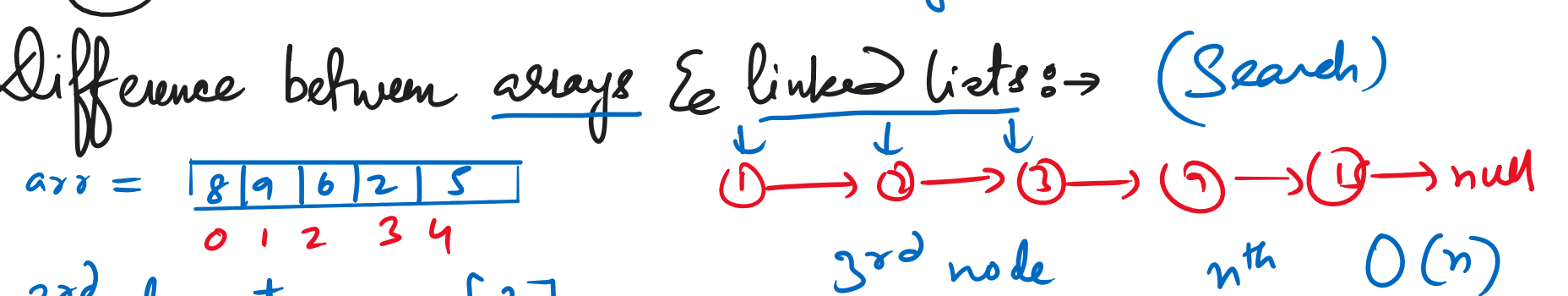
Introduction to Linked Lists :→

It is a linear data structure containing entities called **nodes**. Each node can have various types of connections/pointers to "next" or "previous" nodes.

Depending on the connections, there are three types :→

(i) Singly Linked Lists → Only forward traversal.
(ii) Doubly Linked Lists → Both ways traversal.
(iii) Circular Linked Lists → Cyclic traversal.

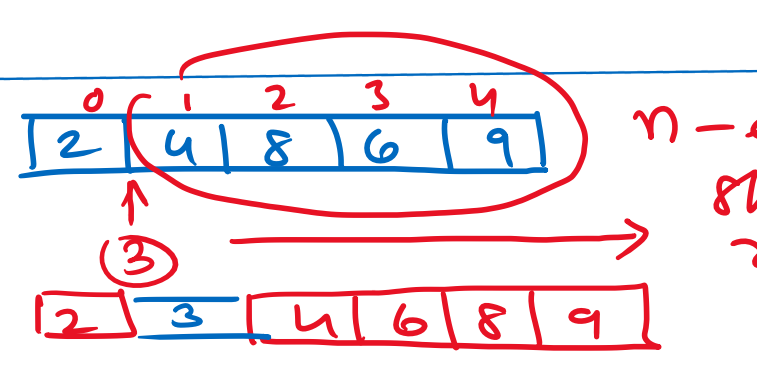
Difference between arrays & linked lists :→ (Search)



3rd element arr[2] — TC O(1) fast (indexing)

3rd node, nth — O(n) — TC O(n) slow

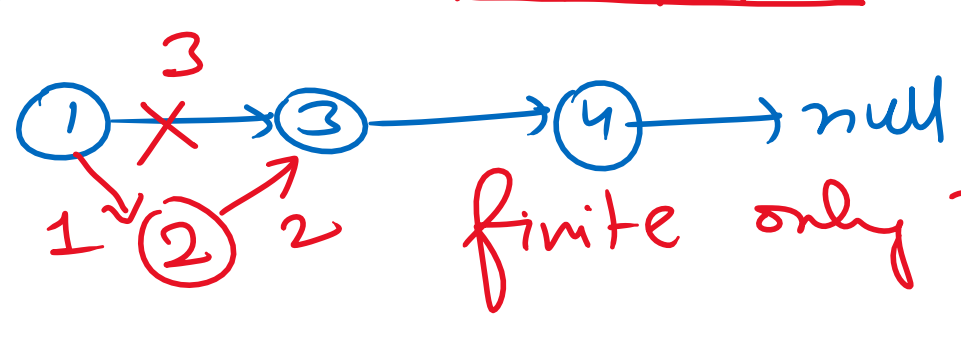
Insert : arr =



n-elements shifted to right

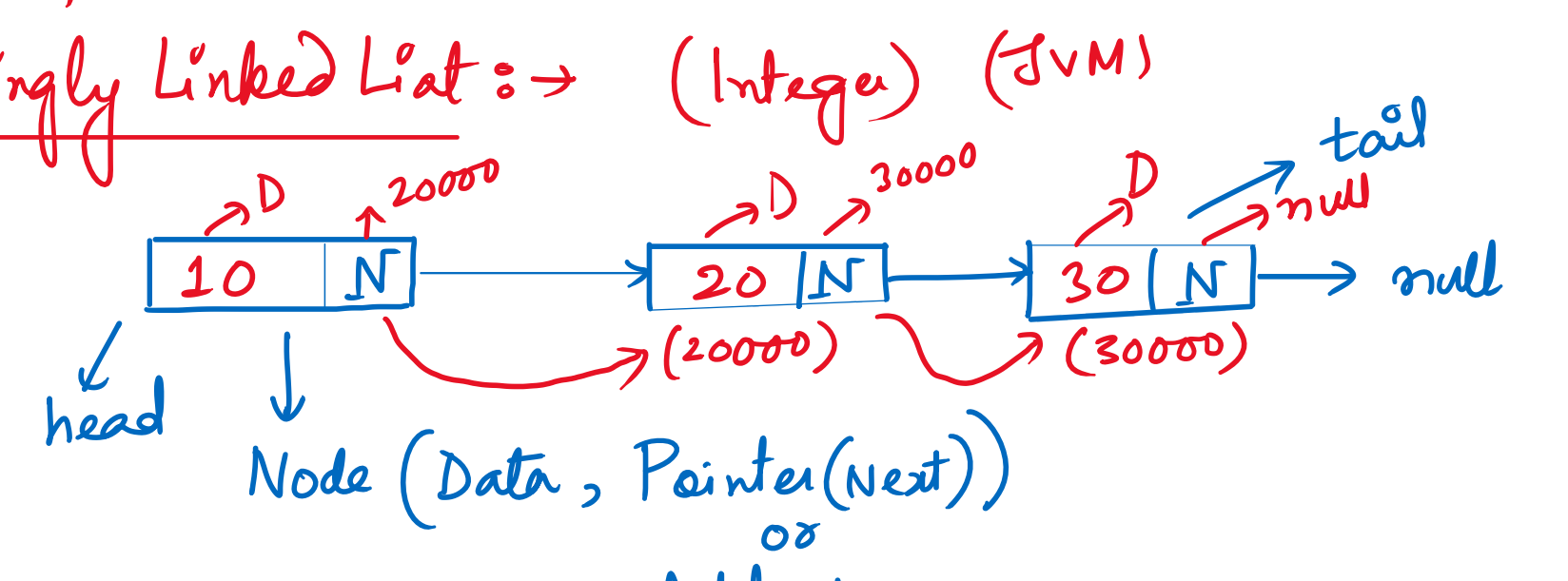
Slow O(n)

Linked List



O(1) fast — finite only 3 steps

Singly Linked List :→ (Integer) (JVM)



Node (Data, Pointer(Next) or Address)

* Each node has some data & the pointer to it's next node.
* The first node is the head node & the only way to traverse the list.
* The last node has no next node, so it's next pointer is "null".

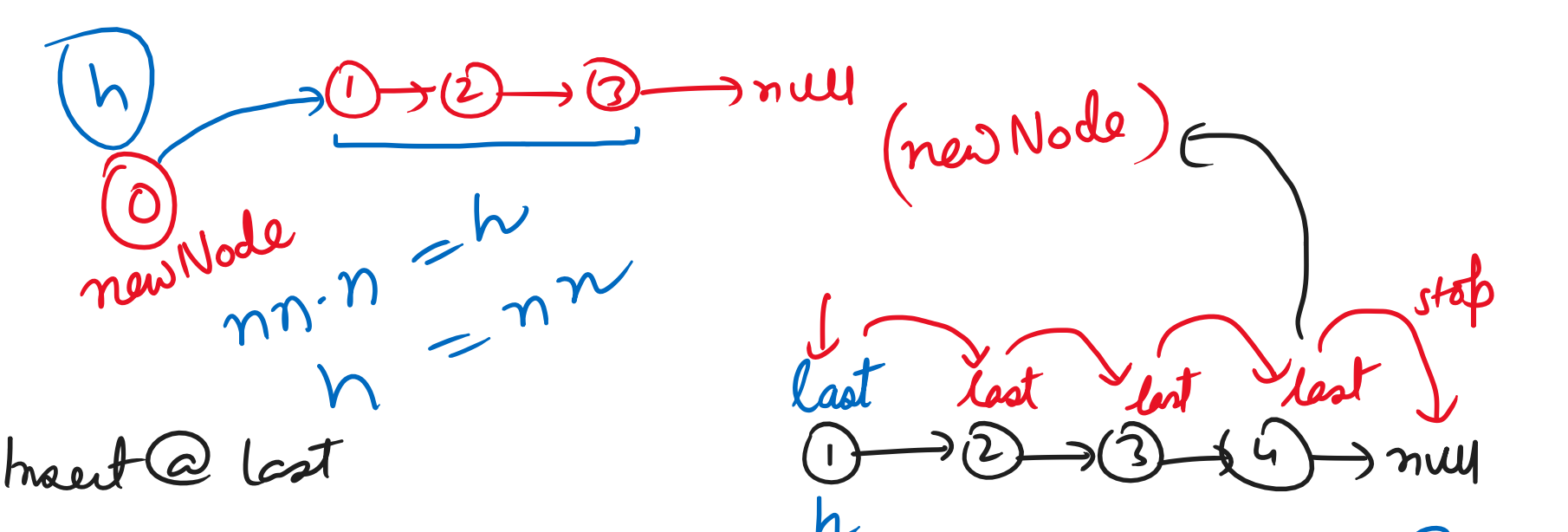
Important Operations:

(i) Insert → @ Head / Front; @ Tail / Back; After specific

(ii) Delete → @ Head; @ Tail; Target Node

(iii) Display

```
class Node {
    String data;
SLL Node next;
DLL Node prev;
```



nn.n = h
h = nn

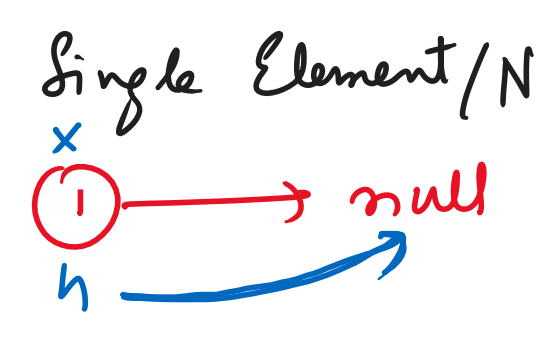
Insert @ last

```
while ( last.next != null ) {
    last = last.next;
}
last.next = newNode;
```

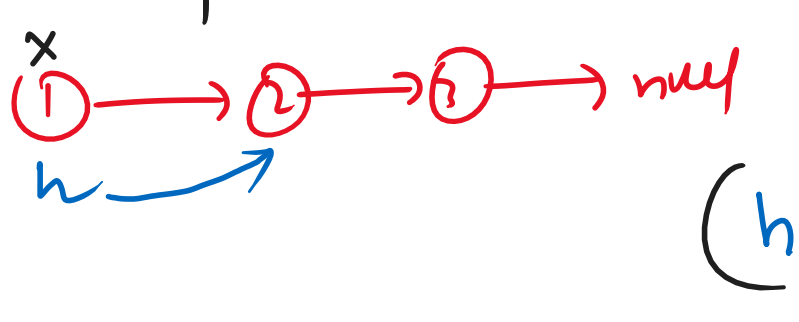
Delete the first node cases:

```
Node {
    String data;
    Node next;
}
next = null;
```

Case 1: Single Element/Node



Case 2: Multiple Nodes



this.head.next;

(h = h.next;)

(1) null ✓ Case 1
(2) (2) ✓ Case 2

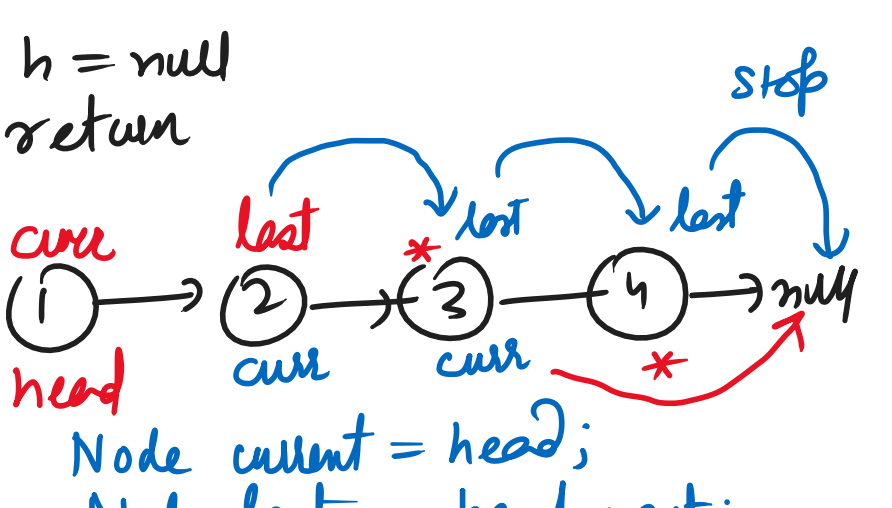
* **Delete last node cases:→**

(i) Single element

1 → null
h

h.next == null
h = null
return

(ii) Multiple nodes:
Two pointer approach.




```
Node current = head;
Node last = head.next;
while ( last.next != null ) {
    curr = last;
    last = last.next;
}
curr.next = null;
```

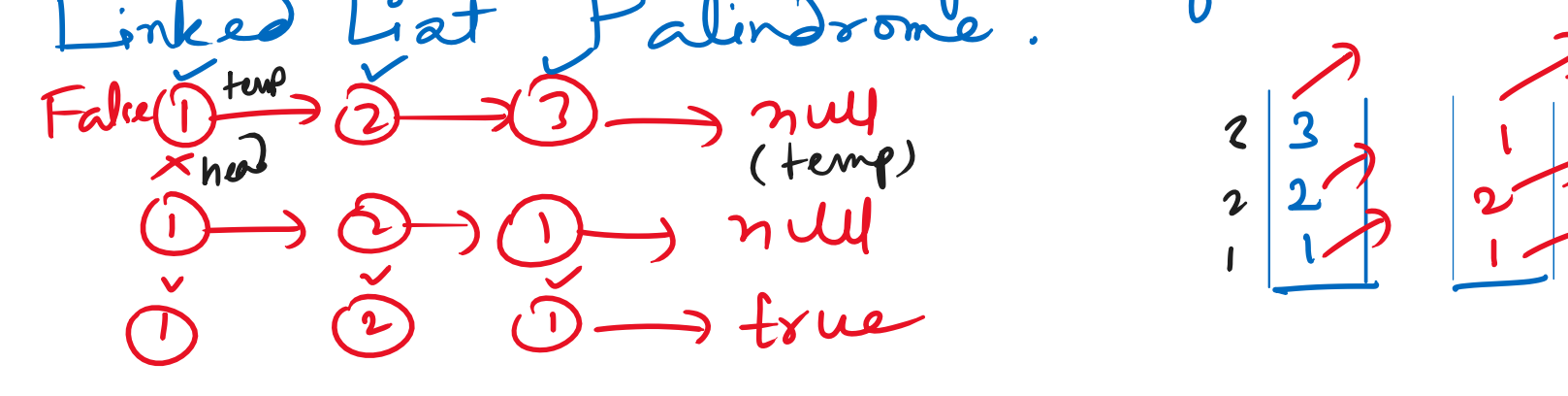
Done 64 JVM last = null — we won't do it.

Important Questions on Linked Lists :→

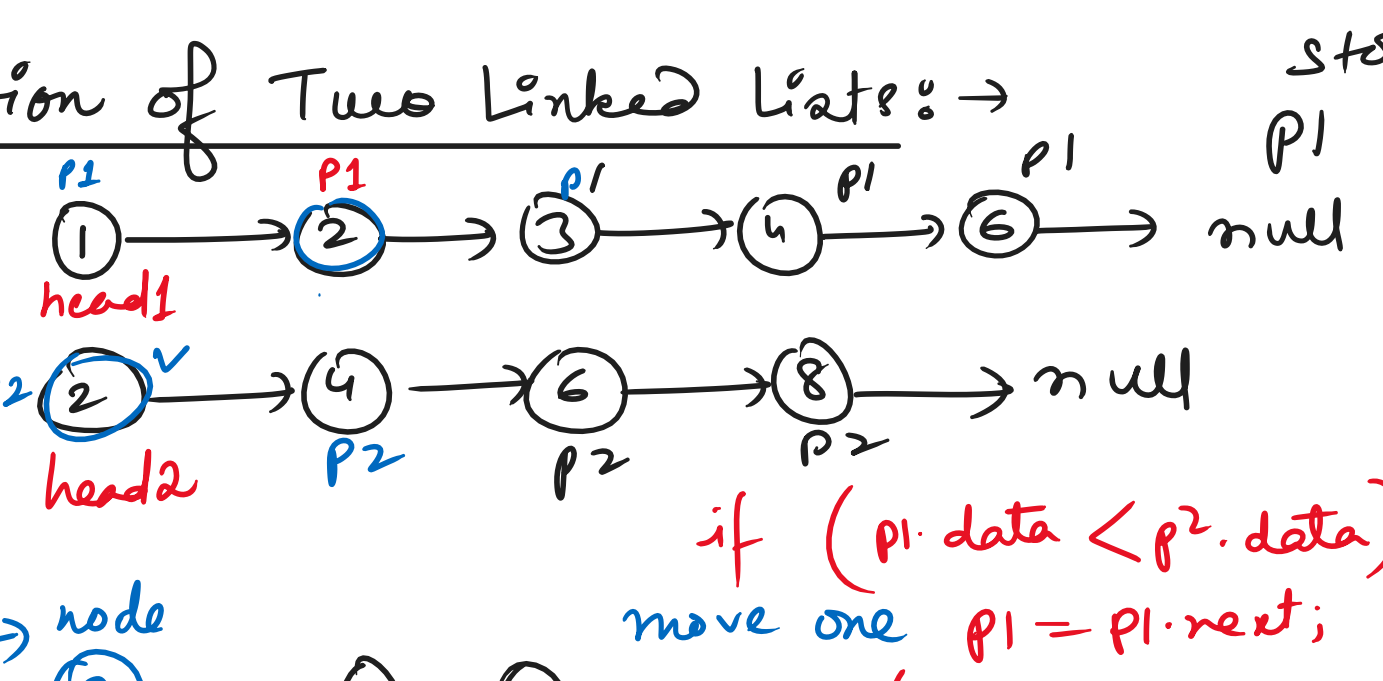
* Find the middle node of a linked list.
* Delete the middle node of a linked list.
* Detect cycle in a linked list.

} Floyd's Algorithm (Hare & Tortoise)

* Intersection of two linked lists.
* Add two numbers with carry using linked lists.
* Merge two sorted linked lists.
* Reverse a linked list.
* Reverse a linked list in groups of k.
* Linked List Palindrome.

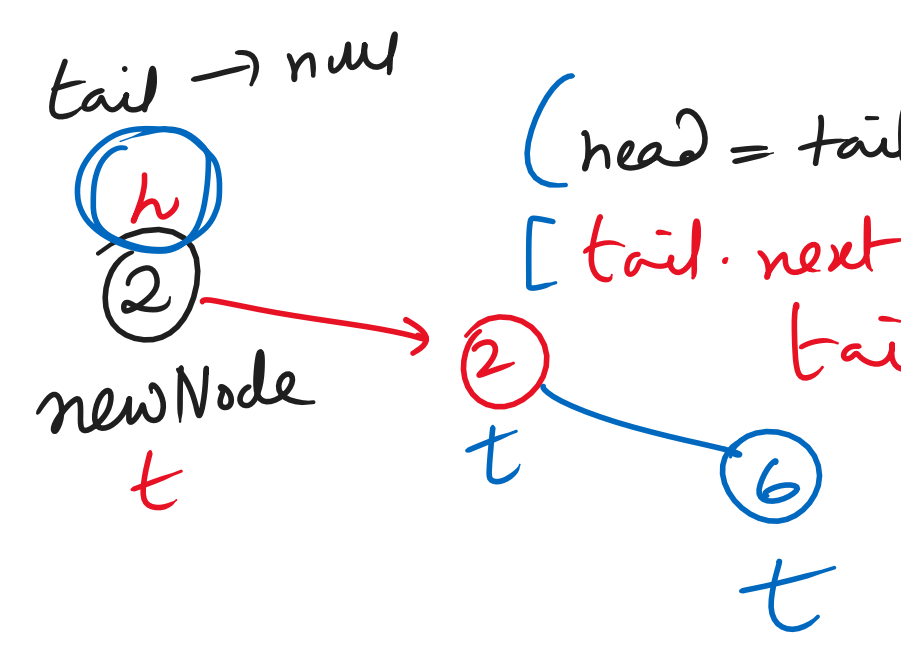


Intersection of Two Linked Lists :→



while p1 != null && p2 != null

```
if ( p1.data < p2.data )
    move one  p1 = p1.next;
else if ( p2.data < p1.data )
    move other  p2 = p2.next;
else (match found)
{ store in new list & move both }
    node.data (p2.data)
    p1 = p1.next;
    p2 = p2.next;
```



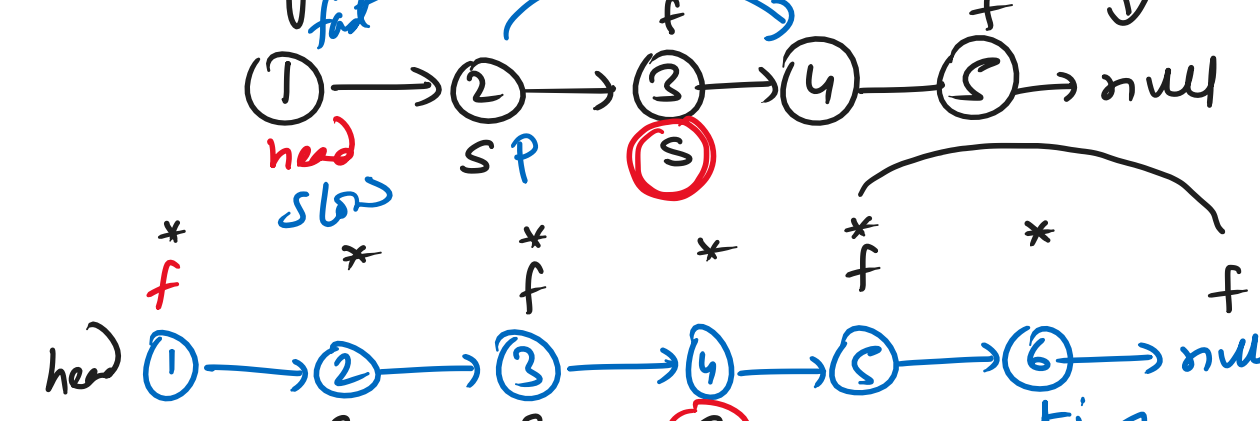
head == null (4) (6)

(head = tail = newNode)
[tail.next = newNode
tail = newNode]

* Floyd's Algorithm (Hare & Tortoise Algo)

* Middle of linked list
* Delete the middle node
* Cycle Detection

Two Pointer Approach
* Slow & Fast Pointers

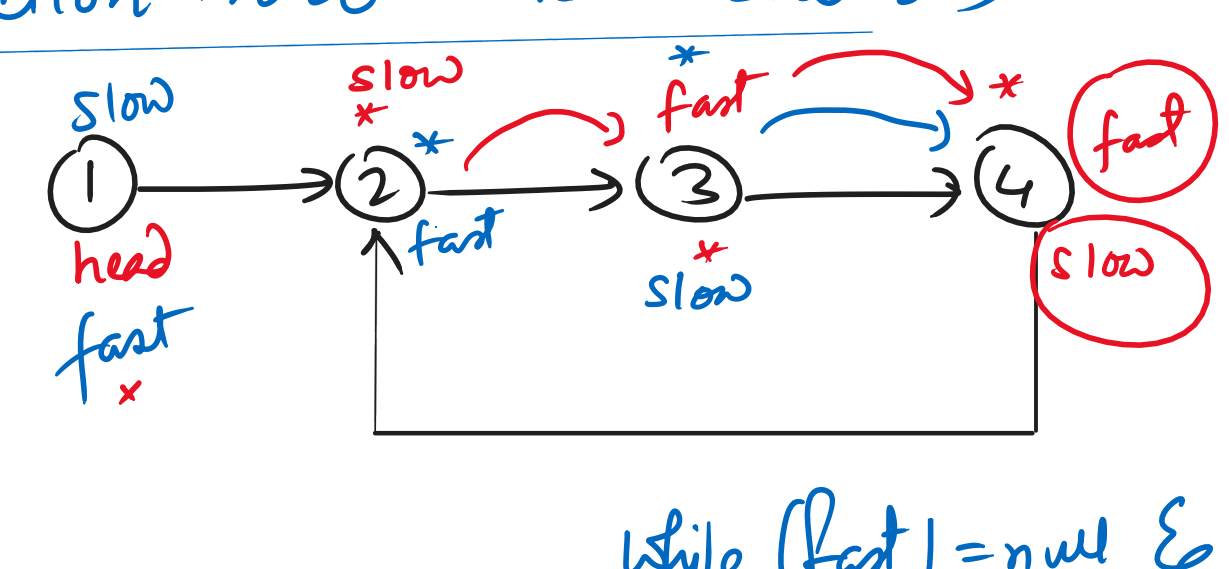


prev = slow
slow = fast
fast = f.n.n;

[prev = s.next;
slow = null;]

```
prev = null;
fast = head;
slow = head;
while (fast != null &&
       fast.next != null)
{
    slow = fast;
    fast = fast.next.next;
}
return slow;
```

Cycle Detection in a Linked List :→



④.next = ② cycle

if slow == fast → true or false.

```
while (fast != null &&
       fast.next != null)
{
  ① slow = slow.next;
  ② fast = fast.next.next;
}
```