21/04/2023

Q1 Find the middle of the linked list.

i/p → (10) → (20) → (30) → (40) → x

o/p → 20 /30

depending on the question

i/p → (10) → (20) → (30) → (40) → (50) → x

o/p → 30

Length = even, middle = $\frac{n}{2}$ node

Length = odd, middle = $\left(\frac{n}{2}+1\right)$ node

The above is one of the solution which has time complexity = $O(n)$ → not in single traversal

Tortoise algorithm

↓ head

(10) → (20) → (30) → (40) → (50) → x

1) Initially slow and fast at the head.

2) Slow pointer moves one step whereas fast moves 2 steps

(i) fast

$10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow X$

slow

fast moves 2 steps
slow moves 1 step.

(ii) $10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow X$

slow fast

fast moves 2 steps
slow moves 1 step

(iii) $10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow X$

slow fast

Now fast can't move 2 steps and hence slow won't move forward & hence slow will be the middle node.

Time complexity = $O(n/2) = O(n) \rightarrow$ Single pass

Code

```
Node * getMiddle (Node * & head) {
    // Empty linked list
    if (head == NULL) {
        return head;
    }
    if (head→next == NULL) { // Single node in linked list
        return head;
    }
```

```
//Creation of pointers
Node * slow = head;
Node * fast = head;
//slow and fast both are valid
While (slow != NULL && fast != NULL){
        fast = fast -> next; //Move 1 step
        if (fast != NULL) {
//Total -> 2 steps   fast = fast -> next; //Move 1 step
                     Slow = slow -> next; //Move 1 step
        }
    }
    return slow; //middle node
}
```

Note → If the question says $\frac{n}{2}$ is the middle in case of even length,
Initialize fast = head → next

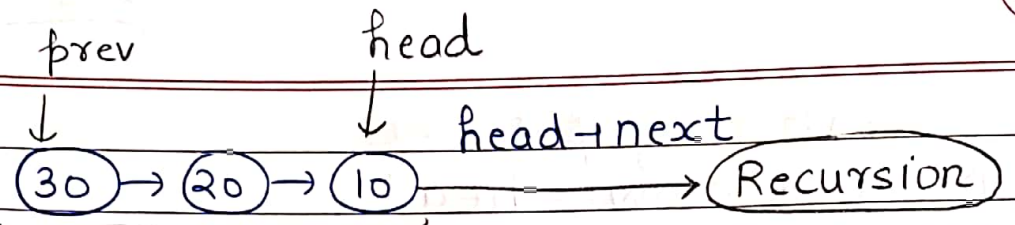
The slow and fast pointer approach will be expected from us in the interview.

Q2 Reverse the linked list in k groups.

i/p → (10) → (20) → (30) → (40) → (50) → (60) → X , k=3
o/p → (30) → (20) → (10) → (60) → (50) → (40) → X

For the above test case, we can simply reverse first k nodes & then recursion will handle

(10) → (20) → (30) → (40) → (50) → (60) → X k=3
Reverse these 3 nodes



Simply reverse a linked list

Simply return prev which is the new head of the linked list

Code

```
Node * reverseKNodes (Node* & head, int k
{
        // Empty linked list case
        if (head == NULL) {
                return head;
        }
        // Invalid case
        int len = getLength (head);
        if (k > len) {
                return head;
        }
        //Step-1 ⇒ Reverse first k nodes
        Node * prev = NULL;
        Node * curr = head;
        int count = 0;
        while (count < k) {
                Node * forward = curr → next;
Reverse         curr → next = prev;
steps           prev = curr;
                curr = forward;
                count ++; //To keep track of k nodes
        }
        //Step-2 Recursion
        if (forward != NULL) { // Still have nodes to
                                  reverse
```

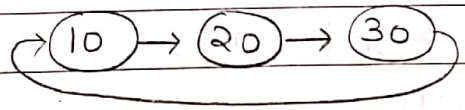
connection step

```
    head->next = reverseKNodes (forward, k);
}                                  head of further LL.
//New head
return prev;
}
```

Steps which we followed

1) Reverse first k nodes.
2) Find answer of recursion.
3) Connection step ⇒ head → next = ans of recursion.
4) Return new head i.e. prev.

Q3 Input linked list is circular or not.

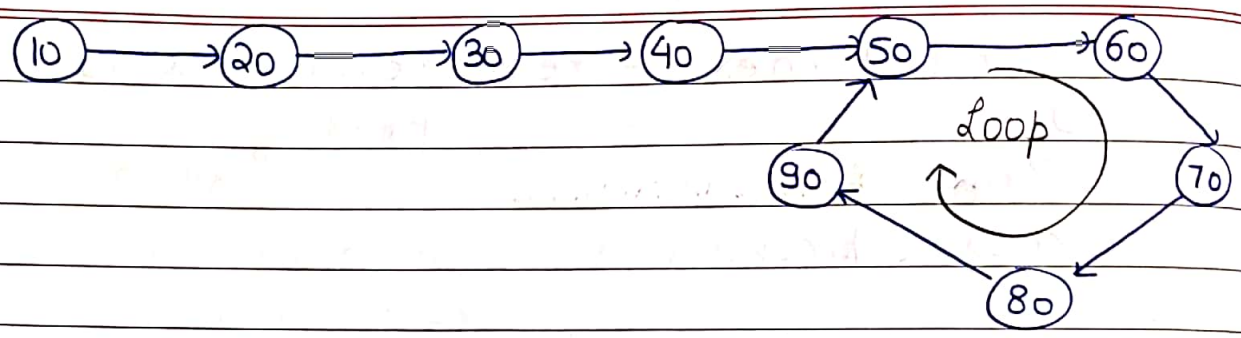
i/p → 

o/p → True

1) Store head in temp
2) Now start from head → next and traverse until we get the same node. If we get NULL, then return false else if we get the same node again, then simply return true.

Q4 Detection and deletion of loop in the linked list.

* Check loop is present or not.
* Starting point of loop
* Remove loop.



* Finding loop is present or not.

Approach-1

This approach is via the map like marking a node visited once traversed and if while adding entry in map, we come across a node that is already visited then loop is present else not present.

```
map<Node*, bool> m;
m[temp] = true;
```

Approach-2

Floyd's cycle detection algorithm. Here we will be playing with slow and fast pointers.

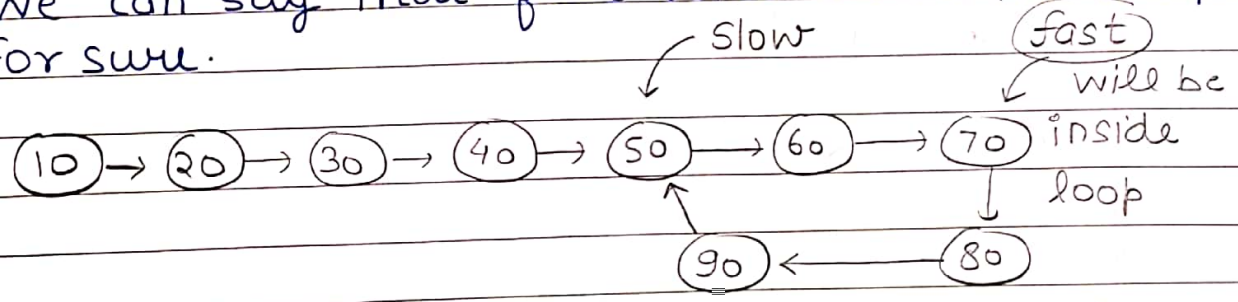
If at some point of time, slow == fast then loop is present else if fast becomes NULL, then loop is not present.

Code

```
bool checkLoop(Node* &head) {
        //Empty linked list case
        if(head == NULL) {
```

```
        return false;
    }
    //2 pointer approach
    Node * slow = head;
    Node * fast = head;
    //If fast becomes NULL, not circular
    while (fast != NULL) {
        fast = fast -> next;
        if (fast != NULL) {
            fast = fast -> next;
            slow = slow -> next;
        } //Loop is present
        if (slow == fast) { return true; }
    }
    return false; //Loop not present
}
```

Why the algorithm is working?
When slow is at the starting point of loop, we can say that fast will be in the loop for sure.



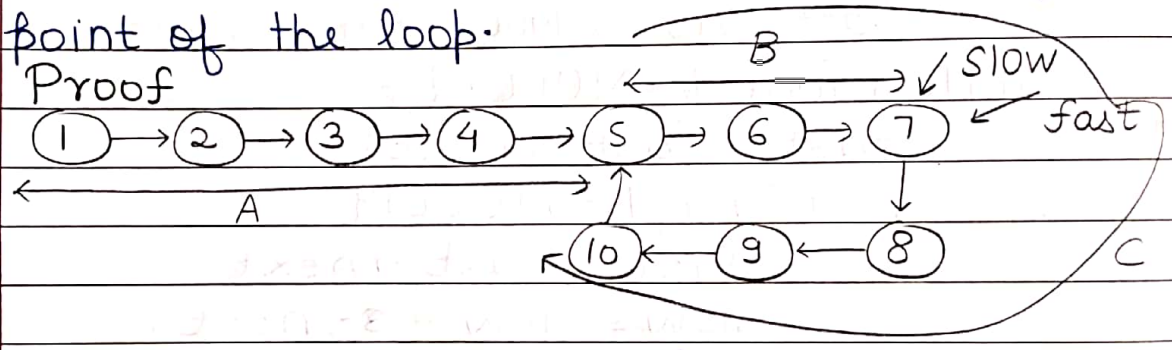
Anticlockwise sense ⇒ distance is of 4 nodes, then 3 nodes and then 2 nodes, 1 nodes and 0 nodes and hence this algo is working.

* Find the starting point in the loop.
1) Apply Floyd's cycle detection algorithm.
Here slow and fast are at same position.

2) Now move slow to head by doing slow = head.

3) Now move both slow and fast by one step each. Now here when slow and fast meet each other, this is the starting point of the loop.

Proof



Distance travelled by fast pointer = 2 × Distance travelled by slow pointer

$$A + \underbrace{x}_{\text{no. of cycle}} C + B = 2 \times (A + yC + B)$$

$$A + xC + B = 2A + 2yC + 2B$$

$$(x - 2y)C = A + B$$

Let $x - 2y = K$

$$A + B = \underbrace{KC}_{\text{some no. of cycles}}$$

A+B → 1 cycle, 2 cycle ... but at the end we reach at starting point of the loop.

slow = head ⇒ Distance travelled will be A and fast travels same distance and hence this algorithm is working.

Code

```
Node *findStart (Node* & head) {
    //Empty LL
    if (head == NULL) {
            return head;
    }
    Node * slow = head;
    Node * fast = head;
    while (fast != NULL) {
        fast = fast -> next;
        if (fast != NULL) {
            fast = fast -> next;
            slow = slow -> next;
        }
        //When we found loop, so simply make slow to head.
        if (slow == fast) {
            slow = head;
            break;
        }
        //Again run loop until slow & fast meet again.
        While (slow != fast) {
            slow = slow -> next;
            fast = fast -> next;
        }
        return slow;
}
```

* Delete loop in the linked list
We just have make the next of the previous of starting node as NULL and the loop is removed.

```
Node * prev = fast;
while (slow != fast) {
        slow = slow -> next;
        fast = fast -> next;
}
//Removal step
prev -> next = NULL;
```

modifications

```
prev = fast;
```