Solution: Linked List Cycle

Let's solve the Linked List Cycle problem using the Fast and Slow Pointers pattern.

We'll cover the following

Statement

Check whether or not a linked list contains a cycle. If a cycle exists, return TRUE. Otherwise, return FALSE. The cycle means that at least one node can be reached again by traversing the `next` pointer.

Constraints:

Let `n` be the number of nodes in a linked list.

- $0 \leq$ `n` ≤ 500
- $-10^5 \leq$ `Node.data` $\leq 10^5$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow, based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to traverse the linked list and store the current node in a set. At each iteration, we check if the current node is already present in the set. If it is, we've found a cycle and return TRUE. Otherwise, we add the node to the set. If the traversal has been completed, return FALSE, since we've reached the end of the linked list without encountering a cycle.

The time complexity is $O(n)$, since we only traverse the linked list once, where n represents the total number of nodes in a linked list. The space complexity of the naive approach is also $O(n)$, since in the worst case, we store n nodes in the set.

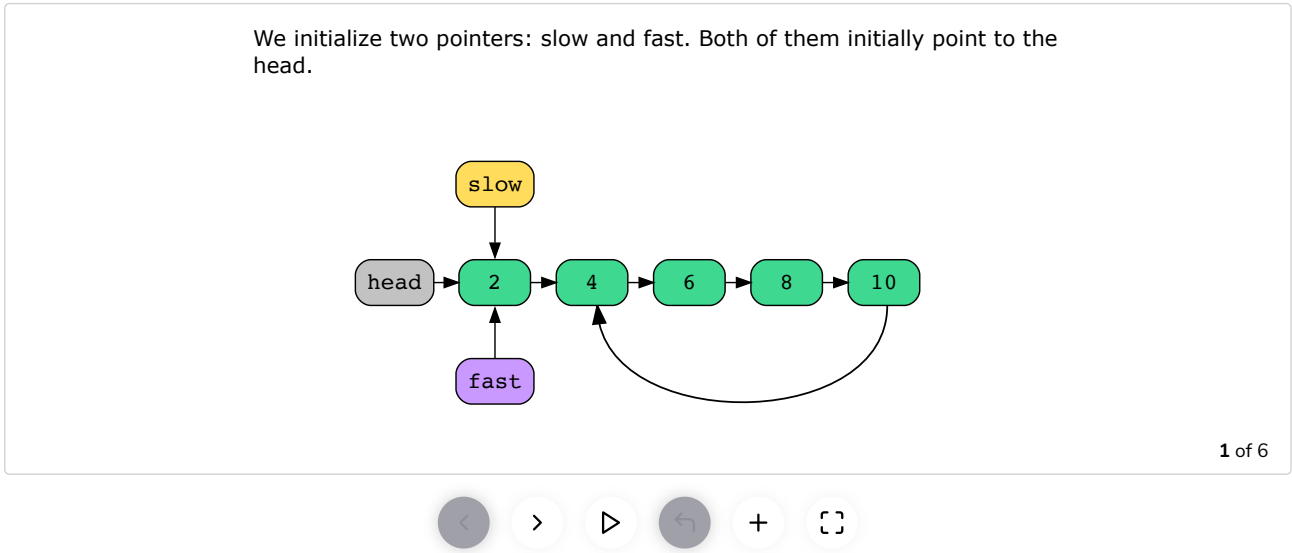
Optimized approach using fast and slow pointers

This problem can be solved efficiently using the fast and slow pointers technique, where the fast pointer moves two steps, and the slow pointer moves one step in the linked list.

If there is no cycle in the linked list, the fast pointer will reach the end of the linked list before the slow pointer. If there is a cycle, the fast pointer will eventually catch up to the slow pointer, since it is moving faster. When this happens, we know that we have a cycle in the linked list. The following steps can be performed to implement the algorithm above:

- We initialize two pointers, `fast` and `slow`, which point to the head of the linked list.
- We traverse the linked list using these two pointers. They move in the following way:
  - The `slow` pointer moves only one node forward in each iteration.
  - The `fast` pointer moves two nodes forward in each iteration, which means that it skips a node.
- If the `fast` pointer becomes NULL, we have reached the end of the linked list. Since no cycle exists, return FALSE.
- If both `slow` and `fast` pointers become equal to each other, return TRUE, since a cycle exists in the linked list.

Here's a demonstration of the algorithm above:



Here's the coded solution:

Java

main.java

LinkedListNode.java

LinkedList.java

PrintList.java

```
1  import java.util.*;
2
3  class CycleDetection {
4    public static boolean detectCycle(LinkedListNode head) {
5      if(head == null){
6        return false;
7      }
8      // Initialize two pointers, slow and fast, to the head of the linked list
9      LinkedListNode slow = head;
10     LinkedListNode fast = head;
11
12     // Run the loop until we reach the end of the
13     // linked list or find a cycle
14     while (fast != null && fast.next != null) {
15       // Move the slow pointer one step at a time
```

```
19
20        // If there is a cycle, the slow and fast pointers will meet
21        if (slow == fast) {
22          return true;
23        }
24      }
25      // If we reach the end of the linked list and haven't found a cycle, retu
26      return false;
27    }
```

Linked List Cycle

Solution summary

To recap, the solution to this problem can be divided into the following three parts:

- Initialize both the slow and fast pointers to the head node.
- Move both pointers at different rates, i.e. the slow pointer will move one step ahead whereas the fast pointer will move two steps.
- If both pointers are equal at some point, we know that a cycle exists.

Time complexity

The time complexity of the algorithm is $O(n)$, where n is the number of nodes in the linked list.

Space complexity

The space complexity of the algorithm above is $O(1)$.

Back
Linked List Cycle

Next
Middle of the Linked L...

Mark as Completed