JavaScript Intersection of Two Linked Lists

Challenge

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`.

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

1st Example

```
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.
```

2nd Example

```
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.
             The two lists do not intersect, so return null.
```

3rd Example

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

Example continues on next page...

```
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**
>
> The intersected node's value is not `1` because the nodes with value `1` in `listA` and `listB` (2nd node in `listA` and 3rd node in `listB`) are different node references. In other words, they point to two different locations in memory, while the nodes with value `8` in `listA` and `listB` (3rd node in `listA` and 4th node in `listB`) point to the same location in memory.

Constraints

- `1 <= m, n <= 3 * 10^4`
- `1 <= Node.val <= 10^5`
- `0 <= skipA < m`
- `0 <= skipB < n`
- The number of nodes of `listA` is in the `m`.
- The number of nodes of `listB` is in the `n`.
- `intersectVal` is `0` if `listA` and `listB` do not intersect.
- `intersectVal == listA[skipA] == listB[skipB]` if `listA` and `listB` intersect.

Solution

```
const getIntersectionNode = (headA, headB) => {
    let currA = headA,
        currB = headB;

    while (currA !== currB) {
        if (!currA) {
            currA = headB;
        } else {
            currA = currA.next;
        }

        if (!currB) {
            currB = headA;
        } else {
            currB = currB.next;
        }
    }

    return currA;
};
```

Explanation

I've defined a function called `getIntersectionNode` that takes in two linked lists as parameters, `headA` and `headB`. The purpose of this function is to find the intersection node of the two linked lists and return it.

The function begins by initializing two variables, `currA` and `currB`, to the heads of the linked lists `headA` and `headB` respectively.

Next, it enters a while loop that continues until `currA` is equal to `currB`, indicating that the intersection node has been found.

Inside the while loop, it checks if `currA` is null, which indicates the end of linked list A. If it is null, it sets `currA` to the head of linked list B, as we need to continue traversing list B to find the intersection.

If `currA` is not null, it moves `currA` to the next node in linked list A by setting it to `currA.next`.

Similarly, it checks if `currB` is null, indicating the end of linked list B. If it is null, it sets `currB` to the head of linked list A, as we need to continue traversing list A to find the intersection.

If `currB` is not null, it moves `currB` to the next node in linked list B by setting it to `currB.next`.

Once the while loop breaks, indicating that `currA` is equal to `currB` and the intersection node has been found, the function returns `currA` which represents the intersection node.

In summary, this function iterates through both linked lists simultaneously, moving to the next node in each list until it finds the intersection node. The intersection node is determined when the two pointers, `currA` and `currB`, are equal.

Author: Trevor Morin