Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Note: Do not modify the linked list.

Follow up:
Can you solve it without using extra space?

Solution 1

my solution is like this: using two pointers, one of them one step at a time. another pointer each take two steps. Suppose the first meet at step ***k***,the length of the Cycle is ***r***. so..**2k-k=nr,k=nr** Now, the distance between the start node of list and the start node of cycle is ***s***. the distance between the start of list and the first meeting node is ***k***(the pointer which wake one step at a time waked k steps).the distance between the start node of cycle and the first meeting node is ***m***, so...**s=k-m, s=nr-m=(n-1)r+(r-m),here we takes n = 1**..so, using one pointer start from the start node of list, another pointer start from the first meeting node, all of them wake one step at a time, the first time they meeting each other is the start of the cycle.

```
    ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL) return NULL;

    ListNode* firstp = head;
    ListNode* secondp = head;
    bool isCycle = false;

    while(firstp != NULL && secondp != NULL) {
        firstp = firstp->next;
        if (secondp->next == NULL) return NULL;
        secondp = secondp->next->next;
        if (firstp == secondp) { isCycle = true; break; }
    }

    if(!isCycle) return NULL;
    firstp = head;
    while( firstp != secondp) {
        firstp = firstp->next;
        secondp = secondp->next;
    }

    return firstp;
}
```

written by wallop original link here

Solution 2

Alogrithm Description:

Step 1: Determine whether there is a cycle

1.1) Using a slow pointer that move forward 1 step each time

1.2) Using a fast pointer that move forward 2 steps each time

1.3) If the slow pointer and fast pointer both point to the same location after several moving steps, there is a cycle;

1.4) Otherwise, if (fast->next == NULL || fast->next->next == NULL), there has no cycle.

Step 2: If there is a cycle, return the entry location of the cycle

2.1) L1 is defined as the distance between the head point and entry point

2.2) L2 is defined as the distance between the entry point and the meeting point

2.3) C is defined as the length of the cycle

2.4) n is defined as the travel times of the fast pointer around the cycle When the first encounter of the slow pointer and the fast pointer

According to the definition of L1, L2 and C, we can obtain:

- the total distance of the slow pointer traveled when encounter is L1 + L2
- the total distance of the fast pointer traveled when encounter is L1 + L2 + n * C
- Because the total distance the fast pointer traveled is twice as the slow pointer, Thus:
- 2 * (L1+L2) = L1 + L2 + n * C => L1 + L2 = n * C =>**L1 = (n - 1)* C + (C - L2)**

It can be concluded that the distance between the head location and entry location is equal to the distance between the meeting location and the entry location along the direction of forward movement.

So, when the slow pointer and the fast pointer encounter in the cycle, we can define a pointer "entry" that point to the head, this "entry" pointer moves one step each time so as the slow pointer. When this "entry" pointer and the slow pointer both point to the same location, this location is the node where the cycle begins.

==========

Here is the code:

```
ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL)
        return NULL;

    ListNode *slow  = head;
    ListNode *fast  = head;
    ListNode *entry = head;

    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {                     // there is a cycle
            while(slow != entry) {              // found the entry location
                slow  = slow->next;
                entry = entry->next;
            }
            return entry;
        }
    }
    return NULL;                                // there has no cycle
}
```

written by ngcl original link here

Solution 3

Define two pointers slow and fast. Both start at head node, fast is twice as fast as slow. If it reaches the end it means there is no cycle, otherwise eventually it will eventually catch up to slow pointer somewhere in the cycle.

Let the distance from the first node to the the node where cycle begins be A, and let say the slow pointer travels travels A+B. The fast pointer must travel 2A+2B to catch up. The cycle size is N. Full cycle is also how much more fast pointer has traveled than slow pointer at meeting point.

```
A+B+N = 2A+2B
N=A+B
```

From our calculation slow pointer traveled exactly full cycle when it meets fast pointer, and since originally it travled A before starting on a cycle, it must travel A to reach the point where cycle begins! We can start another slow pointer at head node, and move both pointers until they meet at the beginning of a cycle.

```
public class Solution {
            public ListNode detectCycle(ListNode head) {
                ListNode slow = head;
                ListNode fast = head;

                while (fast!=null && fast.next!=null){
                    fast = fast.next.next;
                    slow = slow.next;

                    if (fast == slow){
                        ListNode slow2 = head;
                        while (slow2 != slow){
                            slow = slow.next;
                            slow2 = slow2.next;
                        }
                        return slow;
                    }
                }
                return null;
            }
        }
```

written by qgambit2 original link here