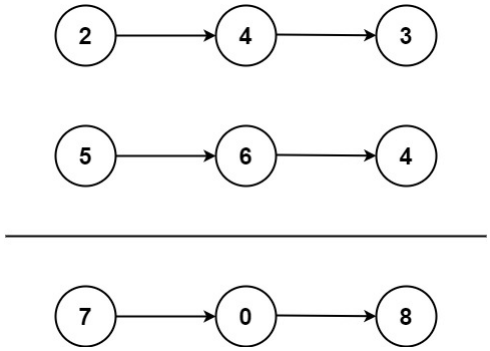
链表

概念 原理

题目

2. 两数相加

> 给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。
>
> 请你将两个数相加，并以相同形式返回一个表示和的链表。
>
> 你可以假设除了数字 0 之外，这两个数都不会以 0 开头。
>
> 示例 1：



输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807.
示例 2：

输入：l1 = [0], l2 = [0]
输出：[0]
示例 3：

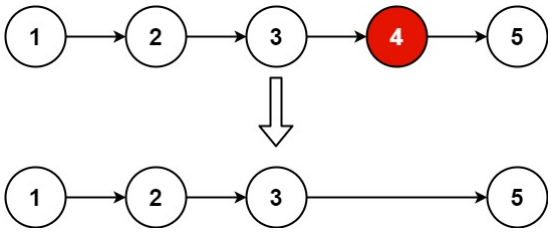
输入：l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
输出：[8,9,9,9,0,0,0,1]

- 模拟
  - 思路
    - 由于输入的两个链表都是 **逆序**存储数字的位数，故两个链表同一位置的数字可直接相加
    - 同时遍历两个链表，逐位计算其和，并与当前位置的进位值相加。具体而言，如果两个链表处相应位置的数字为$n1, n2$，进位值为$carry$，则其和为$n1 + n2 + carry$；其中，结果链表处相应位置的数字为$(n1 + n2 + carry) mod 10$，新进位值为$\lfloor \frac{n1 + n2 + carry}{10} \rfloor$
    - 若两个链表长度不同个，则可认为长度短的链表的后边有若干个0
    - 此外，若链表遍历结束后，有$carry > 0$，还需要在结果链表后附加一个节点，节点的值为$carry$
  - 代码

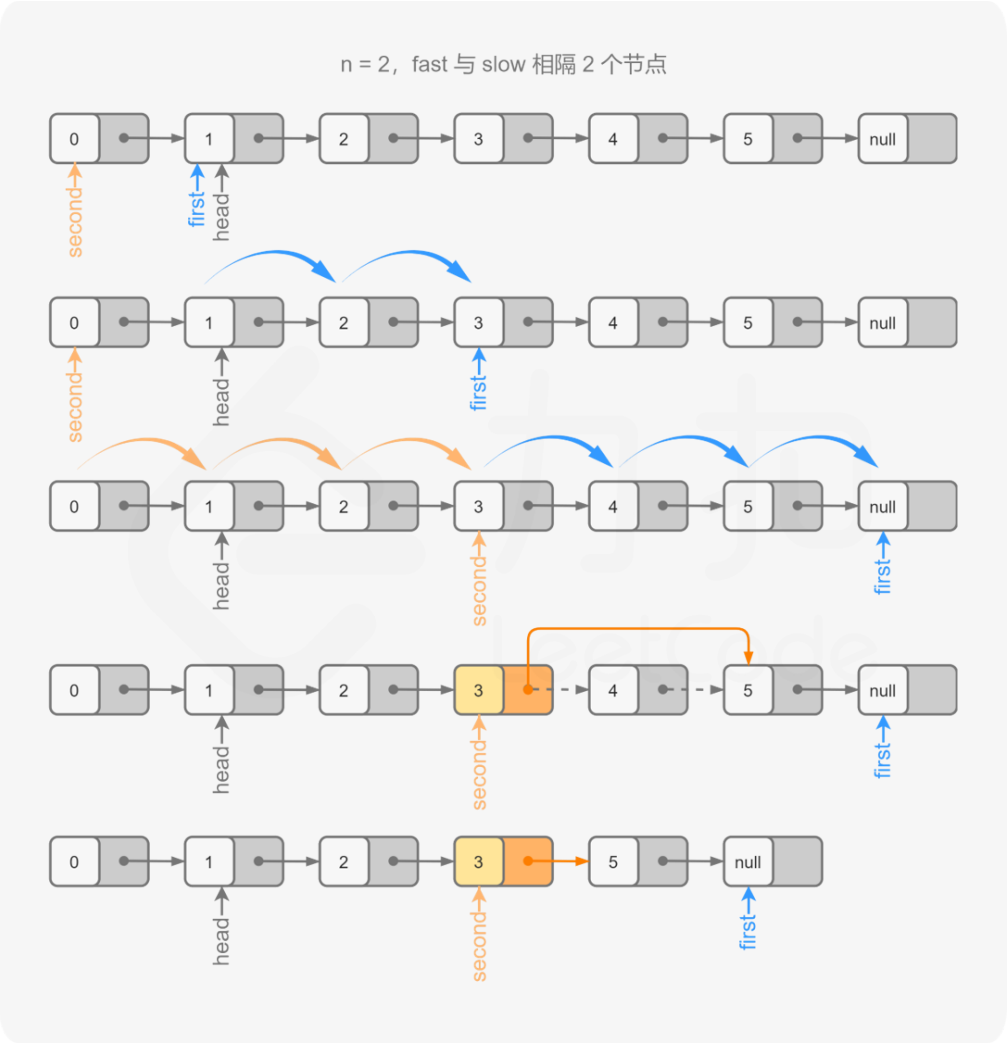
```
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        // 设置一个头结点
        ListNode *head = new ListNode(0), *tmp = head;
        // 进位值
        int carry = 0;
        while (l1 || l2) {
            int val = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + carry;
            tmp->next = new ListNode(val % 10);
            carry = val / 10;
            tmp = tmp->next;
            if (l1) { l1 = l1->next; }
            if (l2) { l2 = l2->next; }
        }

        // 若链表结束后，判断最后进位符，若存在进位，则需要新增节点
        if (carry > 0) { tmp->next = new ListNode(carry); }
        return head->next;
    }
};
```

- 复杂度
  - 时间复杂度：$O(max(m, n))$，其中m和n分别为两个链表的长度。我们要遍历两个链表的全部位置，而处理每个位置只需要$O(1)$的时间
  - 空间复杂度：$O(1)$，注意返回值不计入空间复杂度

19. 删除链表的倒数第 N 个结点

> 给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。
>
> 进阶：你能尝试使用一趟扫描实现吗？
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5], n = 2
> 输出：[1,2,3,5]
> 示例 2：
>
> 输入：head = [1], n = 1
> 输出：[]
> 示例 3：
>
> 输入：head = [1,2], n = 1
> 输出：[1]

- 双指针法
  - 思路：使用快慢指针遍历链表，并且保持快指针超前慢指针n个节点。当快指针遍历到链表末尾时，慢指针恰好处于倒数第n个节点
    - 设置虚拟头结点`dummy`，因为删除节点操作需要保存待删除节点的前一个节点，`dummy`更方便操作
    - 设置快慢指针，`first`指针指向头结点，而`second`指针指向`dummy`
    - 首先移动快指针，移动步数为`n`，此时`first`和`second`相差了`n`个节点
    - 然后，同时移动`first`和`second`指针，当`first`指针指向链表末尾（即`first`为空指针）时，`second`指针恰好指向倒数第`n`个节点的前驱节点
    - 进行删除操作

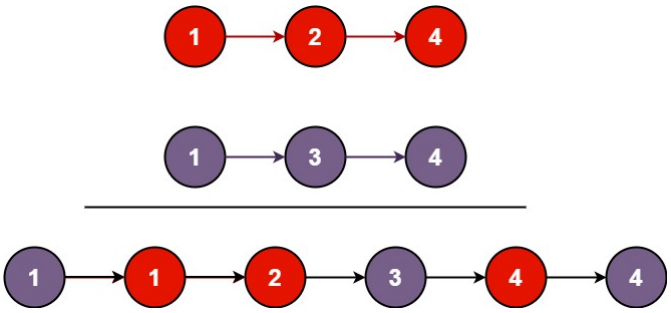


- 代码

```
class Solution {
public:
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        ListNode *dummy = new ListNode(0, head);
        ListNode *first = head, *second = dummy;

        for (int i = 0; i < n; ++i) {
            first = first->next;
        }

        while (first) {
            first = first->next;
            second = second->next;
        }

        second->next = second->next->next;
        ListNode *res = dummy->next;
        delete(dummy);  // 这是个好习惯，清空dummy
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(L)$，其中L是链表长度
  - 空间复杂度：$O(1)$

21. 合并两个有序链表

> 将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。
>
> 示例 1：
>
> 
>
>
> 输入：l1 = [1,2,4], l2 = [1,3,4]
> 输出：[1,1,2,3,4,4]
> 示例 2：
>
> 输入：l1 = [], l2 = []
> 输出：[]
> 示例 3：
>
> 输入：l1 = [], l2 = [0]
> 输出：[0]

- 递归
  - 思路
    - 若`l1`和`l2`是空链表，则不需要合并，直接返回
    - 若两个链表都非空，则判断`l1`和`l2`两链表头结点的值谁更小，递归决定下一个添加到结果里的节点
    - 若两个链表有一个为空，则递归结束
  - 代码

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == nullptr) { return l2; }
        else if (l2 == nullptr) { return l1; }
        else if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n + m)$，其中n和m分别为两个链表的长度。因为每次调用都会去掉$l1$或者$l2$的头结点（直到至少有一个链表为空），函数`mergeTwoLists`至多只会递归调用每个节点一次。因此，时间复杂度取决于合并后的链表长度，即$O(n + m)$

        - 空间复杂度：$O(n + m)$，其中n和m分别为两个链表的长度。递归调用 `mergeTwoLists` 函数时需要消耗栈空间，栈空间大小取决于递归调用深度。结束递归调用时 `mergeTwoLists` 函数最多调用$n + m$次，因此空间复杂度为$O(N + m)$
- 迭代
    - 思路：当 `l1` 和 `l2` 都不为空时，遍历 `l1` 和 `l2`，比较对应元素，将较小节点添加到结果里，并向后移动对应的链表
        - 设置哨兵节点 `dummy`，方便返回合并后的链表，同时维护一个prev指针
        - 遍历 `l1` 和 `l2` 链表，若都不为空，则比较节点值，将较小节点加入到结果中，并移动对应链表指针
        - 若某链表为空后，则退出遍历，并将另一链表直接连接到结果中
    - 代码

```
class Solution {
public:
     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
         ListNode *dummy = new ListNode(0), *prev = dummy;

         while (l1 && l2) {
             if (l1->val <= l2->val) {
                 prev->next = l1;
                 l1 = l1->next;
             } else {
                 prev->next = l2;
                 l2 = l2->next;
             }
             prev = prve->next;
         }

         prev->next = (l1 ? l1 : l2);
         return dummy->next;
     }
};
```

    - 复杂度
        - 时间复杂度：$O(n + m)$，其中n和m分别为两个链表的长度。因为每次迭代循环中，`l1` 和 `l2` 只有一个元素会被放进合并链表中，因此 `while` 循环的次数不会超过两个链表长度之和。所有其他操作的时间复杂度都是常数级别，因此总的时间复杂度为 `O(n + m)`
        - 空间复杂度：$O(1)$，只需要常数空间存放若干变量

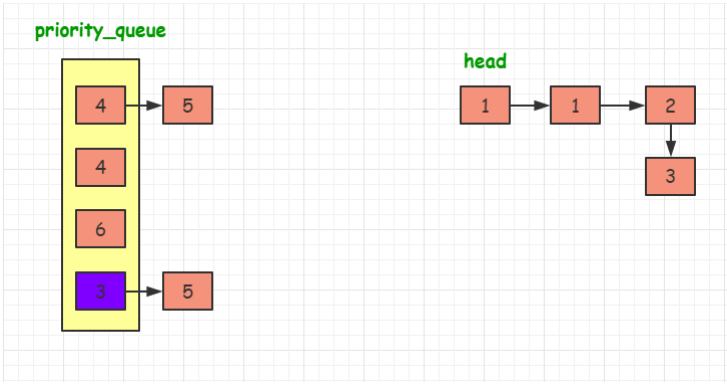
23. 合并K个升序链表

> 给你一个链表数组，每个链表都已经按升序排列。
>
> 请你将所有链表合并到一个升序链表中，返回合并后的链表。
>
> 示例 1：
>
> 输入：lists = [[1,4,5],[1,3,4],[2,6]]
> 输出：[1,1,2,3,4,4,5,6]
> 解释：链表数组如下：
> [
> 1->4->5,
> 1->3->4,

> 2->6
>]
> 将它们合并到一个有序链表中得到。
> 1->1->2->3->4->4->5->6
> 示例 2：
>
> 输入：lists = []
> 输出：[]
> 示例 3：
>
> 输入：lists = [[]]
> 输出：[]

- 参考：https://leetcode.cn/problems/merge-k-sorted-lists/solutions/172800/c-you-xian-dui-lie-liang-liang-he-bing-fen-zhi-he-/
- 优先队列
  - 思路

    维护当前每个链表没有被合并的元素的最前面一个，`k`个链表就最多有`k`个元素，每次在这些元素中选取`val`最小的元素合并到答案中。在选取最小元素的时候，可使用优先队列



  - 代码

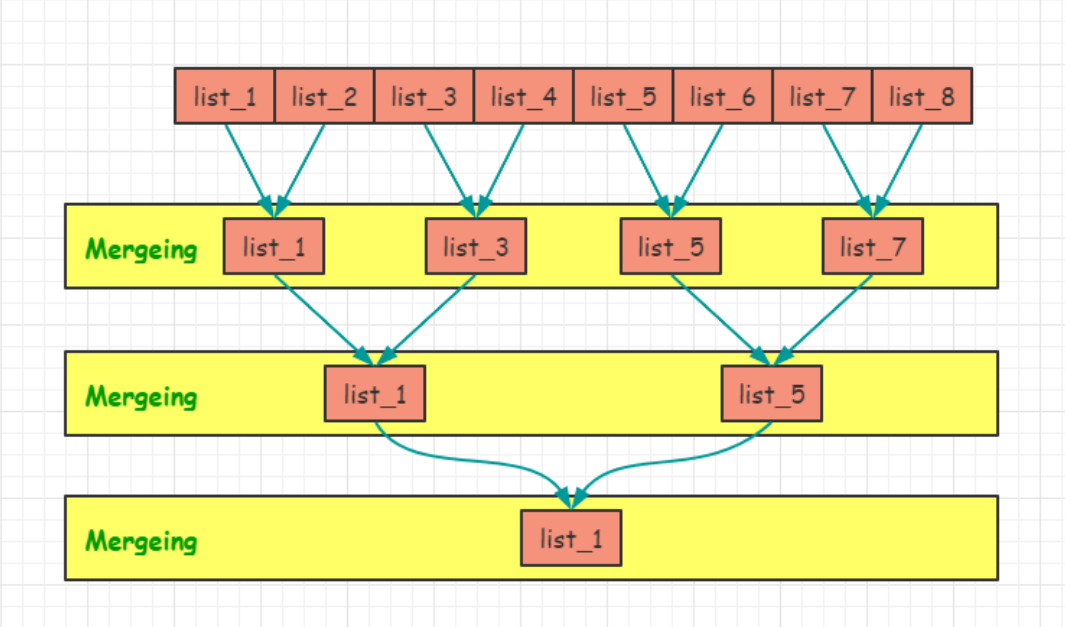
```
// https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/c-you-xian-dui-lie-liang-liang-he-bing-fen-zhi-he-/
 // 优先队列
class Solution {
public:
    struct cmp {
        bool operator() (ListNode *a, ListNode *b) {
            return a->val > b->val;
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, cmp> pq;
        // 建立大小为k的小根堆
        for (auto elem : lists) {
            if (elem != nullptr) {
                pq.push(elem);
            }
        }

        // 新建哑结点/哨兵节点
        ListNode *dummy = new ListNode(0), *cur = dummy;
        while (!pq.empty()) {
            ListNode *tmp = pq.top();
```

```
            pq.pop();
            cur->next = tmp;
            cur = tmp;
            if (tmp->next) {
                pq.push(tmp->next);
            }
        }

        return dummy->next;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n \log(k))$
  - 空间复杂度：$O(k)$

- 分治
  - 思路

    采用21题两两合并的法子，不过这里添加分治的思想，请看下图，这就是分治合并的过程



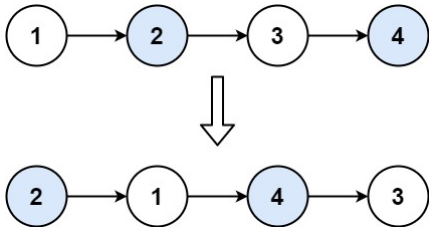
    - k个链表两两配对，进行第一轮合并，结束后k个链表被合并成k/2个链表
    - k/2个链表依然两两配对，进行第二轮合并，结束后k/2个链表被合并成k/4个链表
    - 重复上述过程，进行log(k)次合并，完成总体合并工作
  - 代码

```
class Solution {
public:
    // 分治
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) { return nullptr; }
        return merge(lists, 0, lists.size() - 1);
    }
    ListNode* merge(vector<ListNode*> &lists, int start, int end) {
        if (start == end) { return lists[start]; }

        int mid = (start + end) / 2;
        ListNode *l1 = merge(lists, start, mid);
        ListNode *l2 = merge(lists, mid + 1, end);
        return merge(l1, l2);
```

```
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        if (l1 == nullptr) { return l2; }
        if (l2 == nullptr) { return l1; }
        if (l1->val < l2->val) {
            l1->next = merge(l1->next, l2);
            return l1;
        } else {
            l2->next = merge(l1, l2->next);
            return l2;
        }
    }
};
```

- 复杂度：时间复杂度 O(logk * n)，空间复杂度 O(1)【不考虑递归调用栈】

24. 两两交换链表中的节点

> 给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。
>
> 你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4]
> 输出：[2,1,4,3]
> 示例 2：
>
> 输入：head = []
> 输出：[]
> 示例 3：
>
> 输入：head = [1]
> 输出：[1]

- 递归
  - 思路
    - 终止条件：链表中没有节点或者只有一个节点时，无法进行交换
    - 返回值：返回已经完成交换的新的头结点
    - 本层递归要做的：对于两个节点即`node1->node2`，进行交换，变为`node2->node1`
  - 代码

```
class Solution {
public:
    ListNode *swapPairs(ListNode *head) {
        // 终止条件
        if (head == nullptr || head->next == nullptr) { return head; }

        // 本层要做的
```

```
        ListNode *newHead = head->next;
        head->next = swapPairs(newHead->next);
        newHead->next = head;
        // 返回值
        return newHead;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n是链表节点数量。需要对每个节点进行更新指针的操作
  - 空间复杂度：$O(n)$，其中n是链表节点数量。空间复杂度主要取决于递归调用的栈空间

- 迭代
  - 思路
    - 创建`dummy`节点，表示虚拟头结点，并创建`tmp`指针，作为遍历链表，交换节点的起始指针
    - 交换操作需要`tmp->node1->node2`三个节点才能正确操作

```
tmp->next = node2;
node1->next = node2->next;
node2->next = node1;
```

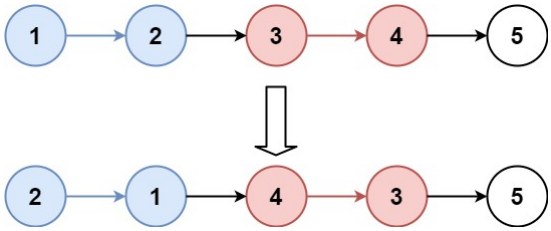
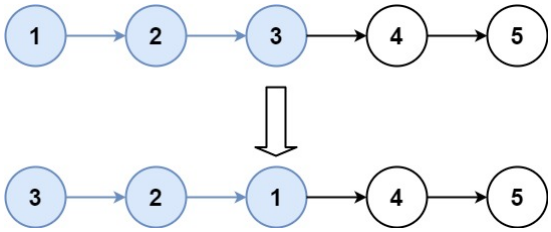
  - 代码

```
class Solution {
public:
    ListNode *swapPairs(ListNode *head) {
        ListNode *dummy = new ListNode(0, head), *tmp = dummy;
        while (tmp->next != nullptr && tmp->next->next != nullptr) {
            ListNode *node1 = tmp->next, *node2 = tmp->next->next;

            tmp->next = node2;
            node1->next = node2->next;
            node2->next = node1;

            tmp = node1;
        }
        return dummy->next;
    }
};
```

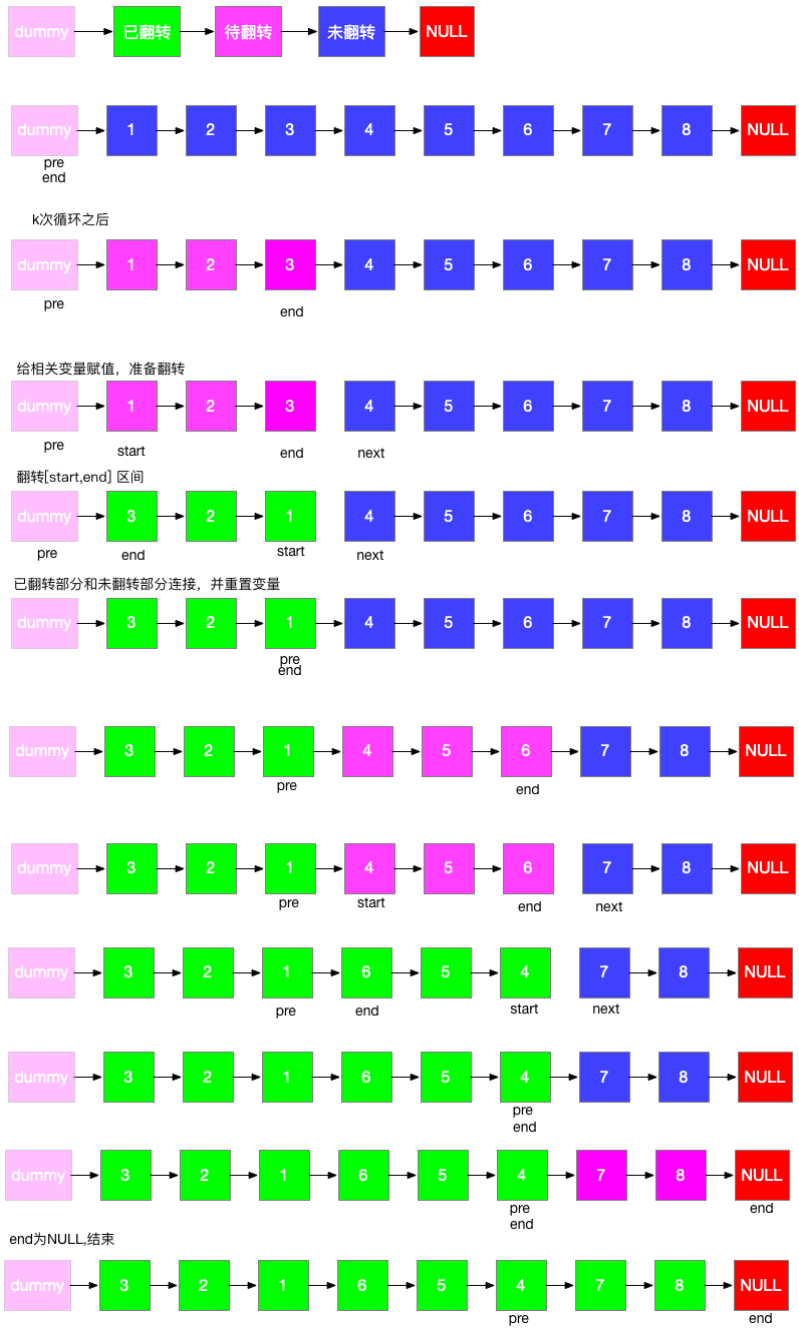
  - 复杂度
    - 时间复杂度：$O(n)$，其中n是链表节点数量。需要对每个节点进行更新指针的操作
    - 空间复杂度有：$O(1)$

25. K 个一组翻转链表

> 给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。
>
> k 是一个正整数，它的值小于或等于链表的长度。
>
> 如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。
>
> 进阶：

> 你可以设计一个只使用常数额外空间的算法来解决此问题吗？
> 你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5], k = 2
> 输出：[2,1,4,3,5]
> 示例 2：
>
> 
>
>
> 输入：head = [1,2,3,4,5], k = 3
> 输出：[3,2,1,4,5]
> 示例 3：
>
> 输入：head = [1,2,3,4,5], k = 1
> 输出：[1,2,3,4,5]
> 示例 4：
>
> 输入：head = [1], k = 1
> 输出：[1]

- 模拟
  - 思路
    - 将链表分为 **已翻转部分 + 待翻转部分 + 未翻转部分**
    - 每次反转前，要确定翻转链表的范围，这个需要通过`k`次循环确定
    - 初始设置几个变量，`start，end`表示翻转链表区间头和尾，`pre，next`表示翻转链表前驱和后继，方便翻转完成后把已翻转部分和未翻转部分连接起来
    - 经过`k`次循环，`end`达到末尾，记录翻转链表后继`next = end->next`
    - 特殊情况，当翻转部分长度不足`k`时，在定位`end`后，`end = null`，已达到末尾，说明题目已完成，直接返回



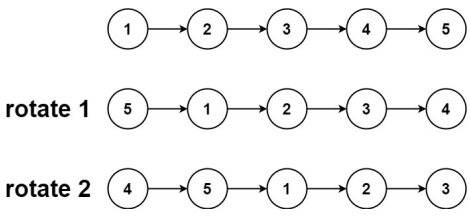
- 代码

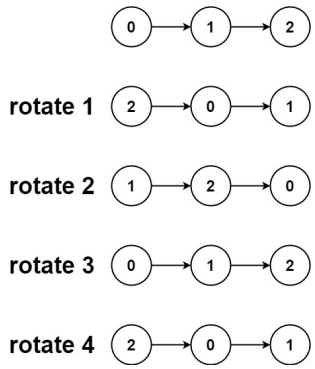
```
 // https://leetcode-cn.com/problems/reverse-nodes-in-k-group/solution/tu-jie-kge-yi-zu-fan-zhuan-lian-biao-by-user7208t/
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode dummy(0, head);
        ListNode *prev = &dummy;
        while (prev != nullptr) {
```

```
            ListNode *start = prev->next, *end = prev;
            int step = k;
            while (step-- && end != nullptr) {
                end = end->next;
            }

            if (end == nullptr) { break; }

            // 找到end，保留next，并切断
            ListNode *next = end->next;
            end->next = nullptr;
            ListNode *new_start = reverse(start);

            prev->next = new_start;
            start->next = next;
            prev = start;
        }

        return dummy.next;
    }

    ListNode *reverse(ListNode *head) {
        ListNode *prev = nullptr;
        while (head != nullptr) {
            ListNode *next = head->next;
            head->next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n \times k)$，其中n是链表节点数量。
  - 空间复杂度有：$O(1)$

61. 旋转链表

> 给你一个链表的头节点 head ，旋转链表，将链表每个节点向右移动 k 个位置。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5], k = 2
> 输出：[4,5,1,2,3]
> 示例 2：



输入：head = [0,1,2], k = 4
输出：[2,0,1]

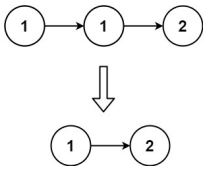
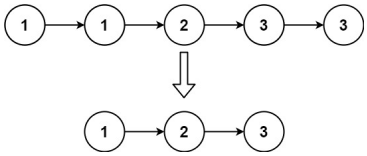
- 闭合为环
  - 思路
    - 记给定链表长度为n，注意当向右移动次数$k \ge n$时，仅需要向右移动$k mod n$次即可。因为每n次移动都会让链表变为原装，故新链表的最后一个节点为原链表的第$n - (k \quad mod \quad n)$个节点（从$n$开始计数）
    - 具体代码中，首先计算出链表长度n，并找到该链表的末尾节点，将其与头结点相连。这样就得到了闭合为环的链表。然后找到新链表的最后一个节点（即原链表的第$n - (k \quad mod \quad n)$个节点），将当前闭合为环的链表断开，即可得到结果
    - 若链表长度不大于1，或者k为n的倍数时，新链表与原链表相同，无需进行任何处理
  - 代码

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        // 链表长度小于等于1 或者 k==0时，不需要移动
        if (k == 0 || head == nullptr || head->next == nullptr) { return
head; }

        // 遍历链表，统计长度，并且iter指向原链表最后一个节点
        ListNode *iter = head;
        int n = 1;
        while (iter->next != nullptr) {
            iter = iter->next;
            ++n;
        }
        // k % n 表示真正需要移动的次数，add表示新链表的最后一个节点
        int add = n - k % n;
        if (add == n) { return head; }

        // 将链表连接为环，并遍历到新链表的最后一个节点
        iter->next = head;
        while (add--) {
            iter = iter->next;
        }
        ListNode *res = iter->next;
        iter->next = nullptr;
        return res;
    }
};
```

  - 复杂度

- 时间复杂度：$O(n)$，最坏情况下，需要遍历该链表两次
- 空间复杂度：$O(1)$，只需要常数空间存储若干变量

83. 删除排序链表中的重复元素

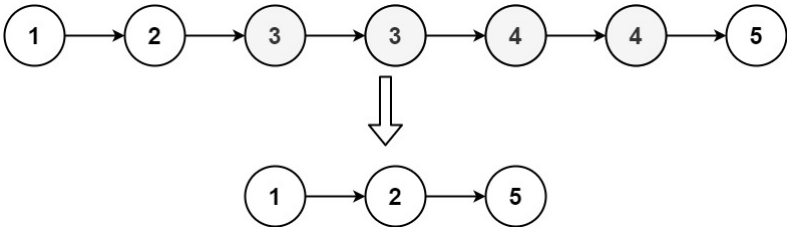
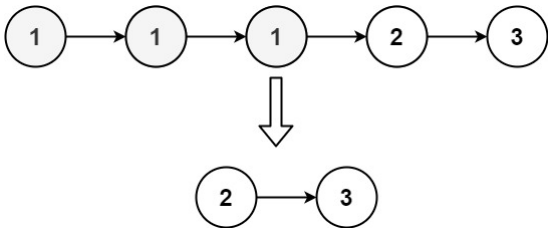
> 存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除所有重复的元素，使每个元素 只出现一次 。
>
> 返回同样按升序排列的结果链表。
>
> 示例 1：
>
> 
>
> 输入：head = [1,1,2]
> 输出：[1,2]
> 示例 2：
>
> 
>
> 输入：head = [1,1,2,3,3]
> 输出：[1,2,3]

- 一次遍历
  - 思路
    - 由于给定链表是排序好的，故重复元素在链表中出现的位置是连续的，故只需要对链表进行一次遍历，就可删除重复元素
    - 具体的，设置一指针`cur`，指向链表头结点，然后遍历链表。若当前`cur`节点与`cur->next`节点对应元素相同，则删除`cur->next`节点；否则说明链表中不存在其他与`cur`节点元素值相同的节点了，继续往下走`cur = cur->next`
    - 遍历完后，返回头结点
    - **注意：代码中没有删除链表节点空间，面试中需要注意沟通**
  - 代码

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == nullptr) { return head; }
        ListNode *cur = head;
        while (cur && cur->next) {
            if (cur->val == cur->next->val) {
                ListNode *tmp = cur->next;
                cur->next = cur->next->next;
                delete(tmp);
            } else {
                cur = cur->next;
            }
        }
        return head;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n是链表节点数量。
  - 空间复杂度：$O(1)$

82. 删除排序链表中的重复元素 II

> 存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中 没有重复出现 的数字。
>
> 返回同样按升序排列的结果链表。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,3,4,4,5]
> 输出：[1,2,5]
> 示例 2：
>
> 
>
>
> 输入：head = [1,1,1,2,3]
> 输出：[2,3]

- 一次遍历
  - 思路
    - 由于给定链表是排序好的，故重复元素在链表中出现的位置是连续的，只需要进行一次遍历，就可以删除重复元素。由于链表头结点可能会被删除，因此我们需要额外使用一个哑结点(dummy node)来指向链表的头结点
    - 具体的，设置`cur`指针，指向链表哑结点，随后对链表进行遍历
    - 如果当前`cur->next`节点元素值与`cur->next->next`节点元素值相等，则需要将`cur->next`以及所有后面拥有相同元素值得链表节点全部删除。我们记下这个元素值x，随后不断将`cur->next`从链表中移除，直到`cur->next`为空节点或者其元素值不等于x为止。至此，已将链表中所有元素值为x的节点全部删除
    - 如果当前`cur->next`节点与`cur->next->next`节点元素值不相等，则说明链表中只有一个元素值为`cur->next`的节点，则可将`cur`指针向后移动
    - 当遍历完整个链表后，返回哑结点的下一个节点`dummy->next`即可
    - 注意：是否需要删除链表节点内存，和面试官沟通好
    - 注意：需要判断`cur->next`以及`cur->next->next`是否可能为空节点，否则可能会出错

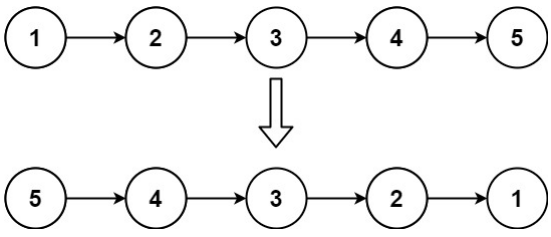
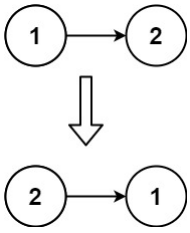
  - 代码

```
// https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list-ii/solution/shan-chu-pai-xu-lian-biao-zhong-de-zhong-oayn/
```

```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == nullptr) { return head; }
        ListNode *dummy = new ListNode(0, head), *cur = dummy;
        while (cur->next != nullptr && cur->next->next != nullptr) {
            // 如果存在两个节点元素值相等
            if (cur->next->val == cur->next->next->val) {
                int x = cur->next->val;
                // 循环判断每个节点元素值是否等于x，相等则删除
                while (cur->next != nullptr && cur->next->val == x) {
                    ListNode *tmp = cur->next;
                    cur->next = cur->next->next;
                    delete(tmp);
                }
            } else {
                cur = cur->next;
            }
        }
        ListNode * res = dummy->next;
        delete(dummy);
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n是链表节点数量。
  - 空间复杂度：$O(1)$

206. 反转链表

> 给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5]
> 输出：[5,4,3,2,1]
> 示例 2：
>
> 
>
>
> 输入：head = [1,2]
> 输出：[2,1]
> 示例 3：

> 输入：head = []
> 输出：[]
>
> **进阶：**链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题？

- 迭代
  - 思路
    - 假设链表 `1 -> 2 -> 3`，需要修改为 `3 -> 2 -> 1`
    - 在遍历链表时，将当前节点的`next`指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前驱节点。在更改引用前，还需存储后继节点。最后返回新的头结点
  - 代码

```
// https://leetcode-cn.com/problems/reverse-linked-list/solution/fan-zhuan-lian-biao-by-leetcode-solution-d1k2/
 // 迭代
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr) {
            return head;
        }

        ListNode *prev = nullptr, *curr = head;
        while (curr != nullptr) {
            ListNode *next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

  - 复杂度
    - 时间复杂度 $O(n)$： 只遍历一次
    - 空间复杂度 $O(1)$
- 递归
  - 思路：递归版本稍复杂，关键在于反向工作。
    - 假设链表的其余部分已经被反转，现在应该如何反转它前面的部分呢？假设链表为

$$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \rightarrow \ldots \rightarrow n_m \rightarrow \varnothing$$

    - 若从根节点$n_{k+1}到n_m$已经被反转，而我们正处于n_k，

$$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \leftarrow \ldots \leftarrow n_m$$

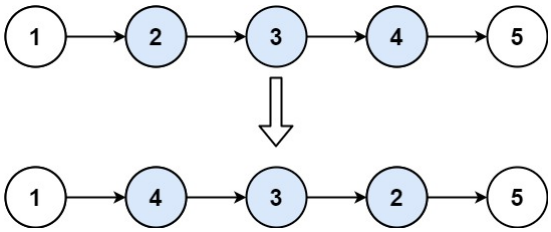
    - 我们希望n_{k+1}的`next`指针指向n_k
    - 所以，$n_k \rightarrow next\rightarrow next = n_k$
    - 需要注意，n_1的下一个节点必须指向$\varnothing$，否则可能会产生环
    - 关于递归思路，假设方法名为`reverse`，可以假设`reverse(node)`的作用就是将`node`及以后的节点组成的链反转，而暂时不关心怎么反转的，只需要使用该方法就可以

- 代码

```
// https://leetcode-cn.com/problems/reverse-linked-list/solution/fan-zhuan-lian-biao-by-leetcode-solution-d1k2/
// 递归方法
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        // newHead只是作为反转后链表的新的头结点，没有其他意义
        ListNode *newHead = reverseList(head->next);
        head->next->next = head;
        head->next = nullptr;
        return newHead;
    }
};
```

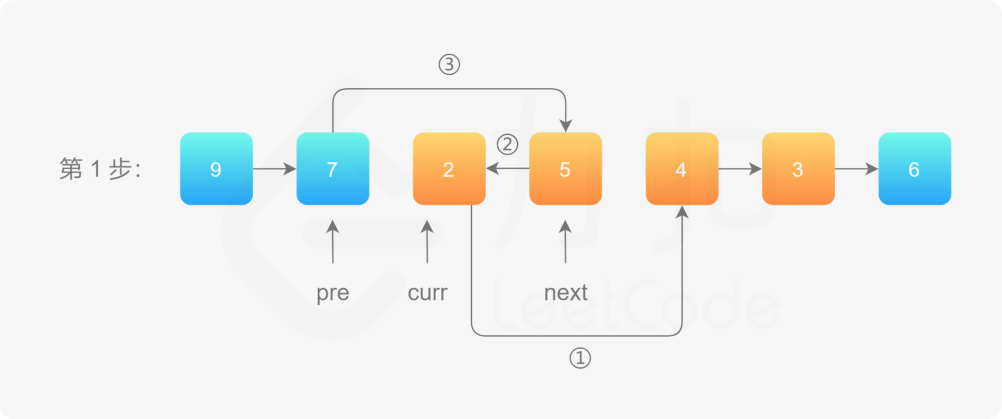
- 复杂度
  - 时间复杂度 $O(n)$： 只遍历一次
  - 空间复杂度 $O(n)$：主要是递归栈空间，最多为`n`层

92. 反转链表 II

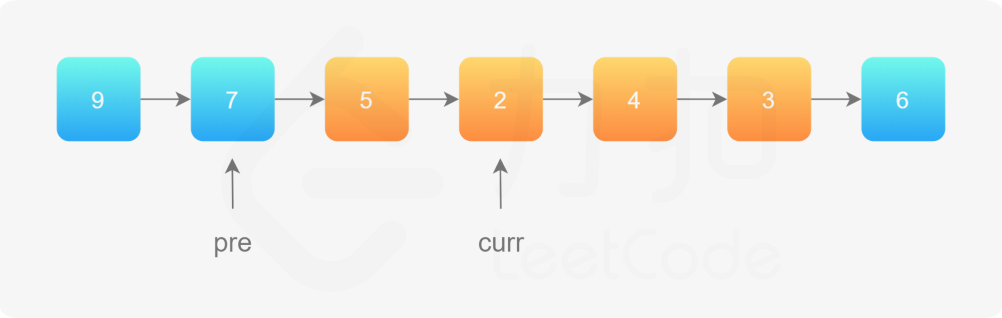
> 给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表 。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5], left = 2, right = 4
> 输出：[1,4,3,2,5]
> 示例 2：
>
> 输入：head = [5], left = 1, right = 1
> 输出：[5]

- 一次遍历：整体思想，在需要反转的区间里，每遍历到一个节点，让这个新节点来到反转部分的起始位置
  - 具体的，使用三个指针变量`pre`， `curr`，`next`来记录反转的过程需要使用的变量
    - `curr`：指向待反转区域的第一个节点`left`
    - `next`：永远指向`curr`的下一个节点，循环过程中，`curr`变化以后`next`会变化
    - `pre`：永远指向待反转区域的第一个节点`left`的前一个节点，在循环过程中不变
  - 第一步
    - 先将`curr`的下一个节点记录为`next`
    - 执行操作`1`：把`curr`的下一个节点指向`next`的下一个节点
    - 执行操作`2`：把`next`的下一个节点`pre`的下一个节点

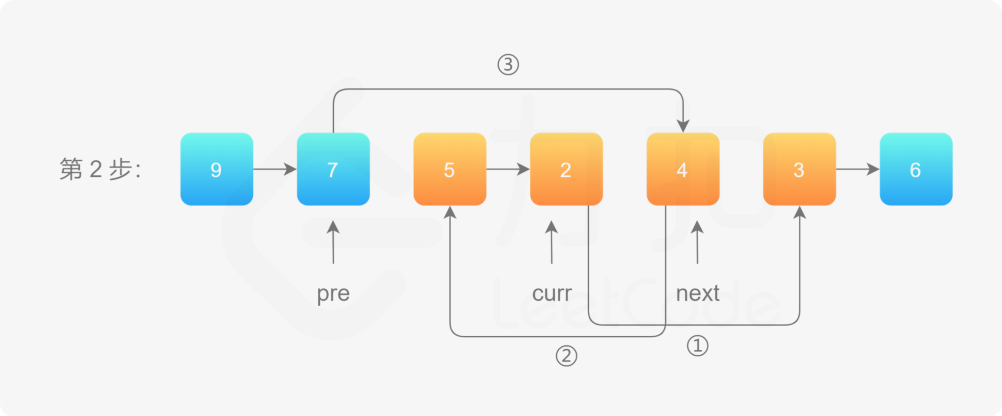
- 执行操作 3：把 pre 的下一个节点指向 next



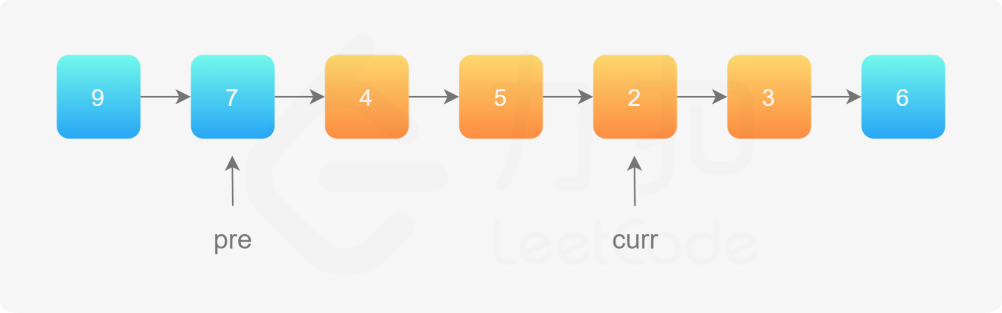
- 第一步完成后，拉直效果如下



- 第二步同理
  - 操作过程如下



  - 拉直效果如下



- 代码

```
// https://leetcode-cn.com/problems/reverse-linked-list-ii/solution/fan-zhuan-lian-biao-ii-by-leetcode-solut-teyq/
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int left, int right) {
        // 需要设置dummy虚拟头结点，防止头结点需要反转
```

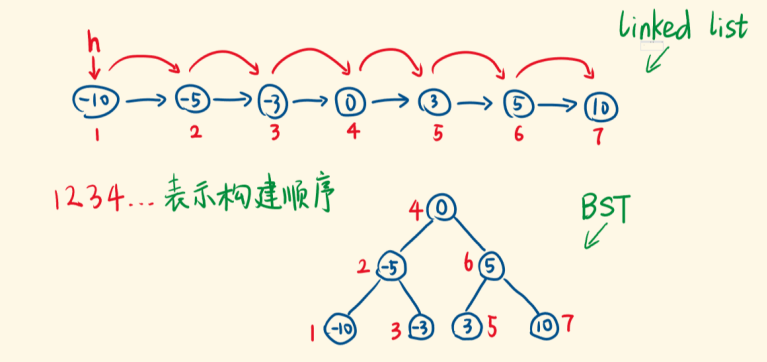
```
        ListNode *dummy = new ListNode(0, head);
        ListNode *pre = dummy;
        // 遍历，找到反转区间范围的前驱节点
        for (int i = 0; i < left - 1; ++i) {
            pre = pre->next;
        }
        ListNode *cur = pre->next;
        ListNode *next;
        for (int i = 0; i < right - left; ++i) {
            next = cur->next;
            cur->next = next->next;
            next->next = pre->next;
            pre->next = next;
        }
        ListNode *res = dummy->next;
        delete(dummy);  // 删除dummy节点
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n为链表长度，最多遍历链表一次，就可完成反转
  - 空间复杂度：$O(1)$，只使用了常数个变量

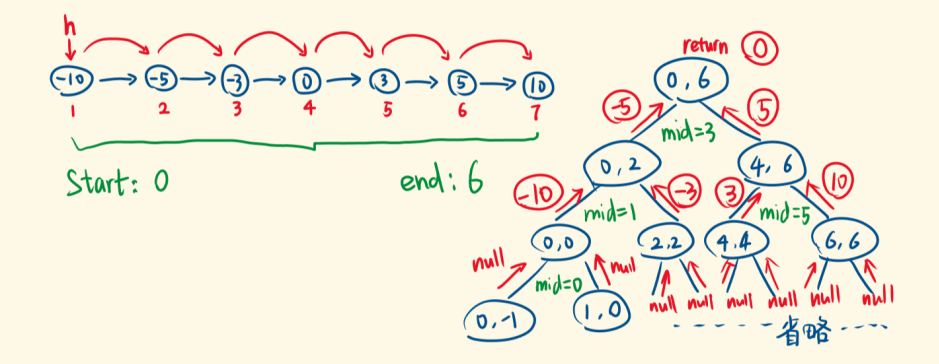
109. 有序链表转换二叉搜索树

> 给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。
>
> 本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。
>
> 示例:
>
> 给定的有序链表： [-10, -3, 0, 5, 9],
>
> 一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树：
>
> ```
>     0
>    / \
>  -3   9
>  /   /
> -10  5
> ```

- 中序遍历+递归
  - 思路：先构建左子树，再构建根节点，再构建右子树，遵循中序遍历
    - 如图，维护指针`h`，从头结点开始，用`h->val`构建节点，构建一个，则链表指针后移一位



- 求出链表节点个数，每次求出链表的中间位置，分治，先用左链递归构建左子树，它会尽可能去划分左右链，直到无法划分就返回NULL；然后创建最左字数的根节点，接上它的两个NULL子节点，然后h指针后移，创建下一个节点



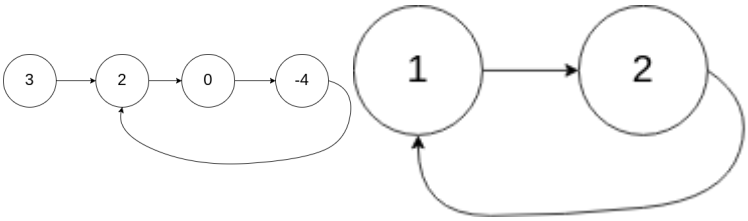
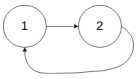
- 代码

```
// 原理参考方法3 https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/solution/shou-hua-tu-jie-san-chong-jie-fa-jie-zhu-shu-zu-ku/
// 代码参考 https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/solution/you-xu-lian-biao-zhuan-huan-er-cha-sou-suo-shu-1-3/550729
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        return helper(head, 0, getLen(head) - 1);
    }

    int getLen(ListNode *head) {
        int n = 0;
        while (head) {
            head = head->next;
            ++n;
        }
        return n;
    }
    // 注意node需要传入引用，helper函数递归调用时，修改node需要在上层递归生效
    TreeNode* helper(ListNode* &head, int left, int right) {
        // 注意二分法边界
        if (left > right) { return nullptr; }
        int mid = left + (right - left) / 2;

        TreeNode* root = new TreeNode();
        // 递归构建左子树
        root->left = helper(head, left, mid - 1);
        // 构建当前根节点
        root->val = head->val;
        head = head->next;
        // 递归构建右子树
        root->right = helper(head, mid + 1, right);

        return root;
    }
};
```

- 复杂度

- 时间复杂度：$O(n)$
- 空间复杂度：$O(\log n)$

141. 环形链表

> 给定一个链表，判断链表中是否有环。
>
> 如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。
>
> 如果链表中存在环，则返回 true 。 否则，返回 false 。
>
> 进阶：
>
> 你能用 O(1)（即，常量）内存解决此问题吗?
>
> 示例 1：
>
> 
>
>
> 输入：head = [3,2,0,-4], pos = 1
> 输出：true
> 解释：链表中有一个环，其尾部连接到第二个节点。
> 示例 2：
>
> 
>
>
> 输入：head = [1,2], pos = 0
> 输出：true
> 解释：链表中有一个环，其尾部连接到第一个节点。
> 示例 3：
>
> 
>
>
> 输入：head = [1], pos = -1
> 输出：false
> 解释：链表中没有环。

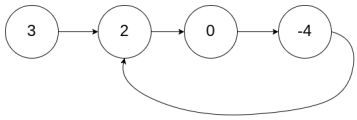
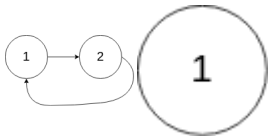
- 快慢指针
  - 思路

    参考 **Floyd判圈算法（龟兔赛跑算法）**，快指针每次移动两个位置，慢指针每次移动一个位置；若存在环，则快指针会追上慢指针
  - 代码

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *fast = head, *slow = head;
        while (fast && fast->next) {
            fast = fast->next->next;
            slow = slow->next;
            if (fast == slow) { return true; }
        }
        return false;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n是链表长度
  - 空间复杂度：$O(1)$，常数空间

142. 环形链表 II

> 给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
>
> 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意，pos 仅仅是用于标识环的情况，并不会作为参数传递到函数中。
>
> 说明：不允许修改给定的链表。
>
> 进阶：
>
> 你是否可以使用 O(1) 空间解决此题?
>
> 示例 1：
>
> 
>
>
> 输入：head = [3,2,0,-4], pos = 1
> 输出：返回索引为 1 的链表节点
> 解释：链表中有一个环，其尾部连接到第二个节点。
> 示例 2：
>
> 
>
>
> 输入：head = [1,2], pos = 0
> 输出：返回索引为 0 的链表节点
> 解释：链表中有一个环，其尾部连接到第一个节点。
> 示例 3：
>
> 
>

> 输入：head = [1], pos = -1
> 输出：返回 null
> 解释：链表中没有环。

- 快慢指针
  - 思路
    - 使用快慢指针`fast, slow`，起始时都指向头结点，快指针每次移动两个位置，慢指针每次移动一个位置，若存在环，则快慢指针终将相遇
    - 寻找入环节点，假设链表长度为 a + b，入环前长度为a（不包含入环节点），环长度为b：
      - f = 2s
      - f = s + nb
      - 由上述两式相减得出 s = nb, f = 2nb，即fast和slow指针分别走了2n、n个环的长度
      - 而从head结点走到入环点需要走 : a + nb， 而slow已经走了nb，那么slow再走a步就是入环点
      - 所以让slow再从head节点走a步
      - => 入环点 = a + nb = s + a
      - => s从交点走a步，f从head走a步，f = s重合时，则找到入环点，即走了a步
    - 故当相遇后，再额外使用一个指针`tmp`，让`tmp`指针和`slow`指针同步移动一个位置，他俩的相遇点就是入环点
  - 代码

```cpp
// 原理参考 https://leetcode.cn/problems/linked-list-cycle-ii/solutions/12616/linked-list-cycle-ii-kuai-man-zhi-zhen-shuang-zhi-/
// 代码自己写的哟
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *fast = head, *slow = head;
        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;

            // 相遇节点
            if (fast == slow) {
                ListNode *tmp = head;
                // tmp和slow同时前进，直到相遇，就是入环节点
                while (tmp != slow) {
                    slow = slow->next;
                    tmp = tmp->next;
                }
                return slow;
            }
        }
        return nullptr;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$，其中n为链表长度。最初判断快慢指针相遇时，慢指针走过的距离不会超过链表总长度，随后寻找入环点时，走过的距离也不会超过链表总长度，故

总的执行时间为$O(n) + O(n) = O(n)$
    - 空间复杂度：$O(1)$，常数空间

143. 重排链表

> 给定一个单链表 L：L0→L1→…→Ln-1→Ln ，
> 将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…
>
> 你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。
>
> 示例 1:
>
> 给定链表 1->2->3->4, 重新排列为 1->4->2->3.
> 示例 2:
>
> 给定链表 1->2->3->4->5, 重新排列为 1->5->2->4->3.

- 寻找链表中点+链表逆序+合并链表
    - 思路
        - 注意结果链表即为将原链表的左半端和反转后的右半端合并后的结果
        - 第一步：找到原链表的中点，可使用 **快慢指针法**
        - 第二步：反转原链表的右半端，可使用 **迭代法**
        - 第三步：将原链表的两端合并，因为两链表的长度相差不超过1，因此可直接合并
    - 代码

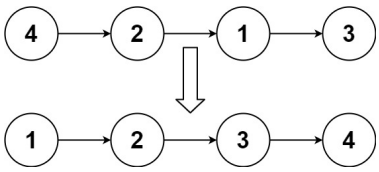
```
// https://leetcode-cn.com/problems/reorder-list/solution/zhong-pai-lian-biao-by-leetcode-solution/
class Solution {
public:
    void reorderList(ListNode* head) {
        ListNode *fast = head, *slow = head;
        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }

        // 构建l1/l2链表
        ListNode *l1 = head, *l2 = slow->next;

        // 分割l1/l2链表
        slow->next = nullptr;

        // 反转l2链表
        l2 = traverse(l2);

        // merge l2/l2 链表
        merge(l1, l2);
    }

    ListNode *traverse(ListNode *head) {
        ListNode *prev = nullptr;
        while (head != nullptr) {
            ListNode *next = head->next;
            head->next = prev;
            prev = head;
            head = next;
        }
```

```
            return prev;
        }

        void merge(ListNode *l1, ListNode *l2) {
            while (l1 && l2) {
                ListNode *l1next = l1->next, *l2next = l2->next;
                l1->next = l2;
                l2->next = l1next;
                l1 = l1next;
                l2 = l2next;
            }
        }
};
```

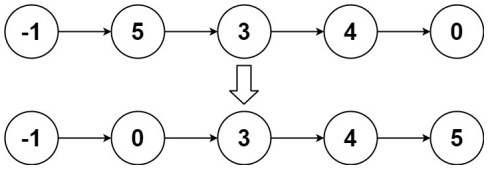
- 复杂度
  - 时间复杂度：$O(n)$，其中n为链表长度。
  - 空间复杂度：$O(1)$，常数空间

148. 排序链表

> 给你链表的头结点 head ， 请将其按 升序 排列并返回 排序后的链表 。
>
> 进阶：
>
> 你可以在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序吗?
>
> 示例 1：



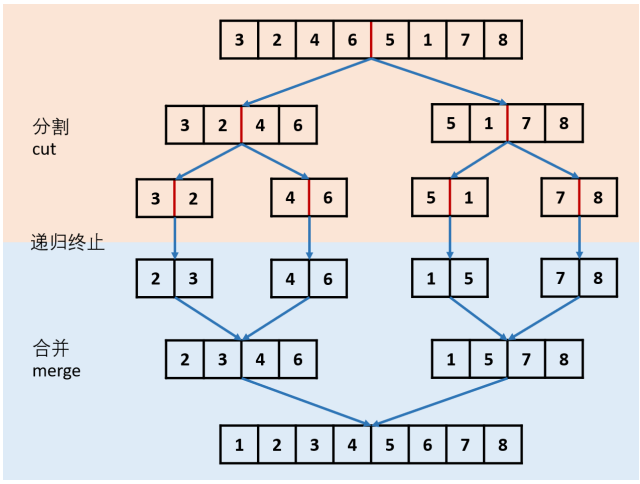
> 输入：head = [4,2,1,3]
> 输出：[1,2,3,4]
> 示例 2：



> 输入：head = [-1,5,3,4,0]
> 输出：[-1,0,3,4,5]
> 示例 3：
>
> 输入：head = []
> 输出：[]

- 前言
  - 题目要求时间复杂度为$O(n \log n)$，空间复杂度为$O(1)$，时间复杂度为$O(n \log n)$的排序算法有 **归并排序、堆排序和快速排序**，其中快排的最差时间复杂度时$O(n^2)$，最适合链表的是归并排序
  - 归并排序基于分治算法，比较容易的实现方式是自顶向下的递归实现，而考虑到递归栈空间，自顶向下归并排序的空间复杂度为$O(n \log n)$。若要达到$O(1)$的空间复杂度，则需要使用自底向上的实现方式
- **自顶向下归并排序**

- 思路
  - **分割cut环节**：找到当前链表 **中点**，并从中点处将链表断开
    - 使用`fast, slow`快慢指针法，奇数个节点找到中点，偶数个节点找到中心左边的节点
    - 找到 **中点** `slow`时，执行`slow->next = nullptr`将链表切割
    - 递归分割时，输入当前链表左端点`head`和中心节点`mid`
    - **cut递归终止条件**：当`head->next == nullptr`时，说明只有一个节点了，直接返回即可
  - **合并merge环节**：将两个排序链表合并，转化为一个排序链表
    - 双指针法合并，建立辅助头结点`dummyNode`
    - 比较两链表值大小，将小的加入到`dummyNode`链中，直至遍历完
    - 返回辅助头结点`dummyNode`的下一个节点



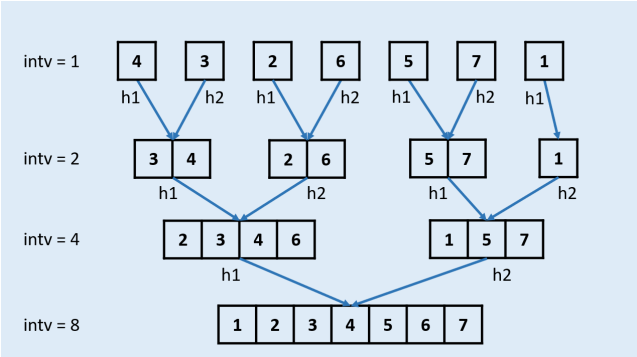
- 代码

```
// 原理参考 https://leetcode-cn.com/problems/sort-list/solution/sort-list-gui-bing-pai-xu-lian-biao-by-jyd/
// 代码参考 https://leetcode-cn.com/problems/sort-list/solution/c-an-bu-jiu-ban-qing-xi-de-gui-bing-pai-0cu85/
// 自顶向下归并排序
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }

        // 找中点，返回值mid是右链头指针，head依旧为左链头指针
        ListNode *mid = middleNode(head);
        // 类似于后序遍历，递归左链
        ListNode *left = sortList(head);
        // 递归右链
        ListNode *right = sortList(mid);
        // 合并当前左、右链
        return mergeTwoLists(left, right);
    }

    // 找中点，记住需要在中点处切割为左右两链
    ListNode *middleNode(ListNode *head) {
        ListNode *prev = nullptr;
        ListNode *fast = head, *slow = head;
```

```
        while (fast != nullptr && fast->next != nullptr) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = nullptr;
        return slow;
    }

    // 合并两链
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0), *tmp = dummy;
        while (l1 && l2) {
            if (l1->val < l2->val) {
                tmp->next = l1;
                l1 = l1->next;
            } else {
                tmp->next = l2;
                l2 = l2->next;
            }
            tmp = tmp->next;
        }
        tmp->next = l1 ? l1 : l2;
        return dummy->next;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n \log n)$，其中n为链表长度。
    - 空间复杂度：$O(\log n)$，其中n为链表长度。空间复杂度主要取决于递归调用的栈空间

- 自底向上归并排序
  - 思路：自底向上归并排序就是使用 **迭代**方式替换 `cut`环节
    - `cut`环节本质就是通过二分法得到链表最小节点单元，再通过多轮合并得到排序结果
    - 每轮合并`merge`操作针对的链都有固定长度
      - 第一轮合并时，步幅为1，即将整个链表切割为多个长度为1的链，并按顺序两两排序合并，合并完成后，已排序单元长度为2
      - 第二轮合并时，步幅为2，即将整个链表切割为多个长度为2的链，并按顺序两两排序合并，合并完成后，已排序单元长度为4
      - 以此类推，直到步幅大于链表长度，代表排序完成
    - 根据以上推论，可以仅根据步幅计算每个单元的边界，并按照链表的每轮排序合并，比如
      - 当步幅为`1`时，将链表第`1`和第`2`个节点排序合并，第`3`和第`4`个节点排序合并，等等等，得到的链是每`2`个节点都是排序好的
      - 当步幅为`2`时，将链表第`1-2`和第`3-4`个节点排序合并，第`5-6`和第`7-8`个节点排序合并，等等等，得到的链是每`4`个节点都是排序好的
      - 当步幅为`4`时，将链表第`1-4`和第`5-8`个节点排序合并，第`9-12`和第`13-16`个节点排序合并，等等等，得到的链是每`8`个节点都是排序好的
      - 以此类推，如下图所示



- 多轮排序合并算法
  - 统计链表长度`length`，用于判断步幅是否大于length，判定是否完成排序
  - 声明辅助头结点`dummyNode`，作为结果链头结点，作用如下
    - 在每轮初始时，即步幅更新时，可通过`dummyNode`找到链表头部
    - 执行排序合并时，需要辅助头结点，
  - 合并流程：略

- 代码

```
// 原理参考 https://leetcode-cn.com/problems/sort-list/solution/sort-list-gui-bing-pai-xu-lian-biao-by-jyd/
// 代码参考 https://leetcode-cn.com/problems/sort-list/solution/pai-xu-lian-biao-by-leetcode-solution/
// 自底向上的归并排序
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr) {
            return head;
        }
        // 获取链表长度
        int length = 0;
        ListNode *node = head;
        while (node != nullptr) {
            node = node->next;
            ++length;
        }

        // 构造dummy头结点
        ListNode *dummyNode = new ListNode(0, head);

        // 归并排序
        for (int idx = 1; idx < length; idx <<= 1) {
            // 获取头结点位置
            ListNode *prev = dummyNode, *curr = dummyNode->next;
            while (curr != nullptr) {
                // 获取左链头指针 l1
                ListNode *l1 = curr;
                for (int i = 1; i < idx && curr->next != nullptr; ++i) {
                    curr = curr->next;
                }

                // 获取右链头指针 l2，当前curr指针左链最后一个节点
                ListNode *l2 = curr->next;

                // 切割左右链
```

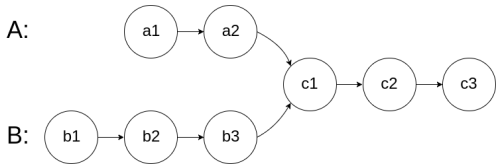
```
                curr->next = nullptr;
                // 重新设置curr指向右链头节点
                curr = l2;

                // 设置curr指向右链最后一个节点
                for (int i = 1; i < idx && curr != nullptr && curr->next != nullptr; ++i) {
                    curr = curr->next;
                }

                // 将右链和整个链表切割，curr指向右链最后一个节点，next指向下一个左右链的头结点
                ListNode *next = nullptr;
                if (curr != nullptr) {
                    next = curr->next;
                    curr->next = nullptr;
                }

                // 合并两链
                ListNode *merged = mergeTwoLists(l1, l2);
                // 将合并后的链头加入到结果链中
                prev->next = merged;

                // 将prev指向结果链的链尾
                while (prev->next != nullptr) {
                    prev = prev->next;
                }

                // 重新设置curr指针，指向下一个左右链的头结点，开启下一次循环
                curr = next;
            }
        }
        return dummyNode->next;
    }

    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode *dummyNode = new ListNode(0), *tmp = dummyNode;
        while (l1 && l2) {
            if (l1->val < l2->val) {
                tmp->next = l1;
                l1 = l1->next;
            } else {
                tmp->next = l2;
                l2 = l2->next;
            }
            tmp = tmp->next;
        }
        tmp->next = l1 ? l1 : l2;
        return dummyNode->next;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n \log n)$，其中n为链表长度。
  - 空间复杂度：$O(1)$，

160. 相交链表

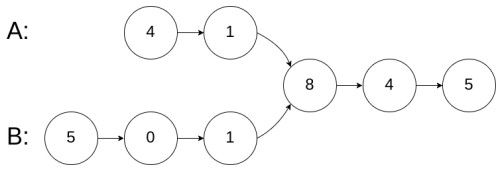
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



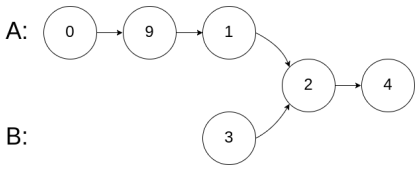
在节点 c1 开始相交。

示例 1：



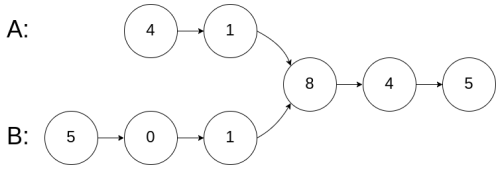
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。

- 双指针法
  - 思路
    - 如果两个链表相交，那么相交点之后的长度是相同的
    - 设链表 `A` 长度为 `a`，链表 `B` 长度为 `b`，相交点之后的长度为 `c`

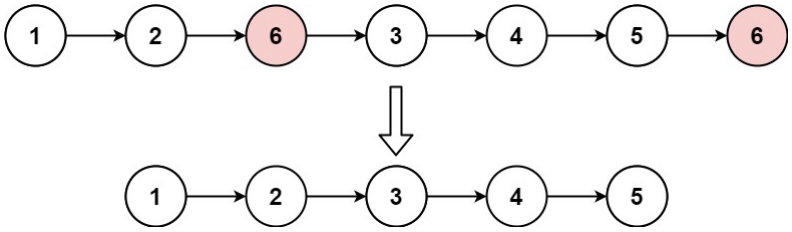
        - 链表`A`头结点到相交点，共有`a - c`个节点
        - 链表`B`头结点到相交点，共有`b - c`个节点
    - 所以，设置指针`headA`指向链表`A`头结点，指针`headB`指向链表`B`头结点，做如下操作
        - `headA`先遍历链表`A`，再开始遍历链表`B`，当再次走到相交点时，`headA`走过的距离为`a + (b - c)`
        - `headB`先遍历链表`B`，再开始遍历链表`A`，当再次走到相交点时，`headB`走过的距离为`b + (a - c)`
    - 此时，指针`headA, headB`重合
        - 若两链表 **有**公共尾部即(`c > 0`)：指针`headA, headB`同时执行 **第一个公共节点相交节点**
        - 若两链表 **无**公共尾部即（`c = 0`）：指针`headA, headB`同时指向`nullptr`
- 代码

```
class Solution {
public:
    // https://leetcode-cn.com/problems/intersection-of-two-linked-lists/solution/intersection-of-two-linked-lists-shuang-zhi-zhen-l/  不会出现在没有交点的情况下无限循环的情况，因为最后pA == pB == nullptr，可看该链接的评论处
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (headA == NULL || headB == NULL) { return NULL; }
        ListNode *pA = headA, *pB = headB;
        while (pA != pB) {
            if (pA == NULL) {
                pA = headB;
            } else {
                pA = pA->next;
            }
            if (pB == NULL) {
                pB = headA;
            } else {
                pB = pB->next;
            }
        }
        return pA;
    }
};
```

- 复杂度
    - 时间复杂度 $O(a + b)$： 最差情况下（即 $|a - b| = 1, c = 0$ ），此时需遍历$a + b$个节点。
    - 空间复杂度 $O(1)$： 节点指针 A , B 使用常数大小的额外空间。

203. 移除链表元素

> 给你一个链表的头节点 head 和一个整数 val ， 请你删除链表中所有满足 Node.val == val 的节点，并返回 新的头节点 。
>
> 示例 1：

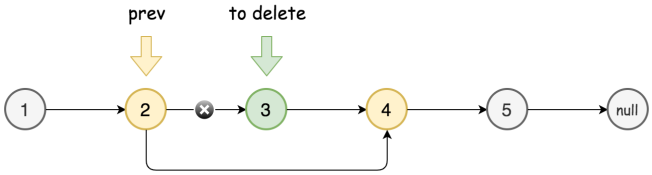


输入：head = [1,2,6,3,4,5,6], val = 6
输出：[1,2,3,4,5]
示例 2：

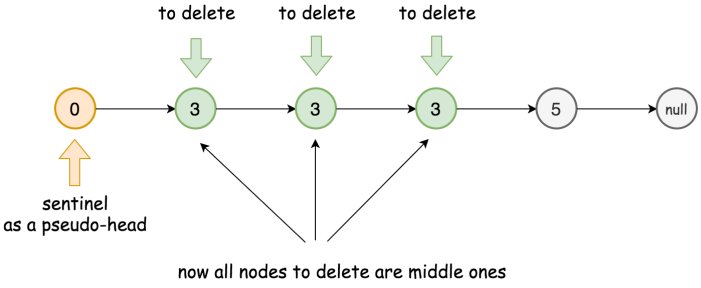
输入：head = [], val = 1
输出：[]
示例 3：

输入：head = [7,7,7,7], val = 7
输出：[]

- 哨兵节点
  - 思路
    - 如果要删除的节点是**中间节点**，则比较简单
      - 选择要删除节点的前一个节点`prev`
      - 将`prev`的`next`设置为要删除节点的`next`



    - 如果要删除的节点是**头结点**，则比较复杂。可使用哨兵节点解决，用来简化插入、删除操作，这里使用哨兵节点用于构造虚拟头结点



  - 算法
    - 初始化哨兵节点为`ListNode(0, head)`
    - 初始化两个指针`curr`和`prev`，指向当前节点和前驱节点
    - 当`curr != nullptr`时
      - 比较当前节点和要删除的节点
        - 如果当前节点就是要删除的节点，则`prev->next = curr->next`
        - 否则设`prev = curr`
      - 遍历下一个元素`curr = curr->next`
    - 返回`dummyNode->next`
  - 代码

```cpp
// https://leetcode-cn.com/problems/remove-linked-list-elements/solution/203yi-chu-lian-biao-yuan-su-by-lewis-dxstabdzew/
// 哨兵节点
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode dummy(0, head);
        ListNode *cur = &dummy;
        while (cur != nullptr && cur->next != nullptr) {
            if (cur->next->val == val) {
                ListNode *tmp = cur->next;
                cur->next = cur->next->next;
                delete(tmp);
            } else {
                cur = cur->next;
            }
        }
        return dummy.next;
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$： 只遍历一次
  - 空间复杂度 $O(1)$

234. 回文链表

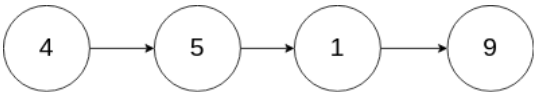
> 请判断一个链表是否为回文链表。
>
> 示例 1:
>
> 输入: 1->2
> 输出: false
> 示例 2:
>
> 输入: 1->2->2->1
> 输出: true
> 进阶:
> 你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题?

- 快慢指针
  - 思路：$O(1)$空间复杂度就是改变输入
    - 可以将链表的后半部分反转（修改链表结构），然后将前半部分和后半部分进行比较。比较完成后将链表恢复成原样。
    - 该方法的空间复杂度为$O(1)$，但在并发环境下，因为函数运行需要修改链表，故需对链表加锁，使得其他线程或进程不可访问该链表
  - 算法
    - 找到链表中点，切割链表：快慢指针法
    - 反转后半部分链表：迭代、递归都可
    - 判断是否回文
    - 恢复链表
    - 返回结果
  - 代码

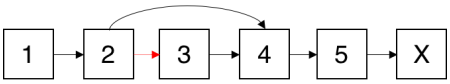
```cpp
// https://leetcode-cn.com/problems/palindrome-linked-list/solution/hui-wen-lian-biao-by-leetcode-solution/
 // 快慢指针法
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return true;
        }

        // 找中点
        ListNode *prev = nullptr, *fast = head, *slow = head;
        while (fast != nullptr && fast->next != nullptr) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }

        // 切割链表
        prev->next = nullptr;

        // 反转后半部分链表
        ListNode* rightHead = reverseList(slow);

        // 比较回文链表
        while (head && rightHead) {
            if (head->val != rightHead->val) {
                return false;
            }
            head = head->next;
            rightHead = rightHead->next;
        }

        // 还原链表
        prev->next = reverseList(rightHead);

        return true;
    }

    ListNode* reverseList(ListNode *head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }

        ListNode *newHead = reverseList(head->next);
        head->next->next = head;
        head->next = nullptr;
        return newHead;
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$： 只遍历一次
  - 空间复杂度 $O(1)$

237. 删除链表中的节点

> 请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为 要被删除的节点 。
>
> 现有一个链表 -- head = [4,5,1,9]，它可以表示为:
>
> 
>
>
> 示例 1：
>
> 输入：head = [4,5,1,9], node = 5
> 输出：[4,1,9]
> 解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
> 示例 2：
>
> 输入：head = [4,5,1,9], node = 1
> 输出：[4,5,9]
> 解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.

- 与下个节点交换值
  - 思路
    - 从链表里删除一个节点 `node` 的最常见方法就是修改 **前驱节点**的 `next` 指针

    - 但是，本题无法获取到待删除节点的 **前驱节点**。故需要其他办法
    - 可以将待删除节点的值和其 **后继节点**的值替换，删除其 **后继节点**，并且题意说明待删除节点不是链表末尾，故该方法可行
  - 代码

```
// https://leetcode-cn.com/problems/delete-node-in-a-linked-list/solution/shan-chu-lian-biao-zhong-de-jie-dian-by-leetcode/
 // 与下一个节点交换
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
    }
};
```

  - 复杂度
    - 时间复杂度 $O(1)$
    - 空间复杂度 $O(1)$

328. 奇偶链表

> 给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。
>
> 请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。
>
> 示例 1:
>
> 输入: 1->2->3->4->5->NULL
> 输出: 1->3->5->2->4->NULL
> 示例 2:
>
> 输入: 2->1->3->5->6->4->7->NULL
> 输出: 2->3->6->7->1->5->4->NULL
> 说明:
>
> 应当保持奇数节点和偶数节点的相对顺序。
> 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

- 双指针-拆分奇链偶链
  - 思路
    - `odd`指针扫描奇数节点，`even`指针扫描偶数节点
      - 奇数节点逐个修改`next`，形成奇链
      - 偶数节点逐个修改`next`，形成偶链
    - 遍历结束后，奇链偶链分开了，此时`odd`指向奇链尾部，将`odd->next = evenHead`，即连上偶链头节点即可
  - 代码

```cpp
// https://leetcode-cn.com/problems/odd-even-linked-list/solution/shou-hua-tu-jie-328qi-ou-lian-biao-odd-even-linked/
// 双指针拆分奇链偶链
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        // 保存偶链头结点
        ListNode *evenHead = head->next;
        ListNode *odd = head, *even = evenHead;

        // 遍历 拆分奇链和偶链
        while (even != nullptr && even->next != nullptr) {
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }

        // 连接奇链和偶链
        odd->next = evenHead;
        return head;
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(1)$

445. 两数相加 II

> 给你两个 非空 链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。
>
> 你可以假设除了数字 0 之外，这两个数字都不会以零开头。
>
> 进阶：
>
> 如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。
>
> 示例：
>
> 输入：(7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
> 输出：7 -> 8 -> 0 -> 7

- 反转链表
  - 反转链表 l1、反转链表 l2
  - l1 l2相加，得到链表 l3
  - 反转链表 l3， 返回 l3，作为答案

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        if (l1 == nullptr || l2 == nullptr) { return l1 ? l1 : l2; }
        ListNode *r1 = reverseList(l1);
        ListNode *r2 = reverseList(l2);

        ListNode *head = addTwo(r1, r2);

        head = reverseList(head);
        return head;
    }
    ListNode *addTwo(ListNode* l1, ListNode *l2) {
        int add = 0;
        ListNode dummy;
        ListNode* cur = &dummy;
        while (l1 ||l2 || add) {
            if (l1 != nullptr) { add += l1->val; l1 = l1->next; }
            if (l2 != nullptr) { add += l2->val; l2 = l2->next; }
            cur->next = new ListNode(add % 10);
            cur = cur->next;
            add /= 10;
        }
        return dummy.next;
    }
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr, *cur = head;
        while (cur != nullptr) {
            ListNode* next = cur->next;
            cur->next = prev;
            prev = cur;
```

```
            cur = next;
        }
        return prev;
    }
};
```

- 栈
  - 思路

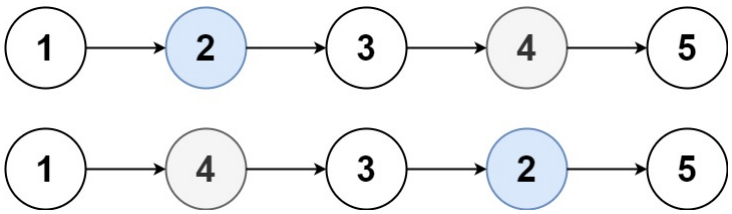
    本题的难点在于链表中数位的顺序与正常加法顺序相反，为了逆序处理所有数位，可使用**栈**：把所有数字压入栈中，再依次取出相加。计算过程中需要注意进位的情况
  - 代码

```
// https://leetcode-cn.com/problems/add-two-numbers-ii/solution/liang-shu-xiang-jia-ii-by-leetcode-solution/
// 栈
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        stack<int> s1, s2;
        // 将数据压入栈中
        while (l1 != nullptr) {
            s1.push(l1->val);
            l1 = l1->next;
        }

        while (l2 != nullptr) {
            s2.push(l2->val);
            l2 = l2->next;
        }

        // 遍历-逆序加法
        ListNode *res = nullptr;
        int carry = 0;
        while (!s1.empty() || !s2.empty() || carry != 0) {
            // 从栈中取值
            int v1 = 0, v2 = 0;
            if (!s1.empty()) {
                v1 = s1.top();
                s1.pop();
            }
            if (!s2.empty()) {
                v2 = s2.top();
                s2.pop();
            }

            // 计算进位值、值
            int v = v1 + v2 + carry;
            carry = v / 10;
            v %= 10;

            // 新建节点，注意要最后是逆序，故res每次都指向新节点
            ListNode *node = new ListNode(v);
            node->next = res;
            res = node;
```

```
        }
        return res;
    }
};
```

- 复杂度
  - 时间复杂度 $O(max(m, n))$，其中`m，n`分别为两个链表的长度。需要遍历两个链表的全部位置，而处理每个位置只需要$O(1)$时间
  - 空间复杂度 $O(m+n)$，其中`m,n`分别为两个链表的长度。空间复杂度主要取决于栈空间

1721. 交换链表中的节点

> 给你链表的头节点 head 和一个整数 k 。
>
> 交换 链表正数第 k 个节点和倒数第 k 个节点的值后，返回链表的头节点（链表 从 1 开始索引）。
>
> 示例 1：
>
> 
>
>
> 输入：head = [1,2,3,4,5], k = 2
> 输出：[1,4,3,2,5]
> 示例 2：
>
> 输入：head = [7,9,6,6,7,8,3,0,9,5], k = 5
> 输出：[7,9,6,6,8,7,3,0,9,5]
> 示例 3：
>
> 输入：head = [1], k = 1
> 输出：[1]
> 示例 4：
>
> 输入：head = [1,2], k = 1
> 输出：[2,1]
> 示例 5：
>
> 输入：head = [1,2,3], k = 2
> 输出：[1,2,3]

- 快慢指针
  - 思路
    - 简单遍历，找到第`k`个节点
    - 快慢指针，找到倒数第`k`个节点
  - 代码

```
// https://leetcode-cn.com/problems/swapping-nodes-in-a-linked-list/solution/1721-jiao-huan-lian-biao-zhong-de-jie-di-t85x/
 // 快慢指针
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
```

```
        // 遍历，找到第k个节点
        ListNode *fast = head;
        for (int i = 1; i < k; ++i) {
            fast = fast->next;
        }
        ListNode *kth = fast;

        // 快慢指针，找到倒数第k个节点
        ListNode *slow = head;
        while (fast->next != nullptr) {
            fast = fast->next;
            slow = slow->next;
        }
        ListNode *reverseKth = slow;

        // 交换
        swap(kth->val, reverseKth->val);

        return head;
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(1)$

725. 分隔链表

> 给定一个头结点为 root 的链表, 编写一个函数以将链表分隔为 k 个连续的部分。
>
> 每部分的长度应该尽可能的相等: 任意两部分的长度差距不能超过 1，也就是说可能有些部分为 null。
>
> 这k个部分应该按照在链表中出现的顺序进行输出，并且排在前面的部分的长度应该大于或等于后面的长度。
>
> 返回一个符合上述规则的链表的列表。
>
> 举例： 1->2->3->4, k = 5 // 5 结果 [[1], [2], [3], [4], null]
>
> 示例 1：
>
> 输入:
> root = [1, 2, 3], k = 5
> 输出: [[1],[2],[3],[],[]]
> 解释:
> 输入输出各部分都应该是链表，而不是数组。
> 例如, 输入的结点 root 的 val= 1, root.next.val = 2, \root.next.next.val = 3, 且 root.next.next.next = null。
> 第一个输出 output[0] 是 output[0].val = 1, output[0].next = null。
> 最后一个元素 output[4] 为 null, 它代表了最后一个部分为空链表。
> 示例 2：
>
> 输入:
> root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3
> 输出: [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]
> 解释:

> 输入被分成了几个连续的部分，并且每部分的长度相差不超过1.前面部分的长度大于等于后面部分的长度。

- 暴力
  - 思路
    - 获取链表长度
    - 计算每段子链表的平均长度和余数
    - 遍历子链表，每个子链表的长度为 **平均长度+1**，直到余数为0
    - 确定子链表长度后，开始分割链表
  - 代码

```
// https://leetcode-cn.com/problems/split-linked-list-in-parts/solution/fei-zi-jie-ti-ku-725-zhong-deng-fen-ge-lian-biao-1/
 // 暴力
class Solution {
public:
    vector<ListNode*> splitListToParts(ListNode* root, int k) {
        vector<ListNode*> res(k, nullptr);
        if (root == nullptr) {
            return res;
        }

        // 获取链表长度
        int length = 0;
        ListNode *tmp = root;
        while (tmp != nullptr) {
            ++length;
            tmp = tmp->next;
        }

        // 计算每个子链表的平均长度和余数
        int avgLength = length / k;
        int reminder = length % k;

        // 遍历链表，拆分成子链表
        ListNode *head = root;
        ListNode *curr = nullptr;
        for (int i = 0; i < k; ++i) { // 定位到res中个每个子链表
            res[i] = head;
            // 将余数均匀分给子链表，直到余数为0
            int currLength = reminder > 0 ? avgLength + 1 : avgLength;
            --reminder;
            // 拆分
            for (int j = 0; j < currLength; ++j) {
                curr = head;
                head = head->next;
            }
            curr->next = nullptr;
        }

        return res;
    }
};
```

  - 复杂度

- 时间复杂度 $O(n + k)$，`n`是链表节点数，若`k`很大，则还需要添加许多空列表
- 空间复杂度 $O(max(n, k))$，存放答案所需空间

876. 链表的中间结点

> 给定一个头结点为 head 的非空单链表，返回链表的中间结点。
>
> 如果有两个中间结点，则返回第二个中间结点。
>
> 示例 1：
>
> 输入：[1,2,3,4,5]
> 输出：此列表中的结点 3 (序列化形式：[3,4,5])
> 返回的结点值为 3 。 (测评系统对该结点序列化表述是 [3,4,5])。
> 注意，我们返回了一个 ListNode 类型的对象 ans，这样：
> ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及 ans.next.next.next = NULL.
> 示例 2：
>
> 输入：[1,2,3,4,5,6]
> 输出：此列表中的结点 4 (序列化形式：[4,5,6])
> 由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

- 快慢指针法
  - 思路
    - 使用快慢指针，`fast, slow`， `fast`指针每次走两步，`slow`指针每次走一步，当`fast`达到尾部时，`slow`则处于中间位置。注意，**偶数节点时，slow指向中间偏右的节点**
  - 代码

```cpp
// https://leetcode-cn.com/problems/middle-of-the-linked-list/solution/lian-biao-de-zhong-jian-jie-dian-by-leetcode-solut/
// 快慢指针法
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode *fast = head, *slow = head;
        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        return slow;
    }
};
```

  - 复杂度
    - 时间复杂度 $O(n)$
    - 空间复杂度 $O(1)$

1019. 链表中的下一个更大节点

> 给出一个以头节点 head 作为第一个节点的链表。链表中的节点分别编号为：node_1, node_2, node_3, ... 。

> 每个节点都可能有下一个更大值（next larger value）：对于 node_i，如果其 next_larger(node_i) 是 node_j.val，那么就有 j > i 且 node_j.val > node_i.val，而 j 是可能的选项中最小的那个。如果不存在这样的 j，那么下一个更大值为 0 。
>
> 返回整数答案数组 answer，其中 answer[i] = next_larger(node_{i+1}) 。
>
> 注意：在下面的示例中，诸如 [2,1,5] 这样的输入（不是输出）是链表的序列化表示，其头节点的值为 2，第二个节点值为 1，第三个节点值为 5 。
>
> 示例 1：
>
> 输入：[2,1,5]
> 输出：[5,5,0]
> 示例 2：
>
> 输入：[2,7,4,3,5]
> 输出：[7,0,5,5,0]
> 示例 3：
>
> 输入：[1,7,5,1,9,2,5,1]
> 输出：[7,9,9,9,0,5,0,0]

- 单调栈
  - 思路
    - 遍历链表，将链表元素转移到`nums`数组中，并记录节点长度
    - 新建`res`数组作为结果返回，元素初始化为0
    - 创建栈，栈中存放**元素值递增的下标**，即从栈底到栈顶每个元素对应的元素值时递增的
    - 遍历`nums`数组，**若栈不为空，且栈顶位置的元素小于当前遍历的元素**，说明栈顶位置的元素找到了它的下一更大节点值，赋值到`res`数组中
  - 代码

```
 //  https://leetcode-cn.com/problems/next-greater-node-in-linked-list/solution/lian-biao-zhong-de-xia-yi-ge-geng-da-jie-f2tu/
 // 单调栈
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        stack<int> s;
        vector<int> nums;
        ListNode *tmp = head;
        int length = 0;
        while (tmp != nullptr) {
            ++length;
            nums.push_back(tmp->val);
            tmp = tmp->next;
        }

        vector<int> res(length, 0);
        for (int i = 0; i < length; ++i) {
            while (!s.empty() && nums[s.top()] < nums[i]) {
                res[s.top()] = nums[i];
                s.pop();
            }
            s.push(i);
        }

        return res;
```

```
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(n)$

1171. 从链表中删去总和值为零的连续节点

> 给你一个链表的头节点 head，请你编写代码，反复删去链表中由 总和 值为 0 的连续节点组成的序列，直到不存在这样的序列为止。
>
> 删除完毕后，请你返回最终结果链表的头节点。
>
> 你可以返回任何满足题目要求的答案。
>
> （注意，下面示例中的所有序列，都是对 ListNode 对象序列化的表示。）
>
> 示例 1：
>
> 输入：head = [1,2,-3,3,1]
> 输出：[3,1]
> 提示：答案 [1,2,1] 也是正确的。
> 示例 2：
>
> 输入：head = [1,2,3,-3,4]
> 输出：[1,2,4]
> 示例 3：
>
> 输入：head = [1,2,3,-3,-2]
> 输出：[1]

- 前缀和
  - 思路

    前缀和，`preSum[i:j] = preSum[j] - preSum[i]`。利用哈希表，两次遍历。第一次遍历保存<sum, node>关系对，相同sum时，只会保留最后出现的节点。第二次遍历时，如果前缀和已存在，且对应节点不是当前遍历节点，则会删除当前节点和前缀和对应节点区间的所有节点，表示该区间的和为0
  - 代码

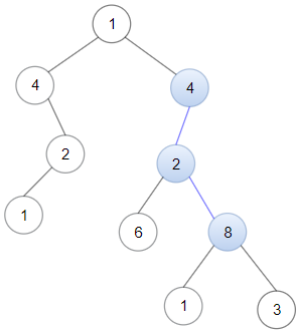
```
// 代码参考 https://leetcode-cn.com/problems/remove-zero-sum-consecutive-nodes-from-linked-list/solution/java-hashmap-liang-ci-bian-li-ji-ke-by-shane-34/
 // 前缀和
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode *dummy = new ListNode(0, head);

        unordered_map<int, ListNode*> preSum;

        // 首次遍历，建立<和，节点>前缀和表
        // 若同一和出现多次，会覆盖。即记录该sum出现的最后一次节点
        int sum = 0;
        for (ListNode *tmp = dummy; tmp != nullptr; tmp = tmp->next) {
            sum += tmp->val;
```

```
                preSum[sum] = tmp;
            }

            // 第二次遍历，若当前节点处sum在之后出现，表明两节点间和为0.之间删除该区间所有节点
            sum = 0;
            for (ListNode *tmp = dummy; tmp != nullptr; tmp = tmp->next) {
                sum += tmp->val;
                tmp->next = preSum[sum]->next;
            }

            return dummy->next;
        }
};
```

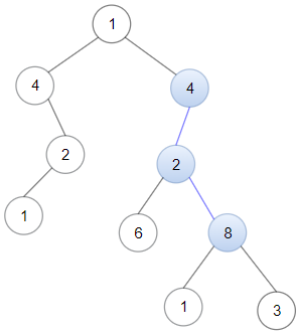
- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(n)$

1367. 二叉树中的列表

> 给你一棵以 root 为根的二叉树和一个 head 为第一个节点的链表。
>
> 如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 head 为首的链表中每个节点的值，那么请你返回 True ， 否则返回 False 。
>
> 一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。
>
> 示例 1：



> 输入：head = [4,2,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
> 输出：true
> 解释：树中蓝色的节点构成了与链表对应的子路径。
> 示例 2：



> 输入：head = [1,4,2,6], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
> 输出：true
> 示例 3：
>
> 输入：head = [1,4,2,6,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
> 输出：false
> 解释：二叉树中不存在一一对应链表的路径。

- 深搜
  - 思路：使用深搜，判断当前树、左孩子、右孩子中是否存在一条路径与链表对应。分为四种情况
    - 链表节点为空，表示匹配成功，返回true
    - 二叉树节点为空，表示匹配失败，返回false
    - 当前链表节点和二叉树节点值不相同，匹配失败，返回false
    - 以上都不满足，则表示值相同。匹配成功，继续更新链表节点在当前树的左孩子、右孩子中查找
  - 代码

```
// 原理参考 https://leetcode-cn.com/problems/linked-list-in-binary-tree/solution/er-cha-shu-zhong-de-lie-biao-by-leetcode-solution/
 // 代码参考 https://leetcode-cn.com/problems/linked-list-in-binary-tree/solution/java-shen-du-you-xian-sou-suo-zhu-xing-z-378y/
 // 深搜
class Solution {
public:
    bool isSubPath(ListNode* head, TreeNode* root) {
        if (root == nullptr) {
            return false;
        }
        // 检查当前子树和左右子树中是否存在满足要求的序列
        return dfs(head, root) || isSubPath(head, root->left) || isSubPath(head, root->right);
    }
    bool dfs(ListNode *head, TreeNode *root) {
        // 检查到了链表末尾，匹配成功
        if (head == nullptr) {
            return true;
        }
        // 树节点空了，链表节点还有，匹配失败
        if (root == nullptr) {
            return false;
        }
        // 值不相同，失败
        if (head->val != root->val) {
            return false;
        }
        // 当前节点匹配，检查左右子树和链表下一节点是否符合要求
        return dfs(head->next, root->left) || dfs(head->next, root->right);
    }
};
```

  - 复杂度
    - 时间复杂度 $O(n \times min(2^{len+ 1}, n))$。最坏情况下对所有节点进行匹配。假设一共有`n`个节点，对于一个节点为根的子树，如果是满二叉树。且每次匹配均为链表的

最后一节点失败，则被匹配到的节点数位$2^{len + 1} - 1$，即该节点为根的子树的往下`len`层的满二叉树的节点数目，其中`len`为链表长度，而二叉树总结点数目最多为`n`个，故最多匹配$min(2^{len + 1}, n)$次

- 空间复杂度 $O(height)$。递归栈空间，最多不会超过树高度

剑指 Offer 06. 从尾到头打印链表

> 输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。
>
> **示例 1：**
>
> ```
> 输入：head = [1,3,2]
> 输出：[2,3,1]
> ```

- 递归
  - 思路

    假设`reversePrint`函数返回的是逆序打印的数组，只需要在本层递归加上当前节点的值，并返回即可
  - 代码

```cpp
// https://leetcode-cn.com/problems/cong-wei-dao-tou-da-yin-lian-biao-lcof/solution/mian-shi-ti-06-cong-wei-dao-tou-da-yin-lian-biao-b/272164
 // 递归
class Solution {
public:
    vector<int> reversePrint(ListNode* head) {
        if (head == nullptr) {
            return {};
        }
        vector<int> res = reversePrint(head->next);
        res.push_back(head->val);
        return res;
    }
};
```

  - 复杂度
    - 时间复杂度 $O(n)$
    - 空间复杂度 $O(n)$

剑指 Offer 35. 复杂链表的复制

> 请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。
>
> 示例 1：
>
> 输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
> 输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
> 示例 2：

> 输入：head = [[1,1],[2,1]]
> 输出：[[1,1],[2,1]]
> 示例 3：
>
> 输入：head = [[3,null],[3,0],[3,null]]
> 输出：[[3,null],[3,0],[3,null]]
> 示例 4：
>
> 输入：head = []
> 输出：[]
> 解释：给定的链表为空（空指针），因此返回 null。

- 哈希表
  - 思路

    利用哈希表的查询特点，考虑构建 **<原链表节点：新链表节点>**的映射关系，再遍历构建新链表各节点的`next`和`random`引用指向即可
  - 算法流程
    - 若头结点`head`为空，直接返回`nullptr`
    - **初始化**：哈希表`lookup`，节点`curr`指向头结点
    - **复制链表**
      - 建立新节点，添加映射关系 **<原curr节点：新curr节点>**
      - `curr`更新到下一节点
    - **构建新链表的引用指向**
      - 构建新节点的`next`和`random`引用指向
      - `curr`更新到原链表下一节点
  - 代码

```cpp
// https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/solution/jian-zhi-offer-35-fu-za-lian-biao-de-fu-zhi-ha-xi-/
// 哈希表
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (head == nullptr) {
            return head;
        }

        Node *cur = head;
        unordered_map<Node*, Node*> lookup;

        // 复制节点，建立<原节点：新节点>的映射关系
        while (cur != nullptr) {
            lookup[cur] = new Node(cur->val);
            cur = cur->next;
        }

        cur = head;

        // 构建新链表的next和random指向
        while (cur != nullptr) {
            lookup[cur]->next = lookup[cur->next];
```

```
            lookup[cur]->random = lookup[cur->random];
            cur = cur->next;
        }

        // 返回新链表的头结点
        return lookup[head];
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(n)$，哈希表

- 拼接+拆分
  - 思路

    考虑构建 `原节点1->新节点1->原节点2->新节点2-> ......` 的拼接链表，如此可在访问原节点的 `random` 指向节点的同时找到新节点的 `random` 指向节点
  - 算法流程
    - **复制各节点，构建拼接链表**

      设原链表为 `node1->node2->...`，构建的拼接链为 `node1->node1_{new}->node2->node2_{new}...`
    - **构建新链表各节点的 `random` 指向**

      当访问原节点 `cur` 的随机指向节点 `cur->random` 时，对应新节点 `cur->next` 的随机指向节点为 `cur->random->next`
    - **拆分原/新链表**

      设置 `pre/cur` 分别指向 `原/新` 链表头结点，遍历执行 `pre->next = pre->next->next` 和 `cur->next = cur->next->next` 将两链表分开
    - 返回新链表的头结点 `res` 即可
  - 代码

```
// https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/solution/jian-zhi-offer-35-fu-za-lian-biao-de-fu-zhi-ha-xi-/
// 拼接+拆分
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (head == nullptr) {
            return nullptr;
        }

        // 1. 复制各节点，构建拼接链表
        Node *cur = head;
        while (cur != nullptr) {
            Node *tmp = new Node(cur->val);
            tmp->next = cur->next;
            cur->next = tmp;
            cur = tmp->next;
        }

        // 2. 构建新节点的random指向
        cur = head;
```

```
        while (cur != nullptr) {
            if (cur->random != nullptr) {
                cur->next->random = cur->random->next;
            }
            cur = cur->next->next;
        }

        // 3. 拆分链表
        cur = head->next;
        Node *pre = head, *res = head->next;
        while (cur->next != nullptr) {
            pre->next = pre->next->next;
            cur->next = cur->next->next;
            pre = pre->next;
            cur = cur->next;
        }
        // 处理原链表尾结点
        pre->next = nullptr;

        // 返回新链表头结点
        return res;
    }
};
```

- 复杂度
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(1)$