# Question

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

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

**Notice** that you **should 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.

**Constraints:**

* The number of the nodes in the list is in the range [0, 104].
* -105 <= Node.val <= 105
* pos is -1 or a **valid index** in the linked-list.

**Follow up:** Can you solve it using O(1) (i.e. constant) memory?

# Solution

**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.

|  |
| --- |
| public class Solution {  public ListNode detectCycle(ListNode head) {  Set<ListNode> visited = new HashSet<ListNode>();  ListNode node = head;  while (node != null) {  if (visited.contains(node)) {  return node;  }  visited.add(node);  node = node.next;  }  return null;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*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*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*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*n*, so the runtime is linear in the number of nodes.

* Space complexity : O(n)*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*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.

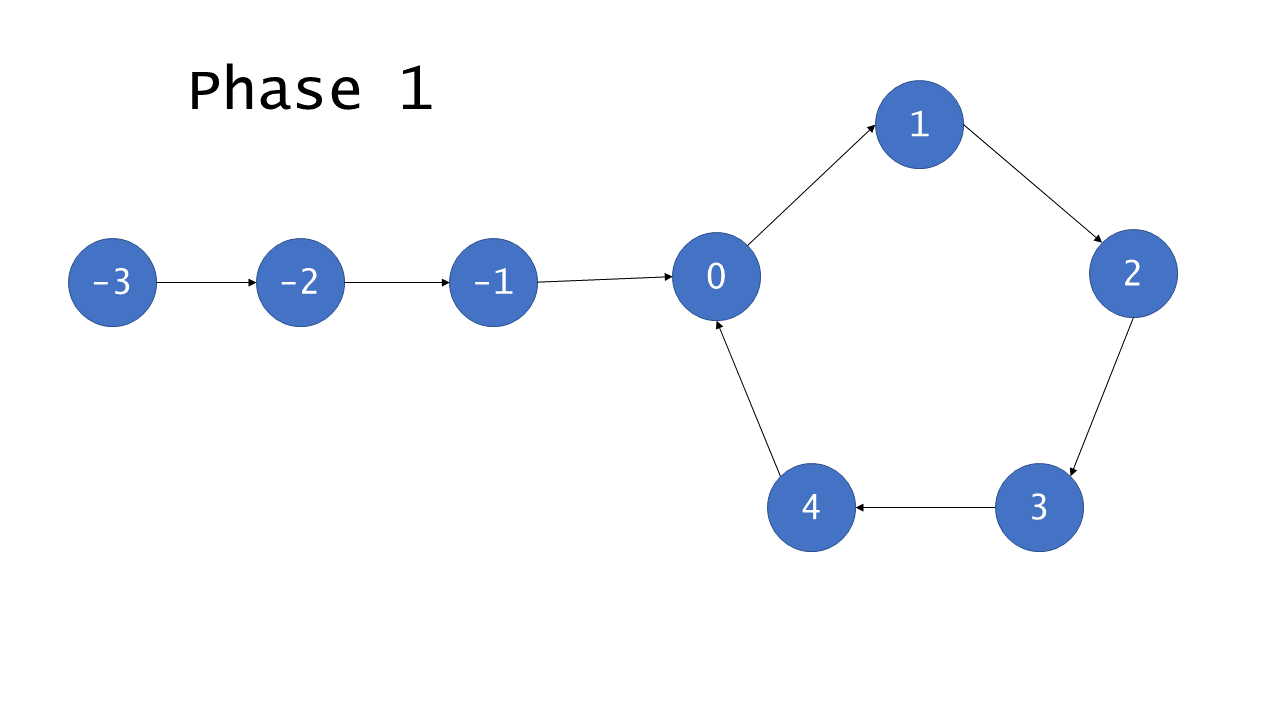
**Algorithm**

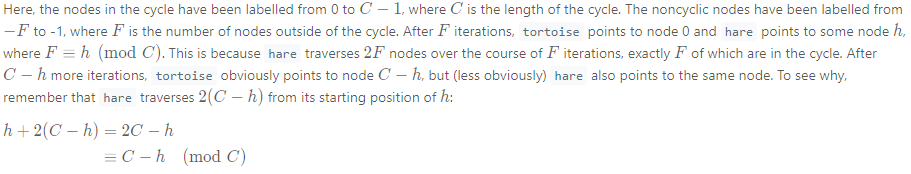
Floyd's algorithm is separated into two distinct phases. In the first phase, it determines whether a cycle is present in the list. If no cycle is present, it returns null immediately, as it is impossible to find the entrance to a nonexistant cycle. Otherwise, it uses the located "intersection node" to 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]](https://leetcode.com/problems/linked-list-cycle-ii/solution/#fn1) 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:



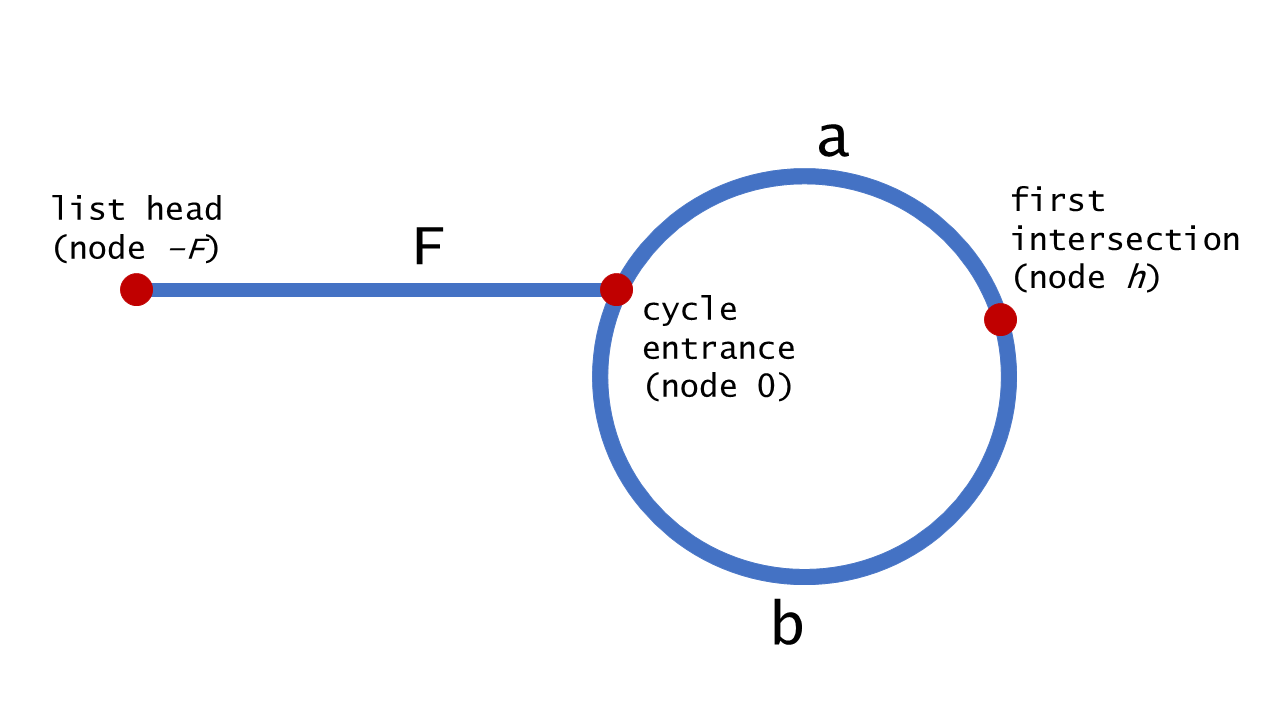


Therefore, given that the list is cyclic, hare and tortoise will eventually both point to the same node, known henceforce as the intersection.

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*h*, hare has traversed twice as many nodes. Using this fact, we deduce the following:

To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, i.e. 2d(\text{tortoise}) = d(\text{hare})2*d*(tortoise)=*d*(hare), that means

2(*F*+*a*)=*F*+*nC*+*a* , where n*n* is some integer.

Hence the coordinate of the intersection point is *F*+*a*=*nC*.

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

1 / 13

|  |
| --- |
| **public class Solution {**  **private ListNode getIntersect(ListNode head) {**  **ListNode tortoise = head;**  **ListNode 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 != null && hare.next != null) {**  **tortoise = tortoise.next;**  **hare = hare.next.next;**  **if (tortoise == hare) {**  **return tortoise;**  **}**  **}**  **return null;**  **}**  **public ListNode detectCycle(ListNode head) {**  **if (head == null) {**  **return null;**  **}**  **// 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.**  **ListNode intersect = getIntersect(head);**  **if (intersect == null) {**  **return null;**  **}**  **// To find the entrance to the cycle, we have two pointers traverse at**  **// the same speed -- one from the front of the list, and the other from**  **// the point of intersection.**  **ListNode ptr1 = head;**  **ListNode ptr2 = intersect;**  **while (ptr1 != ptr2) {**  **ptr1 = ptr1.next;**  **ptr2 = ptr2.next;**  **}**  **return ptr1;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : O(n)*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* ≤ *F*+*C*=*n*, so phase 1 runs in O(n)*O*(*n*) time. Phase 2 runs for F < n*F*<*n* iterations, so it also runs in O(n)*O*(*n*) time.

For acyclic lists, hare will reach the end of the list in roughly 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)*O*(1)

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