Previous (/articles/delete-and-earn/) Next (/articles/diameter-of-binary-tree/)

142. Linked List Cycle II (/problems/linked-list-cycle-ii/)

Dec. 7, 2017 | 27.4K views

Average Rating: 4.83 (24 votes)

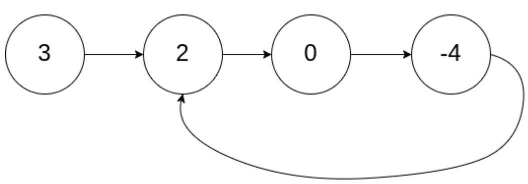
Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

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.

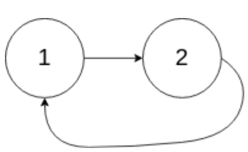
Example 1:

```
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.
```



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.
```



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 Articles > 142. Linked List Cycle II Copy

```
class Solution(object):
    def detectCycle(self, head):
        visited = set()

        node = head
        while node is not None:
            if node in visited:
                return node
            else:
                visited.add(node)
                node = node.next

        return None
```

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 nth node points to `null`, causing the loop to terminate. For cyclic lists, the `if` condition will cause the function to return after visiting the nth 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.

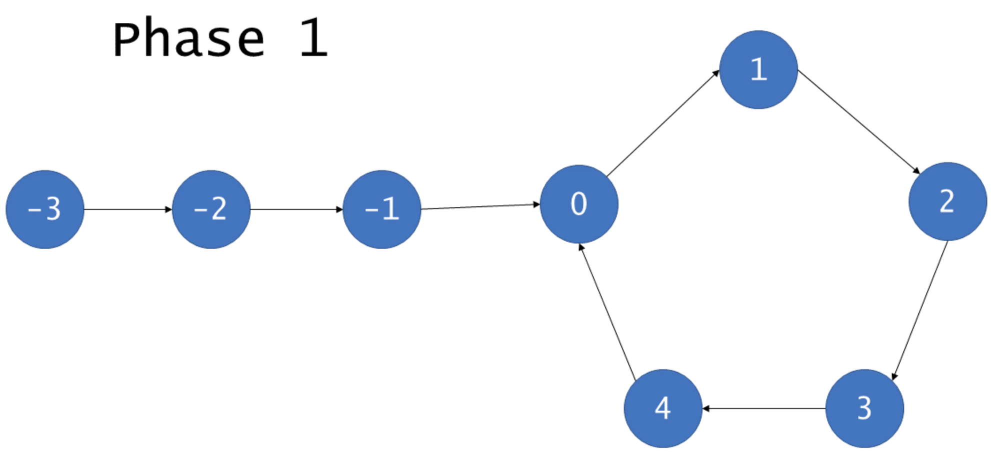
Algorithm

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 nonexistant 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:

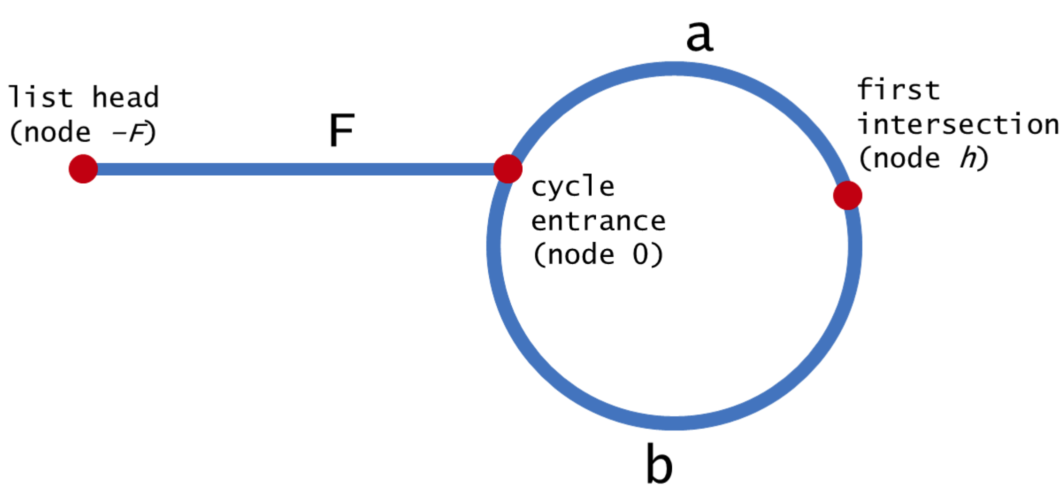
$$\begin{aligned} h + 2(C-h) &= 2C - h \\ &\equiv C - h \pmod{C} \end{aligned}$$

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

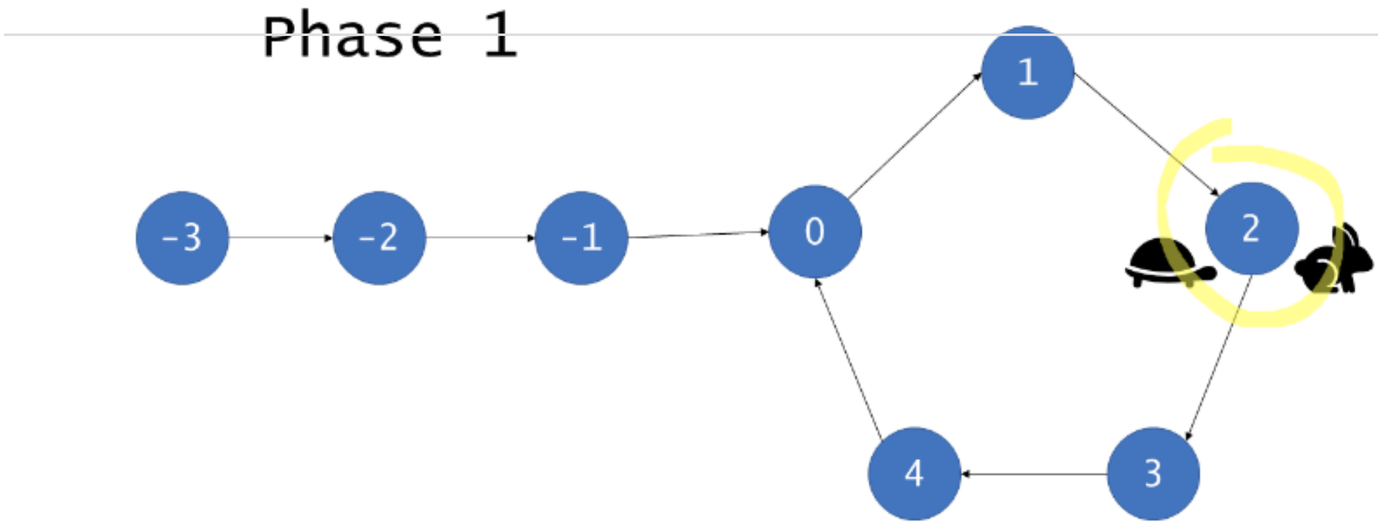


We can harness the fact that `hare` moves twice as quickly as `tortoise` to assert that when `hare` and `tortoise` meet at node h, `hare` has traversed twice as many nodes. Using this fact, we deduce the following:

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

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:



8 / 1

Java | Python | Copy

```
class Solution(object):
    def getIntersect(self, head):
        tortoise = head
        hare = head

        # A fast pointer will either loop around a cycle and meet the slow
        # pointer or reach the `null` at the end of a non-cyclic list.
        while hare is not None and hare.next is not None:
            tortoise = tortoise.next
            hare = hare.next.next
            if tortoise == hare:
                return tortoise

        return None

    def detectCycle(self, head):
        if head is None:
            return None

        # If there is a cycle, the fast/slow pointers will intersect at some
        # node. Otherwise, there is no cycle, so we cannot find an entrance to
        # a cycle.
        intersect = self.getIntersect(head)
        if intersect is None:
            return None

        # To find the entrance to the cycle, we have two pointers traverse at
```

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 (/articles/delete-and-earn/)

Next (/articles/diameter-of-binary-tree/)

Comments: 14

Sort By

Type comment here... (Markdown is supported)

Preview Post

(/mglezer)

mglezer (mglezer) ★ 66 August 14, 2018 10:17 AM

`F = b` is incorrect, which is obvious when `F` is very large and `C` is very small. Rather, from `d(tortoise) = d(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 `F=nC-a` nodes. The hare started at `F+a`, so the

Read More

43 Share Reply

SHOW 8 REPLIES