Intersection of two linked list

BRUTE FORCE:

Select a node of listA and iterate the whole listB if the selected node is equal to any node then return it otherwise keep iterating

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        while(headA!=NULL)
        {
            ListNode* temp=headB;
            while(temp!=NULL)
            {
                if(temp==headA)
                {
                    return temp;
                }
                temp=temp->next;
            }
            headA=headA->next;
        }
        return NULL;
    }
};
```

- Time Complexity : O(M*N)

For each node the other list is traversed so the complexity is O(M*N)

- Space Complexity : O(1)

Better Approach : Using Hashing

Instead of traversing the whole second list again and again we store the one list's all node addresses in the map and iterate the other list only. If while iterating any node is found in map we will return that node.

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        unordered_map<ListNode*,int> m;
        ListNode* temp1=headA;
        while(temp1!=NULL)
        {
            m[temp1]++;
            temp1=temp1->next;
        }
        ListNode* temp=headB;
        while(temp!=NULL)
        {
           if(m.find(temp)!=m.end())
            return temp;
           temp=temp->next;
        }
        return NULL;
    }
};
```

- Time Complexity : O(M)+O(N)
- Space Complexity : O(M)

Optimised Approach :

We find the difference in length of both list and then position one of the pointer to the abs of the lists. We position second pointer at beginning of the list and then wherever both pointer points to same address we find the intersection if no intersection is found and the pointers reach null then we return NULL.

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int m=0,n=0;
        ListNode* temp1=headA;
        ListNode* temp2=headB;
        while(temp1!=NULL)
        {
            m++;
            temp1=temp1->next;
```

```
        }
        while(temp2!=NULL)
        {
            n++;
            temp2=temp2->next;
        }
        int diff=abs(m-n);

        if(m>n)
        {
            temp1=headA;
            temp2=headB;
        }
        else
        {
            temp1=headB;
            temp2=headA;
        }
       for(int i=0;i<diff;i++)
       {
           temp1=temp1->next;
       }
       while(temp1 && temp2)
       {
           if(temp1==temp2)
            return temp1;
            temp1=temp1->next;
            temp2=temp2->next;
       }
    return NULL;
    }
};
```

- Time Complexity : O(M)+O(N)
- Space Complexity : O(1)

Optimised Approach 2 :

We traverse till either reach NULL. Then we position the pointer that reach NULL to the opposite list starting. for ex: d1 reach NULL then we place d1 at starting of list2 now againstart moving the pointers when the second pointer reaches NULL it means that the next pointer has reached the differenec position we then reset d2 pointer also to starting of other list. We keep moving the pointers now and comparing the pointers if they point to same node. return if same node found else return NUll when loop ends.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int m=0,n=0;
        ListNode* temp1=headA;
        ListNode* temp2=headB;
        if(headA==headB)
        {
            return headA;
        }
        while(temp1!=NULL || temp2!=NULL)
        {

            if(temp1)
            {
                temp1=temp1->next;
            }
            if(temp2)
            {
                temp2=temp2->next;
            }
            if(temp1==NULL && temp2!=NULL)
            {
                temp1=headB;
            }
            if(temp2==NULL && temp1!=NULL)
            {
                temp2=headA;
            }

            if(temp1==temp2)
            {
                return temp1;
            }
        }

    return NULL;
    }
};
```

- Time Complexity : O(max(M,N))
- Space Complexity : O(1)