Problems in Linked Lists

1) Middle of LL

Naive

To go through the whole LL and keep a variable count to know the length.

Then use a second traversal till count/2 to reach the element.

Efficent

2 pointer fast & slow

```
void printMiddle(Node head)
{
    if (head == null) return;
    Node slow = head;
    Node fast = head;
    for(Node i = head; i<?) ⟶ ∴ we use while.
    while(fast != null && fast.next != null)
    {
        slow = slow.next;
        fast = fast.next.next;
    }
    System.out.println(slow.data);
```

Note:

When dealing with LL take special care of corner cases :-

1) List is null.
2) List has only 1 element
3) Element is first or last element.

Nth node from end of LL.

Naive

1) Take out count using one traversal.
& print $[count(-)(\frac{n}{+1})]$ th element in second traversal.

2) Efficient

2 pointer approach.

* If $n \geq 3$, take first & keep it at 3rd position & second at head.

* Now we move them at same speed, & when first reached null second reaches the 3rd last element.

Prog.

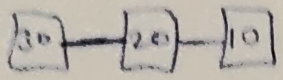
```
void printNthfromEnd(Node head, int n)
{
    if (head == null) return;

    Node first = head;
    for(int i=0; i<n; i++){
        if (first == null) return;
        first = first.next;
    }

    Node second = head;
    while( first != null)
    {
        second = second.next;
        first = first.next;
    }
    System.out.println(second.data);
}
```

3) Reverse a LL

I/P → 

O/P → 

(1) → Naive.

copy linked list to Arraylist
then reverse copy back.

} Ax Space → O(n)
Two Traversal.

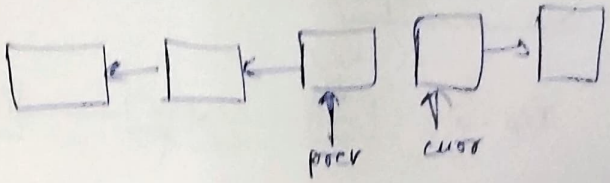
Prog.

```
Node revList(Node head)
{
    Arraylist<Integer> arr = new Arraylist<Integer>();
    for(Node curr=head; curr!=null; curr=curr.next)
    {
        arr.add(curr.data);
    }
    for(Node curr.head; curr!=null; curr=curr.next)
    {
        curr.data = arr.remove(arr.size()-1);
    }
    return head.
}
```

---

(2) Efficient → Changing links instead of data.

Java (reversing algo)




```
next = curr.next;
curr.next = prev;
prev = curr;
curr = next;
```

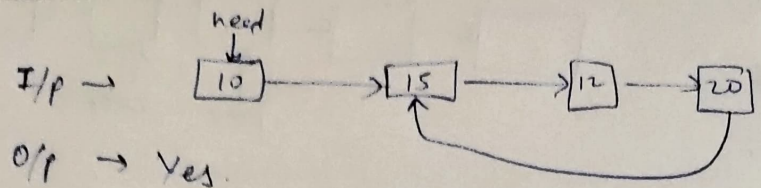
```
void Reverse() {
if (head == null || head.next == null) return;
                                  ↳ return head.data;
Node curr, prev = head;
  while (curr != null)
  {
Node next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
print LL. ∞ return prev  → new head
```

---

4) Remove duplicate from Sorted List.

```
> void removeDupli(Node head)
{ Node curr = head;
  while (curr != null && curr.next != null)
  { if(curr != null && curr.next != null)
    { curr.next = curr.next.next; }
    else { curr = curr.next }
  }
}
```

Detect loops.



Naive

→ 2 loops used

for every ith node compare .next to every other .next. } $O(n^2)$.

Efficient

→ We modify struct of Node.

```
class Node {
    Node next;
    int value;
    boolean visited;
}
```

} $O(n)$.

Efficient – 2

→ We modify links of all linked lists. Pointing every node to dummy node.

→ If .next is already pointing to dummy node.

→ Destroys the ll.

Eff - 3

→ Traverse & hash. $O(n)$ → Space & Time.

```
boolean isLoop(Node head)
{ HashSet<Node> s = new HashSet<Node>();
for (Node curr = head; curr != null; curr = curr.next)
{ if (s.contains(curr)){return true;}
    s.insert(curr);
}
return false;
```

Floyd cycle detection (O(n) → time
O(1) → space)

2 pointer → slow & fast pointers.

if loop is there, they will meet.

```
while (fast != null && fast.next != null)
{   slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) {return true;}
}
return false;
```