147. Insertion Sort List

Medium Linked List Sorting Leetcode Link

Problem Description

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 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 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:

1. Start with the second element of the list (since a list with one element is already sorted), and for each element—
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.

Intuition

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:

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.
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.
8. Repeat the process until `cur` reaches the end of the list.
9. Return `dummy.next`, which now points to the head of the newly sorted linked list.

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.

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.

The code uses a `dummy` node, which is a common technique when dealing with linked list operations, to simplify handling edge cases like inserting at the beginning of the list.

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.
- 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 right position relative to the `pre` node, and both `pre` and `cur` advance to the next nodes.
- 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.
- The code then uses another pointer, `p`, which starts from the `dummy` and traverses the list to find the insert position. This loop continues until `p.next.val` is greater than `cur.val`, indicating the position where `cur` should be placed.
- Next, `cur` is removed from its current position, and its `next` pointer is stored temporarily in `t`.
- `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.
- `cur` is updated to `t`, and the cycle continues until all elements are sorted.

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.

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 additional data structures.

Example Walkthrough

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

`4 -> 2 -> 1 -> 3`

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

Initial Setup:

- We create a `dummy` node.
- 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`).

Now, we iterate through the list to sort it:

Step 1:

- List: `dummy -> 4 -> 2 -> 1 -> 3`
- Since there's nothing preceding `4`, it's already in the correct position (the sorted part has only one element).
- Move `cur` to the next node (`2`). `pre` still points to the `dummy`.

Step 2:

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

Step 3:

- List: `2 -> 4 -> 1 -> 3` (with `dummy` pointing to `2`)
- 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`.
- Since `1` is less than `2`, it should be inserted at the beginning.
- Connect `dummy.next` to `1`, and `1.next` to `2`.
- The sorted list now is `dummy -> 1 -> 2 -> 4`.

Step 4:

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

After sorting:

- Finally, we have `dummy -> 1 -> 2 -> 3 -> 4`.
- Since `dummy` is our initial node, the head of the sorted list is `dummy.next`, which points to `1`.

The final sorted linked list is:

`1 -> 2 -> 3 -> 4`

This walkthrough demonstrates how the insertion sort algorithm is applied to a singly linked list, efficiently repositioning nodes by manipulating pointers and ensuring the list stays connected throughout the sorting process.

Python Solution

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, value=0, next_node=None):
        self.value = value
        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.
        if head is None or head.next is None:
            return head

        # Initialize a dummy node to help with insertion later.
        dummy_node = ListNode(0)
        dummy_node.next = head

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

        # Iterate while there are nodes to sort.
        while current:
            if previous.value <= current.value:
                # The current node is already in the correct place.
                previous, current = current, current.next
            else:
                # Find the correct place to insert the current node.
                insert_position = dummy_node
                while insert_position.next.value <= current.value:
                    insert_position = insert_position.next

                # Perform the insertion.
                temp = current.next
                current.next = insert_position.next
                insert_position.next = current
                previous.next = temp

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

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

```

Java Solution

```java
class Solution {
    public ListNode insertionSortList(ListNode head) {
        // If the list is empty or has only one item, it's already sorted.
        if (head == null || head.next == null) {
            return head;
        }

        // Initialize the dummy node with an initial value.
        ListNode dummy = new ListNode(0);
        // Previous and current pointers for traversal.
        ListNode previous = dummy, current = head;

        // Iterate over the list to sort each element.
        while (current != null) {
            // If the value at previous is less than or equal to the current's value, move the pointers forward.
            if (previous.val <= current.val) {
                previous = current;
                current = current.next;
                continue;
            }
            // Find the correct position for the current node in the sorted part of the list
            ListNode position = dummy;
            while (position.next != null && position.next.val <= current.val) {
                position = position.next;
            }
            // Insert current node in the correct position.
            ListNode temp = current.next;
            current.next = position.next;
            position.next = current;

            // Move the previous pointer's next to temp and update current to temp.
            previous.next = temp;
            current = temp;
        }
        // Return the next node of dummy since the first node is a placeholder.
        return dummy.next;
    }
}

// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next;
    // Constructor to initialize a node with no next element.
    ListNode() {}

    // Constructor to initialize a node with a value.
    ListNode(int val) { this.val = val; }

    // Constructor to initialize a node with a value and a next element.
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

```

C++ Solution

```cpp
// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;

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

/**
 * Sorts a linked list using the insertion sort algorithm.
 * @param head - The head of the singly linked list to be sorted.
 * @return The head of the linked list sorted in ascending order.
 */
ListNode* insertionSortList(ListNode* head) {
    // If the list is empty or has only one node, no sorting is needed.
    if (!head || !(head->next)) {
        return head;
    }

    // The dummy node is used as a placeholder to help with inserting nodes at the beginning of the list.
    ListNode dummy(0);
    ListNode *previousNode = &dummy;
    ListNode *currentNode = head;

    while (currentNode) {
        // If the current node is already in the correct position, just move forward.
        if (previousNode->val <= currentNode->val) {
            previousNode = currentNode;
            currentNode = currentNode->next;
            continue;
        }

        // Find the correct position to insert the current node by traversing from the start of the list.
        ListNode *positionToInsert = &dummy;
        while (positionToInsert->next && positionToInsert->next->val <= currentNode->val) {
            positionToInsert = positionToInsert->next;
        }

        // Perform the insertion.
        ListNode *nextNode = currentNode->next;
        currentNode->next = positionToInsert->next;
        positionToInsert->next = currentNode;

        // Link the previous part to the new next position as needed.
        // 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
        // is the first node being inserted in the sorted part.
        // After insertion, the previous (dummy) node directly points to the new current node.
        if(previousNode != currentNode)
            previousNode->next = nextNode;

        // Move the current node to the next position.
        currentNode = nextNode;
    }

    // The dummy's next node is now the head of the sorted list.
    return dummy.next;
}

// The above function can be utilized by creating `ListNode` objects and passing the head to `insertionSortList`.

```

Typescript Solution

```typescript
/**
 * Definition for singly-linked list.
 */
class ListNode {
    val: number;
    next: ListNode | null;

    constructor(val: number = 0, next: ListNode | null = null) {
        this.val = val;
        this.next = next;
    }
}

/**
 * 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.
 */
const insertionSortList = (head: ListNode | null): ListNode | null => {
    // If the list is empty or has only one node, no sorting is needed.
    if (head === null || head.next === null) {
        return head;
    }

    // The dummy node is used as a placeholder to help with inserting nodes at the beginning of the list.
    let dummy: ListNode = new ListNode(0);
    let previousNode: ListNode = dummy;
    let currentNode: ListNode | null = head;

    while (currentNode !== null) {
        // If the current node is already in the correct position, just move forward.
        if (previousNode.val <= currentNode.val) {
            previousNode = currentNode;
            currentNode = currentNode.next;
            continue;
        }

        // 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) {
            positionToInsert = positionToInsert.next;
        }

        // Perform the insertion.
        let nextNode: ListNode | null = currentNode.next;
        currentNode.next = positionToInsert.next;
        positionToInsert.next = currentNode;

        // Link the previous part to the next position.
        previousNode.next = nextNode;

        // Move the current node to the next position.
        currentNode = nextNode;
    }

    // The dummy's next node is now the head of the sorted list.
    return dummy.next;
};

// The above function can be utilized by creating `ListNode` objects and passing the head to `insertionSortList`.

```

Time and Space Complexity

The time complexity of the insertion sort algorithm is generally `O(n^2)` for a linked list, where `n` is the number of elements in the linked list. This is because the algorithm consists of two nested loops: the outer loop runs once for every element, and the inner loop 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 `O(n^2)`.

The space complexity of the code is `O(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.