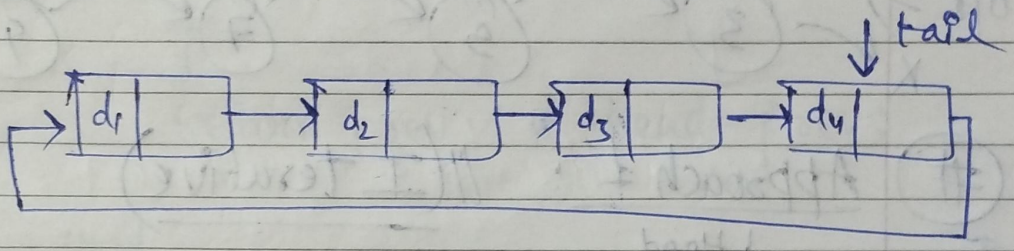
Page No.
Date

(*) Circular linked list :

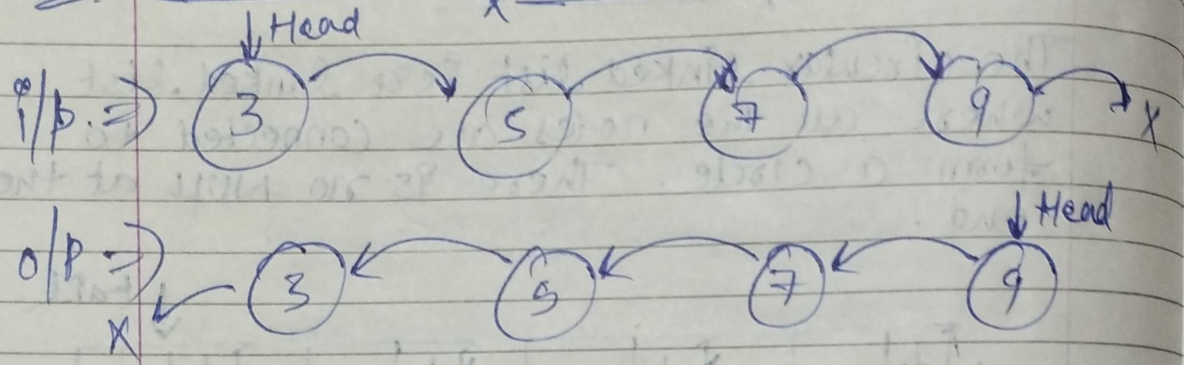
The circular linked list is a linked list where all the nodes are connected to form a circle. There is no NULL at the end.



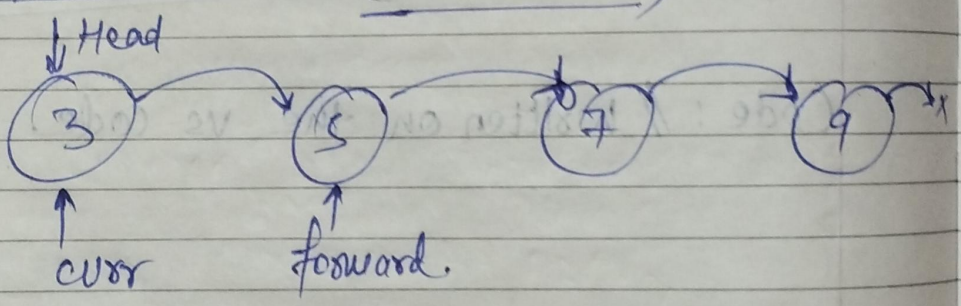
Code : // written on the vs-code.

Page No.
Date

Q. Reverse a linked list :-



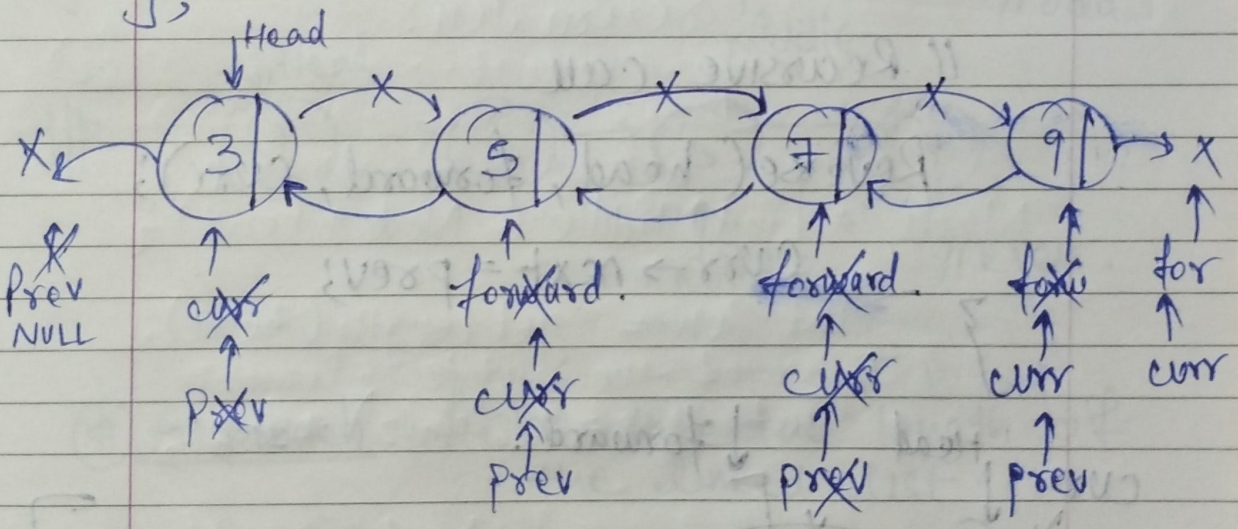
Approach 1 : (Iterative)



Prev = NULL

```
{
    // empty list or Single Node
    if ( head == NULL || head->next == NULL){
        return head;
    }

    // More than 1 Node.
    Node * prev = NULL;
    Node * curr = head;
    Node * forward = NULL;
```

```
while( curr != NULL) {
    forward = curr->next;
    curr->next = prev;
    prev = curr;
    curr = forward;
}
return prev;  // head = prev;
};
```

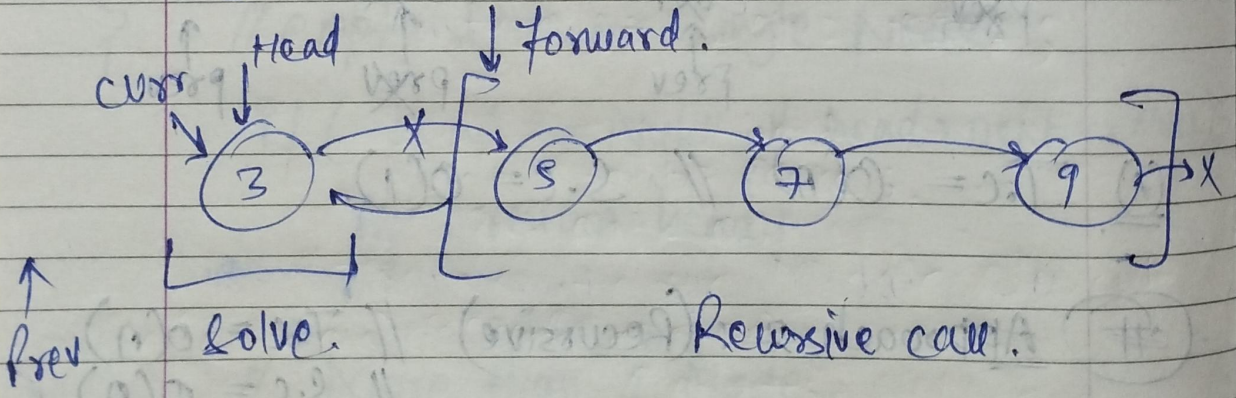


(*) T.C = O(n) // S.C = O(1).

(#) Approach 2 : (Recursive) // T.C = O(n)
// S.C = O(n)

```
Void Reverse(Node*&head, Node* curr,
             Node * prev)
{
```

```
// base case
if (curr == NULL){
    head = Prev;
    return ;
}

Node *forward = curr->next;

// Recursive call
Reverse( head, forward, curr);

    curr->next = prev;
}
```



Solve only for first case and Recursion will handle the rest of the cases.

Make Sure to connect the Return of Recursion to prev node.

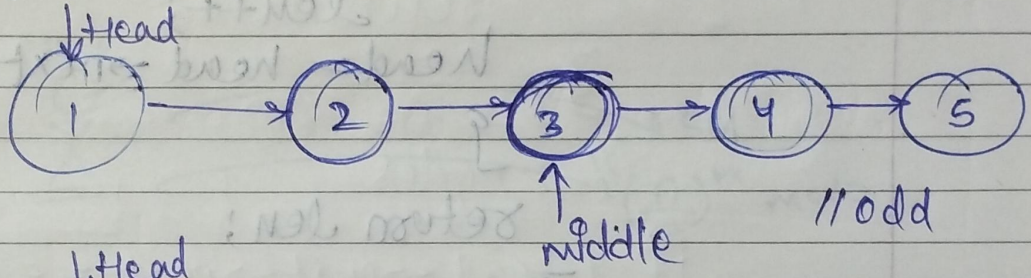
Page No.
Date

* Imp

Q. Middle of the linked list :-

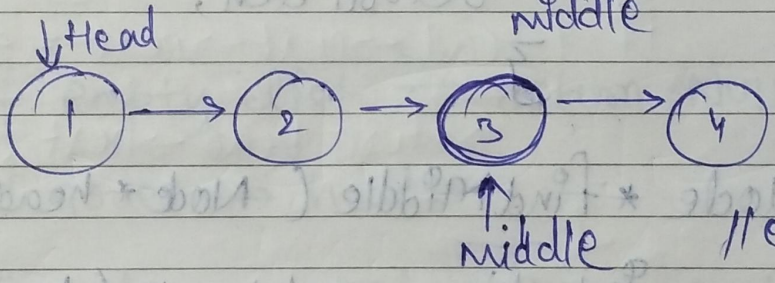
If there are an odd no. of element, return the middle element; if there are even elements return the one which is farther from head.

i/p.



// odd

i/p



// even.

(#) Approach : Calculate the length of the linked list.

len → even ⇒ $\left(\frac{len}{2}+1\right)$ → middle.

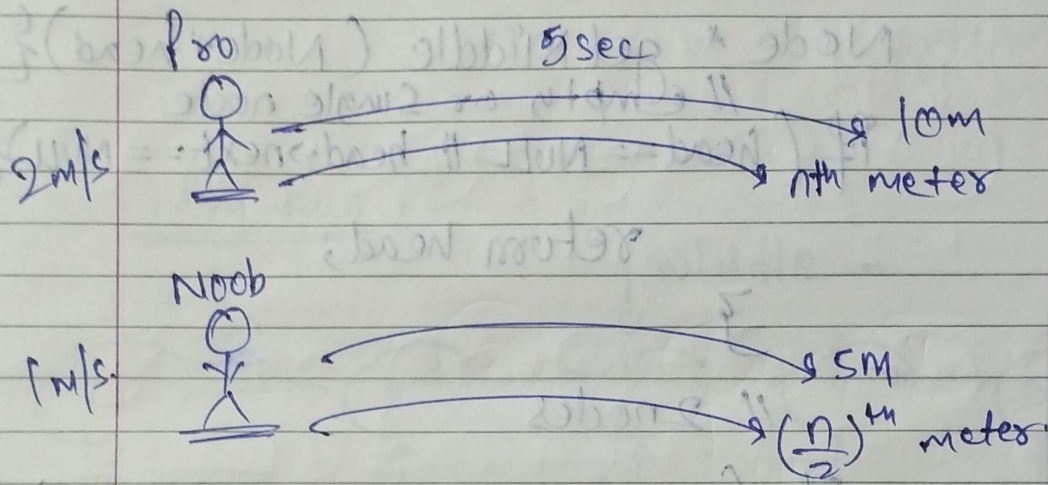
len → odd ⇒ $\left(\frac{len}{2}+1\right)$ → odd.

(*) T.C = O(n)

Code:

```
int getLength (Node *head){
    int len = 0;
    while ( head != NULL){
        len++;
        head = head->next;
    }
    return len;
}

Node *findMiddle ( Node * head) {
    int len = getLength (head);
    // Index of the middle element
    int ans = (len/2) ;

    Node *temp = head;
    int cnt = 0;
    // Traversing to the Middle element
while (cnt < ans) {
        cnt++;
        temp = temp->next;
    }
    return temp;
}
```

(*) Approach2 : optimised code.

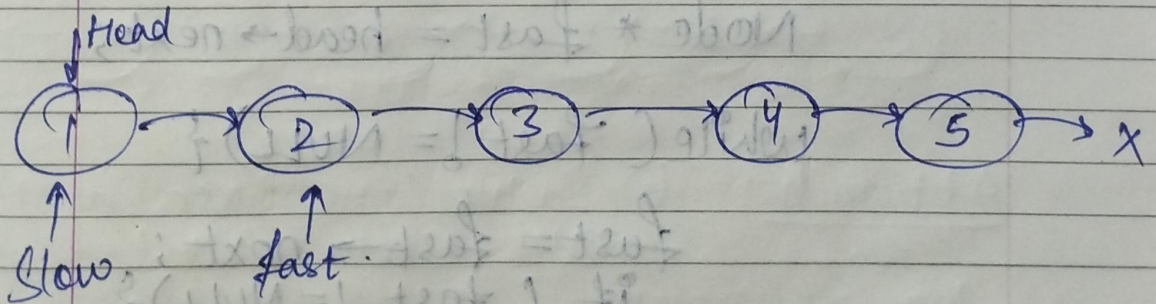


(*) if empty list → return NULL;

(*) 1 node → head;

(*) 2 node → return (head→next);

(*) More than 2 node → Algo ←



Node * slow = head; // Increase by 1

Node * fast = head→next; // Increase by 2

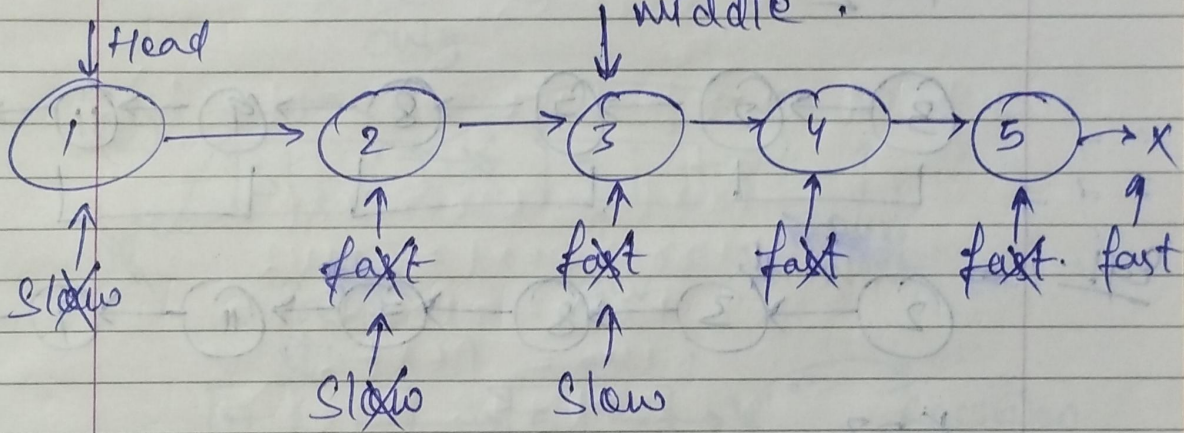
Code:

```
Node * getMiddle (Node* head){
    // empty or single node
    if (head == NULL || head->next == NULL){
        return head;
    }
    // 2 nodes
    if (head->next->next == NULL){
        return head->next;
    }
    // Algo
    Node * slow = head;
    Node * fast = head->next;
    while (fast != NULL){
        fast = fast->next;
        if (fast != NULL){
            fast = fast->next;
        }
        slow = slow->next;
    }
```

return slow;

}

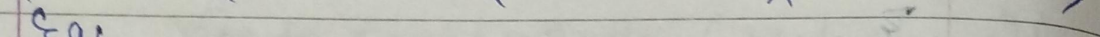
dry Run :



Microsoft

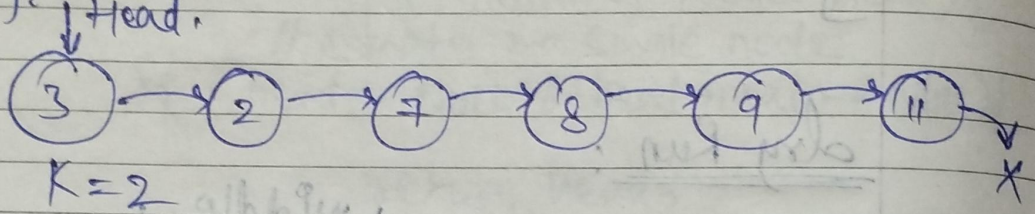
Page No.
Date

Q Reverse a linked list on K^{th} element.

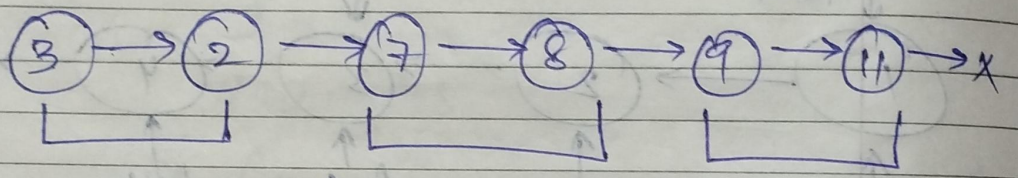


Eg:-

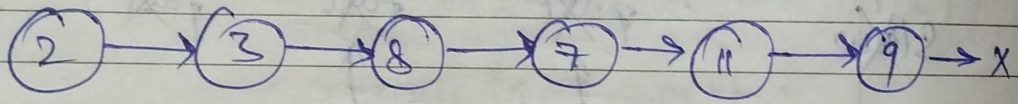
i/p



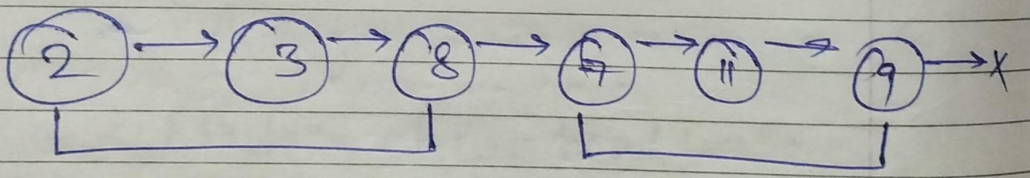
K=2

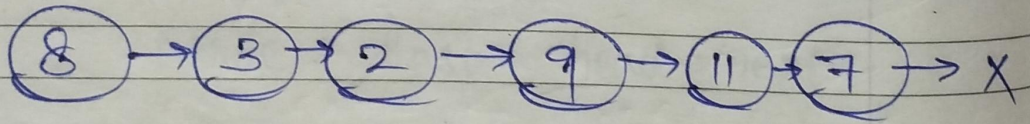


Eg!



K=3

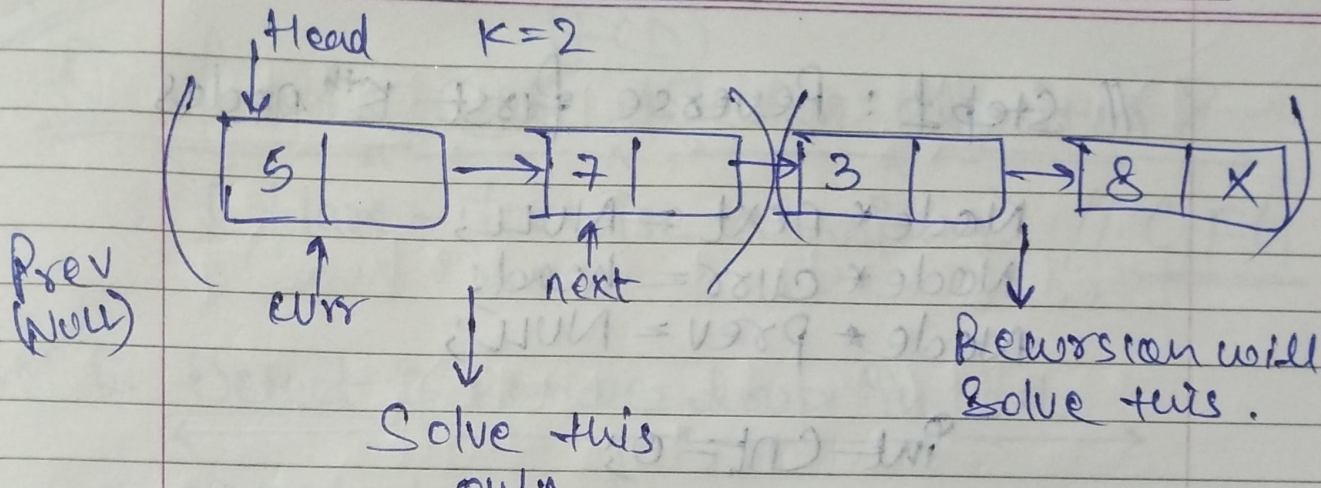


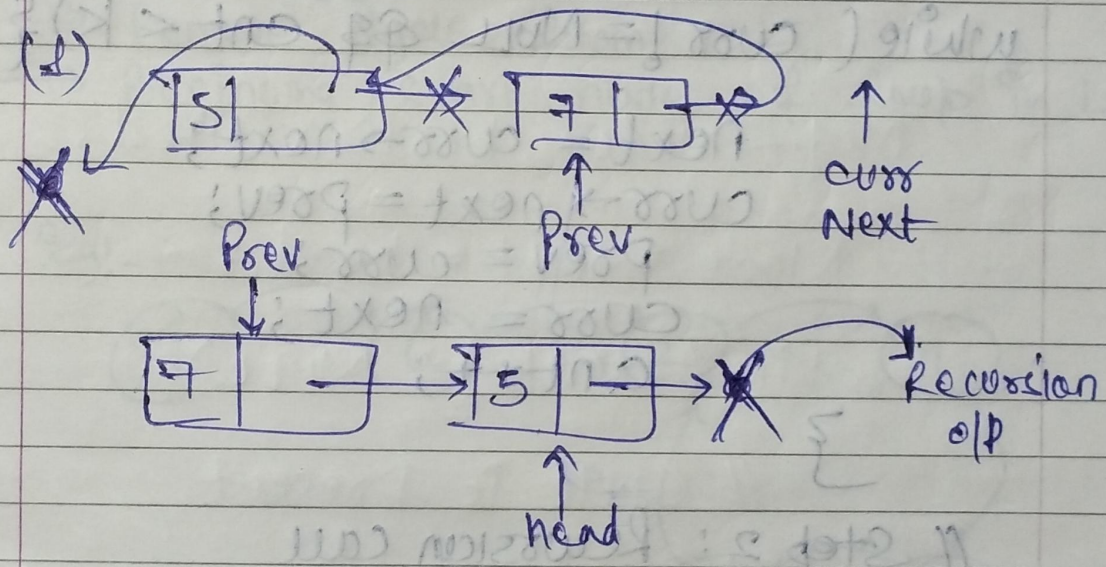


Approach: (*) Solve one case and the rest will be solved by recursion.

(*) head→next = recursion o/p;

(*) return prev;





(2) head → next = Recursion call;

(3) return prev;

Code :-

```
Node * kReverse( Node *head, int k){

  // base case : empty list
    if( head == NULL) {

        return NULL;
    }
```

// Step 1 : Reverse first K^{th} nodes

```
Node* next = NULL;
Node* curr = head;
Node* prev = NULL;

int cnt = 0;

while( curr != NULL && cnt < k){
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
        cnt++;
    }

// Step 2 : Recursion call
if( next != NULL){
head->next = kReverse(next, k);
}

return prev;
}
```

T.C = O(n)
S.C = O(n)

Page No.
Date

Imp

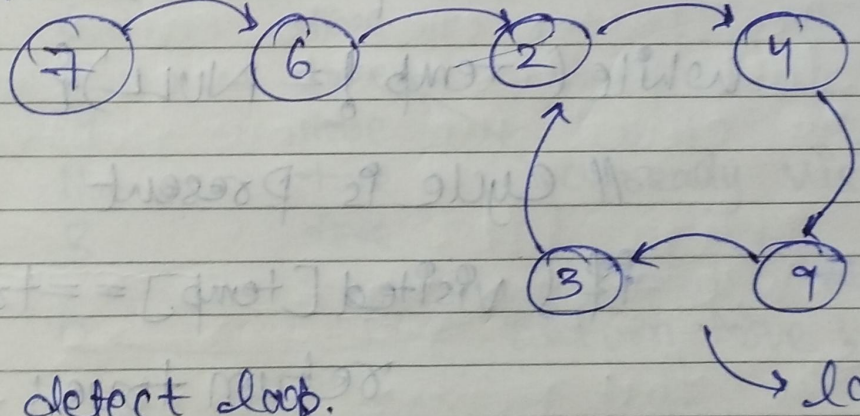
(FCD)

Q. Floyd's cycle detection algorithm ?
(Hare - tortoise algorithm). // GFG

*** Q. Detect & Remove loop in L.L

✓ → Detect cycle in LL
→ Remove cycle from L.L
✓ → Beginning / start node of loop in LL

Eg:-



// detect loop.

Approach: create a map and store the visited node value as true

map < Node*, bool > visited ;

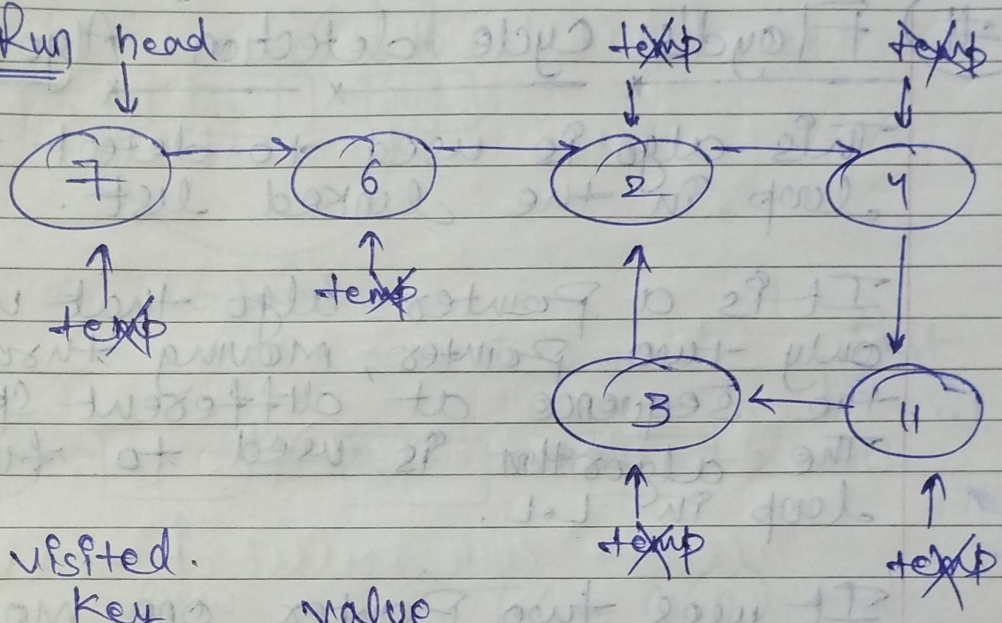
T.C = O(n)
S.C = O(n)

Page No.
Date

Detect cycle in Linked List

```
bool detectLoop (Node* head) {

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

    Map<Node*, bool> visited;

    Node *temp = head;

    while ( temp != NULL){

        // cycle is present

        if (visited[temp] == true){
            return true;
        }

        visited[temp] = true;
        temp = temp->next;
    }

    return false;
}
```

dry Run



visited.

temp	Key	value
~~→~~	7	true
~~→~~	6	true
~~→~~	2	true
→	4	true
~~→~~	11	true
~~→~~	3	true
→	2	

already visited

return true;

? Q. → Can we reduce the size for this?

→ Ans: By using the floyd's cycle detection algorithm.

Imp
*****

Page No.
Date

Floyd's Cycle detection Algorithm

This algo is used to detect the loop in the linked list.

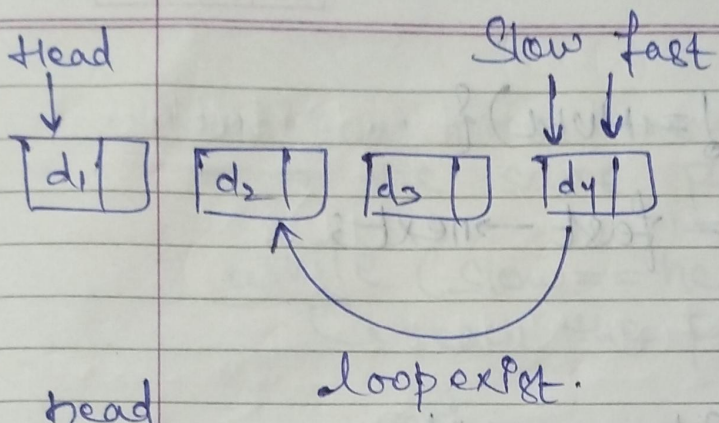
It is a pointer algo that uses only two pointer, moving through the sequence at different speed. The algorithm is used to find a loop in L.L.

It uses two pointer one moving twice as fast as the other one. The faster one is called the fast pointer and the slow one is called the slow pointer

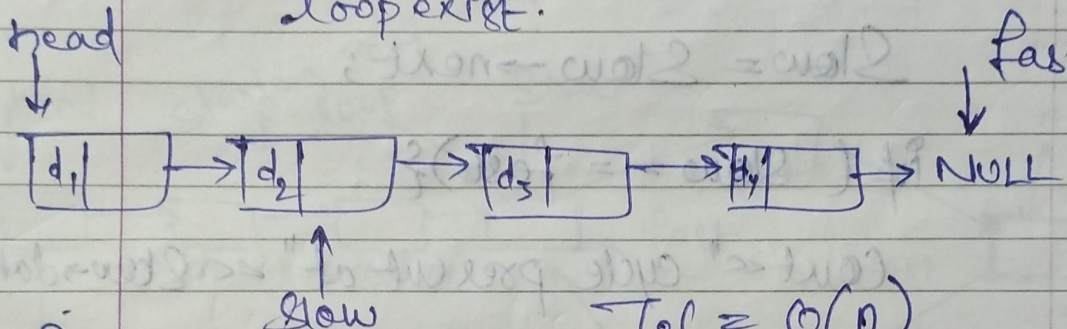
→ While traversing :-

(1) The fast pointer may reach the end (NULL) this shows that there is no loop in L.L

(2) The fast pointer again catches the slow pointer at some time therefore a loop exist in the L.L.



Loop exist.



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

// method to detect cycle in loop.

*** Code :→

```
Node *floydDetectLoop (Node *head){
    if (head == NULL){
        return NULL;
    }

    Node *slow = head;
    Node *fast = head;

    while (fast != NULL && slow != NULL){
        fast = fast->next;
```

```
if (fast != NULL){
    fast = fast->next;
}

slow = slow->next;

if ( slow == fast){
cout<<" cycle present at" << slow->data;
    return slow;
}
}

return NULL;
}
```

How to find the starting point of the cycle (loop)?

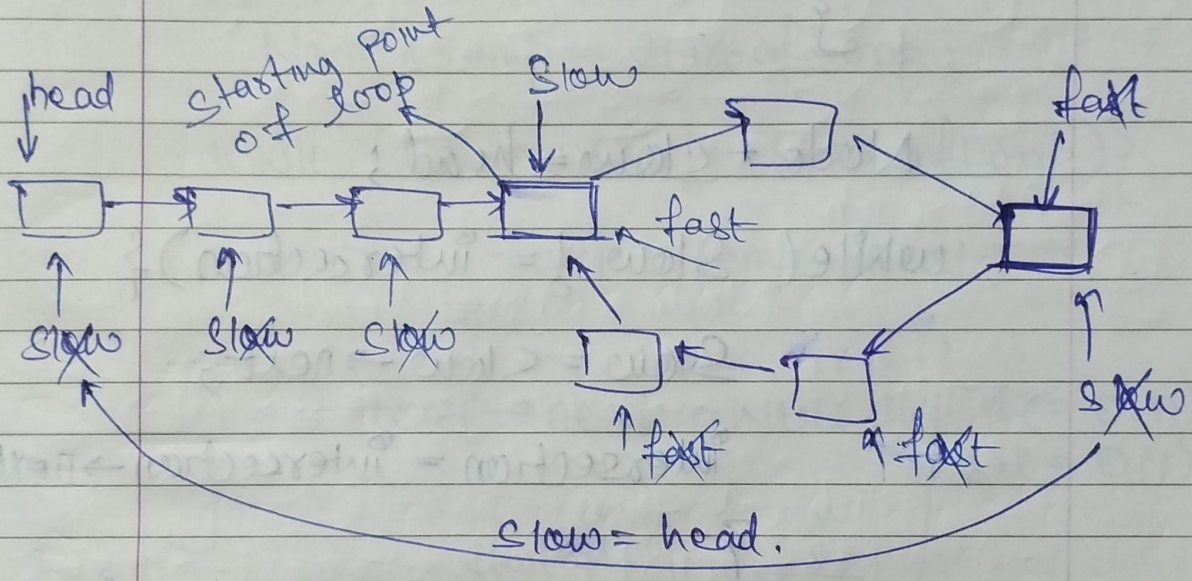
Approach:

(I) → fCD algo → Point of Intersection

(II) → slow = head

and now move the both pointers with the same pace.

while (slow == head)
↳ reach the point of intersection

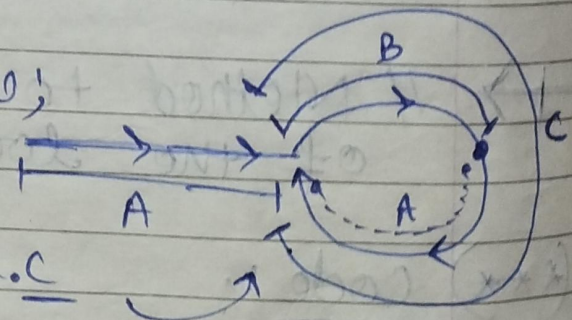


→ // Method to get the starting node of the loop.

(***) Code :-

```
Node * getStartingLoop( Node *head){
    if( head == NULL){
        return NULL;
    }
}
```

```
Node * intersection = floydDetectLoop(head);
if (intersection == NULL){
        return NULL;
}

Node * slow = head;
while( slow != intersection){
        slow = slow -> next;
        intersection = intersection -> next;
}

return slow;
}
```



A + B = K.C

⇒ // Method to remove the loop from the L.L

(***) Code:

```
void removeLoop (Node *head){
```

```
if (head == NULL) {
    return ;
}
Node * StartofLoop = getStartingNode(head);
Node * temp = StartofLoop ;
while (temp->next != StartofLoop){
    temp = temp->next;
}
temp->next = NULL;
```

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

}

dry Run :-