宫水三叶的刷题日记

链表

Author ：宫水三叶
Date ：2021/10/07
QQ Group: 703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「链表」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「链表」即可获取最新下载链接。

💡**下面介绍使用本合集的最佳使用实践：**

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「链表」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **2. 两数相加** ，难度为 **中等**。

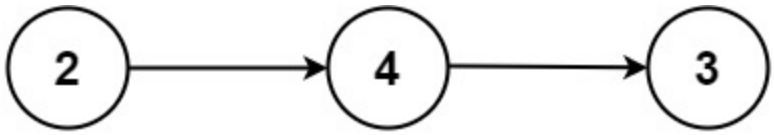
Tag：「递归」、「链表」、「数学」、「模拟」

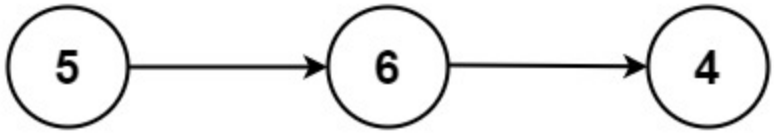
给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

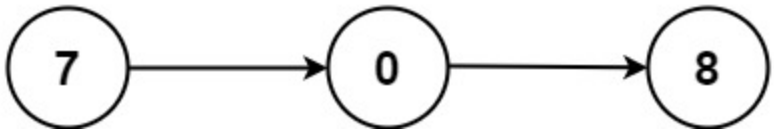
请你将两个数相加，并以相同形式返回一个表示和的链表。

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]
```

提示：

- 每个链表中的节点数在范围 [1, 100] 内
- 0 <= Node.val <= 9
- 题目数据保证列表表示的数字不含前导零

朴素解法（哨兵技巧）

这是道模拟题，模拟人工竖式做加法的过程：

从最低位至最高位，逐位相加，如果和大于等于 10，则保留个位数字，同时向前一位进 1 如果最高位有进位，则需在最前面补 1。

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

代码：

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode tmp = dummy;
        int t = 0;
        while (l1 != null || l2 != null) {
            int a = l1 != null ? l1.val : 0;
            int b = l2 != null ? l2.val : 0;
            t = a + b + t;
            tmp.next = new ListNode(t % 10);
            t /= 10;
            tmp = tmp.next;
            if (l1 != null) l1 = l1.next;
            if (l2 != null) l2 = l2.next;
        }
        if (t > 0) tmp.next = new ListNode(t);
        return dummy.next;
    }
}
```

- 时间复杂度：m 和 n 分别代表两条链表的长度，则遍历到的最远位置为 $max(m, n)$，复杂度为 $O(max(m, n))$
- 空间复杂度：创建了 $max(m, n) + 1$ 个节点（含哨兵），复杂度为 $O(max(m, n))$（忽略常数）

注意：事实上还有可能创建 $max(m + n) + 2$ 个节点，包含哨兵和最后一位的进位。但复杂度仍为 $O(max(m, n))$。

题目描述

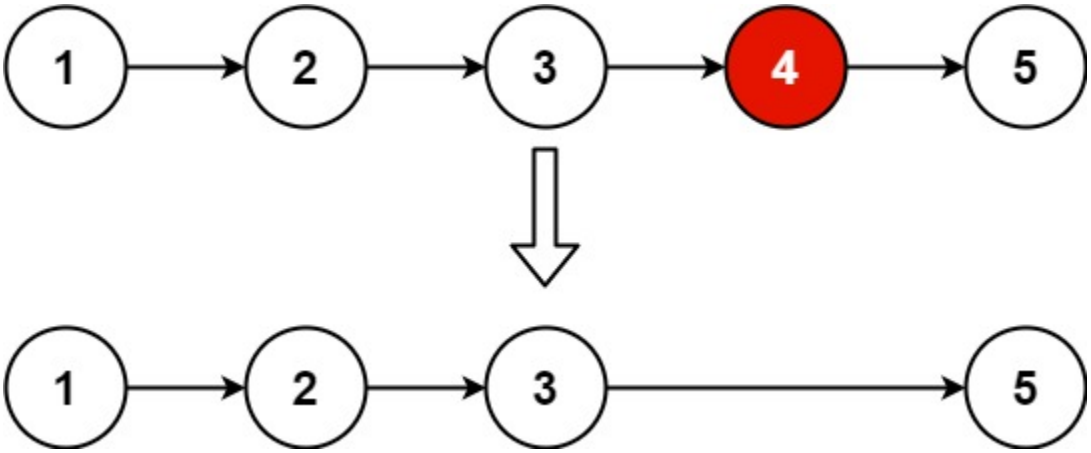
这是 LeetCode 上的 **19. 删除链表的倒数第 N 个结点** ，难度为 **中等**。

Tag：「链表」、「快慢指针」、「双指针」

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]
```

提示：

- 链表中结点的数目为 sz
- 1 <= sz <= 30
- 0 <= Node.val <= 100
- 1 <= n <= sz

快慢指针

删除链表的倒数第 `n` 个结点，首先要确定倒数第 `n` 个节点的位置。

我们可以设定两个指针，分别为 `slow` 和 `fast`，刚开始都指向 head。

然后先让 `fast` 往前走 `n` 步，`slow` 指针不动，这时候两个指针的距离为 `n`。

再让 `slow` 和 `fast` 同时往前走（保持两者距离不变），直到 `fast` 指针到达结尾的位置。

这时候 `slow` 会停在待删除节点的前一个位置，让 `slow.next = slow.next.next` 即可。

但这里有一个需要注意的边界情况是：如果链表的长度是 `L`，而我们恰好要删除的是倒数第 `L` 个节点（删除头节点），这时候 `fast` 往前走 `n` 步之后会变为 null，此时我们只需要让 `head = slow.next` 即可删除。

代码：

```Java
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if (head.next == null) return null;

        ListNode slow = head;
        ListNode fast = head;
        while (n-- > 0) fast = fast.next;

        if (fast == null) {
            head = slow.next;
        } else {
            while (fast.next != null) {
                slow = slow.next;
                fast = fast.next;
            }
            slow.next = slow.next.next;
        }
        return head;
    }
}
```

- 时间复杂度：需要扫描的长度为链表的长度。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

题目描述

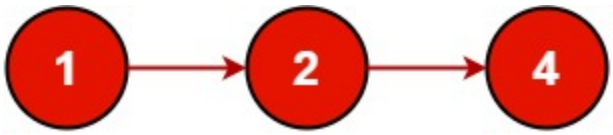
这是 LeetCode 上的 **21. 合并两个有序链表** ，难度为 **简单**。

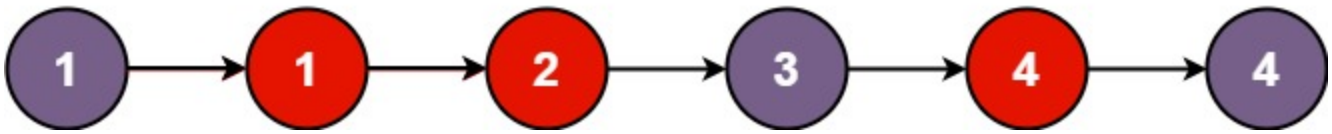
Tag : 「多路归并」、「链表」

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]
```

提示：

- 两个链表的节点数目范围是 [0, 50]
- -100 <= Node.val <= 100
- l1 和 l2 均按 非递减顺序 排列

多路归并（哨兵技巧）

哨兵技巧我们之前在「2. 两数相加」讲过啦，让三叶来帮你回忆一下：

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

由于两条链表本身就是有序的，只需要在遍历过程中进行比较即可：

代码：

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) return l2;
        if (l2 == null) return l1;

        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                cur = cur.next;
                l1 = l1.next;
            } else {
                cur.next = l2;
                cur = cur.next;
                l2 = l2.next;
            }
        }

        while (l1 != null) {
            cur.next = l1;
            cur = cur.next;
            l1 = l1.next;
        }
        while (l2 != null) {
            cur.next = l2;
            cur = cur.next;
            l2 = l2.next;
        }

        return dummy.next;
    }
}
```

- 时间复杂度：对两条链表扫描一遍。复杂度为 $O(n)$

- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **23. 合并K个升序链表** ，难度为 **中等**。

Tag：「优先队列」、「堆」、「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：[]
```

提示：

- k == lists.length
- 0 <= k <= 10^4
- 0 <= lists[i].length <= 500
- -10^4 <= lists[i][j] <= 10^4
- lists[i] 按 升序 排列
- lists[i].length 的总和不超过 10^4

优先队列（哨兵技巧）

哨兵技巧我们在前面的多道链表题讲过，让三叶来帮你回忆一下：

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

由于所有链表本身满足「升序」，一个直观的做法是，我们比较每条链表的头结点，选取值最小的节点，添加到结果中，然后更新该链表的的头结点为该节点的 next 指针。循环比较，直到所有的节点都被加入结果中。

对应到代码的话，相当于我们需要准备一个「集合」，将所有链表的头结点放入「集合」，然后每次都从「集合」中挑出最小值，并将最小值的下一个节点添加进「集合」（如果有的话），循环这个过程，直到「集合」为空（说明所有节点都处理完，进过集合又从集合中出来）。

而「堆」则是满足这样要求的数据结构。

代码：

```java
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        ListNode dummy = new ListNode(-1), tail = dummy;
        PriorityQueue<ListNode> q = new PriorityQueue<>((a, b) -> a.val - b.val);
        for (ListNode node : lists) {
            if (node != null) q.add(node);
        }
        while (!q.isEmpty()) {
            ListNode poll = q.poll();
            tail.next = poll;
            tail = tail.next;
            if (poll.next != null) q.add(poll.next);
        }
        return dummy.next;
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

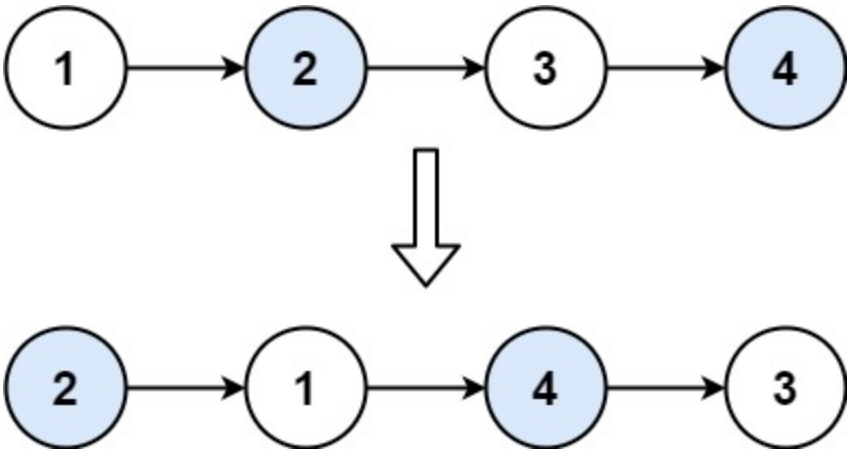
这是 LeetCode 上的 **24. 两两交换链表中的节点** ，难度为 **中等**。

Tag：「递归」、「链表」

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]
```

提示：

- 链表中节点的数目在范围 [0, 100] 内
- 0 <= Node.val <= 100

进阶：你能在不修改链表节点值的情况下解决这个问题吗?（也就是说，仅修改节点本身。）

递归解法（哨兵技巧）

哨兵技巧我们之前在前面的多道链表题讲过，让三叶来帮你回忆一下：

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

链表和树的题目天然适合使用递归来做。

我们可以设计一个递归函数，接受一个 `ListNode` 节点 root 作为参数，函数的作用是将 root 后面的两个节点进行交换，交换完成后再将下一个节点传入 ...

交换的前提条件：节点 root 后面至少有两个节点。同时别忘了应用我们的「哨兵技巧」。

代码：

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        recursive(dummy);
        return dummy.next;
    }
    void recursive(ListNode root) {
        if (root.next != null && root.next.next != null) {
            ListNode a = root.next, b = a.next;
            root.next = b;
            a.next = b.next;
            b.next = a;
            recursive(a);
        }
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

迭代解法（哨兵技巧）

所有的递归都能转化为迭代。

代码：

```java
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        for (ListNode cur = dummy; cur.next != null && cur.next.next != null;) {
            ListNode a = cur.next, b = a.next;
            cur.next = b;
            a.next = b.next;
            b.next = a;
            cur = a;
        }
        return dummy.next;
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **25. K 个一组翻转链表** ，难度为 **困难**。

Tag : 「递归」、「迭代」、「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。

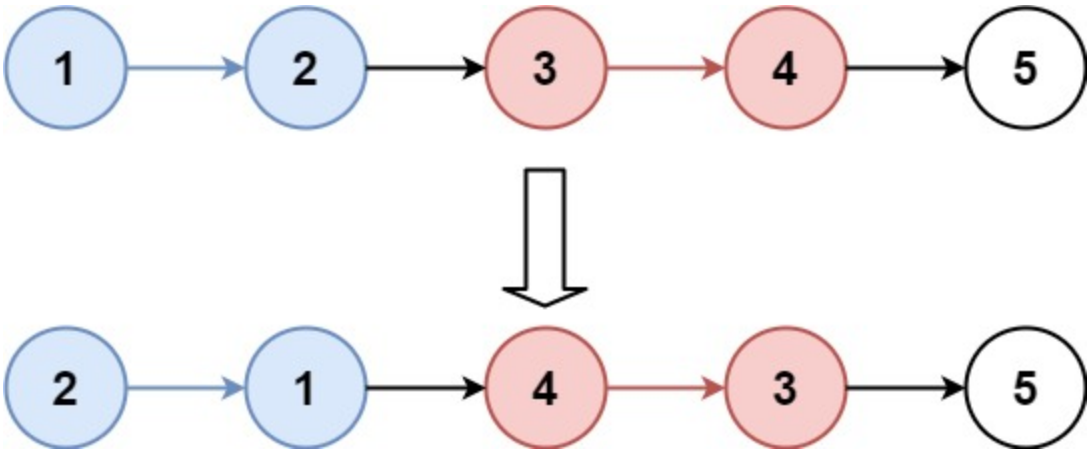
k 是一个正整数，它的值小于或等于链表的长度。

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。

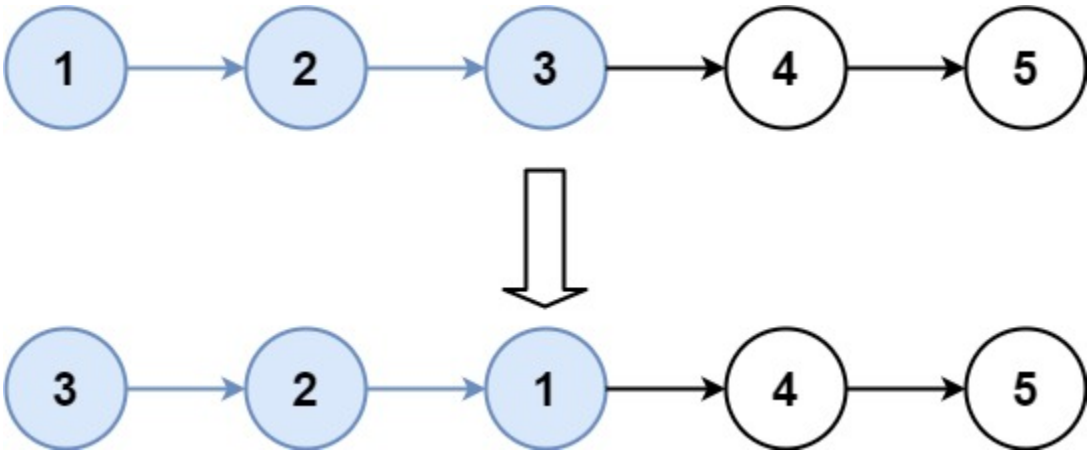
示例 1：




```
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]
```

提示：

- 列表中节点的数量在范围 sz 内
- 1 <= sz <= 5000
- 0 <= Node.val <= 1000
- 1 <= k <= sz

迭代解法（哨兵技巧）

哨兵技巧我们在前面的多道链表题讲过，让三叶来帮你回忆一下：

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

链表和树的题目天然适合使用递归来做。

但这次我们先将简单的「递归版本」放一放，先搞清楚迭代版本该如何实现。

我们可以设计一个翻转函数 `reverse` ：

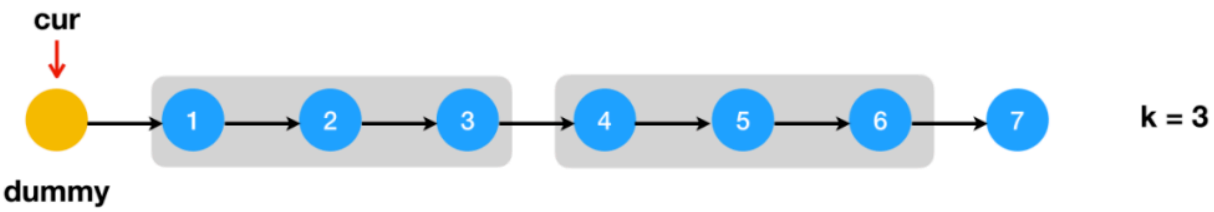
传入节点 `root` 作为参数，函数的作用是将以 `root` 为起点的 k 个节点进行翻转。

当以 `root` 为起点的长度为 k 的一段翻转完成后，再将下一个起始节点传入，直到整条链表都被处理完成。

当然，在 `reverse` 函数真正执行翻转前，需要先确保节点 `root` 后面至少有 k 个节点。

我们可以结合图解再来体会一下这个过程：

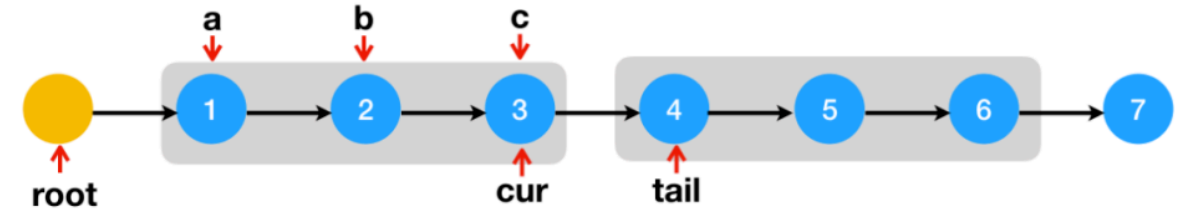
假设当前样例为 `1->2->3->4->5->6->7` 和 `k = 3` ：

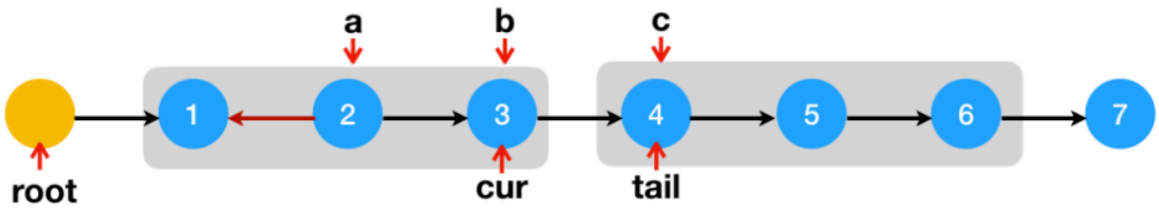


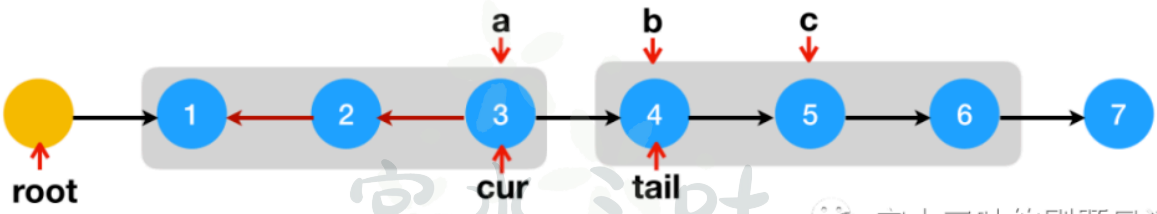
执行 reverse(cur, 3)

然后我们调用 `reverse(cur, k)`，在 `reverse()` 方法内部，几个指针的指向如图所示，会通过先判断 `cur` 是否为空，从而确定是否有足够的节点进行翻转：

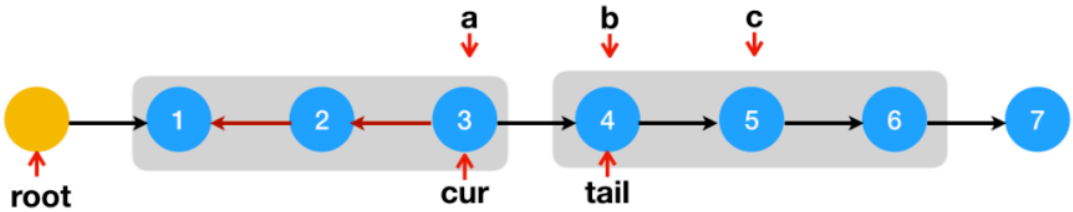
然后先通过 `while` 循环，将中间的数量为 `k - 1` 的 next 指针进行翻转：

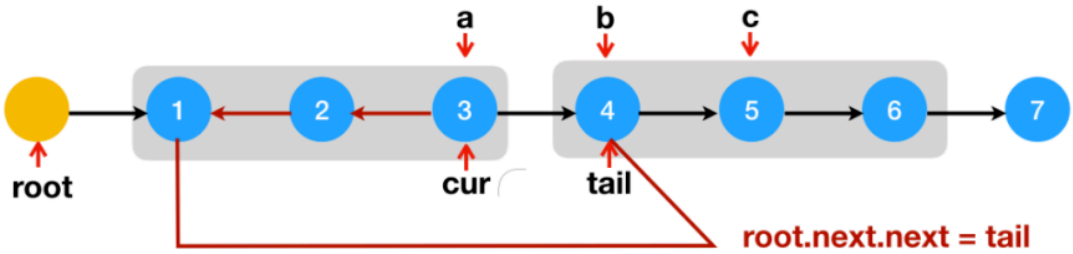


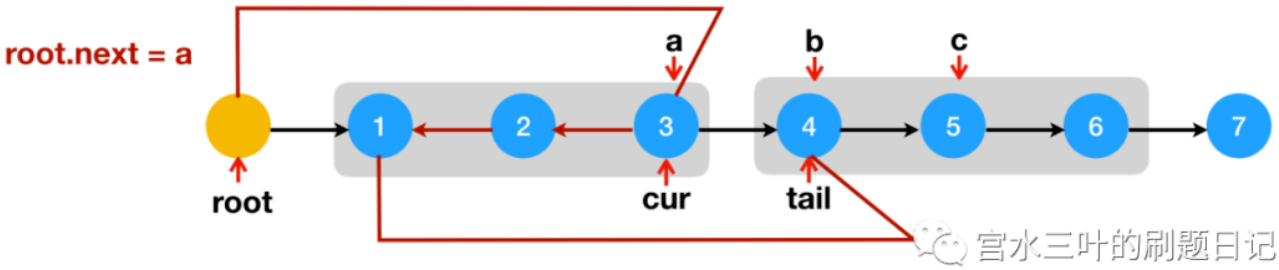




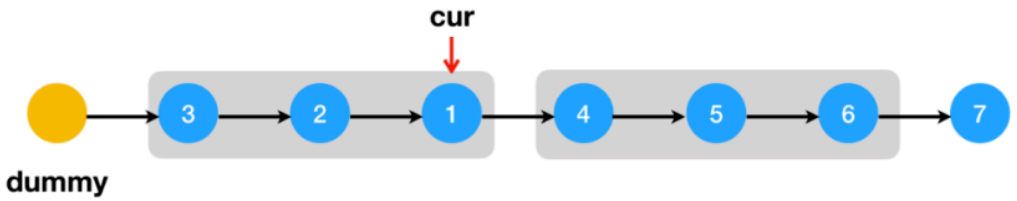
最后再处理一下局部的头结点和尾结点，这样一次 `reverse(cur, k)` 执行就结束了：







回到主方法，将 `cur` 往前移动 `k` 步，再调用 `reverse(cur, k)` 实现 `k` 个一组翻转：



cur 往前走 k 步，执行 reverse(cur, 3)

代码：

```Java
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode cur = dummy;
        while (cur != null) {
            reverse(cur, k);
            int u = k;
            while (u-- > 0 && cur != null) cur = cur.next;
        }
        return dummy.next;
    }
    // reverse 的作用是将 root 后面的 k 个节点进行翻转
    void reverse(ListNode root, int k) {
        // 检查 root 后面是否有 k 个节点
        int u = k;
        ListNode cur = root;
        while (u-- > 0 && cur != null) cur = cur.next;
        if (cur == null) return;

        // 进行翻转
        ListNode tail = cur.next;
        ListNode a = root.next, b = a.next;
        // 当需要翻转 k 个节点时，中间就有 k - 1 个 next 指针需要翻转
        while (k-- > 1) {
            ListNode c = b.next;
            b.next = a;
            a = b;
            b = c;
        }
        root.next.next = tail;
        root.next = a;
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

递归解法

搞懂了较难的「迭代哨兵」版本之后，常规的「递归无哨兵」版本写起来应该更加容易了。

需要注意的是，当我们不使用「哨兵」时，检查是否足够 k 位，只需要检查是否有 $k-1$ 个 $next$ 指针即可。

代码：

```
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        int u = k;
        ListNode p = head;
        while (p != null && u-- > 1) p = p.next;
        if (p == null) return head;

        ListNode tail = head;
        ListNode prev = head, cur = prev.next;
        u = k;
        while (u-- > 1) {
            ListNode tmp = cur.next;
            cur.next = prev;
            prev = cur;
            cur = tmp;
        }
        tail.next = reverseKGroup(cur, k);
        return prev;
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：只有忽略递归带来的空间开销才是 $O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

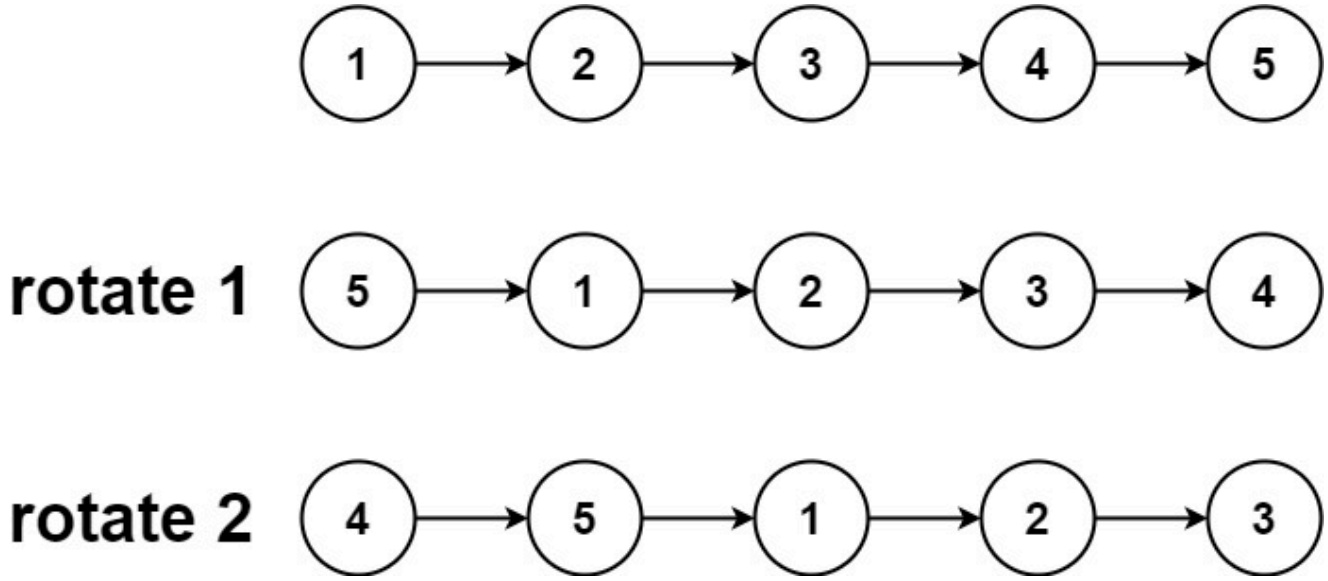
题目描述

这是 LeetCode 上的 **61. 旋转链表** ，难度为 **中等**。

Tag：「链表」、[快慢指针]

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。

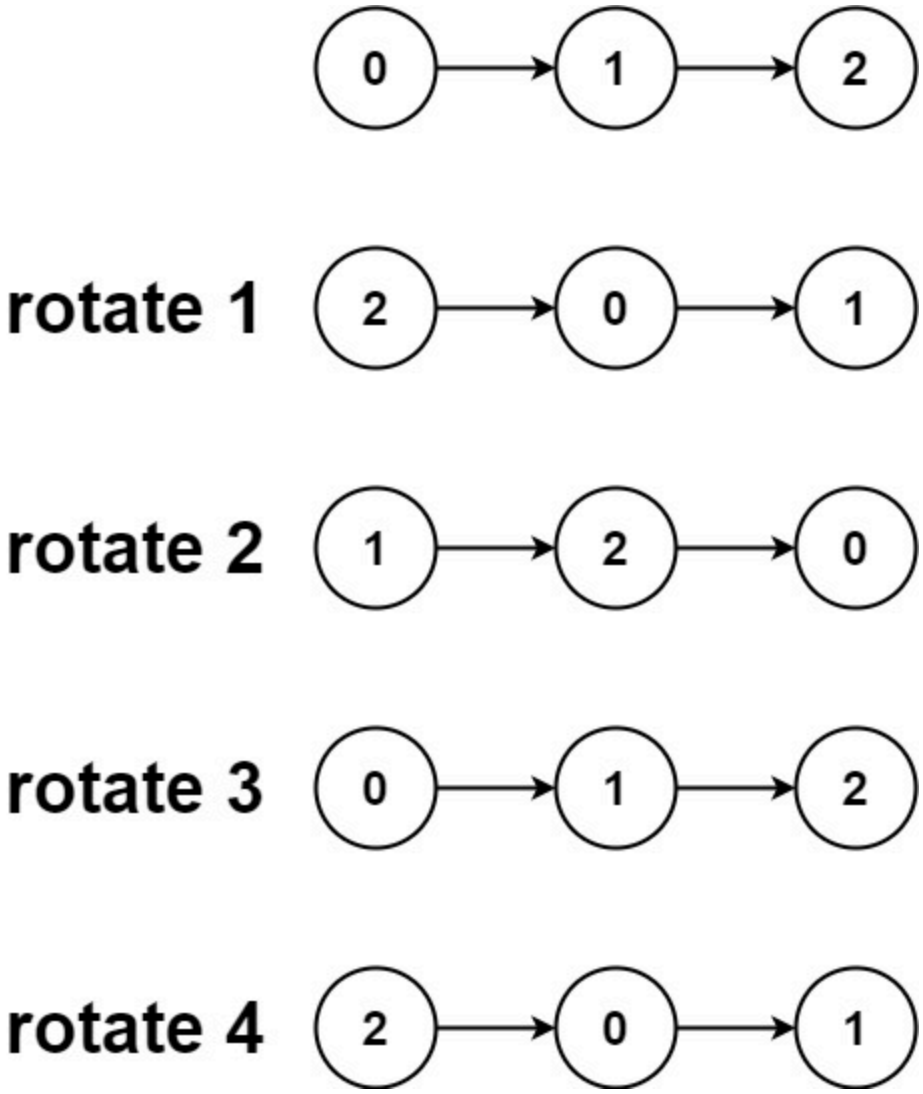
示例 1：



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]
```

提示：

- 链表中节点的数目在范围 [0, 500] 内
- -100 <= Node.val <= 100
- 0 <= k <= 2 * 10^9

快慢指针

本质还是道模拟题，分步骤处理即可：

- 避免不必要的旋转：与链表长度成整数倍的「旋转」都是没有意义的（旋转前后链表不变）
- 使用「快慢指针」找到倒数第 k 个节点（新头结点），然后完成基本的链接与断开与断开操作

代码（感谢 @Die Eule 同学提供的 cpp 和 js 版本）：

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || k == 0) return head;
        // 计算有效的 k 值：对于与链表长度成整数倍的「旋转」都是没有意义的（旋转前后链表不变）
        int tot = 0;
        ListNode tmp = head;
        while (tmp != null && ++tot > 0) tmp = tmp.next;
        k %= tot;
        if (k == 0) return head;

        // 使用「快慢指针」找到倒数第 k 个节点（新头结点）：slow 会停在「新头结点」的「前一位」，也就是
        ListNode slow = head, fast = head;
        while (k-- > 0) fast = fast.next;
        while (fast.next != null) {
            slow = slow.next;
            fast = fast.next;
        }
        // 保存新头结点，并将新尾结点的 next 指针置空
        ListNode nHead = slow.next;
        slow.next = null;
        // 将新链表的前半部分（原链表的后半部分）与原链表的头结点链接上
        fast.next = head;
        return nHead;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

闭合成环

另外一个做法是，先成环，再断开：

- 找到原链表的最后一个节点，将其与原链表的头结点相连（成环），并统计链表长

度，更新有效 k 值

- 从原链表的头节点出发，找到需要断开的点，进行断开

代码：

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || k == 0) return head;
        // 先将链表成环，并记录链表的长度
        // tmp 会记录住原链表最后一位节点
        int tot = 0;
        ListNode tmp = head;
        while (tmp.next != null && ++tot > 0) tmp = tmp.next;
        tot++;
        k %= tot;
        if (k == 0) return head;

        // 正式成环
        tmp.next = head;

        // 从原链表 head 出发，走 tot - k - 1 步，找到「新尾结点」进行断开，并将其下一个节点作为新节点
        k = tot - k - 1;
        while (k-- > 0) head = head.next;
        ListNode nHead = head.next;
        head.next = null;
        return nHead;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **83. 删除排序链表中的重复元素** ，难度为 **简单**。

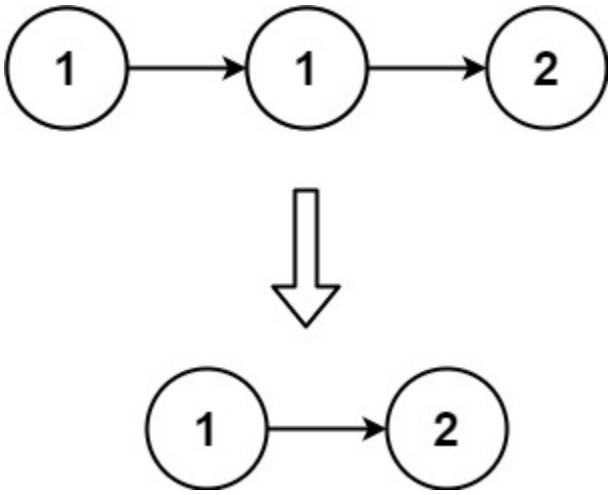
Tag：「链表」

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 。

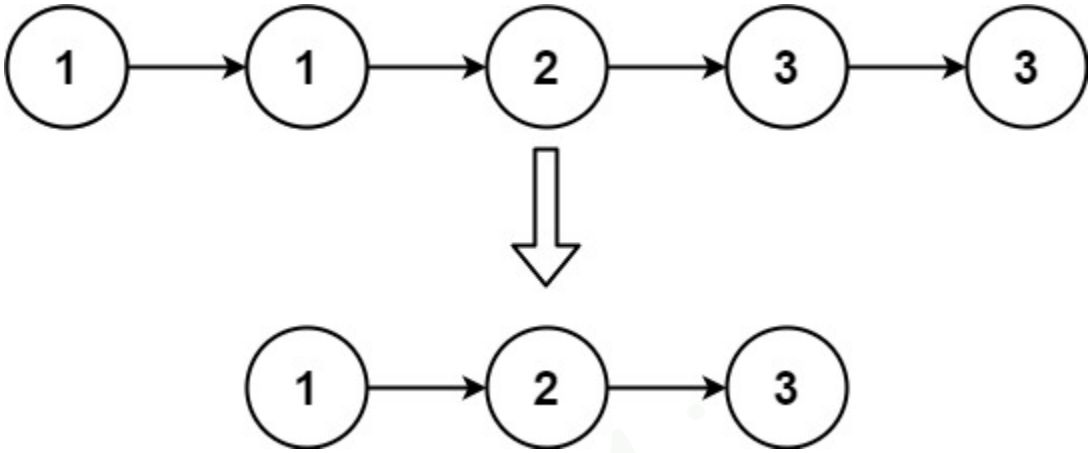
返回同样按升序排列的结果链表。

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]
```

提示：

- 链表中节点数目在范围 [0, 300] 内
- -100 <= Node.val <= 100
- 题目数据保证链表已经按升序排列

基本思路

还是与 82. 删除排序链表中的重复元素 II 相似的解题：

1. 建一个「虚拟头节点」dummy 以减少边界判断，往后的答案链表会接在 dummy 后面
2. 使用 tail 代表当前有效链表的结尾
3. 通过原输入的 head 指针进行链表扫描

代码：

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return head;
        ListNode dummy = new ListNode(-109);
        ListNode tail = dummy;
        while (head != null) {
            // 值不相等才追加，确保了相同的节点只有第一个会被添加到答案
            if (tail.val != head.val) {
                tail.next = head;
                tail = tail.next;
            }
            head = head.next;
        }
        tail.next = null;
        return dummy.next;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

拓展

- 「重复元素全部删除」，该如何实现？

82. 删除排序链表中的重复元素 II

```java
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode();
        ListNode tail = dummy;
        while (head != null) {
            // 进入循环时，确保了 head 不会与上一节点相同
            if (head.next == null || head.val != head.next.val) {
                tail.next = head;
                tail = tail.next;
            }
            // 如果 head 与下一节点相同，跳过相同节点
            while (head.next != null && head.val == head.next.val) head = head.next;
            head = head.next;
        }
        tail.next = null;
        return dummy.next;
    }
}
```

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

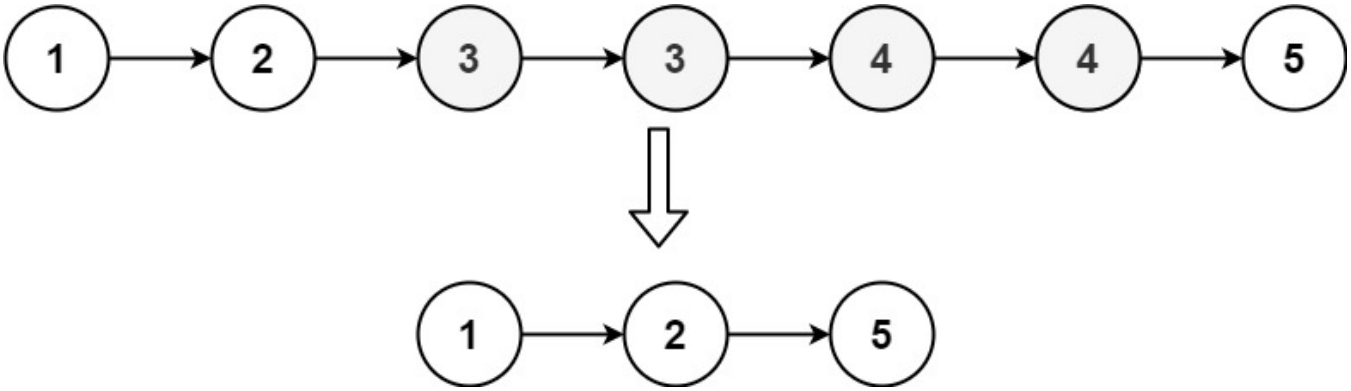
这是 LeetCode 上的 **82. 删除排序链表中的重复元素 II** ，难度为 **中等**。

Tag：「链表」

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。

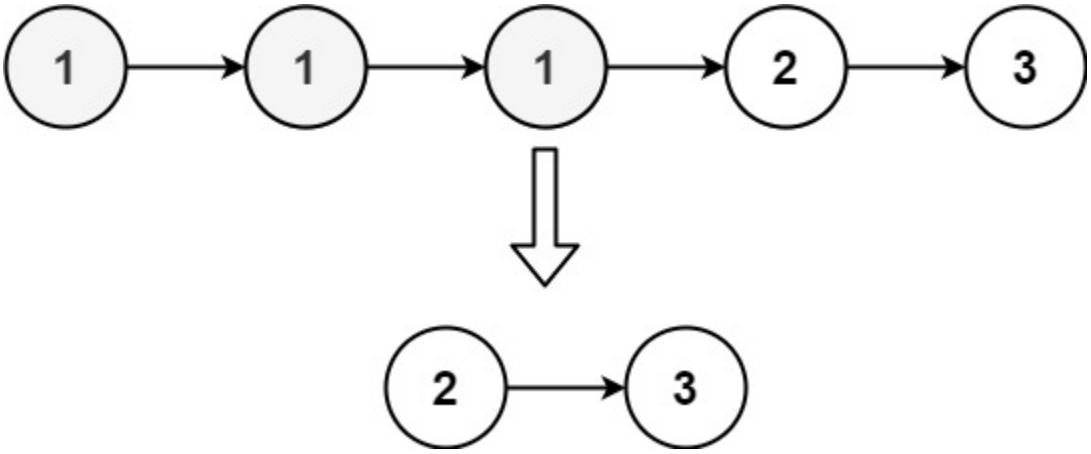
返回同样按升序排列的结果链表。

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]
```

提示：

- 链表中节点数目在范围 [0, 300] 内
- -100 <= Node.val <= 100
- 题目数据保证链表已经按升序排列

基本思路

几乎所有的链表题目，都具有相似的解题思路。

1. 建一个「虚拟头节点」dummy 以减少边界判断，往后的答案链表会接在 dummy 后面
2. 使用 tail 代表当前有效链表的结尾
3. 通过原输入的 head 指针进行链表扫描

我们会确保「进入外层循环时 head 不会与上一节点相同」，因此插入时机：

1. head 已经没有下一个节点，head 可以被插入
2. head 有一下个节点，但是值与 head 不相同，head 可以被插入

代码：

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode();
        ListNode tail = dummy;
        while (head != null) {
            // 进入循环时，确保了 head 不会与上一节点相同
            if (head.next == null || head.val != head.next.val) {
                tail.next = head;
                tail = head;
            }
            // 如果 head 与下一节点相同，跳过相同节点
            while (head.next != null && head.val == head.next.val) head = head.next;
            head = head.next;
        }
        tail.next = null;
        return dummy.next;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

拓展

- 如果问题变为「相同节点保留一个」，该如何实现？

83. 删除排序链表中的重复元素

```java
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return head;
        ListNode dummy = new ListNode(-109);
        ListNode tail = dummy;
        while (head != null) {
            // 值不相等才追加，确保了相同的节点只有第一个会被添加到答案
            if (tail.val != head.val) {
                tail.next = head;
                tail = tail.next;
            }
            head = head.next;
        }
        tail.next = null;
        return dummy.next;
    }
}
```

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

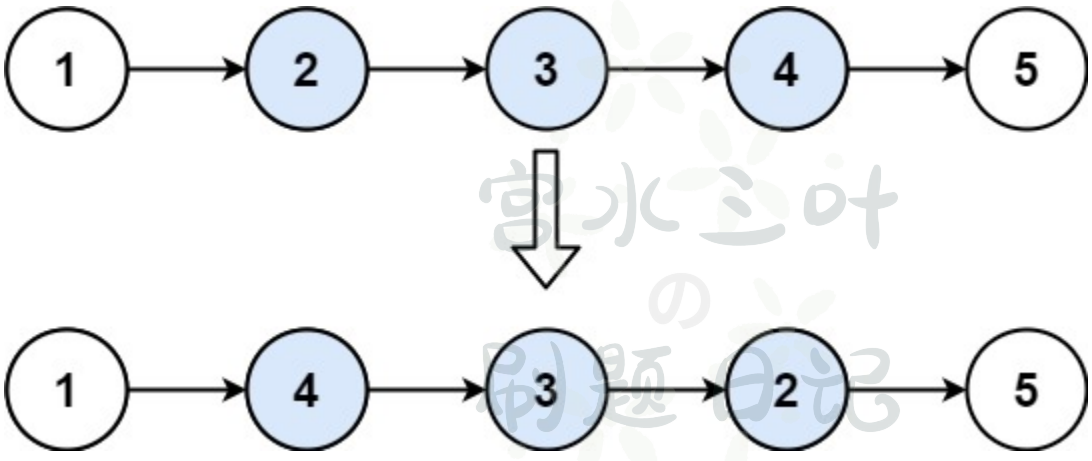
题目描述

这是 LeetCode 上的 **92. 反转链表 II** ，难度为 **中等**。

Tag：「链表」

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]
```

提示：

- 链表中节点数目为 n
- 1 <= n <= 500
- -500 <= Node.val <= 500
- 1 <= left <= right <= n

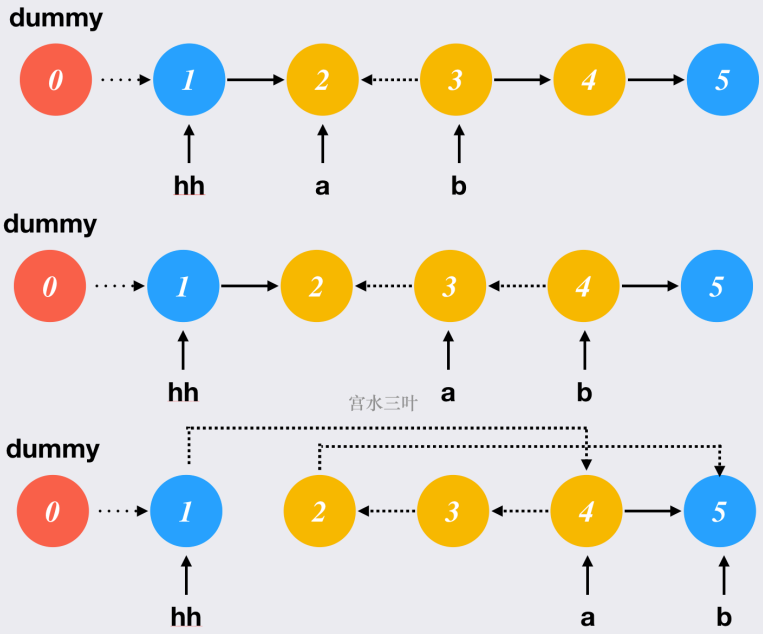
朴素解法

为了减少边界判断，我们可以建立一个虚拟头结点 dummy，使其指向 head，最终返回 dummy.next。

这种「哨兵」技巧能应用在所有的「链表」题目。

黄色部分的节点代表需要「翻转」的部分：

之后就是常规的模拟，步骤我写在示意图里啦 ~



1. 建一个虚拟头结点 *dummy* ，指向 *head* 节点
2. 建立 *hh* 指针，一直往右移动至 *left* 的前一位置
3. 使用 *a*、*b* 指针，将目标节点的 *next* 指针翻转
4. 让 *hh.next*（也就是 *left* 节点）的 *next* 指针指向 *b*
5. 让 *hh* 的 *next* 指针指向 *a*
6. 返回 *dummy.next*

代码：

```java
class Solution {
    public ListNode reverseBetween(ListNode head, int l, int r) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        r -= l;
        ListNode hh = dummy;
        while (l-- > 1) hh = hh.next;

        ListNode a = hh.next, b = a.next;
        while (r-- > 0) {
            ListNode tmp = b.next;
            b.next = a;
            a = b;
            b = tmp;
        }

        hh.next.next = b;
        hh.next = a;
        return dummy.next;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **138. 复制带随机指针的链表** ，难度为 **中等**。

Tag：「哈希表」、「链表」

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

构造这个链表的 深拷贝。 深拷贝应该正好由 n 个 全新 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。复制链表中的指针都不应指向原链

表中的节点 。

例如，如果原链表中有 X 和 Y 两个节点，其中 X.random --> Y 。那么在复制链表中对应的两个节点 x 和 y ，同样有 x.random --> y 。

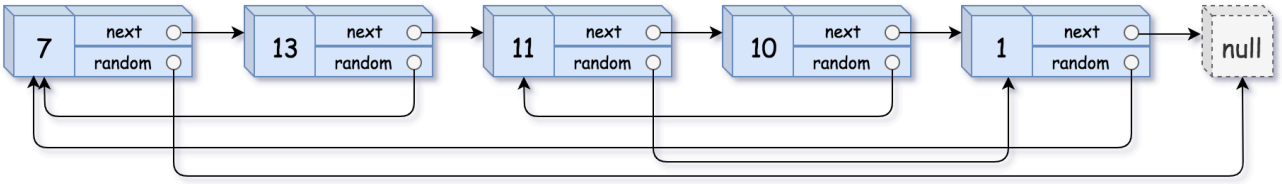
返回复制链表的头节点。

用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

- val：一个表示 Node.val 的整数。
- random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null 。

你的代码 只 接受原链表的头节点 head 作为传入参数。

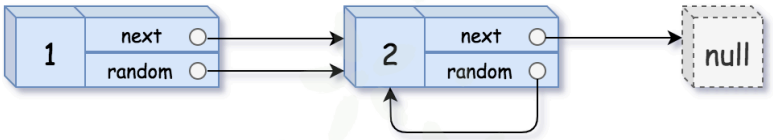
示例 1：




```
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]]
```

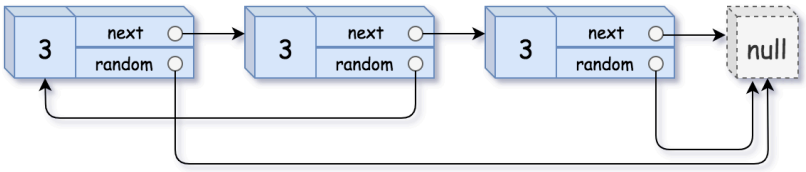
示例 2：




```
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。
```

提示：

- 0 <= n <= 1000
- -10000 <= Node.val <= 10000

模拟 + 哈希表

如果不考虑 `random` 指针的话，对一条链表进行拷贝，我们只需要使用两个指针：一个用于遍历原链表，一个用于构造新链表（始终指向新链表的尾部）即可。这一步操作可看做是「创建节点 + 构建 `next` 指针关系」。

现在在此基础上增加一个 `random` 指针，我们可以将 `next` 指针和 `random` 指针关系的构建拆开进行：

1. 先不考虑 `random` 指针，和原本的链表复制一样，创建新新节点，并构造 `next` 指针关系，同时使用「哈希表」记录原节点和新节点的映射关系；
2. 对原链表和新链表进行同时遍历，对于原链表的每个节点上的 `random` 都通过「哈希表」找到对应的新 `random` 节点，并在新链表上构造 `random` 关系。

代码：

```
class Solution {
    public Node copyRandomList(Node head) {
        Map<Node, Node> map = new HashMap<>();
        Node dummy = new Node(-1);
        Node tail = dummy, tmp = head;
        while (tmp != null) {
            Node node = new Node(tmp.val);
            map.put(tmp, node);
            tail.next = node;
            tail = tail.next;
            tmp = tmp.next;
        }
        tail = dummy.next;
        while (head != null) {
            if (head.random != null) tail.random = map.get(head.random);
            tail = tail.next;
            head = head.next;
        }
        return dummy.next;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

模拟（原地算法）

显然时间复杂度上无法优化，考虑如何降低空间（不使用「哈希表」）。

我们使用「哈希表」的目的为了实现原节点和新节点的映射关系，更进一步的是为了快速找到某个节点 `random` 在新链表的位置。

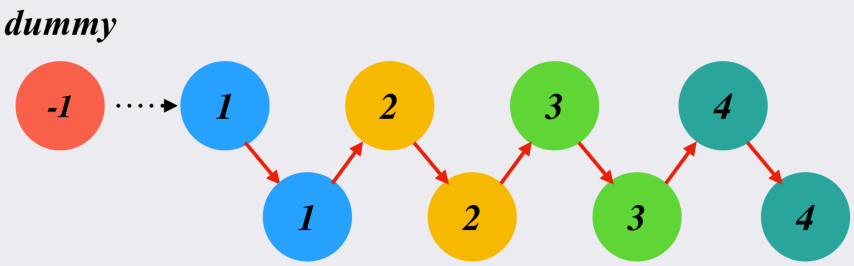
那么我们可以利用原链表的 `next` 做一个临时中转，从而实现映射。

具体的，我们可以按照如下流程进行：

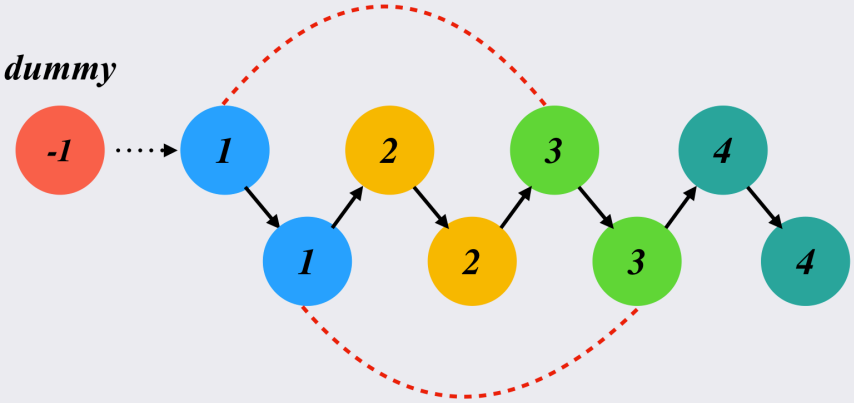
1. 对原链表的每个节点节点进行复制，并追加到原节点的后面；
2. 完成 1 操作之后，链表的奇数位置代表了原链表节点，链表的偶数位置代表了新链表节点，且每个原节点的 `next` 指针执行了对应的新节点。这时候，我们需要构造

新链表的 `random` 指针关系，可以利用 `link[i + 1].random = link[i].random.next`，i 为奇数下标，含义为 **新链表节点的 `random` 指针指向旧链表对应节点的 `random` 指针的下一个值**；

3. 对链表进行拆分操作。



1. 创建节点，并在原链表上利用 next 指针构造“映射关系”



2. 利用“映射关系”，构建 random 关系

代码：

```Java
class Solution {
    public Node copyRandomList(Node head) {
        if (head == null) return null;
        Node dummy = new Node(-1);
        dummy.next = head;
        while (head != null) {
            Node node = new Node(head.val);
            node.next = head.next;
            head.next = node;
            head = node.next;
        }
        head = dummy.next;
        while (head != null) {
            if (head.random != null) {
                head.next.random = head.random.next;
            }
            head = head.next.next;
        }
        head = dummy.next;
        Node ans = head.next;
        while (head != null) {
            Node tmp = head.next;
            if (head.next != null) head.next = head.next.next;
            head = tmp;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

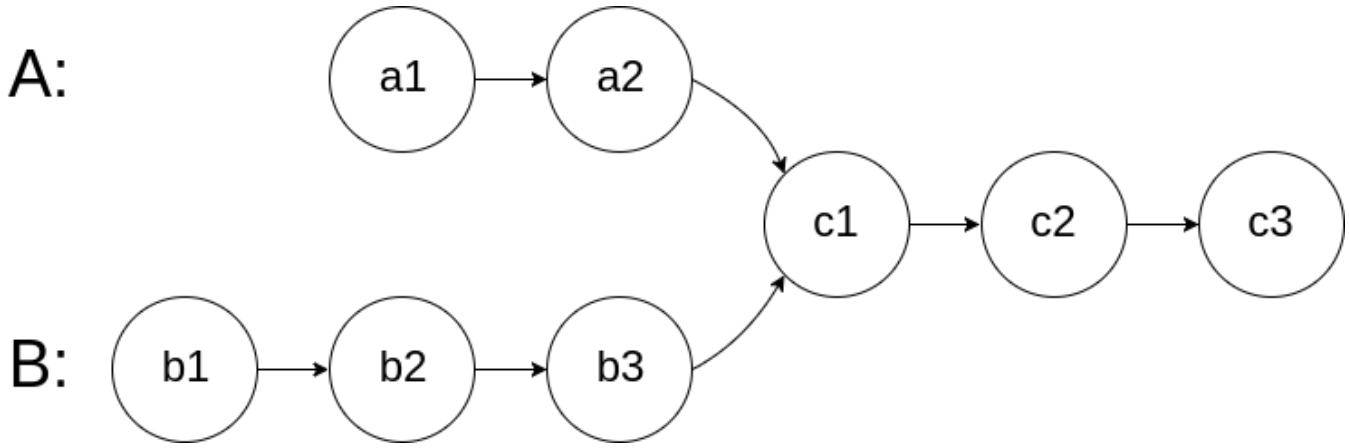
这是 LeetCode 上的 **160. 相交链表** ，难度为 **简单**。

Tag：「链表」、「栈」

给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果

两个链表没有交点，返回 null 。

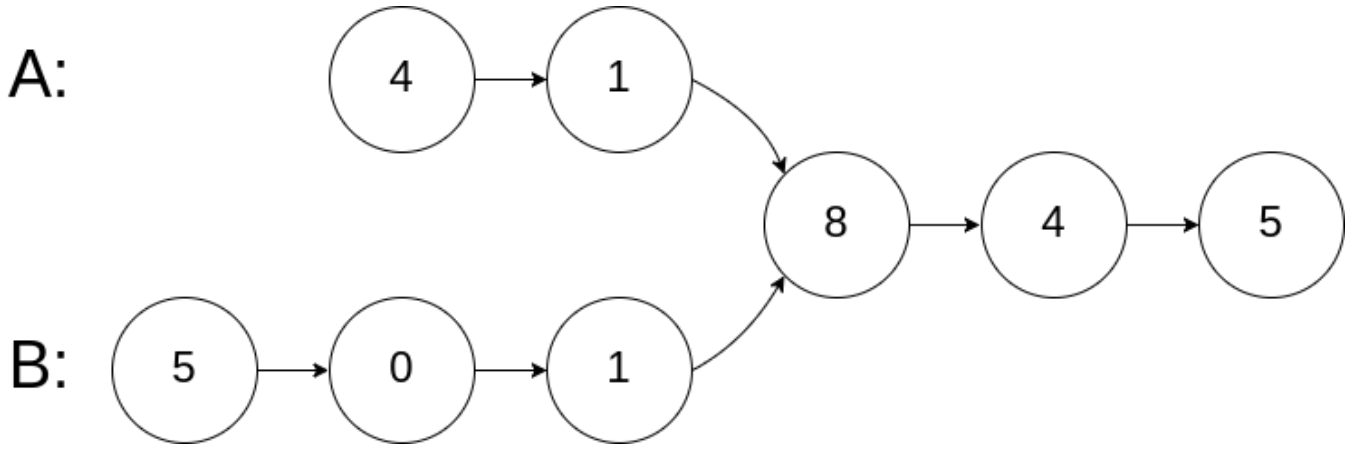
图示两个链表在节点 c1 开始相交：



题目数据 保证 整个链式结构中不存在环。

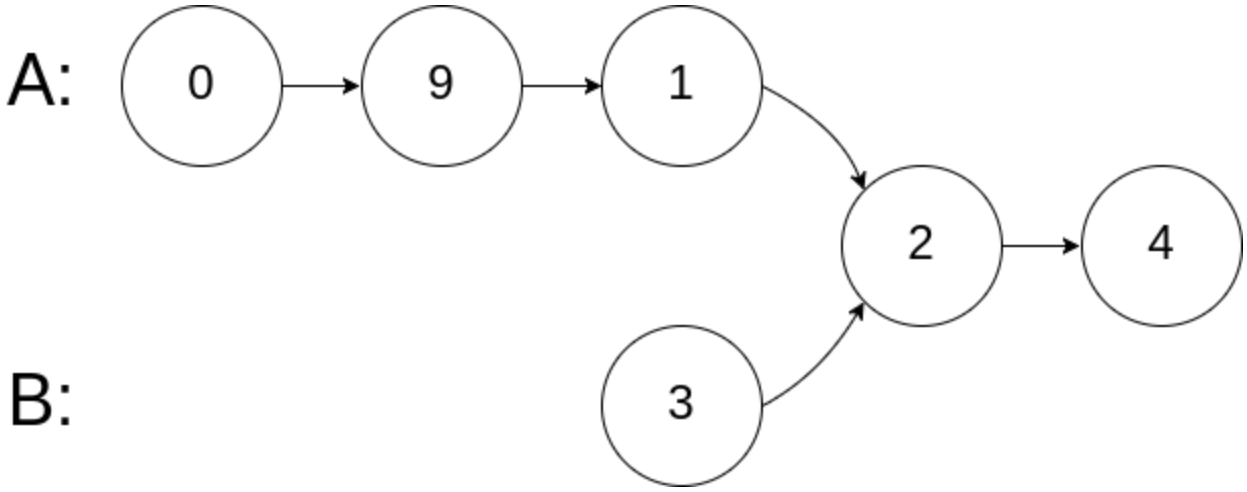
注意，函数返回结果后，链表必须 保持其原始结构 。

示例 1：



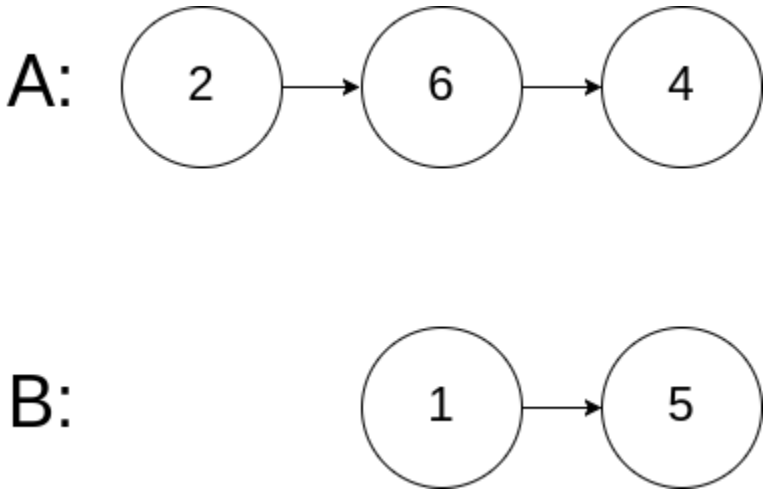

```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Intersected at '8'
解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

示例 2：



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Intersected at '2'
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
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。
这两个链表不相交，因此返回 null 。

提示：

- listA 中节点数目为 m
- listB 中节点数目为 n

- 0 <= m, n <= 3 * 10^4
- 1 <= Node.val <= 10^5
- 0 <= skipA <= m
- 0 <= skipB <= n
- **如果 listA 和 listB 没有交点**，intersectVal 为 0
- **如果 listA 和 listB 有交点**，intersectVal == listA[skipA + 1] == listB[skipB + 1]

进阶：你能否设计一个时间复杂度 $O(n)$ 、仅用 $O(1)$ 内存的解决方案？

朴素解法

一个朴素的做法是两层循环：当遇到第一个相同的节点时说明找到了；全都走完了还没遇到相同，说明不存在交点。

代码：

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode a, ListNode b) {
        for (ListNode h1 = a; h1 != null ; h1 = h1.next) {
            for (ListNode h2 = b; h2 != null ; h2 = h2.next) {
                if (h1.equals(h2)) return h1;
            }
        }
        return null;
    }
}
```

- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(1)$

栈解法

将两条链表分别压入两个栈中，然后循环比较两个栈的栈顶元素，同时记录上一位栈顶元素。

当遇到第一个不同的节点时，结束循环，上一位栈顶元素即是答案。

代码：

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode a, ListNode b) {
        Deque<ListNode> d1 = new ArrayDeque(), d2 = new ArrayDeque();
        while (a != null) {
            d1.addLast(a);
            a = a.next;
        }
        while (b != null) {
            d2.addLast(b);
            b = b.next;
        }
        ListNode ans = null;
        while (!d1.isEmpty() && !d2.isEmpty() && d1.peekLast().equals(d2.peekLast())) {
            ListNode c1 = d1.pollLast(), c2 = d2.pollLast();
            ans = c1;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n+m)$
- 空间复杂度：$O(n+m)$

差值解法

先对两条链表扫描一遍，取得两者长度，然后让长的链表先走「两者的长度差值」，然后再同时走，遇到第一个节点即是答案。

代码：

```
public class Solution {
    public ListNode getIntersectionNode(ListNode a, ListNode b) {
        int c1 = 0, c2 = 0;
        ListNode t1 = a, t2 = b;
        while (t1 != null && ++c1 > 0) t1 = t1.next;
        while (t2 != null && ++c2 > 0) t2 = t2.next;
        int t = Math.abs(c1 - c2);
        while (t-- > 0) {
            if (c1 > c2) a = a.next;
            else b = b.next;
        }
        while (a != null && b != null) {
            if (a.equals(b)) {
                return a;
            } else {
                a = a.next;
                b = b.next;
            }
        }
        return null;
    }
}
```

- 时间复杂度：$O(n + m)$
- 空间复杂度：$O(1)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **146. LRU 缓存机制** ，难度为 **中等**。

Tag：「设计」、「链表」、「哈希表」

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制 。
实现 LRUCache 类：

- LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存

- int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
- void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 $O(1)$ 时间复杂度内完成这两种操作？

示例：

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

提示：

- 1 <= capacity <= 3000
- 0 <= key <= 3000
- 0 <= value <= 10^4
- 最多调用 3 * 10^4 次 get 和 put

基本分析

LRU 是一种十分常见的页面置换算法。

将 LRU 翻译成大白话就是：**当不得不淘汰某些数据时（通常是容量已满），选择最久未被使用**

的数据进行淘汰。

题目让我们实现一个容量固定的 `LRUCache` 。如果插入数据时，发现容器已满时，则先按照 LRU 规则淘汰一个数据，再将新数据插入，其中「插入」和「查询」都算作一次“使用”。

可以通过  来理解，假设我们有容量为 2 的 `LRUCache` 和 测试键值对 `[1-1,2-2,3-3]` ，将其按照顺序进行插入 & 查询：

- 插入 `1-1` ，此时最新的使用数据为 `1-1`
- 插入 `2-2` ，此时最新使用数据变为 `2-2`
- 查询 `1-1` ，此时最新使用数据为 `1-1`
- 插入 `3-3` ，由于容器已经达到容量，需要先淘汰已有数据才能插入，这时候会淘汰 `2-2` ， `3-3` 成为最新使用数据

键值对存储方面，我们可以使用「哈希表」来确保插入和查询的复杂度为 $O(1)$。

另外我们还需要额外维护一个「使用顺序」序列。

我们期望当「新数据被插入」或「发生键值对查询」时，能够将当前键值对放到序列头部，这样当触发 LRU 淘汰时，只需要从序列尾部进行数据删除即可。

期望在 $O(1)$ 复杂度内调整某个节点在序列中的位置，很自然想到双向链表。

双向链表

具体的，我们使用哈希表来存储「键值对」，键值对的键作为哈希表的 Key，而哈希表的 Value 则使用我们自己封装的 `Node` 类， `Node` 同时作为双向链表的节点。

- 插入：检查当前键值对是否已经存在于哈希表：
  - 如果存在，则更新键值对，并将当前键值对所对应的 `Node` 节点调整到链表头部（ `refresh` 操作）
  - 如果不存在，则检查哈希表容量是否已经达到容量：
    - 没达到容量：插入哈希表，并将当前键值对所对应的 `Node` 节点调整到链表头部（ `refresh` 操作）
    - 已达到容量：先从链表尾部找到待删除元素进行删除（ `delete` 操作），然后再插入哈希表，并将当前键值对所对应的 `Node` 节点调整到链表头部（ `refresh` 操作）

- 查询：如果没在哈希表中找到该 Key，直接返回 -1；如果存在该 Key，则将对应的值返回，并将当前键值对所对应的 `Node` 节点调整到链表头部（ `refresh` 操作）

一些细节：

- 为了减少双向链表左右节点的「判空」操作，我们预先建立两个「哨兵」节点 `head` 和 `tail`。

代码：

```
class LRUCache {
    class Node {
        int k, v;
        Node l, r;
        Node(int _k, int _v) {
            k = _k;
            v = _v;
        }
    }
    int n;
    Node head, tail;
    Map<Integer, Node> map;
    public LRUCache(int capacity) {
        n = capacity;
        map = new HashMap<>();
        head = new Node(-1, -1);
        tail = new Node(-1, -1);
        head.r = tail;
        tail.l = head;
    }

    public int get(int key) {
        if (map.containsKey(key)) {
            Node node = map.get(key);
            refresh(node);
            return node.v;
        }
        return -1;
    }

    public void put(int key, int value) {
        Node node = null;
        if (map.containsKey(key)) {
            node = map.get(key);
            node.v = value;
        } else {
            if (map.size() == n) {
                Node del = tail.l;
                map.remove(del.k);
                delete(del);
            }
            node = new Node(key, value);
            map.put(key, node);
        }
        refresh(node);
    }
```

```
    // refresh 操作分两步：
    // 1. 先将当前节点从双向链表中删除（如果该节点本身存在于双向链表中的话）
    // 2. 将当前节点添加到双向链表头部
    void refresh(Node node) {
        delete(node);
        node.r = head.r;
        node.l = head;
        head.r.l = node;
        head.r = node;
    }

    // delete 操作：将当前节点从双向链表中移除
    // 由于我们预先建立 head 和 tail 两位哨兵，因此如果 node.l 不为空，则代表了 node 本身存在于双向链
    void delete(Node node) {
        if (node.l != null) {
            Node left = node.l;
            left.r = node.r;
            node.r.l = left;
        }
    }
}
```

- 时间复杂度：各操作均为 $O(1)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

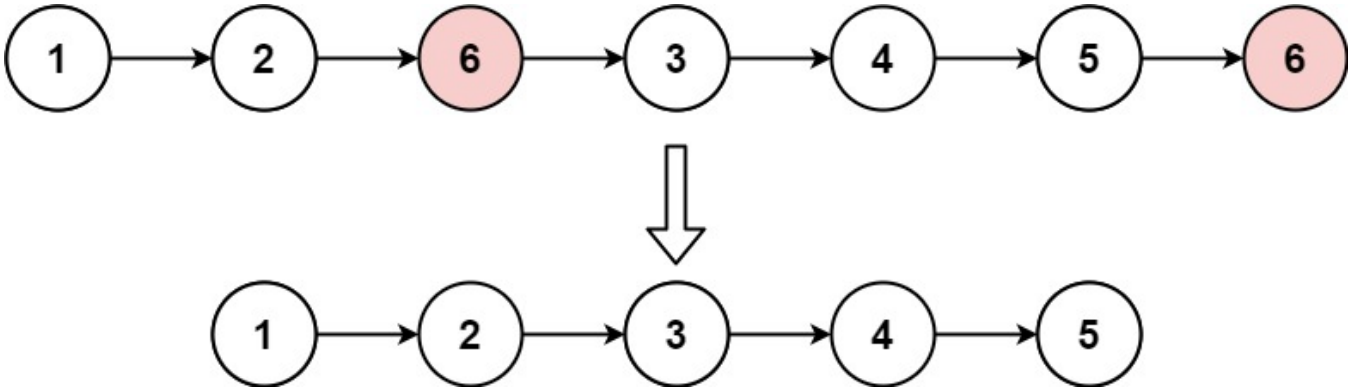
题目描述

这是 LeetCode 上的 **203. 移除链表元素** ，难度为 **简单**。

Tag：「链表」

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：[]
```

提示：

- 列表中的节点在范围 [0, 10^4] 内
- 1 <= Node.val <= 50
- 0 <= k <= 50

递归

一个直观的做法是：写一个递归函数来将某个值为 `val` 的节点从链表中移除。

由于是单链表，无法通过某个节点直接找到「前一个节点」，因此为了方便，我们可以为递归函数多设置一个入参，代表「前一个节点」。

代码：

```Java
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        dfs(dummy, dummy.next, val);
        return dummy.next;
    }
    void dfs(ListNode prev, ListNode root, int val) {
        if (root == null) return ;
        if (root.val == val) {
            prev.next = root.next;
        } else {
            prev = root;
        }
        dfs(prev, prev.next, val);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：忽略递归带来的额外空间开销。复杂度为 $O(1)$

迭代

同理，我们可以使用「迭代」方式来实现，而迭代有 `while` 和 `for` 两种写法。

代码：

```Java
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        for (ListNode tmp = dummy.next, prev = dummy; tmp != null; tmp = tmp.next) {
            if (tmp.val == val) {
                prev.next = tmp.next;
            } else {
                prev = tmp;
            }
        }
        return dummy.next;
    }
}
```

```java
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode tmp = dummy.next, prev = dummy;
        while (tmp != null) {
            if (tmp.val == val) {
                prev.next = tmp.next;
            } else {
                prev = tmp;
            }
            tmp = tmp.next;
        }
        return dummy.next;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **430. 扁平化多级双向链表** ，难度为 **中等**。

Tag：「链表」、「迭代」、「递归」

多级双向链表中，除了指向下一个节点和前一个节点指针之外，它还有一个子链表指针，可能指向单独的双向链表。这些子列表也可能会有一个或多个自己的子项，依此类推，生成多级数据结构，如下面的示例所示。

给你位于列表第一级的头节点，请你扁平化列表，使所有结点出现在单级双链表中。

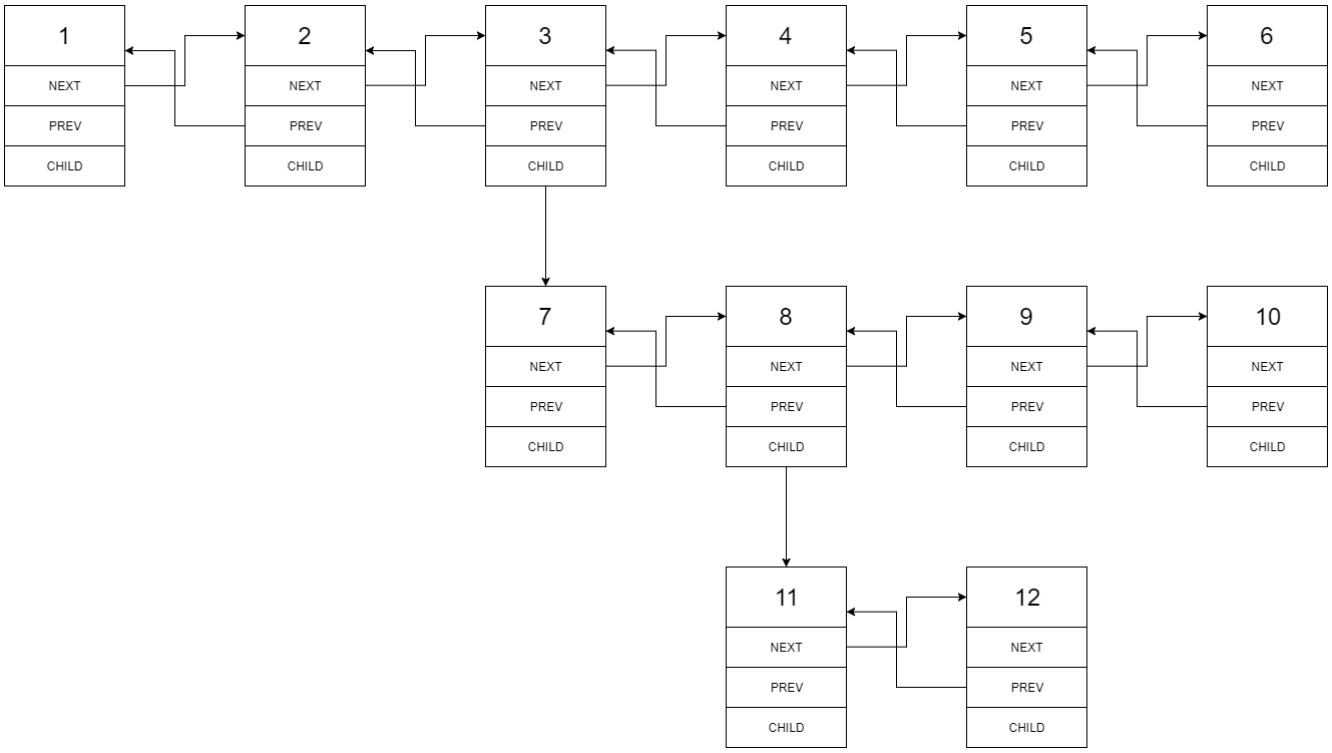
示例 1：

输入：`head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]`

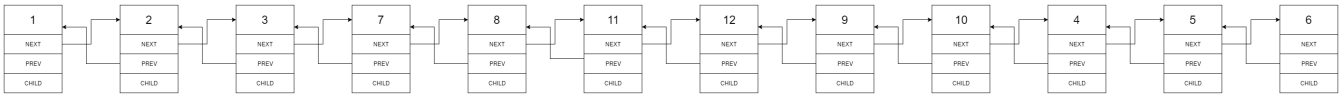
输出：`[1,2,3,7,8,11,12,9,10,4,5,6]`

解**释**：

输入的多级列表如下图所示：



扁平化后的链表如下图：



示例 2：

```
输入：head = [1,2,null,3]

输出：[1,3,2]

解释：
输入的多级列表如下图所示：

  1---2---NULL
  |
  3---NULL
```

示例 3：

```
输入：head = []

输出：[]
```

递归

一道常规链表模拟题。

利用 `flatten` 函数本身的含义（将链表头为 $head$ 的链表进行扁平化，并将扁平化后的头结点进行返回），我们可以很容易写出递归版本。

为防止空节点等边界问题，起始时建立一个哨兵节点 $dummy$ 指向 $head$，然后利用 $head$ 指针从前往后处理链表：

- 当前节点 $head$ 没有 $child$ 节点：直接让指针后即可，即 $head = head.next$；
- 当前节点 $head$ 有 $child$ 节点：将 $head.child$ 传入 `flatten` 函数递归处理，拿到普遍化后的头结点 $chead$，然后将 $head$ 和 $chead$ 建立“相邻”关系（注意要先存起来原本的 $tmp = head.next$ 以及将 $head.child$ 置空），然后继续往后处理，直到扁平化的 $chead$ 链表的尾部，将其与 tmp 建立“相邻”关系。

重复上述过程，直到整条链表被处理完。

执行结果： 通过 显示详情 >

添加备注

执行用时： 1 ms ，在所有 Java 提交中击败了 21.64% 的用户

内存消耗： 36 MB ，在所有 Java 提交中击败了 99.67% 的用户

通过测试用例： 26 / 26

炫耀一下：

写题解，分享我的解题思路

代码：

```java
class Solution {
    public Node flatten(Node head) {
        Node dummy = new Node(0);
        dummy.next = head;
        while (head != null) {
            if (head.child == null) {
                head = head.next;
            } else {
                Node tmp = head.next;
                Node chead = flatten(head.child);
                head.next = chead;
                chead.prev = head;
                head.child = null;
                while (head.next != null) head = head.next;
                head.next = tmp;
                if (tmp != null) tmp.prev = head;
                head = tmp;
            }
        }
        return dummy.next;
    }
}
```

- 时间复杂度：最坏情况下，每个节点会被访问 h 次（h 为递归深度，最坏情况下

$h = n$）。整体复杂度为 $O(n^2)$

- 空间复杂度：最坏情况下所有节点都分布在 `child` 中，此时递归深度为 n。复杂度为 $O(n)$

递归（优化）

在上述解法中，由于我们直接使用 `flatten` 作为递归函数，导致递归处理 $head.child$ 后不得不再进行遍历来找当前层的“尾结点”，这导致算法复杂度为 $O(n^2)$。

一个可行的优化是，额外设计一个递归函数 `dfs` 用于返回扁平化后的链表“尾结点”，从而确保我们找尾结点的动作不会在每层发生。

代码：

```java
class Solution {
    public Node flatten(Node head) {
        dfs(head);
        return head;
    }
    Node dfs(Node head) {
        Node last = head;
        while (head != null) {
            if (head.child == null) {
                last = head;
                head = head.next;
            } else {
                Node tmp = head.next;
                Node childLast = dfs(head.child);
                head.next = head.child;
                head.child.prev = head;
                head.child = null;
                if (childLast != null) childLast.next = tmp;
                if (tmp != null) tmp.prev = childLast;
                last = head;
                head = childLast;
            }
        }
        return last;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：最坏情况下所有节点都分布在 `child` 中，此时递归深度为 n。复杂度为 $O(n)$

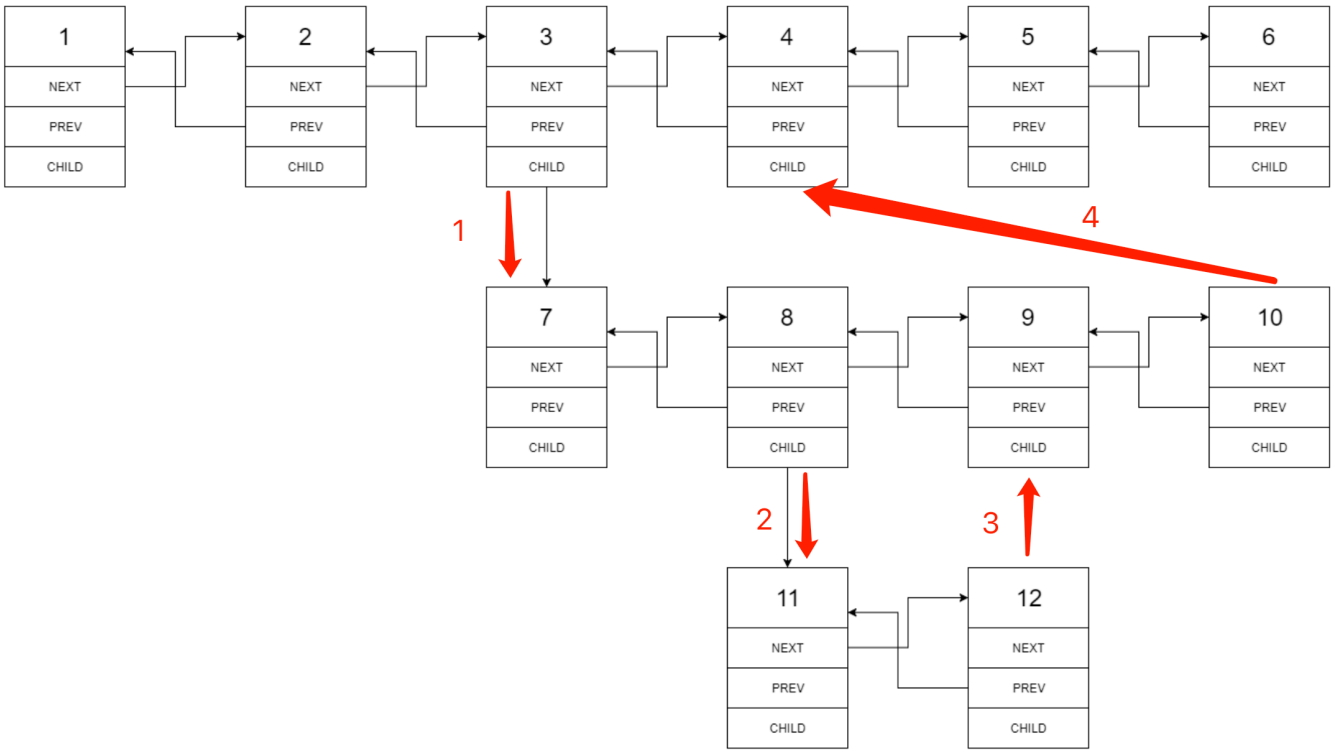
迭代

自然也能够使用迭代进行求解。

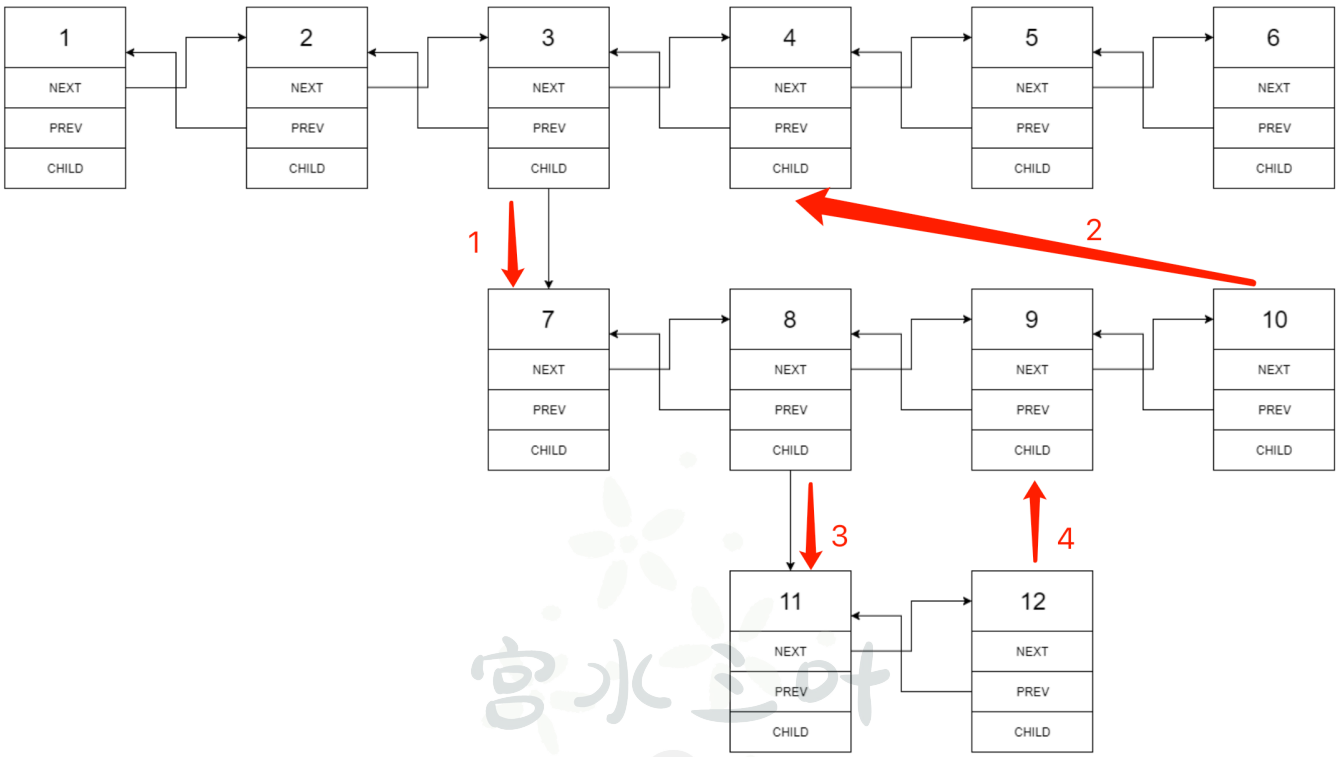
与「递归」不同的是，「迭代」是以“段”为单位进行扁平化，而「递归」是以深度（方向）进行扁平化，这就导致了两种方式对每个扁平节点的处理顺序不同。

已样例 1 为  。

递归的处理节点（新的 $next$ 指针的构建）顺序为：



迭代的处理节点（新的 $next$ 指针的构建）顺序为：



但由于链表本身不存在环，「迭代」的构建顺序发生调整，仍然可以确保每个节点被访问的次数为常数次。

执行结果： 通过 显示详情 >

添加备注

执行用时： 0 ms ，在所有 Java 提交中击败了 100.00% 的用户

内存消耗： 36 MB ，在所有 Java 提交中击败了 99.67% 的用户

通过测试用例： 26 / 26

炫耀一下：

写题解，分享我的解题思路

代码：

```
class Solution {
    public Node flatten(Node head) {
        Node dummy = new Node(0);
        dummy.next = head;
        for (; head != null; ) {
            if (head.child == null) {
                head = head.next;
            } else {
                Node tmp = head.next;
                Node child = head.child;
                head.next = child;
                child.prev = head;
                head.child = null;
                Node last = head;
                while (last.next != null) last = last.next;
                last.next = tmp;
                if (tmp != null) tmp.prev = last;
                head = head.next;
            }
        }
        return dummy.next;
    }
}
```

- 时间复杂度：可以发现，迭代写法的扁平化过程并不与遍历方向保持一致（以段为单位进行扁平化，而非像递归那样总是往遍历方向进行扁平化），但每个节点被访问的次数仍为常数次。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **460. LFU 缓存** ，难度为 **困难**。

Tag：「链表」、「双向链表」、「设计」

请你为 **最不经常使用（LFU）** 缓存算法设计并实现数据结构。

实现 `LFUCache` 类：

- `LFUCache(int capacity)` - 用数据结构的容量 `capacity` 初始化对象
- `int get(int key)` - 如果键存在于缓存中，则获取键的值，否则返回 -1。
- `void put(int key, int value)` - 如果键已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量时，则应该在插入新项之前，使最不经常使用的项无效。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除 **最近最久未使用** 的键。

注意「项的使用次数」就是自插入该项以来对其调用 `get` 和 `put` 函数的次数之和。使用次数会在对应项被移除后置为 0 。

为了确定最不常使用的键，可以为缓存中的每个键维护一个 使用计数器 。使用计数最小的键是最久未使用的键。

当一个键首次插入到缓存中时，它的使用计数器被设置为 1 (由于 `put` 操作)。对缓存中的键执行 `get` 或 `put` 操作，使用计数器的值将会递增。

示例：

```
输入：
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
输出：
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

解释：
// cnt(x) = 键 x 的使用计数
// cache=[] 将显示最后一次使用的顺序（最左边的元素是最近的）
LFUCache lFUCache = new LFUCache(2);
lFUCache.put(1, 1);   // cache=[1,_], cnt(1)=1
lFUCache.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lFUCache.get(1);      // 返回 1
                      // cache=[1,2], cnt(2)=1, cnt(1)=2
lFUCache.put(3, 3);   // 去除键 2 ，因为 cnt(2)=1 ，使用计数最小
                      // cache=[3,1], cnt(3)=1, cnt(1)=2
lFUCache.get(2);      // 返回 -1（未找到）
lFUCache.get(3);      // 返回 3
                      // cache=[3,1], cnt(3)=2, cnt(1)=2
lFUCache.put(4, 4);   // 去除键 1 ，1 和 3 的 cnt 相同，但 1 最久未使用
                      // cache=[4,3], cnt(4)=1, cnt(3)=2
lFUCache.get(1);      // 返回 -1（未找到）
lFUCache.get(3);      // 返回 3
                      // cache=[3,4], cnt(4)=1, cnt(3)=3
lFUCache.get(4);      // 返回 4
                      // cache=[3,4], cnt(4)=2, cnt(3)=3
```

提示：

- 0 <= capacity, key, value <= 10^4
- 最多调用 10^5 次 `get` 和 `put` 方法

进阶：你可以为这两种操作设计时间复杂度为 $O(1)$ 的实现吗？

基本分析

前两天我们刚讲过 146. LRU 缓存机制 ，简单理解 LRU 就是「移除最久不被使用的元素」。

因此对于 LRU 我们只需要在使用「哈希表」的同时，维护一个「双向链表」即可：

- 每次发生 `get` 或 `put` 的时候就将元素存放双向链表头部

- 当需要移除元素时，则从双向链表尾部开始移除

LFU 简单理解则是指「移除使用次数最少的元素」，如果存在多个使用次数最小的元素，则移除「最近不被使用的那个」（LRU 规则）。同样的 `get` 和 `put` 都算作一次使用。

因此，我们需要记录下每个元素的使用次数，并且在 $O(1)$ 的复杂度内「修改某个元素的使用次数」和「找到使用次数最小的元素」。

桶排序 + 双向链表

我们可以使用「桶排序」的思路，搭配「双向链表」实现 $O(1)$ 操作。

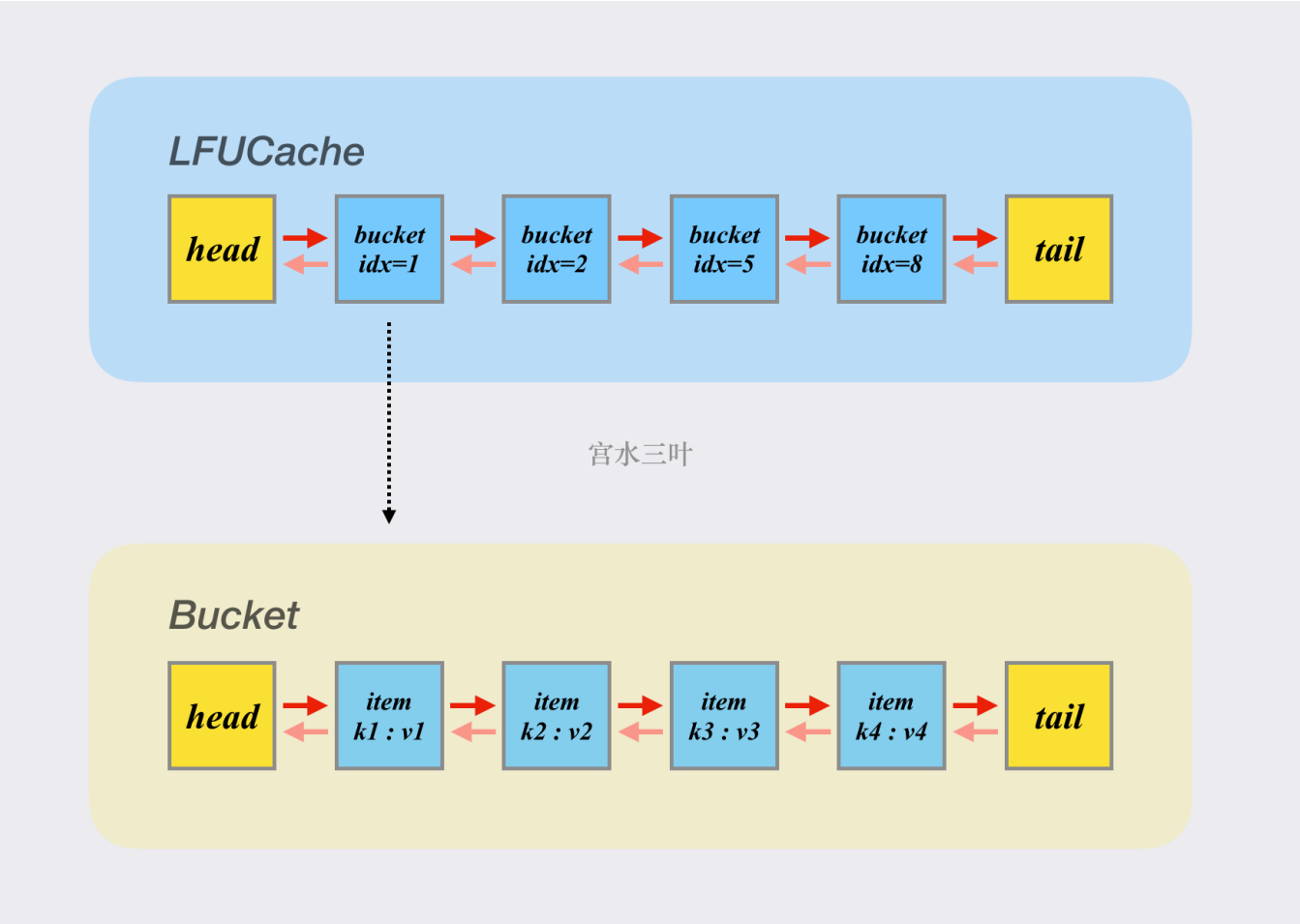
在 `LFUCache` 中，我们维护一个由 `Bucket` 作为节点的双向链表，每个 `Bucket` 都有一个 `idx` 编号，代表当前桶存放的是「使用了多少次」的键值对（ `idx = 1` 的桶存放使用一次的键值对；`idx = 2` 的桶存放的是使用两次的键值对 ... ）。

同时 `LFUCache` 持有一个「哈希表」，用来记录哪些 `key` 在哪个桶内。

在 `Bucket` 内部则是维护了一条以 `Item` 作为节点的双向链表，`Item` 是用作存放真实键值对的。

同样的，`Bucket` 也持有一个「哈希表」，用来记录 `key` 与 `Item` 的映射关系。

因此 `LFUCache` 其实是一个「链表套链表」的数据结构：



对应到 `LFUCache` 的几种操作：

- `get` ：先通过 `LFUCache` 持有的哈希表进行查找，如果不存在返回 -1，如果存在找到键值对所在的桶 `cur` ：
    - 调用对应的 `cur` 的 `remove` 操作，得到键值对对应的 `item`（移除代表当前键值对使用次数加一了，不会在存在于原来的桶中）。
    - 将 `item` 放到 `idx` 为 $cur.idx + 1$ 的桶 `target` 中（代表代表当前键值对使用次数加一，应该放到新的目标桶中）。
    - 如果目标桶 `target` 不存在，则创建；如果原来桶 `cur` 移除键值对后为空，则销毁。
    - 更新 `LFUCache` 中哈希表的信息。
- `put` ： 先通过 `LFUCache` 持有的哈希表进行查找：
    - 如果存在：找到键值对所在的桶 `cur` ，调用 `cur` 的 `put` 操作，更新键值对，然后调用 `LFUCache` 的 `get` 操作实现使用次数加一。
    - 如果不存在：先检查容量是否达到数量：
        - 容量达到数量的话需要调用「编号最小的桶」的 `clear` 操

作，在 `clear` 操作内部，会从 `item` 双向链表的尾部开始移除元素。完成后再执行插入操作。

- 插入操作：将键值对添加到 $idx = 1$ 的桶中（代表当前键值对使用次数为 1），如果桶不存在则创建。

代码：

```
class LFUCache {

    class Item {
        Item l, r;
        int k, v;
        public Item(int _k, int _v) {
            k = _k;
            v = _v;
        }
    }

    class Bucket {
        Bucket l, r;
        int idx;
        Item head, tail;
        Map<Integer, Item> map = new HashMap<>();
        public Bucket(int _idx) {
            idx = _idx;
            head = new Item(-1, -1);
            tail = new Item(-1, -1);
            head.r = tail;
            tail.l = head;
        }
        void put(int key, int value) {
            Item item = null;
            if (map.containsKey(key)) {
                item = map.get(key);
                // 更新值
                item.v = value;
                // 在原来的双向链表位置中移除
                item.l.r = item.r;
                item.r.l = item.l;
            } else {
                item = new Item(key, value);
                // 添加到哈希表中
                map.put(key, item);
            }
            // 增加到双向链表头部
            item.r = head.r;
            item.l = head;
            head.r.l = item;
            head.r = item;
        }
        Item remove(int key) {
            if (map.containsKey(key)) {
                Item item = map.get(key);
```

```
                // 从双向链表中移除
                item.l.r = item.r;
                item.r.l = item.l;
                // 从哈希表中移除
                map.remove(key);
                return item;
            }
            return null; // never
        }
        Item clear() {
            // 从双向链表尾部找到待删除的节点
            Item item = tail.l;
            item.l.r = item.r;
            item.r.l = item.l;
            // 从哈希表中移除
            map.remove(item.k);
            return item;
        }
        boolean isEmpty() {
            return map.size() == 0;
        }
    }

    Map<Integer, Bucket> map = new HashMap<>();
    Bucket head, tail;
    int n;
    int cnt;
    public LFUCache(int capacity) {
        n = capacity;
        cnt = 0;
        head = new Bucket(-1);
        tail = new Bucket(-1);
        head.r = tail;
        tail.l = head;
    }

    public int get(int key) {
        if (map.containsKey(key)) {
            Bucket cur = map.get(key);

            Bucket target = null;
            if (cur.r.idx != cur.idx + 1) {
                // 目标桶空缺
                target = new Bucket(cur.idx + 1);
                target.r = cur.r;
                target.l = cur;
```

```
                cur.r.l = target;
                cur.r = target;
            } else {
                target = cur.r;
            }

            // 将当前键值对从当前桶移除，并加入新的桶
            Item remove = cur.remove(key);
            target.put(remove.k, remove.v);
            // 更新当前键值对所在桶信息
            map.put(key, target);

            // 如果在移除掉当前键值对后，当前桶为空，则将当前桶删除（确保空间是 O(n) 的）
            // 也确保调用编号最小的桶的 clear 方法，能够有效移除掉一个元素
            deleteIfEmpty(cur);

            return remove.v;
        }
        return -1;
    }

    public void put(int key, int value) {
        if (n == 0) return;
        if (map.containsKey(key)) {
            // 元素已存在，修改一下值
            Bucket cur = map.get(key);
            cur.put(key, value);
            // 调用一下 get 实现「使用次数」+ 1
            get(key);
        } else {
            // 容器已满，需要先删除元素
            if (cnt == n) {
                // 从第一个桶（编号最小、使用次数最小）中进行清除
                Bucket cur = head.r;
                Item clear = cur.clear();
                map.remove(clear.k);
                cnt--;

                // 如果在移除掉键值对后，当前桶为空，则将当前桶删除（确保空间是 O(n) 的）
                // 也确保调用编号最小的桶的 clear 方法，能够有效移除掉一个元素
                deleteIfEmpty(cur);
            }

            // 需要将当前键值对增加到 1 号桶
            Bucket first = null;
```

```
            // 如果 1 号桶不存在则创建
            if (head.r.idx != 1) {
                first = new Bucket(1);
                first.r = head.r;
                first.l = head;
                head.r.l = first;
                head.r = first;
            } else {
                first = head.r;
            }

            // 将键值对添加到 1 号桶
            first.put(key, value);
            // 更新键值对所在桶信息
            map.put(key, first);
            // 计数器加一
            cnt++;
        }
    }

    void deleteIfEmpty(Bucket cur) {
        if (cur.isEmpty()) {
            cur.l.r = cur.r;
            cur.r.l = cur.l;
            cur = null; // help GC
        }
    }
}
```

- 时间复杂度：各操作均为 $O(1)$
- 时间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **725. 分隔链表** ，难度为 **中等**。

Tag：「模拟」、「链表」

给你一个头结点为 `head` 的单链表和一个整数 `k` ，请你设计一个算法将链表分隔为 `k` 个连续

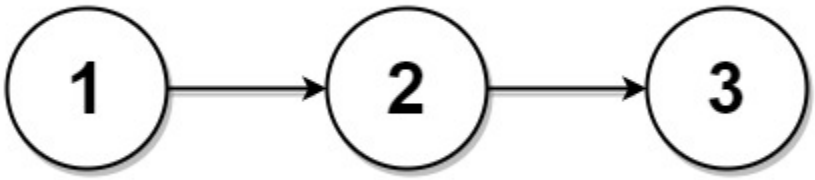
的部分。

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 。这可能会导致有些部分为 null 。

这 `k` 个部分应该按照在链表中出现的顺序排列，并且排在前面的部分的长度应该大于或等于排在后面的长度。

返回一个由上述 k 部分组成的数组。

示例 1：

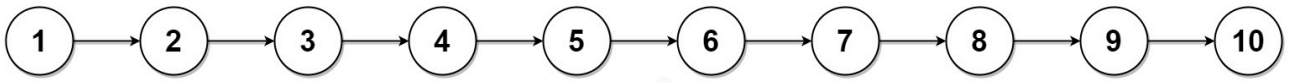



```
输入：head = [1,2,3], k = 5

输出：[[1],[2],[3],[],[]]

解释：
第一个元素 output[0] 为 output[0].val = 1 ，output[0].next = null 。
最后一个元素 output[4] 为 null ，但它作为 ListNode 的字符串表示是 [] 。
```

示例 2：




```
输入：head = [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。
```

提示：

- 链表中节点的数目在范围 [0, 1000]
- 0 <= Node.val <= 1000
- 1 <= k <= 50

模拟

根据题意，我们应当近可能将链表平均分为 k 份。

我们可以采取与 (题解) 68. 文本左右对齐 类似的思路（在 68 中，填充空格的操作与本题一致：尽可能平均，无法均分时，应当使前面比后面多）。

回到本题，我们可以先对链表进行一次扫描，得到总长度 cnt，再结合需要将将链表划分为 k 份，可知每一份的 **最小** 分配单位 $per = \lfloor \frac{cnt}{k} \rfloor$（当 $cnt < k$ 时，per 为 0）。

然后从前往后切割出 k 份链表，由于是在原链表的基础上进行，因此这里的切分只需要在合适的位置将节点的 $next$ 指针置空即可。

当我们需要构造出 $ans[i]$ 的链表长度时，首先可以先分配 per 的长度，如果 `已处理的链表长度 + 剩余待分配份数 * per < cnt`，说明后面「待分配的份数」如果按照每份链表分配 per 长度的话，会有节点剩余，基于「不能均分时，前面的应当比后面长」原则，此时只需为当前 $ans[i]$ 多分一个单位长度即可。

代码：

```java
class Solution {
    public ListNode[] splitListToParts(ListNode head, int k) {
        // 扫描链表，得到总长度 cnt
        int cnt = 0;
        ListNode tmp = head;
        while (tmp != null && ++cnt > 0) tmp = tmp.next;
        // 理论最小分割长度
        int per = cnt / k;
        // 将链表分割为 k 份（sum 代表已经被处理的链表长度为多少）
        ListNode[] ans = new ListNode[k];
        for (int i = 0, sum = 1; i < k; i++, sum++) {
            ans[i] = head;
            tmp = ans[i];
            // 每次首先分配 per 的长度
            int u = per;
            while (u-- > 1 && ++sum > 0) tmp = tmp.next;
            // 当「已处理的链表长度 + 剩余待分配份数 * per < cnt」，再分配一个单位长度
            int remain = k - i - 1;
            if (per != 0 && sum + per * remain < cnt && ++sum > 0) tmp = tmp.next;
            head = tmp != null ? tmp.next : null;
            if (tmp != null) tmp.next = null;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1600. 皇位继承顺序** ，难度为 **中等**。

Tag：「单链表」、「哈希表」

一个王国里住着国王、他的孩子们、他的孙子们等等。每一个时间点，这个家庭里有人出生也有人死亡。

这个王国有一个明确规定的皇位继承顺序，第一继承人总是国王自己。我们定义递归函

数 Successor(x, curOrder) ，给定一个人 x 和当前的继承顺序，该函数返回 x 的下一继承人。

```
Successor(x, curOrder):
    如果 x 没有孩子或者所有 x 的孩子都在 curOrder 中：
        如果 x 是国王，那么返回 null
        否则，返回 Successor(x 的父亲, curOrder)
    否则，返回 x 不在 curOrder 中最年长的孩子
```

比方说，假设王国由国王，他的孩子 Alice 和 Bob （Alice 比 Bob 年长）和 Alice 的孩子 Jack 组成。

1. 一开始， curOrder 为 [“king”].
2. 调用 Successor(king, curOrder) ，返回 Alice ，所以我们将 Alice 放入 curOrder 中，得到 [“king”, “Alice”] 。
3. 调用 Successor(Alice, curOrder) ，返回 Jack ，所以我们将 Jack 放入 curOrder 中，得到 [“king”, “Alice”, “Jack”] 。
4. 调用 Successor(Jack, curOrder) ，返回 Bob ，所以我们将 Bob 放入 curOrder 中，得到 [“king”, “Alice”, “Jack”, “Bob”] 。
5. 调用 Successor(Bob, curOrder) ，返回 null 。最终得到继承顺序为 [“king”, “Alice”, “Jack”, “Bob”] 。
   通过以上的函数，我们总是能得到一个唯一的继承顺序。

请你实现 ThroneInheritance 类：

- ThroneInheritance(string kingName) 初始化一个 ThroneInheritance 类的对象。国王的名字作为构造函数的参数传入。
- void birth(string parentName, string childName) 表示 parentName 新拥有了一个名为 childName 的孩子。
- void death(string name) 表示名为 name 的人死亡。一个人的死亡不会影响 Successor 函数，也不会影响当前的继承顺序。你可以只将这个人标记为死亡状态。
- string[] getInheritanceOrder() 返回 除去 死亡人员的当前继承顺序列表。

示例：

```
输入：
["ThroneInheritance", "birth", "birth", "birth", "birth", "birth", "birth", "getInheritanceOrder", "death"
[["king"], ["king", "andy"], ["king", "bob"], ["king", "catherine"], ["andy", "matthew"], ["bob", "alex"],

输出：
[null, null, null, null, null, null, null, ["king", "andy", "matthew", "bob", "alex", "asha", "catherine"]

解释：
ThroneInheritance t= new ThroneInheritance("king"); // 继承顺序：king
t.birth("king", "andy"); // 继承顺序：king > andy
t.birth("king", "bob"); // 继承顺序：king > andy > bob
t.birth("king", "catherine"); // 继承顺序：king > andy > bob > catherine
t.birth("andy", "matthew"); // 继承顺序：king > andy > matthew > bob > catherine
t.birth("bob", "alex"); // 继承顺序：king > andy > matthew > bob > alex > catherine
t.birth("bob", "asha"); // 继承顺序：king > andy > matthew > bob > alex > asha > catherine
t.getInheritanceOrder(); // 返回 ["king", "andy", "matthew", "bob", "alex", "asha", "catherine"]
t.death("bob"); // 继承顺序：king > andy > matthew > bob（已经去世）> alex > asha > catherine
t.getInheritanceOrder(); // 返回 ["king", "andy", "matthew", "alex", "asha", "catherine"]
```

提示：

- 1 <= kingName.length, parentName.length, childName.length, name.length <= 15
- kingName，parentName， childName 和 name 仅包含小写英文字母。
- 所有的参数 childName 和 kingName 互不相同。
- 所有 death 函数中的死亡名字 name 要么是国王，要么是已经出生了的人员名字。
- 每次调用 birth(parentName, childName) 时，测试用例都保证 parentName 对应的人员是活着的。
- 最多调用 10^5 次birth 和 death 。
- 最多调用 10 次 getInheritanceOrder 。

单向链表 & 标记删除

根据题意，我们需要将「新儿子」插入到「父亲」的「最后一个儿子」的「儿子们」的后面（**注意这是个递归过程**）；如果该「父亲」还没有任何儿子，则直接插到「父亲」后面。

因此，我们需要在节点 `Node` 中使用一个 `last` 记录该节点的「最后一个儿子」，同时因为删除的时候，我们无法在 $O(1)$ 的复杂度内更新 `last` 信息，所以只能使用「标记删除」的方式。

代码：

```java
class ThroneInheritance {
    class Node {
        String name;
        Node next;
        Node last; // 记录最后一个儿子
        boolean isDeleted = false;
        Node (String _name) {
            name = _name;
        }
    }
    Map<String, Node> map = new HashMap<>();
    Node head = new Node(""), tail = new Node("");
    public ThroneInheritance(String name) {
        Node root = new Node(name);
        root.next = tail;
        head.next = root;
        map.put(name, root);
    }

    public void birth(String pname, String cname) {
        Node node = new Node(cname);
        map.put(cname, node);
        Node p = map.get(pname);
        Node tmp = p;
        while (tmp.last != null) tmp = tmp.last;
        node.next = tmp.next;
        tmp.next = node;
        p.last = node;
    }

    public void death(String name) {
        Node node = map.get(name);
        node.isDeleted = true;
    }

    public List<String> getInheritanceOrder() {
        List<String> ans = new ArrayList<>();
        Node tmp = head.next;
        while (tmp.next != null) {
            if (!tmp.isDeleted) ans.add(tmp.name);
            tmp = tmp.next;
        }
        return ans;
    }
}
```

- 时间复杂度：`birth` 和 `getInheritanceOrder` 操作为 $O(n)$；其余操作为 $O(1)$
- 时间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「链表」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码