C 语言链表操作题解

目录

题目 A：删除重复的节点

题目描述

给定一个按升序排列的链表，链表中可能包含一些重复的节点。请编写一个函数，删除链表中所有重复的节点，使得最终链表中只包含唯一出现过的节点。注意，重复的节点不保留。

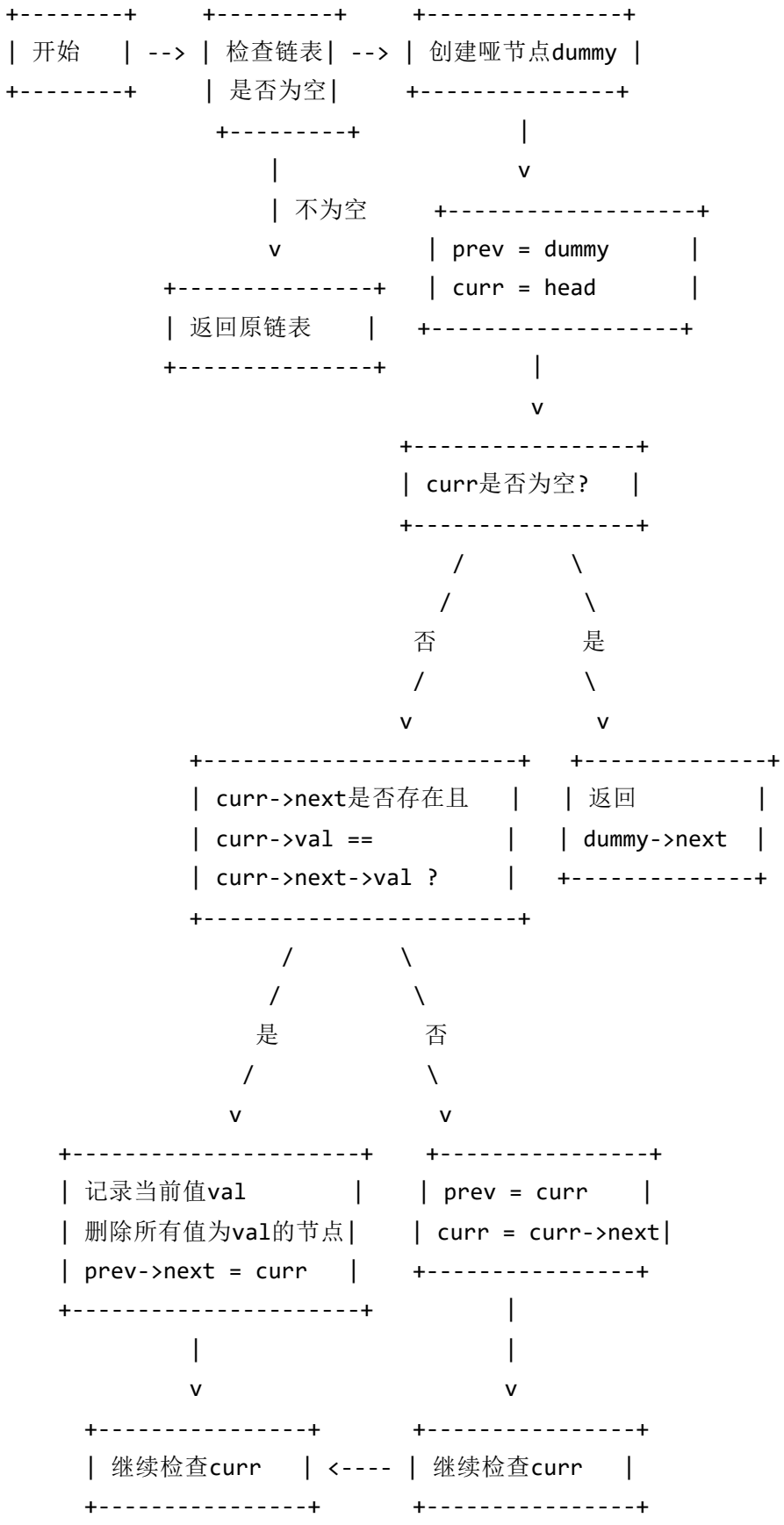
思路分析

这道题目的核心是要删除所有重复出现的节点，而不是仅保留一个。也就是说，如果有多个节点的值相同，这些节点都要被删除。

解题思路如下：

1. 由于链表是有序的，所以相同的节点一定是连续的。
2. 我们可以使用一个哑节点（dummy node）作为链表的新头部，这样可以统一处理链表头部可能被删除的情况。
3. 使用两个指针： `prev` 和 `curr`
    - `prev` 指向上一个确定不重复的节点
    - `curr` 用于遍历链表
4. 当发现 `curr` 与其下一个节点值相同时，记录这个值，然后删除所有具有该值的节点。
5. 如果 `curr` 与下一个节点值不同，则移动两个指针。

流程图


开始
检查链表是否为空
不为空
返回原链表
创建哑节点dummy
prev = dummy
curr = head
curr是否为空?
否
是
curr->next是否存在且
curr->val ==
curr->next->val ?
返回
dummy->next
是
否
记录当前值val
删除所有值为val的节点
prev->next = curr
prev = curr
curr = curr->next
继续检查curr
继续检查curr

代码分析

```
ListNode *delete_duplicates(ListNode *head)
{
    // 如果链表为空或只有一个节点，无需处理
    if (!head || !head->next)
    {
        return head;
    }

    // 创建哑节点，统一处理头节点可能被删除的情况
    ListNode *dummy = new ListNode(0);
    dummy->next = head;

    ListNode *prev = dummy; // 前一个确定不重复的节点
    ListNode *curr = head;  // 当前遍历的节点

    while (curr)
    {
        // 如果当前节点与下一个节点值相同，说明有重复
        if (curr->next && curr->val == curr->next->val)
        {
            int val = curr->val; // 记录重复值

            // 删除所有值为 val 的节点
            while (curr && curr->val == val)
            {
                ListNode *temp = curr;  // 保存当前节点以便删除
                curr = curr->next;      // 移动到下一个节点
                delete temp;            // 释放内存
            }

            // 将前一个节点直接连接到下一个不重复的节点
            prev->next = curr;
        }
        else
        {
            // 如果当前节点不重复，移动两个指针
            prev = curr;
            curr = curr->next;
        }
    }
```

```
    // 获取新的头节点
    ListNode *new_head = dummy->next;
    delete dummy; // 删除哑节点，避免内存泄漏

    return new_head;
}
```

易错提醒

1. **内存泄漏**：删除节点时，必须先保存节点指针，再移动当前指针，最后释放内存。否则会导致内存泄漏。
2. **边界条件处理**：
   - 链表为空时的处理
   - 只有一个节点时的处理
   - 链表头部节点需要被删除时的处理
3. **指针更新顺序**：当删除节点时，必须先更新指针，再释放内存，否则会造成指针悬挂。
4. **哑节点的使用**：使用哑节点可以简化代码逻辑，特别是处理头节点可能被删除的情况。但记得最后要释放哑节点的内存。
5. **循环结束条件**：在循环中删除节点后，要确保 `curr` 指针移动到了下一个有效位置。
6. **连接剩余节点**：删除一串重复节点后，记得将 `prev->next` 指向 `curr` ，保持链表的连续性。

题目 B：第一个公共结点

题目描述

输入两个链表，编写一个函数来找出它们的第一个公共结点。第一个公共结点的定义：从这一结点开始，两个链表的后续结点序列的值相同。如果两个链表没有公共结点，则返回 NULL。

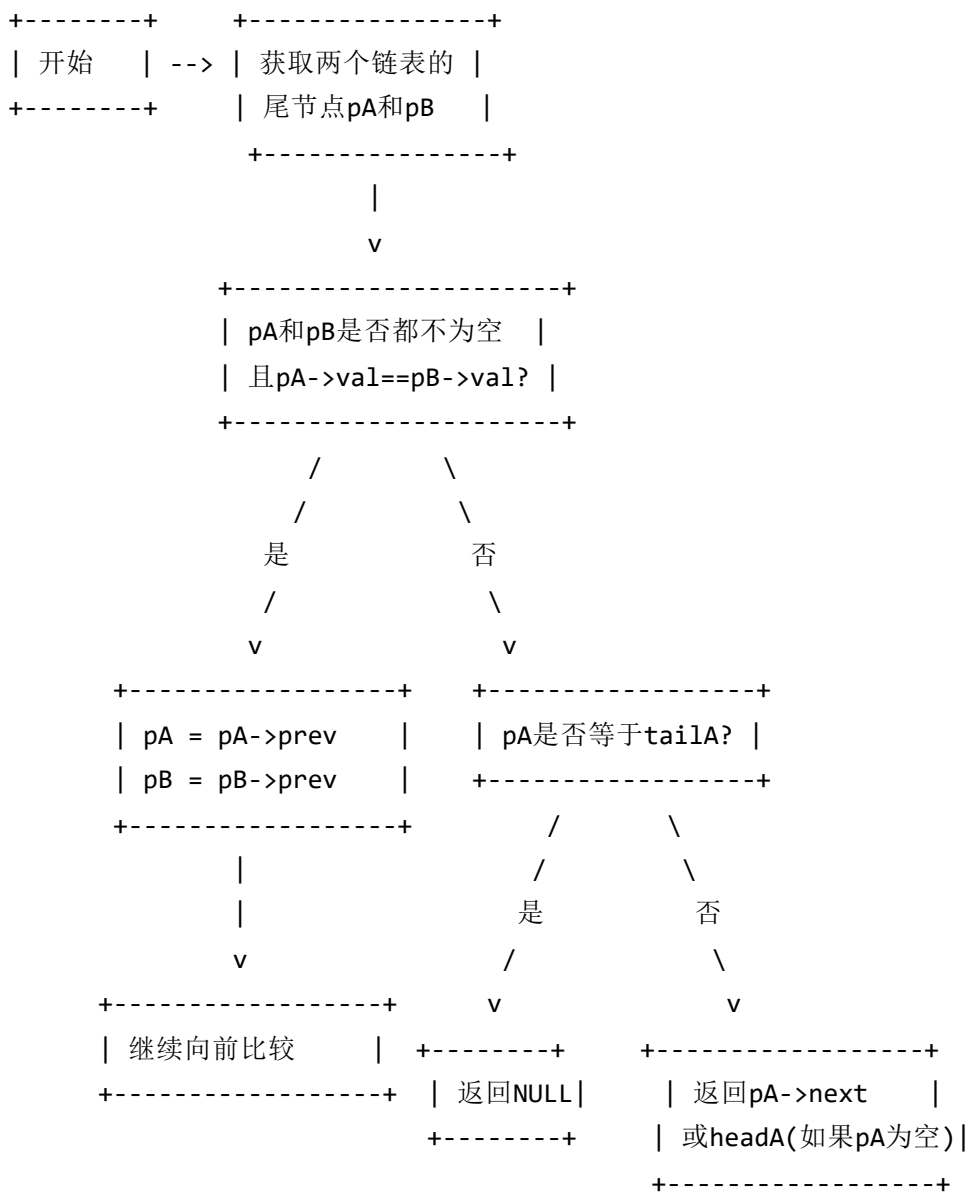
思路分析

本题使用的是双向链表，每个节点都有 `prev` 和 `next` 指针。我们需要找到两个链表的第一个公共节点，也就是从这个节点开始，后面的所有节点都是共用的。

解题思路：

1. 由于题目中的链表是基于节点值比较的，而不是节点地址，我们需要从链表尾部开始比较。
2. 两个链表的公共部分如果存在，必然是从某个节点开始到链表尾部的一段。
3. 从两个链表的尾部开始，向前比较，直到找到第一个不相等的节点。
4. 这个不相等节点的下一个节点就是第一个公共节点。

流程图


开始
获取两个链表的
尾节点pA和pB
pA和pB是否都不为空
且pA->val==pB->val?
是
否
pA = pA->prev
pB = pB->prev
pA是否等于tailA?
是
否
继续向前比较
返回NULL
返回pA->next
或headA(如果pA为空)

代码分析

```
ListNode *findFirstCommonNode(ListNode *headA, ListNode *tailA,
                              ListNode *headB, ListNode *tailB)
{
    // 从尾部开始比较
    ListNode *pA = tailA;
    ListNode *pB = tailB;

    // 向前走，只要值相等就继续
    while (pA && pB && pA->val == pB->val)
    {
        pA = pA->prev;
        pB = pB->prev;
    }

    // 如果根本没有匹配（一次循环都没进），则没有公共后缀
    if (pA == tailA)
    {
        return NULL;
    }

    // 否则公共后缀的起点就是 pA 的下一个结点
    // （当比较失败跳出时，pA 已经指向第一个不匹配的结点）
    return pA ? pA->next : headA;
}
```

易错提醒

1. **理解题意**：本题中的"公共节点"是基于节点值比较，而不是节点地址。两个链表的对应节点值相同，就认为是公共部分。
2. **从尾部开始比较**：公共部分一定是从某个节点开始到链表尾部的一段，所以应该从尾部开始比较。
3. **循环条件**：在循环中需要检查三个条件：
   - `pA` 不为空
   - `pB` 不为空
   - `pA->val` 等于 `pB->val`
4. **返回值处理**：
   - 如果没有公共节点（`pA == tailA`），返回 NULL。
   - 否则，返回第一个公共节点，即 `pA->next`。
   - 特殊情况：如果 `pA` 为空，说明整个链表 A 都是公共部分，返回 `headA`。

5. **边界情况**：考虑一个链表是另一个链表的子链表的情况。
6. **双向链表操作**：熟悉双向链表的 `prev` 和 `next` 指针的用法。

题目 C：把字符串转换成整数

题目描述

请你写一个函数 StrToInt，实现把字符串转换成整数这个功能。需要处理以下情况：

1. 空字符串：空输入。
2. 普通数字：基本转换。
3. 前导零：去掉前导零的逻辑。
4. 负号：需处理负号。
5. 前导空格：空格忽略。
6. 非数字字符：只提取数字部分。

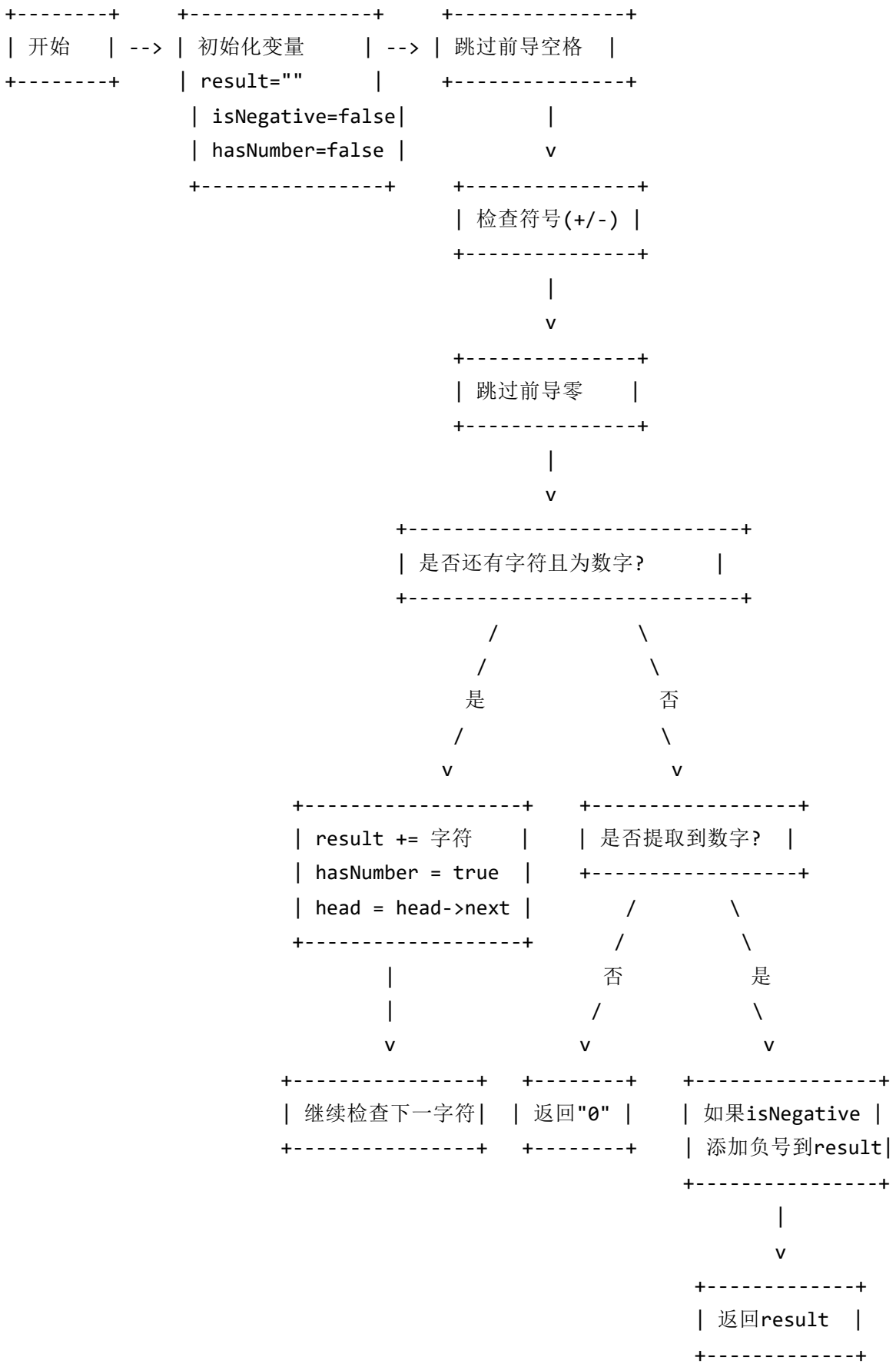
思路分析

这道题目要求我们将字符串转换为整数，需要处理多种特殊情况。题目中使用了链表来存储字符串，我们需要按顺序遍历链表节点并进行相应处理。

解题思路：

1. 跳过前导空格。
2. 处理可能存在的正负号。
3. 跳过前导零。
4. 提取连续的数字字符，构建结果字符串。
5. 如果没有有效数字，返回"0"；否则，根据符号添加正负号并返回结果。

流程图


开始
初始化变量
result=""
isNegative=false
hasNumber=false
跳过前导空格
检查符号(+/-)
跳过前导零
是否还有字符且为数字?
是
否
result += 字符
hasNumber = true
head = head->next
是否提取到数字?
否
是
继续检查下一字符
返回"0"
如果isNegative
添加负号到result
返回result

代码分析

```
string StrToInt(CharNode *head)
{
    string result;
    bool isNegative = false;
    bool hasNumber = false;

    // 跳过前导空格
    while (head && head->val == ' ')
    {
        head = head->next;
    }

    // 处理符号
    if (head && (head->val == '+' || head->val == '-'))
    {
        if (head->val == '-')
        {
            isNegative = true;
        }
        head = head->next;
    }

    // 跳过前导零
    while (head && head->val == '0')
    {
        head = head->next;
    }

    // 提取数字
    while (head && isdigit(head->val))
    {
        result += head->val;
        hasNumber = true;
        head = head->next;
    }

    // 如果没有任何数字，返回0
    if (!hasNumber)
    {
        return "0";
    }
```

```
    // 添加符号
    if (isNegative)
    {
        result = "-" + result;
    }

    return result;
}
```

易错提醒

1. **空指针检查**：在每次使用 `head` 之前，必须先检查它是否为空，避免空指针引用。
2. **前导空格处理**：需要跳过所有的前导空格，直到遇到非空格字符。
3. **符号处理**：
   - 只处理第一个符号（在跳过空格后）
   - 如果是负号，设置 `isNegative` 标志
   - 处理完符号后，移动指针到下一个节点
4. **前导零处理**：在处理完符号后，需要跳过所有前导零。
5. **数字提取**：
   - 使用 `isdigit()` 函数检查字符是否为数字
   - 遇到第一个非数字字符时停止
   - 使用 `hasNumber` 标志记录是否提取到了数字
6. **空输入处理**：如果没有提取到任何数字（包括输入为空、只有符号、只有前导零等情况），返回"0"。
7. **符号添加**：只有在 `isNegative` 为真时，才在结果前添加负号。

题目 D：左旋字符串

题目描述

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请编写一个函数实现字符串左旋转操作的功能。比如输入字符串 "abcdefg" 和数字 2，该函数将返回左旋转 2 位得到的结果 "cdefgab"。

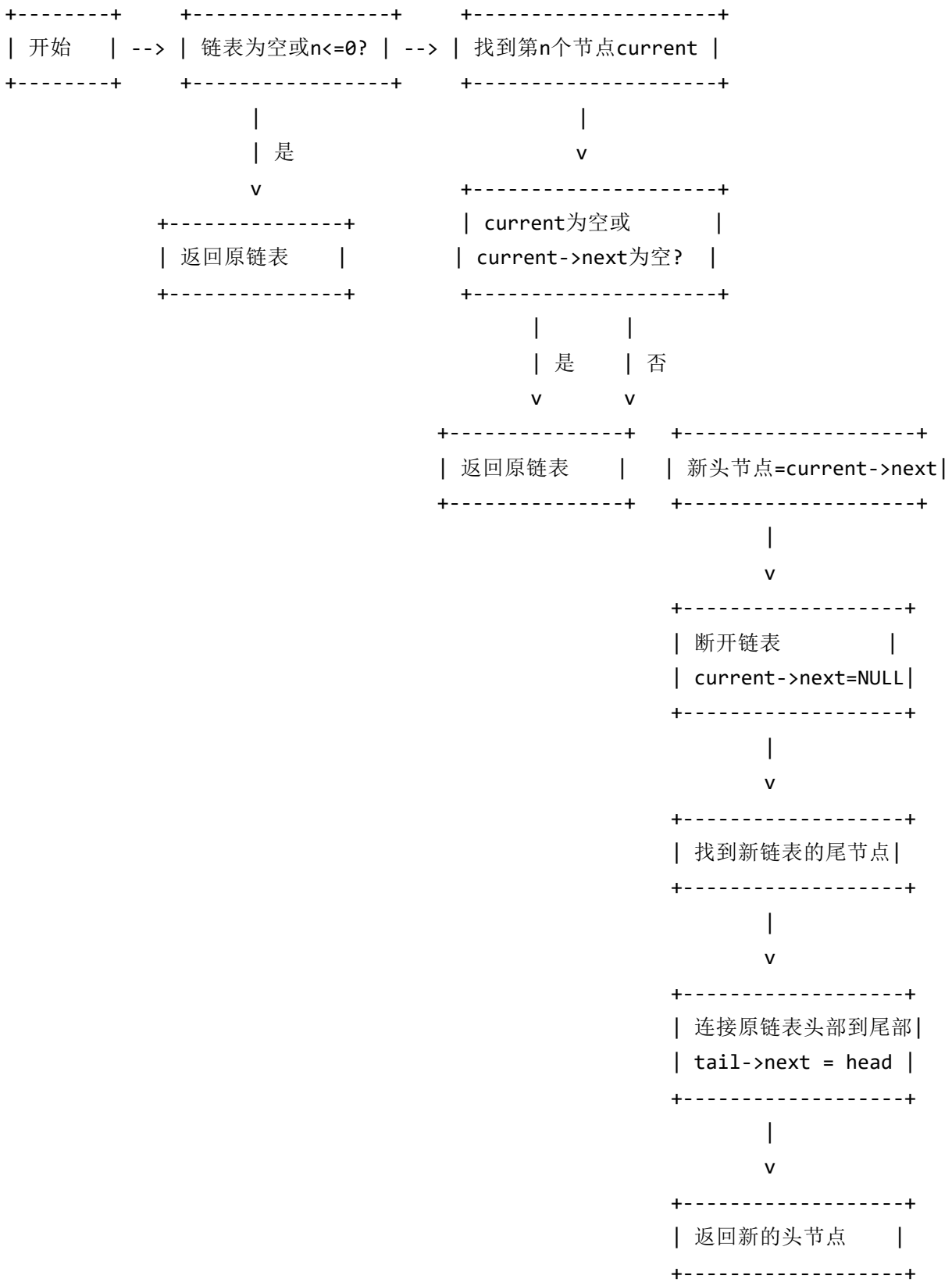
思路分析

本题要求实现字符串的左旋转操作，题目中使用链表来存储字符串。我们需要将链表前面的 n 个节点移动到链表尾部。

解题思路：

1. 如果链表为空或 n <= 0，直接返回原链表。
2. 找到第 n 个节点，将其作为分割点。
3. 将第 n 个节点的 next 置为 NULL，断开链表。
4. 新链表的头节点是第 n+1 个节点。
5. 找到原链表尾部，将原链表的头部连接到尾部。
6. 返回新的链表头节点。

流程图


开始
链表为空或n<=0?
是
返回原链表
找到第n个节点current
current为空或
current->next为空?
是
否
返回原链表
新头节点=current->next
断开链表
current->next=NULL
找到新链表的尾节点
连接原链表头部到尾部
tail->next = head
返回新的头节点

代码分析

```
CharNode *LeftRotate(CharNode *head, int n)
{
    // 处理特殊情况：链表为空或 n <= 0
    if (!head || n <= 0)
    {
        return head;
    }

    // 找到第 n 个节点
    CharNode *current = head;
    for (int i = 1; i < n && current; ++i)
    {
        current = current->next;
    }

    // 如果 n 大于链表长度或 current->next 为空，直接返回原链表
    if (!current || !current->next)
    {
        return head;
    }

    // 新链表的头节点是第 n+1 个节点
    CharNode *newHead = current->next;

    // 断开链表，将第 n 个节点的 next 置为 NULL
    current->next = NULL;

    // 找到原链表的尾部
    CharNode *tail = newHead;
    while (tail->next)
    {
        tail = tail->next;
    }

    // 连接链表：将原链表头部连接到尾部
    tail->next = head;

    return newHead;
}
```

易错提醒

1. **链表为空的处理**：如果链表为空或 n <= 0，应该直接返回原链表。
2. **节点计数**：从 1 开始计数，找到第 n 个节点。注意在循环中要检查 `current` 是否为空。
3. **n 值超出链表长度**：如果 n 大于链表长度，或者第 n 个节点的下一个节点为空，应该直接返回原链表。
4. **断开链表**：在设置新的头节点后，需要将第 n 个节点的 next 指针置为 NULL，断开链表。
5. **找到尾节点**：需要从新的头节点开始遍历，找到链表尾部。
6. **连接链表**：将原链表的头部连接到新链表的尾部，形成新的链表。
7. **内存管理**：注意本题中没有释放节点内存的操作，因为我们只是移动了节点的位置，没有创建新节点。

综合提醒

1. **链表基础**：熟悉链表的基本操作，包括遍历、插入、删除等。
2. **指针操作**：理解指针概念，掌握指针的赋值、移动、比较等操作。
3. **内存管理**：在创建新节点时使用 `new` 分配内存，删除节点时使用 `delete` 释放内存，避免内存泄漏。
4. **边界条件**：处理各种边界情况，如链表为空、只有一个节点、操作涉及头节点等。
5. **临时变量**：使用临时变量保存中间状态，避免指针丢失。
6. **代码可读性**：编写清晰、易读的代码，添加适当的注释说明功能和逻辑。
7. **测试验证**：用各种输入测试代码，验证代码的正确性和鲁棒性。
8. **算法效率**：考虑算法的时间复杂度和空间复杂度，尽量优化算法效率。