

142. Linked List Cycle II

Dec 7, 2017 | 56.4K views

Previous Next
Average Rating: 4.48 (61 votes)

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

To represent a cycle in the given linked list, we use an integer `pos` which represents the position (0-indexed) in the linked list where tail connects to. If `pos` is `-1`, then there is no cycle in the linked list.

Note: Do not modify the linked list.

Example 1:

Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node with value 2.

Example 2:

Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node with value 1.

Example 3:

Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

Follow-up:

Can you solve it without using extra space?

Approach 1: Hash Table

Intuition

If we keep track of the nodes that we've seen already in a `Set`, we can traverse the list and return the first duplicate node.

Algorithm

First, we allocate a `Set` to store `ListNode` references. Then, we traverse the list, checking `visited` for containment of the current node. If the node has already been seen, then it is necessarily the entrance to the cycle. If any other node were the entrance to the cycle, then we would have already returned that node instead. Otherwise, the `if` condition will never be satisfied, and our function will return `null`.

The algorithm necessarily terminates for any list with a finite number of nodes, as the domain of input lists can be divided into two categories: cyclic and acyclic lists. An acyclic list resembles a `null`-terminated chain of nodes, while a cyclic list can be thought of as an acyclic list with the final `null` replaced by a reference to some previous node. If the `while` loop terminates, we return `null`, as we have traversed the entire list without encountering a duplicate reference. In this case, the list is acyclic. For a cyclic list, the `while` loop will never terminate, but at some point the `if` condition will be satisfied and cause the function to return.

Java Python

```
1 class Solution(object):
2     def detectCycle(self, head):
3         visited = set()
4
5         node = head
6         while node is not None:
7             if node in visited:
8                 return node
9             else:
10                visited.add(node)
11                node = node.next
12
13        return None
```

Copy

Complexity Analysis

- Time complexity: $O(n)$

For both cyclic and acyclic inputs, the algorithm must visit each node exactly once. This is transparently obvious for acyclic lists because the n th node points to `null`, causing the loop to terminate. For cyclic lists, the `if` condition will cause the function to return after visiting the n th node, as it points to some node that is already in `visited`. In both cases, the number of nodes visited is exactly n , so the runtime is linear in the number of nodes.

- Space complexity: $O(n)$

For both cyclic and acyclic inputs, we will need to insert each node into the `Set` once. The only difference between the two cases is whether we discover that the "last" node points to `null` or a previously-visited node. Therefore, because the `Set` will contain n distinct nodes, the memory footprint is linear in the number of nodes.

Approach 2: Floyd's Tortoise and Hare

Intuition

What happens when a fast runner (a hare) races a slow runner (a tortoise) on a circular track? At some point, the fast runner will catch up to the slow runner from behind.

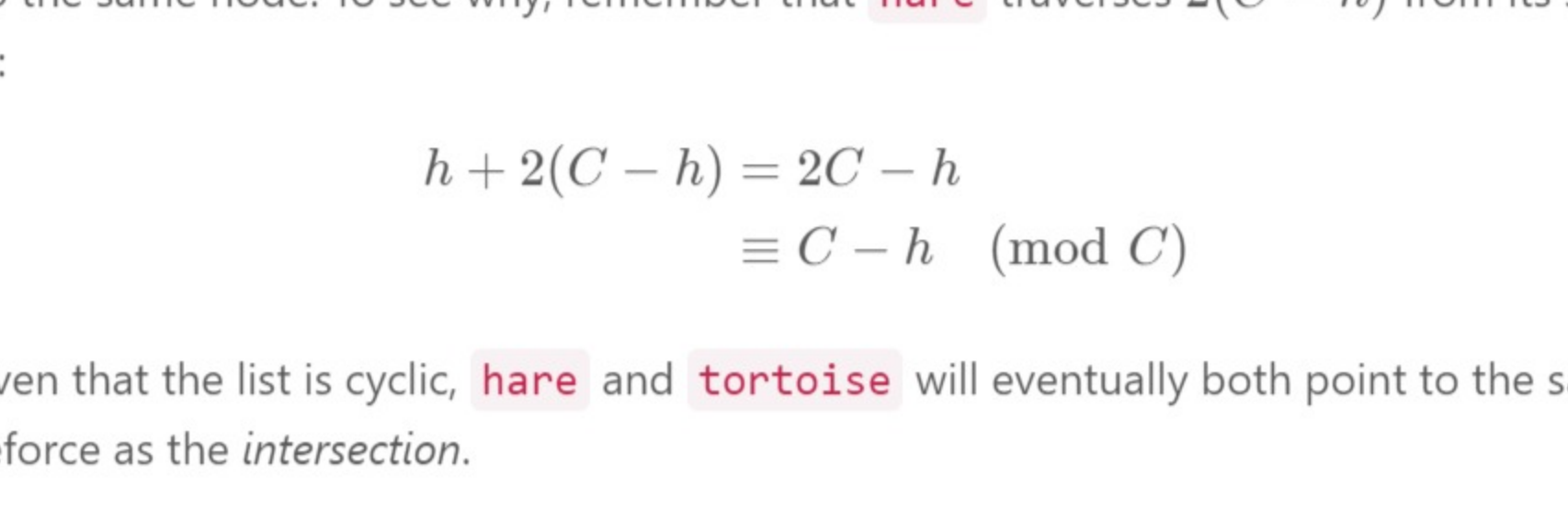
Algorithm

Floyd's algorithm is separated into two distinct *phases*. In the first phase, it determines whether a cycle is present in the list. If no cycle is present, it returns `null` immediately, as it is impossible to find the entrance to a non-existent cycle. Otherwise, it uses the located "intersection node" to find the entrance to the cycle.

Phase 1

Here, we initialize two pointers - the fast `hare` and the slow `tortoise`. Then, until `hare` can no longer advance, we increment `tortoise` once and `hare` twice.¹ If, after advancing them, `hare` and `tortoise` point to the same node, we return it. Otherwise, we continue. If the `while` loop terminates without returning a node, then the list is acyclic, and we return `null` to indicate as much.

To see why this works, consider the image below:



Here, the nodes in the cycle have been labelled from 0 to $C - 1$, where C is the length of the cycle. The noncyclic nodes have been labelled from $-F'$ to -1 , where F' is the number of nodes outside of the cycle. After F' iterations, `tortoise` points to node 0 and `hare` points to some node h , where $F' \equiv h \pmod C$. This is because `hare` traverses $2F'$ nodes over the course of F' iterations, exactly F' of which are in the cycle. After $C - h$ more iterations, `tortoise` obviously points to node $C - h$, but (less obviously) `hare` also points to the same node. To see why, remember that `hare` traverses $2(C - h)$ from its starting position of h :

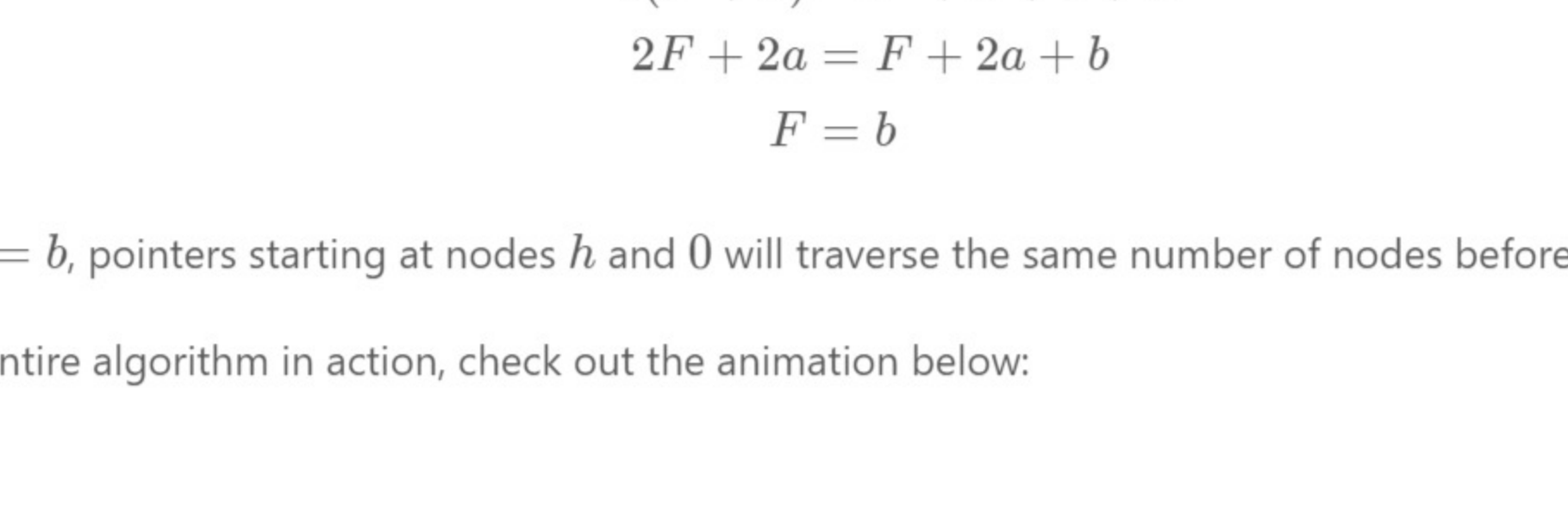
$$\begin{aligned} h + 2(C - h) &= 2C - h \\ &\equiv C - h \pmod C \end{aligned}$$

Therefore, given that the list is cyclic, `hare` and `tortoise` will eventually both point to the same node, known henceforward as the *intersection*.

Phase 2

Given that phase 1 finds an intersection, phase 2 proceeds to find the node that is the entrance to the cycle. To do so, we initialize two more pointers: `ptr1`, which points to the head of the list, and `ptr2`, which points to the intersection. Then, we advance each of them by 1 until they meet; the node where they meet is the entrance to the cycle, so we return it.

Use the diagram below to help understand the proof of this approach's correctness.

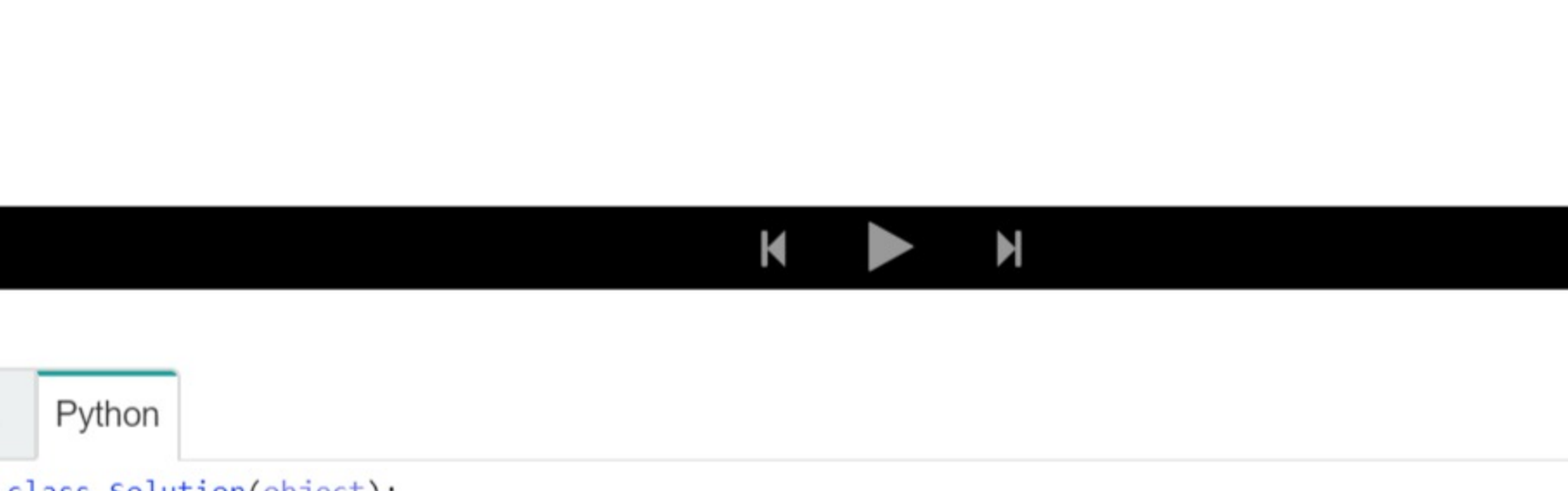


We can harness the fact that `hare` moves twice as quickly as `tortoise` to assert that when `hare` and `tortoise` meet at node h , `hare` has traversed twice as many nodes. Using this fact, we deduce the following:

$$\begin{aligned} 2 \cdot \text{distance}(\text{tortoise}) &= \text{distance}(\text{hare}) \\ 2(F + a) &= F + a + b + a \\ 2F + 2a &= F + 2a + b \\ F &= b \end{aligned}$$

Because $F = b$, pointers starting at nodes h and 0 will traverse the same number of nodes before meeting.

To see the entire algorithm in action, check out the animation below:



Java Python

```
1 class Solution(object):
2     def getIntersection(self, head):
3         tortoise = head
4         hare = head
5
6         # A fast pointer will either loop around a cycle and meet the slow
7         # pointer or reach the 'null' at the end of a non-cyclic list.
8         while hare is not None and hare.next is not None:
9             tortoise = tortoise.next
10            hare = hare.next.next
11            if tortoise == hare:
12                return tortoise
13
14        return None
15
16 def detectCycle(self, head):
17     if head is None:
18         return None
19
20     # If there is a cycle, the fast/slow pointers will intersect at some
21     # node. Otherwise, there is no cycle, so we cannot find an entrance to
22     # a cycle.
23     intersect = self.getIntersection(head)
24     if intersect is None:
25         return None
26
27     # To find the entrance to the cycle, we have two pointers traverse at
```

Copy

Complexity Analysis

- Time complexity: $O(n)$

For cyclic lists, `hare` and `tortoise` will point to the same node after $F + C - h$ iterations, as demonstrated in the proof of correctness. $F + C - h \leq F + C = n$, so phase 1 runs in $O(n)$ time. Phase 2 runs for $F < n$ iterations, so it also runs in $O(n)$ time.

For acyclic lists, `hare` will reach the end of the list in roughly $\frac{n}{2}$ iterations, causing the function to return before phase 2. Therefore, regardless of which category of list the algorithm receives, it runs in time linearly proportional to the number of nodes.

- Space complexity: $O(1)$

Floyd's Tortoise and Hare algorithm allocates only pointers, so it runs with constant overall memory usage.

Footnotes

1. It is sufficient to check only `hare` because it will always be ahead of `tortoise` in an acyclic list.

Rate this article: ★★★★★

Previous Next

Comments: 25

Sort By

Type comment here... (Markdown is supported)

Preview

Post

- mglezer

★146

August 14, 2018 10:47 PM

Report

`F = b` is incorrect, which is obvious when `F` is very large and `C` is very small. Rather, from $2 \cdot d(\text{tortoise}) = d(\text{hare})$, we have $2(F+a) = F+nC+a$, for some integer n , so $F+a = nC$, or $F = nC-a$. In phase two, you dispatch another tortoise from the head of the list, and slow down the hare in the cycle to advance one node per step. After F steps the tortoise will travel F nodes, and the hare will travel $2F+a$ nodes. The two have started at $F+a = nC-a$ on the hare's position after F steps so

Read More

100

Share

Reply

SHOW 19 REPLIES
- wise666

★22

January 17, 2018 11:16 AM

Report

I don't think this solution is faithfulness enough. The F should be equals to $nC+b$, ($n>=0$) rather than b , since F may be very large, but b may be very small.

But the method still works well.

22

Share

Reply
- henrykira

★221

March 12, 2019 7:33 PM

Report

Please, fix your output statement in the description...! felt like I need to output a string.

Why do you put something like "Output: tail connects to node index 0" when you actually want "Node"?

17

Share

Reply

SHOW 2 REPLIES
- alemtuev

★16

November 3, 2019 9:37 PM

Report

This guy explains it well: <https://www.youtube.com/watch?v=LUm2ABqAs1w>

Read More

10

Share

Reply
- topcoder811

★152

June 17, 2019 12:08 AM

Report

$F \equiv h \pmod C$, its other way around $h \equiv F \pmod C$. Please fix it.

6

Share

Reply
- kk20080142

★35

March 7, 2019 5:25 AM

Report

The proof of `F=b` is wrong, instead it should be `F=b (mod C)`

7

Share

Reply

SHOW 1 REPLY
- s961206

★735

July 16, 2019 1:59 PM

Report

Simply your solution 1:

```
public ListNode detectCycle(ListNode head) {
    Set<ListNode> set = new HashSet();
    while(head != null && set.add(head)) head = head.next;
    return head;
}
```

Read More

1

Share

Reply
- quyu8290

★45

October 4, 2018 9:30 PM

Report

I have a question about the certainty that hare and tortoise will eventually intersect: for each iteration, The article suggests:

 1. advance one step tortoise
 2. advance two steps hare, check if tortoise and hare intersect

Read More

1

Share

Reply

SHOW 3 REPLIES
- WIND03

★3

January 9, 2018 5:50 PM

Report

$F=b + (n-1)(a-b) \cdot n \geq 1$ so they while meet at 0

1

Share

Reply
- progressivo15

★16

December 13, 2017 7:18 PM

Report

perfect guide for me. thx for writing this article.

1

Share

Reply