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 :

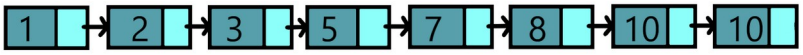
l1 = {3,7,10}, l2 = {1,2,5,8,10}

Output :

{1,2,3,5,7,8,10,10}

Explanation :

These are the 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.*

Solution 1: Using an externally linked list to store answers.

Approach :

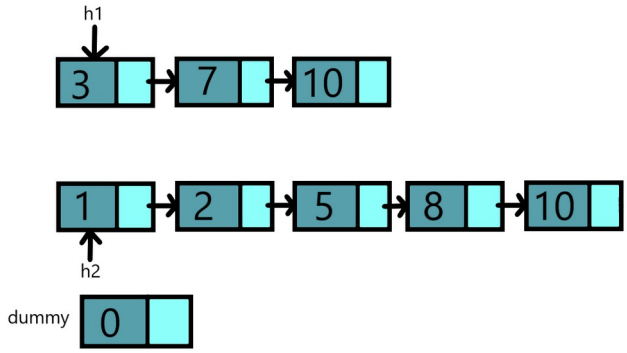
Step 1: Create a new dummy node. It will have the value 0 and will point to NULL respectively. This will be the head of the new list. Another pointer to keep track of traversals in the new list.

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

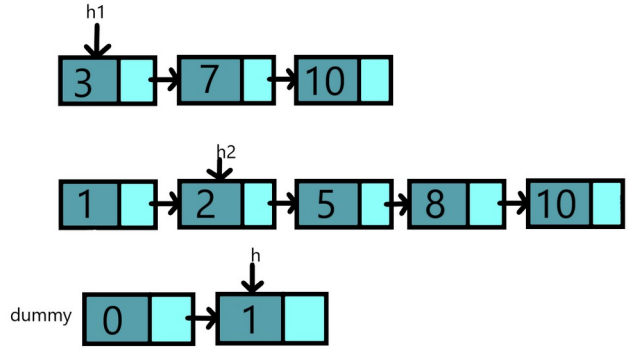
Step 3: Move the head pointer to the next node of the list whose value is stored in the new list.

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.

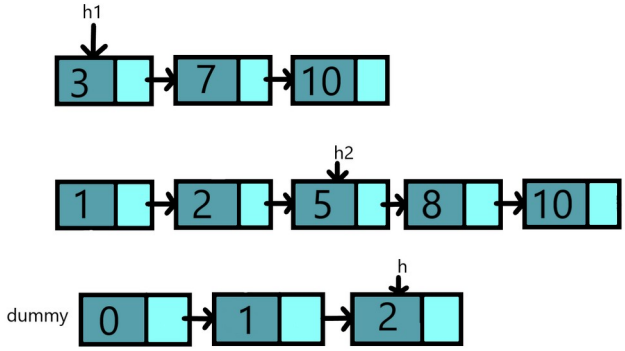
Dry Run :
Creating a new dummy node. This will help to keep track as the head of the new list to store merged sorted lists.



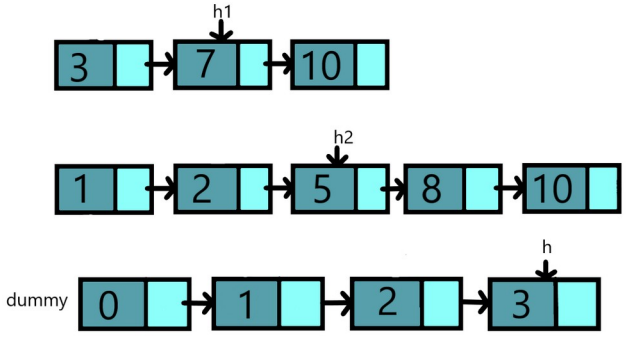
Find the smallest among the two pointed by pointers h1 and h2 in each list. Copy that node and insert it after the dummy node. Here 1 < 3, therefore 1 will be inserted after the dummy node. Move h2 pointer to next node.



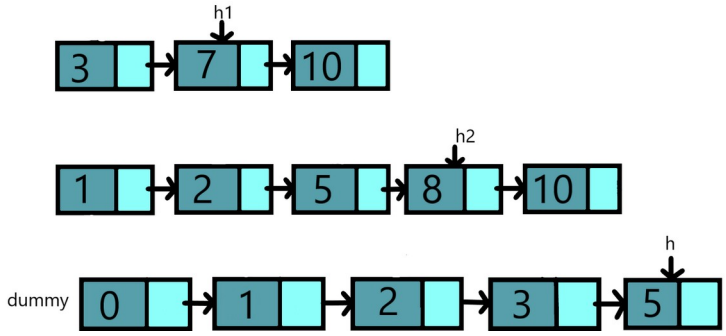
2 < 3, therefore, 2 is copied to another node and inserted at the end of the new list. h2 is moved to the next node.



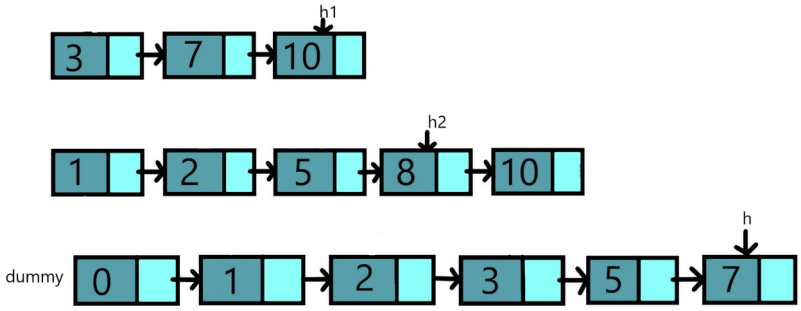
5 > 3, so 3 will be copied to another node and inserted at the end. h1 is moved to the next node.



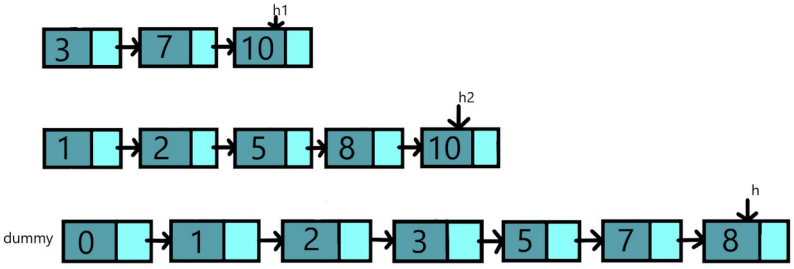
5 < 7, so 5 will be copied to another node and inserted at the end. h2 is moved to the next node.



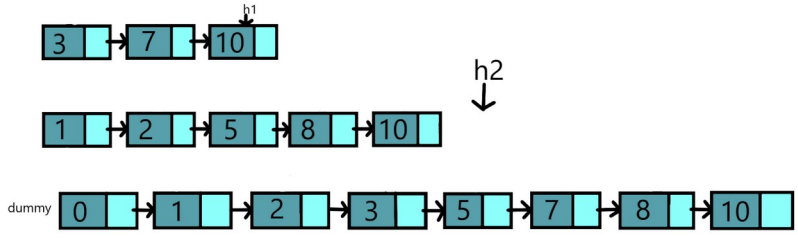
8 > 7, so 7 will be copied into a new node and inserted at the end. h1 is moved to the next node.



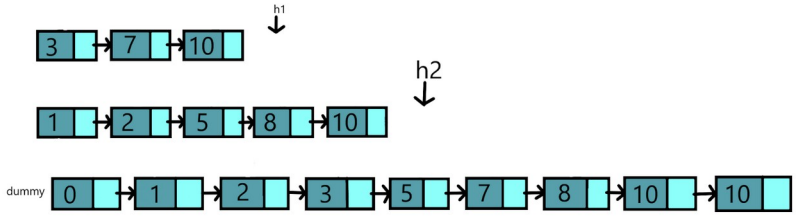
8 < 10, so 8 will be copied into a new node and inserted at the end. h1 is moved to the next node.



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

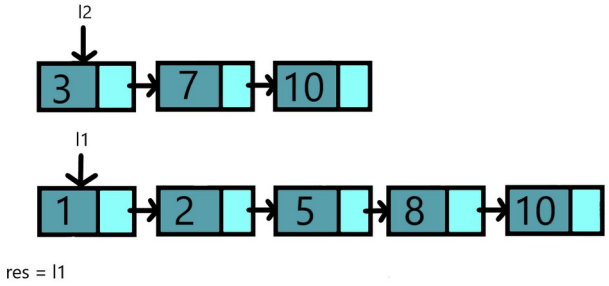


Now, list 1 has only nodes that are not inserted in the new list. So, we will insert the remaining nodes present in list 1 into the list.

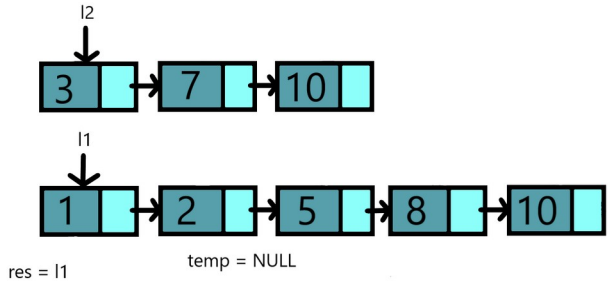


dummy->next will result in the head of the new list.
Time Complexity: O(N+M).

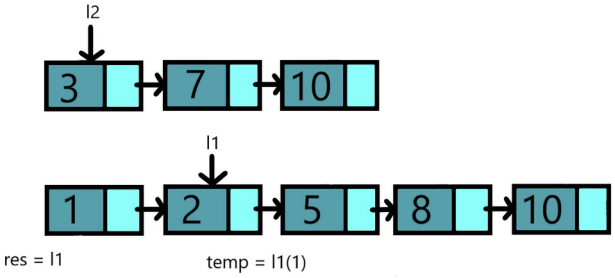
Let N be the number of nodes in list l1 and M be the number of nodes in list l2. We have to iterate through both lists. So, the total time complexity is O(N+M).
Space Complexity: O(N+M).
We are creating another linked list that contains the (N+M) number of nodes in the list. So, space complexity is O(N+M).
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.



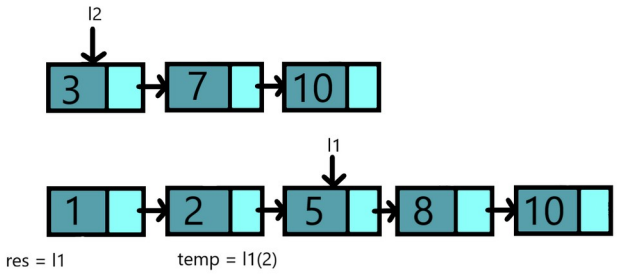
Now, start iterating. A variable temp will always be equal to NULL at the start of the iteration.



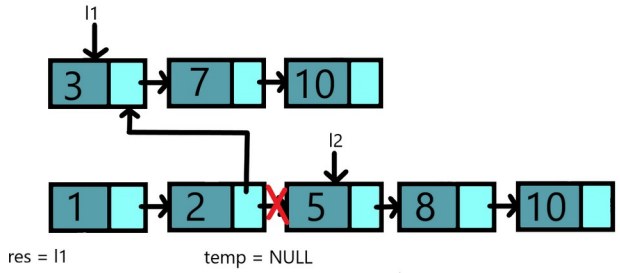
1 < 3. temp will store nodes pointed by l1. Then move l1 to the next node.



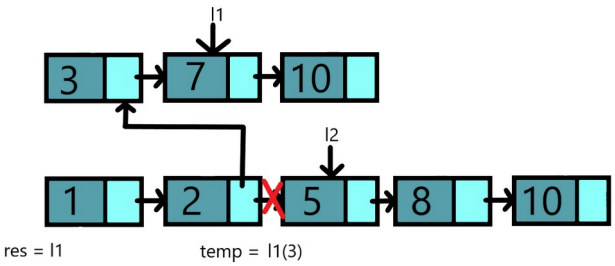
2 < 3. temp will store node l1(2) and then move l1 to the next node.



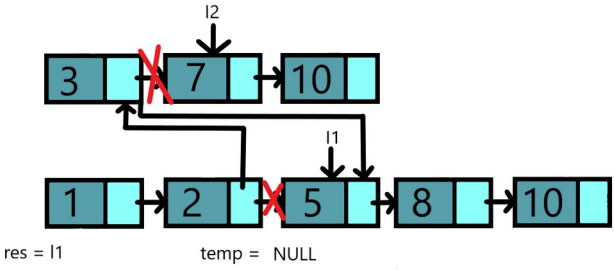
5 > 3. Now, the very first iteration completes. Now, the temp storing node is connected to the node pointed by l2, i.e 2 links to 3. Swap l1 and l2. Initialize temp to NULL.



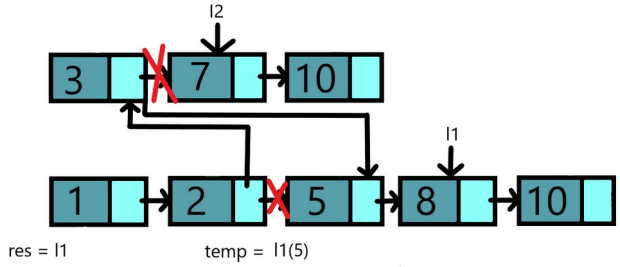
The second iteration starts. 3 < 5. So, first store l1(3) in temp then move l1 to the next connected node.



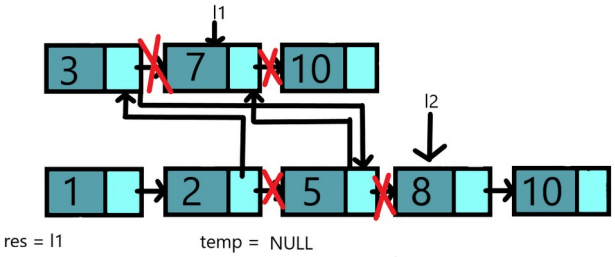
7 > 5. The second iteration stops here. Link node stored in temp node pointed by l2, i.e, 3 links to 5. Swap l1 and l2. temp is assigned to NULL at the start of the third iteration.



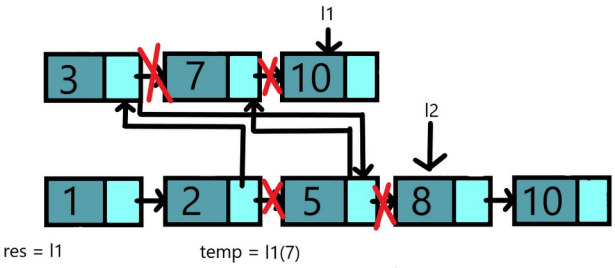
5 < 7. temp will store l1(5) and move l1 to the next linked node.



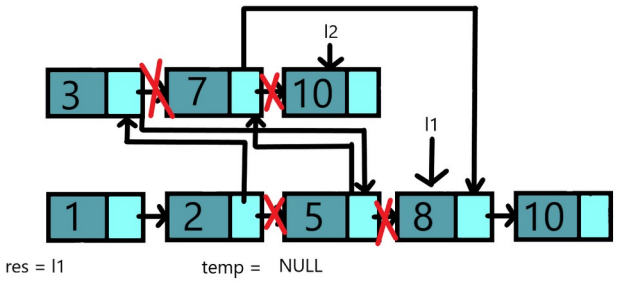
8 > 7. The third iteration stops. Link node stored in temp to node pointed by l2, i.e 5 links to 7. Swap l1 and l2. Assign temp to NULL at the start of the fourth iteration.



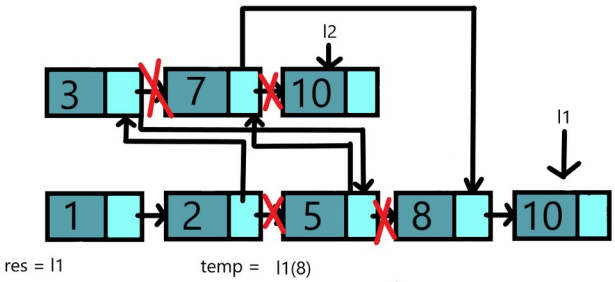
7 < 8. temp stores l1(7). l1 moves to the next node.



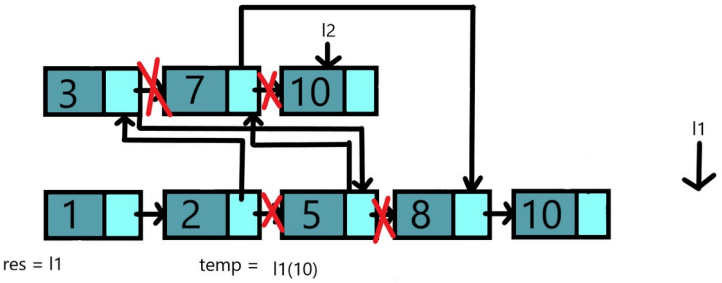
10 > 8. The fourth iteration stops here. 7 is linked to 8. Swap l1 and l2. The start of the fifth iteration initializes temp to NULL.



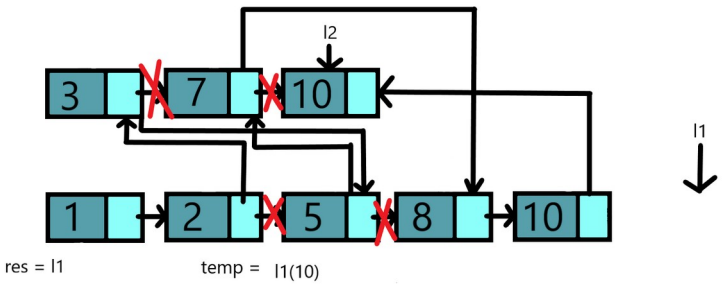
8 < 10. temp stores l1(8). l1 moves to the next node.



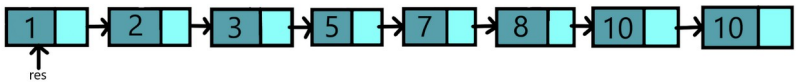
10 = 10. temp stores l1(10). l1 moves forward and is now equal to NULL.



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 :

- C++ Code
- Python 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).