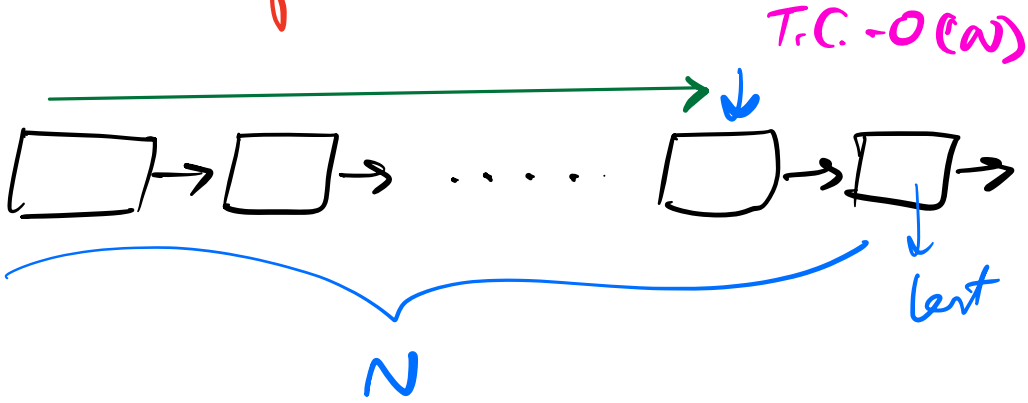
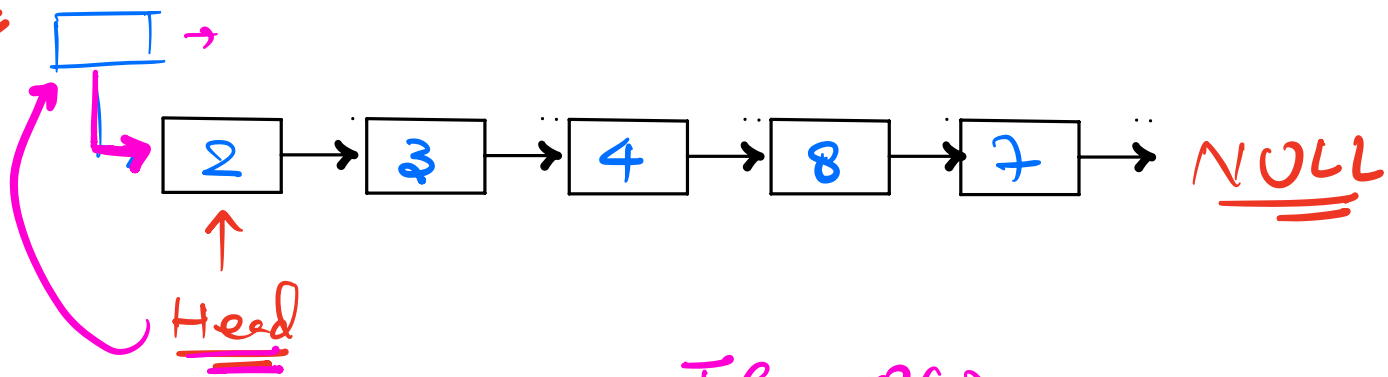
Agenda:
1) Merge two sorted LL
2) Merge Sort on LL
3) Cycle Detection
4) find starting node of cycle.

Quiz! T.C. of Delete.

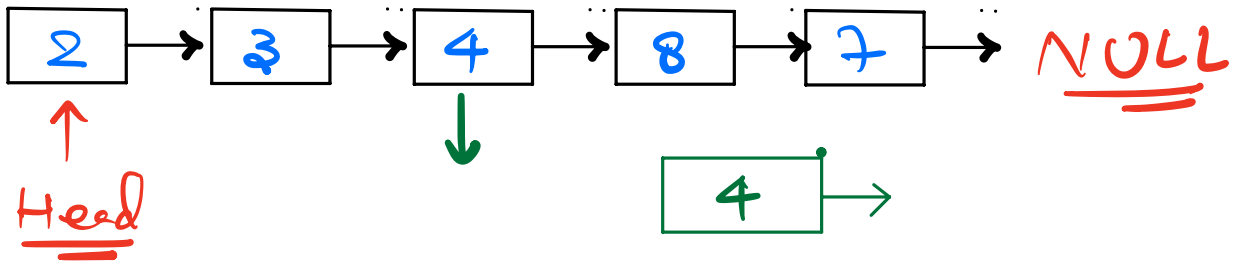


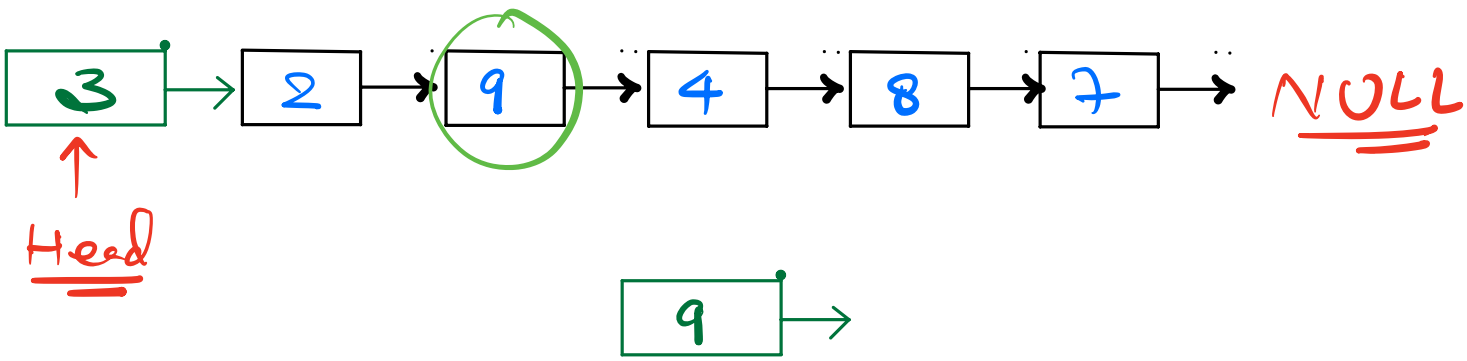
T.C. to insert node at Head.



T.C. = O(1)

Given a LL. find & return the middle node.



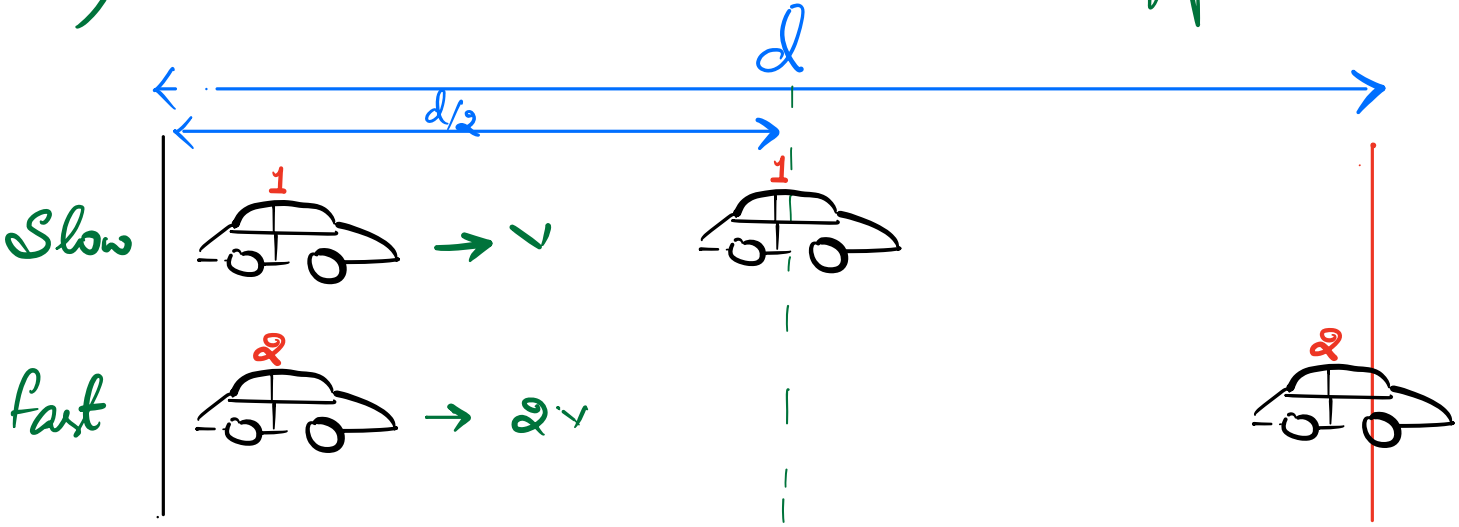


Soln 1) Using Length

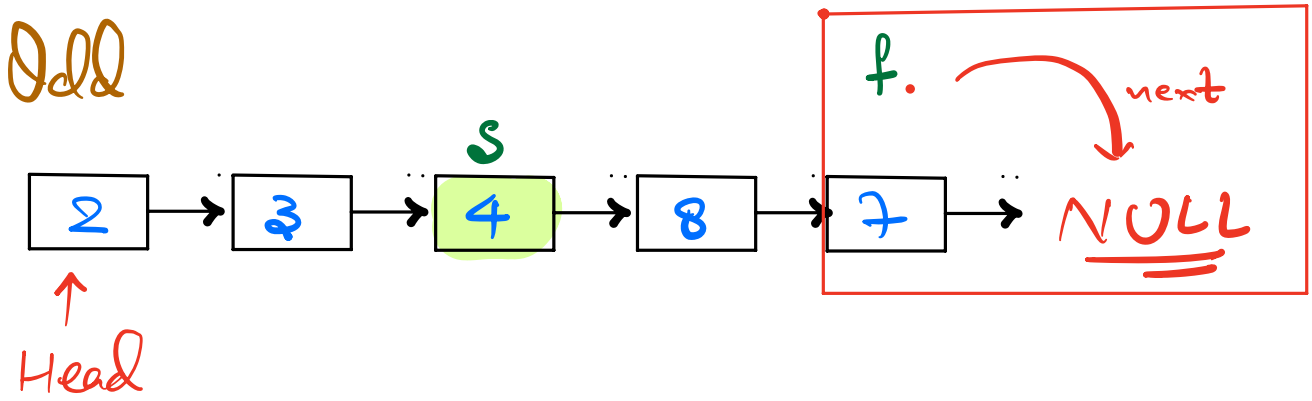
O(N) 1) find Length of LL
O(1) 2) find index of mid.
O(N) 3) Traverse to mid & return

T.C. = O(N)

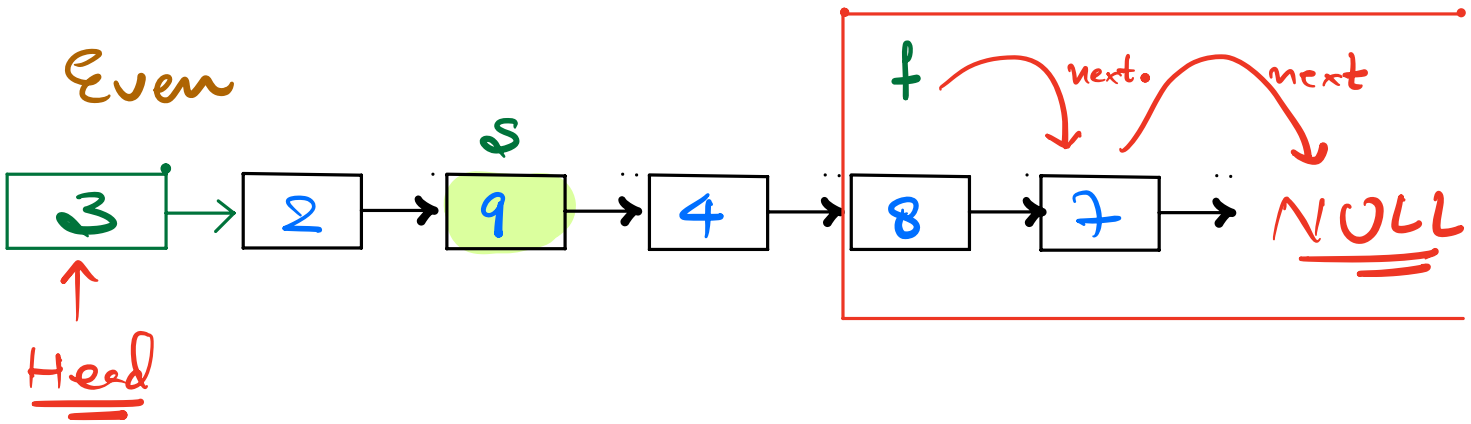
2) Slow & fast Pointer Approach



Odd



Even



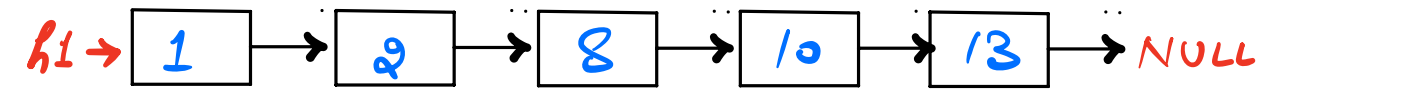
Code

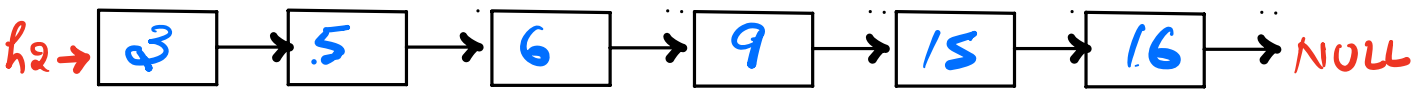
```
Node getMiddle (Node Head) {

    if (head == NULL) { return Head; }

    Node slow = Head;
    Node fast = Head;

    while (fast.next != NULL && fast.next.next != NULL) {

        slow = slow.next;
        fast = fast.next.next;
```

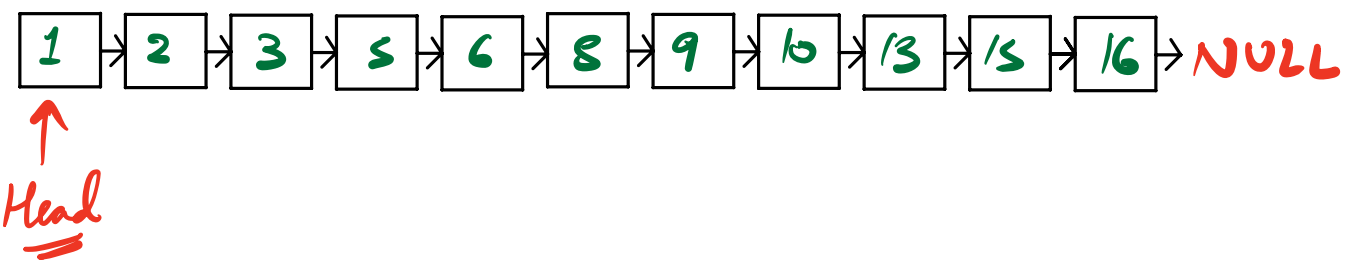
```
    }
    return slow;
}
```

$T.C = O(N)$

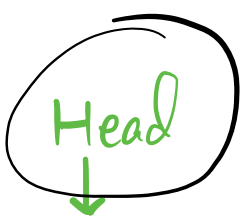
Given two sorted LL. Merge them & return the Head.

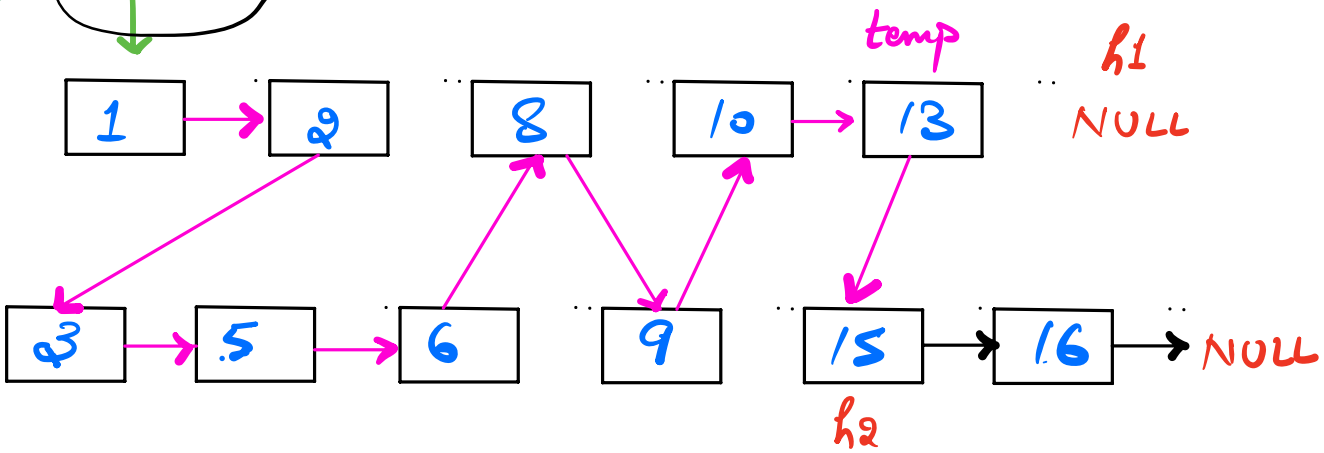






Soln





```
Node merge (Node h1, Node h2) {

    if (h1 == NULL) { return h2; }

    if (h2 == NULL) { return h1; }

    Node Head;
    if (h1.data < h2.data) {
            Head = h1;
            h1 = h1.next;
    } else {
            Head = h2;
            h2 = h2.next;
    }

Node temp = Head;
while (h1 != NULL && h2 != NULL) {

        if (h1.data < h2.data) {
                temp.next = h1;
                h1 = h1.next;
        } else {
                temp.next = h2;
                h2 = h2.next;
        }
        temp = temp.next;
}
```

```
    if (h1 == NULL) {
        temp.next = h2;
    } else {
        temp.next = h1;
    }
    return Head;
}
```

Length (h1) = N1

Length (h2) = N2

T.C. = O(N1 + N2)
S.C. = O(1)

Q Given a LL. Sort it using Merge Sort

Code

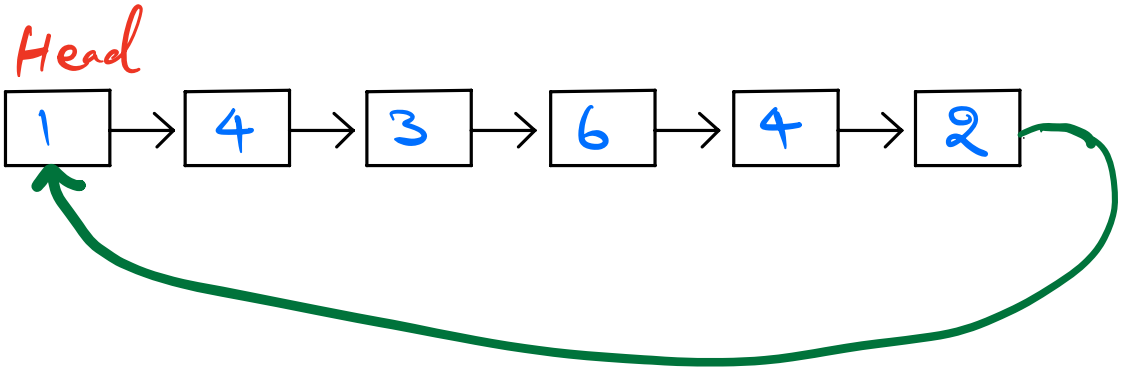
```
Node mergeSort (Node Head) {
    if (Head == NULL || Head.next == NULL) {
        return Head;
    }
    Node mid = getMiddle (Head);   O(N)
    Node h2 = mid.next;
```

```
mid.next = NULL
Node h1 = mergeSort(Head);
h2 = mergeSort(h2);

return merge(h1, h2);   O(N)
}
```

T.C. = $N \times \log(N)$

S.C. = $O(\log N)$

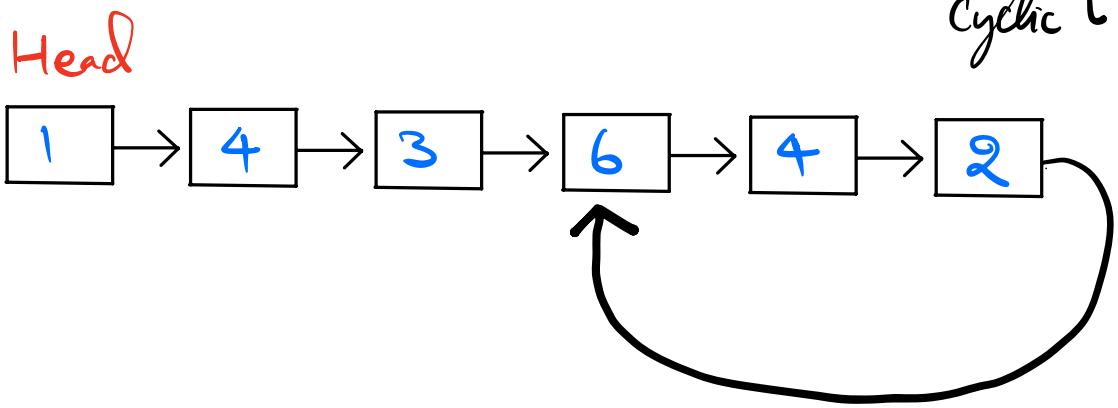
Circular Linked List



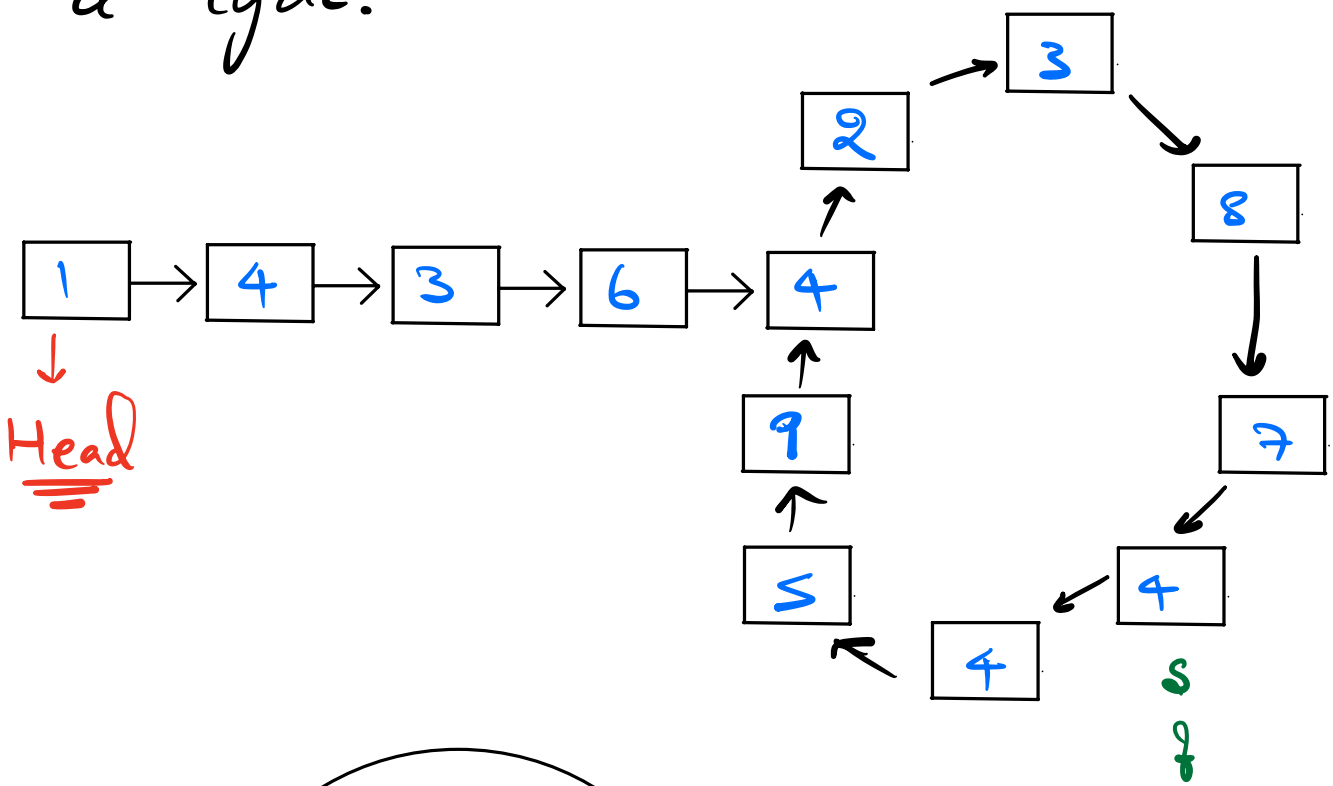
Given a LL. Check if it is circular?

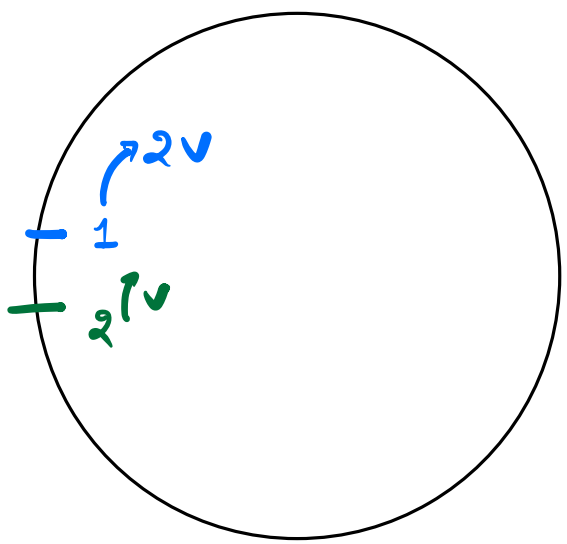
Linked List with Cycle

Cyclic LL



Given a LL. Check if it contains a cycle.



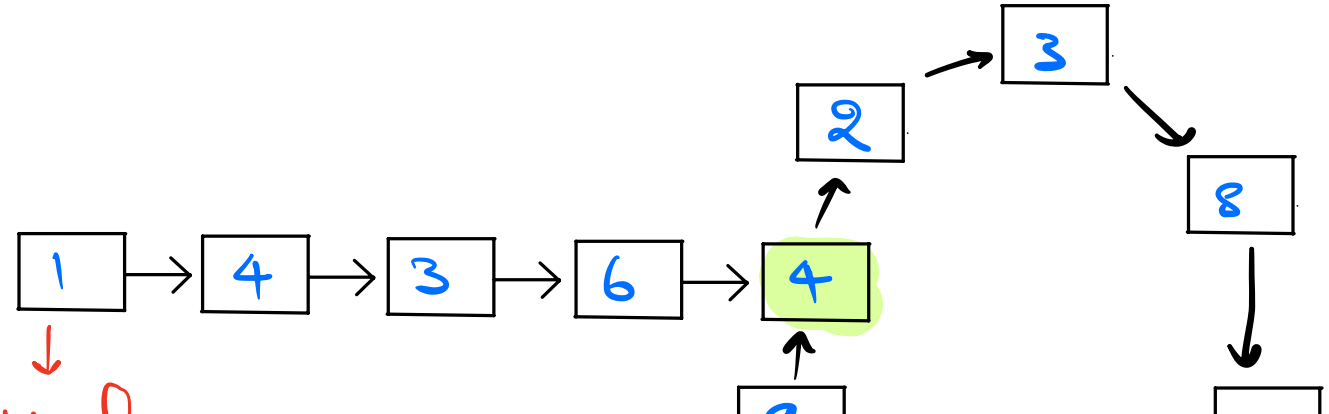


Code

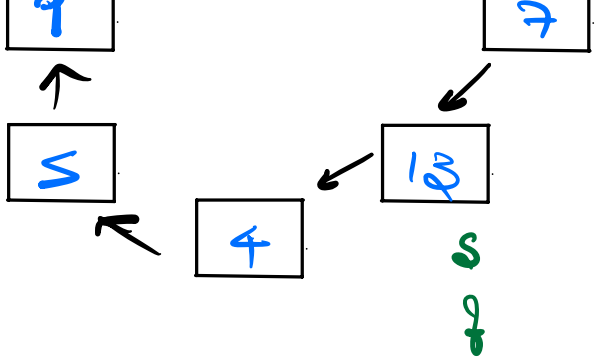
```
bool checkCycle (Node Head) {

    Node fast = Head;
    Node slow = Head;

    while (fast != NULL && fast.next != NULL) {

        slow = slow.next;
        fast = fast.next.next;

        if (slow == fast) {
            return true;
        }
    }
    return false;
}
```

T.C. = O(N)
S.C. = O(1)

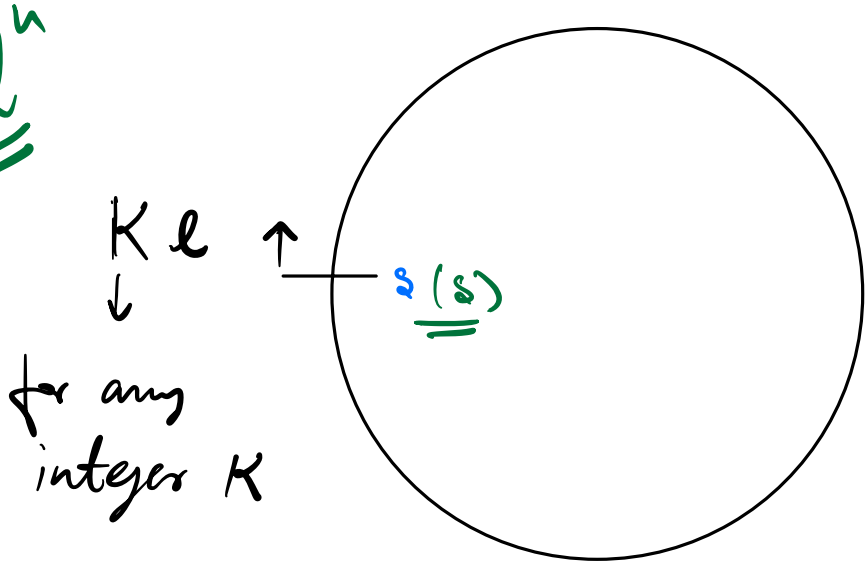
Given a cyclic LL. Return the first (start) node of the cycle.



Head

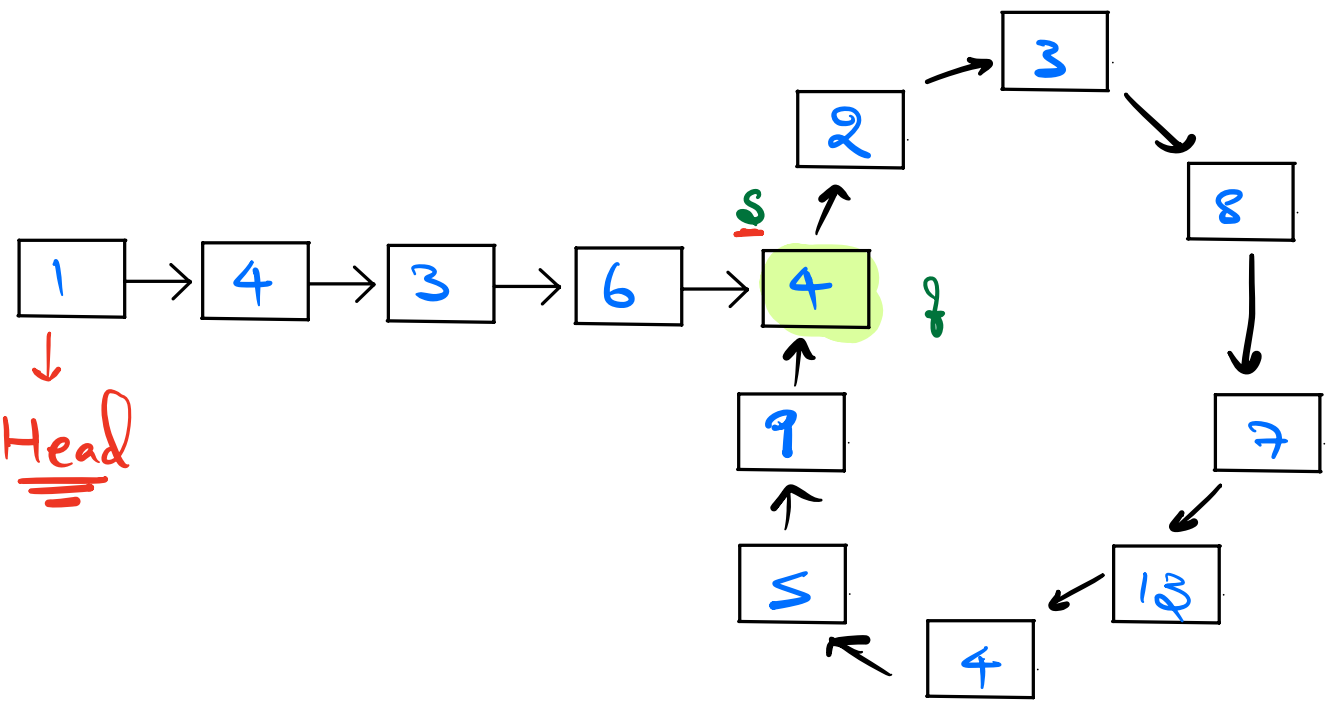


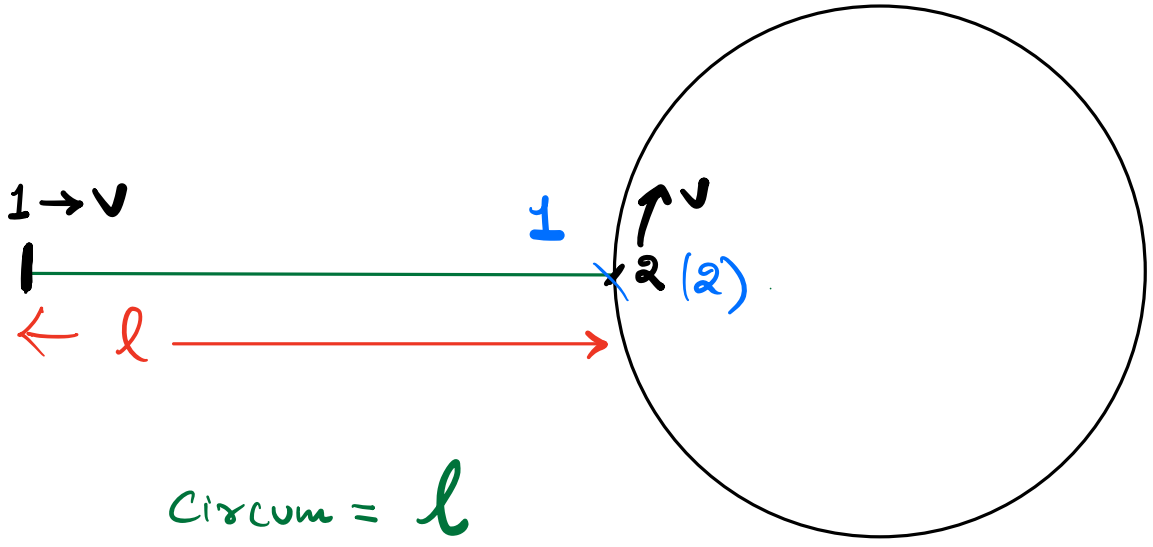
Sol^n



Circum = ℓ

1) Using Length of loops



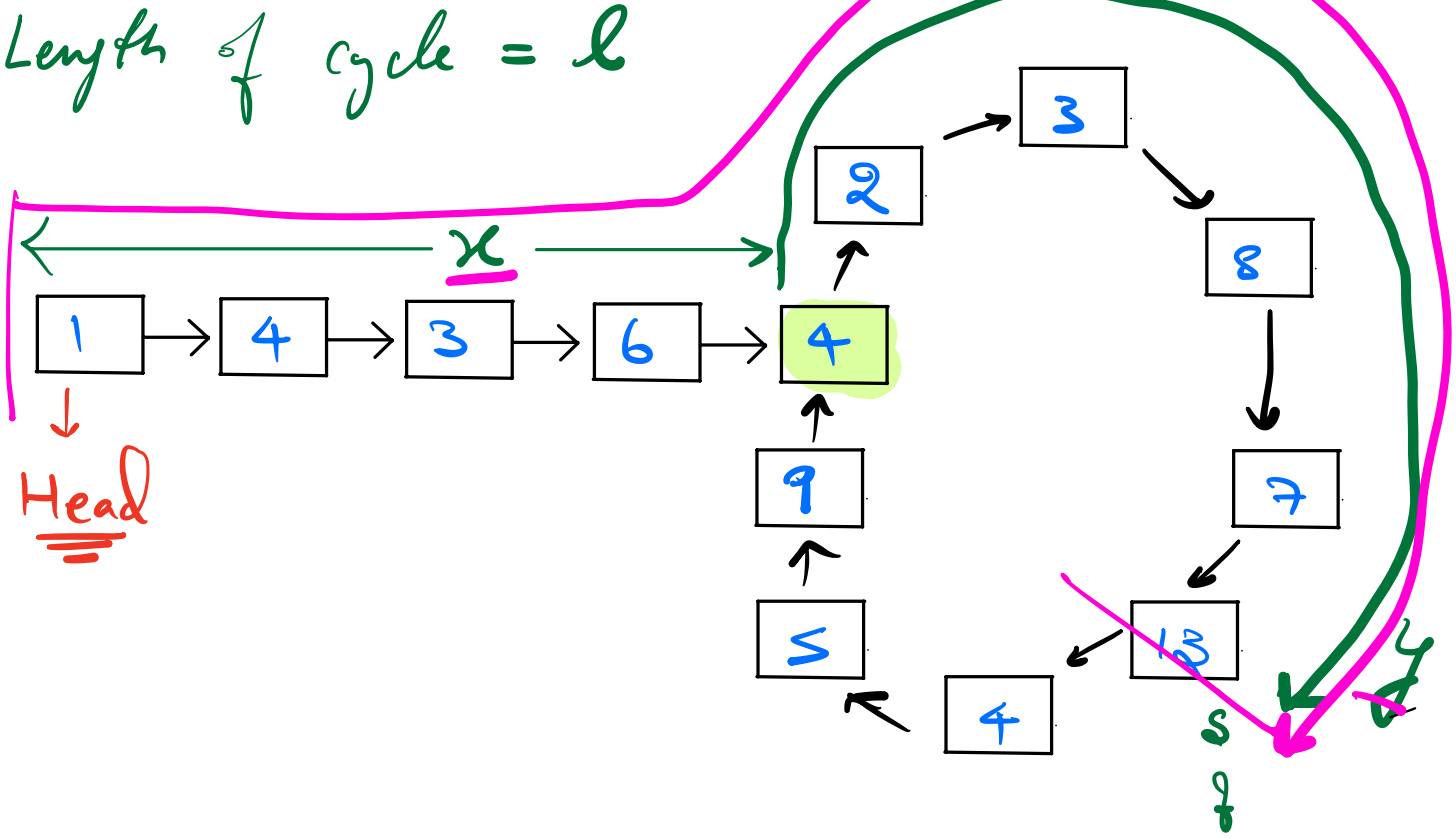


1) Get the length of cycle.

1) Once the slow & fast pointer meets, we stop the slow pointer
2) We move the fast & count the elements till we reach back to slow.

Code ⇒ H.W.

2) Approach 2 (without length)



Distance slow = $x + y + K_1 l$

Distance fast = $x + y + K_2 l$

Distance fast = $2 \times$ (Slow)

$(x + y + K_2 l) = 2x\,(x + y + K_1 l)$

$x + y + K_2 l = 2x + 2y + 2K_1 l$

$(K_2 - 2K_1)\, l = (x + y)$

$(K_2 - 2K_1)$ — Integer

The slow & fast are meeting at a distance of l integral multiple of l.

Code

```
Node getStart(Node Head){
    Node fast = Head;
    Node slow = Head;
    bool isCyclic = false;
    while(fast!=NULL && fast.next!=NULL){
        slow = slow.next;
        fast = fast.next.next;
        if(slow==fast){
            isCyclic = true;
            break;
        }
    }
    if(isCyclic==false){
        return NULL;
    }
```

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

11:00 AM ⇒ PS session