Day 120/180 LinkedList Problems

1: Merge Lists Alternatingly

The mergeList function takes two linked lists, listA and listB, and merges them by interleaving nodes from both lists. It iterates through both lists, connecting nodes from listB into listA. After merging, the pointer to listB is updated to point to the remaining nodes in listB. The merged result is formed by interleaving nodes from listA and listB.

```
void mergeList(struct Node **firstList, struct Node **secondList)
{
    Node* firstListCurrent = (*firstList);
    Node* secondListCurrent = (*secondList);

    Node* firstListNext = NULL;
    Node* secondListNext = NULL;

    while (firstListCurrent != NULL && secondListCurrent != NULL) {
        // Save the next nodes in each list
        firstListNext = firstListCurrent->next;
        secondListNext = secondListCurrent->next;

        // Connect the nodes from the second list into the first list
        firstListCurrent->next = secondListCurrent;
        secondListCurrent->next = firstListNext;

        // Move to the next nodes in both lists
        firstListCurrent = firstListNext;
        secondListCurrent = secondListNext;
    }

    // Update the pointer of the second list to the remaining nodes
    (*secondList) = secondListCurrent;
    return;
}
```

2: Rearrange a linked list

The rearrangeEvenOdd function separates a linked list into two parts: odd-indexed nodes and even-indexed nodes. It achieves this by iterating through the list and creating two separate sublists for odd and even nodes. Finally, it connects the last odd node to the first even node, producing the rearranged linked list.

```
void rearrangeEvenOdd(Node *head)
{
    // Check if the linked list is empty or has only one node
    if (head == NULL || head->next == NULL)
        return;

    // Initialize pointers for odd and even nodes
    Node *odd = head;
    Node *even = head->next;

    // Move the head to the next node after even
    head = head->next->next;

    // Disconnect odd and even nodes from the rest of the list
    odd->next = NULL;
    even->next = NULL;

    // Temporary pointers to keep track of odd and even nodes
    Node *t1 = odd;
    Node *t2 = even;

    // Iterate through the remaining nodes in the original list
    while (head != NULL)
```

```
    {
        // Connect the odd node to the next node and move
pointers
        t1->next = head;
        head = head->next;
        t1 = t1->next;
        t1->next = NULL;

        // Check if there is another node in the list
        if (head != NULL)
        {
            // Connect the even node to the next node and move
pointers
            t2->next = head;
            head = head->next;
            t2 = t2->next;
            t2->next = NULL;
        }
    }

    // Connect the last odd node to the even nodes
    t1->next = even;
}
```

3: Given a linked list of 0s, 1s and 2s, sort it (Solve the 3rd problem without using count sort and do it in-place).

The function uses three separate linked lists (zeroHead, oneHead, and twoHead) to temporarily store nodes with values 0, 1, and 2, respectively. It then iterates through the original linked list (head), placing each node in the appropriate temporary list based on its value.

After processing all nodes, it connects the temporary lists to form a new linked list where nodes with a value of 0 come first, followed by nodes with a value of 1, and then nodes with a value of 2.

Finally, the function returns the head of the rearranged linked list.

```
Node* segregate(Node *head) {

        // Add code here
       if (head == nullptr || head->next == nullptr)
        return head;

    Node* zeroHead = new Node(-1);
    Node* zero = zeroHead;
    Node* oneHead = new Node(-1);
    Node* one = oneHead;
    Node* twoHead = new Node(-1);
    Node* two = twoHead;

    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == 0) {
            zero->next = temp;
            zero = temp;
        } else if (temp->data == 1) {
            one->next = temp;
            one = temp;
        } else {
            two->next = temp;
            two = temp;
        }
```

```
        temp = temp->next;
    }

    zero->next = (oneHead->next != nullptr) ? oneHead->next :
twoHead->next;
    one->next = twoHead->next;
    two->next = nullptr;

    Node* newHead = zeroHead->next;
    return newHead;

    }
```