Day 127/180

LinkedList Problems 

---

1. Partition a Linked List around a given value

```
struct Node* partition(struct Node* head, int x)
{
    /* Let us initialize first and last nodes of
       three linked lists
        1) Linked list of values smaller than x.
        2) Linked list of values equal to x.
        3) Linked list of values greater than x.*/
    struct Node *smallerHead = NULL, *smallerLast = NULL;
    struct Node *greaterLast = NULL, *greaterHead = NULL;
    struct Node *equalHead = NULL, *equalLast = NULL;

    // Now iterate original list and connect nodes
    // of appropriate linked lists.
    while (head != NULL) {
        // If current node is equal to x, append it
        // to the list of x values
        if (head->data == x) {
            if (equalHead == NULL)
                equalHead = equalLast = head;
            else {
                equalLast->next = head;
                equalLast = equalLast->next;
```

```
        }
    }

    // If current node is less than X, append
    // it to the list of smaller values
    else if (head->data < x) {
        if (smallerHead == NULL)
            smallerLast = smallerHead = head;
        else {
            smallerLast->next = head;
            smallerLast = head;
        }
    }
    else // Append to the list of greater values
    {
        if (greaterHead == NULL)
            greaterLast = greaterHead = head;
        else {
            greaterLast->next = head;
            greaterLast = head;
        }
    }

    head = head->next;
}

// Fix end of greater linked list to NULL if this
// list has some nodes
if (greaterLast != NULL)
    greaterLast->next = NULL;
```

```
    // Connect three lists

    // If smaller list is empty
    if (smallerHead == NULL) {
        if (equalHead == NULL)
            return greaterHead;
        equalLast->next = greaterHead;
        return equalHead;
    }

    // If smaller list is not empty and equal list is empty
    if (equalHead == NULL) {
        smallerLast->next = greaterHead;
        return smallerHead;
    }

    // If both smaller and equal list are non-empty
    smallerLast->next = equalHead;
    equalLast->next = greaterHead;
    return smallerHead;
}
```

Code Explanation and Complexity

To solve this problem we can use the partition method of Quick Sort but this would not preserve the original relative order of the nodes in each of the two partitions.

Below is the algorithm to solve this problem :

- Initialize first and last nodes of below three linked lists as NULL.
    a. Linked list of values smaller than x.
    b. Linked list of values equal to x.

c. Linked list of values greater than x.

- Now iterate through the original linked list. If a node's value is less than x then append it at the end of the smaller list. If the value is equal to x, then at the end of the equal list. And if a value is greater, then at the end of the greater list.
- Now concatenate three lists.

Time complexity: O(n) where n is the size of the linked list
Auxiliary Space: O(n)

2. Subtraction in Linked List

```
class Solution {
public:

  void printList(Node* n) {
    while (n) {
      cout << n->data;
      n = n->next;
    }
    cout << endl;
  }

  int length(Node* n) {
    int ret = 0;
    while (n) {
      ret++;
      n = n->next;
    }
    return ret;
  }

  Node* reverse(Node* head) {
    Node* prev =  NULL;
    Node* current = head;
    Node* next =  NULL;

    while (current !=  NULL) {
      next = current->next; // storing next node
      current->next = prev; // linking current node to previous
```

```
        prev = current; // updating prev
        current = next; // updating current
    }

    return prev;
}

Node* subLinkedList(Node* l1, Node* l2) {

    while (l1 !=  NULL && l1->data == 0) {
        l1 = l1->next;
        // removing trailing zeroes from l1
    }

    while (l2 !=  NULL && l2->data == 0) {
        l2 = l2->next;
        // removing trailing zeroes from l2
    }

    int n1 = length(l1);
    int n2 = length(l2);

    if (n1 == 0 && n2 == 0) {
        return new Node(0);
    }

    if (n2 > n1) {
        std::swap(l1, l2);
        // making sure l1 list has the bigger number
    }

    if (n1 == n2) {
        Node* t1 = l1;
        Node* t2 = l2;
        while (t1->data == t2->data) {
            // finding which number is greater
            t1 = t1->next;
            t2 = t2->next;
```

```
            if (t1 ==  NULL) {
                return new Node(0);
                // returning zero if both numbers are same
            }
        }

        if (t2->data > t1->data) {
            std::swap(l1, l2);
            // making sure l1 list has the bigger number
        }
    }

    l1 = reverse(l1);
    l2 = reverse(l2);

    Node* res =  NULL;
    Node* t1 = l1;
    Node* t2 = l2;

    while (t1 !=  NULL) {
        int small = 0;
        if (t2 !=  NULL) {
            small = t2->data;
            // 'small' holds the next digit of number to be subtracted
        }

        if (t1->data < small) {
            t1->next->data -= 1;
            t1->data += 10;
            // taking carry
        }

        Node* n = new Node(t1->data - small);
        // creating new node for storing difference
        n->next = res;
        res = n;

        t1 = t1->next;
        if (t2 !=  NULL) {
```

```
            t2 = t2->next;
        }
    }

    while (res->next !=  NULL && res->data == 0) {
        res = res->next;
        // removing trailing zeroes from result list
    }
    return res;
  }
};
```

Code Explanation and Complexity

Intuition

The idea is to reverse the linked list and then subtract them.

Implementation

1. Remove trailing zeros from both linked lists to ensure that any leading zeros in the reversed linked lists do not affect the subtraction process.
2. Determine the lengths of the two linked lists (n1 and n2).
    - If the length of l2 is greater than the length of l1, swap the lists to ensure that l1 always represents the larger number.
    - If the lengths of the two linked lists are equal, iterate through them to find which number is greater. If both numbers are equal, return a new linked list with a single node containing the value 0.
3. Reverse both linked lists (l1 and l2) using the reverse method. This step simplifies the subtraction process, allowing it to be performed from left to right.
4. Iterate through the reversed linked lists (t1 and t2) while subtracting corresponding digits.
5. If the digit in t1 is less than the corresponding digit in t2, borrow 10 from the next higher digit in t1.
6. Create a new node for the difference and update the result linked list (res).
7. After the subtraction is complete, remove trailing zeros from the result linked list.
8. Return the resulting linked list, representing the subtraction of the two input linked lists.

Time Complexity: O(max(n1, n2)) - Dominated by the iteration through the linked lists for subtraction.

Space Complexity: O(max(n1, n2)) - Mainly due to the space required for the result linked list.

_____ END _____