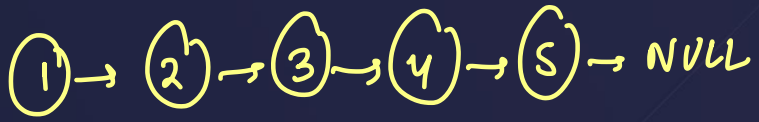
Linked List

Part – 4

Raghav Garg

Revisiting the limitations of "Singly Linked List"



T.C. of get, delete

- If you are at a particular node, then you can never go back

Solution : Doubly linked list

Introducing Doubly Linked List

Can we make our insertion and deletion more efficient?

Introducing Doubly Linked List

Node Class

```
class Node{
    int val;
    Node* next;
    Node* prev;
    Node(int val){
        this->val = val;
        this->next = NULL;
        this->prev = NULL;
    }
};
```

Note that we take up more memory

Implementation

Insertion and Deletion

Implementation

Insertion and Deletion

Some benefits of Doubly Linked List :

1) If you want to display LL in reverse order. (via Tail node)

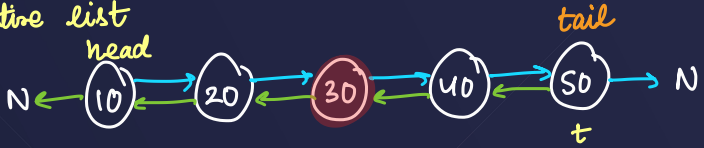

```
display(head){
    if(head == null) return;
    cout << head -> val;
    display(head -> next);
}
```

normal

```
display(head){
    if(head == null) return;
    display(head -> next);
    cout << head -> val;
}
```

reverse

2) If any node of list is given, we can traverse through the entire list




```
while (t → prev != NULL){
|     t = t → prev;
}
Node* head = t;
```

```
while (t → next != NULL)
|   t = t → next;
}
Node* tail = t
```

Doubly LL Class :

1) insert (tail, head, index)

2) delete (tail, head, index)
   - tail → O(1)
   - head → O(1)
   - index → O(n)

3) get (tail, head, index)

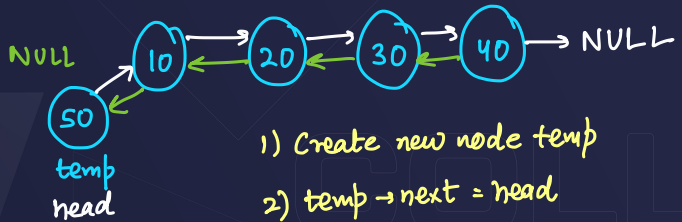
Insert At Tail

insertAtTail(50);

1) Create Node temp
2) tail → next = temp
3) temp → prev = tail
4) tail = temp

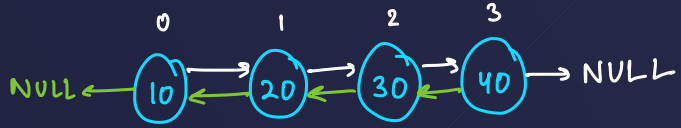
Insert At Head

insertAtHead (50)

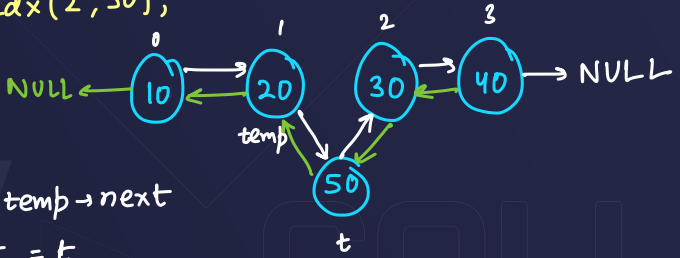


1) Create new node temp
2) temp → next = head
3) head → prev = temp
4) head = temp

Insert At Index



insert At Idx(2, 50);



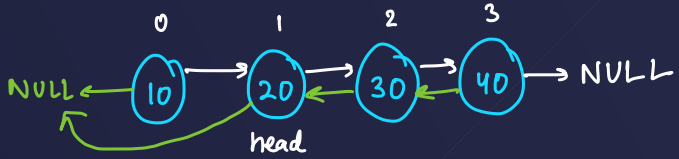
t → next = temp → next

temp → next = t

t → prev = temp

t → next → prev = t

Delete At Head :

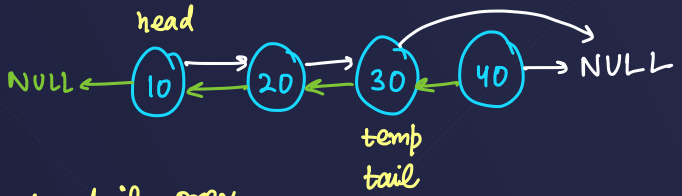


head = head → next;

* if(head != NULL) head → prev = NULL;

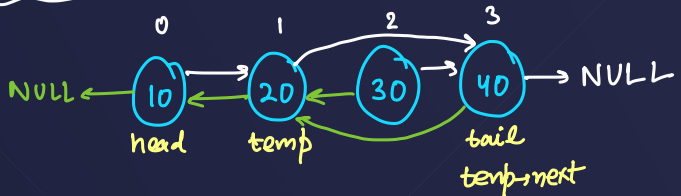
if (head == NULL) tail = NULL;

* Delete At Tail (Better than SLL)




```
Node* temp = tail -> prev
temp -> next = NULL;
tail = temp;
```

Delete at Index



deleteAtIndex(2);

1) temp = head
2) traverse temp to idx-1
3) temp→next = temp→next→next
4) temp→next→prev = temp

Get At Index (Optimised)

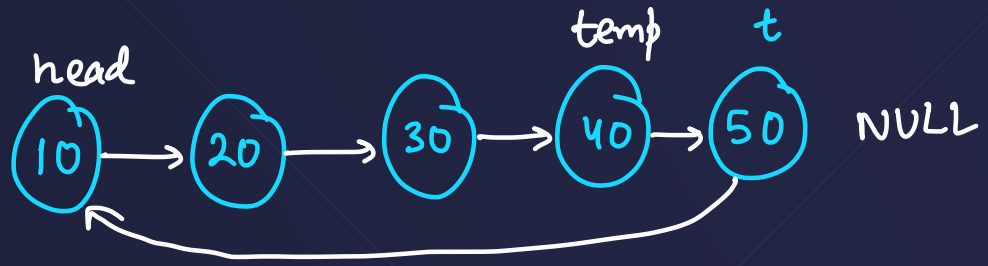
- Problem Statement : 1) you have a DLL of size 100 & head, tail, size
  2) you have to delete 80^{th} index element

M-I : Node* temp = head.
traverse temp 'idx' time

M-II : Node* temp = tail;
traverse temp (100-idx) times

Other types of Linked List

1) Circular SLL

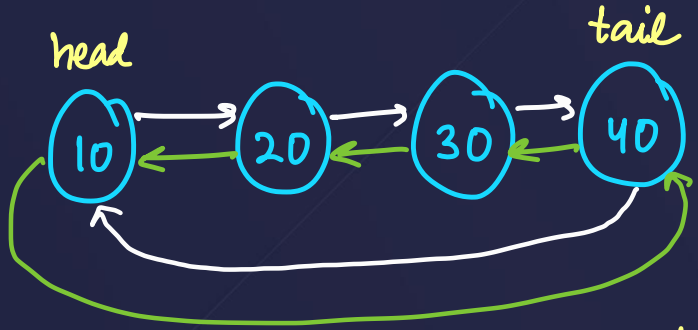



```
Node* temp = head;
while (temp→next != head)
|    temp = temp→next;
2
Node* t = new Node(val);
temp→next = t;
t→next = head;
```

```
Node* add(Node* head, int val){

}
```

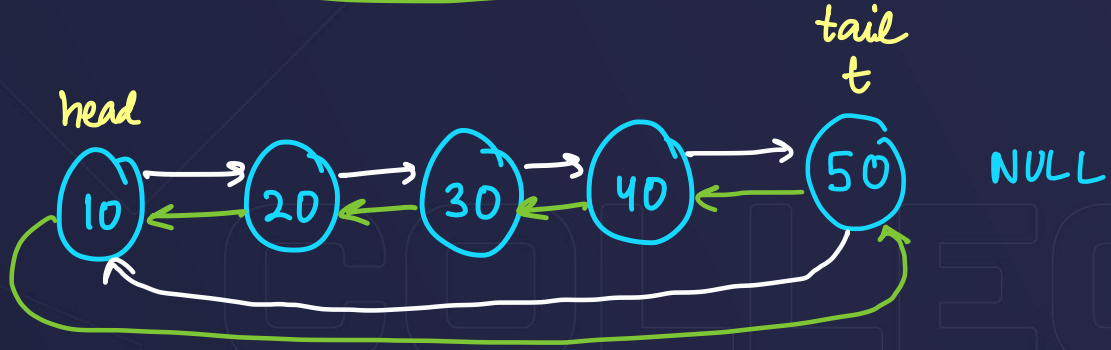
Other types of Linked List

2) Circular Doubly LL

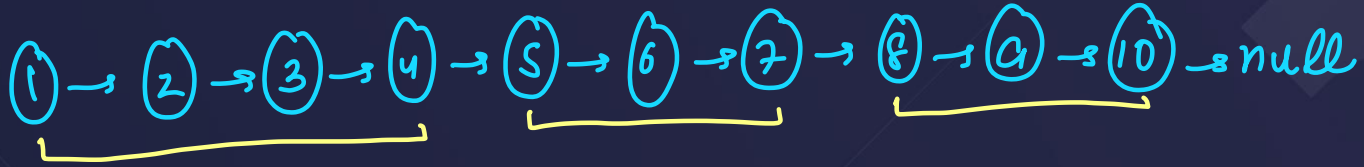


insert At Tail (50)

```
tail->next = t;
t->prev = tail;
t->next = head;
head->prev = t;
tail = t
```

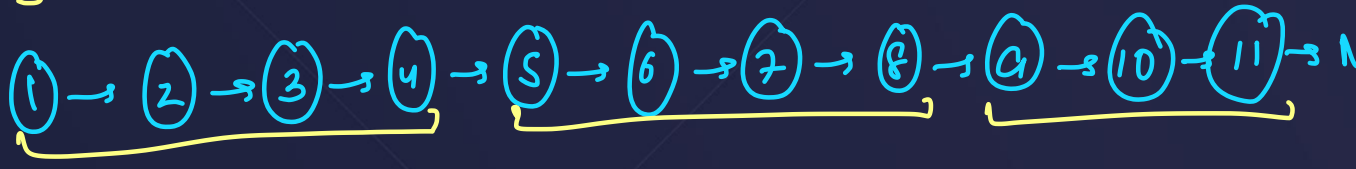


Ques: Split Linked List in Parts [Leetcode - 725]

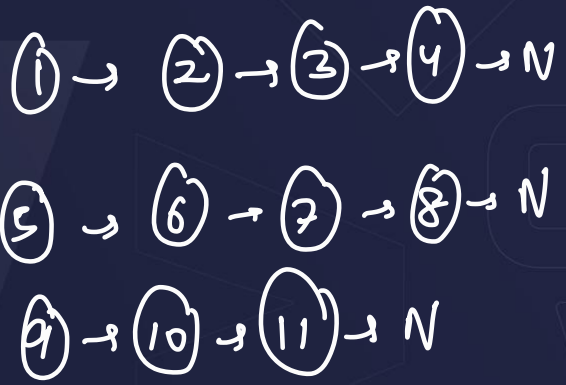


K = 3

$\frac{n}{k} = \frac{10}{3} = 3$

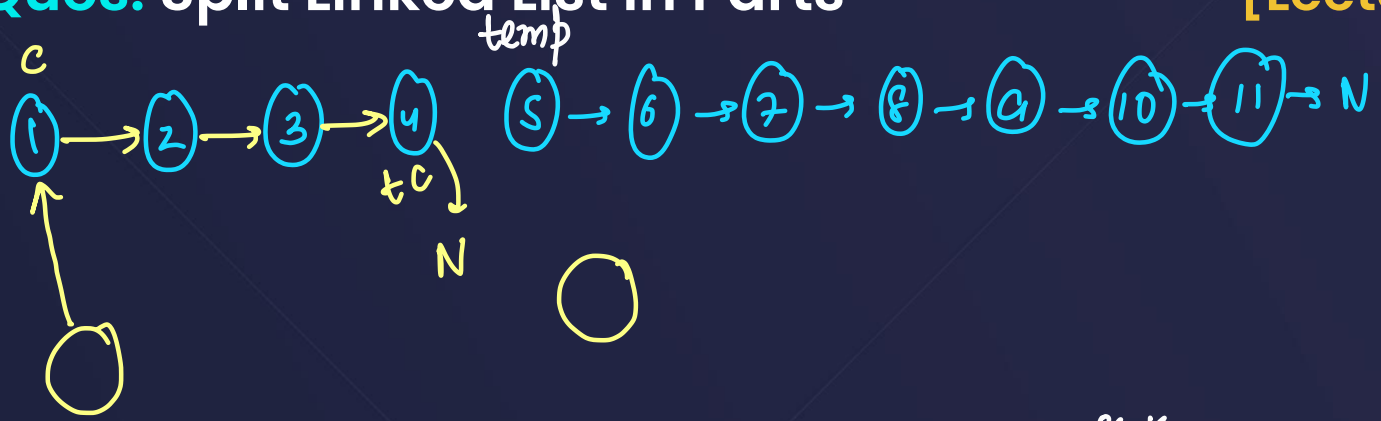


K = 3



vector<ListNode*> v

Ques: Split Linked List in Parts [Leetcode - 725]

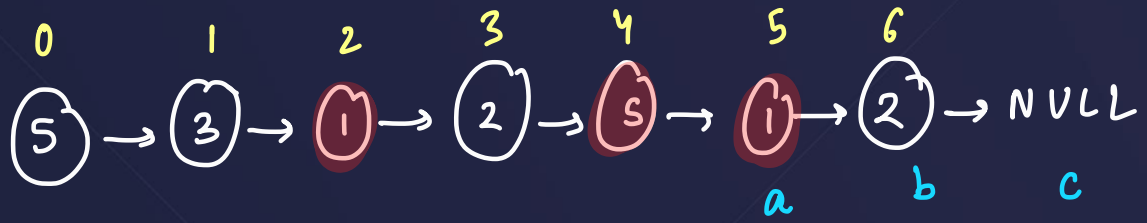


K = 3
↓
no. of parts

int rem = n % K
= 11 % 3 = 2
↓
ece parts jinki length loonga vo ek extra hogi

Length of parts = $\frac{n}{K}$, $\frac{n}{K}+1$, 0
↓
$\frac{11}{3} = 3$

Ques: Find the Minimum and Maximum Number of Nodes between Critical Points [Leetcode - 2058]



mind = ~~INT_MAX~~ 3 2 1

fidx = -1 -1 2 4

sidx = -1 2 4 5

max distance = sidx - fidx = 5 - 2 = 3

fidx = sidx

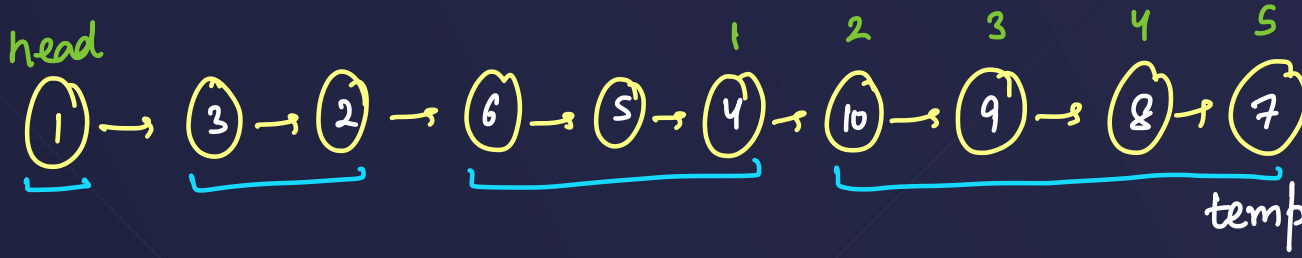
sidx = idx

int d = 3 2 1

fidx = -1 2

sidx = -1 4 5

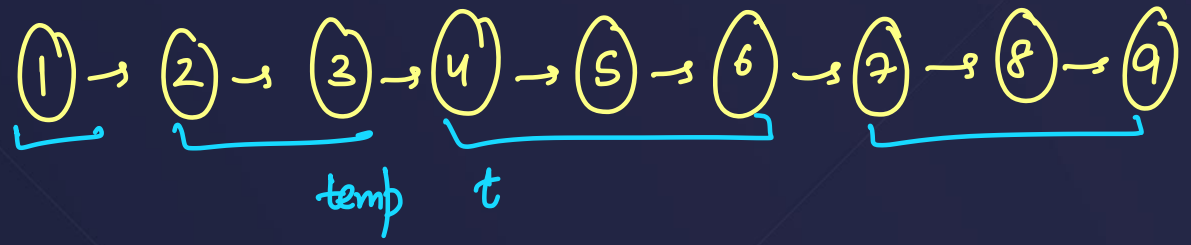
Ques: Reverse Nodes in Even Length Groups [Leetcode - 2074]



gap = 1 2 3 4

reverseBw(temp, 2, 2+gap);

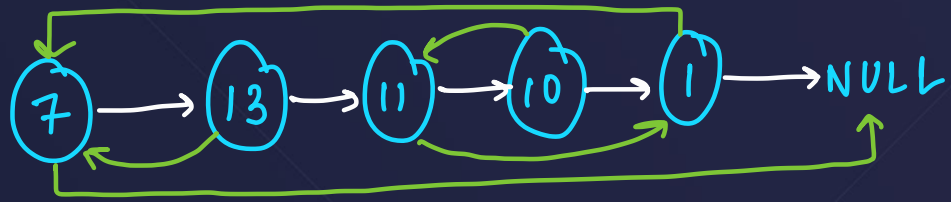
Ques: Reverse Nodes in Even Length Groups [Leetcode - 2074]

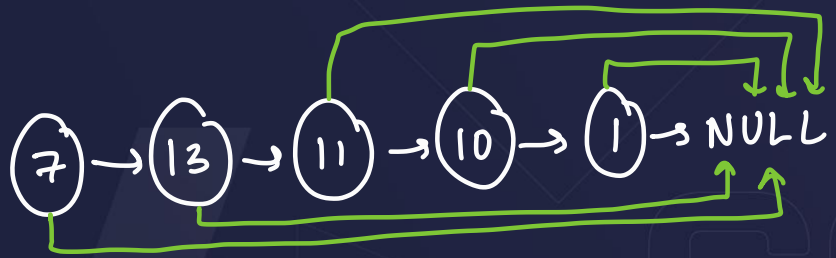


if (remLen < gap+1) gap = remLen-1

Ques: Copy List with Random Pointer [Leetcode - 138]

Step-1 : Create a deep copy (without random)

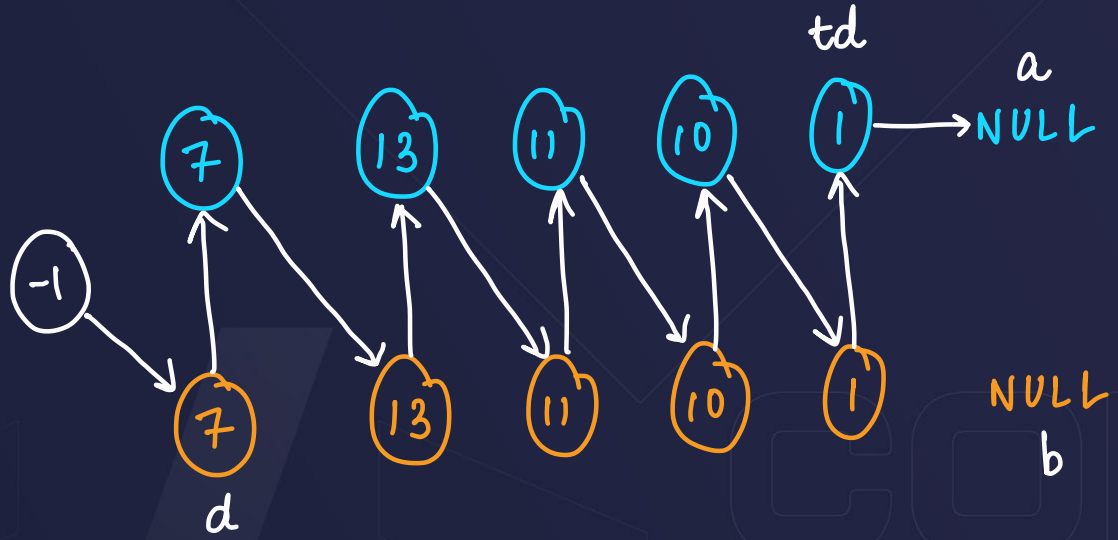





```
class Node{
    int val;
    Node* next;
    Node* random;
};
```

Ques: Copy List with Random Pointer [Leetcode - 138]

Step-2 : Create alternate connections (merge)

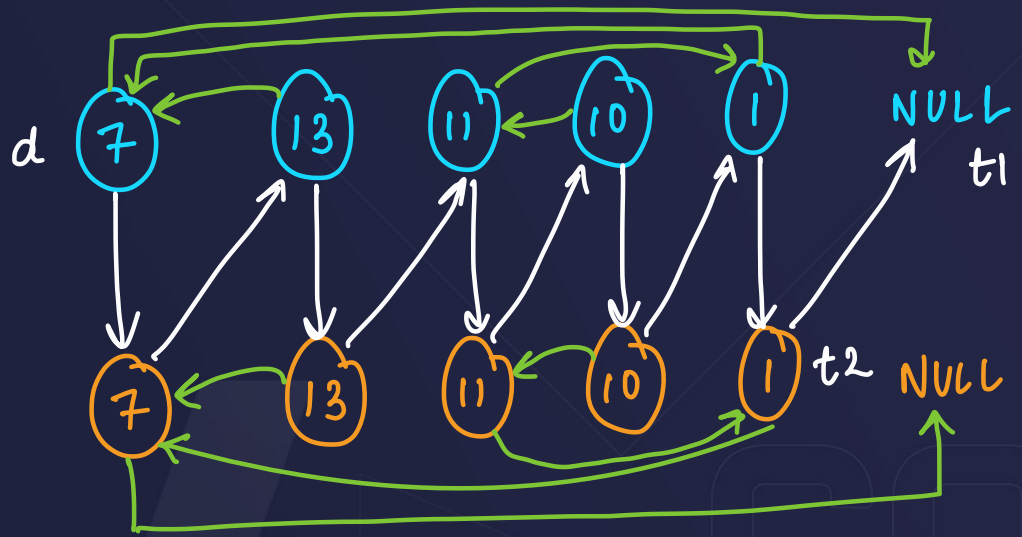


[7, N], [13, 7], [11, 1] [10, 11]
[1, 7]

Ques: Copy List with Random Pointer [Leetcode - 138]

Step-3 : Assigning random pointer of duplicate

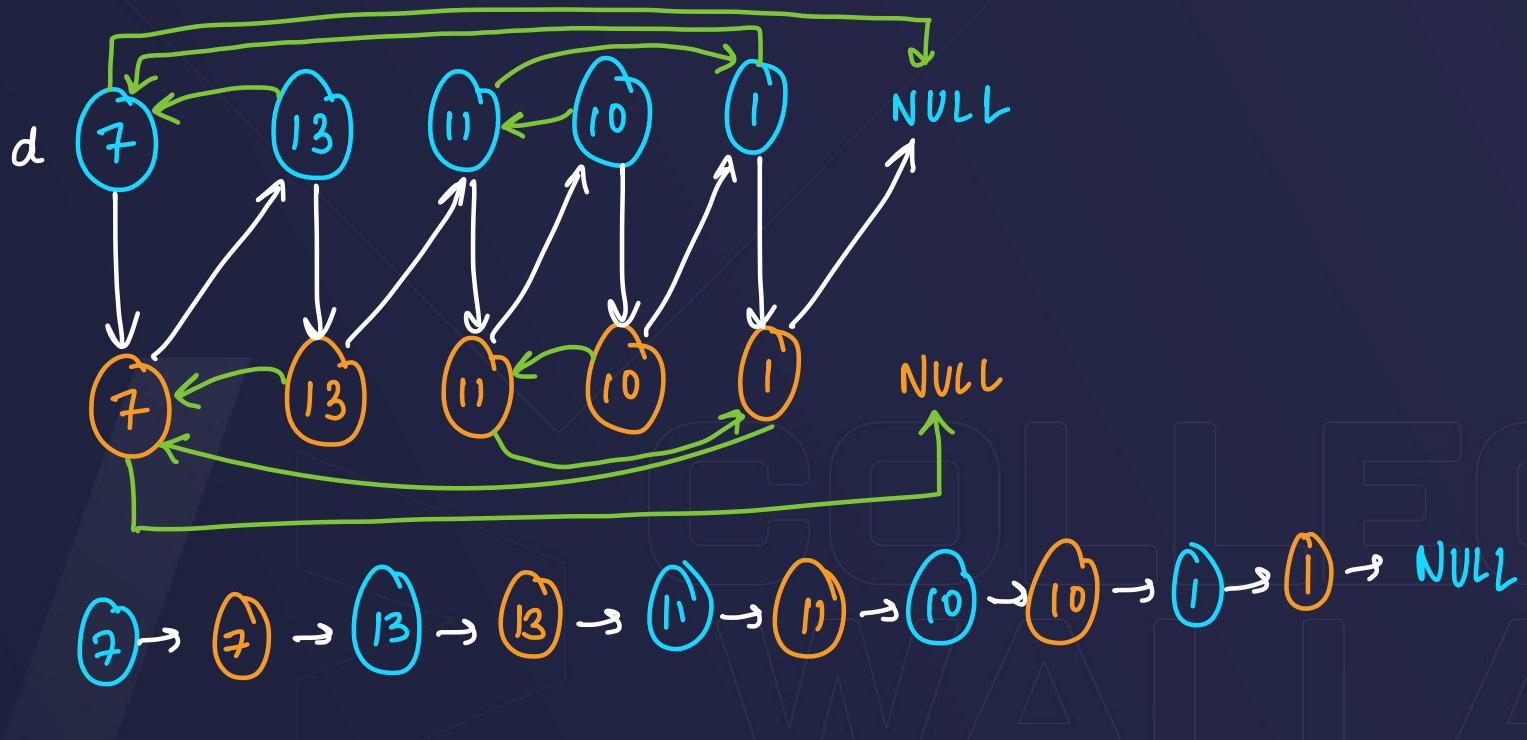
[7, N], [13, 7], [11, 1] [10, 11]
[1, 7]




```
t2 = t1->next
if(t1->random) t2->random = t1->random->next
t1 = t1->next->next
```

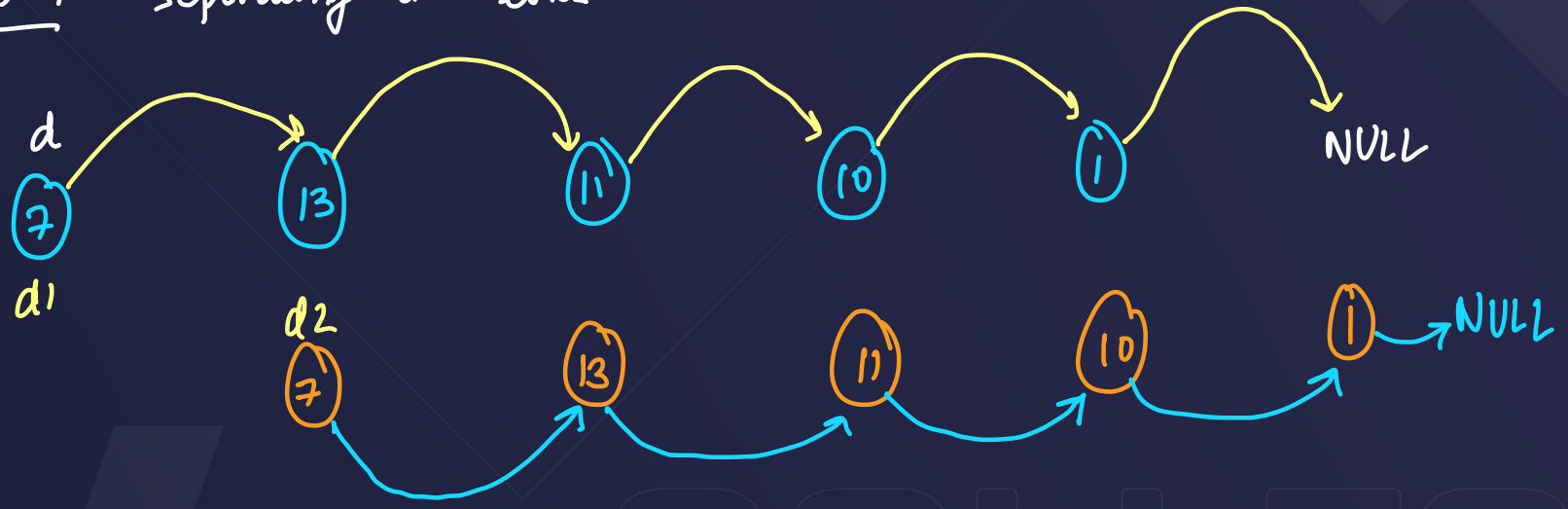
Ques: Copy List with Random Pointer [Leetcode - 138]

Step-4 Seperating the lists



Ques: Copy List with Random Pointer [Leetcode - 138]

Step-4 Seperating the lists

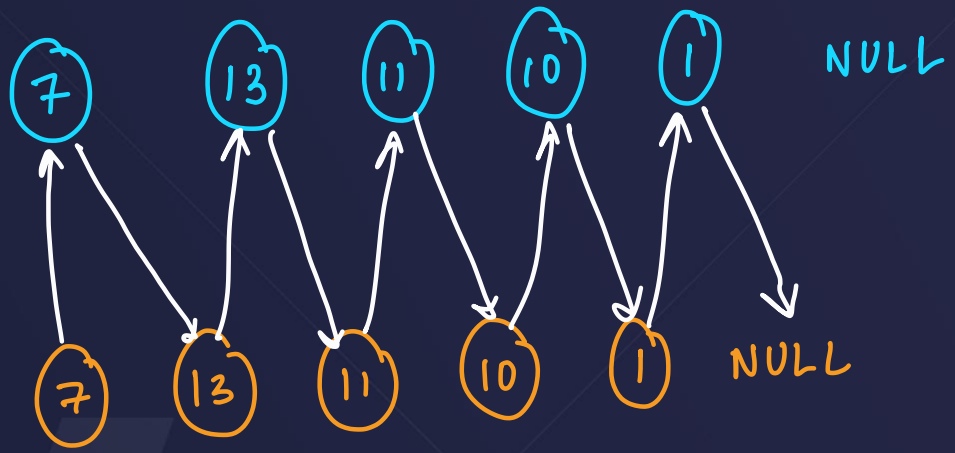


t1 → next = t
t = t → next
t1 = t1 → next

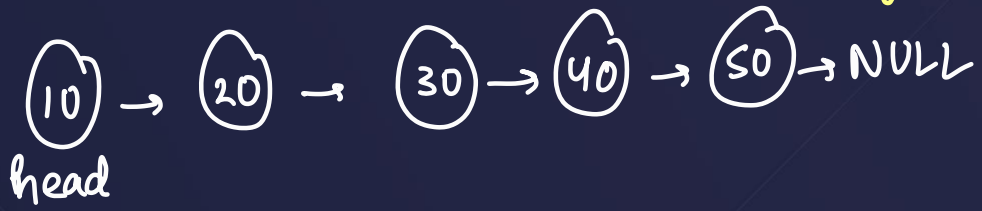
t2 → next = t
t = t → next
t2 = t2 → next

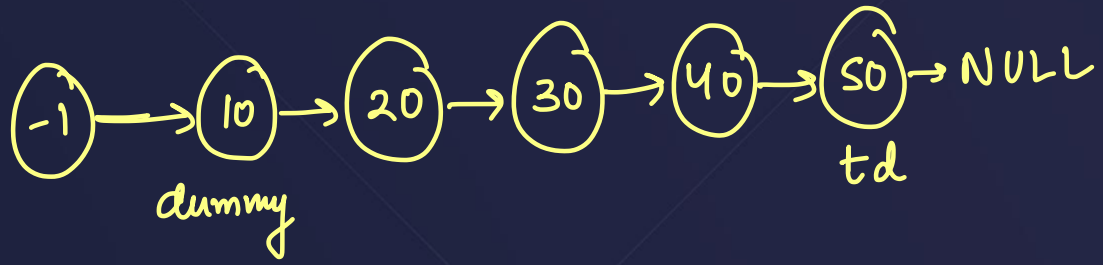
t1 → next = NULL
t2 → next = NULL

Ques: Copy List with Random Pointer [Leetcode - 138]

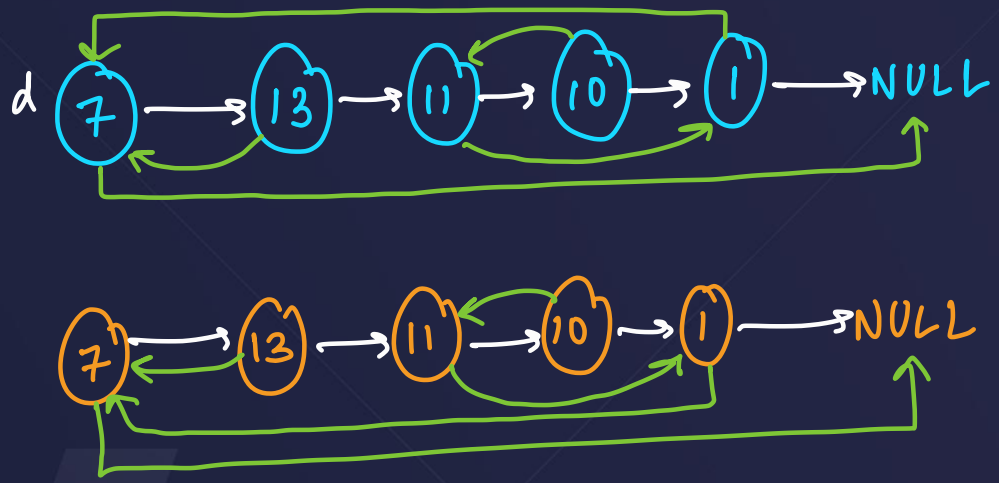


Ques: Copy List with Random Pointer [Leetcode - 138]



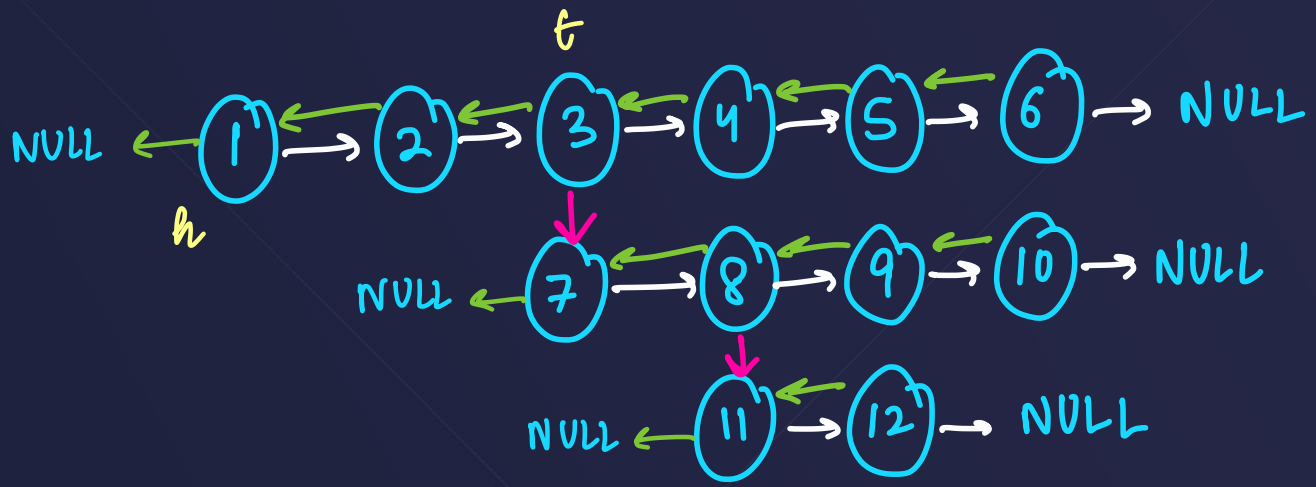


Ques: Copy List with Random Pointer [Leetcode - 138]



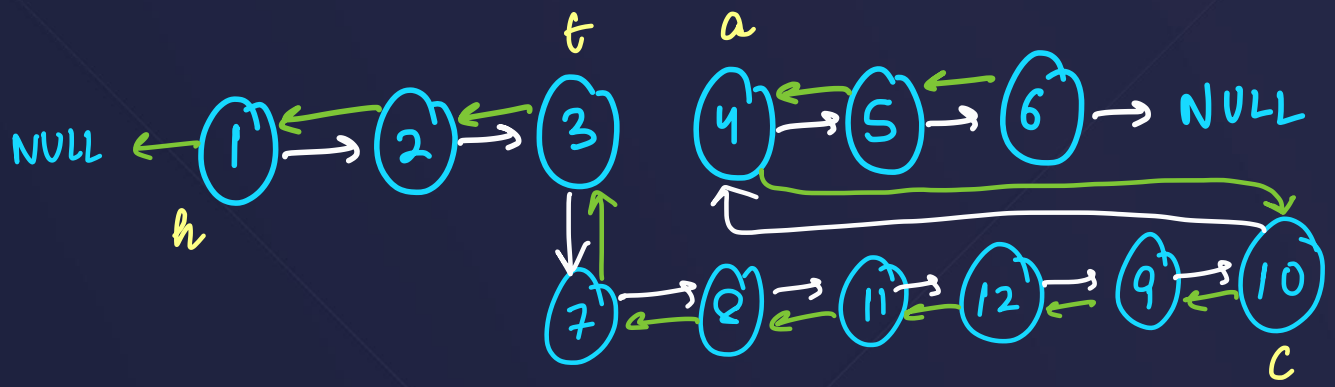

```
t2 = t1 -> next
if (t1 -> random != NULL)  t2 -> random = t1 -> random -> next
t1 = t1 -> next -> next
```

Ques: Flatten a Multilevel Doubly Linked List [Leetcode - 430]



Hint :
Recursion

Ques: Flatten a Multilevel Doubly Linked List [Leetcode - 430]

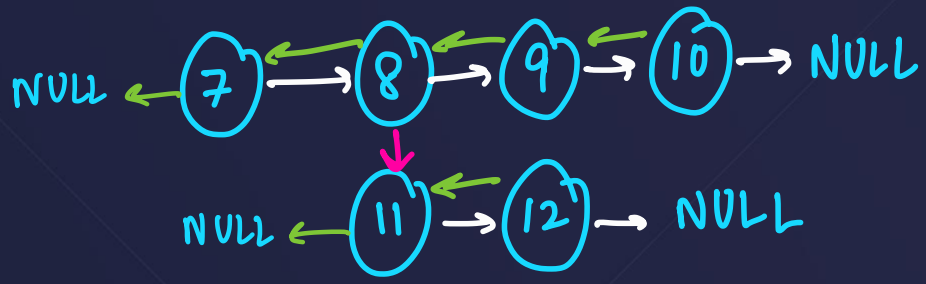


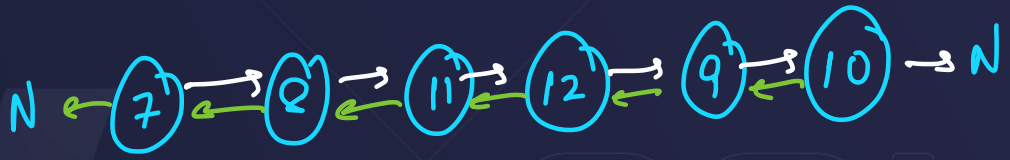
Hint:
Recursion

```
a = t→next
c = t→child;
c = flatten(c);
t→child = NULL

t→next = c
c→prev = t
```

```
while (c→next)
    c = c→next

c→next = a
a→prev = c
```

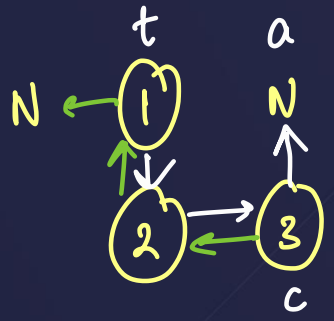
Ques: Flatten a Multilevel Doubly Linked List [Leetcode - 430]

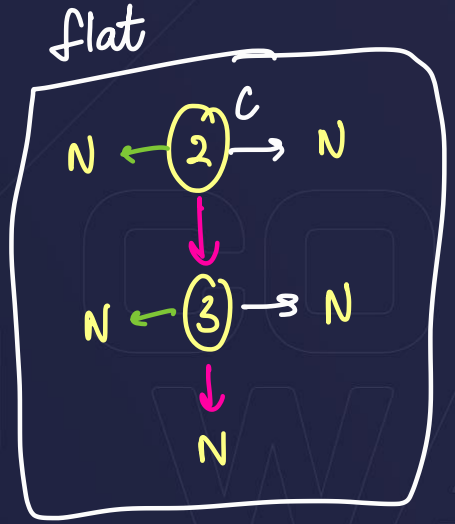




Ques: Flatten a Multilevel Doubly Linked List [Leetcode - 430]

```
Node* temp = head;
while(temp!=NULL){
    Node* a = temp->next;
    if(temp->child!=NULL){
        Node* c = temp->child;
        temp->child = NULL; // V IMP
        c = flatten(c); // recursion
        temp->next = c;
        c->prev = temp;
        while(c->next!=NULL){
            c = c->next;
        }
        c->next = a;
        a->prev = c; // error
    }
    temp = a;
}
return head;
```





Next Lecture

Introduction to Stacks! & Queues