160. Intersection of Two Linked Lists

Two Pointers

Linked List

Problem Description

Hash Table

Easy

the lists must remain unchanged after the function executes. The <u>linked list</u> structure in question is a common data structure where each element (a node) contains data (val) and a reference (the next pointer) to the next node in the sequence. The head of a linked list refers to its first node.

The problem is about finding a common intersection node (if one exists) for two singly linked lists. These lists may completely

diverge, in which case no intersection occurs, or they can share a common sequence of nodes. The point at which they start to

intersect becomes their intersection node. This problem does not consider linked lists with cycles and the structural integrity of

We are given two heads of two singly linked lists (headA and headB), and we need to find the node from where they start

intersecting. Intuition

The challenge is to compare two singly linked lists of potentially different lengths and identify if and where they intersect. To find the intersection without extra memory for data structures like hash sets or arrays (for storing visited nodes), we can use the twopointer technique.

Intuitively, if you traverse both lists in tandem, they will reach the intersection point at the same time if they are of the same length. However, when they're of different lengths, simply traversing them simultaneously won't work because one pointer will

reach the end before the other. The key insight here is to switch lists when one pointer reaches the end. This way, both pointers end up covering the extra distance on the longer list and hence reconcile any difference in lengths. Concretely, the solution involves two pointers, starting at headA and headB respectively. They each move forward one node at a time. When one pointer reaches the end of its list, it continues from the beginning of the other list. If there is an intersection, the

pointers will meet at the intersection node after traversing lengthA + lengthB nodes (where lengthA is the length of list A

including the intersection, and lengthB is the length of list B including the intersection). If there is no intersection, both pointers

will eventually be null after the same number of steps, effectively also reaching the end of a hypothetically combined list.

The provided Python code prescribes this approach. It uses a simple while-loop that continues until the two pointers are equal (either at the intersection node or null). It's a concise and efficient solution as it only traverses each list once, with a time complexity of O(N + M), where N and M are the lengths of the two lists. Solution Approach

The implementation of the solution makes use of the two-pointer technique. The algorithm involves keeping two pointers, a and b, initially set to headA and headB respectively. Both a and b traverse the respective linked lists simultaneously. If any of the pointers reaches the end of its linked list, it is reassigned to the head of the other linked list. This swapping of starting points

after reaching the end ensures that the two pointers will eventually traverse equal distances even if the linked lists have different

The pseudocode of the solution is as follows: 1. Initialize two pointers, a and b, to headA and headB respectively.

2. While a is not equal to b:

lengths.

∘ If a has not reached the end of its list, move to the next node (a = a.next), otherwise, start from the beginning of headB (a = headB). ∘ If b has not reached the end of its list, move to the next node (b = b.next), otherwise, start from the beginning of headA (b = headA). 3. The loop exits when either both a and b are null (no intersection), or a and b meet at the intersection node. In the actual implementation, here's what the critical part of the code looks like: while a != b: a = a.next if a else headB

return a

b = b.next if b else headA

```
the pointer starts at the head of the opposite linked list upon reaching the end of its current list. If the lists do not intersect, both
pointers will become null simultaneously after traversing both lists in their entirety, thus breaking the loop and returning null.
The beauty of this approach is its simplicity and the fact that it uses O(1) extra space, which is an improvement over methods
that would require additional data structures to mark visited nodes.
```

This solution has a linear time complexity of O(N+M), where N is the length of linked list headA and M is the length of linked list

headB. This is because, in the worst case, each pointer traverses both lists completely. The space complexity is O(1), as the

This loop will keep running until a and b point to the same node. The if a else headB and if b else headA logic ensures that

• List A: 1 -> 2 -> 3 -> 4 -> 5 • List B: 9 -> 4 -> 5

Here, nodes with values 4 and 5 are shared among both lists, making node 4 the intersection point.

Now, we will walk through the two-pointer strategy using this example. Initialize two pointers: Pointer a at the head of List A (1)

Move a to 3, move b to 5.

Move b to 2.

Example Walkthrough

 Move a to 4, and since pointer b has no next node, it moves to the head of List A (1). Keep proceeding with the traversal:

Since pointer a has no next node, it moves to the head of List B (9).

algorithm only uses two pointers irrespective of the size of the input lists.

Let's assume we have two singly linked lists that intersect at some node:

Pointer a moves to 4, pointer b moves to 3.

Continue the process:

Pointer b at the head of List B (9)

Move a to 2, move b to 4.

Continue moving forward:

Move each pointer forward one node at a time:

 Pointer b moves to 4 in List A (since b was previously at 3). The pointers now meet at the node with value 4, which is the intersection node.

At this point, the crucial step occurs:

Pointer a is now at the node with the value 4 on List B.

will be the node with the value 4 as it's the point of intersection.

 Pointer a: 1 -> 2 -> 3 -> 4 -> 1 -> 9 -> 4 Pointer b: 9 -> 4 -> 5 -> 1 -> 2 -> 3 -> 4

Definition for singly-linked list.

def init (self. val=0):

self.val = val

self.next = None

Python

class ListNode:

class Solution:

Java

class ListNode {

In this example, each pointer traversed:

Solution Implementation

def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:

Otherwise, move to the next node.

Otherwise, move to the next node.

pointerA = pointerA.next if pointerA else headB

pointerB = pointerB.next if pointerB else headA

* Find the intersection node of two singly-linked lists.

pointerA = pointerA ? pointerA->next : headB:

pointerB = pointerB ? pointerB->next : headA;

* Function to find the intersection of two singly-linked lists.

* @returns The intersection node or null if there is no intersection.

function getIntersectionNode(headA: ListNode | null, headB: ListNode | null): ListNode | null {

// Traverse both lists until the pointers meet or both reach the end (null)

* @param headA - The head of the first singly-linked list.

* @param headB - The head of the second singly-linked list.

return pointerA;

// Pointers to traverse both lists

while (pointerA !== pointerB) {

let pointerA: ListNode | null = headA;

let pointerB: ListNode | null = headB;

* @param headA the head of the first linked list

* @param headB the head of the second linked list

If pointerA reaches the end, redirect it to the head of list B.

If pointerB reaches the end, redirect it to the head of list A.

* An intersection node is defined as the node at which the two lists intersect.

* @return the intersection node or null if there is no intersection

Return either the intersection node or None if the two lists do not intersect.

pointer cover the length of both lists, regardless of their initial lengths.

Initialize two pointers, one for each linked list. pointerA, pointerB = headA, headB # Continue iterating until the two pointers meet or both reach the end. while pointerA != pointerB:

The loop has exited because a is equal to b, signifying that the intersection node has been found. The output of our algorithm

This shows that after pointer b traversed its own list, and part of List A, and pointer a traversed its own list, and part of List B,

they met at the intersection node. Thus, they have traversed equal lengths, which is the basis of our algorithm – having each

```
int value; // Node value
ListNode next; // Reference to the next node in the list
ListNode(int x) {
```

value = x:

public class Solution {

/**

*/

next = null;

* Definition for singly-linked list.

return pointerA

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
       ListNode pointerA = headA; // Pointer for traversing the first list
        ListNode pointerB = headB; // Pointer for traversing the second list
       // Continue traversing until the two pointers are the same (either at intersection or both null)
        while (pointerA != pointerB) {
            // If we have reached the end of list A, switch to list B; otherwise, move to the next node
            pointerA = (pointerA == null) ? headB : pointerA.next;
            // If we have reached the end of list B. switch to list A; otherwise, move to the next node
            pointerB = (pointerB == null) ? headA : pointerB.next;
        // At the end of the traversal, pointerA and pointerB will be either at the intersection node or null
        return pointerA; // return the intersection node or null
C++
// Definition for singly-linked list.
struct ListNode {
    int value:
    ListNode *next;
    ListNode(int x) : value(x), next(nullptr) {}
};
class Solution {
public:
    // Function to get the node at which the intersection of two singly linked lists begins.
    ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {
        ListNode *pointerA = headA; // Initialize a pointer for list A
        ListNode *pointerB = headB; // Initialize a pointer for list B
        // Iterate through both lists until the two pointers meet, which means an intersection was found.
        while (pointerA != pointerB) {
           // If at the end of one list, move the pointer to the beginning of the other list.
           // This ensures that each pointer traverses both lists.
           // If at any point they meet, that's the intersection node.
```

// At the end of the while loop, either the intersection node or nullptr (if no intersection) is returned.

};

/**

*/

TypeScript

```
// If pointerA reaches the end of list A, redirect it to the head of list B
       // This allows it to "catch up" in length to the other list on the second pass
       pointerA = pointerA ? pointerA.next : headB;
       // Similarly, if pointerB reaches the end of list B, redirect it to the head of list A
       pointerB = pointerB ? pointerB.next : headA;
   // Once both pointers meet, they are either at the intersection node or at the end (null)
   // In either case, return the pointer as the result.
   return pointerA;
# Definition for singly-linked list.
class ListNode:
   def init (self, val=0):
       self.val = val
       self.next = None
class Solution:
   def getIntersectionNode(self. headA: ListNode. headB: ListNode) -> ListNode:
       # Initialize two pointers, one for each linked list.
       pointerA, pointerB = headA, headB
       # Continue iterating until the two pointers meet or both reach the end.
       while pointerA != pointerB:
           # If pointerA reaches the end, redirect it to the head of list B.
           # Otherwise, move to the next node.
           pointerA = pointerA.next if pointerA else headB
           # If pointerB reaches the end, redirect it to the head of list A.
           # Otherwise, move to the next node.
           pointerB = pointerB.next if pointerB else headA
       # Return either the intersection node or None if the two lists do not intersect.
       return pointerA
```

Time Complexity The time complexity of the provided code is O(N + M), where N is the length of headA and M is the length of headB. This is

because in the worst-case scenario, the code will iterate through all nodes in both lists exactly once, before either finding the intersection or reaching the end of both lists simultaneously. **Space Complexity**

The space complexity of the provided code is 0(1). The algorithm only uses a fixed amount of additional space for two pointers a and b, regardless of the size of the linked lists. Therefore, the space used does not scale with the size of the input.

Time and Space Complexity