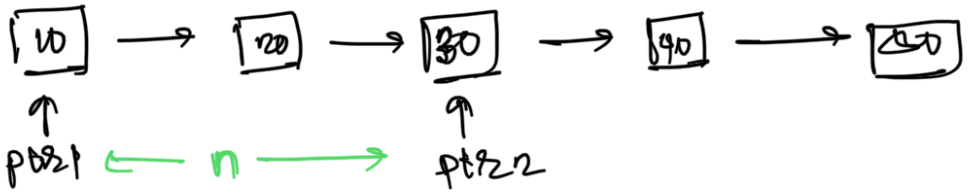
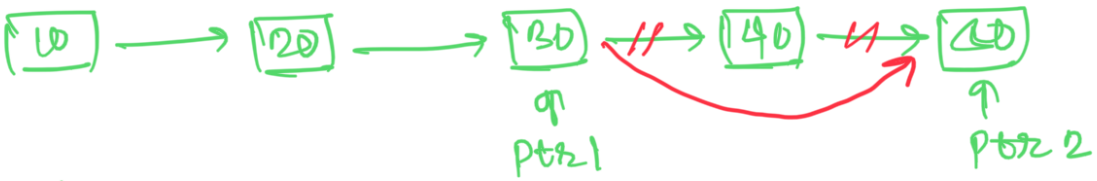
// Remove Nth node from the end of linked list

eg: n = 2

here n, we need to remove 40



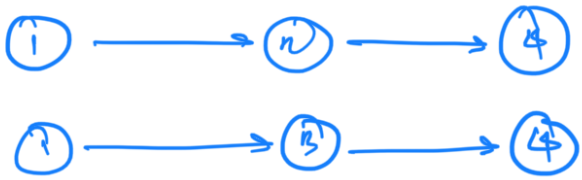
Iterate through till ptr2 reaches last node



Link ptr1 with ptr2


```
ptr1 = head
ptr2 = head
for (i in range n):
    ptr2 = ptr2.next
if ptr2.next == None:        // which means need to delete head.
    return head.next
while ptr2.next:
    ptr1 = ptr1.next
    ptr2 = ptr2.next
ptr2.next = ptr1.next.next
return head
```

2) Merge two sorted linked lists



To approach this kind of problem, start by creating a dummy node to avoid edge cases.

Output → dummy → 1 → 1 → 2 → 3 → 4 → 4

```
def mergeLists(l1, l2):
    dummy = ListNode()
    tail = dummy
    while l1 & l2:
        if l1.val > l2.val:
            tail.next = l2
            l2 = l2.next
        else:
            tail.next = l1
            l1 = l1.next
        tail = tail.next

    if l1:
        tail.next = l1
    elif l2:
        tail.next = l2
```

// condition in case of any list is exhausted.

---

3) Delete duplicates in sorted list

Approach is to change the ptrs whenever you encounter duplicate in the next pointer.

```
curr = head
while curr & curr.next:
    if curr.val == curr.next.val:
        curr.next = curr.next.next
```

```
else:
    curr = curr.next
return head
```

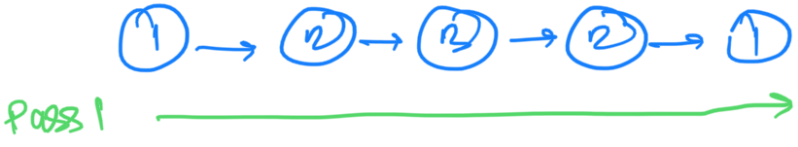
---

4) Palindrome Linked List

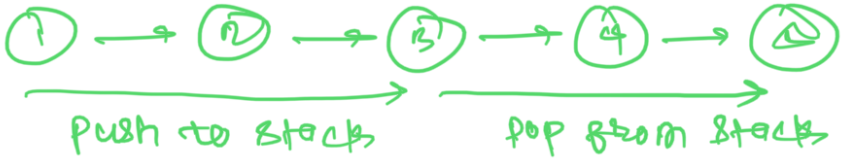
Approach 1

→ Iterate through the linked list to find the length.

→ Iterate through half of linked list store in stack, then for next pop the stack, if stack is empty. True.



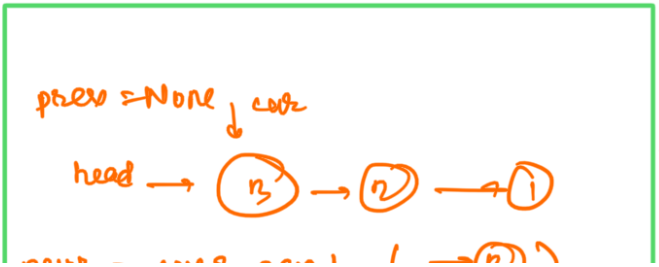
Pass 1 ——————————→ Length



Approach 2

mid

Reverse the pointers.



```
next = curr.next      (curr → ②)
curr.next = prev      (① → None)
prev = curr           (②)
curr = next           (curr = ②)
```

None ← ③ → ② → ①

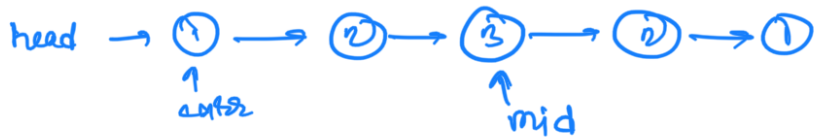
prev curr

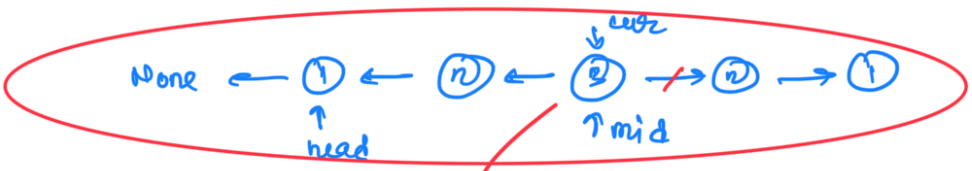
- find the mid pointer of linked list
- reverse the list from mid pointer.

```
curr = head
while last :
    if curr.val != last.val :
        return false.
    curr = curr.next
    last = last.next
return true.
```

Approach 2

find the mid & set ptr to mid





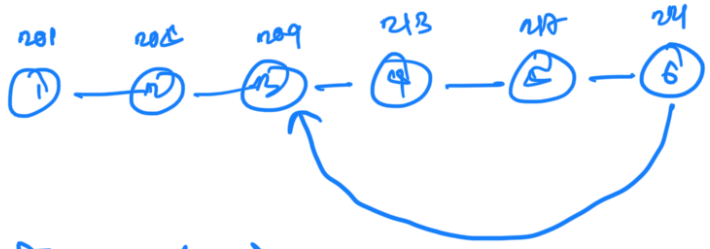
This approach will not works as you cansee ③ is started pointing in reverse, so we'll lose the pointer to the forward ②

---

2. Detect cycle in a Linked List

Approach 1

Approach - Iterate & store address in a set and also keep checking for existence while iterating. until reach Null or find a address which already exists.

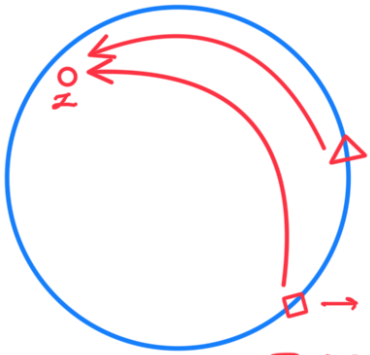


Set → { }

Iter	Set	Node
0	(201)	1
1	(201, 205)	2
2	(201, 205, 209)	3
⋮	⋮	⋮
6	(201, 205 ... 221)	6

after this iteration we'll encounter address 209 again, which means loop, is there

Approach



△ → running at x km/hr

□ → running at 2x km/hr

There will always be a point where △ & □ will meet. let's suppose point is z.

→ we will take two pointers fast & slow,
fast → 2x slow
then if there is a loop then fast & slow will surely meet.

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

if slow == fast:
    return True
return False.