题目地址(160. 相交链表)

https://leetcode-cn.com/problems/intersection-of-two-linked-lists/

入选理由

1. 讲义里面的题目，不应该不会
2. 考察频率相当的高

标签

- 双指针
- 链表

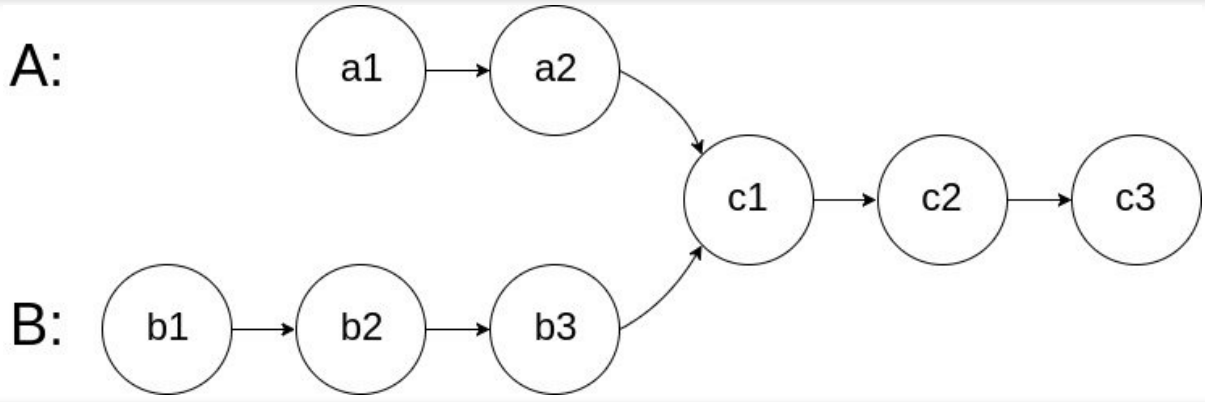
难度

- 简单

题目描述

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` 。

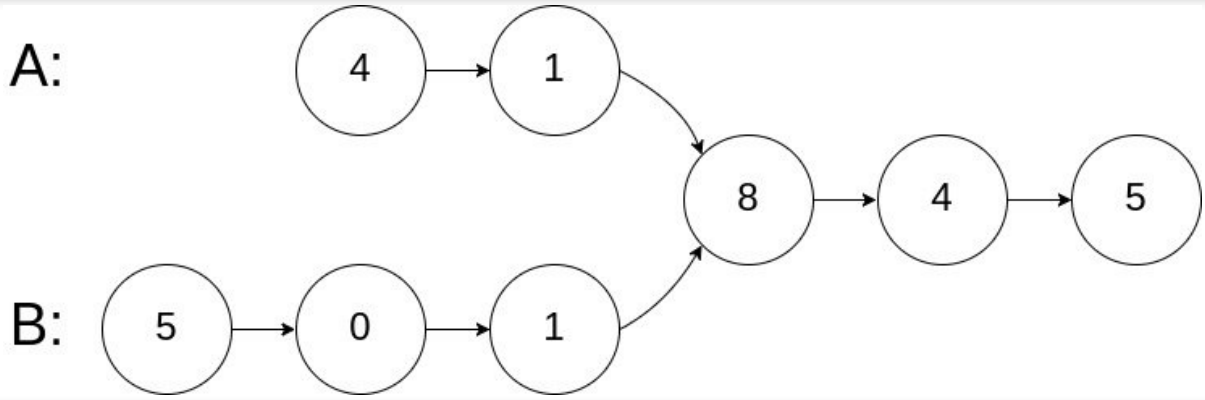
图示两个链表在节点 `c1` 开始相交：



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：

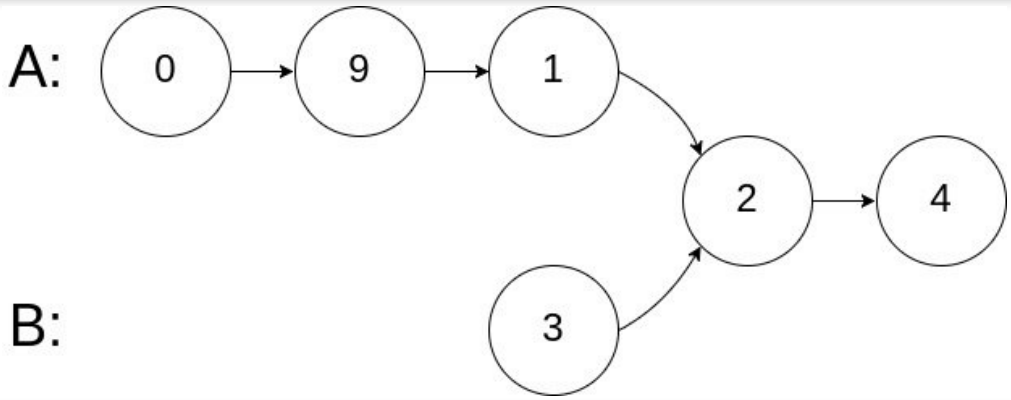


输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

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：

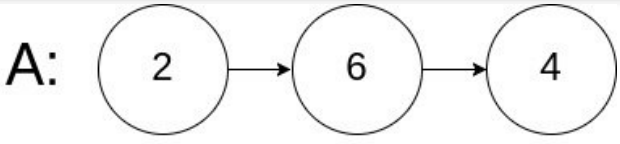


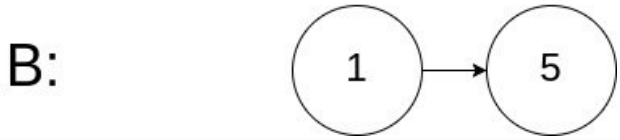
输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

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 * 104
1 <= Node.val <= 105
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？

前置知识

- 链表
- 双指针

哈希法

思路

- 有 A, B 两条链表, 先遍历其中一个，比如 A 链表，并将 A 中的所有节点存入哈希表。
- 接着遍历 B 链表，检查每个节点是否在哈希表中, 存在于哈希表中的那个节点就是 A 链表和 B 链表的相交节点。

伪代码：

```
data = new Set() // 存放A链表的所有节点的地址

while A不为空{
```

```
  哈希表中添加A链表当前节点
  A指针向后移动
}

while B不为空{
  if 如果哈希表中含有B链表当前节点
    return B
  B指针向后移动
}

return null // 两条链表没有相交点
```

代码(JS/C++)

JS Code:

```
let data = new Set();
while (A !== null) {
  data.add(A);
  A = A.next;
}
while (B !== null) {
  if (data.has(B)) return B;
  B = B.next;
}
return null;
```

C++ Code:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (headA == NULL || headB == NULL) return NULL;

    map<ListNode*, bool> seen;
    while (headA) {
        seen.insert(pair<ListNode*, bool>(headA, true));
        headA = headA->next;
    }
    while (headB) {
        if (seen.find(headB) != seen.end()) return headB;
        headB = headB->next;
    }
    return NULL;
}
```

复杂度分析

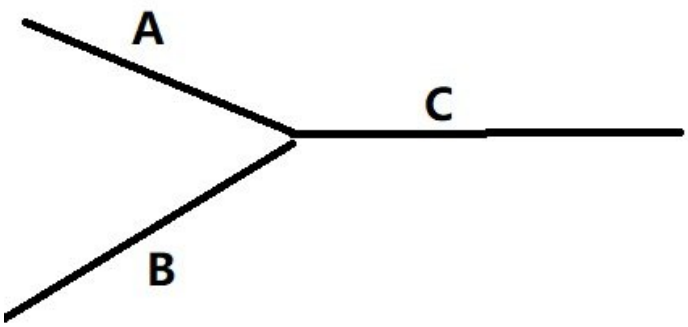
- 时间复杂度：$O(N)$

- 空间复杂度：$O(N)$

双指针

思路

- 使用两个指针如指针 a, b 分别指向 A, B 这两条链表的头节点, 两个指针以相同的速度向后移动。
- 当 a 到达链表 A 的尾部时，将它重定位到链表 B 的头节点；
- 当 b 到达链表 B 的尾部时，将它重定位到链表 A 的头节点；
- 若在此过程中 a, b 指针相遇，则相遇节点为两链表相交的起始节点，否则说明两个链表不存在相交点。



（图 5）

为什么 a, b 指针相遇的点一定是相交的起始节点？我们证明一下：

1. 将两条链表按相交的起始节点继续截断，链表 1 为：`A + C`，链表 2 为：`B + C`；
2. 当 a 指针将链表 1 遍历完后，重定位到链表 2 的头节点，然后继续遍历直至相交点，此时 a 指针遍历的距离为 `A + C + B`；
3. 同理 b 指针遍历的距离为 `B + C + A`；

伪代码：

```
a = headA
b = headB
while a,b指针不相等时 {
    if a指针为空时
      a指针重定位到链表 B的头结点
    else
      a指针向后移动一位
    if b指针为空时
      b指针重定位到链表 A的头结点
    else
      b指针向后移动一位
```

```
}
return a
```

代码(JS/Python/C++)

JS Code:

```
var getIntersectionNode = function (headA, headB) {
  let a = headA,
    b = headB;
  while (a != b) {
    a = a === null ? headB : a.next;
    b = b === null ? headA : b.next;
  }
  return a;
};
```

Python Code:

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        a, b = headA, headB
        while a != b:
            a = a.next if a else headB
            b = b.next if b else headA
        return a
```

C++ Code:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (headA == NULL || headB == NULL) return NULL;

    ListNode* pA = headA;
    ListNode* pB = headB;
    while (pA != pB) {
        pA = pA == NULL ? headB : pA->next;
        pB = pB == NULL ? headA : pB->next;
    }

    return pA;
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

上一页 下一页