Slow and Fast Pointer Technique

We maintain 2 pointers that move in the same direction, but one of the pointers makes larger strides as compared to the other. We make use of the fact that when the slower pointer points to the nth element, the faster pointer has made 2*n steps. This has applications in problems like finding the middle node of the linked list and detecting cycles.

Let's try to see how this works by the use of an Example.

Problem Statement: Given a linked list, find the middle node.

Example: Linked-List : 1->2->3->4->5

Output: 3

Approach:

We place two pointers simultaneously at the head node, each one moves at different paces, the slow pointer moves one step, and the fast moves two steps instead. When the fast pointer reaches the end, the slow pointer will stop in the middle. For the loop, we only need to check on the faster pointer, make sure fast pointer and fast.next is not NULL so that we can successfully visit the fast.next.next. When the length is odd, the fast pointer will point at the end node, because fast.next is NULL, when it is even, the fast pointer will point at None node, it terminates because fast is None.

```
function findMiddle(head)
        // If head is null just return null
        if head is null
                return head

        // If the Linked List has just 1 element that element is the middle
        if head->next is null
                return head
```

```
fast = head
slow = head

// moving the fast and slow pointer until the fast pointer reaches the end
while fast is not null and fast->next is not null
        fast = fast->next->next
        slow = slow->next

return slow
```

Time Complexity: O(n) Since the loop runs at most n/2 times to reach the middle node, the time complexity turns out to be O(n), where ‘n’ is the length of the linked list.

Space Complexity: O(1), Since we are using constant space to store the fast and slow pointers