83. Remove Duplicates from Sorted List

Easy

Problem Description

Linked List

In this problem, we are provided with the head of a <u>linked list</u> that is already sorted. Our task is to remove any duplicate values from the list. A duplicate is identified when two or more consecutive nodes have the same value. It's important to note that after removing duplicates, the remaining linked list should still be sorted. We must return the modified list with all duplicates deleted, ensuring that each value in the list appears exactly once.

Intuition The intuition behind the solution comes from the fact that the <u>linked list</u> is already sorted. Since the linked list is sorted, all duplicates of a particular value will be adjacent to each other. We can simply traverse the linked list from the head to the end, and for each node, we check if its value is the same as the value of the next node. If it is, we have found a duplicate, and we need to remove the next node by changing pointers. We update the current node's next pointer to the next node's next pointer,

effectively skipping over the duplicate node and removing it from the list. If the values are not identical, we move on to the next node. We repeat this process until we have checked all nodes. The given Python code implements this approach by using a while loop that continues as long as there are more nodes to examine (cur and cur next are not None). **Solution Approach** The implementation of the solution involves a classical algorithm for removing duplicates from a sorted linked list. The algorithm utilizes the fact that a linked list allows for efficient removal of a node by simply rerouting the next pointer of the previous node.

Here's a step-by-step explanation of how the given Python code works:

Step 1: Initialize A pointer named cur is initialized to point to the head of the linked list.

We use a while loop to go through the linked list. The loop runs as long as cur is not None (indicating that we haven't reached the

Step 2: Traverse the Linked List

end of the list) and cur.next is not None (indicating that there is at least one more node to examine for potential duplicates). while cur and cur.next:

Step 4: Remove Duplicates

if cur.val == cur.next.val:

cur.next = cur.next.next

Step 6: Return the Updated List

Step 3: Check for Duplicates

If cur. val equals cur. next. val, we've found a duplicate. Instead of removing the current node, which would be more challenging, we remove the next node. This is accomplished by updating the next pointer of cur to skip the next node and point to the

If no duplicate was found (the else branch), we simply move the pointer cur to the next node to continue the process:

Inside the loop, we compare the current node's value cur. val with the value of the next node cur.next.val.

This effectively removes the duplicate node from the list without disturbing the rest of the list's structure.

Step 5: Move to the Next Distinct Element

cur = cur.next

else:

following one:

Once the loop is finished (meaning we've reached the end of the list or there are no more items in the linked list), we return the head of the list, which now points to the updated, duplicate-free sorted linked list.

Using this simple yet effective approach, we ensure that the list stays sorted, as we're only removing nodes and not altering the

Let's use a small example to illustrate the solution approach. Consider the following sorted linked list where some values are

order of the remaining nodes. **Example Walkthrough**

duplicated:

Step 1: Initialize

Step 3: Check for Duplicates

1 -> 2 -> 2 -> 3 -> 3 -> 4 -> 4 -> 5

 We start with a pointer cur pointing to the head (the node with value 1). **Step 2: Traverse the Linked List**

• The loop continues, and now cur points to the node with value 2, and cur.next points to the node with value 3, which is distinct. We move to

Step 8: Final Linked List After the while loop finishes, we've removed all duplicates, and our final linked list looks like this:

• With no more duplicates left to remove, we exit the while loop and return the head of the updated list, which is the reference to the first node in

Step 4: Is there a Duplicate?

Now cur points to the node with value 2.

Step 5: Remove Duplicates

• The linked list now looks like this: 1 -> 2 -> 3 -> 3 -> 4 -> 4 -> 5.

• The linked list now looks like: 1 -> 2 -> 3 -> 4 -> 4 -> 5.

• We compare cur.val (1) with cur.next.val (2). They are different, so we move to the next node.

We update cur.next to point to cur.next.next. The duplicate node with value 2 is now skipped.

• We compare cur.val (2) with cur.next.val (also 2). This time they are the same, signaling a duplicate.

• Since cur (1) and cur.next (2) are not None, we enter the while loop.

We want to remove the duplicate values so that each number is unique in the list.

Step 7: Repeat Steps 3 to 5

the next node.

1 -> 2 -> 3 -> 4 -> 5

class ListNode:

class Solution:

our duplicate-free, sorted linked list.

Definition for singly-linked list.

self.value = value

self.next_node = next_node

Traverse the linked list

def __init__(self, value=0, next_node=None):

while current and current.next_node:

current = current.next_node

} else { // Otherwise, move to the next node

current = current.next;

// Return the head of the modified list

ListNode(): val(0), next(nullptr) {}

ListNode* deleteDuplicates(ListNode* head) {

current = current->next;

// Return the head of the modified list

* Definition for singly-linked list node.

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

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

// Function to delete duplicate elements from a sorted linked list

while (current != nullptr && current->next != nullptr) {

if (current->val == current->next->val) {

current->next = current->next->next;

// Otherwise, move to the next node

ListNode* current = head; // Create a pointer to iterate through the list

// Continue iterating as long as the current node and its successor are not null

// If the current node's value equals the next node's value, skip the next node

return head;

* struct ListNode {

class Solution {

int val;

ListNode *next;

} else {

return head;

interface ListNode {

next: ListNode | null;

current = head

else:

return head

Time and Space Complexity

Traverse the linked list

while current and current.next node:

val: number;

* Definition for singly-linked list.

C++

/**

* };

public:

Return the head of the updated list

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

Initialize current to point to the head of the list

"""Remove all duplicates from a sorted linked list such that

each element appears only once and return the modified list."""

If the current value is equal to the value in the next node

Solution Implementation

Step 6: Continue Traversing

Step 9: Return the Updated List

We continue this process for the remaining nodes with value 4 and finally remove all duplicates.

• Now cur points to the node with value 3, and we find that cur.next.val is also 3. We remove the duplicate as before.

Python

if current.value == current.next_node.value: # Bypass the next node as it's a duplicate current.next_node = current.next_node.next_node else: # Move to the next unique value if no duplicate is found

return head

Java

current = head

```
/**
* Definition for singly-linked list.
class ListNode {
              // Value of the node
   int val;
   ListNode next; // Reference to the next node in the list
   // Constructor to create a node with no next node
   ListNode() {}
   // Constructor to create a node with a given value
   ListNode(int val) { this.val = val; }
   // Constructor to create a node with a given value and next node
   ListNode(int val, ListNode next) { this.val = val; this.next = next; }
class Solution {
    * Deletes all duplicates such that each element appears only once.
    * @param head The head of the input linked list.
    * @return The head of the linked list with duplicates removed.
    */
   public ListNode deleteDuplicates(ListNode head) {
       // Initialize current to the head of the linked list
       ListNode current = head;
       // Iterate over the linked list
       while (current != null && current.next != null) {
           // If the current node's value is equal to the value of the next node, skip the next node
           if (current.val == current.next.val) {
               current.next = current.next.next;
```

}; **TypeScript**

/**

/** * Deletes all duplicates such that each element appears only once. * @param {ListNode | null} head - The head of the linked list. * @return {ListNode | null} The modified list head with duplicates removed. */ const deleteDuplicates = (head: ListNode | null): ListNode | null => { let currentNode: ListNode | null = head; // Loop through the list while the current node and the next node are not null while (currentNode && currentNode.next) { // Compare the current node value with the next node value if (currentNode.val === currentNode.next.val) { // If they are equal, skip the next node by pointing to node after next. currentNode.next = currentNode.next.next; } else { // If they are not equal, move to the next node currentNode = currentNode.next; // Return the head of the modified list return head; # Definition for singly-linked list. class ListNode: def ___init___(self, value=0, next_node=None): self.value = value self.next_node = next_node class Solution: def deleteDuplicates(self, head: ListNode) -> ListNode: """Remove all duplicates from a sorted linked list such that each element appears only once and return the modified list.""" # Initialize current to point to the head of the list

If the current value is equal to the value in the next node

Move to the next unique value if no duplicate is found

if current.value == current.next_node.value:

Bypass the next node as it's a duplicate

current.next_node = current.next_node.next_node

The time complexity of the provided code is O(n), where n is the number of nodes in the linked list. This is because it involves a

current = current.next_node

Return the head of the updated list

single traversal through all the nodes of the list, and for each node, it performs a constant amount of work by checking if the next node has a duplicate value and potentially skipping over duplicates. The space complexity of the code is 0(1), as it only uses a fixed amount of additional memory for the cur pointer. No extra space

proportional to the size of the input is needed, whatever the size of the linked list is.