Agenda

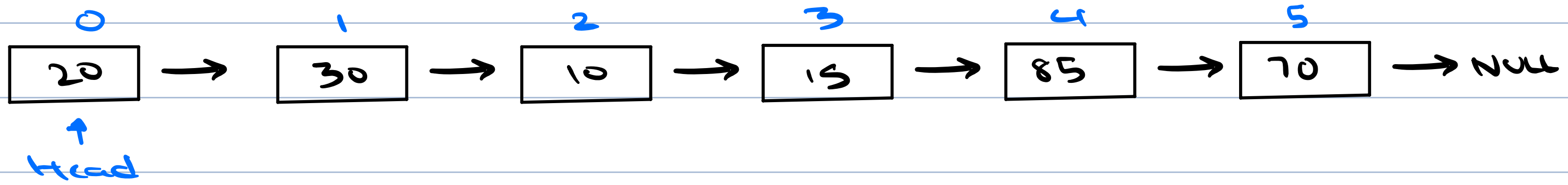
1. Traversal and search
2. Insertion of a node
3. Deletion of a node
4. Reverse a LL
5. Deep clone a LL

```
class Node {
    int data
    Node next

    Node (int x) {
        data = x
        next = NULL
    }
}
```

https://notability.com/n/dqpRUn107SuLpkMqqhh83

1. Search for value x

Return true if value x exists in the linked list, else return false.



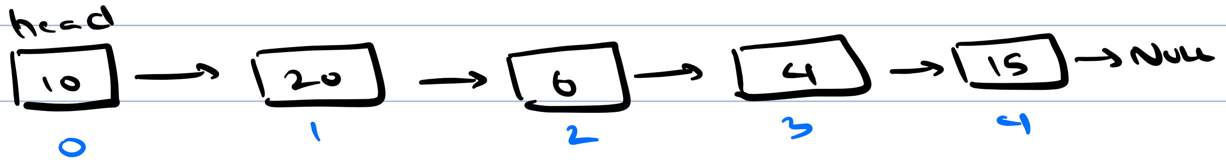
x = 10 ans = true | x = 80 ans = false

```
bool findx (Node head, int x){
        Node tmp = head
    while ( tmp != NULL ) {
          if( tmp.data == x)
                    return true
          tmp = tmp.next
    }
    return false
}
```

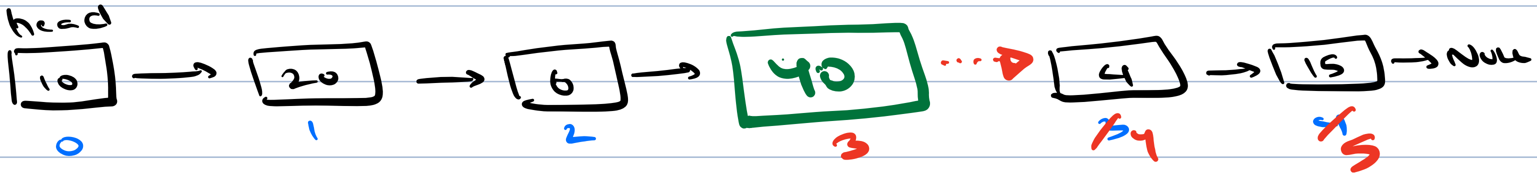
TC: O(N)

SC: O(1)

2. Insert a new node with data v at index p in the linked list (index p will always be valid)

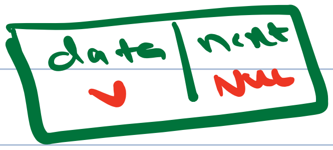


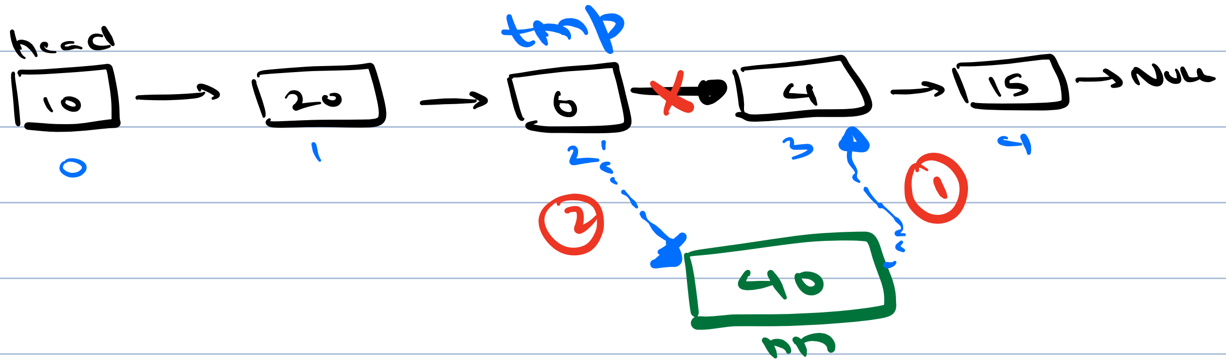
v = 40
p=3



① create a new node with data v

Node nn = new Node(v)





p = 3

② Traverse to (p-1) Node

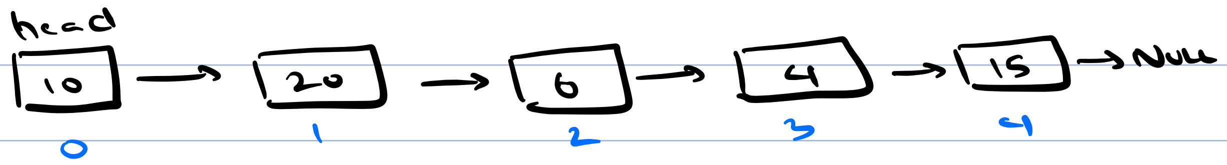
```
Node tmp = head
for (i=1 ; i ≤ p-1 ; i++) {
    tmp = tmp. next
}
```

③

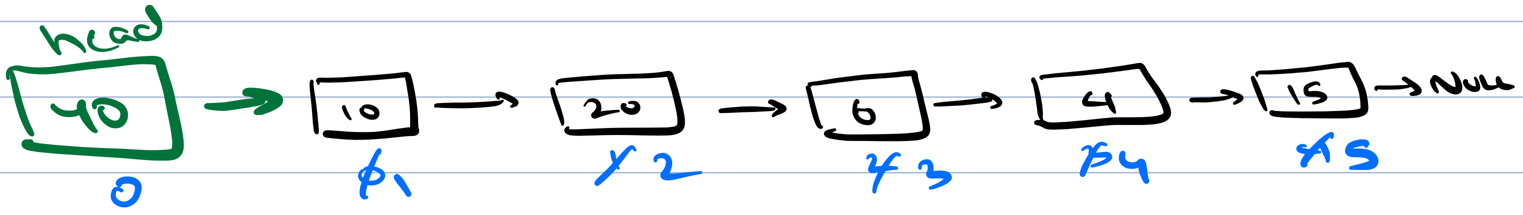
```
nn. next = tmp. next
tmp. next = nn
```

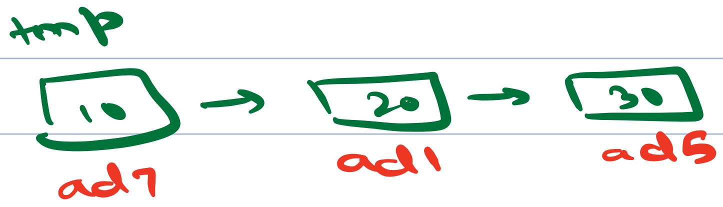
```
void insertNode (Node head, int v, int p) {
    Node nn = new Node (v)
    if (p = = 0) {
        nn. next = head
        head = nn
    }
    else {
        Node tmp = head
        for (i=1 ; i ≤ p-1 ; i++) {
            if (tmp = = NULL) return
            tmp = tmp. next
        }
        if (tmp = = NULL) return
        nn. next = tmp. next
        tmp.next = nn
    }
}
```

TC : O(min(P,N))



v=40
p=0





print (tmp)	ad7
print (tmp.data)	10
print (tmp.next)	ad1
$\underbrace{\underbrace{tmp}_{!Null}.next}_{!Null}.next$	ad5

_ .data
_ .next

3. Deletion in Linked List

Delete first occurrence of value x in given linked list. If element is not present, leave as is.

head

Ⓐ List : 1 → 8 → 4 → -2 → 12

x = -2

head

1 → 8 → 4 → 12

head

Ⓑ List : 1 → 8 → 4 → -2 → 12

x = 4

head

1 → 8 → -2 → 12

head

Ⓒ List : 1 → 8 → 4 → -2 → 12

x = 1

head

8 → 4 → -2 → 12

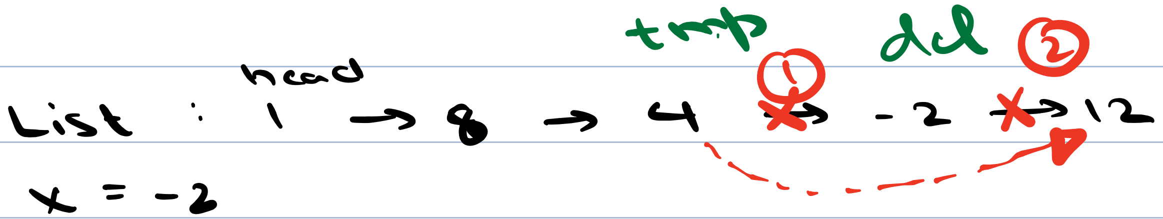
head

Ⓒ List : 1 → 8 → 4 → -2 → 12

x = 20 (Not present)

head

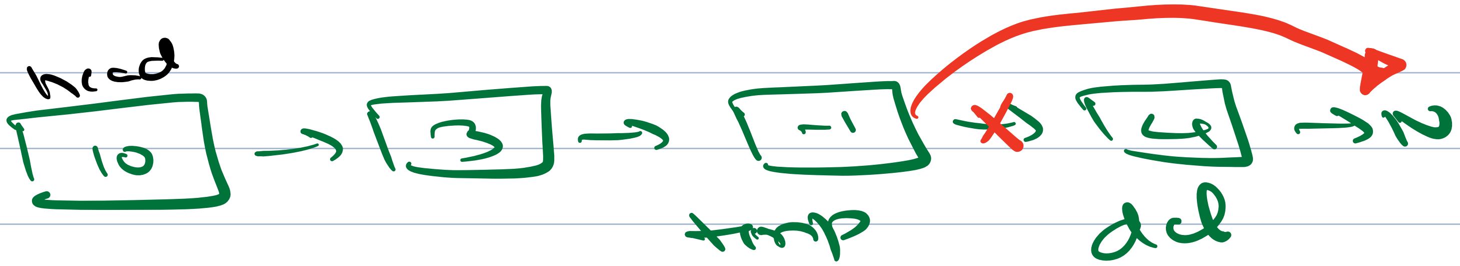
1 → 8 → 4 → -2 → 12



x = -2

Traverse LL. stop when temp.next = x

```
void deleteNode (Node head, int x) {
    if (head == NULL) return
    if (head.data == x) {
        Node del = head
        head = head.next
        del.next = NULL
        free (del)
    }
    else {
        Node temp = head
        while (temp != NULL && temp.next != NULL
        && temp.next.data != x)
            temp = temp.next
        }
        if (temp == NULL || temp.next == NULL)
                                        return
        Node del = temp.next
        temp.next = del.next
        del.next = NULL
        free (del)        // C++
}   }
```



$x = 4$

① Pos

Deletion at head, middle, last

② Size

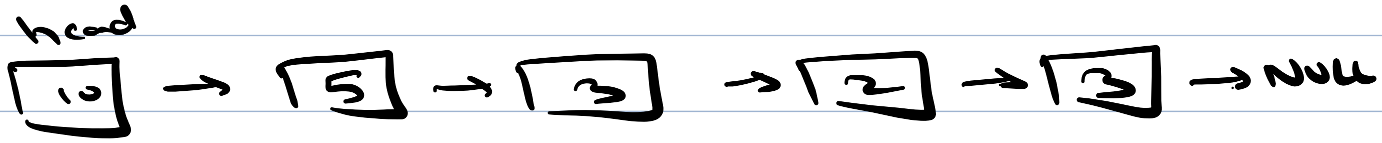
If LL is empty, 1 size

Deleting from empty LL

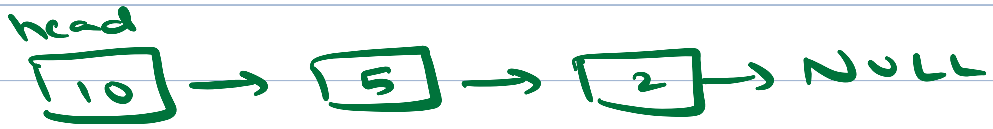
Deleting head node

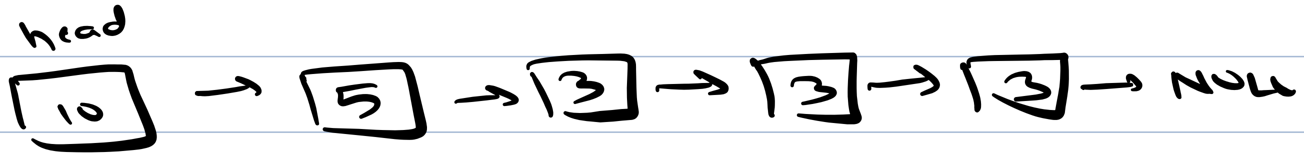
middle

3 ⓑ Delete multiple occurrences of x



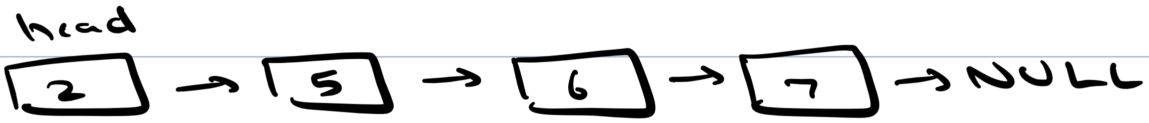
$x = 3$

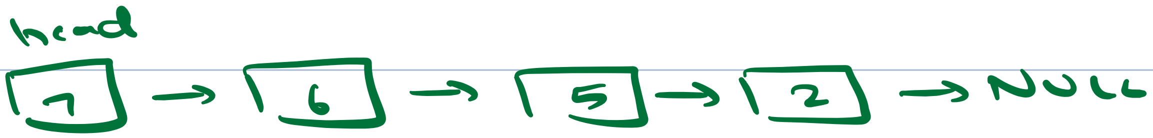




head
10
5
3
3
3
NULL

Prob 3 : Reverse given LL





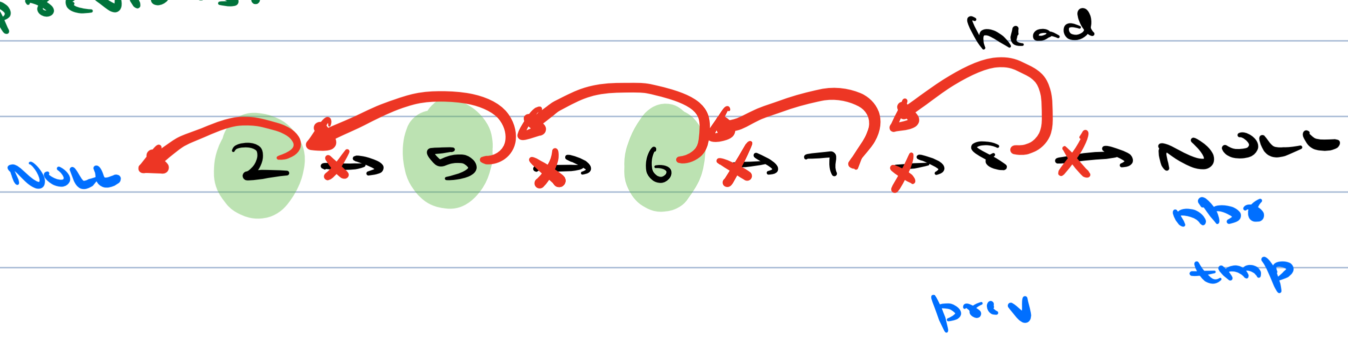
head

2 → 5 → 6 → 7 → NULL

head

NULL ← 2 ← 5 ← 6 ← 7

Go to every node, make it point to previous.



```
< Node tmp = head, prev = NULL

    Node nbr = temp.next
    tmp.next = prev
        prev = tmp
        tmp = nbr
```

When temp reaches NULL (end of LL)
prev reaches last node (head of

reversed list)

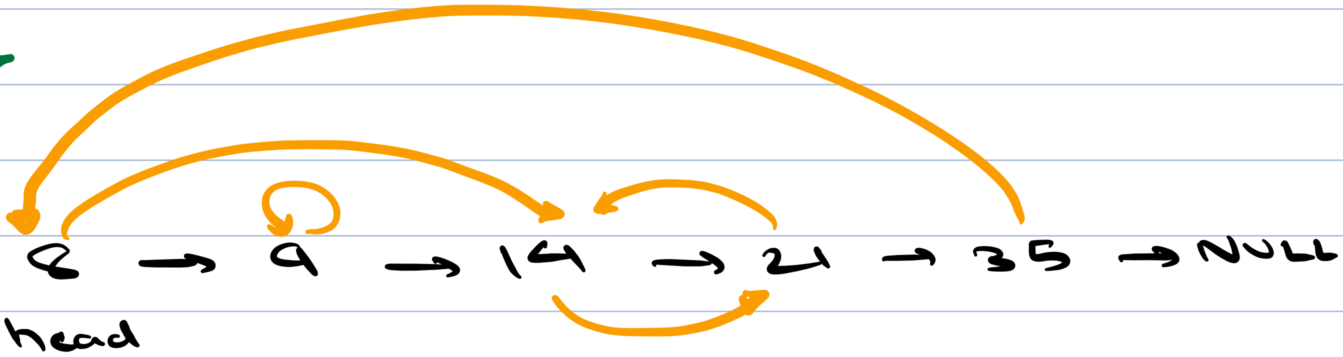
head = prev

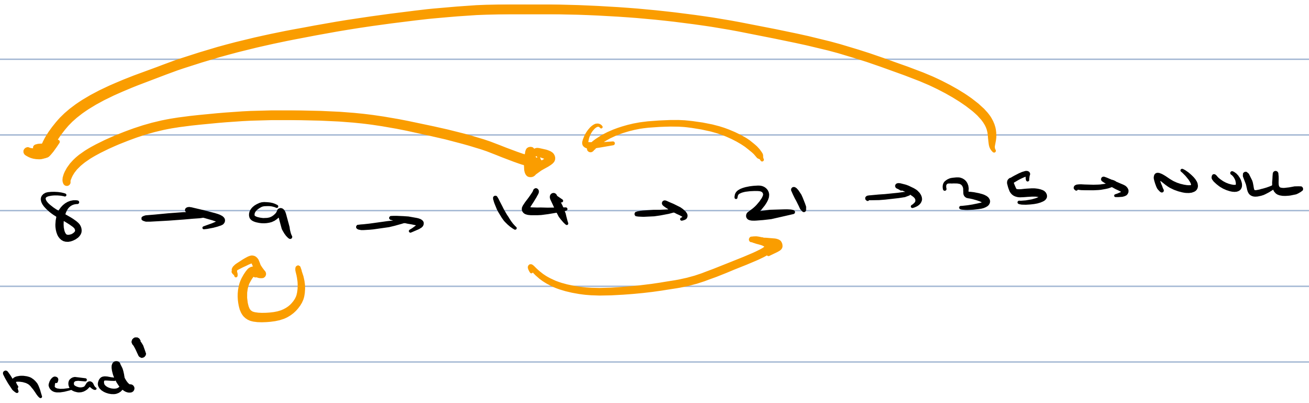
```
void reverse (Node head) {
    Node tmp = head, prev = NULL

    while (tmp != NULL) {
        Node nbr = temp.next
        tmp.next = prev
          prev = tmp
            tmp = nbr
    }

    head = prev
}
```

TC : O(N)
SC : O(1)

Clone / Deep copy a LL

```
class Node {
    int data
    Node next, random
}
```

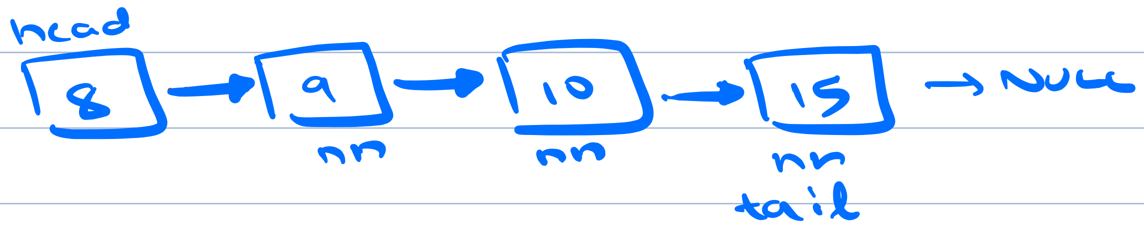




head

8 → 9 → 10 → 15 → NULL

tmp



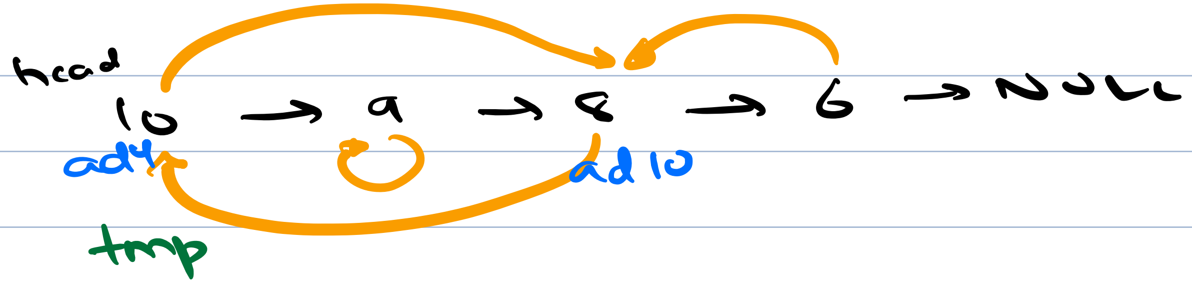
① create nn

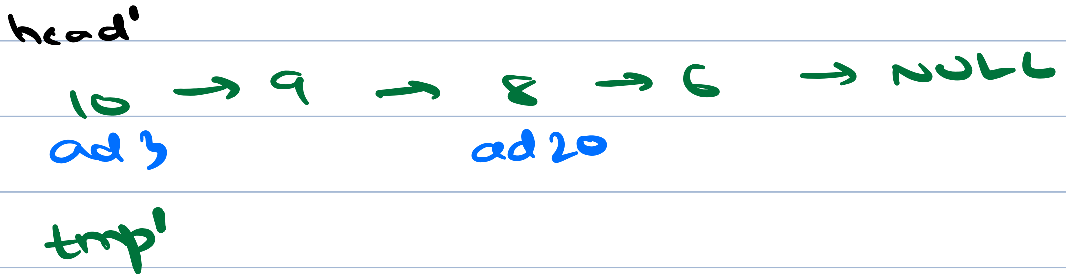
② tail. next = nn

tail = nn

clone Normal list (Node head)

```
Node head' = new Node (head.data)
Node tail = head'

Node tmp = head. next
while (tmp ! = NULL) {

    Node nn = new Node (temp.data)
    tail. next = nn
    tail = nn
    hm.put (tmp, nn)

    tmp = tmp.next
```





TC : O(N)

SC : O(N)
↓
Hashmap

Hashmap

Old Node	New Node
ad4	ad3
ad10	ad20

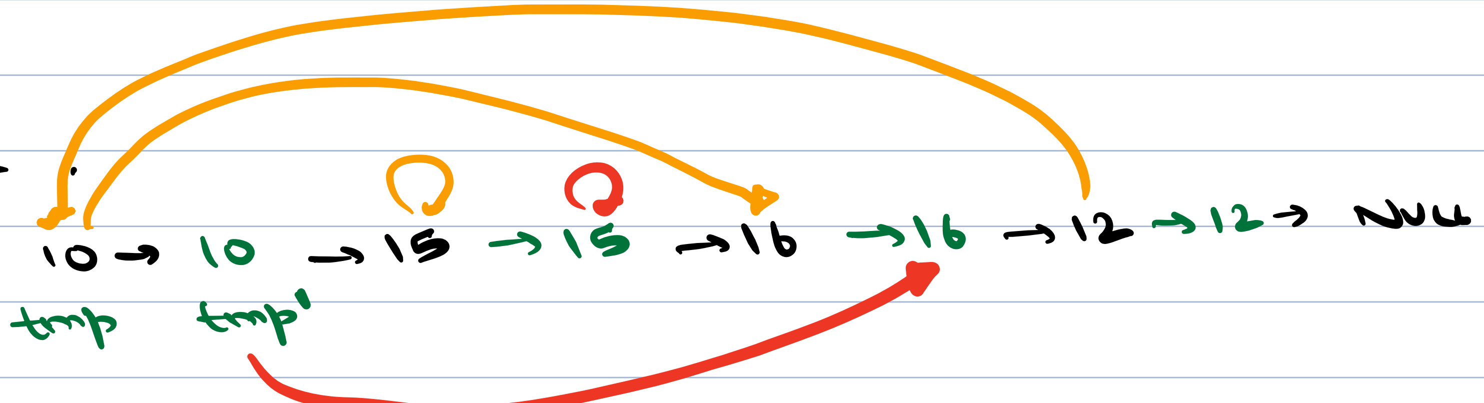
---

3 step solution.

1. head
10 → 15 → 16 → 12 → NULL

10 → 10 → 15 → 15 → 16 → 16 → 12 → 12 → NULL

2.



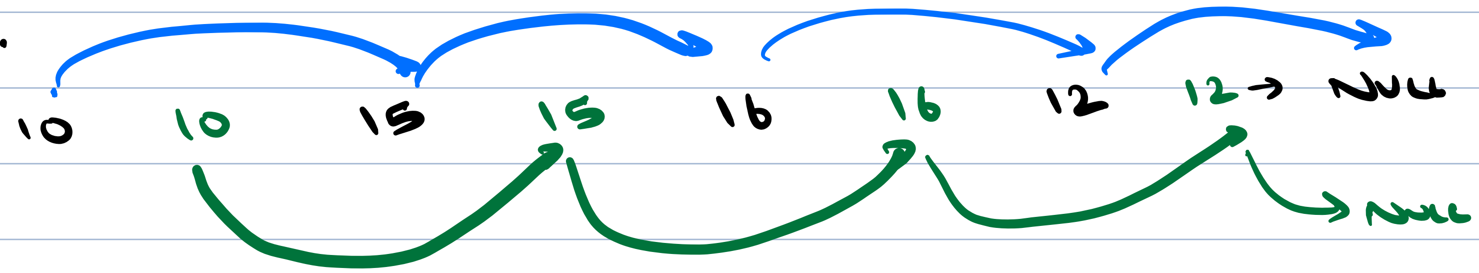
10 . random = 16

tmp' .random = tmp. random . next

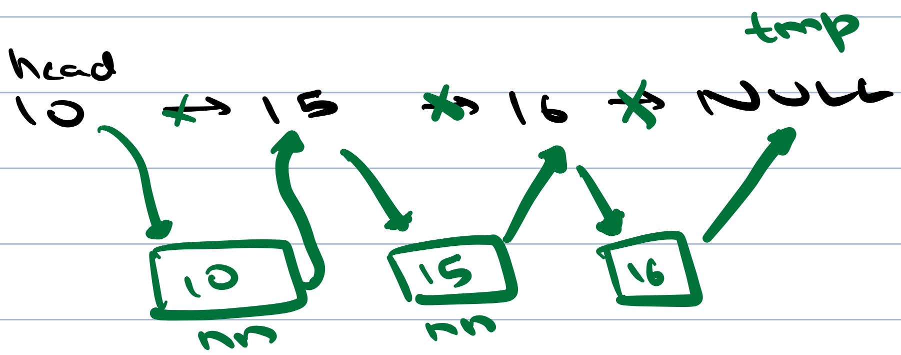
old node = tmp

New node = tmp'

3.



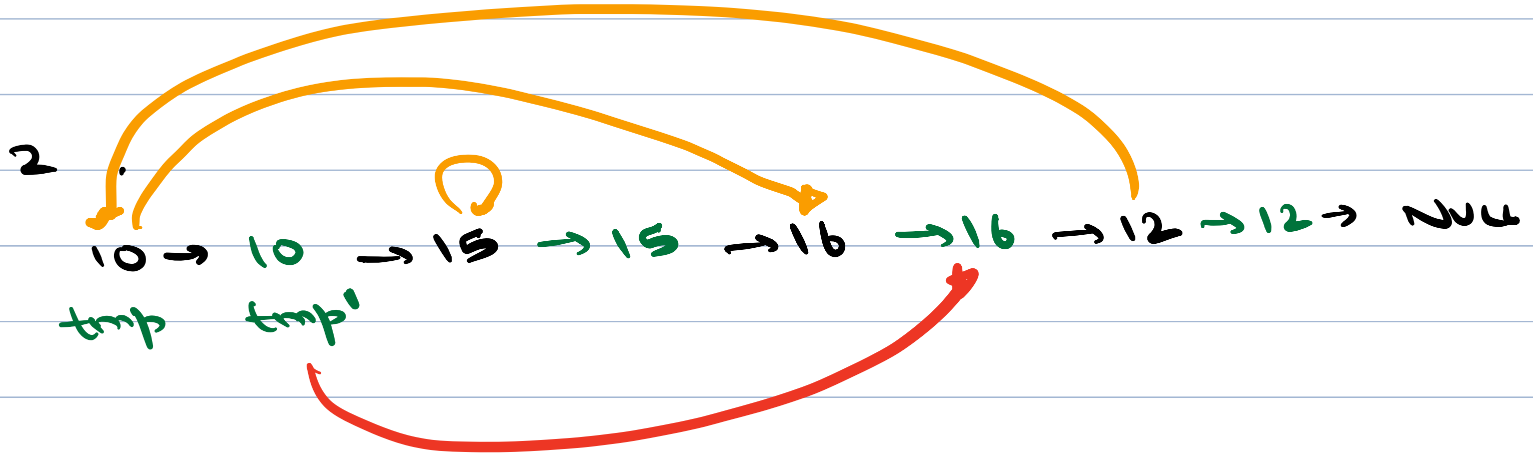
Step I:



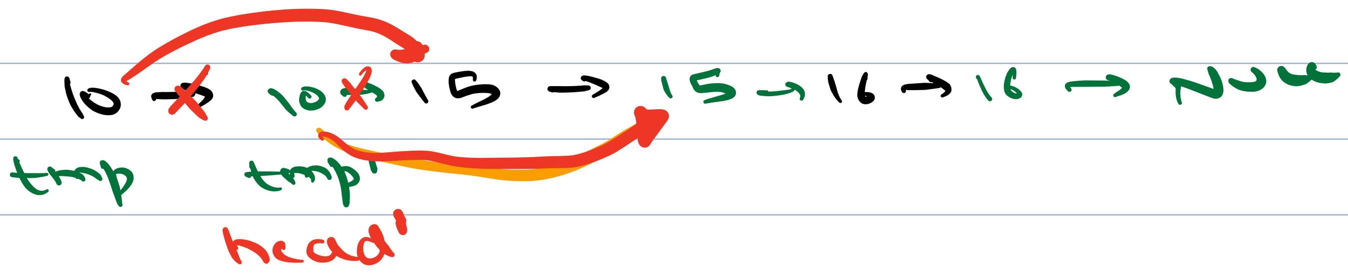
```
Node tmp = head
while (tmp != NULL) <
    Node nn = new Node (tmp.data)
    nn.next = tmp.next
    tmp.next = nn
    tmp = nn.next
>
```

Step 2 : For all new nodes, find their 'random' partners

```
Node tmp = head, tmp' = head.next
while (tmp != NULL) {
    if (temp.random != NULL) {
        tmp'.random = tmp.random.next
    }
    tmp = tmp.next.next
    if (tmp != NULL)
    tmp' = tmp'.next.next
}
```



Step 3 : Detangling both LL



```
Node tmp = head, tmp' = head.next
Node head' = tmp'
while (tmp != NULL) {
    tmp.next = tmp'.next
    if (tmp'.next != NULL) {
        tmp'.next = tmp'.next.next
    }
    tmp = tmp.next
    tmp' = tmp'.next
}
return head'
```

TC : O(N)
SC : O(1)