## Problem Description

taking one element at a time and inserting it into its correct position in a separate sorted part of the list. The process for insertion sort in the context of a linked list involves the following steps: Start with the second element of the list (since a list with one element is already sorted), and for each element—

where each node contains a value and a reference to the next node in the list. The insertion sort algorithm sorts a list or an array by

In this problem, the goal is to sort a singly linked list using the insertion sort algorithm. A singly linked list is a collection of nodes

Leetcode Link

2. Compare with the elements in the sorted part of the list (starting from the beginning) to find the correct position.

- 3. Insert the element at the found position in the sorted part. 4. Continue the process until we reach the end of the original list, and no elements are left to sort.
- The challenge in this problem lies in properly manipulating the pointers in the linked list without losing track of the nodes.
- The provided solution uses a dummy node to simplify edge cases and ensure that there's always a 'previous' node for comparison,

thus streamlining the insertion process.

## Here is the intuition behind the provided solution code:

Intuition

1. If the list is empty or has only one element, there is no need to sort, so we return the head as is. 2. We create a dummy node and set its next pointer to the head of the list. 3. We use two pointers pre and cur, starting at the dummy and head positions, respectively, to traverse the list. The pre pointer

checks if the cur pointer's value is in the right position (i.e., if it's greater than or equal to the current sorted part).

4. If the current node value is in the right place, simply move both pointers one step forward.

8. Repeat the process until cur reaches the end of the list.

- 5. If the current node needs to be repositioned, we use another pointer p that starts at the dummy node and finds the correct
- position in the sorted part of the list where the current node (cur) should be inserted. 6. Insert the current node (cur) into the sorted part of the list at the correct position.
- 7. Reconnect the pre.next to cur.next to maintain the unsorted part of the list without the newly inserted node.
- The core idea of this solution is keeping track of where the node needs to be inserted in the already sorted part of the list and carefully managing node pointers to maintain linked list integrity throughout the sort.

9. Return dummy.next, which now points to the head of the newly sorted linked list.

- **Solution Approach** The solution implements the insertion sort algorithm specifically tailored to a singly linked list. To understand this approach, we
- visualize the list as comprising two parts: the sorted part and the unsorted part. Initially, the sorted part contains just the first element, and the rest of the list is considered unsorted.

Here's how the code works step-by-step: A dummy node is created and points to the head of the list. This dummy node acts as the anchor for the start of the sorted list.

Two pointers, pre and cur, are used. pre starts at the dummy node and cur starts at the actual head of the list.

The while loop runs as long as cur is not None, which means that there are elements in the unsorted part.

the right position relative to the pre node, and both pre and cur advance to the next nodes.

Next, cur is removed from its current position, and its next pointer is stored temporarily in t.

cur is updated to t, and the cycle continues until all elements are sorted.

The dummy node's next points to 4, the head of our unsorted list.

pre starts at the dummy node, and cur starts at the head (value 4).

Move cur to the next node (2). pre still points to the dummy.

We introduce pointer p starting at dummy to look for the insertion point for 2.

Since dummy.next (value 4) is greater than 2, we insert 2 after dummy.

Update the connections: dummy.next becomes 2, 2.next becomes 4.

Now, cur is pointing to 1. We compare it with the sorted part of the list.

Again p starts from dummy and moves to find where to insert 1.

The code uses a dummy node, which is a common technique when dealing with linked list operations, to simplify handling edge cases

Inside the loop, we initially check if the pre value is less than or equal to cur value. If it is, it means that the cur node is already in

• The code then uses another pointer, p, which starts from the dummy and traverses the list to find the insert position. This loop

• If the pre value is greater than the cur value, we need to find the correct spot to insert cur into the already-sorted part of the list.

additional data structures.

· We create a dummy node.

4 -> 2 -> 1 -> 3

**Initial Setup:** 

like inserting at the beginning of the list.

- cur is then inserted into the sorted part by setting cur.next to p.next and then setting p.next to cur. The old pre node is then connected to t, effectively removing the original cur and continuing with the rest of the list.
- This code neatly handles the insertion sort logic without needing an additional data structure other than the given linked list. The use of pointers to nodes (pre, cur, p, and t) is essential to perform the required operations without losing any part of the list.

We want to sort this list in ascending order using the insertion sort algorithm described in our solution approach.

• Since there's nothing preceding 4, it's already in the correct position (the sorted part has only one element).

continues until p.next.val is greater than cur.val, indicating the position where cur should be placed.

Example Walkthrough Let's use a small example to illustrate the solution approach. Imagine we have a singly linked list with the following values:

This algorithm has a time complexity of O(n^2) in the worst case because, for each element, it potentially iterates through the entire

sorted part to find the insert position. The space complexity is O(1) since the sort is performed in place without allocating any

Now, we iterate through the list to sort it: Step 1: List: dummy -> 4 -> 2 -> 1 -> 3

## Compare cur (value 2) with pre.next (value 4). Since 2 is less than 4, 2 must be moved to the correct position.

Step 3:

Step 4:

After sorting:

The final sorted linked list is:

**Python Solution** 

class ListNode:

Step 2:

List: 2 -> 4 -> 1 -> 3 (with dummy pointing to 2)

Now the sorted list is dummy -> 2 -> 4.

The sorted list now is dummy -> 1 -> 2 -> 4.

Sorted list is now dummy -> 1 -> 2 -> 3 -> 4.

Finally, we have dummy -> 1 -> 2 -> 3 -> 4.

List: dummy -> 4 -> 2 -> 1 -> 3

- Since 1 is less than 2, it should be inserted at the beginning. Connect dummy.next to 1, and 1.next to 2.
- List: 1 -> 2 -> 4 -> 3 (with dummy pointing to 1) cur is now pointing to 3. As before, use p to find the position for 3.

Since dummy is our initial node, the head of the sorted list is dummy next, which points to 1.

manipulating pointers and ensuring the list stays connected throughout the sorting process.

 We place 3 between 2 and 4, since 3 is greater than 2 and less than 4. Update pointers: 2.next becomes 3, and 3.next becomes 4.

## 1 -> 2 -> 3 -> 4 This walkthrough demonstrates how the insertion sort algorithm is applied to a singly linked list, efficiently repositioning nodes by

10

12

13

14

15

16

17

18

19

20

21

27

28

29

30

31

32

33

34

35

36

37

38

39

40

10

11

class Solution {

self.next = next\_node class Solution: def insertionSortList(self, head: ListNode) -> ListNode: # If the list is empty or has only one element, it's already sorted.

def \_\_init\_\_(self, value=0, next\_node=None):

if head is None or head.next is None:

previous, current = head, head.next

# Iterate while there are nodes to sort.

insert\_position = dummy\_node

current.next = insert\_position.next

# Set 'current' to the next node to sort.

insert\_position.next = current

# Perform the insertion.

public ListNode insertionSortList(ListNode head) {

// Initialize the dummy node with an initial value.

// Previous and current pointers for traversal.

ListNode previous = dummy, current = head;

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

ListNode dummy = new ListNode(0);

return head;

temp = current.next

previous.next = temp

current = temp

# Initialize a dummy node to help with insertion later.

1 # Definition for singly-linked list.

self.value = value

return head

while current:

dummy\_node = ListNode(0)

dummy\_node.next = head

if previous.value <= current.value:</pre> 23 # The current node is already in the correct place. 24 previous, current = current, current.next 25 else: # Find the correct place to insert the current node. 26

# The list is now sorted, return it starting from the node after dummy node.

while insert\_position.next.value <= current.value:</pre>

// If the list is empty or has only one item, it's already sorted.

insert\_position = insert\_position.next

return dummy\_node.next 41 42 Java Solution

# 'previous' tracks the last node before the one being sorted, 'current' is the node being sorted.

```
12
13
           // Iterate over the list to sort each element.
           while (current != null) {
14
               // If the value at previous is less than or equal to the current's value, move the pointers forward.
15
16
               if (previous.val <= current.val) {</pre>
17
                    previous = current;
                    current = current.next;
18
19
                    continue;
20
               // Find the correct position for the current node in the sorted part of the list
21
22
               ListNode position = dummy;
23
               while (position.next != null && position.next.val <= current.val) {</pre>
24
                    position = position.next;
25
26
               // Insert current node in the correct position.
27
               ListNode temp = current.next;
28
                current.next = position.next;
29
                position.next = current;
30
31
               // Move the previous pointer's next to temp and update current to temp.
32
                previous.next = temp;
33
                current = temp;
34
35
           // Return the next node of dummy since the first node is a placeholder.
36
           return dummy.next;
37
38 }
39
   // Definition for singly-linked list.
   class ListNode {
       int val;
42
       ListNode next;
43
       // Constructor to initialize a node with no next element.
       ListNode() {}
47
       // Constructor to initialize a node with a value.
       ListNode(int val) { this.val = val; }
48
49
       // Constructor to initialize a node with a value and a next element.
50
       ListNode(int val, ListNode next) { this.val = val; this.next = next; }
51
52 }
53
C++ Solution
   // Definition for singly-linked list.
2 struct ListNode {
       int val;
```

```
57
       // The dummy's next node is now the head of the sorted list.
58
       return dummy.next;
59 }
60
   // The above function can be utilized by creating `ListNode` objects and passing the head to `insertionSortList`.
62
```

ListNode \*next;

8

9

14

16

17

18

19

20

21

22

23

24

25

26

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

\*/

ListNode(int x) : val(x), next(nullptr) {}

ListNode\* insertionSortList(ListNode\* head) {

if (!head || !(head->next)) {

ListNode \*previousNode = &dummy;

// Perform the insertion.

ListNode \*currentNode = head;

return head;

ListNode dummy(0);

while (currentNode) {

continue;

ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* @param head - The head of the singly linked list to be sorted.

\* @return The head of the linked list sorted in ascending order.

// If the list is empty or has only one node, no sorting is needed.

// The dummy node is used as a placeholder to help with inserting nodes at the beginning of the list.

// Find the correct position to insert the current node by traversing from the start of the list.

// If the current node is already in the correct position, just move forward.

while (positionToInsert->next && positionToInsert->next->val <= currentNode->val) {

// This does not change in the case when the previous node is equal to current node

// Such case can happen when the previous node is the dummy node and the current node

// After insertion, the previous (dummy) node directly points to the new current node.

\* Sorts a linked list using the insertion sort algorithm.

if (previousNode->val <= currentNode->val) {

positionToInsert = positionToInsert->next;

// Link the previous part to the new next position as needed.

// is the first node being inserted in the sorted part.

currentNode = currentNode->next;

previousNode = currentNode;

ListNode \*positionToInsert = &dummy;

ListNode \*nextNode = currentNode->next;

positionToInsert->next = currentNode;

previousNode->next = nextNode;

// Move the current node to the next position.

if(previousNode != currentNode)

currentNode = nextNode;

currentNode->next = positionToInsert->next;

```
Typescript Solution
   /**
    * Definition for singly-linked list.
    */
   class ListNode {
       val: number;
       next: ListNode | null;
       constructor(val: number = 0, next: ListNode | null = null) {
 8
           this.val = val;
 9
           this.next = next;
10
12 }
13
   /**
14
    * Sorts a linked list using insertion sort algorithm.
    * @param {ListNode | null} head - The head of the singly linked list to be sorted.
    * @return {ListNode | null} The head of the linked list sorted in ascending order.
18
    */
   const insertionSortList = (head: ListNode | null): ListNode | null => {
       // If the list is empty or has only one node, no sorting is needed.
20
       if (head === null || head.next === null) {
21
22
           return head;
23
24
25
       // The dummy node is used as a placeholder to help with inserting nodes at the beginning of the list.
26
       let dummy: ListNode = new ListNode(0);
27
       let previousNode: ListNode = dummy;
28
       let currentNode: ListNode | null = head;
29
       while (currentNode !== null) {
30
           // If the current node is already in the correct position, just move forward.
31
32
           if (previousNode.val <= currentNode.val) {</pre>
33
               previousNode = currentNode;
34
               currentNode = currentNode.next;
35
               continue;
36
37
38
           // Find the correct position to insert the current node by traversing from the start.
           let positionToInsert: ListNode = dummy;
           while (positionToInsert.next !== null && positionToInsert.next.val <= currentNode.val) {</pre>
40
                positionToInsert = positionToInsert.next;
41
42
43
44
           // Perform the insertion.
45
           let nextNode: ListNode | null = currentNode.next;
           currentNode.next = positionToInsert.next;
46
           positionToInsert.next = currentNode;
           // Link the previous part to the next position.
49
50
           previousNode.next = nextNode;
51
52
           // Move the current node to the next position.
53
           currentNode = nextNode;
54
55
56
       // The dummy's next node is now the head of the sorted list.
57
       return dummy.next;
58 };
59
   // The above function can be utilized by creating `ListNode` objects and passing the head to `insertionSortList`.
61
Time and Space Complexity
```

can run up to n times in the worst case (when the list is in reverse order). In the best case, when the list is already sorted, the time complexity is O(n). However, for the average and worst cases, since each element could potentially be compared with all the already sorted elements, the resultant complexity remains 0(n^2).

linked list. This is because the algorithm consists of two nested loops: the outer loop runs once for every element, and the inner loop

The time complexity of the insertion sort algorithm is generally  $0(n^2)$  for a linked list, where n is the number of elements in the

The space complexity of the code is 0(1) because it does not use additional space proportional to the input size. The sorting is done in place within the input list, and only a fixed number of extra pointers are used, irrespective of the input size.