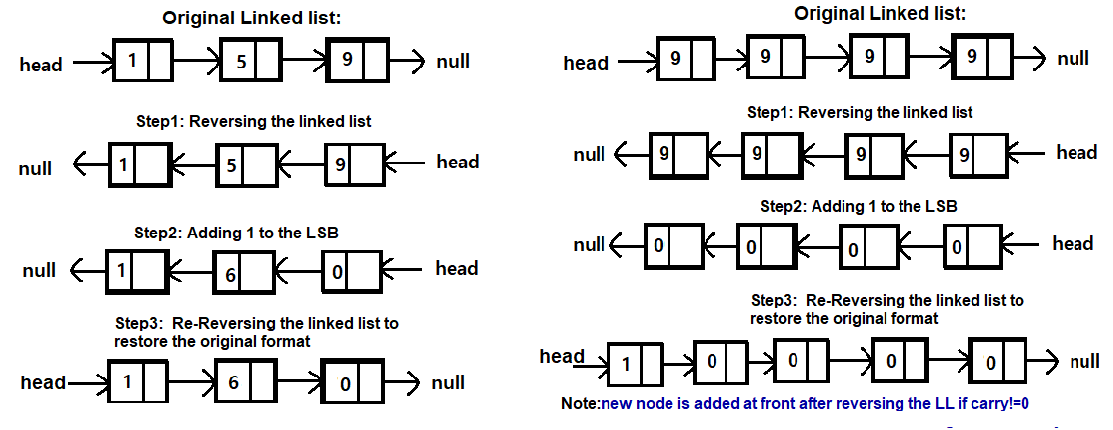
**Example:** **Original linked list:** 1->5->9 **Modified linked list:** 1->6->0

* **Solution1: Brute Force Approach**

**Step1: Reverse the linked list**, as we need to add 1 to the LSB which is stored at the tail and carry forward the generated **carry** to the next significant bit but backward traversal is not possible in singly linked list. **O(n)**

**Step2: Adding 1 to the LSB,** after we reverse the linked list LSB will be at the head. So add 1 to the data of head node and add carry to the data of next node if **carry!=0**. If carry==0 break the loop as there is nothing to increment. **O(n)**

**Step3: Re-Reversing the linked list,** in order to restore the original format i.e. MSB at head and LSB at tail. **O(n)**

**Step4: Adding new node:** In caseIf carry!=0, than add new node with data as carry in the beginning of the list.

**Code:**

static Node **addOneToNumber**(Node head){

/\* Edge case: If list is empty or has only single node \*/

if(head==null || head.next==null){

return null;

}

head = reverse(head);

Node temp=head;

int carry=1; // As we need to add 1 to a number take initial carry as 1

while(temp!=null){

int sum = temp.data + carry;

temp.data = sum % 10;

carry = sum / 10;

if(carry==0)

break;

temp=temp.next;

}

head=reverse(head);

if(carry!=0){

Node newNode = new Node(carry);

newNode.next = head;

return newNode;

}

return head;

}

**Output:** Original Linked List: 9->9->9->9->null

Modified Linked List: 1->0->0->0->0->null

**Time Complexity: O(3n),** reversing, traversing for adding and re-reversing.

**Space Complexity: O(1)**  no extra space is used.

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

1. **R**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

1. **R**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

1. **R**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

1. **R**

**Problem Statement:** Given

* **Solution1:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**

* **Solution2:**

**Code:**

**Output:**

**Time Complexity:**

**Space Complexity:**