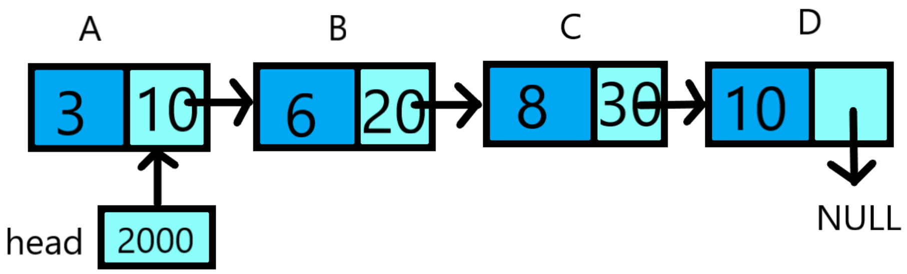
Reverse a Linked List

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*.

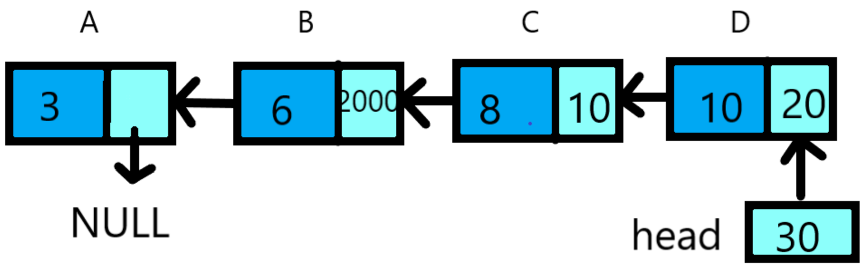
Examples:

```
Input Format:
head = [3,6,8,10]
This means the given linked list is 3->6->8->10 with head pointer at node 3.

Result:
Output = [10,6,8,3]
This means, after reversal, the list should be 10->6->8->3 with the head pointer at node 10.
```

Explanation:



This is how a linked list looks for a given input. the *head* is a reference that points to the initial or starting node of the list. To reverse it, we need to invert linking between nodes. That is, node D should point to node C, node C to node B and node B to node A. Then *head* points to node D and node A has *NULL*.



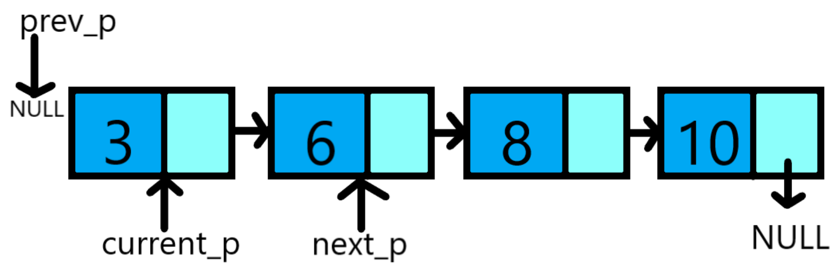
```
Input Format:
head = []
This means the given linked list is empty.

Result:
Output = []

Explanation:
The linked list is empty. That is, no nodes are present in the list. Thus, even reversing will
give the list as empty.
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Reverse Linked List : Iterative

We will use three-pointers to traverse through the entire list and interchange links between nodes. One pointer to keep track of the current node in the list. The second one is to keep track of the previous node to the current node and change links. Lastly, a pointer to keep track of nodes in front of current nodes.



STEP 1:

- currrent_p is a pointer to keep track of current nodes. Set it to head.
- prev_p is a pointer to keep track of previous nodes. Set it to NULL.
- next_p is a pointer to keep track of next nodes.

STEP 2:

- Set next_p to point next node to node pointed by current_p.
- Change link between nodes pointed by current_p and prev_p.
- Update prev_p to current_p and current_p pointer to next_p.

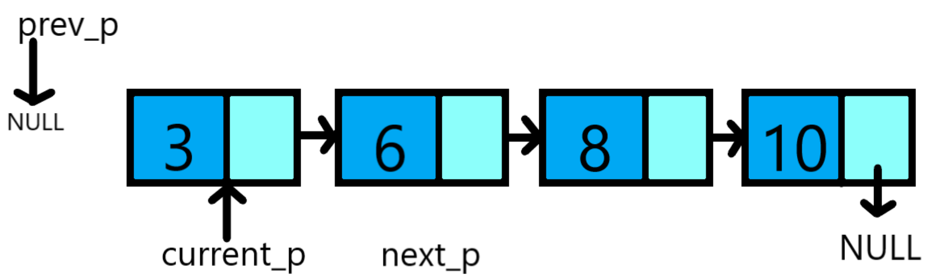
Perform STEP 2 until current_p reaches the end of the list.

STEP 3:

Set head as prev_p. This makes the head point at the last node.

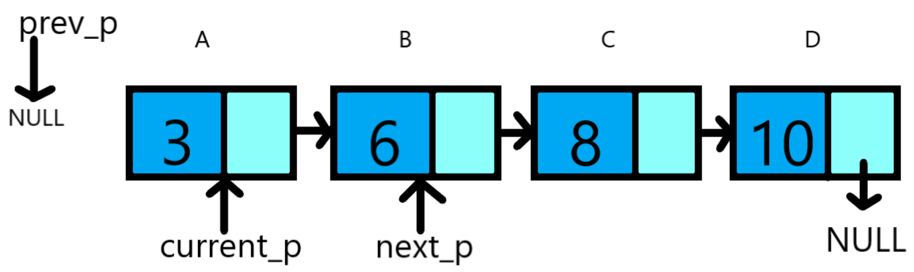
Dry Run:

At beginning,



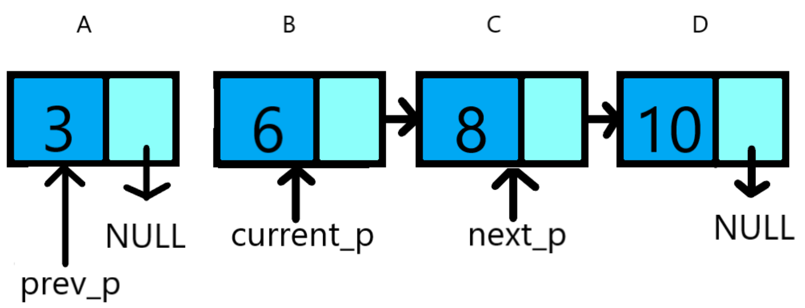
As per this example, current_p points node with 3 and prev_p points to NULL

Now, we start iterating throughout the list until current_p has a value equal to NULL. Set next_p to the next node of the current_p pointed node present in the list.

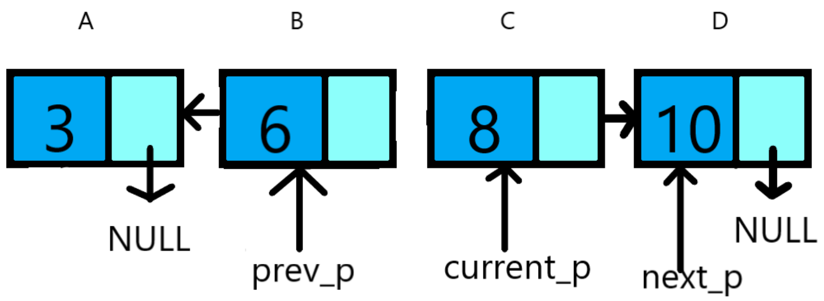


Since the next node to current_p pointed node is B. Therefore next_p points to node B.

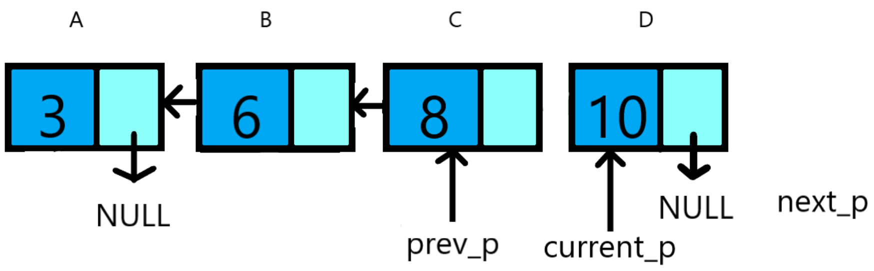
Now, interchange linking between nodes pointed by prev_p and next_p,i.e, node A and NULL. Then, prev_p moves to node A, current_p moves to B, and next_p to C.



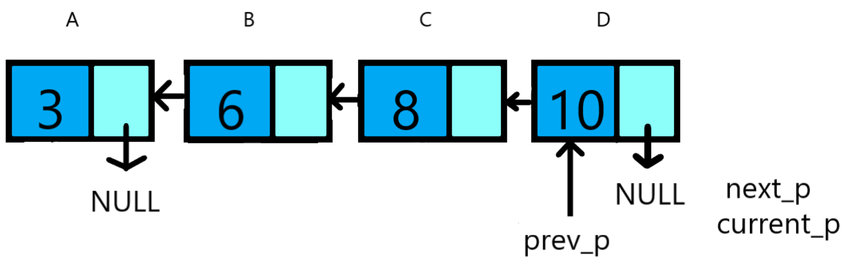
Thus, at the first iteration, we pointed node A from node B to NULL.
In the second iteration, we point node B from node C to node A. Move next_p to node D, current_p to node C and prev_p to node A. This is performed by the same steps.



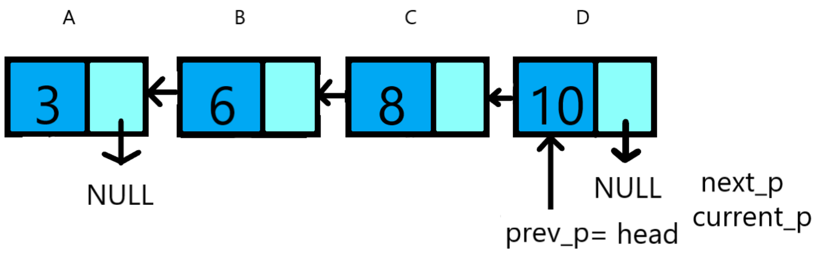
In the third iteration, we point node C from node D to node B. Move next_p to out of list, current_p to node D and prev_p to node C.



In the fourth iteration, we point node D from NULL to node C. Move current_p out of list and prev_p to node D.



Since current_p value is equal to NULL so iteration stops. We set head as prev_p.

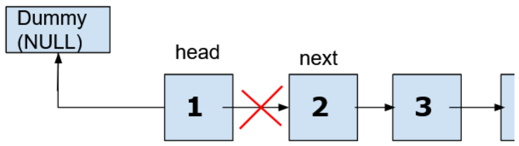


Hence, we achieved our reversed list.

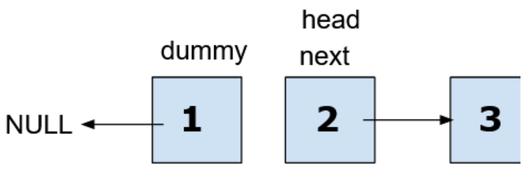
Code:
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
       //step 1
        ListNode* prev_p = NULL;
        ListNode* current_p = head;
        ListNode* next_p;
       //step 2
        while(current_p) {
            next_p = current_p->next;
            current_p->next = prev_p;

            prev_p = current_p;
            current_p = next_p;
        }

        head = prev_p; //step 3
        return head;
    }
};
```
Time Complexity:
Since we are iterating only once through the list and achieving reversed list. Thus, the time complexity is O(N) where N is the number of nodes present in the list.
Space Complexity:
To perform given tasks, no external spaces are used except three-pointers. So, space complexity is O(1).

Reverse a Linked List : Recursive

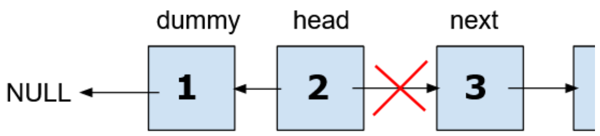
Intuition: This approach is very similar to the above 3 pointer approach. In the process of reversing, the base operation is manipulating the pointers of each node and at the end, the original head should be pointing towards NULL and the original last node should be 'head' of the reversed Linked List.
Approach:
In this type of scenario we first take a dummy node that will be assigned to NULL.
Then we take a next pointer which will be initialized to head->next and in future iterations, next will again be set to head->next
Now coming to changes on head node, as we have set dummy node as NULL and next to head->next, we can now update the next pointer of the head to dummy node.



Before moving to next iteration dummy is set to head and then head is set to next node.



Now coming to next iteration : We'll follow a similar process to set next as head->next and updating head->next = dummy, dummy set to head and head set to next



These iterations will keep going while the head of original Linked List is not NULL, i.e. we'll reach end of the original Linked List and the Linked List has been reversed.

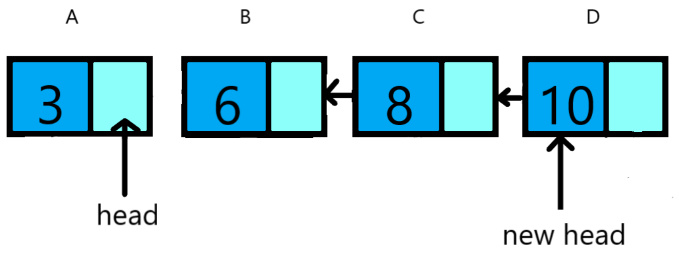
Code:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *newHead = NULL;
        while (head != NULL) {
            ListNode *next = head->next;
            head->next = newHead;
            newHead = head;
            head = next;
        }
        return newHead;
    }
};
```

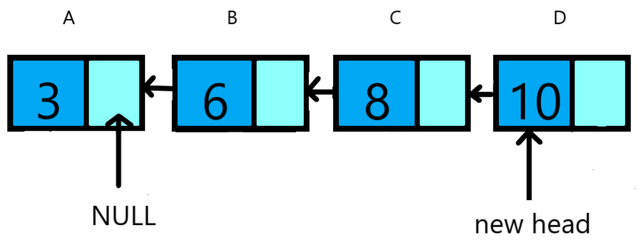
Solution 3: Using Recursion
We traverse to the end of the list recursively.
As we reach the end of the list, we make the end node the head. Then receive previous nodes and make them connected to the last one.
At last, we link the second node to the head and the first node to NULL. We return to our new head.
Code:

```
class Solution {
public:

    ListNode* reverseList(ListNode* &head) {

        if (head == NULL||head->next==NULL)
            return head;

        ListNode* nnode = reverseList(head->next);
        head->next->next = head;
        head->next = NULL;
        return nnode;
    }
};
```

Dry Run:
Once we call a function to reach the end of the list, we get a reversed list except the first node and second node.



Then we link node B to node A and node A to NULL.



Time Complexity:
We move to the end of the list and achieve our reversed list. Thus, the time complexity is O(N) where N represents the number of nodes.
Space Complexity:
Apart from recursion using stack space, no external storage is used. Thus, space complexity is O(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.

Examples:

```
Input Format:
( Pointer / Access to the head of a Linked list )
head = [1,2,3,4,5]

Result: [3,4,5]
( As we will return the middle of Linked list the further linked list will be still available )

Explanation: The middle node of the list is node 3 as in the below image.

Input Format:
Input: head = [1,2,3,4,5,6]

Result: [4,5,6]

Explanation:
Since the list has two middle nodes with values 3 and 4, we return the second one.
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: 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:

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
            int n = 0;
            ListNode* temp = head;
            while(temp) {
            n++;
            temp = temp->next;
            }
            temp = head;

            for(int i = 0; i < n / 2; i++) {
                    temp = temp->next;
            }
            return temp;
    }
};
```

Solution 2: [Efficient] Tortoise-Hare-Approach

Unlike the above approach, we don’t have to maintain nodes count here and we will be able to find the middle node in a single traversal so this approach is more efficient.

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 than that of the slow pointer. So when the fast ptr will be at the end of Linked List, slow ptr would have covered half of Linked List till then. So the slow ptr will be pointing towards the middle of Linked List.

Approach:

1. Create two pointers slow and fast and initialize them to a head pointer.
2. Move slow ptr by one step and simultaneously fast ptr by two steps until fast ptr is NULL or next of fast ptr is NULL.
3. When the above condition is met, we can see that the slow ptr is pointing towards the middle of Linked List and hence we can return the slow pointer.

Code:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode *slow = head, *fast = head;
        while (fast && fast->next)
            slow = slow->next, fast = fast->next->next;
        return slow;
    }
};
```

Time Complexity: O(N)
Space Complexity: O(1)
Follow up question you can try: Detect and remove Loop in Linked List

Merge two sorted Linked Lists

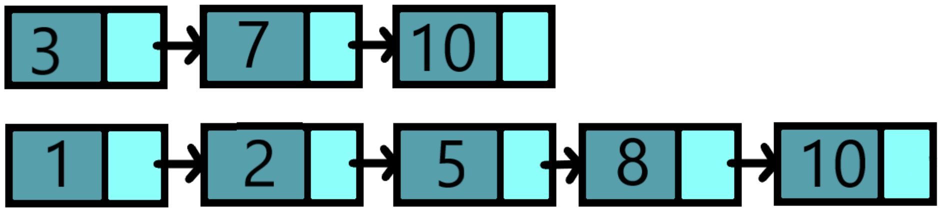
In this article we will solve the most asked coding interview question: ” Merge two sorted Linked Lists “

Problem Statement: Given two singly linked lists that are sorted in increasing order of node values, merge two **sorted** linked lists and return them as a sorted list. The list should be made by splicing together the nodes of the first two lists.

Example 1:

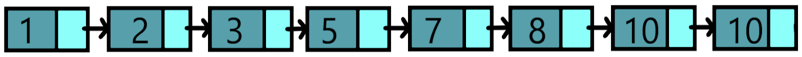
```
Input Format :
l1 = {3,7,10}, l2 = {1,2,5,8,10}

Output :
{1,2,3,5,7,8,10,10}
```

Explanation :



Merge two sorted linked lists example

These are two lists given. Both lists are sorted. We have to merge both lists and create a list that contains all nodes from the above nodes and it should be sorted.



Example 2:

```
Input Format :
l1 = {}, l2 = {3,6}

Output :
{3,6}

Explanation :
l1 is an empty list. l2 has two nodes. So, when we merge them, we will have the same list as l2.
```

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

<u>**Solution 1:** Using an externally linked list to store answers.</u>

Approach :

Step 1: Create a new dummy node. It will have the value 0 and will point to NULL respectively. This will be the head of the new list. Another pointer to keep track of traversals in the new list.

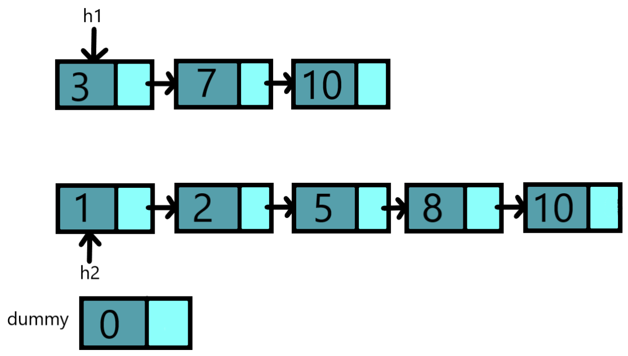
Step 2: Find the smallest among two nodes pointed by the head pointer of both input lists. Store that data in a new list created.

Step 3: Move the head pointer to the next node of the list whose value is stored in the new list.

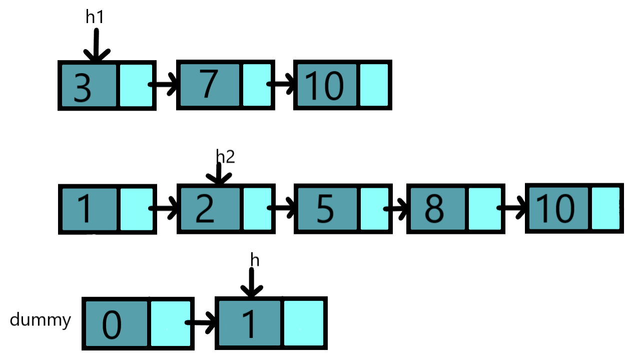
Step 4: Repeat the above steps till any one of the head pointers stores NULL. Copy remaining nodes of the list whose head is not NULL in the new list.

Dry Run :

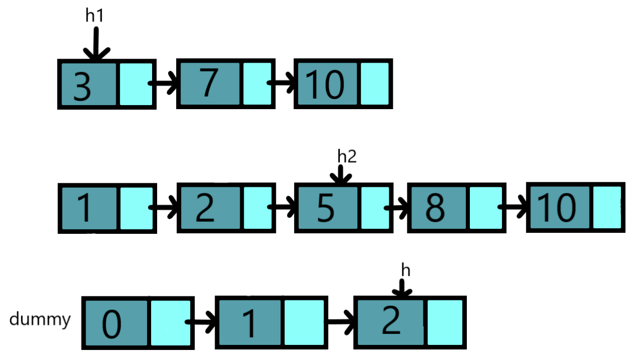
Creating a new dummy node. This will help to keep track as head of the new list to store merged sorted lists.



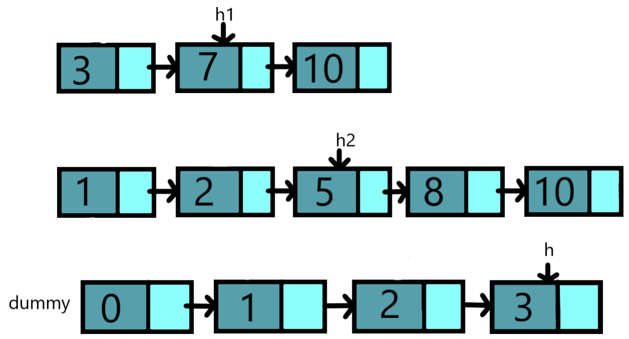
Find smallest among the two pointed by pointer h1 and h2 in each list. Copy that node and insert it after the dummy node. Here 1 < 3, therefore 1 will be inserted after the dummy node. Move h2 pointer to next node.



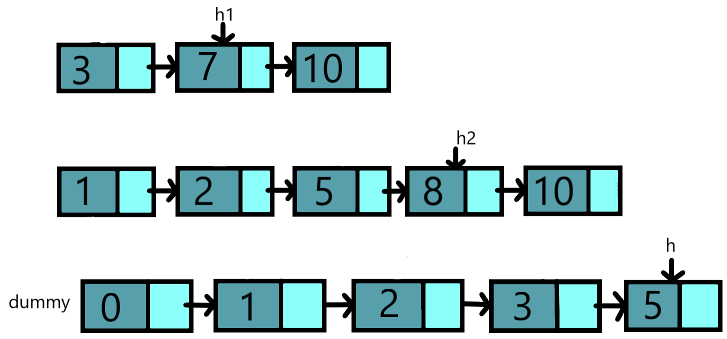
2 < 3, therefore, 2 is copied to another node and inserted at the end of the new list. h2 is moved to the next node.



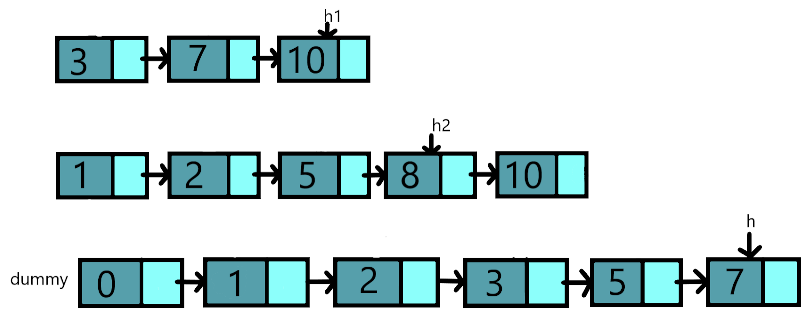
5 > 3, so 3 will be copied to another node and inserted at the end. h1 is moved to the next node.



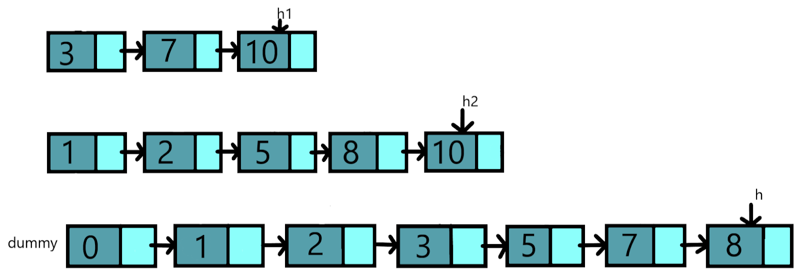
5 < 7, so 5 will be copied to another node and inserted at the end. h2 is moved to the next node.



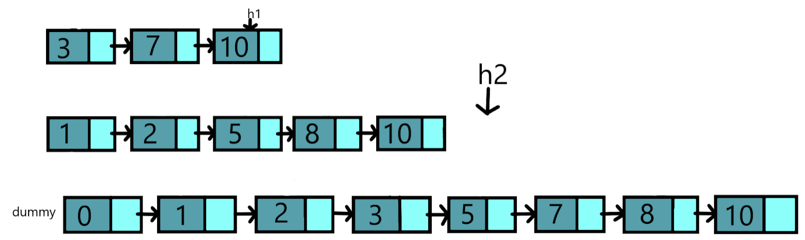
8 > 7, so 7 will be copied into a new node and inserted at the end. h1 is moved to the next node.



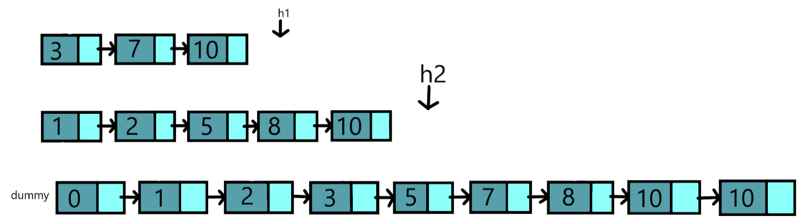
8 < 10, so 8 will be copied into a new node and inserted at the end. h1 is moved to the next node.



10 = 10, so 10 will be copied into a new node and inserted at the end. h2 is now NULL.



Now, list 1 has only nodes that are not inserted in the new list. So, we will insert the remaining nodes present in list 1 into the list.



dummy->next will result in the head of the new list.

Time Complexity: O(N+M).

Let N be the number of nodes in list l1 and M be the number of nodes in list l2. We have to iterate through both lists. So, the total time complexity is O(N+M).

Space Complexity: O(N+M).

We are creating another linked list that contains the (N+M) number of nodes in the list. So, space complexity is O(N+M).

Solution 2: Inplace method without using extra space.

The idea to do it without extra space is to play around with the next pointers of nodes in the two input lists and arrange them in a fashion such that all nodes are linked in increasing order of values.

Approach :

Step 1: Create two pointers, say l1 and l2. Compare the first node of both lists and find the small among the two. Assign pointer l1 to the smaller value node.

Step 2: Create a pointer, say res, to l1. An iteration is basically iterating through both lists till the value pointed by l1 is less than or equal to the value pointed by l2.

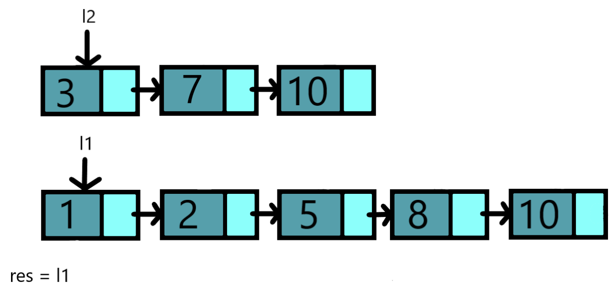
Step 3: Start iteration. Create a variable, say, temp. It will keep track of the last node sorted list in an iteration.

Step 4: Once an iteration is complete, link node pointed by temp to node pointed by l2. Swap l1 and l2.

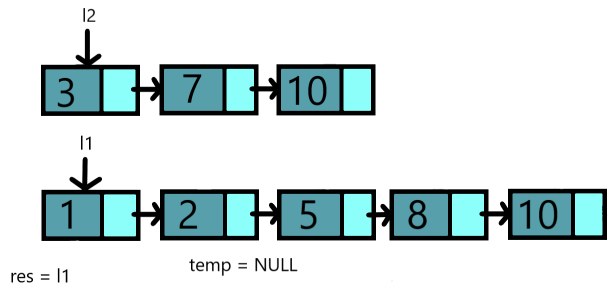
Step 5: If any one of the pointers among l1 and l2 is NULL, then move the node pointed by temp to the next higher value node.

Dry Run :

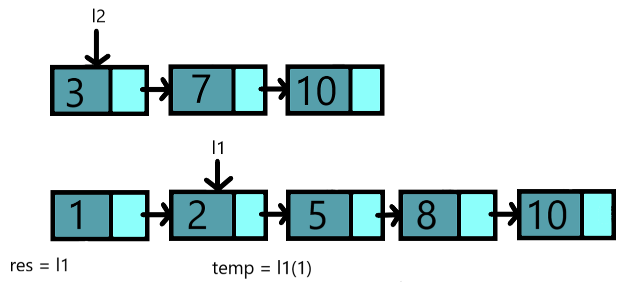
Created two pointers l1 and l2. Comparing the first node of both lists. Pointing l1 to the smaller one among the two. Create variable res and store the initial value of l1. This ensures the head of the merged sorted list.



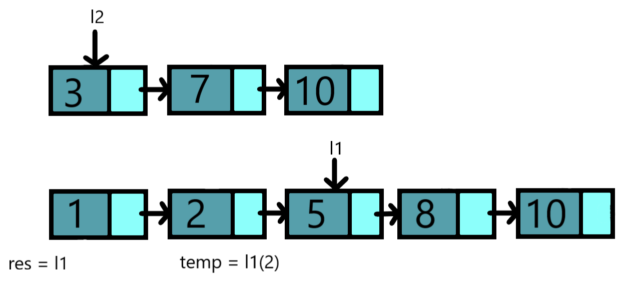
Now, start iterating. A variable temp will always be equal to NULL at the start of iteration.



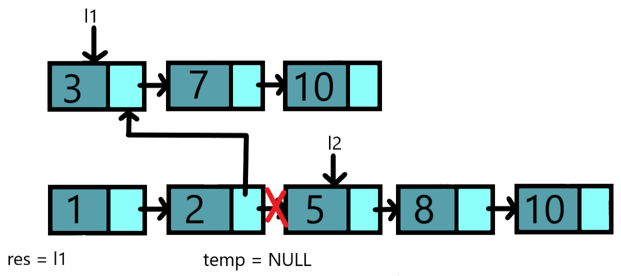
1 < 3. temp will store nodes pointed by l1. Then move l1 to the next node.



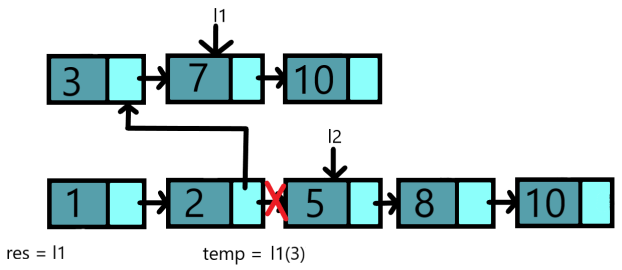
2 < 3. temp will store node l1(2) and then move l1 to the next node.



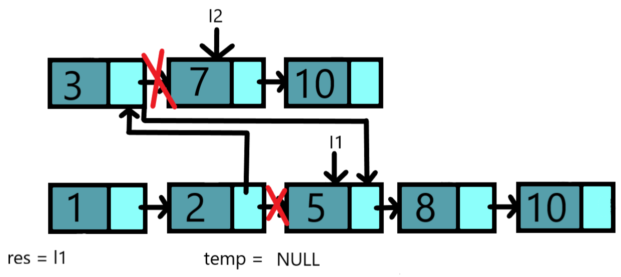
5 > 3. Now, the very first iteration completes. Now, the temp storing node is connected to the node pointed by l2, i.e 2 links to 3. Swap l1 and l2. Initialize temp to NULL.



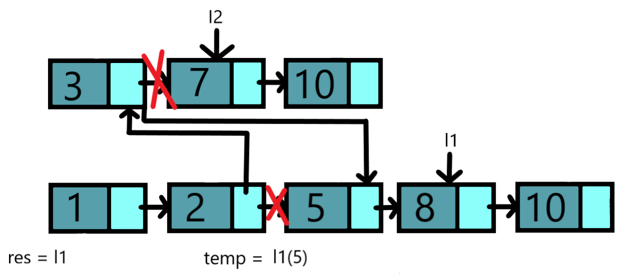
The second iteration starts. 3 < 5. So, first store l1(3) in temp then move l1 to the next connected node.



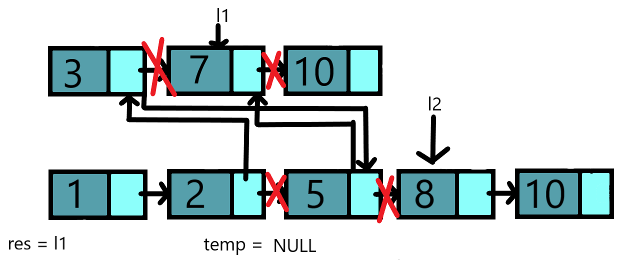
7 > 5. The second iteration stops here. Link node stored in temp node pointed by l2, i.e, 3 links to 5. Swap l1 and l2. temp is assigned to NULL at the start of the third iteration.



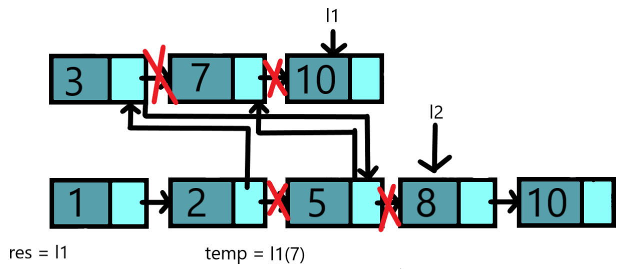
5 < 7. temp will store l1(5) and move l1 to the next linked node.



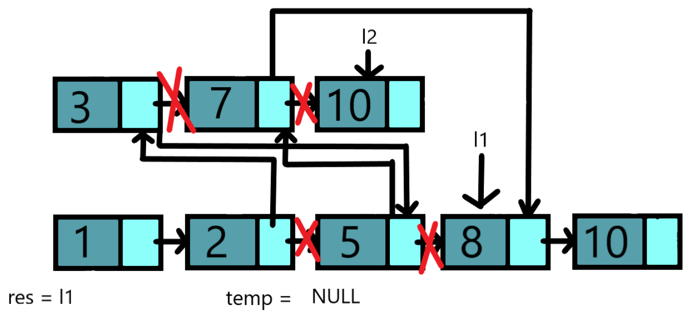
8 > 7. The third iteration stops. Link node stored in temp to node pointed by l2, i.e 5 links to 7. Swap l1 and l2. Assign temp to NULL at the start of the fourth iteration.



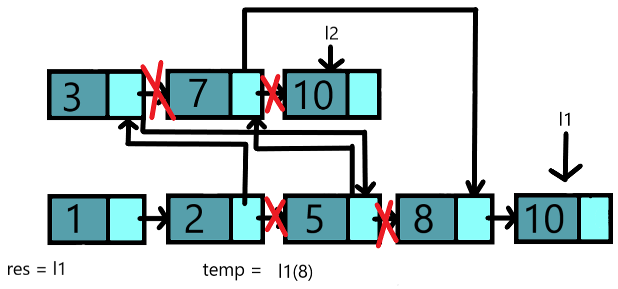
7 < 8. temp stores l1(7). l1 moves to the next node.



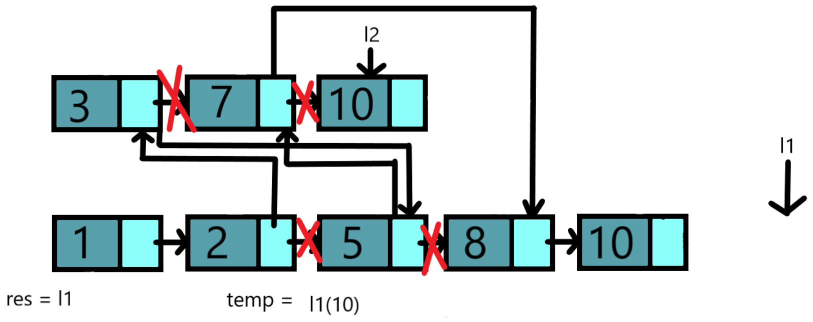
10 > 8. The fourth iteration stops here. 7 is linked to 8. Swap l1 and l2. The start of the fifth iteration initializes temp to NULL.



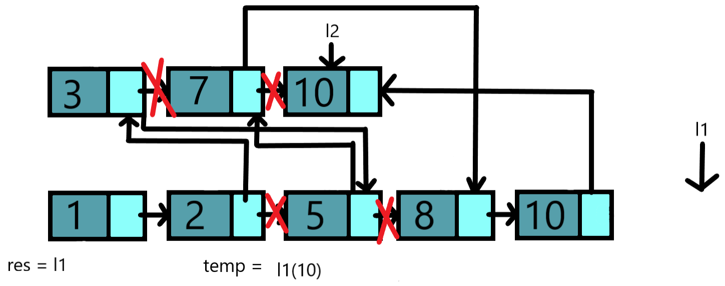
8 < 10. temp stores l1(8). l1 moves to the next node.



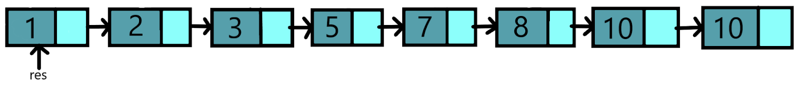
10 = 10. temp stores l1(10). l1 moves forward and is now equal to NULL.



As l1 is equal to NULL, so complete iteration stops. We link 10, which is stored in variable temp, is linked to 10 pointed by l2.



Hence, we achieved our sorted merge list.



Code :

```
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

        // when list1 is empty then
        // our output will be same as list2
        if(l1 == NULL) return l2;

        // when list2 is empty then our output
        // will be same as list1
        if(l2 == NULL) return l1;

        // pointing l1 and l2 to smallest and greatest one
        if(l1->val > l2->val) std::swap(l1,l2);

        // act as head of resultant merged list
        ListNode* res = l1;

        while(l1 != NULL && l2 != NULL) {

            ListNode* temp = NULL;

            while(l1 != NULL && l1->val <= l2->val) {

                temp = l1;//storing last sorted node
                l1 = l1->next;
            }

            // link previous sorted node with
            // next larger node in list2
            temp->next = l2;
            std::swap(l1,l2);
        }

        return res;
    }
};
```

Time Complexity :

We are still traversing both lists entirely in the worst-case scenario. So, it remains the same as O(N+M) where N is the number of nodes in list 1 and M is the number of nodes in list 2.

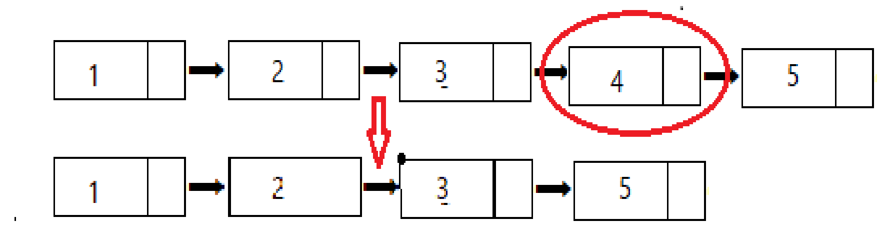
Space Complexity :
We are using the same lists just changing links to create our desired list. So no extra space is used. Hence, its space complexity is O(1).

Remove N-th node from the end of a Linked List

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

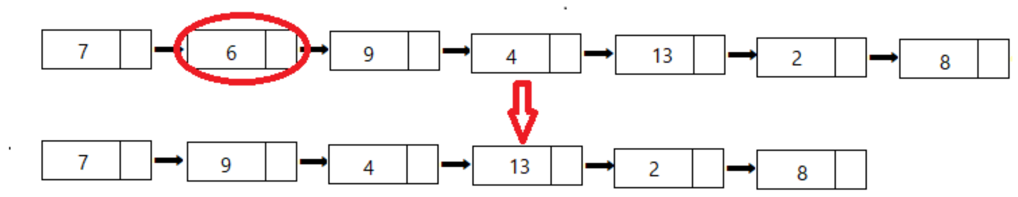
Example 1 :

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

Output: [1,2,3,5]

Explanation: Refer Below Image
```



Example 2:

```
Input: head = [7,6,9,4,13,2,8], n = 6

Output: [7,9,4,13,2,8]

Explanation: Refer Below Image
```



Example 3:

```
Input: head = [1,2,3], n = 3

Output: [2,3]
```

Solution :

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive Approach [Traverse 2 times]

Intuition: We can traverse through the Linked List while maintaining a count of nodes, let's say in variable ***count*** , and then traversing for the 2nd time for (n – ***count***) nodes to get to the nth node of the list.

Solution 2: [Efficient] Two pointer method
Unlike the above approach, we don't have to maintain the count value, we can find the nth node just by one traversal by using 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:
1. We have the flexibility of using two-pointers now.
2. We know, the n-th node from the end is the same as (total nodes – n)th node from start.
3. 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 end? Will it make anything easier for us?
Yes, with two pointers in hand out of which one at n-th node from start, we can just advance both of them till end simultaneously, once the faster reaches the end, the slower will stand at the n-th node from the end.

Approach :
1. Take two dummy nodes, who's next will be pointing to the head.
2. Take another node to store the head, initially it,s a dummy node(start), and the next of the node will be pointing to the head.The reason why we are using this extra dummy node, is because there is an edge case. If the node is equal to the length of the linkedlist, then this slow's will point to slow's next→ next. And we can say our dummy start node will be broken, and will be connected to the slow's next→ next.
3. Start traversing until the fast pointer reaches the nth node.

```
for(int i = 1; i <= n; ++i)
    fast = fast->next;
```

4. Now start traversing by one step both of the pointers until the fast pointers reach the end.

```
while(fast->next != NULL)
{
    fast = fast->next;
    slow = slow->next;
}
```

5. When the traversal is done, just do the deleting part. Make slow pointer's next to the next of next of the slow pointer to ignore/disconnect the given node.

```
slow->next = slow->next->next;
```

6. Last, return the next of start.

Dry Run: We will be taking the first example for the dry run, so, the linkedlist is [1,2,3,4,5] and the node which has to be deleted is 2 from the last. For the first time fast ptr starts traversing from node 1 and reaches to 2, as it traverses for node number 2, then the slow ptr starts increasing one, and as well as the fast ptr, until it reaches the end.

- 1st traversal : fast=3, slow=1
- 2nd traversal : fast=4, slow=2
- 3rd traversal : fast=5, slow=3

Now, the slow->next->next will be pointed to the slow->next
So , the new linked list will be [1,2,3,5]
Code :

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode * start = new ListNode();
        start -> next = head;
        ListNode* fast = start;
        ListNode* slow = start;

        for(int i = 1; i <= n; ++i)
            fast = fast->next;

        while(fast->next != NULL)
        {
            fast = fast->next;
            slow = slow->next;
        }

        slow->next = slow->next->next;

        return start->next;
    }
};
```

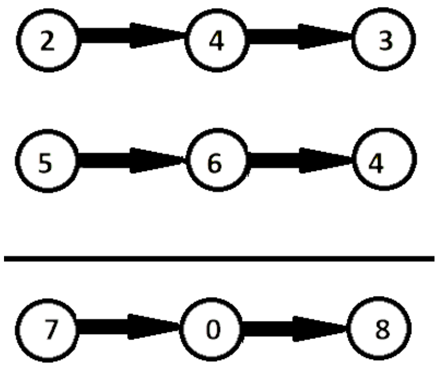
Time Complexity: O(N)
Space Complexity: O(1)

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.

Examples:

```
Input Format:
(Pointer/Access to the head of the two linked lists)

num1  = 342, num2 = 564

l1 = [2,4,3]
l2 = [5,6,4]

Result: sum = 807; L = [7,0,8]

Explanation: Since the digits are stored in reverse order, reverse the numbers first to get the
    or                                                 original number and then add them as → 342
    + 465 = 807. Refer to the image below.
```

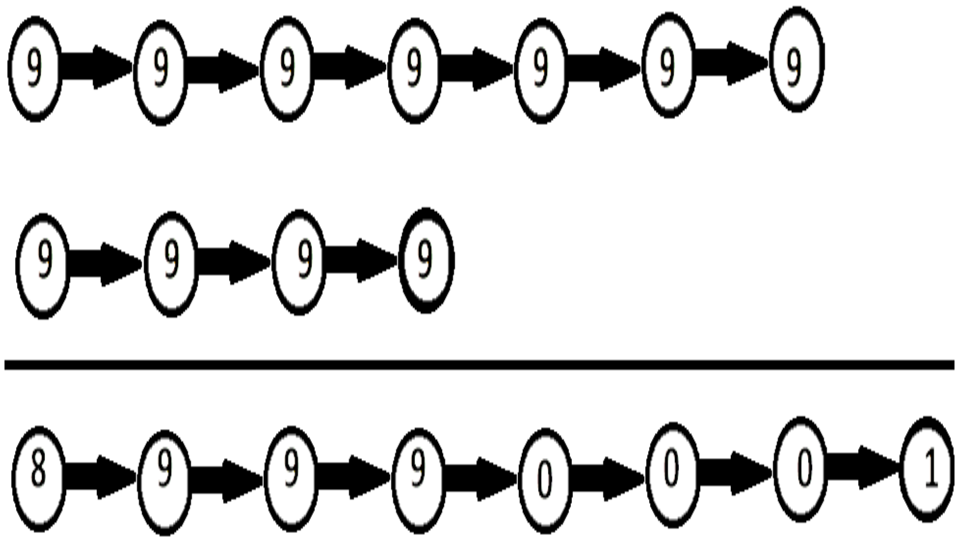


Input Format:
(Pointer/Access to the head of the two linked lists)

```
l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
```

Result: [8,9,9,9,0,0,0,1]

Explanation: Since the digits are stored in reverse order, reverse the numbers first to get the original number and then add them as → 9999999 + 9999 = 8999001. ***Refer to the image below.***



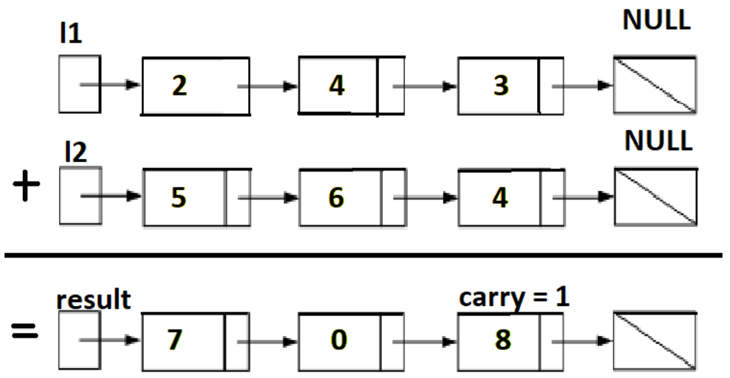
Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Elementary Math

Intuition: Keep track of the ***carry*** using a variable and simulate digits-by-digits sum starting from the head of the list, which contains the least significant digit.

Approach:



Visualization of the addition of two numbers:
342 + 465 = 807
342+465=807.
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.

Psuedocode:

```
● Create a dummy node which is the head of new linked list.
● Create a node temp, initialise it with dummy.
● Initialize carry to 0.
● Loop through lists l1 and l2 until you reach both ends, and until carry is present.
    ○ Set sum=l1.val+ l2.val + carry.
    ○ Update carry=sum/10.
    ○ Create a new node with the digit value of (sum%10) and set it to temp node’s next, then advance temp node to next.
    ○ Advance both l1 and l2.
● Return dummy’s next node.
```

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.

Take extra caution in the following cases:

st case	planation
[0,1], l2=[0,1,2]	ıen one list is longer than the other.
[], l2=[0,1]	ıen one list is null, which means an empty list.
[9,9], l2=[1]	e sum could have an extra carry of one at the end, which is easy to forget.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode();
        ListNode *temp = dummy;
        int carry = 0;
        while( (l1 != NULL || l2 != NULL) || carry) {
            int sum = 0;
            if(l1 != NULL) {
                sum += l1->val;
                l1 = l1 -> next;
            }

            if(l2 != NULL) {
                sum += l2 -> val;
                l2 = l2 -> next;
            }

            sum += carry;
            carry = sum / 10;
            ListNode *node = new ListNode(sum % 10);
            temp -> next = node;
            temp = temp -> next;
        }
        return dummy -> next;
    }
};
```

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.

Delete given node in a Linked List : O(1) approach

Problem Statement: Write a function to **delete a node** in a singly-linked list. You will **not** be given access to the head of the list instead, you will be given access to **the node to be deleted** directly. It is **guaranteed** that the node to be deleted is **not a tail node** in the list.

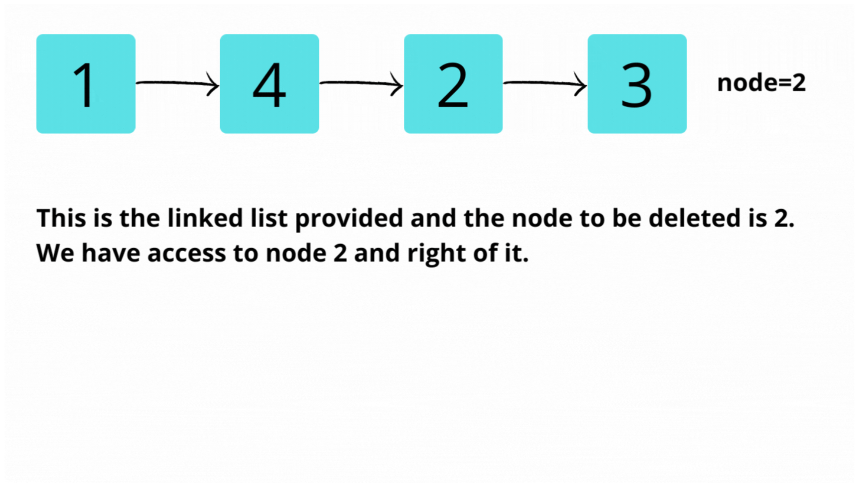
Examples:

```
Example 1:
Input:
 Linked list: [1,4,2,3]
       Node = 2
Output:
Linked list: [1,4,3]
Explanation: Access is given to node 2. After deleting nodes, the linked list will be modified
   to [1,4,3].
```

```
Example 2:
Input:
 Linked list: [1,2,3,4]
       Node = 1
Output: Linked list: [2,3,4]
Explanation:
 Access is given to node 1. After deleting nodes, the linked list will be modified to [2,3,4].
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Approach:
We are given access to nodes that we have to delete from the linked list. So, whatever operation we want to do in the linked list, we can operate in the right part of the linked list from the node to be deleted.
The approach is to copy the next node's value in the deleted node. Then, link node to next of next node. This does not delete that node but indirectly it removes that node from the linked list.
Dry Run:


1
4
2
3
node=2
This is the linked list provided and the node to be deleted is 2.
We have access to node 2 and right of it.

Code:
```
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
    }
};
```

Output:
Given Linked List:
1->4->2->3
Linked List after deletion:
1->4->3
Time Complexity: O(1)
Reason: It is a two-step process that can be obtained in constant time.
Space Complexity: O(1)
Reason: No extra data structure is used.