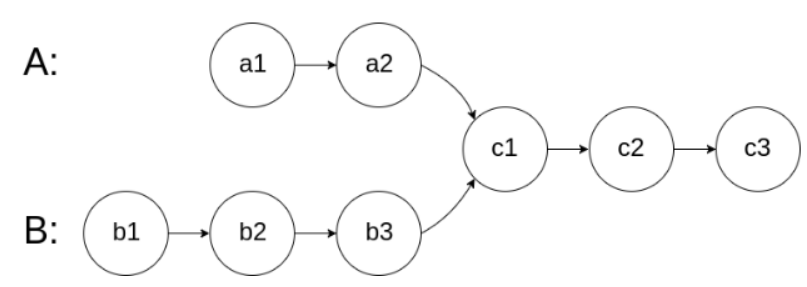
<https://leetcode.com/problems/intersection-of-two-linked-lists/>

Given the heads of two singly linked-lists headA and headB, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

**Note** that the linked lists must **retain their original structure** after the function returns.

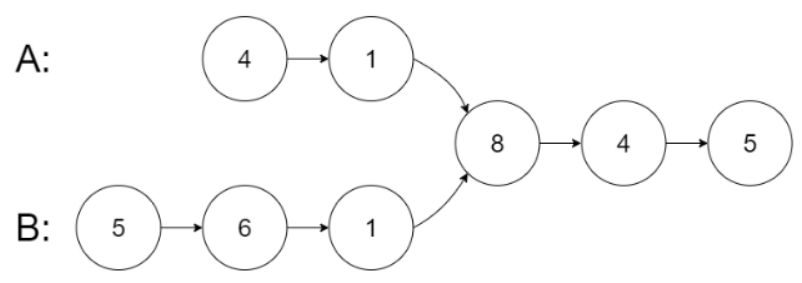
**Custom Judge:**

The inputs to the **judge** are given as follows (your program is **not** given these inputs):

* intersectVal - The value of the node where the intersection occurs. This is 0 if there is no intersected node.
* listA - The first linked list.
* listB - The second linked list.
* skipA - The number of nodes to skip ahead in listA (starting from the head) to get to the intersected node.
* skipB - The number of nodes to skip ahead in listB (starting from the head) to get to the intersected node.

The judge will then create the linked structure based on these inputs and pass the two heads, headA and headB to your program. If you correctly return the intersected node, then your solution will be **accepted**.

**Example 1:**



Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

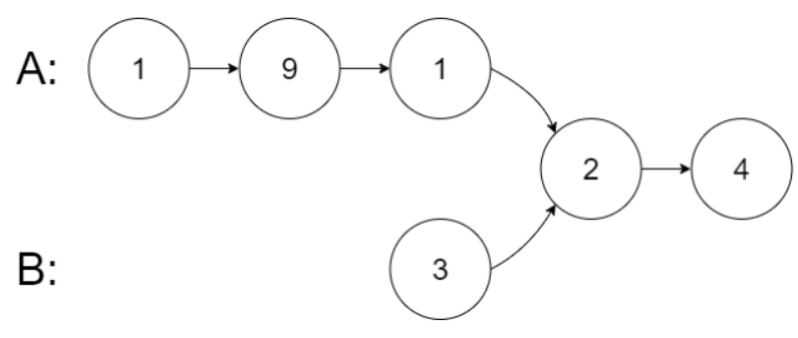
Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

- Note that the intersected node's value is not 1 because the nodes with value 1 in A and B (2nd node in A and 3rd node in B) are different node references. In other words, they point to two different locations in memory, while the nodes with value 8 in A and B (3rd node in A and 4th node in B) point to the same location in memory.

**Example 2:**



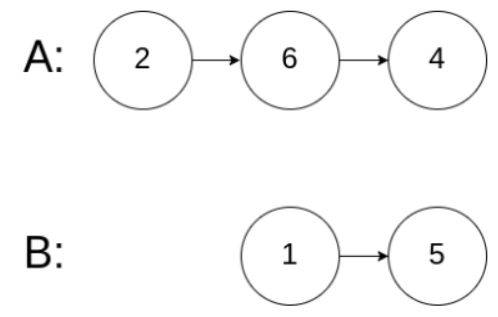
Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

**Example 3:**



Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

Output: No intersection

Explanation: From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

Explanation: The two lists do not intersect, so return null.

**Constraints:**

* The number of nodes of listA is in the m.
* The number of nodes of listB is in the n.
* 1 <= m, n <= 3 \* 104
* 1 <= Node.val <= 105
* 0 <= skipA < m
* 0 <= skipB < n
* intersectVal is 0 if listA and listB do not intersect.
* intersectVal == listA[skipA] == listB[skipB] if listA and listB intersect.

**Follow up:** Could you write a solution that runs inO(m + n)time and use onlyO(1)memory?

**Attempt 1: 2023-02-21**

**Solution 1:  No check on length but continue on the other linked list when finish on itself (30 min)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

ListNode iter1 = headA;

ListNode iter2 = headB;

while(iter1 != iter2) {

if(iter1 == null) {

iter1 = headB;

} else {

iter1 = iter1.next;

}

if(iter2 == null) {

iter2 = headA;

} else {

iter2 = iter2.next;

}

}

return iter1;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/intersection-of-two-linked-lists/solutions/49785/java-solution-without-knowing-the-difference-in-len/>

I found most solutions here preprocess linkedlists to get the difference in len. Actually we don't care about the "value" of difference, we just want to make sure two pointers reach the intersection node at the same time.

We can use two iterations to do that. In the first iteration, we will reset the pointer of one linkedlist to the head of another linkedlist after it reaches the tail node. In the second iteration, we will move two pointers until they points to the same node. Our operations in first iteration will help us counteract the difference. So if two linkedlist intersects, the meeting point in second iteration must be the intersection point. If the two linked lists have no intersection at all, then the meeting pointer in second iteration must be the tail node of both lists, which is null

Below is my commented Java code:

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

//boundary check

if(headA == null || headB == null) return null;

ListNode a = headA;

ListNode b = headB;

//if a & b have different len, then we will stop the loop after second iteration

while( a != b){

//for the end of first iteration, we just reset the pointer to the head of another linkedlist

a = a == null? headB : a.next;

b = b == null? headA : b.next;

}

return a;

}

**Visualization of this solution:**

**Case 1 (Have Intersection & Same Len):**

a

A: a1 → a2 → a3

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

a

A: a1 → a2 → a3

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

a

A: a1 → a2 → a3

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

A: a1 → a2 → a3

↘ a

c1 → c2 → c3 → null

↗ b

B: b1 → b2 → b3

Since a == b is true, end loop while(a != b), return the intersection node a = c1.

**Case 2 (Have Intersection & Different Len):**

a

A: a1 → a2

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

a

A: a1 → a2

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

A: a1 → a2

↘ a

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

b

A: a1 → a2

↘ a

c1 → c2 → c3 → null

↗ b

B: b1 → b2 → b3

A: a1 → a2

↘ a

c1 → c2 → c3 → null

↗ b

B: b1 → b2 → b3

A: a1 → a2

↘ a = null, then a = b1

c1 → c2 → c3 → null

↗ b

B: b1 → b2 → b3

A: a1 → a2

↘

c1 → c2 → c3 → null

↗ b = null, then b = a1

B: b1 → b2 → b3

a

b

A: a1 → a2

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

a

b

A: a1 → a2

↘

c1 → c2 → c3 → null

↗

B: b1 → b2 → b3

a

A: a1 → a2

↘ b

c1 → c2 → c3 → null

↗ a

B: b1 → b2 → b3

Since a == b is true, end loop while(a != b), return the intersection node a = c1.

**Case 3 (Have No Intersection & Same Len):**

a

A: a1 → a2 → a3 → null

B: b1 → b2 → b3 → null

b

a

A: a1 → a2 → a3 → null

B: b1 → b2 → b3 → null

b

a

A: a1 → a2 → a3 → null

B: b1 → b2 → b3 → null

b

a = null

A: a1 → a2 → a3 → null

B: b1 → b2 → b3 → null

b = null

Since a == b is true (both refer to null), end loop while(a != b), return a = null.

**Case 4 (Have No Intersection & Different Len):**

a

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

b

a

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

b

a

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

b

a

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

b = null, then b = a1

b a = null, then a = b1

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

b

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

a

b

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

a

b

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

a

b = null

A: a1 → a2 → a3 → a4 → null

B: b1 → b2 → b3 → null

a = null

Since a == b is true (both refer to null), end loop while(a != b), return a = null.

Notice that if list A and list B have the **same length**, this solution will terminate in **no more than 1 traversal**; if both lists have **different lengths**, this solution will terminate in **no more than 2 traversals** -- in the second traversal, swapping a and b synchronizes a and b before the end of the second traversal. By synchronizing a and b I mean both have the same remaining steps in the second traversal so that it's guaranteed for them to reach the first intersection node, or reach null at the same time (technically speaking, in the same iteration) -- see **Case 2 (Have Intersection & Different Len)** and **Case 4 (Have No Intersection & Different Len)**.