86. Partition List

Medium Linked List Two Pointers Leetcode Link

Problem Description

The problem presents us with a singly linked list and a value `x`. Our task is to rearrange the nodes in the linked list in such a way that all nodes having values less than `x` are placed before the nodes with values greater than or equal to `x`. Importantly, we need to maintain the original relative order of the nodes that are less than `x` and those that are greater than or equal to `x`.

Intuition

The primary intuition for solving this problem is to create two separate linked lists:

1. The first for nodes with values less than `x` (we'll call this list "smaller").
2. The second for nodes with values greater than or equal to `x` (we'll call this list "greater").

As we iterate through the original list, we evaluate each node's value. If a node's value is less than `x`, we append it to the "smaller" list. If a node's value is greater than or equal to `x`, we append it to the "greater" list. After iterating through the whole list, we connect the end of the "smaller" list to the beginning of the "greater" list to form the final reordered linked list. This approach ensures that the original relative ordering is preserved within each partition.

It is important to be careful with edge cases, such as when the linked list has no nodes or all nodes are smaller or larger than `x`. However, the approach will correctly handle these scenarios as the lists will simply be empty or one of them will not be used.

Solution Approach

In the given solution approach, we use two dummy nodes `d1` and `d2` as the heads of the two new lists: one for elements less than `x` (the "smaller" list) and one for elements greater than or equal to `x` (the "greater" list). Using these dummy nodes allows us to append to these lists without having to check if they are empty.

We then iterate through the original linked list with the following steps:

1. We check the value of the current node pointed to by `head`.
2. If the value is less than `x`, we move the node to the end of the "smaller" list, done by setting `t1.next` to the current node, and then advancing `t1` to `t1.next`.
3. If the value is greater than or equal to `x`, we move the node to the end of the "greater" list, executed by setting `t2.next` to the current node, and then advancing `t2` to `t2.next`.
4. In both cases, after moving the node, we advance `head` to `head.next` to continue to the next node in the list.

Once we finish iterating through the entire list, we merge the two lists. We set the `next` pointer of the tail of the "smaller" list (`t1.next`) to the head of the "greater" list (`d2.next`). It is crucial then to set the next of the last node in the "greater" list to `None` to indicate the end of the linked list.

This partitioning keeps the nodes in their original relative order within the two partitions because nodes are moved individually and the next pointers of the original list are preserved. The code concludes by returning `d1.next`, which is the head of the merged, partitioned list (excluding the dummy head `d1`).

This approach uses the following patterns:

- **Two Pointer Technique**: Having separate pointers `t1` and `t2` that keep track of the current end of the "smaller" and "greater" lists, respectively, allows us to efficiently append to these lists.
- **Sentinel Node (Dummy Node)**: By using dummy head nodes `d1` and `d2`, we avoid having to write special case code for when the heads of the "smaller" or "greater" lists are `None`.

In essence, the algorithm separates and then recombines the linked list's nodes based on their values in relation to `x`, while maintaining their original relative order.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we are given the following linked list and the value `x = 3`:

```
Linked List: 1 -> 4 -> 3 -> 2 -> 5
```

Our task is to rearrange this list so that all nodes with values less than 3 are before nodes with values greater than or equal to 3, while maintaining their original relative order.

We start by initiating two dummy nodes, `d1` and `d2`, which will serve as the heads of our "smaller" and "greater" lists. We also have two pointers `t1` and `t2` that start at these dummy nodes respectively.

```
d1 -> None
d2 -> None
t1 -> d1
t2 -> d2
```

Now we iterate over the original list with the help of `head` pointer:

- `head` points to `1`, which is less than `3`, so we attach `head` to the "smaller" list and move `t1`.
```
smaller list: d1 -> 1
greater list: d2 -> None
t1 now points to 1
t2 still points to d2
```
- `head` now points to `4`, which is greater than or equal to `3`, so we attach `head` to the "greater" list and move `t2`.
```
smaller list: d1 -> 1
greater list: d2 -> 4
t1 still points to 1
t2 now points to 4
```
- `head` now points to `3`, which is equal to `3`, so `head` is attached to the "greater" list and `t2` is moved.
```
smaller list: d1 -> 1
greater list: d2 -> 4 -> 3
t1 still points to 1
t2 now points to 3
```
- `head` now points to `2`, which is less than `3`, so `head` is attached to the "smaller" list and `t1` is moved.
```
smaller list: d1 -> 1 -> 2
greater list: d2 -> 4 -> 3
t1 now points to 2
t2 still points to 3
```
- Finally, `head` points to `5`, which is greater than `3`, so `head` is attached to the "greater" list and `t2` is moved.
```
smaller list: d1 -> 1 -> 2
greater list: d2 -> 4 -> 3 -> 5
t1 still points to 2
t2 now points to 5
```

After iterating through the entire list, we connect the "smaller" list to the "greater" list:

- `t1.next` (which is `2.next`) is set to `d2.next` (which is `4`), merging the lists.
- `t2.next` (which is `5.next`) is set to `None`, indicating the end of the linked list.

The final merged list, omitting the dummy node `d1`, is:

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

This is the reordered list where nodes with values less than `3` are placed before nodes with values greater than or equal to `3`, and their relative order is maintained as per the problem requirement.

Python Solution

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def partition(self, head: Optional[ListNode], x: int) -> Optional[ListNode]:
        # Dummy nodes to start the lists for elements less than x and not less than x
        less_head = ListNode()
        not_less_head = ListNode()

        # Tail pointers for each list, which will help in appending new nodes
        less_tail = less_head
        not_less_tail = not_less_head

        # Traverse the original list
        while head:
            # If the current value is less than x, append it to the list of less_tail
            if head.val < x:
                less_tail.next = head
                less_tail = less_tail.next
            # If the current value is not less than x, append it to the list of not_less_tail
            else:
                not_less_tail.next = head
                not_less_tail = not_less_tail.next
            # Move to the next node in the original list
            head = head.next

        # Connect the two lists together
        less_tail.next = not_less_head.next
        # The last node of the new list should point to None to indicate the end of the list
        not_less_tail.next = None

        # Return the head of the list with nodes less than x followed by nodes not less than x
        return less_head.next

```

Java Solution

```java
// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next;

    // Constructor to initialize the node with a value.
    ListNode() {}

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

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

class Solution {

    // This method partitions a linked list around a value x, such that all nodes less than x come before nodes greater than or equal
    public ListNode partition(ListNode head, int x) {
        ListNode lessHead = new ListNode(0); // Dummy node for the 'less' sublist.
        ListNode greaterHead = new ListNode(0); // Dummy node for the 'greater' sublist.
        ListNode lessTail = lessHead; // Tail pointer for the 'less' sublist.
        ListNode greaterTail = greaterHead; // Tail pointer for the 'greater' sublist.

        // Iterate over the original list and divide the nodes into 'less' and 'greater' sublists.
        while (head != null) {
            if (head.val < x) {
                // Append to 'less' sublist.
                lessTail.next = head;
                lessTail = head;
            } else {
                // Append to 'greater' sublist.
                greaterTail.next = head;
                greaterTail = head;
            }
            head = head.next; // Move to the next node in the original list.
        }

        // Connect the 'less' sublist with the 'greater' sublist.
        lessTail.next = greaterHead.next;
        greaterTail.next = null; // Ensure the last node of 'greater' sublist points to null.

        // Return the head of the 'less' sublist, which now contains all nodes in the partitioned order.
        return lessHead.next;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int value;
 *     ListNode *next;
 *     ListNode() : value(0), next(nullptr) {}
 *     ListNode(int x) : value(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : value(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        // Create two dummy head nodes for the two partitions
        ListNode* lessHead = new ListNode(); // Head of the list for elements < x
        ListNode* greaterHead = new ListNode(); // Head of the list for elements >= x

        // Use two pointers to keep track of the current end of the two partitions
        ListNode* lessTail = lessHead; // Tail of the list for elements < x
        ListNode* greaterTail = greaterHead; // Tail of the list for elements >= x

        // Iterate through the original list and separate nodes into the two partitions
        while (head) {
            if (head->value < x) {
                // Append to the less-than partition
                lessTail->next = head;
                lessTail = lessTail->next;
            } else {
                // Append to the greater-or-equal partition
                greaterTail->next = head;
                greaterTail = greaterTail->next;
            }
            // Move to the next node in the original list
            head = head->next;
        }

        // Connect the two partitions
        lessTail->next = greaterHead->next;

        // Ensure the end of the greater-or-equal partition is null
        greaterTail->next = nullptr;

        // The start of the less-than partition is the new head of the modified list
        ListNode* newHead = lessHead->next;

        // Clean up the dummy head nodes
        delete lessHead;
        delete greaterHead;

        return newHead;
    }
};

```

Typescript Solution

```typescript
// ListNode class definition for TypeScript
class ListNode {
    val: number;
    next: ListNode | null;

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

/**
 * Partition a linked list around a value x, such that all nodes less than x come
 * before nodes greater than or equal to x.
 * @param {ListNode | null} head - The head of the input linked list.
 * @param {number} x - The partition value, where elements are repositioned around.
 * @return {ListNode | null} - The head of the modified linked list.
 */
const partition = (head: ListNode | null, x: number): ListNode | null => {
    // Two dummy nodes to start the less and greater sublist
    const lessDummy: ListNode = new ListNode();
    const greaterDummy: ListNode = new ListNode();

    // Two pointers to track the current end node of the sublists
    let lessTail: ListNode = lessDummy;
    let greaterTail: ListNode = greaterDummy;

    // Iterate through the linked list and partition the nodes
    while (head) {
        if (head.val < x) {
            // If current node value is less than x, add it to the less sublist
            lessTail.next = head;
            lessTail = lessTail.next;
        } else {
            // If current node value is greater or equal to x, add it to the greater sublist
            greaterTail.next = head;
            greaterTail = greaterTail.next;
        }
        // Move to the next node in the list
        head = head.next;
    }

    // Connect the end of the less sublist to the beginning of the greater sublist
    lessTail.next = greaterDummy.next;

    // The last node of the greater sublist should point to null to end the list
    greaterTail.next = null;

    // Return the head of the less sublist, since it is now the head of the partitioned list
    return lessDummy.next;
};

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the linked list. This is because the code iterates through all the nodes exactly once, performing a constant amount of work for each node by checking the value and rearranging the pointers.

Space Complexity

The space complexity of the code is `O(1)`. Despite creating two dummy nodes (`d1` and `d2`) and two tail pointers (`t1` and `t2`), the amount of extra space used does not scale with the size of the input (the number of nodes `n`); rather, it remains constant. The rearrangement of the nodes is done in place without allocating any additional nodes.