# **Two-Pointer Traversal**

* We can use the two-pointer technique to iterate over a linked list.
* This allows us to solve a number of linked list related problems.
* For linked lists, we can either use

1. Slow-Fast Pointer Traversal (“Tortoise and Hare”)
2. Opposite End Pointer Traversal

# **Find the Middle of Linked List**

* Problem Statement:

Given the **head** of a singly linked list, return the***middle node of the linked list***.

* Constraints

The number of nodes in the list is in the range [1, 100].

1 <= Node.val <= 100

* Solution

We can use the slow-fast two-pointer technique.

We have two pointers, **slow** and **fast** that take 1 step forward and 2 steps forward, respectively, during each iteration of a loop.

The fast pointer will always reach the end of the list first, therefore we need a condition to check whether it has reached the end of the list.

This condition depends on whether the **list size is odd or even**:

1. **Even List Sizes**

There can be two possible middle nodes

* 1. if you want the left of the two nodes, check fast.next.next != null because this will make the loop stop one traversal early

ListNode slow = head;

ListNode fast = head;

while(fast.next != null && fast.next.next != null) // left of two

{

slow = slow.next;

fast = fast.next.next;

}

The fast.next != null accounts for an odd list, which stays the same.

The fast.next.next != null accounts for an even list, and allows the slow pointer to land on the left of the two middle nodes.

* 1. if you want the right of the two middle nodes, use the (fast.next != null) condition for the fast pointer just like how we do for odd sized linked lists.

// same as odd size code

ListNode slow = head;

ListNode fast = head;

while(fast != null && fast.next != null) // right of two

{

slow = slow.next;

fast = fast.next.next;

}

The fast.next != null accounts for an odd list, which stays the same.

The fast != null accounts for an even list, and allows the slow pointer to land on the right of the two middle nodes.

1. **Odd List Size**

You can use either algorithm to get the middle node of an odd length list.

The fast pointer will be at the last node either way.

ListNode slow = head, fast = head;

while(fast != null && fast.next != null)

{

slow = slow.next;

fast = fast.next.next;

}

or

ListNode slow = head, fast = head;

while(fast.next != null && fast.next.next != null)

{

slow = slow.next;

fast = fast.next.next;

}

Diagram

Description automatically generated

Try tracing both algorithms on this visualization of a list. It will work either way because the fast pointer will end up being on the last node.

Therefore, when choosing one of these methods (that only differ in the condition check), consider your use case on even lists.

# **Delete Middle of Linked List**

* Problem Statement

You are given the head of a linked list.

**Delete** the **middle node**, and return the head of the modified linked list.

The **middle node** of a linked list of size n is the ⌊n / 2⌋th node from the **start** using **0-based indexing**, where ⌊x⌋ denotes the largest integer less than or equal to x.

* For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.
* Constraints

The number of nodes in the list is in the range [1, 105].

1 <= Node.val <= 105

* Solution:
* We essentially use the same ‘find the middle node’ algorithm as we do for getting the middle node for odd list sizes or the right middle node for even list sizes.
* Since we need to delete this node, we need some way to keep track of the node before the middle node.
* We can do this by performing a late/early assignment of the slow pointer to a third ListNode pointer called pre.

ListNode pre;

ListNode slow = head, fast = head;

while(fast.next != null && fast.next.next != null)

{

pre = slow;

slow = slow.next;

fast = fast.next.next;

}