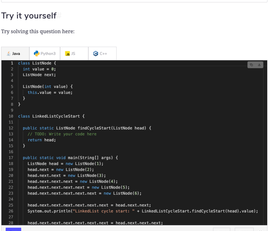
Back To Course Home

Grokking the Coding Interview: Patterns for Coding Questions

Start of LinkedList Cycle (medium)

Problem Statement

Given the head of a **Singly LinkedList** that contains a cycle, write a function to find the **starting node of the cycle**.



Try it yourself

Try solving this question here:

```
class ListNode {
  int value = 0;
  ListNode next;

  ListNode(int value) {
    this.value = value;
  }
}

class LinkedListCycleStart {

  public static ListNode findCycleStart(ListNode head) {
    // TODO: Write your code here
    return head;
  }

  public static void main(String[] args) {
    ListNode head = new ListNode(1);
    head.next = new ListNode(2);
    head.next.next = new ListNode(3);
    head.next.next.next = new ListNode(4);
    head.next.next.next.next = new ListNode(5);
    head.next.next.next.next.next = new ListNode(6);

    head.next.next.next.next.next.next = head.next.next;
    System.out.println("LinkedList cycle start: " + LinkedListCycleStart.findCycleStart(head).value);

    head.next.next.next.next.next.next = head.next.next.next;
  }
}
```

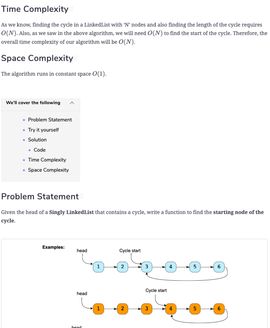
Run Save Reset

Solution

If we know the length of the **LinkedList** cycle, we can find the start of the cycle through the following steps:

1. Take two pointers. Let's call them `pointer1` and `pointer2`.
2. Initialize both pointers to point to the start of the LinkedList.
3. We can find the length of the LinkedList cycle using the approach discussed in LinkedList Cycle. Let's assume that the length of the cycle is 'K' nodes.
4. Move `pointer2` ahead by 'K' nodes.
5. Now, keep incrementing `pointer1` and `pointer2` until they both meet.
6. As `pointer2` is 'K' nodes ahead of `pointer1`, which means, `pointer2` must have completed one loop in the cycle when both pointers meet. Their meeting point will be the start of the cycle.

Let's visually see this with the above-mentioned Example-1:



We can use the algorithm discussed in LinkedList Cycle to find the length of the cycle and then follow the above-mentioned steps to find the start of the cycle.

Code

Here is what our algorithm will look like:

```
class ListNode {
  int value = 0;
  ListNode next;

  ListNode(int value) {
    this.value = value;
  }
}

class LinkedListCycleStart {

  public static ListNode findCycleStart(ListNode head) {
    int cycleLength = 0;
    // find the LinkedList cycle
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
      fast = fast.next.next;
      slow = slow.next;
      if (slow == fast) { // found the cycle
        cycleLength = calculateCycleLength(slow);
        break;
      }
    }

    return findStart(head, cycleLength);
  }
```

Run Save Reset

Time Complexity

As we know, finding the cycle in a LinkedList with 'N' nodes and also finding the length of the cycle requires $O(N)$. Also, as we saw in the above algorithm, we will need $O(N)$ to find the start of the cycle. Therefore, the overall time complexity of our algorithm will be $O(N)$.

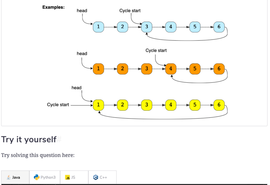
Space Complexity

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

We'll cover the following
- Problem Statement
- Try it yourself
- Solution
  - Code
- Time Complexity
- Space Complexity

Problem Statement

Given the head of a **Singly LinkedList** that contains a cycle, write a function to find the **starting node of the cycle**.



Try it yourself

Try solving this question here:

```
class ListNode {
  int value = 0;
  ListNode next;

  ListNode(int value) {
    this.value = value;
  }
}

class LinkedListCycleStart {

  public static ListNode findCycleStart(ListNode head) {
    // TODO: Write your code here
    return head;
  }

  public static void main(String[] args) {
    ListNode head = new ListNode(1);
    head.next = new ListNode(2);
    head.next.next = new ListNode(3);
    head.next.next.next = new ListNode(4);
    head.next.next.next.next = new ListNode(5);
    head.next.next.next.next.next = new ListNode(6);

    head.next.next.next.next.next.next = head.next.next;
    System.out.println("LinkedList cycle start: " + LinkedListCycleStart.findCycleStart(head).value);

    head.next.next.next.next.next.next = head.next.next.next;
  }
}
```

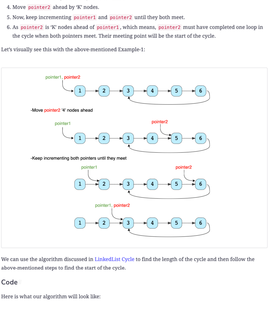
Run Save Reset

Solution

If we know the length of the **LinkedList** cycle, we can find the start of the cycle through the following steps:

1. Take two pointers. Let's call them `pointer1` and `pointer2`.
2. Initialize both pointers to point to the start of the LinkedList.
3. We can find the length of the LinkedList cycle using the approach discussed in LinkedList Cycle. Let's assume that the length of the cycle is 'K' nodes.
4. Move `pointer2` ahead by 'K' nodes.
5. Now, keep incrementing `pointer1` and `pointer2` until they both meet.
6. As `pointer2` is 'K' nodes ahead of `pointer1`, which means, `pointer2` must have completed one loop in the cycle when both pointers meet. Their meeting point will be the start of the cycle.

Let's visually see this with the above-mentioned Example-1:



We can use the algorithm discussed in LinkedList Cycle to find the length of the cycle and then follow the above-mentioned steps to find the start of the cycle.

Code

Here is what our algorithm will look like:

```
class ListNode {
  int value = 0;
  ListNode next;

  ListNode(int value) {
    this.value = value;
  }
}

class LinkedListCycleStart {

  public static ListNode findCycleStart(ListNode head) {
    int cycleLength = 0;
    // find the LinkedList cycle
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
      fast = fast.next.next;
      slow = slow.next;
      if (slow == fast) { // found the cycle
        cycleLength = calculateCycleLength(slow);
        break;
      }
    }

    return findStart(head, cycleLength);
  }
```

Run Save Reset

Time Complexity

As we know, finding the cycle in a LinkedList with 'N' nodes and also finding the length of the cycle requires $O(N)$. Also, as we saw in the above algorithm, we will need $O(N)$ to find the start of the cycle. Therefore, the overall time complexity of our algorithm will be $O(N)$.

Space Complexity

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