Grokking the Coding Interview: Patterns for Coding Questions

13% completed

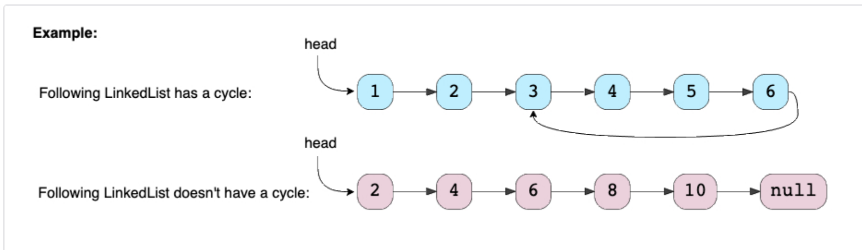
Search Course

LinkedList Cycle (easy)

We'll cover the following

Problem Statement

Given the head of a **Singly LinkedList**, write a function to determine if the LinkedList has a **cycle** in it or not.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
class Node {
  constructor(value, next=null){
    this.value = value;
    this.next = next;
  }
}

const has_cycle = function(head) {
  // TODO: Write your code here
  return false
}


head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)
head.next.next.next.next.next = new Node(6)
console.log(`LinkedList has cycle: ${has_cycle(head)}`)

head.next.next.next.next.next.next = head.next.next
console.log(`LinkedList has cycle: ${has_cycle(head)}`)

head.next.next.next.next.next.next = head.next.next.next
console.log(`LinkedList has cycle: ${has_cycle(head)}`)
```

RUN | SAVE | RESET

Close

Output 4.630s

```
LinkedList has cycle: false
LinkedList has cycle: false
LinkedList has cycle: false
```

Solution

Imagine two racers running in a circular racing track. If one racer is faster than the other, the faster racer is bound to catch up and cross the slower racer from behind. We can use this fact to devise an algorithm to determine if a LinkedList has a cycle in it or not.

Imagine we have a slow and a fast pointer to traverse the LinkedList. In each iteration, the slow pointer moves one step and the fast pointer moves two steps. This gives us two conclusions:

1. If the LinkedList doesn't have a cycle in it, the fast pointer will reach the end of the LinkedList before the slow pointer to reveal that there is no cycle in the LinkedList.
2. The slow pointer will never be able to catch up to the fast pointer if there is no cycle in the LinkedList.

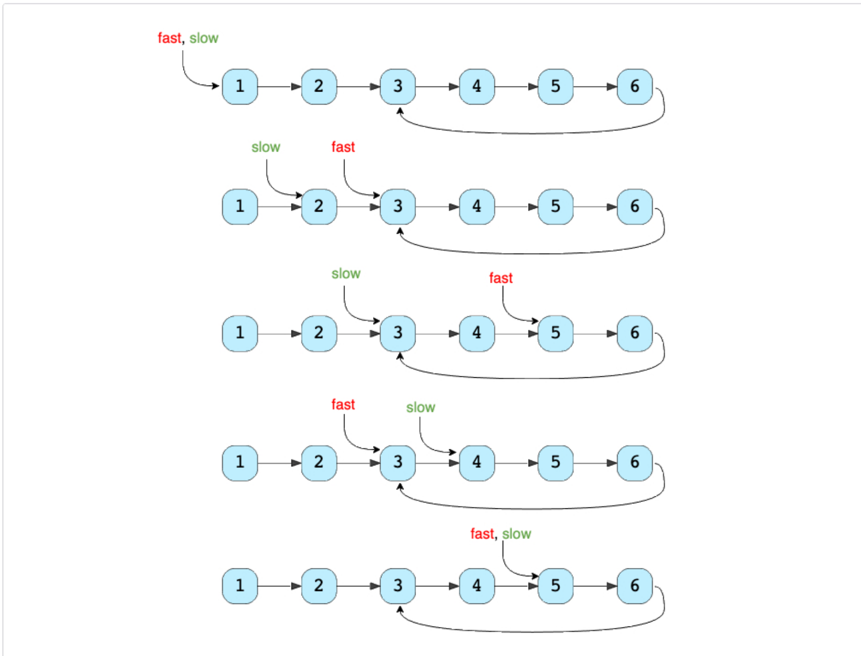
If the LinkedList has a cycle, the fast pointer enters the cycle first, followed by the slow pointer. After this, both pointers will keep moving in the cycle infinitely. If at any stage both of these pointers meet, we can conclude that the LinkedList has a cycle in it. Let's analyze if it is possible for the two pointers to meet. When the fast pointer is approaching the slow pointer from behind we have two possibilities:

1. The fast pointer is one step behind the slow pointer.
2. The fast pointer is two steps behind the slow pointer.

All other distances between the fast and slow pointers will reduce to one of these two possibilities. Let's analyze these scenarios, considering the fast pointer always moves first:

1. **If the fast pointer is one step behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step, and they both meet.
2. **If the fast pointer is two steps behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step. After the moves, the fast pointer will be one step behind the slow pointer, which reduces this scenario to the first scenario. This means that the two pointers will meet in the next iteration.

This concludes that the two pointers will definitely meet if the LinkedList has a cycle. A similar analysis can be done where the slow pointer moves first. Here is a visual representation of the above discussion:



Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
class Node {
  constructor(value, next = null) {
    this.value = value;
    this.next = next;
  }
}

function has_cycle(head) {
  let slow = head,
    fast = head;
  while (fast !== null && fast.next !== null) {
    fast = fast.next.next;
    slow = slow.next;
    if (slow === fast) {
      return true; // found the cycle
    }
  }
  return false;
}


const head = new Node(1);
head.next = new Node(2);
head.next.next = new Node(3);
head.next.next.next = new Node(4);
head.next.next.next.next = new Node(5);
head.next.next.next.next.next = new Node(6);
console.log(`LinkedList has cycle: ${has_cycle(head)}`);
```

RUN SAVE RESET

Close

Output 2.546s

```
LinkedList has cycle: false
LinkedList has cycle: true
LinkedList has cycle: true
```

Time Complexity

As we have concluded above, once the slow pointer enters the cycle, the fast pointer will meet the slow pointer in the same loop. Therefore, the time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space Complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1: Given the head of a LinkedList with a cycle, find the length of the cycle.

Solution: We can use the above solution to find the cycle in the LinkedList. Once the fast and slow pointers meet, we can save the slow pointer and iterate the whole cycle with another pointer until we see the slow pointer again to find the length of the cycle.

Here is what our algorithm will look like:

Java Python3 C++ JS

```
}


function calculate_cycle_length(slow) {
  let current = slow,
    cycle_length = 0;
  while (true) {
    current = current.next;
    cycle_length += 1;
    if (current === slow) {
      break;
    }
  }
  return cycle_length;
}


const head = new Node(1);
head.next = new Node(2);
head.next.next = new Node(3);
head.next.next.next = new Node(4);
head.next.next.next.next = new Node(5);
head.next.next.next.next.next = new Node(6);
head.next.next.next.next.next.next = head.next.next;
console.log(`LinkedList cycle length: ${find_cycle_length(head)}`);

head.next.next.next.next.next.next = head.next.next.next;
console.log(`LinkedList cycle length: ${find_cycle_length(head)}`);
```

RUN SAVE RESET

Time and Space Complexity: The above algorithm runs in $O(N)$ time complexity and $O(1)$ space complexity.

MARK AS COMPLETED

← Back
Introduction

Next →
Start of LinkedList Cycle (medium)

Report an Issue