Linked List

Lecture Flow

- Pre-requisites
- Problem definitions and applications
- Types of linked list
- Different approaches
- Variants
- Recognizing in questions
- Common Pitfalls
- Practice Questions

Pre-requisites

1. Class
2. Two Pointer Technique
3. Linear Data Structure

Definitions

Linked List is a linear data structure that stores value and grows dynamically

Linked List consists of nodes where each node contains a data field and a reference(link) to the next node in the list

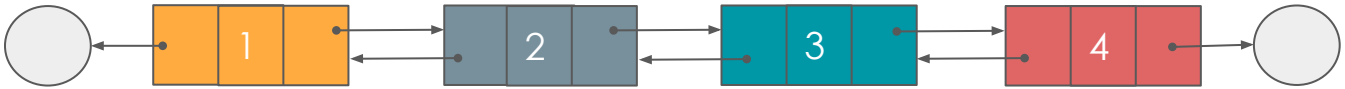
Node

- stores the value and the reference to the next node.
- The simplest linked list example

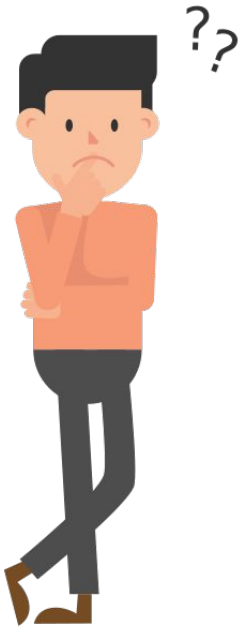
```
class Node:

    def __init__(self, value):

        self.value = value

        self.next = None # Type: Node
```

Singly Linked List is when nodes have only next's node reference.

Doubly Linked List is when nodes have both previous and next node reference.



Why do we need linked list when we have arrays?



Why Linked List when you have Arrays?

- Arrays by default don’t grow dynamically
- Inserting in the middle of an array is costly
- Removing elements from the middle of array is costly

Arrays vs Linked List

Array	Linked List
Fixed size	Dynamic size
Insertions and Deletions are inefficient	Insertions and Deletions are efficient
Random access	No random access
Possible waste of memory	No waste of memory
Sequential access is faster	Sequential access is slow

Problem Definition

A problem with requirement of O(1) deletion and insertion

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Constraints:

- `1 <= capacity <= 3000`
- `0 <= key <= 10^4`
- `0 <= value <= 10^5`
- At most `2 * 10^5` calls will be made to `get` and `put`.

How would we approach the problem with array?

What if we use linked lists?

Linked list can grow and shrink in constant time.

What about for the middle elements?

We can use a dictionary to give us the nodes in constant time.

So, problems you observed from arrays can be resolved here.

Questions?

Common Operations on Linked List

Linked List Class

- The LinkedList class serves as the container for the nodes.
- The __init__ method initializes an empty linked list with a head pointing to None.

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Traversing a Linked List

- Start with the head of the list. Access the content of the head node if it is not null
- Go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the last node)

Implementation

```
def traverse(self, head) -> None:
    currentNode = head
    while currentNode:
        print(currentNode.val)
        currentNode = currentNode.next
```

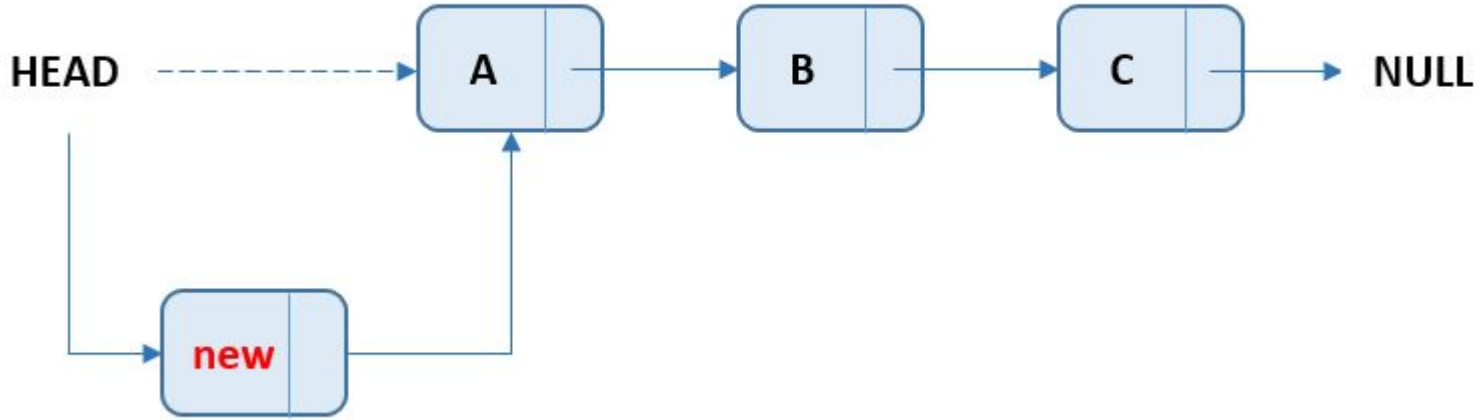
Pair Programming

Design Linked List(implement get)

Inserting a node in linked list

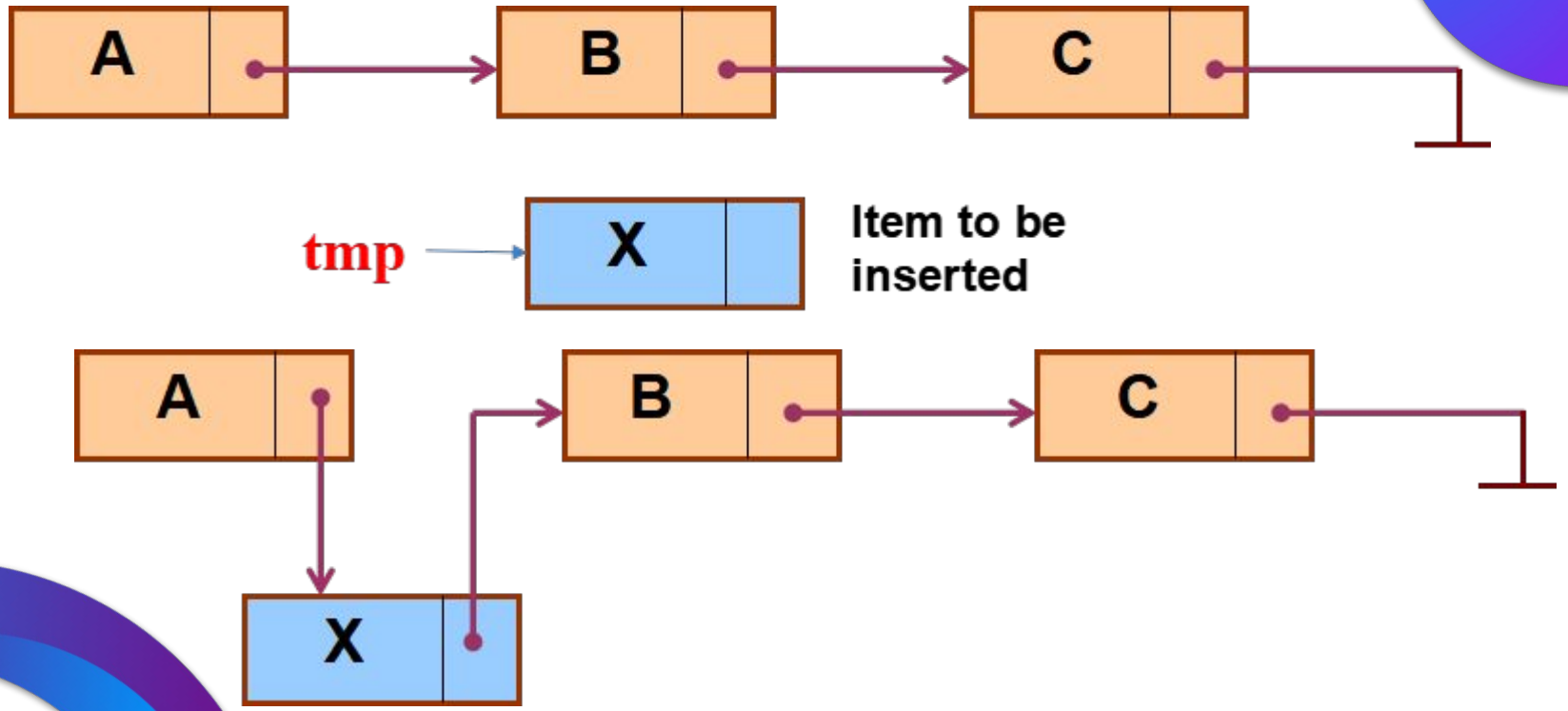
Insert at the beginning

- If list is empty
  - make new node the head of the list
- Otherwise
  - connect new node to the current head
  - make new node the head of the list.



Insert at any position

- Find the insert position and the previous node
- And then make the next of new node as the next of previous node
- Finally, make the next of the previous node the new node


A
B
C
tmp
X
Item to be inserted
A
B
C
X

Can we avoid using two approaches when we are inserting(beginning & any place)? How?

Yes, we can use Dummy Node.

Dummy Node

- is a node that points to the head of a linked list which will be discarded at the end
- When to use a Dummy Node?
  - if you are potentially modifying the head of linked list, use dummy node

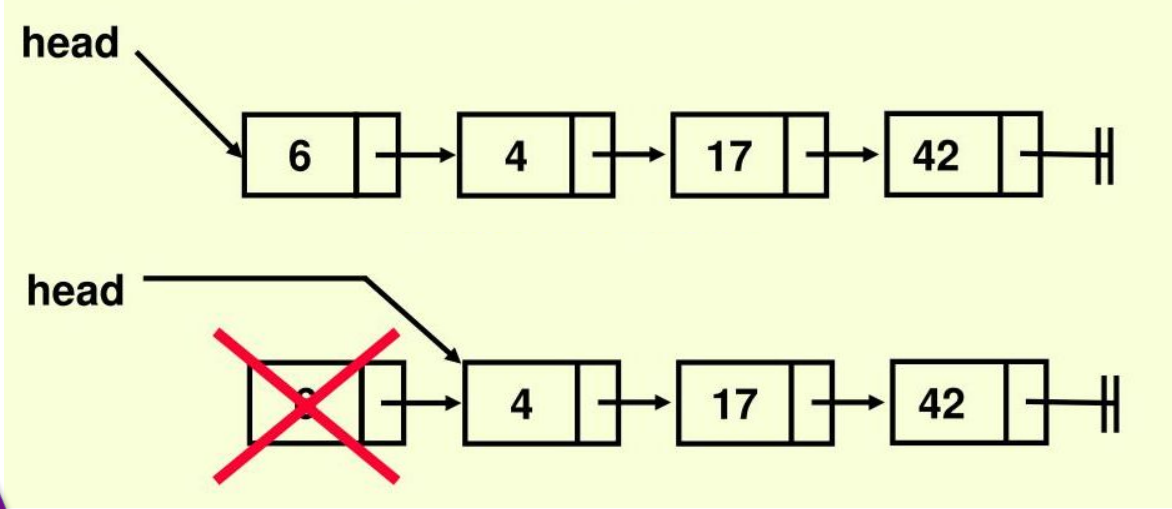
Pair Programming

Design Linked List(implement addAtAnyIndex)

Delete a node from the linked list

Delete a node at the beginning

- Make the second node as head
- Discard the memory allocated for the first node.

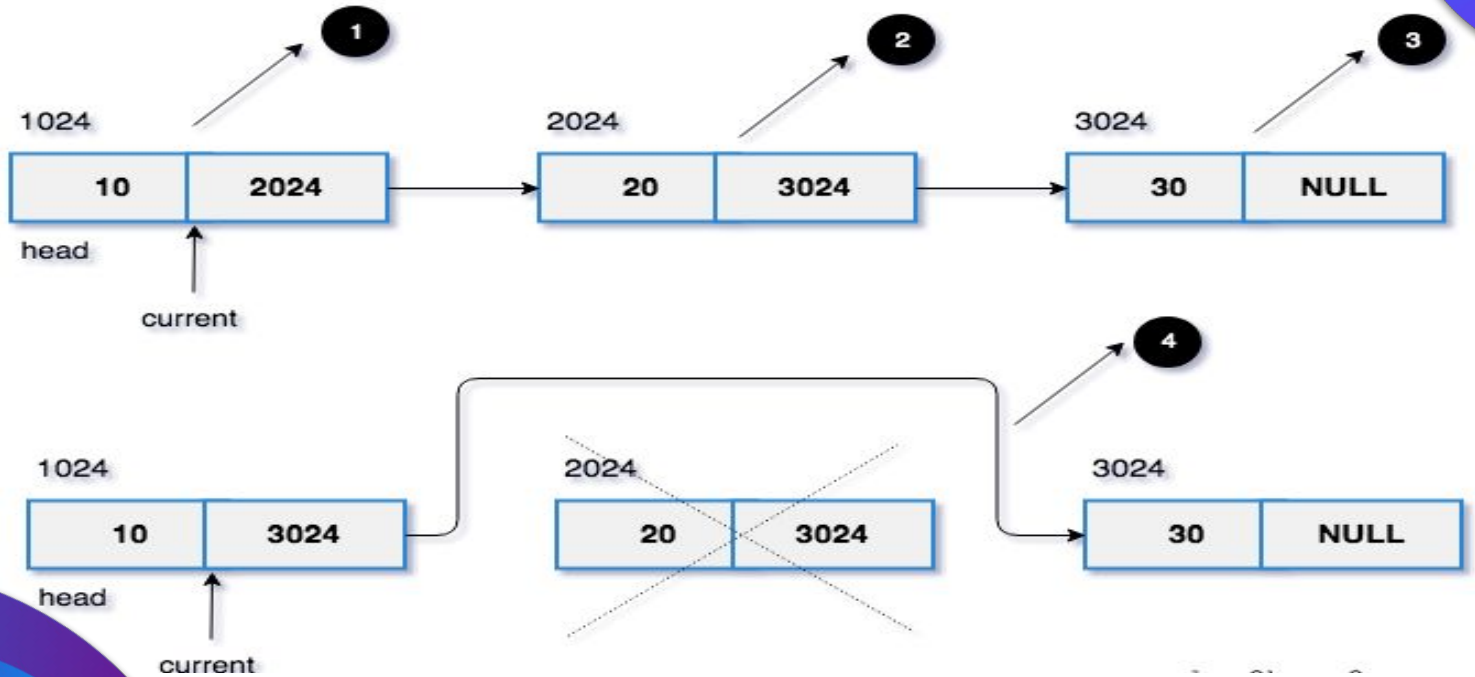


Implementation

```
def removeFirstNode(head):
    if not head:
        return None


    # Move the head pointer to the next node
    head = head.next
    return head
```

Delete a node at any position

- Find a match the node to be deleted
- Get the previous node
- Make the previous node next point to the next of the deleted node


1
2
3
1024
2024
3024
10
2024
20
3024
30
NULL
head
current
4
1024
2024
3024
10
3024
20
3024
30
NULL
head
current

Can we avoid using two approaches when we are deleting nodes? How?

Yes again! We can use Dummy Node.

Pair Programming

Design Linked List(implement deleteAtIndex)

Pros and Cons?

Advantages of using linked list

- Dynamic data structure
- No memory wastage
- Insertion and Deletion Operations

Disadvantages

- Traversal
- Reverse Traversing
- Random Access

Check Point

[Link]

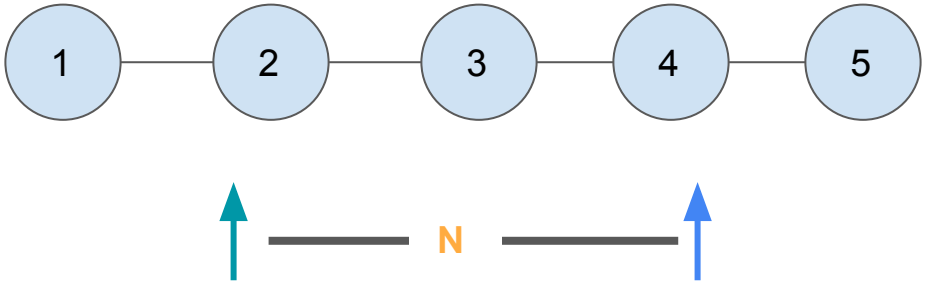
Two Pointer Technique in Linked List

Problem I

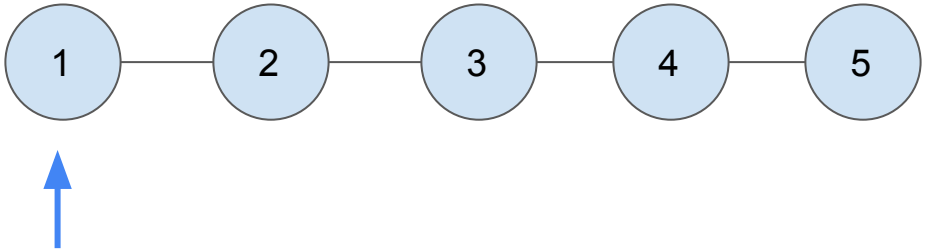
Return the n-th last element of a singly linked list.

Approach I

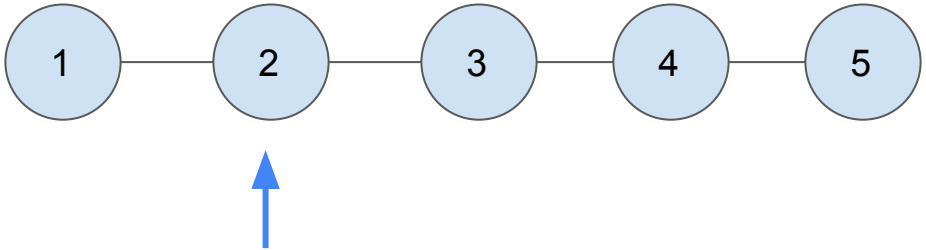
- iterate with two pointers
- one seeking the tail and the other lagging by the nth amount.



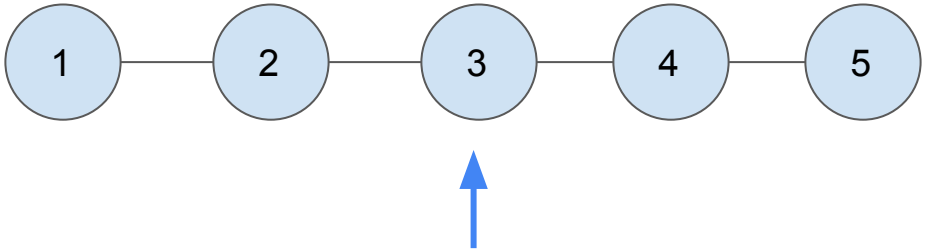
Simulation



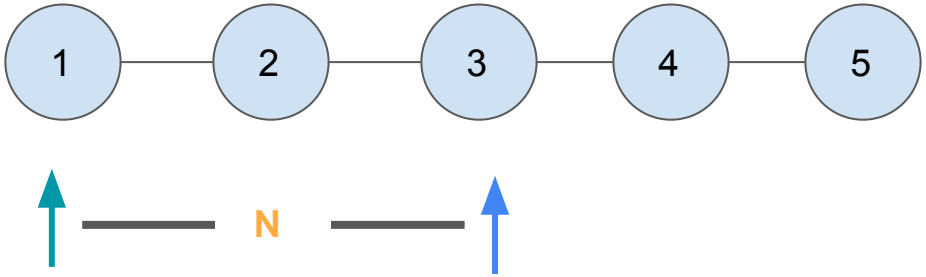
Simulation



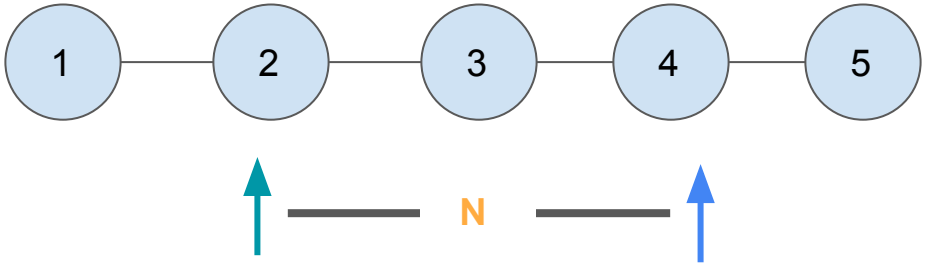
Simulation



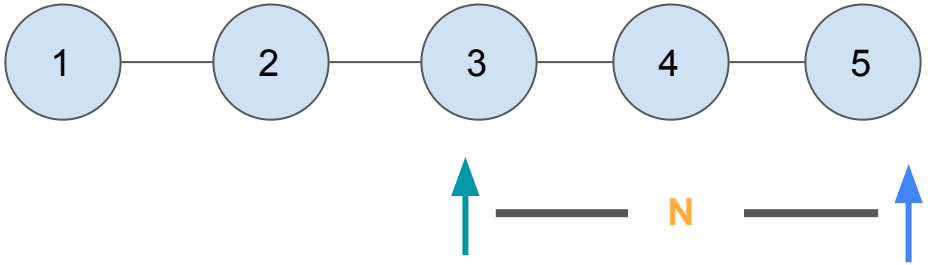
Simulation



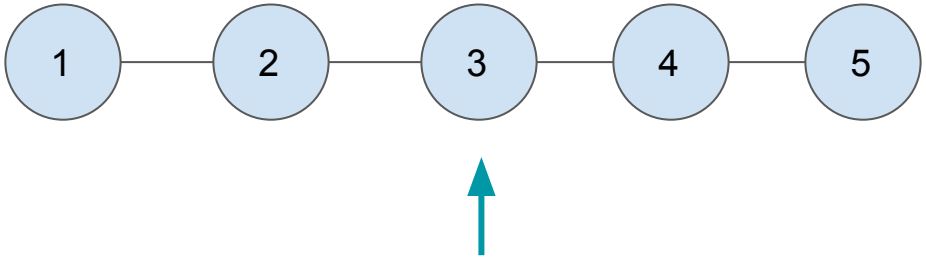
Simulation



Simulation



Simulation



Nth element from right

Implementation

```
def nthElementFromRight(self, head, n):
    # phase I
    aheadPtr = head
    while n > 0:
        aheadPtr = aheadPtr.next
        n -= 1
    # phase II
    behindPtr = head
    while behindPtr:        headptr
        behindPtr = behindPtr.next
        aheadPtr = aheadPtr.next
    return behindPtr
```

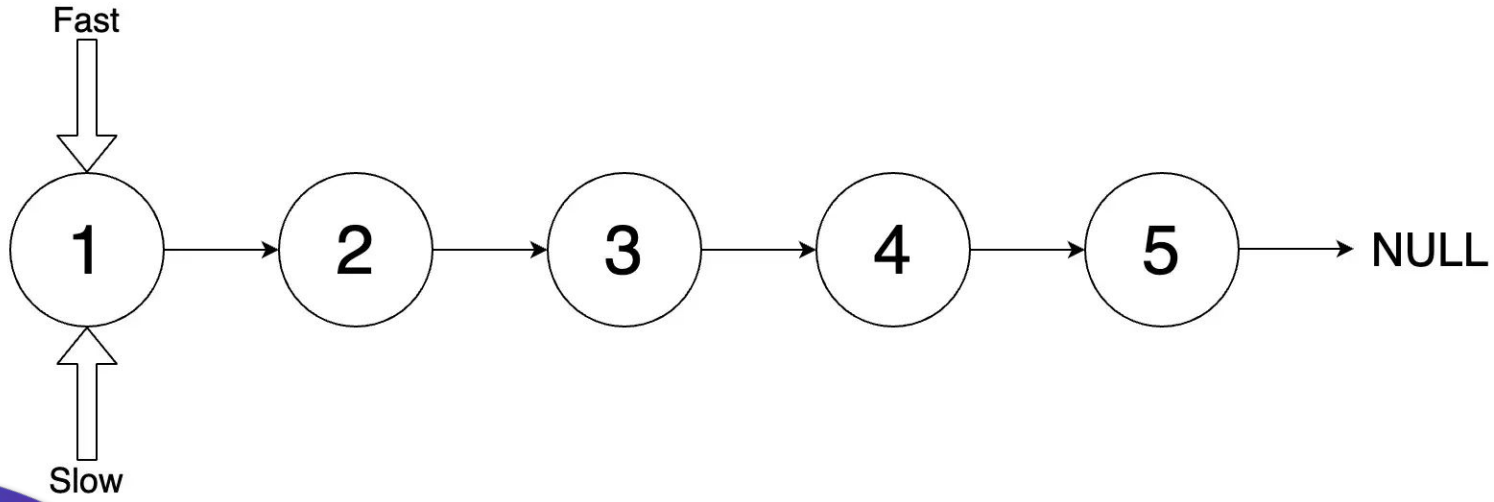
Approach : Fast and slow Pointers

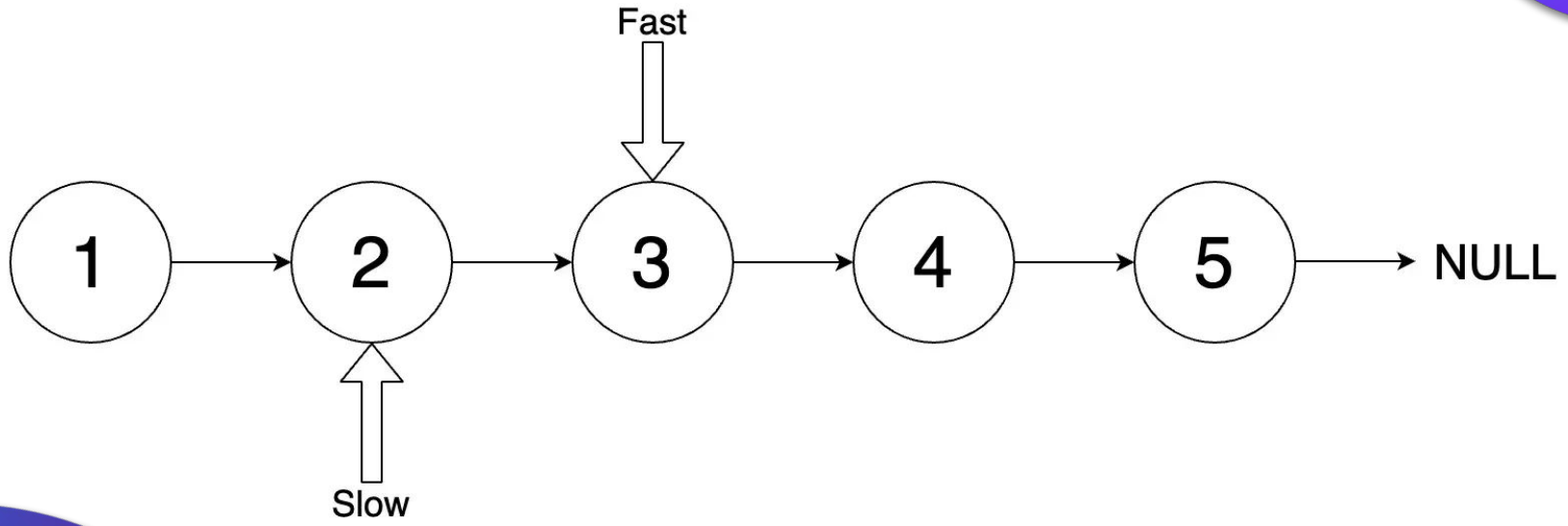
- Iterate with two pointers: Fast and slow
- Slow pointer goes one step at a time
- Fast goes two steps at a time
- When the fast pointer moves to the very end of the Linked List, the slow pointer is going to point to the middle element of the linked list.

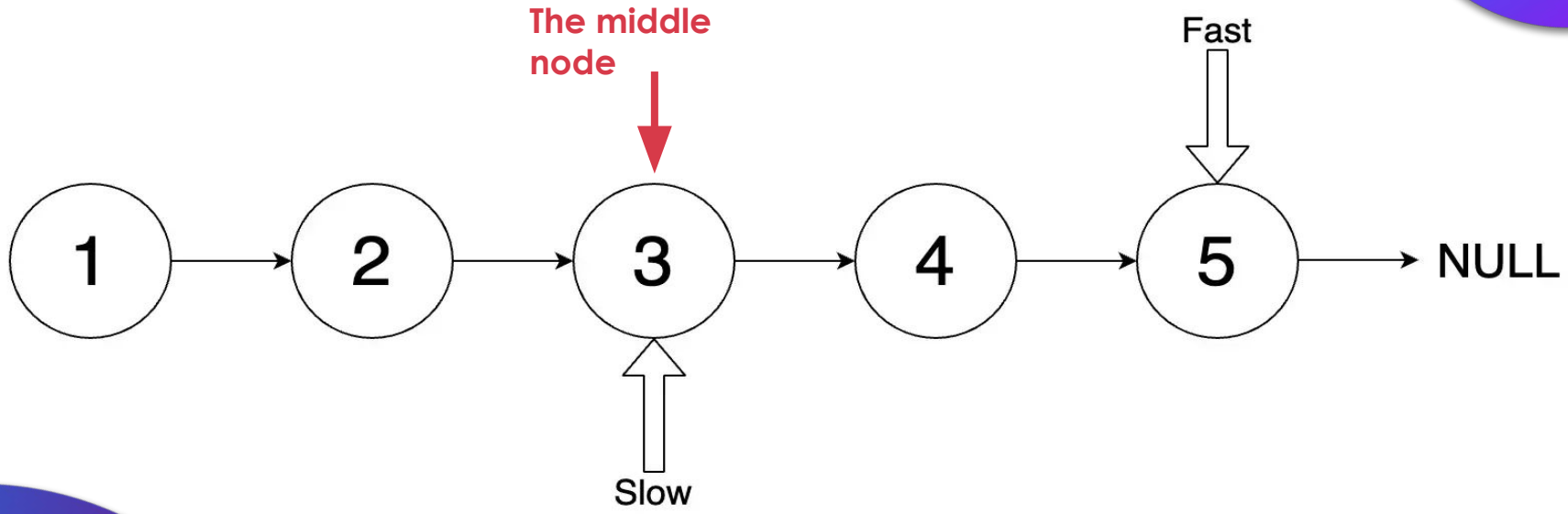
Problem II

Return the middle element of a linked List.

Simulation




Fast
1
2
3
4
5
NULL
Slow


The middle node
Fast
1
2
3
4
5
NULL
Slow

Pair Programming

[Middle element of a linked list]

Floyd’s Cycle Finding Algorithm

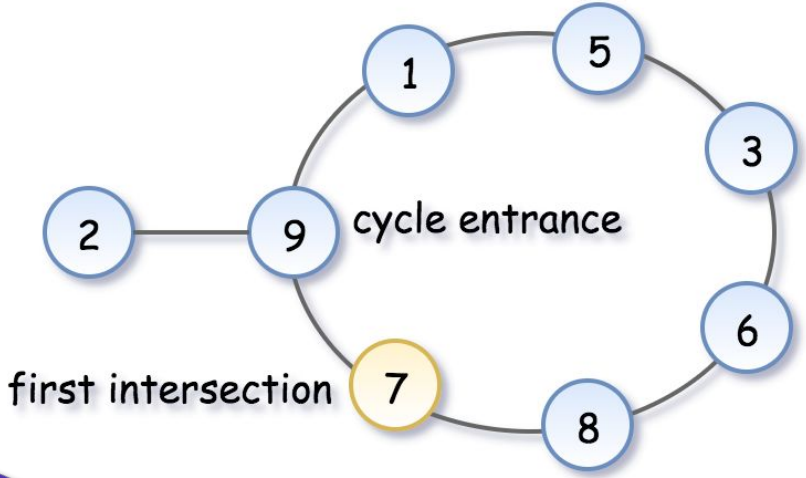
Problem

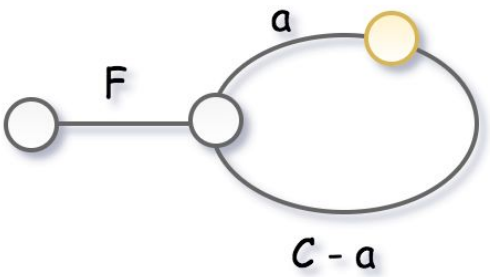
Given a linked list, return the node where the cycle begins?

Floyd's algorithm consists of two phases and uses two pointers, usually called tortoise and rabbit.

Phase I - Cycle Detection

- Rabbit = cur.next.next goes twice as fast as tortoise = cur.next
- Since tortoise is moving slower, the rabbit catches up to the tortoise at some *intersection* point
- Note that the intersection point is not the cycle entrance in the general case.


1
5
3
2
9
cycle entrance
6
first intersection
7
8


a
F
C - a

Phase II - Determine intersection point

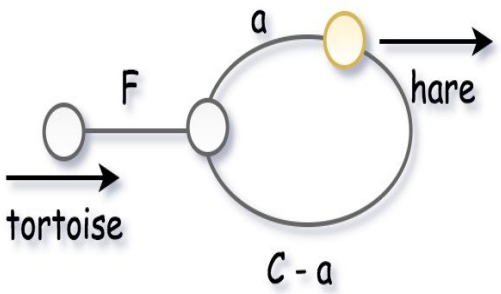
To compute the intersection point, let's note that the rabbit has traversed twice as many nodes as the tortoise, *i.e.*

$2 \cdot d_{tortoise} = d_{rabbit}$, implying:

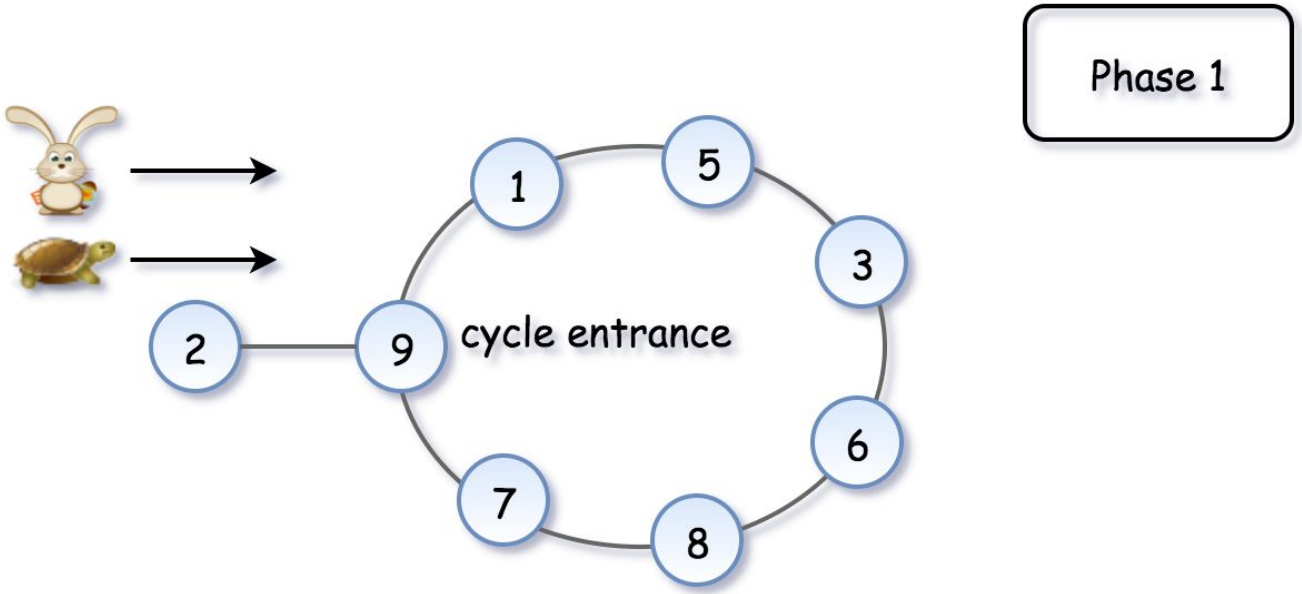
$2 \cdot (F + m \cdot C + a) = F + n \cdot C + a$, where n, m is some positive integer.

Hence the coordinate of the intersection point is $F = (n-2 \cdot m) \cdot C - a$

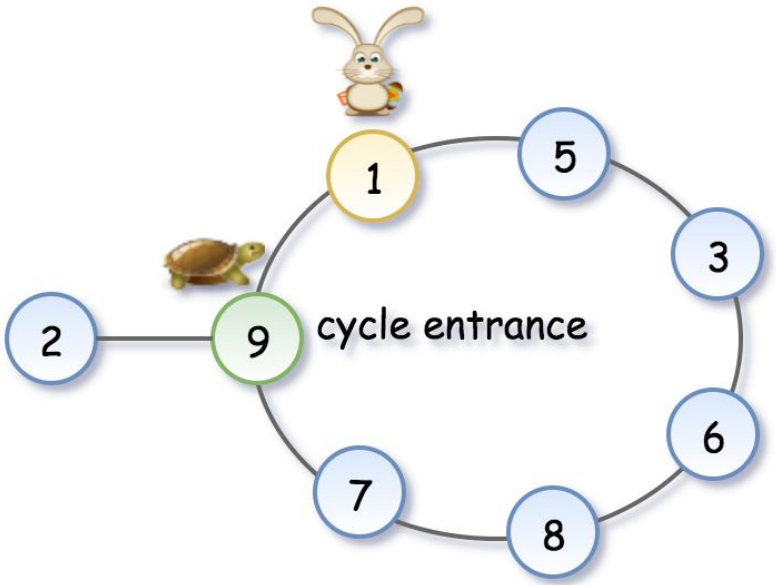
- we give the tortoise a second chance by slowing down the rabbit,
  - tortoise = tortoise.next
  - rabbit = rabbit.next
- The tortoise is back at the starting position, and the rabbit star



Phase 1

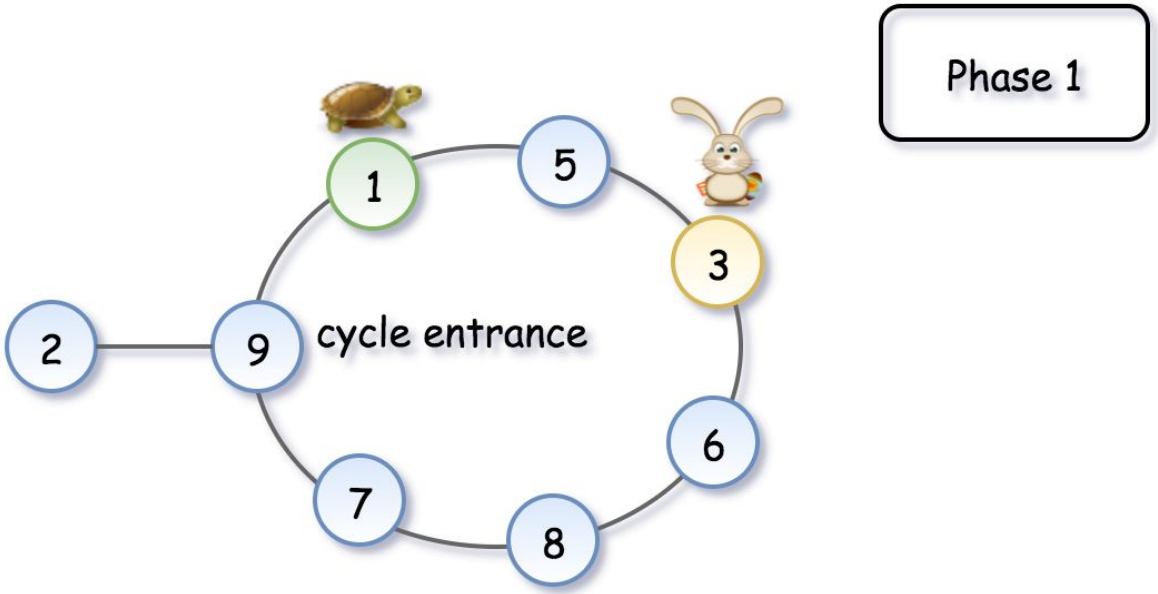


Phase 1

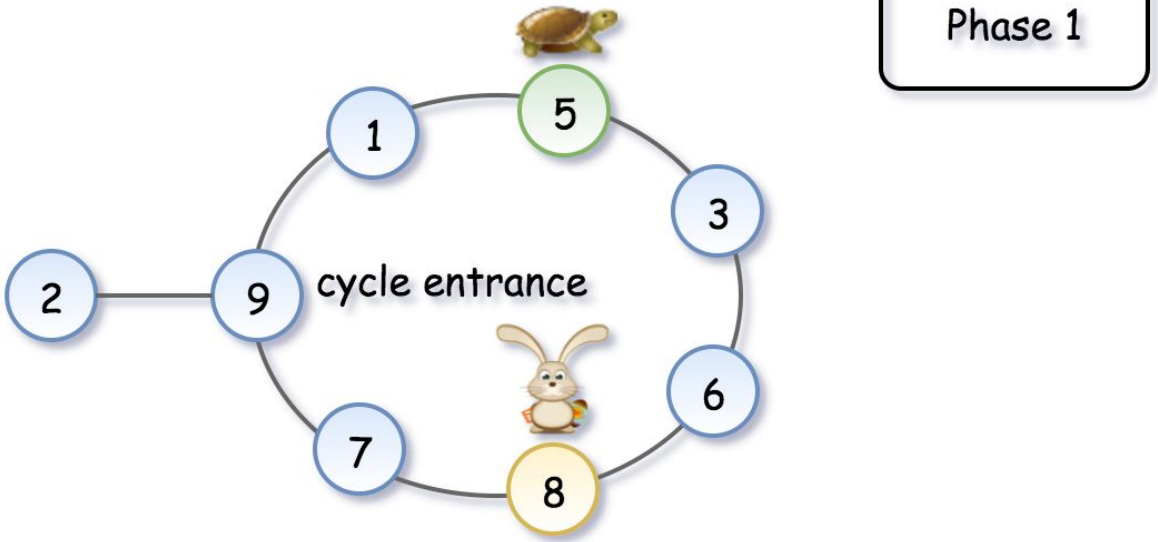


Phase 1

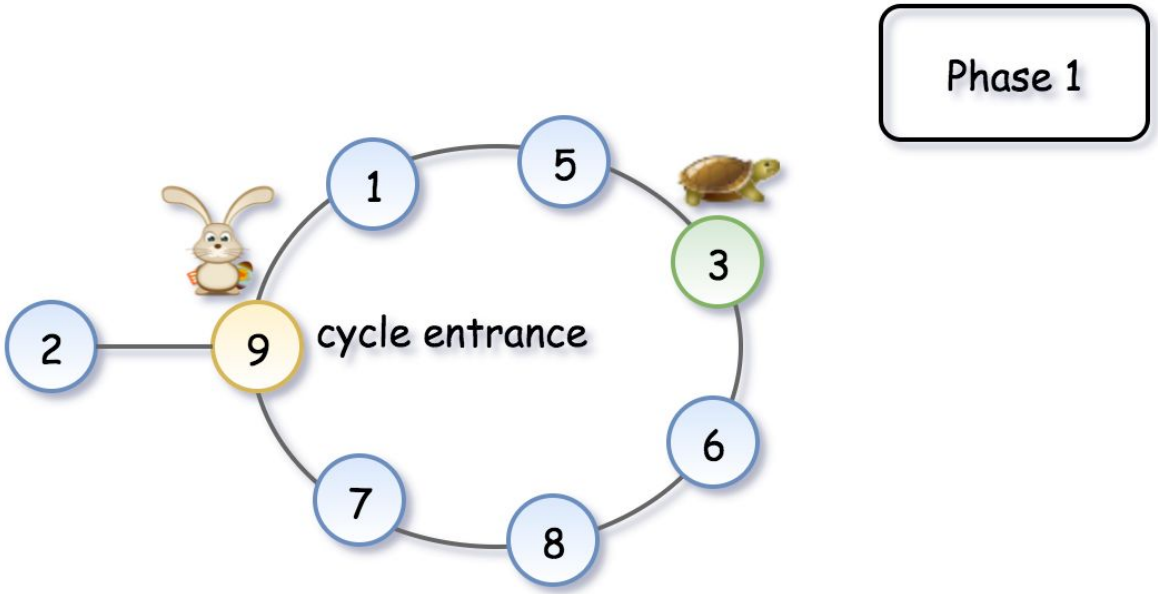
Phase 1



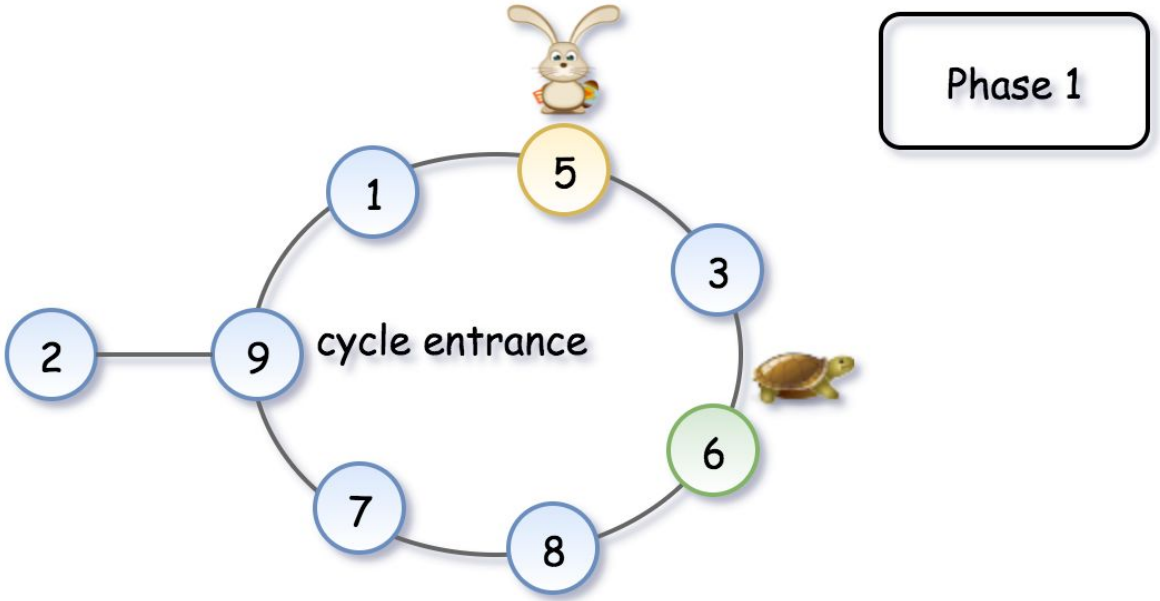
Phase 1



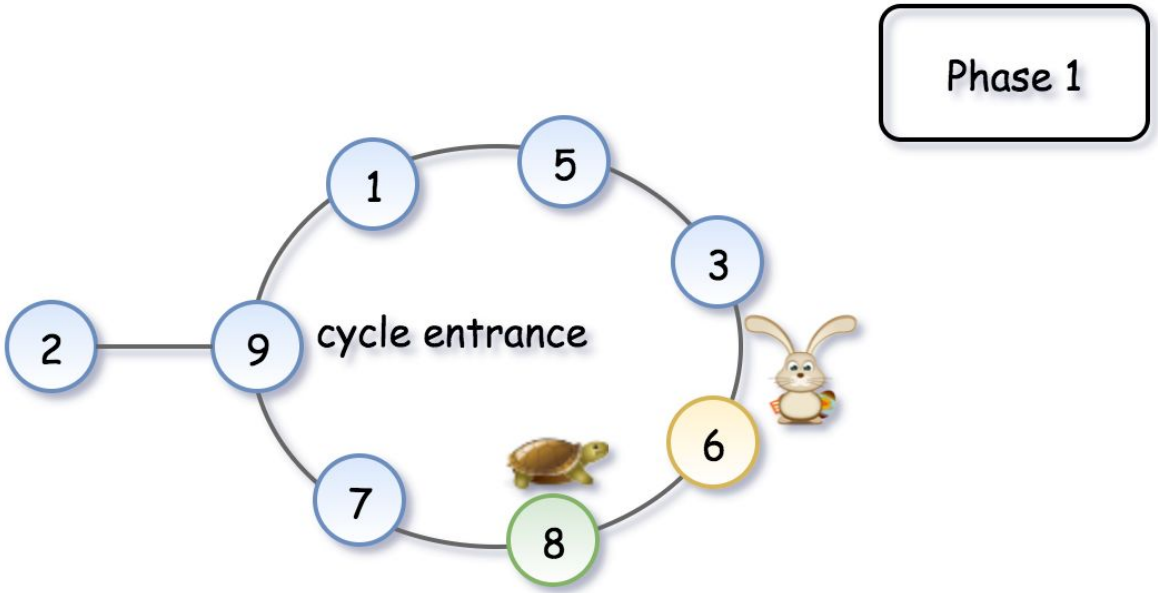
Phase 1



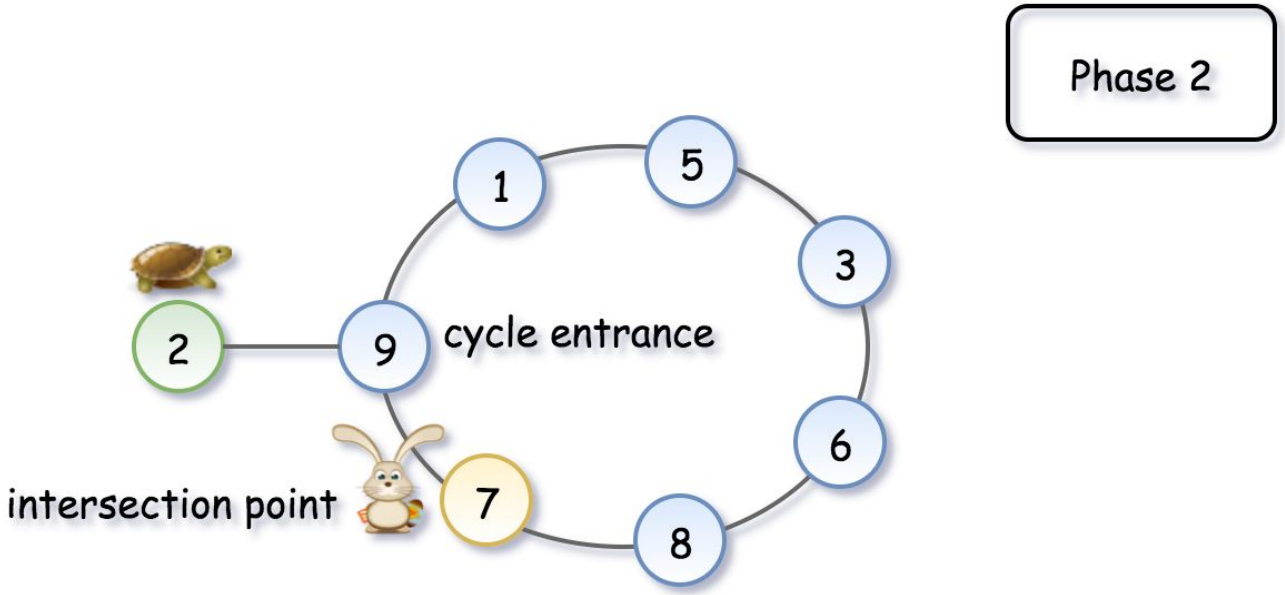
Phase 1



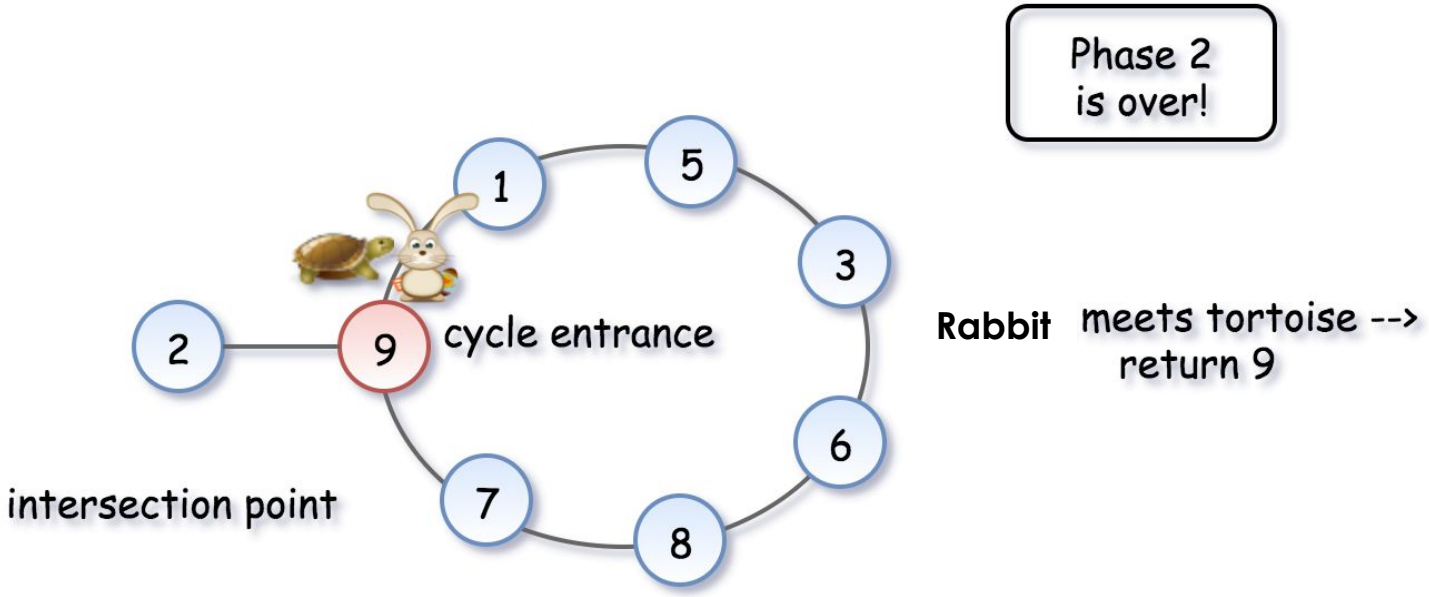
Phase 1



Phase 2



Phase 2 is over!



Pair Programming

[Linked List Cycle]

Things to pay attention

Can you spot the error?

```
def containsTarget(head, target):
    while head.val != target:
        head = head.next

    return head.val == target
```

Null Pointer Example

```
def containsTarget(head, target):
    while head.val != target:
        head = head.next

    return head.val == target
```

What if head is null?

Solution

```
def containsTarget(head, target):
    while head and head.val != target:
        head = head.next

    return head.val == target
```

Can you spot the error?

```
def middleNode(self, head):
    slow = head
    fast = head

    while fast:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Solution

```
def middleNode(self, head):
    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Pitfalls

- Null pointer
  - head.next <> check existence of head
  - head.next.next <> check existence of head and head.next

Pitfalls

```
def deleteNodeAtGivenPosition(position,head):
    if head is None:
        return
    index = 0
    previous = None
    while head.next and index < position:
        previous = head
        head = head.next
        index += 1
    previous.next = head.next
    return head
```

Losing the reference to head of the linked lis
Example

```
def deleteNodeAtGivenPosition(position,head):
    if head is None:
        return
    index = 0
    previous = None
    while head.next and index < position:
        previous = head
        head = head.next
        index += 1
    previous.next = head.next
    return head
```

Is head the head of the modified linked list?

Pitfalls - Continued

- Losing the reference to head of the linked list
    - Put the head in a variable before any operation to return the head at the end

Practice Questions

- Reverse Linked List
- Palindrome Linked List
- Merge Two Sorted Lists
- Remove duplicates from sorted list
- Partition List
- Linked List Cycle II
- LRU Cache
- Delete Node in a Linked List

Resources

- Leetcode Explore Card: has excellent track path with good explanations
- Leetcode Solution (Find the Duplicate Number) : has good explanation about Floyd's cycle detection algorithm with good simulation
- Elements of Programming Interview book: has a very good Linked List Problems set