GeeksMan
Linked List
Lesson 5

Detect Loop in linked list

Given a linked list of N nodes. The task is to check if the linked list has a loop. Linked list can contain a self loop.

Example :

Input:
N = 3
value[] = {1,3,4}
x = 2

Output: True

Explanation: In above test case N = 3.
The linked list with nodes N = 3 is given. Then the value of x=2 is given which means the last node is connected with the xth node of the linked list. Therefore, there exists a loop.

Your Task:

The task is to complete the function detectloop() which contains reference to the head as the only argument. This function should return 1 if the linked list contains a loop, else return 0.

Expected Time Complexity: $O(N)$

Expected Auxiliary Space: $O(1)$

Constraints:
1 <= N <= 10^4
1 <= Data on Node <= 10^3

(1)Modify the data structure

Have a visited flag with each node.Traverse the linked list and keep marking visited nodes.If you see a visited node again then there is a loop.

```
struct Node
{
  int data;
  struct Node* next;
  int flag;
};
```

Time complexity:O(n).

Only one traversal of the loop is needed.

Auxiliary Space:O(1).

No extra space is needed.

(2) Marking visited nodes without modifying the linked list data structure

In this method, a temporary node is created. The next pointer of each node that is traversed is made to point to this temporary node. This way we are using the next pointer of a node as a flag to indicate whether the node has been traversed or not.
Every node is checked to see if the next is pointing to a temporary node or not. In the case of the first node of the loop, the second time we traverse it this condition will be true, hence we find that loop exists.
If we come across a node that points to null then the loop doesn't exist.

Time complexity: O(n).

Only one traversal of the loop is needed.

Auxiliary Space: O(1).

There is no space required.

(3)Hashing Approach:

Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

```
1.  bool detectLoop(struct Node* h)
2.  {
3.
4.      unordered_set<Node*> s;
5.
6.      while (h != NULL)
7.  {
8.              if (s.find(h) != s.end())
9.              return true;
10.
11.             s.insert(h);
12.             h = h->next;
13.     }
14.
15.     return false;
16. }
```

Time complexity: O(n).

Only one traversal of the loop is needed.

Auxiliary Space: O(n).

n is the space required to store the value in a hashmap.

(4)Floyd's Cycle-Finding Algorithm :

Concept of a slow and a fast pointer

Traverse linked list using two pointers.Move one pointer(slow) by one and another pointer(fast) by two.If these pointers meet at the same node then there is a loop. If pointers do not meet then the linked list doesn't have a loop.

```
1.  bool detectLoop(Node* head)
2.  {
3.     Node *fp = head , *sp = head;
4.     while(fp && sp && sp->next)
5.     {
6.        fp = fp->next;
7.        sp = sp->next->next;
8.        if(fp == sp)
9.        {
10.          return true;
11.       }
12.    }
13.    return false;
14. }
```

Time complexity: O(n).

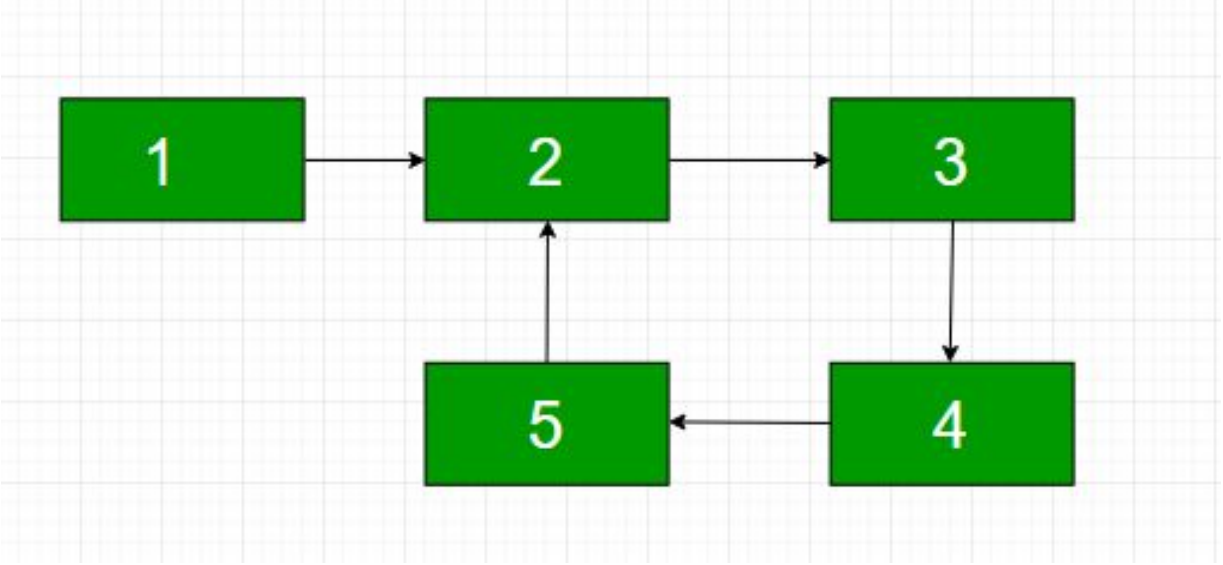
Only one traversal of the loop is needed.

Auxiliary Space:O(1).

There is no space required.

Find length of Loop

Given a linked list of size N. The task is to complete the function countNodesinLoop() that checks whether a given Linked List contains a loop or not and if the loop is present then return the count of nodes in a loop or else return 0. *C* is the position of the node to which the last node is connected. If it is 0 then no loop.



Example 1:

Input:
N = 10
value[]={25,14,19,33,10,21,39,90,58,45}
C = 4

Output: 7

Explanation: The loop is 45->33. So the length of the loop is 33->*10*->21->39->90->58->*45* = 7. The number 33 is connected to the last node to form the loop because according to the input the 4th node from the beginning(1 based index) will be connected to the last node for the loop.

Example 2:

Input:
N = 2
value[] = {1,0}
C = 1

Output: 2

Explanation: The length of the loop is 2.

Your Task:

The task is to complete the function countNodesinLoop() which contains the only argument as reference to the head of the linked list and return the length of the loop (0 if there is no loop).

Expected Time Complexity: *O*(N)

Expected Auxiliary Space: *O*(1)

Constraints:

1 <= N <= 500

0 <= *C* <= N-1

Algorithm:

Find the common point in the loop by using the Floyd's Cycle detection algorithmStore the pointer in a temporary variable and keep a count = 0.Traverse the linked list until the same node is reached again and increase the count while moving to the next node.Print the count as length of loop

```
1.  int countNodesinLoop(struct Node *head)
2.  {
3.     Node *slow=head,*fast=head;
4.     while(fast->next && fast->next->next && fast->next!=slow)
5.     {
6.        slow=slow->next;
7.        fast=fast->next->next;
8.     }
9.     if(fast->next!=slow)
10.       return 0;
11.    int n=1;
12.    while(slow!=fast)
13.    {
14.       n++;
15.       slow=slow->next;
16.    }
17.    return n;
18. }
```

Remove loop in Linked List

You are given a linked list of N nodes. Remove the loop from the linked list, if present.

Note: X is the position of the node to which the last node is connected to. If it is 0, then there is no loop.

Example 1:

Input:
N = 3
value[] = {1,3,4}
X = 2
Output: 1
Explanation: The link list looks like
1 -> 3 -> 4
A loop is present. If you remove it successfully, the answer will be 1.
Your Task:

You don't need to read input or print anything. Your task is to complete the function removeLoop() which takes the head of the linked list as input parameter. Simply remove the loop in the list (if present) without disconnecting any nodes from the list. The driver code will print 1 if your code is correct.

Expected time complexity : O(n)

Expected auxiliary space : O(1)

Constraints:

1 <= N <= 10^4

(1)Check one by one

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. Store the address of this common node in a pointer variable say ptr2. After that start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. Whenever we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get the pointer to the previous of this node.

```
int detectAndRemoveLoop(struct Node* list)
 1.  {
 2.    struct Node *slow_p = list, *fast_p = list;
 3.
 4.    while (slow_p && fast_p && fast_p->next)
 5.    {
 6.       slow_p = slow_p->next;
 7.       fast_p = fast_p->next->next;
 8.
 9.       if (slow_p == fast_p)
10.      {
11.          removeLoop(slow_p, list);
12.          return 1;
13.      }
14.  }
15.  return 0;
16.}
17.
```

```
18. void removeLoop(struct Node* loop_node, struct Node* head)
19. {
20.  struct Node* ptr1;
21.  struct Node* ptr2;
22.
23.  ptr1 = head;
24.  while (1)
25.  {
26.      ptr2 = loop_node;
27.      while (ptr2->next != loop_node && ptr2->next != ptr1)
28.          ptr2 = ptr2->next;
29.
30.      if (ptr2->next == ptr1)
31.          break;
32.
33.      ptr1 = ptr1->next;
34.  }
35.  ptr2->next = NULL;
36.}
```

(2)This method is also dependent on Floyd's Cycle detection algorithm.

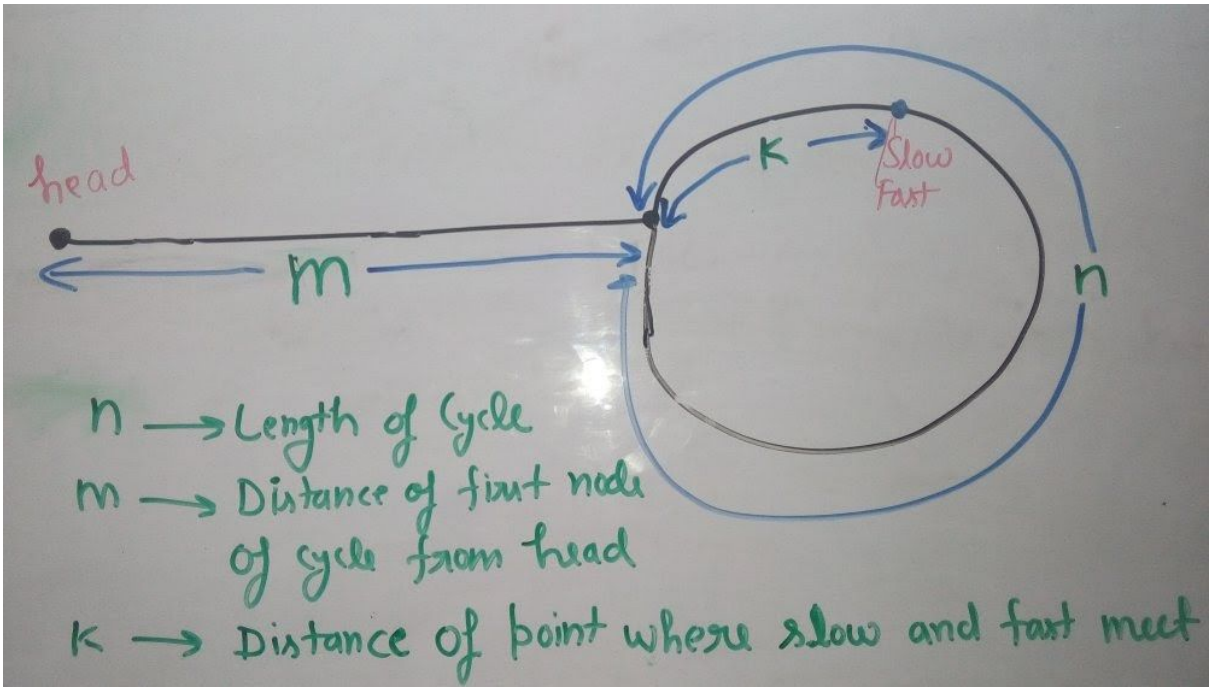
Detect Loop using Floyd's Cycle detection algorithm and get the pointer to a loop node.Count the number of nodes in the loop. Let the count be k.**Fix one pointer to the head and another to a kth node from the head.**Move both pointers at the same pace, they will meet at the loop starting node.Get a pointer to the last node of the loop and make next of it as NULL.

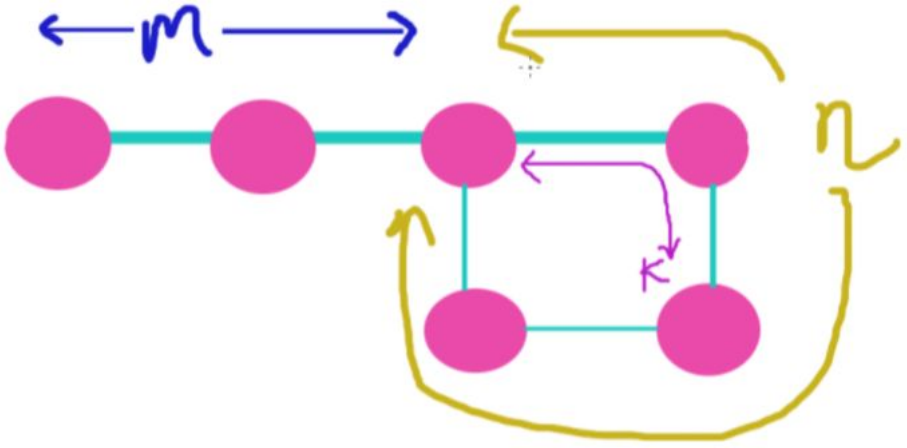
```
1.  void removeLoop1(struct Node* loop_node, struct Node* head)
2.  {
3.    struct Node* ptr1 = loop_node;
4.    struct Node* ptr2 = loop_node;
5.
6.    // Count the number of nodes in loop
7.    unsigned int k = 1, i;
8.    while (ptr1->next != ptr2)
9.    {
10.      ptr1 = ptr1->next;
11.      k++;
12.   }
13.
14.   ptr1 = head;
15.   ptr2 = head;
16.
17.   for (i = 0; i < k; i++)
18.      ptr2 = ptr2->next;
19.
20.   while (ptr2 != ptr1)
21.   {
22.      ptr1 = ptr1->next;
```

```
23.     ptr2 = ptr2->next;
24.  }
25.
26.  while (ptr2->next != ptr1)
27.     ptr2 = ptr2->next;
28.
29.   ptr2->next = NULL;
30.}
31.
32.void removeLoop(Node* list)
33.{
34.  struct Node *slow_p = list, *fast_p = list;
35.
36.  while (slow_p && fast_p && fast_p->next)
37.  {
38.     slow_p = slow_p->next;
39.     fast_p = fast_p->next->next;
40.
41.     if (slow_p == fast_p)
42.     {
43.        removeLoop1(slow_p, list);
44.     }
45.  }
46.}
47.
```

(3)m + k = i*n

Without counting the number of nodes.





We do not need to count the number of nodes in the Loop. After detecting the loop, if we start slow pointer from head and move both

slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of the loop.

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when the cycle is found.

Distance traveled by fast pointer = 2 * (Distance traveled by slow pointer)

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, the fast was moving at twice speed.

x --> Number of complete cyclic rounds made by fast pointer before they meet first time

y --> Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x-2y)*n$$

Which means m+k is a multiple of n.

Thus we can write, $m + k = i*n$ or $m = i*n - k$.

Hence, distance moved by slow pointer: m, is equal to distance moved by fast pointer:$i*n - k$ or $(i-1)*n + n - k$ (cover the loop completely i-1 times and start from n-k).

So if we start moving both pointers again at same speed such that one pointer (say slow) begins from the head node of the linked list and another pointer (say fast) begins from the meeting point. When a slow pointer reaches the beginning of the loop (has made m steps), a fast pointer would have also moved m steps as they are now moving at the same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning.

```
1. void detectAndRemoveLoop(Node* head)
2. {
```

```
3.      if (head == NULL || head->next == NULL)
4.       return;
5.    Node *slow = head, *fast = head;
6.    slow = slow->next;
7.    fast = fast->next->next;
8.      while (fast && fast->next) {
9.       if (slow == fast)
10.          break;
11.      slow = slow->next;
12.      fast = fast->next->next;
13.   }
14.    if (slow == fast)
15.   {
16.      slow = head;
17.      //if full ll is loop
18.        if(slow == fast)
19.     {
20.           while(fast->next != slow)
21.           fast = fast->next;
22.     }
23.       else {
24.         while (slow->next != fast->next) {
25.            slow = slow->next;
26.            fast = fast->next;
27.         }
28.     }
29.     fast->next = NULL;
30.   }
```

(4)Hashing: Hash the address of the linked list nodes

Time complexity = O(N)

Space complexity = O(N)

We can hash the addresses of the linked list nodes in an unordered map and just check if the element already exists in the map. If it exists, we have reached a node which already exists by a cycle, hence we need to make the last node's next pointer NULL

```
void hashAndRemove(Node* head)
1.  {
2.      unordered_map<Node*, int> node_map;
3.      Node* last = NULL;
4.    while (head != NULL) {
5.           if (node_map.find(head) == node_map.end()) {
6.           node_map[head]++;
7.           last = head;
8.           head = head->next;
9.       }
10.      else
11. {
12.          last->next = NULL;
13.          break;
14.      }
15.  }
16. }
```

Intersection Point in Y Shaped Linked Lists

Given two singly linked lists of size N and M, write a program to get the point where two linked lists intersect each other.

Example 1:

Input:

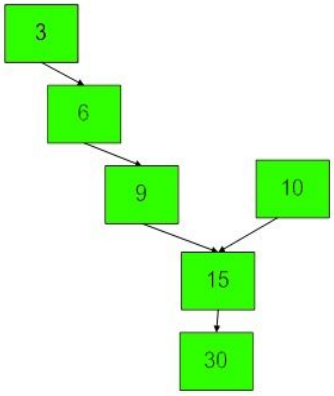
LinkList1 = 3->6->9->common

LinkList2 = 10->common

common = 15->30->NULL

Output: 15

Explanation:



Example 2:

Input:

Linked List 1 = 4->1->common

Linked List 2 = 5->6->1->common

common = 8->4->5->NULL

Output: 8

Your Task:

You don't need to read input or print anything. The task is to complete the function intersetPoint() which takes the pointer to the head of linklist1(head1) and linklist2(head2) as input parameters and returns data value of a node where two linked lists intersect. If the linked list does not merge at any point, then it should return -1.

Expected Time Complexity: O(N+M)

Expected Auxiliary Space: O(1)
Constraints:

1 ≤ N + M ≤ 2*105

-1000 ≤ value ≤ 1000

1)Use two loops and for each node in L1 check each node for L2
Use 2 nested for loops. The outer loop will be for each node of the 1st list and the inner loop will be for the 2nd list. In the inner loop, check if any of the nodes of the 2nd list is the same as the current node of the first linked list. The time complexity of this method will be O(M * N) where m and n are the numbers of nodes in two lists.

2) Calculate difference of length and traverse from equal distance from the intersection
Get count of the nodes in the first list, let count be c1.

- Get count of the nodes in the second list, let count be c2.

- Get the difference of counts d = abs(c1 – c2)
- Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes
- Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

```
1.  int intersectPoint(Node* head1, Node* head2)
2.  {
3.    int l1=0,l2=0,l;
4.    Node *h1=head1,*h2=head2;
5.    while(h1!=NULL)
6.    {
7.       h1=h1->next;
8.       l1++;
9.    }
10.   while(h2!=NULL)
11.   {
12.      h2=h2->next;
13.      l2++;
14.   }
15.   l=abs(l1-l2);
16.   if(l1>l2)
17.   {
18.      while(l>0)
19.      {
20.         head1=head1->next;
21.         l--;
```

```
    }
}
else
{
    while(l>0)
    {
        head2=head2->next;
        l--;
    }
}
while(head1!=NULL && head2 != NULL)
{
    if(head1==head2)
    return (head1->data);
    head1=head1->next;
    head2=head2->next;
}
return -1;
}
```

3)Mark Visited Node

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse the second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works

in O(m+n) but requires additional information with each node. A variation of this solution that doesn't require modification to the basic data structure can be implemented using a hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in the hash then return the intersecting node.

4)Make a circle for the first list and now all you have to do is to find the starting node of the circle O(M+N).

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember the last node so that we can break the circle later on).

2. Now view the problem as finding the loop in the second linked list. So the problem is solved.

3. Since we already know the length of the loop(size of the first linked list) we can traverse those many numbers of nodes in the second list, and then start another pointer from the beginning of the second list. we have to traverse until they are equal, and that is the required intersection point.

4. remove the circle from the linked list.

Time Complexity: O(m+n)

Auxiliary Space: O(1)

5)Hashing

Basically, we need to find a common node of two linked lists. So we hash all nodes of the first list and then check the second list.

1) Create an empty hash set.

2) Traverse the first linked list and insert all nodes' addresses in the hash set.

3) Traverse the second list. For every node check if it is present in the hash set. If we find a node in the hash set, return the node.

6)2-pointer method

Using Two pointers :

Initialize two pointers ptr1 and ptr2 at the head1 and head2.Traverse through the lists,one node at a time.When ptr1 reaches the end of a list, then redirect it to the head 2.similarly when ptr2 reaches the end of a list, redirect it the head1.Once both of them go through reassigning,hey will be equidistant from the collision pointIf at any node ptr1 meets ptr2, then it is the intersection node.After the second iteration if there is no intersection node it returns NULL.

```
1. Node* intersectPoint(Node* head1, Node* head2)
2. {
3.   Node* ptr1 = head1;
4.   Node* ptr2 = head2;
5.     if (ptr1 == NULL || ptr2 == NULL) {
```

```
6.        return NULL;
7.    }
8.     while (ptr1 != ptr2) {
9.
10.      ptr1 = ptr1->next;
11.      ptr2 = ptr2->next;
12.      if (ptr1 == ptr2)
13.      {
14.          return ptr1;
15.      }
16.    if (ptr1 == NULL)
17.      {
18.          ptr1 = head2;
19.      }
20.      if (ptr2 == NULL)
21.      {
22.          ptr2 = head1;
23.      }
24.  }
25.  return ptr1;
26.}
```