Find intersection of Two Linked Lists

Problem Statement: 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**.

Examples:

Example 1:

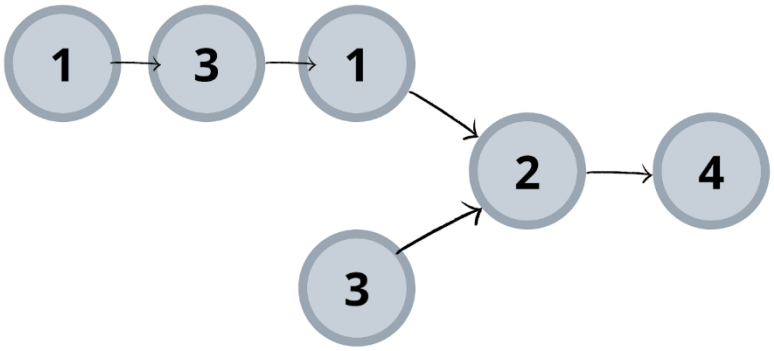
Input:

```
List 1 = [1,3,1,2,4], List 2 = [3,2,4]
```

Output:

```
2
```

Explanation: Here, both lists intersecting nodes start from node 2.



Example 2:

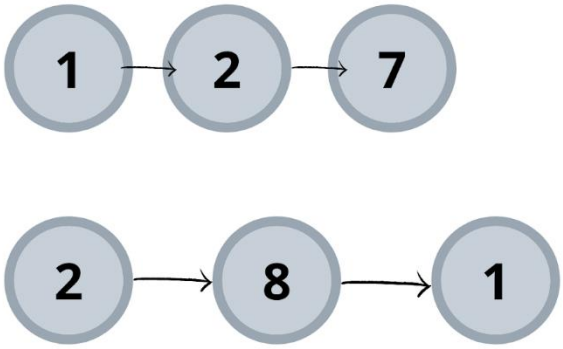
Input:

```
 List1 = [1,2,7], List 2 = [2,8,1]
```

Output:

```
 Null
```

Explanation: Here, both lists do not intersect and thus no intersection node is present.



Solution 1: Brute-Force

Approach: We know intersection means a common attribute present between two entities. Here, we have linked lists as given entities.

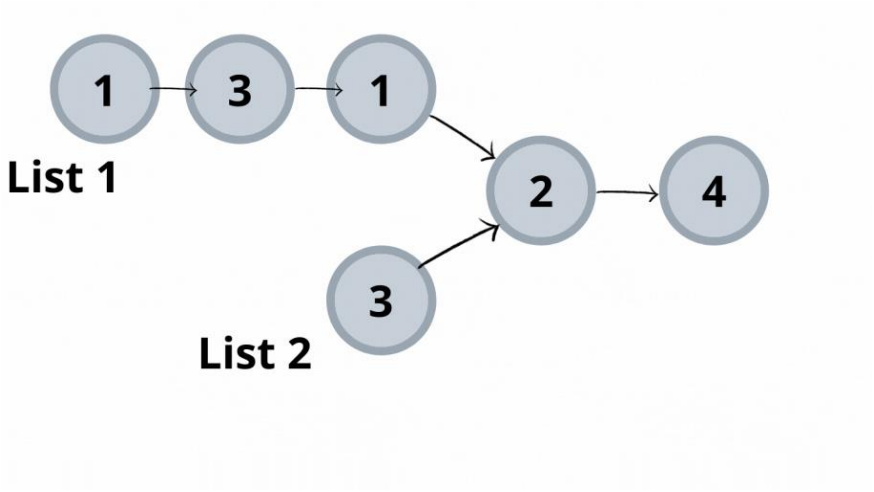
What should be the common attribute for two linked lists?

If you believe a common attribute is a node's value, then think properly! If we take our example 1, there we can see both lists have nodes of value 3. But it is not the first intersection node. So what's the common attribute?

It is the node itself that is the common attribute. So, the process is as follows:-

- Keep any one of the list to check its node present in the other list. Here, we are choosing the second list for this task.
- Iterate through the other list. Here, it is the first one.
- Check if the both nodes are the same. If yes, we got our first intersection node.
- If not, continue iteration.
- If we did not find an intersection node and completed the entire iteration of the second list, then there is no intersection between the provided lists. Hence, return *null*.

Dry Run:



Source Code:

```
static Node intersectionPresent(Node head1,Node head2) {
    while(head2 != null) {
        Node temp = head1;
        while(temp != null) {
            //if both nodes are same
            if(temp == head2) return head2;
            temp = temp.next;
        }
        head2 = head2.next;
    }
    //intersection is not present between the lists return null
    return null;
}
```

Time Complexity: O(m*n)

Reason: For each node in list 2 entire lists 1 are iterated.

Space Complexity: O(1)

Reason: No extra space is used.

Solution 2: Hashing

Approach:

Can we improve brute-force time complexity? In brute force, we are basically performing "searching". We can also perform searches by Hashing. Taking into consideration that hashing process takes O(1) time complexity. So the process is as follows:-

- Iterate through list 1 and hash its node address. Why? (Hint: depends on common attribute we are searching)
- Iterate through list 2 and search the hashed value in the hash table. If found, return node.

Code:

```
static Node intersectionPresent(Node head1,Node head2) {
     HashSet<Node> st=new HashSet<>();
    while(head1 != null) {
       st.add(head1);
       head1 = head1.next;
    }
    while(head2 != null) {
        if(st.contains(head2)) return head2;
        head2 = head2.next;
    }
    return null;
}
```

Time Complexity: O(n+m)

Reason: Iterating through list 1 first takes O(n), then iterating through list 2 takes O(m).

Space Complexity: O(n)

Reason: Storing list 1 node addresses in HashSet.

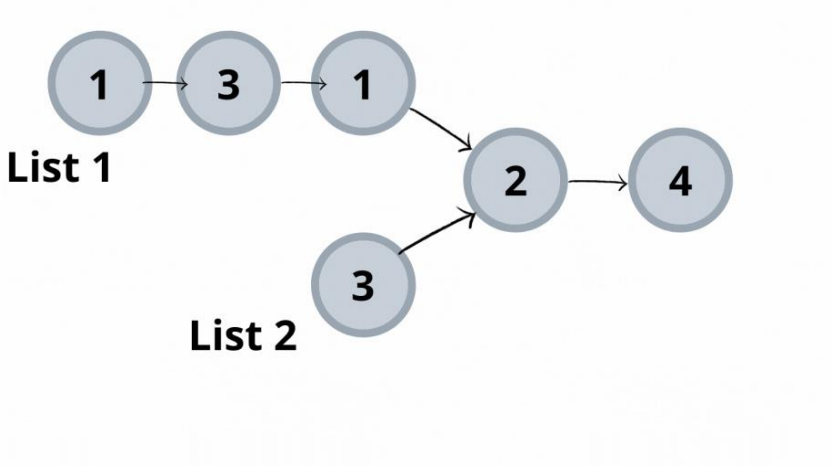
Solution 3: Difference of length

Approach:

We will reduce the search length. This can be done by searching the length of the shorter linked list. How? Let's see the process.

- Find length of both the lists.
- Find the positive difference of these lengths.
- Move the dummy pointer of the larger list by difference achieved. This makes our search length reduced to the smaller list length.
- Move both pointers, each pointing two lists, ahead simultaneously if both do not collide.

Dry Run:



Code:

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int a=0,b=0;
        ListNode curr1=headA,curr2=headB;
        while(curr1!=null||curr2!=null)
        {
            if(curr1!=null)
            {
                a++;
                curr1=curr1.next;
            }
            if(curr2!=null)
            {
                 curr2=curr2.next;
                b++;
            }
        }
        if(a>b)
        {
            curr=headA;
            headA=headB;
            headB=curr;
        }
        int p=Math.abs(b-a);
        for(int i=0;i<p&&headB!=null;i++)
        {
            headB=headB.next;
        }
```

```
        System.out.println("headA : "+headA.val+" headB : "+headB.val);
         while(headA!=null&&headB!=null)
         {
             if(headA==headB)
                 return headA;
             headA=headA.next;
             headB=headB.next;
         }
         return null;
    }
```

Time Complexity:

O(2max(length of list1,length of list2))+O(abs(length of list1-length of list2))+O(min(length of list1,length of list2))

Reason: Finding the length of both lists takes max(length of list1, length of list2) because it is found simultaneously for both of them. Moving the head pointer ahead by a difference of them. The next one is for searching.

Space Complexity: O(1)

Reason: No extra space is used.

Solution 4: Optimised

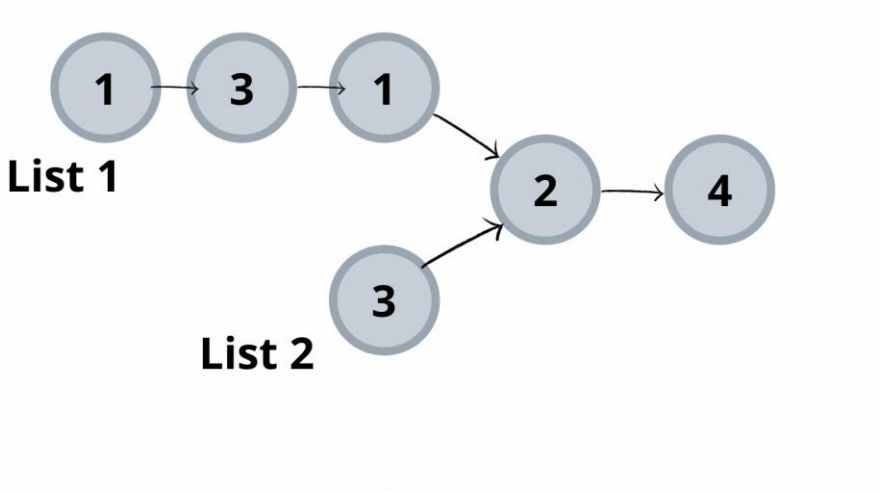
Approach:

The difference of length method requires various steps to work on it. Using the same concept of difference of length, a different approach can be implemented. The process is as follows:-

- Take two dummy nodes for each list. Point each to the head of the lists.

- Iterate over them. If anyone becomes null, point them to the head of the opposite lists and continue iterating until they collide.

Dry Run:



Code:

```
int intersectPoint(Node head1, Node head2)
    {
        Node a= head1;
        Node b=head2;
        while(a!=b)
        {
            a=a==null?head2:a.next;
            b=b==null?head1:b.next;
        }
        return a==null?-1:a.data;
    }
```