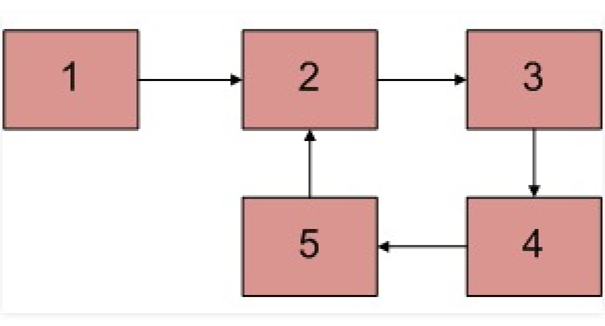
Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We strongly recommend that you click here and practice it, before moving on to the solution.

We also recommend to read following post as a prerequisite of the solution discussed here.

Write a C function to detect loop in a linked list

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When 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 pointer to the previous of this node.

C Java Python

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
   struct node *ptr1;
   struct node *ptr2;

   /* Set a pointer to the beging of the Linked List and
      move it one by one to find the first node which is
      part of the Linked List */
   ptr1 = head;
   while (1)
   {
     /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
     ptr2 = loop_node;
     while (ptr2->next != loop_node && ptr2->next != ptr1)
         ptr2 = ptr2->next;

     /* If ptr2 reahced ptr1 then there is a loop. So break the
        loop */
     if (ptr2->next == ptr1)
        break;

     /* If ptr2 did't reach ptr1 then try the next node after ptr1 */
     ptr1 = ptr1->next;
   }

   /* After the end of loop ptr2 is the last node of the loop. So
     make next of ptr2 as NULL */
   ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

Run on IDE

Output:

```
Linked List after removing loop
50 20 15 4 10
```

Method 2 (Better Solution)

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

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

Thanks to WgpShashank for suggesting this method.

C Java Python

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
      ptr2 = ptr2->next;

    /*  Move both pointers at the same pace,
      they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while (ptr2->next != ptr1)
       ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
      to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

Run on IDE

Output:

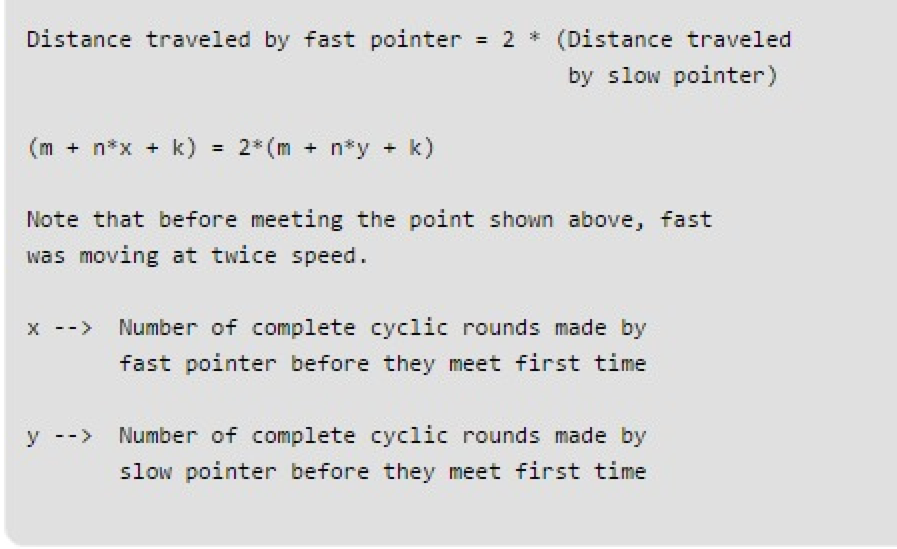
```
Linked List after removing loop
50 20 15 4 10
```

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in 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 linked list.

How does this work?

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



We can conclude below from above diagram

```
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, 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.
```

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

C++ Java Python

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}
```

Run on IDE

Output:

```
Linked List after removing loop
50 20 15 4 10
```

Thanks to Gaurav Ahirwar for suggesting above solution.