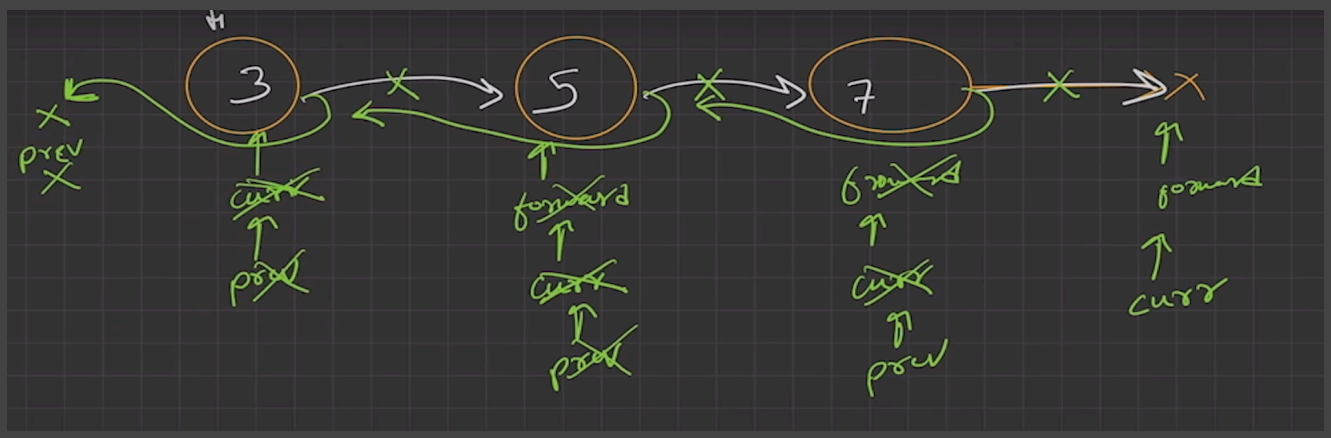
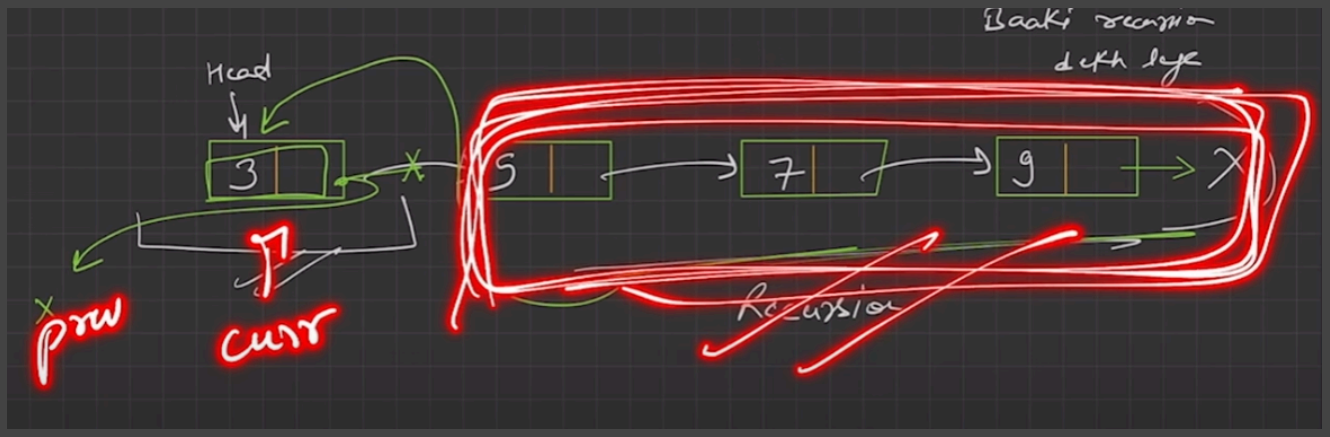
Linked-List

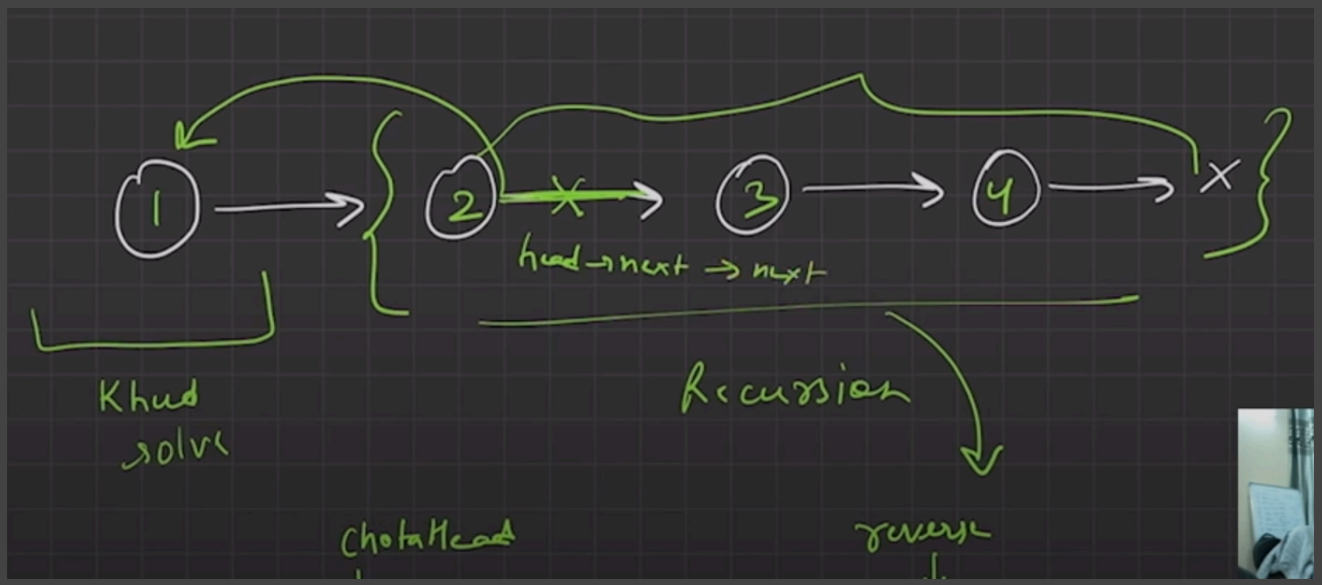
Reverse A Singly Linked List Approach-I:



Reverse A Singly Linked List Approach-II:

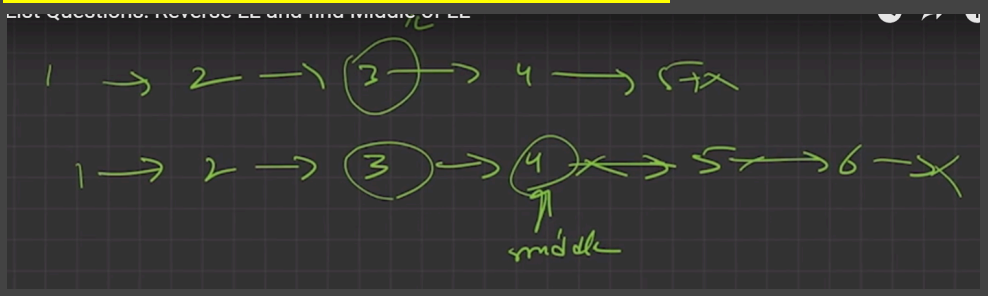


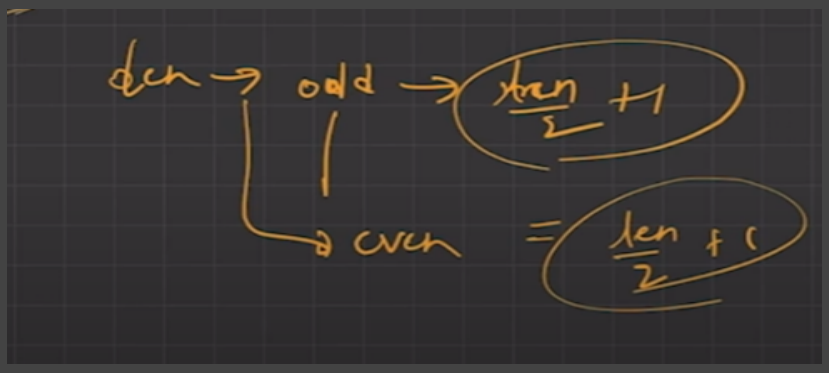
Reverse A Singly Linked List Approach-III:



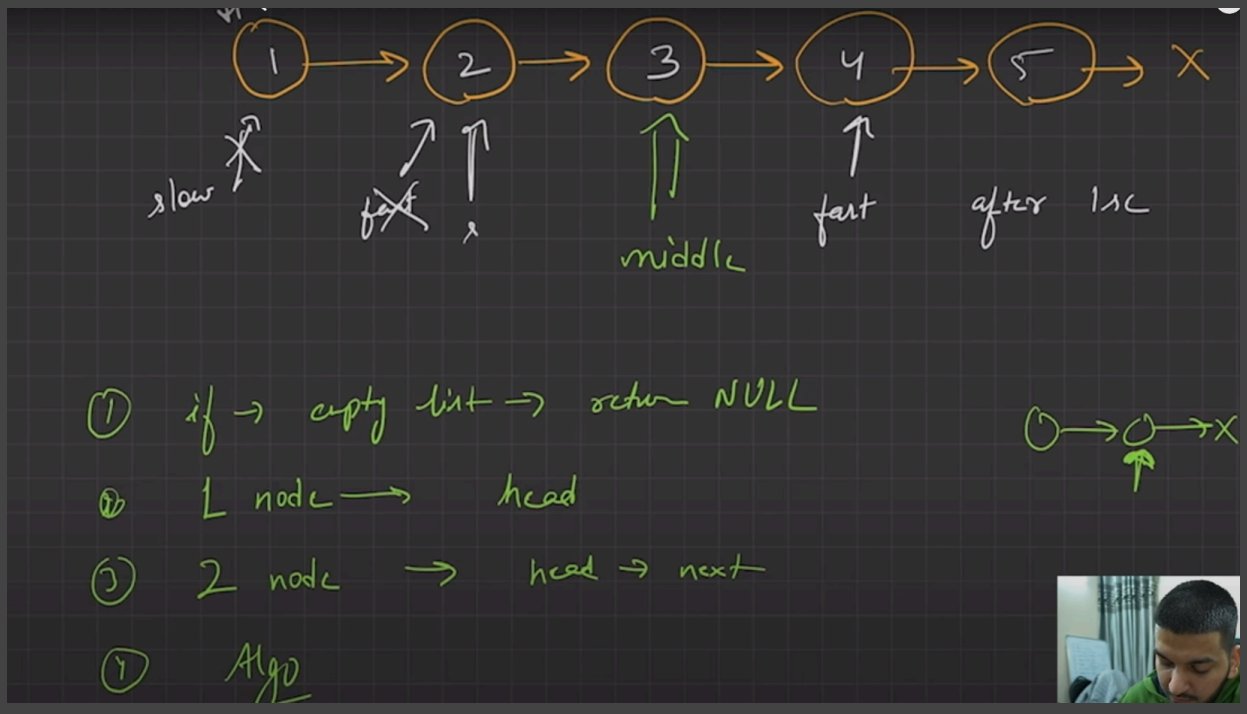
Reverse A Doubly Linked List

Find Middle In A Linked List Approach-I:

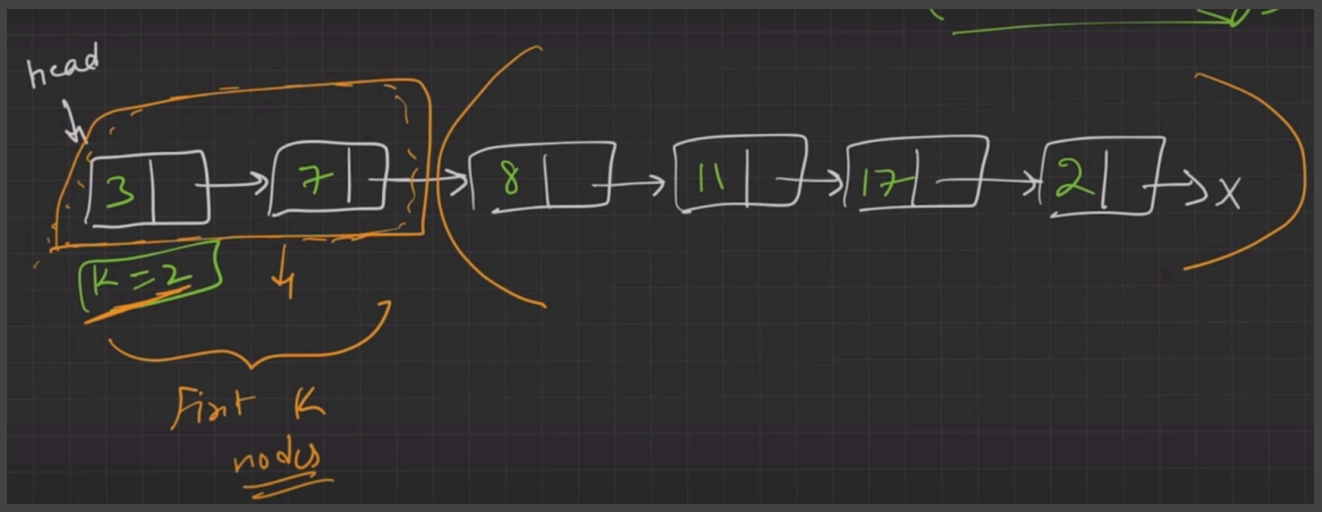


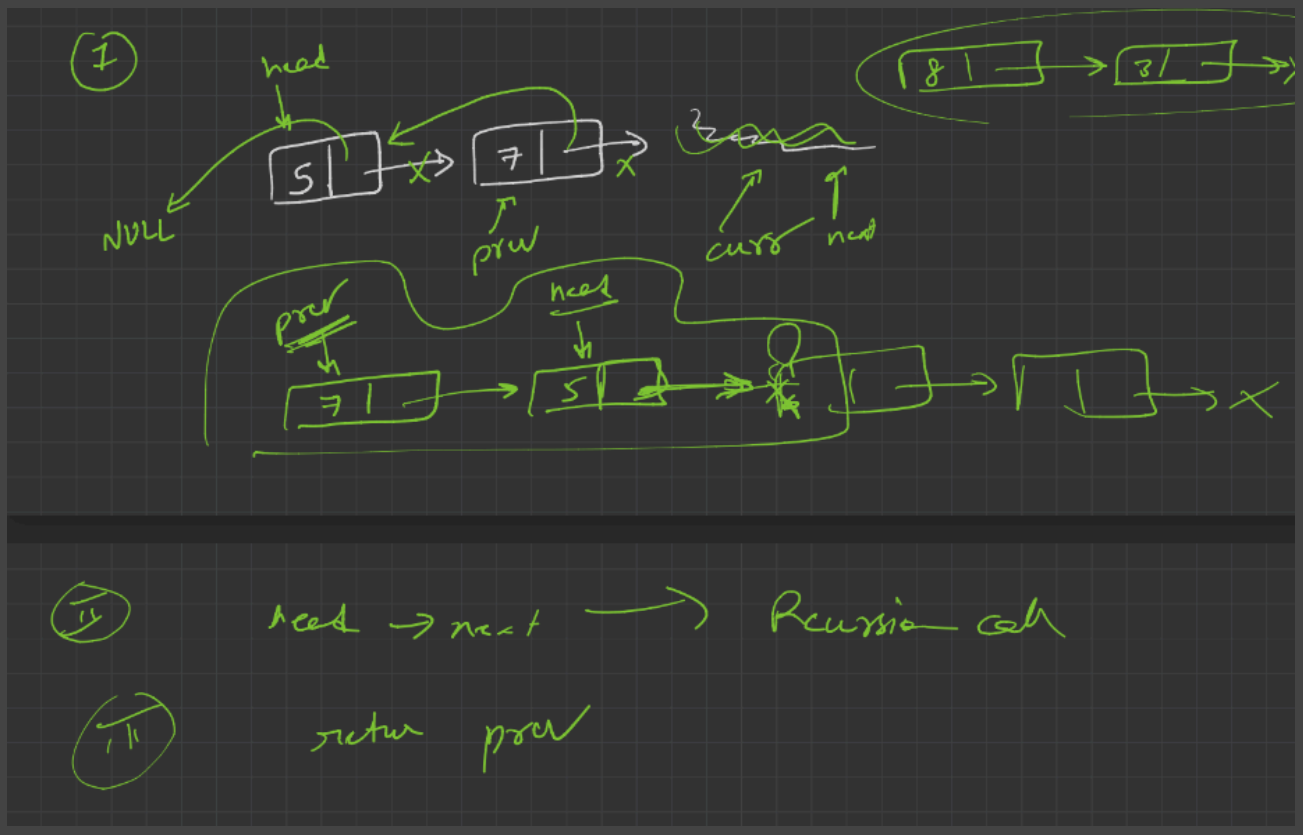


Find Middle In A Linked List Approach-II:

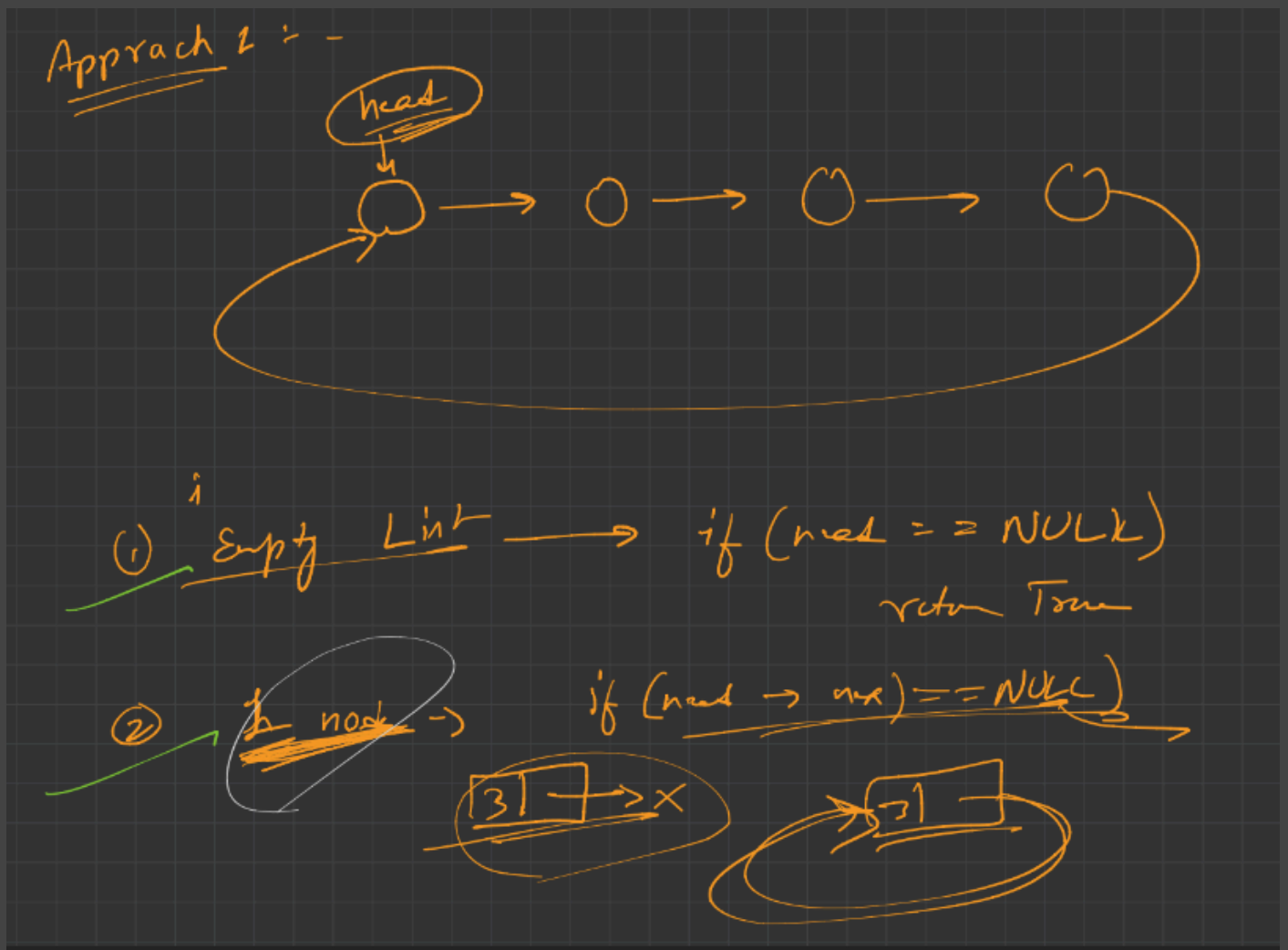


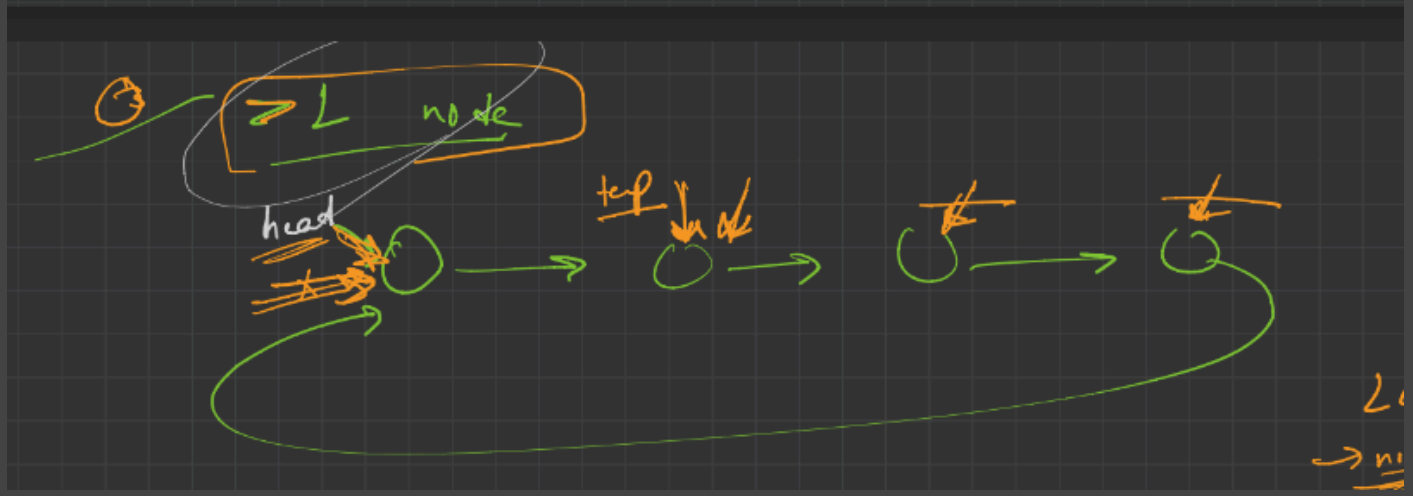
Reverse Linked List In 'K' groups:



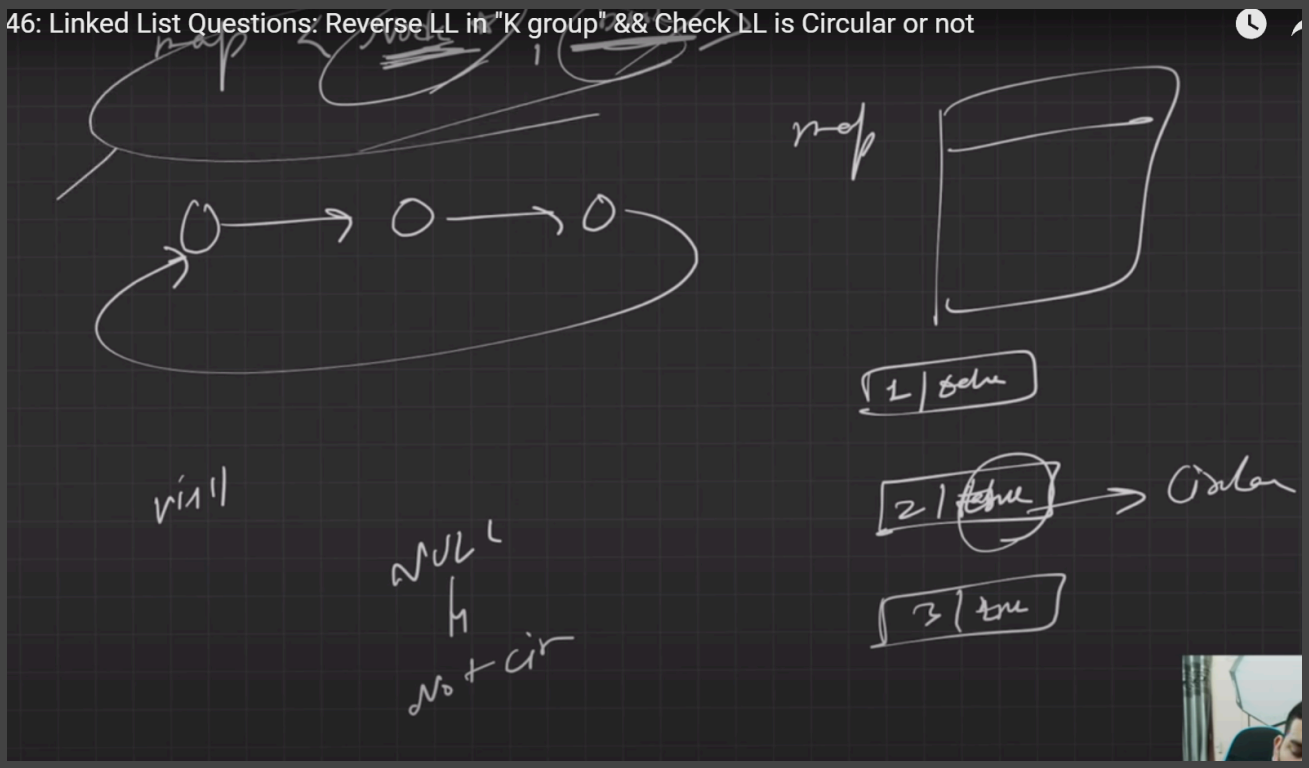


Check Whether A Linked List is Circular or Not Approach-I:

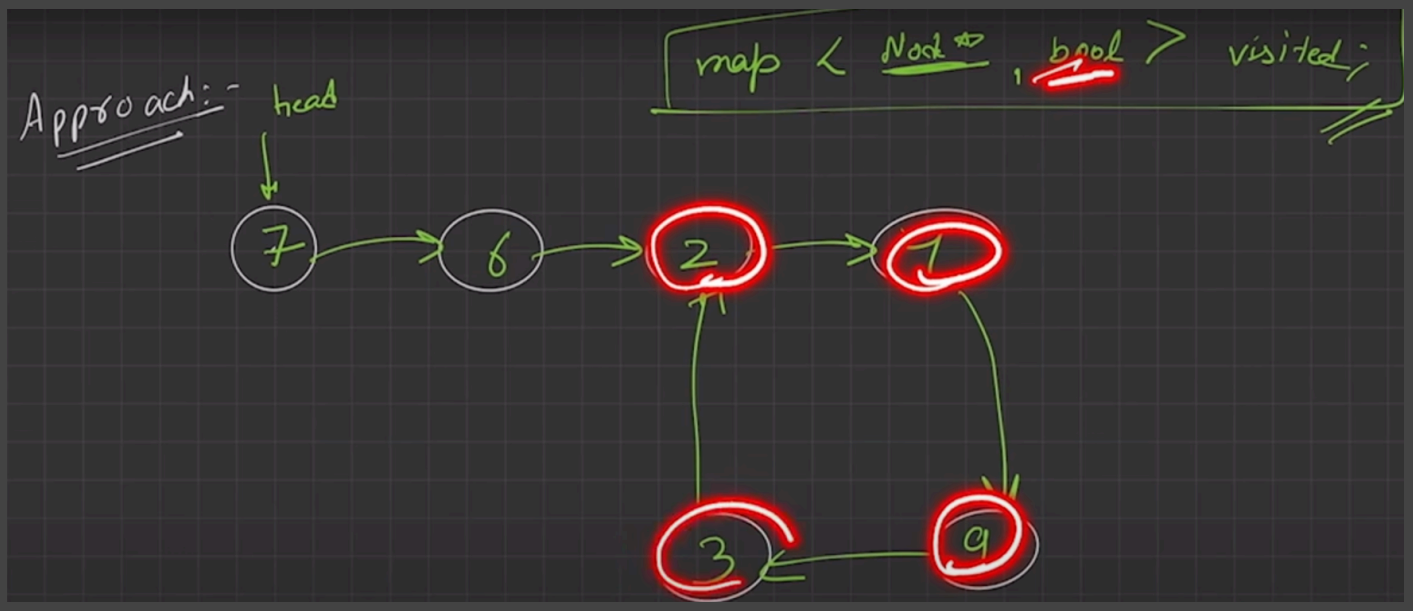




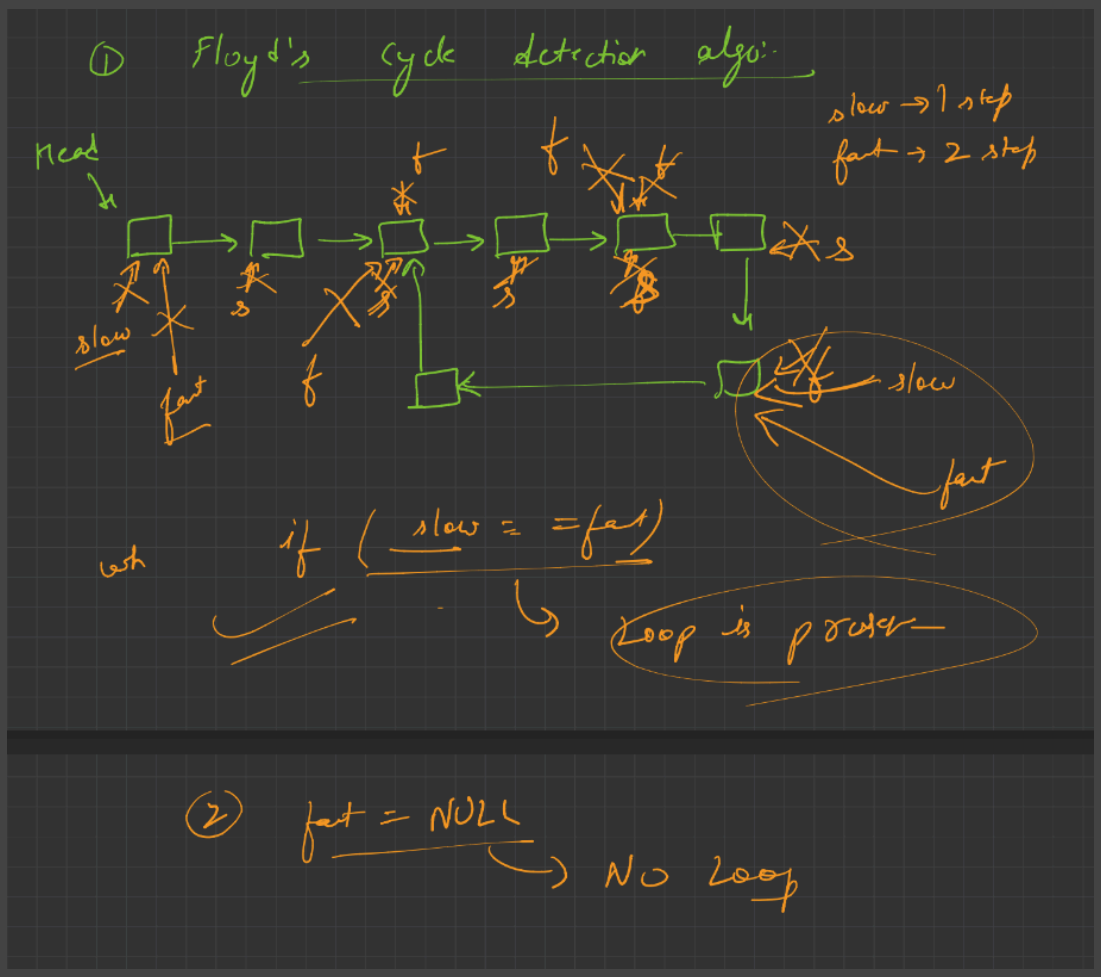
Check Whether A Linked List is Circular or Not Approach-II:

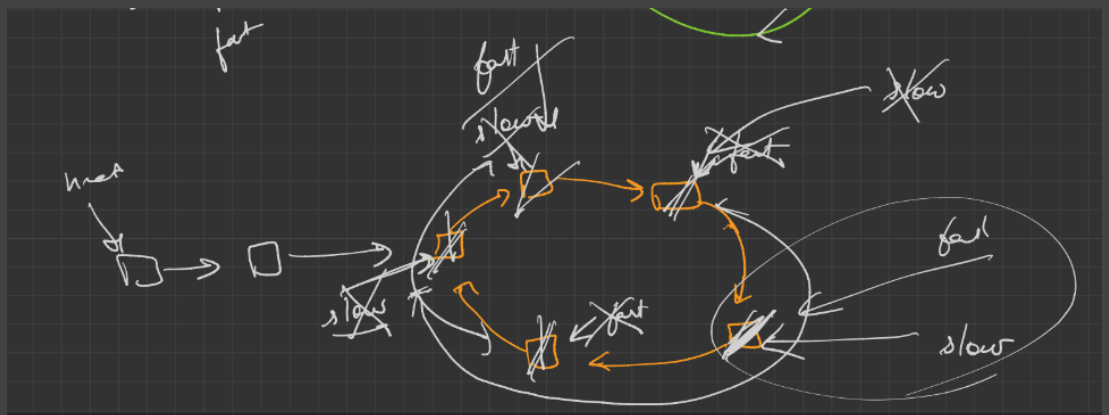


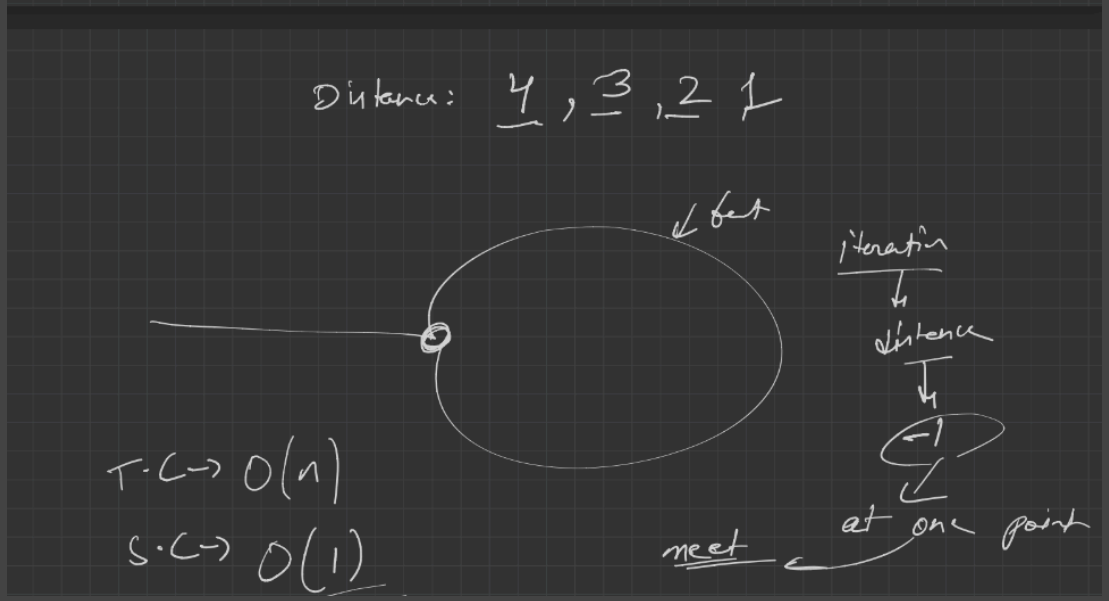
Detect Whether A Linked List Forms a Loop or Not:



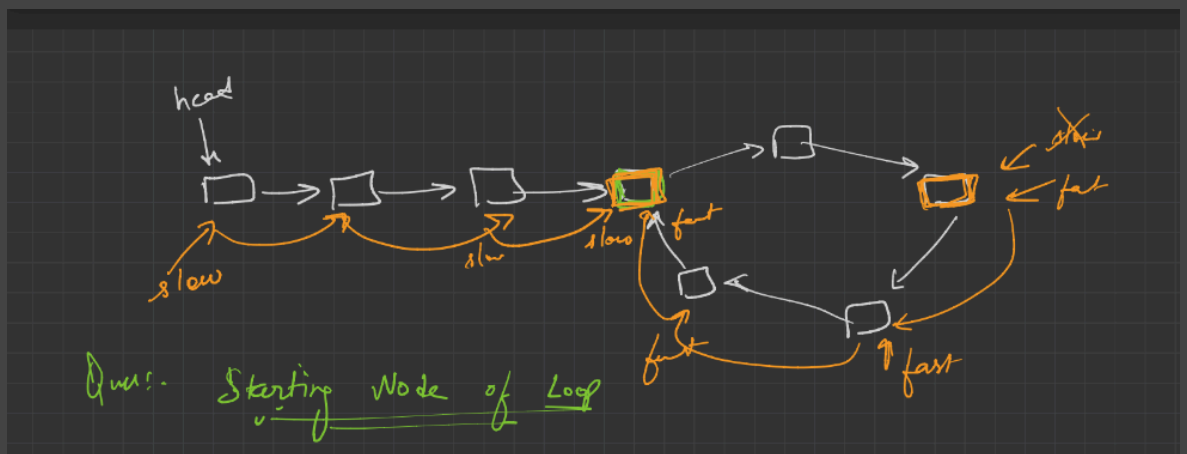
Floyd's Cycle Detection:







Find The Starting Node Of The Loop:



Ques: Starting Node of Loop

Approach:-

I → FCD Ifo → Point of Intersection

II → slow = head

↳ slow, fast → same pace
1, 1/1

↙

while(slow == fast)

↳ starting point of Loop

Distance by fast pointer = 2 * Distance by slow pointer

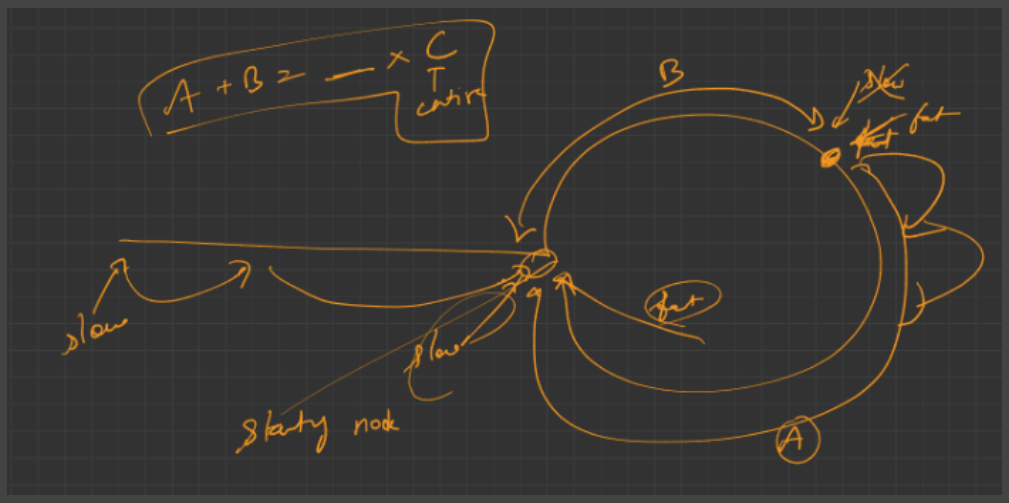
$$(A + x * C + B) = 2 * (A + y * C + B)$$

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

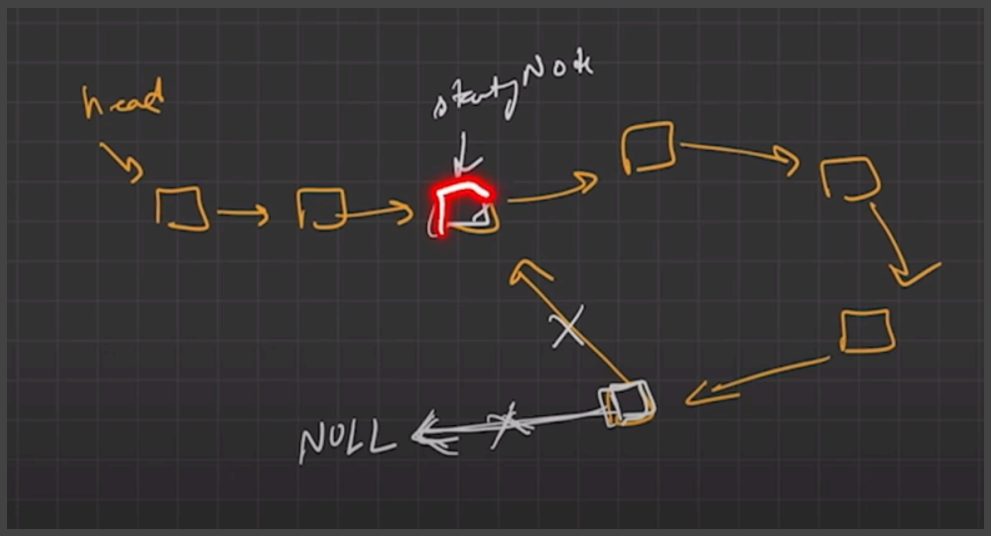
$$C(x - 2y) = A + B$$

$$\frac{C(x-2y)}{K} = A + B$$

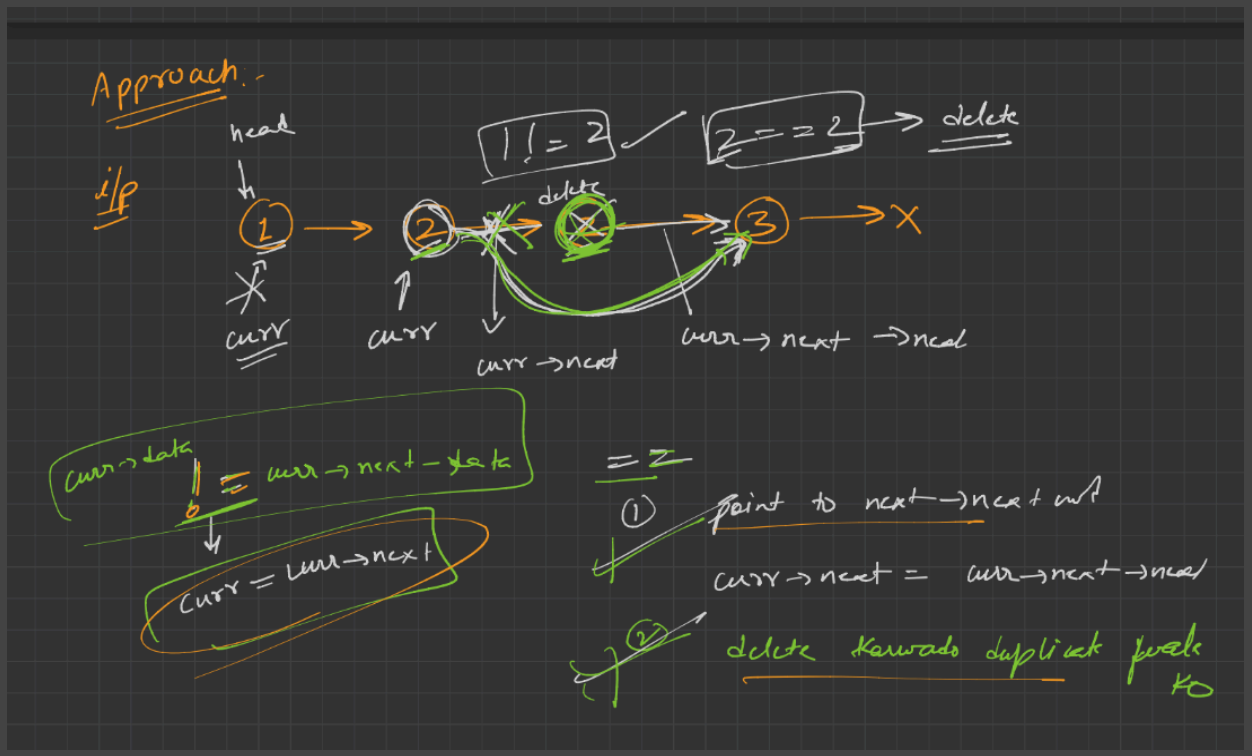
A + B = K times C



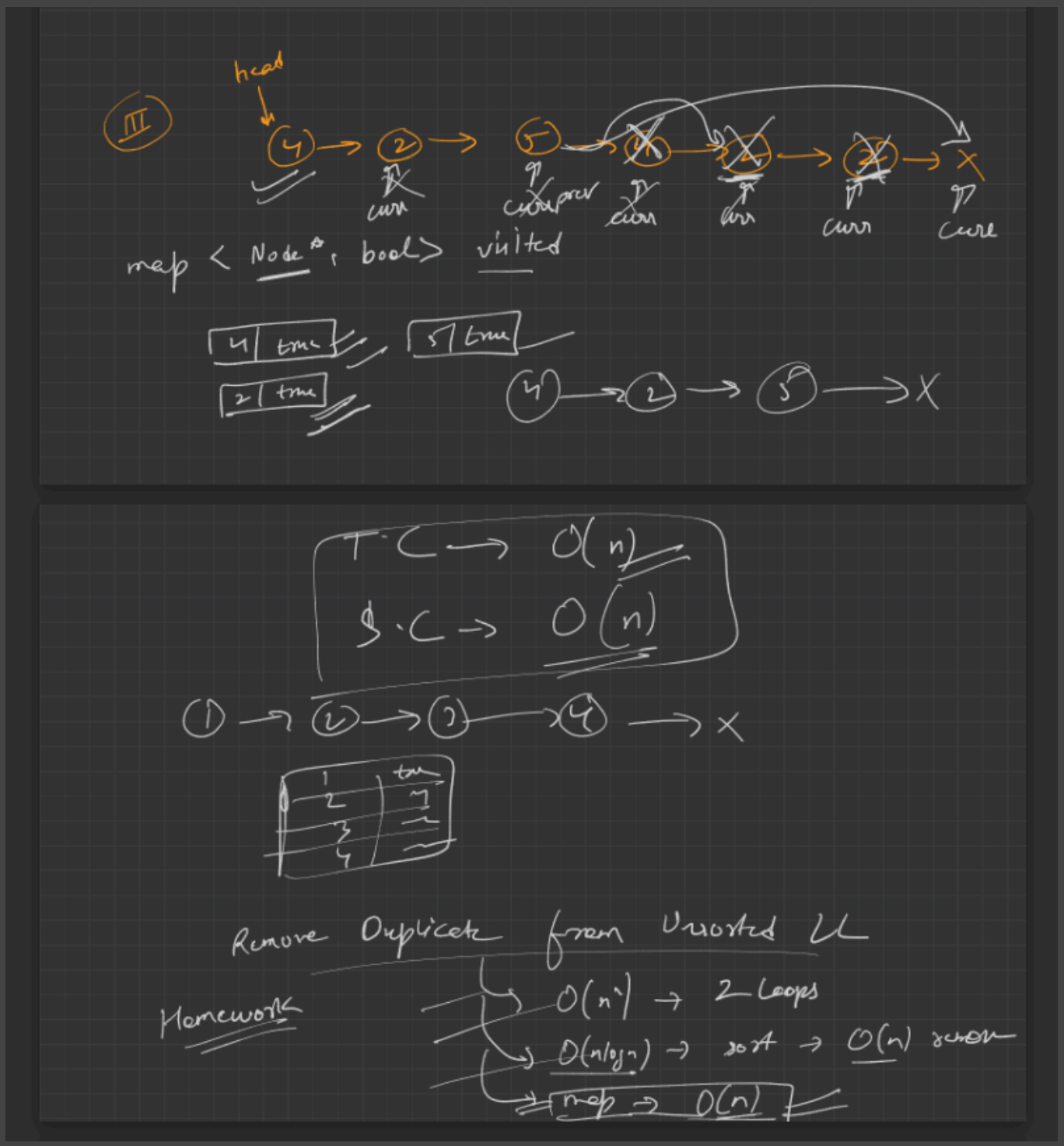
Remove The Loop:



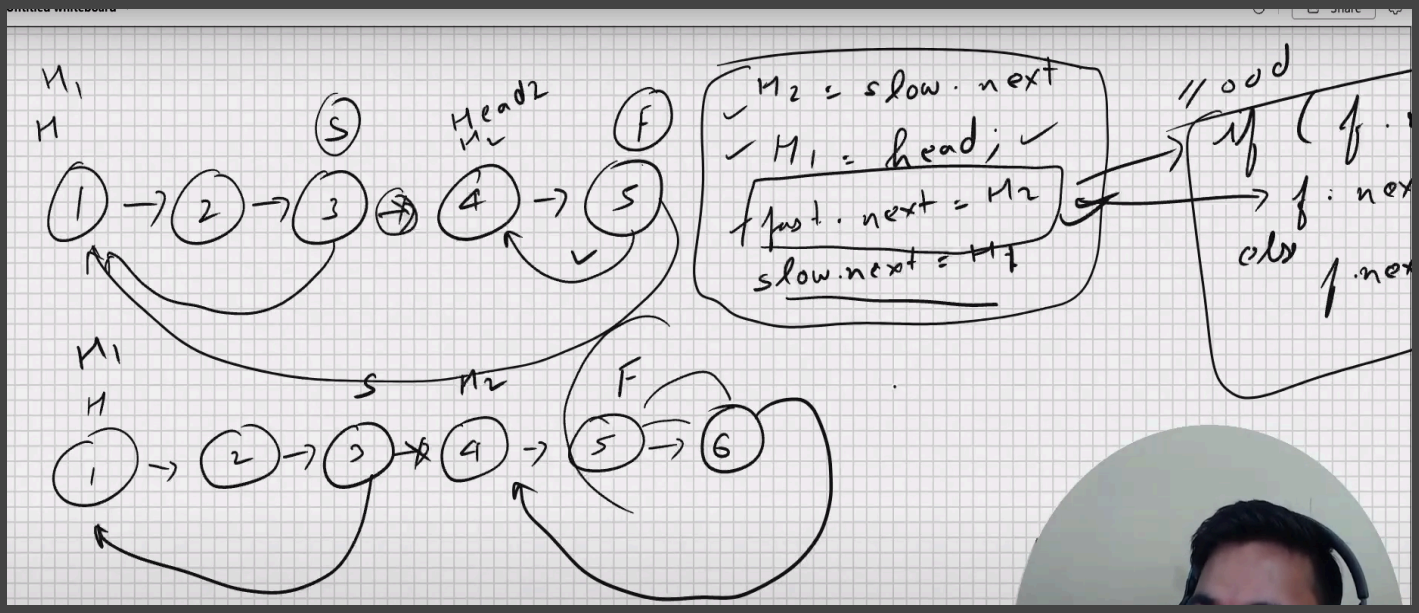
Remove Duplicates From A Sorted Linked List:



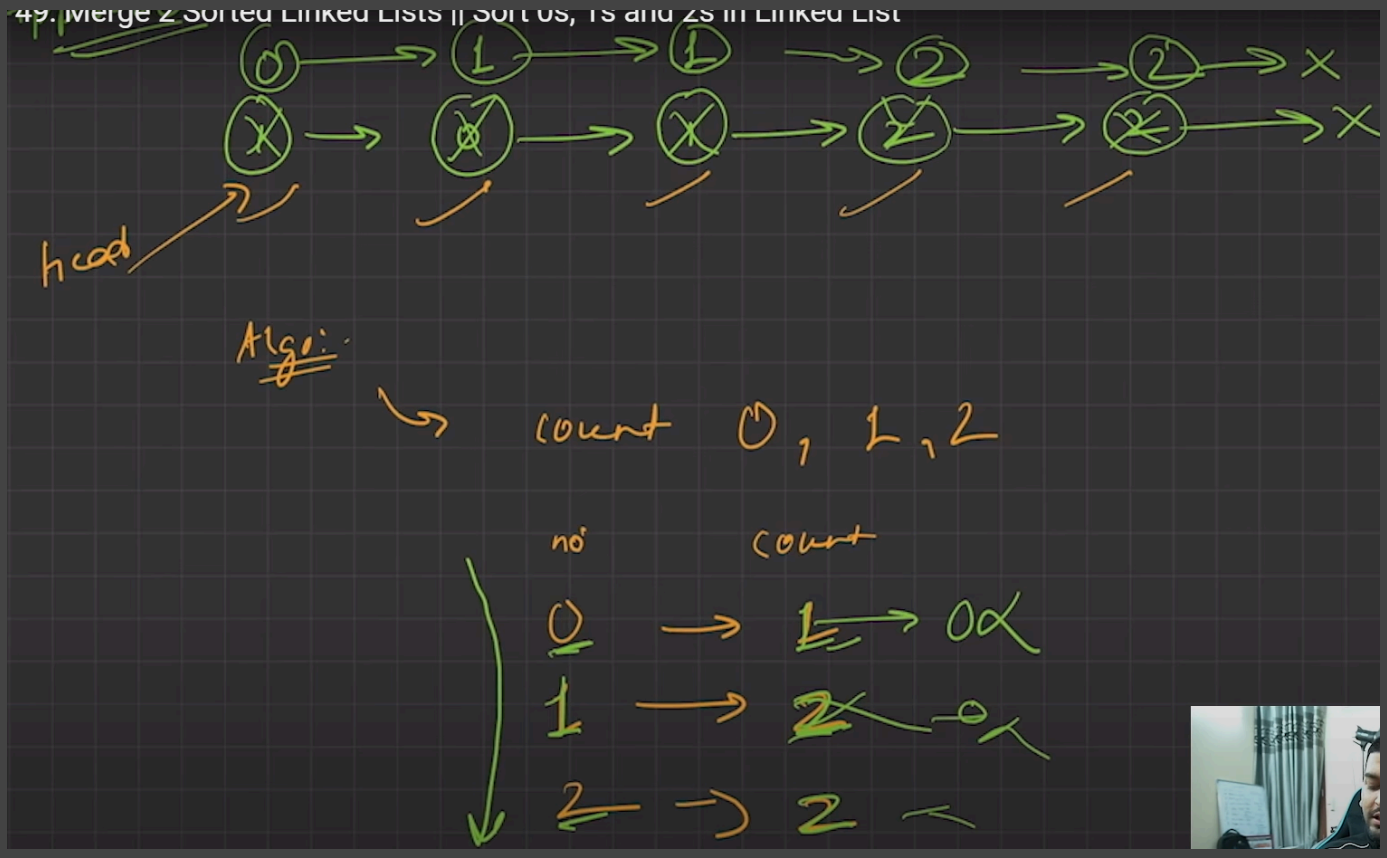
Remove Duplicates From A Unsorted Linked List:



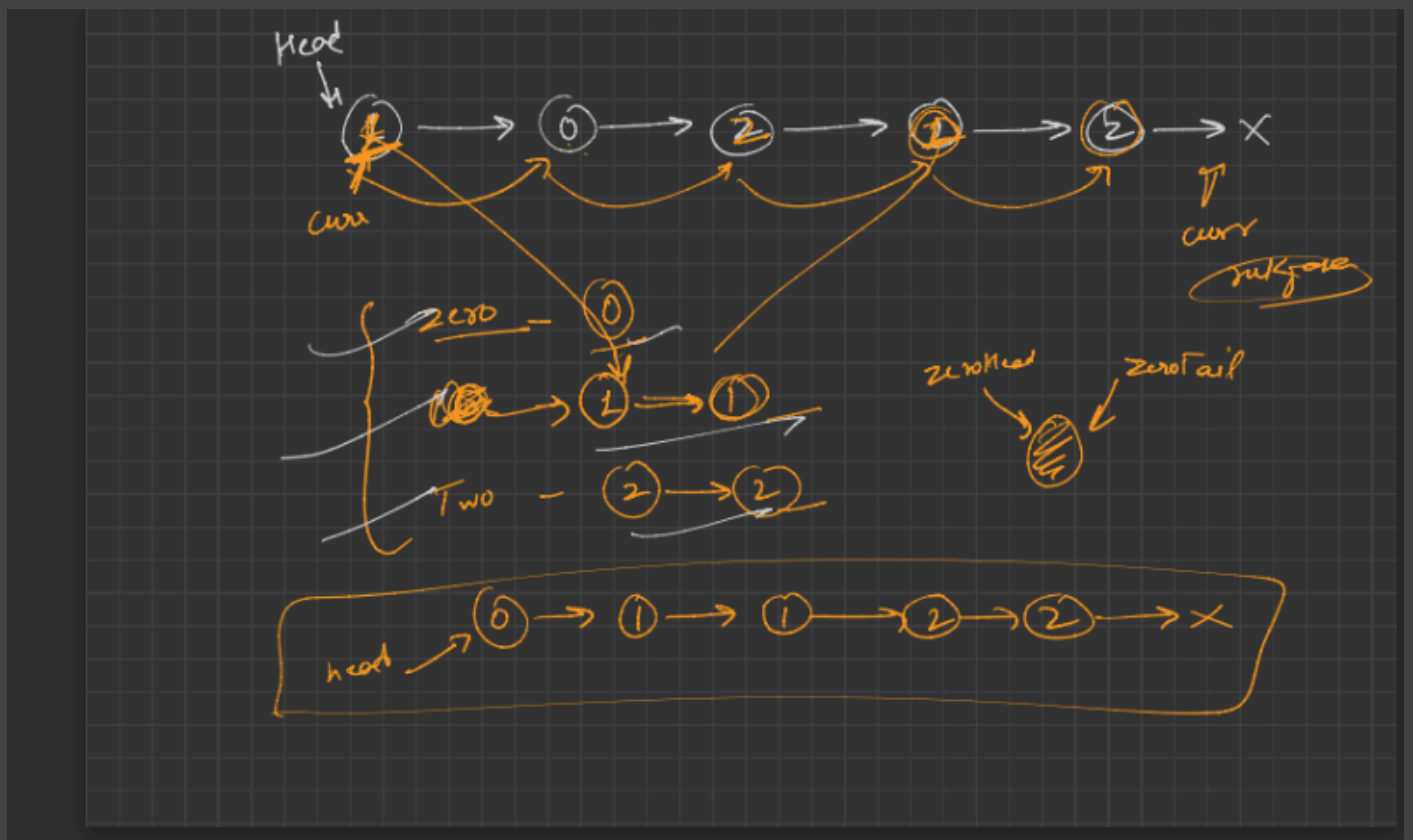
Split A Circular LL Into Two Halves:

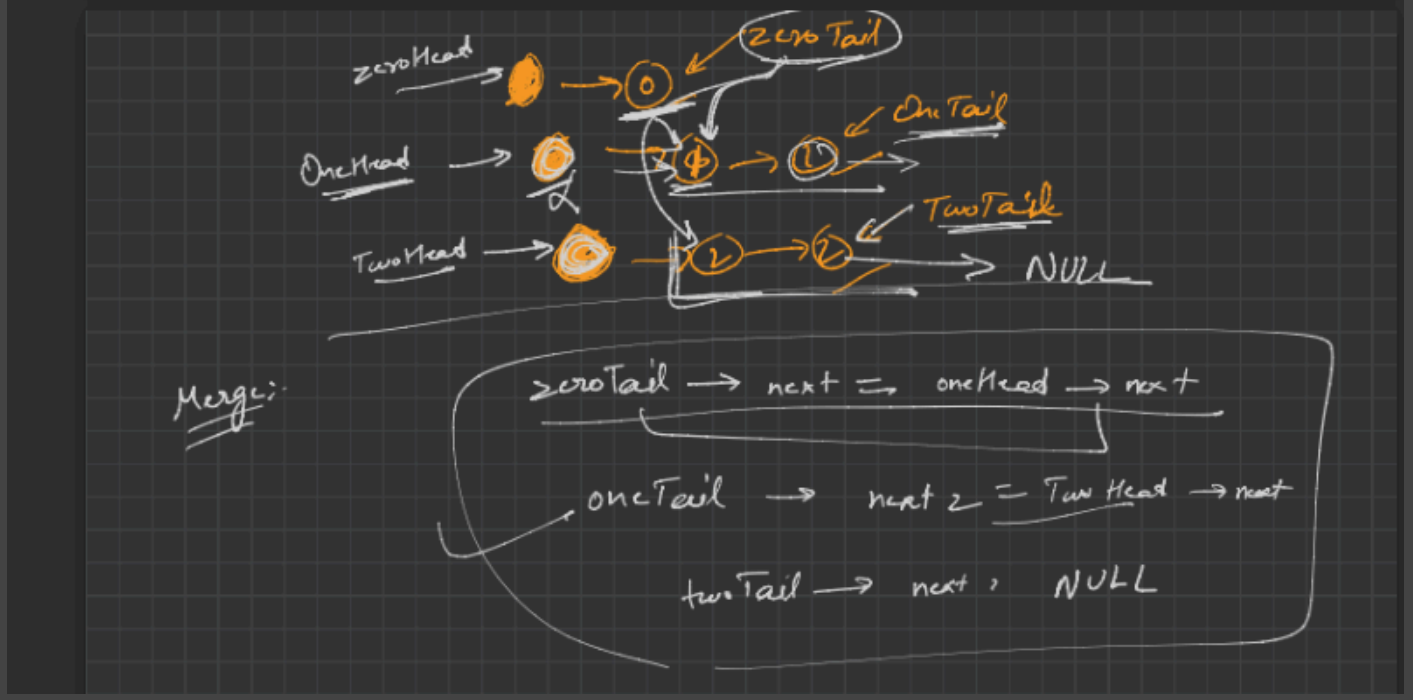


Sort 0s,1s & 2s In A LL Approach-I:

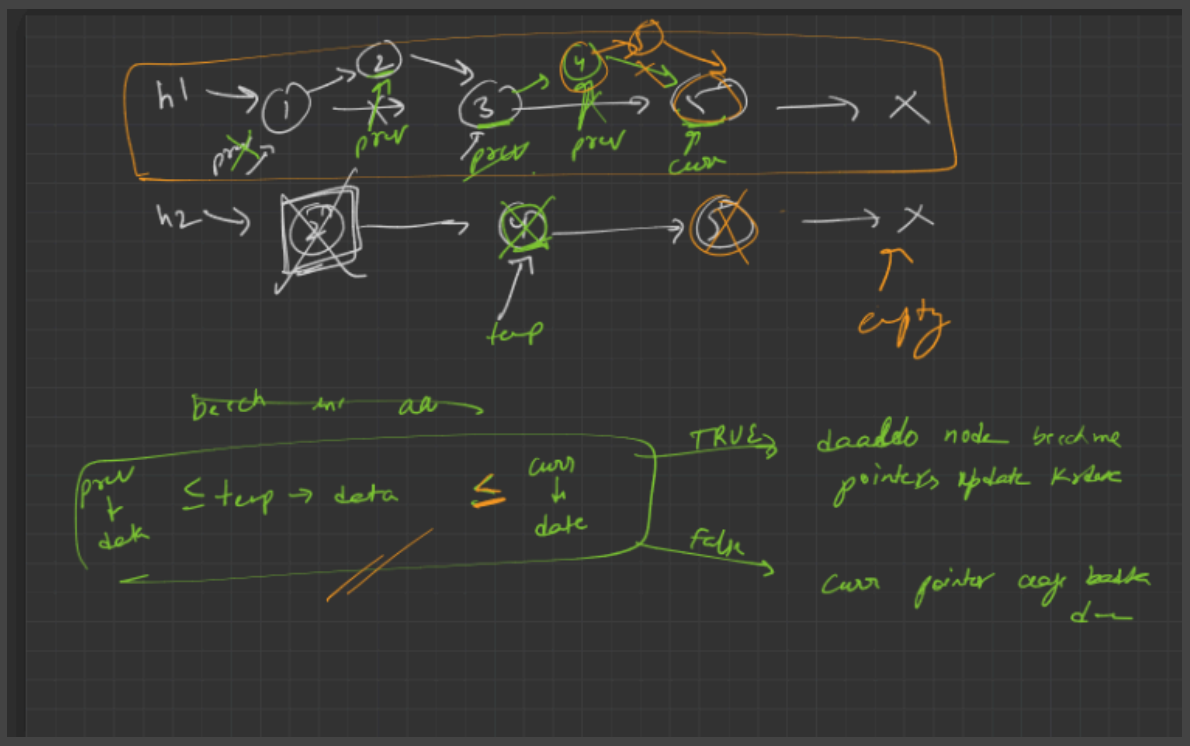


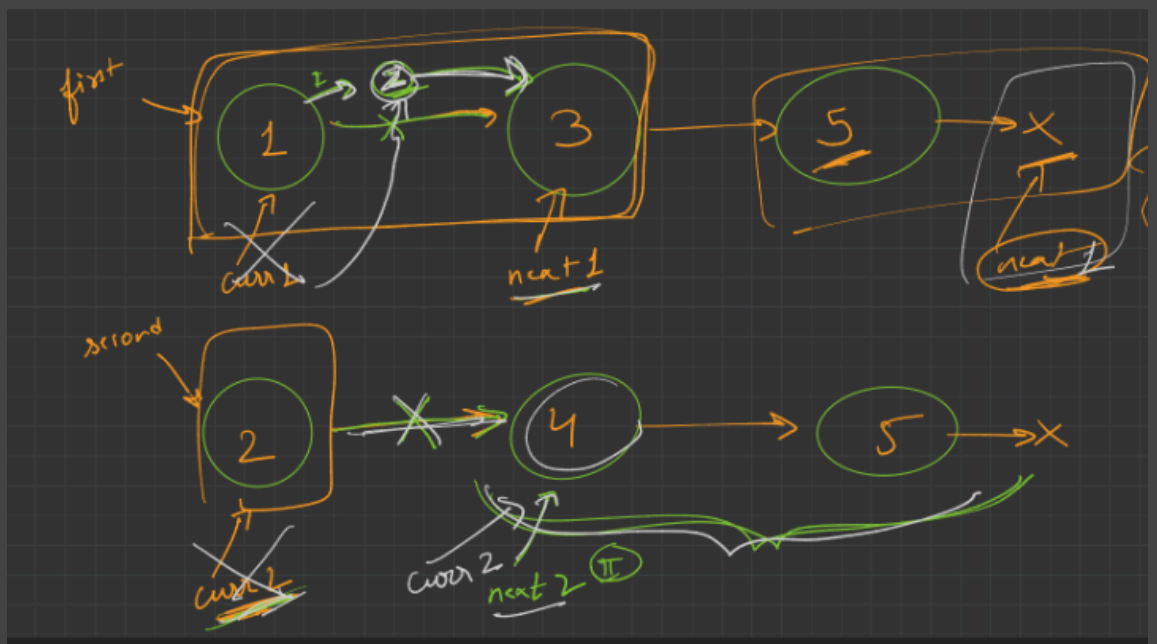
Sort 0s,1s & 2s In A LL Approach-II:





Merge Two Sorted LL:





I → curr1 → next = curr2

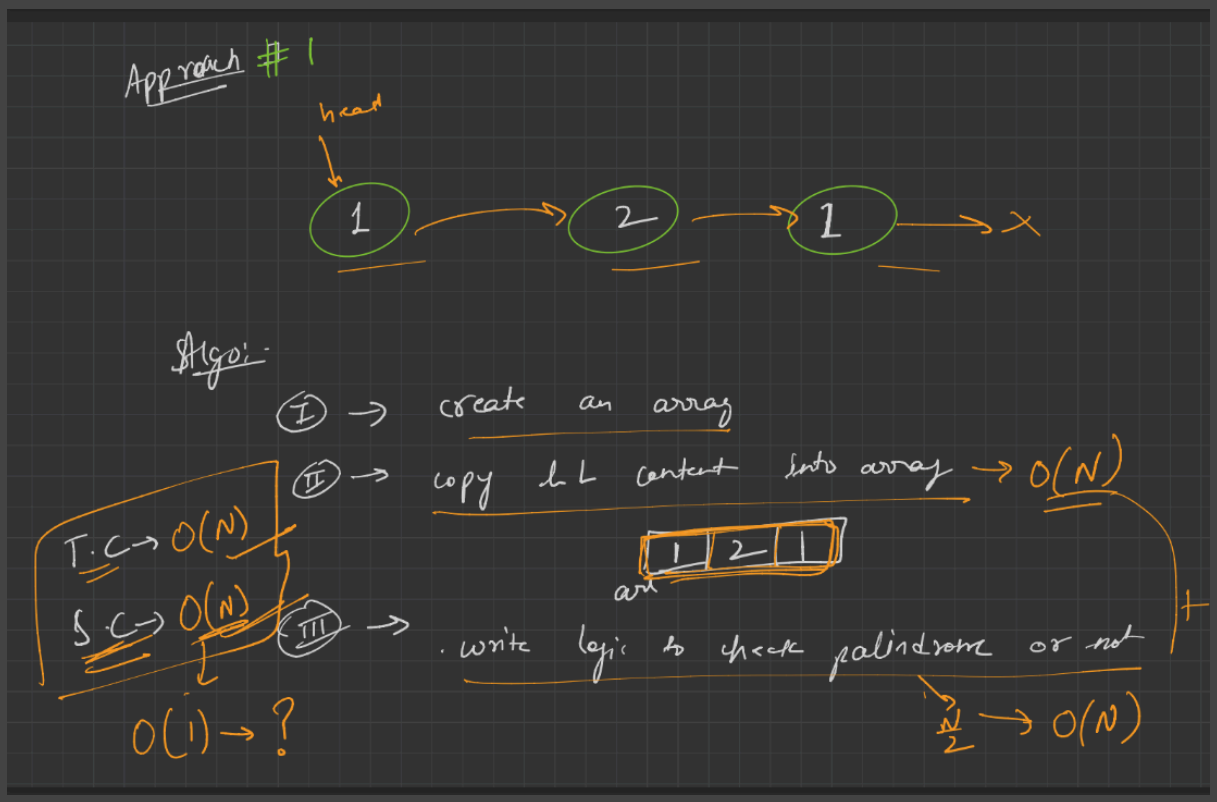
II → next2 = curr2 → next

III → curr2 → next = next 1

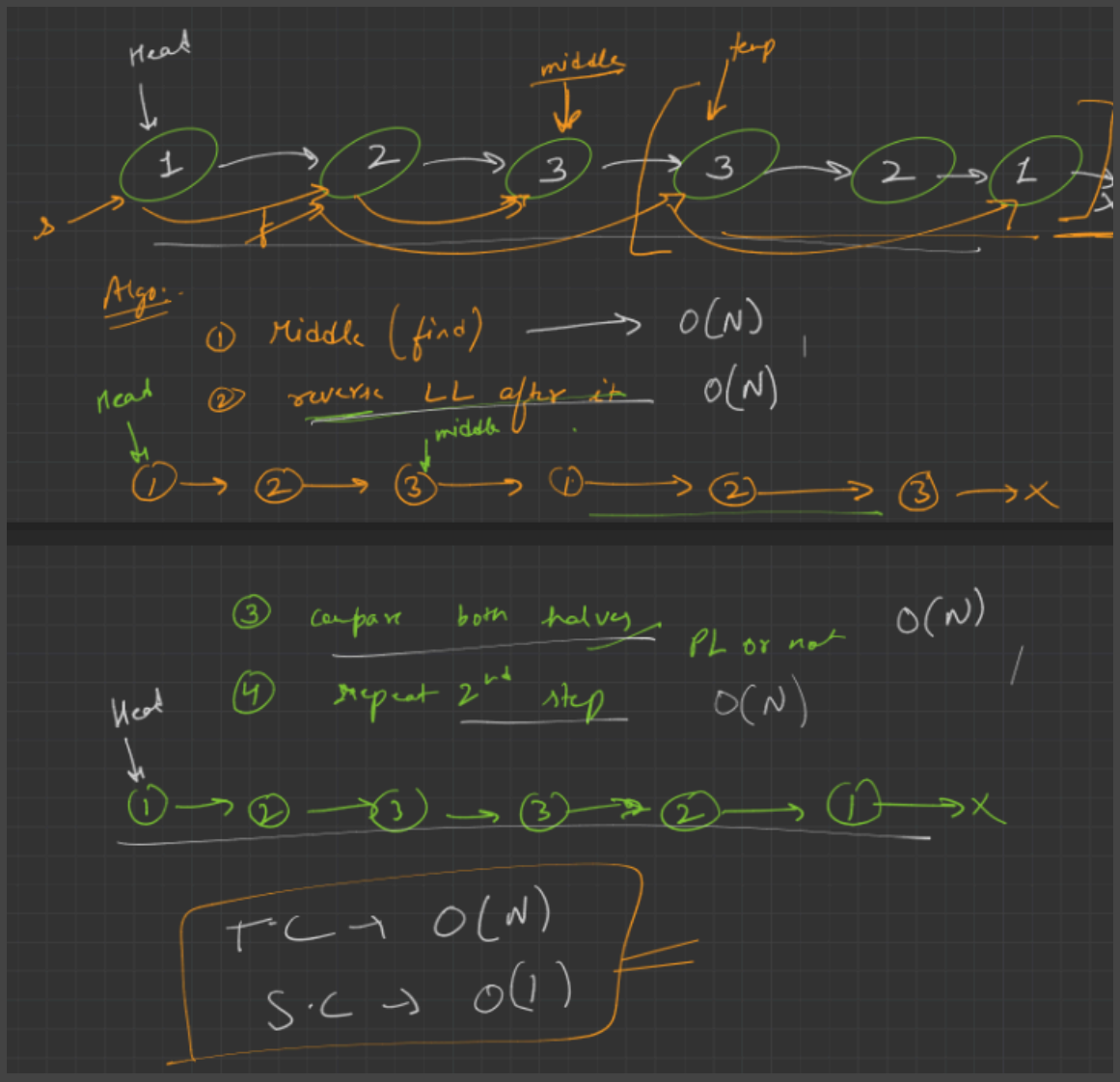
IV ⟹ curr 1 = curr 2

V = Curr 2 = ne

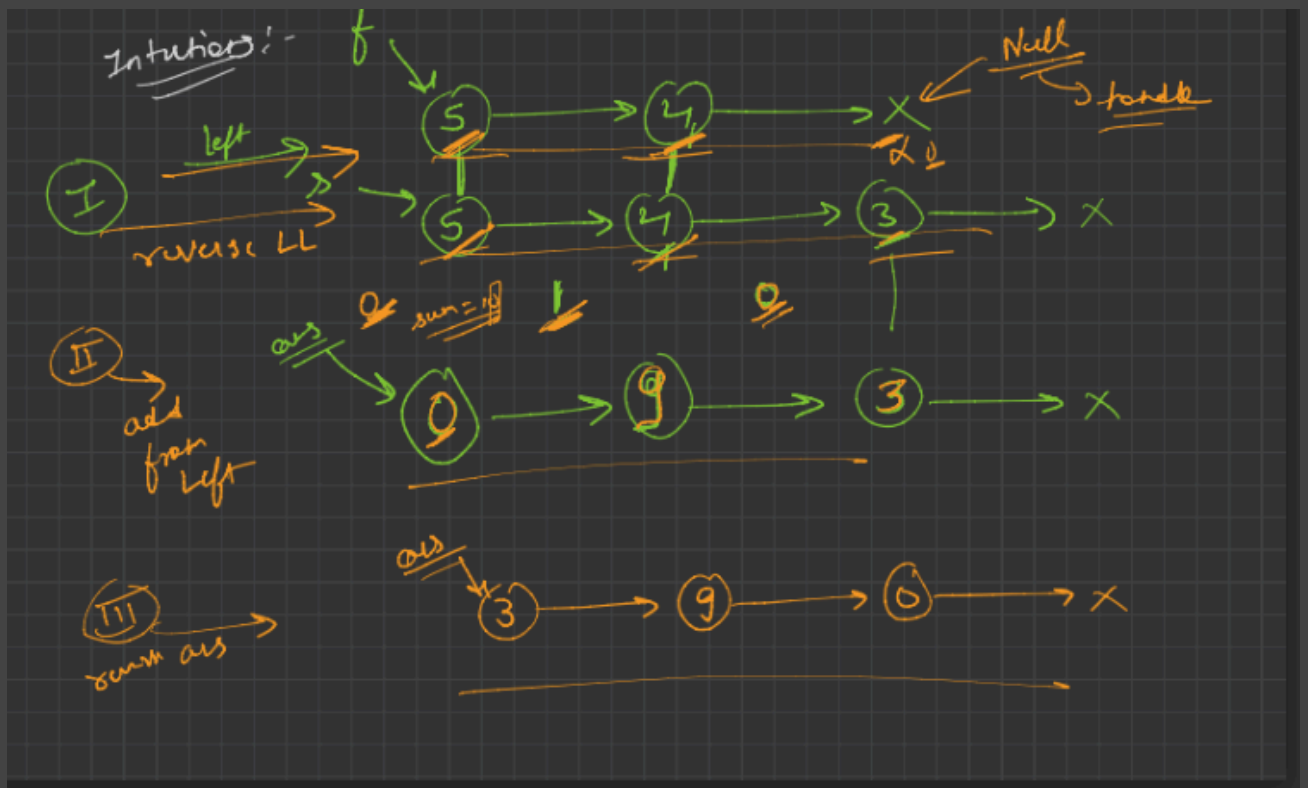
Check Palindrome In LL Approach-I:



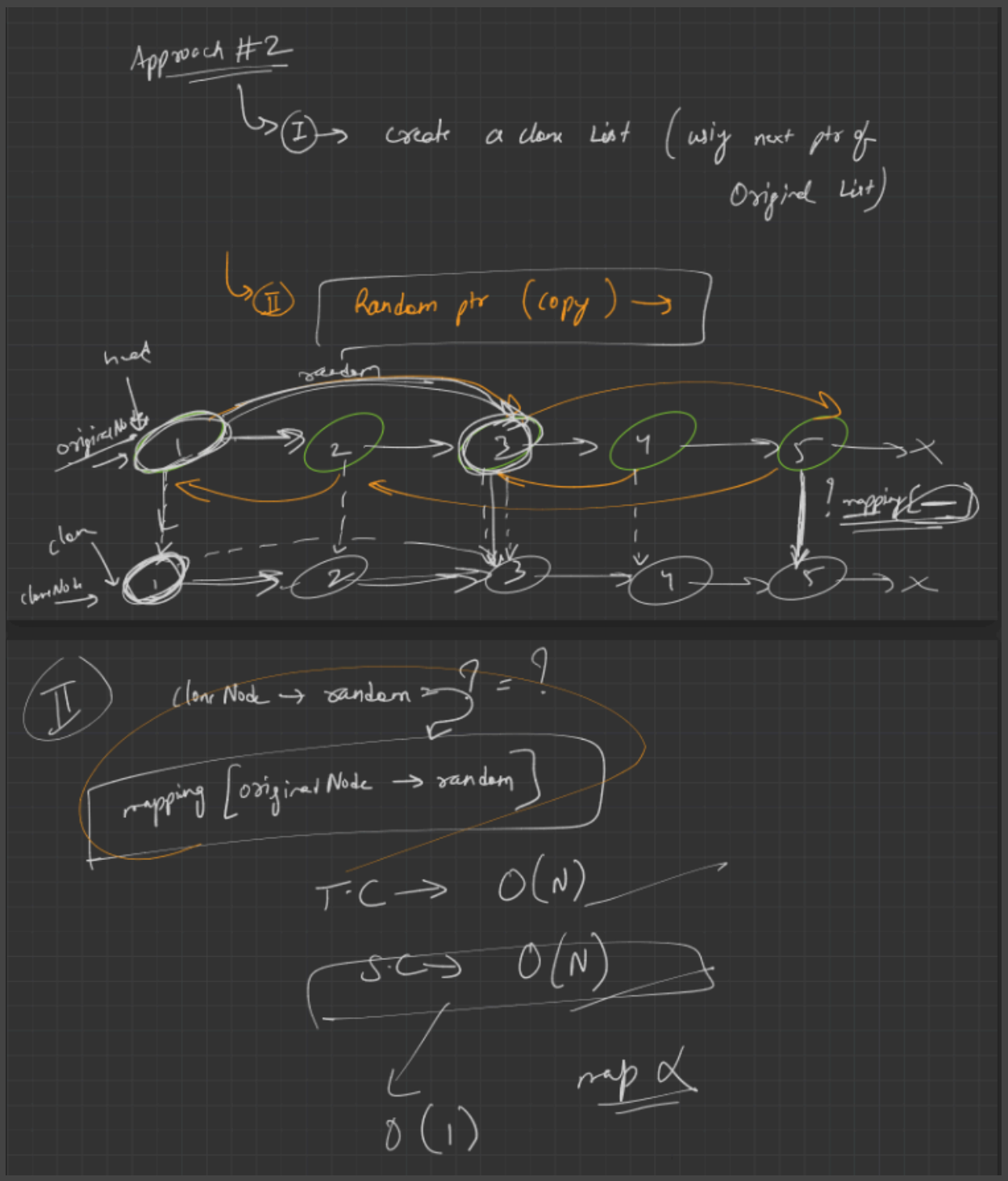
Check Palindrome In LL Approach-II:



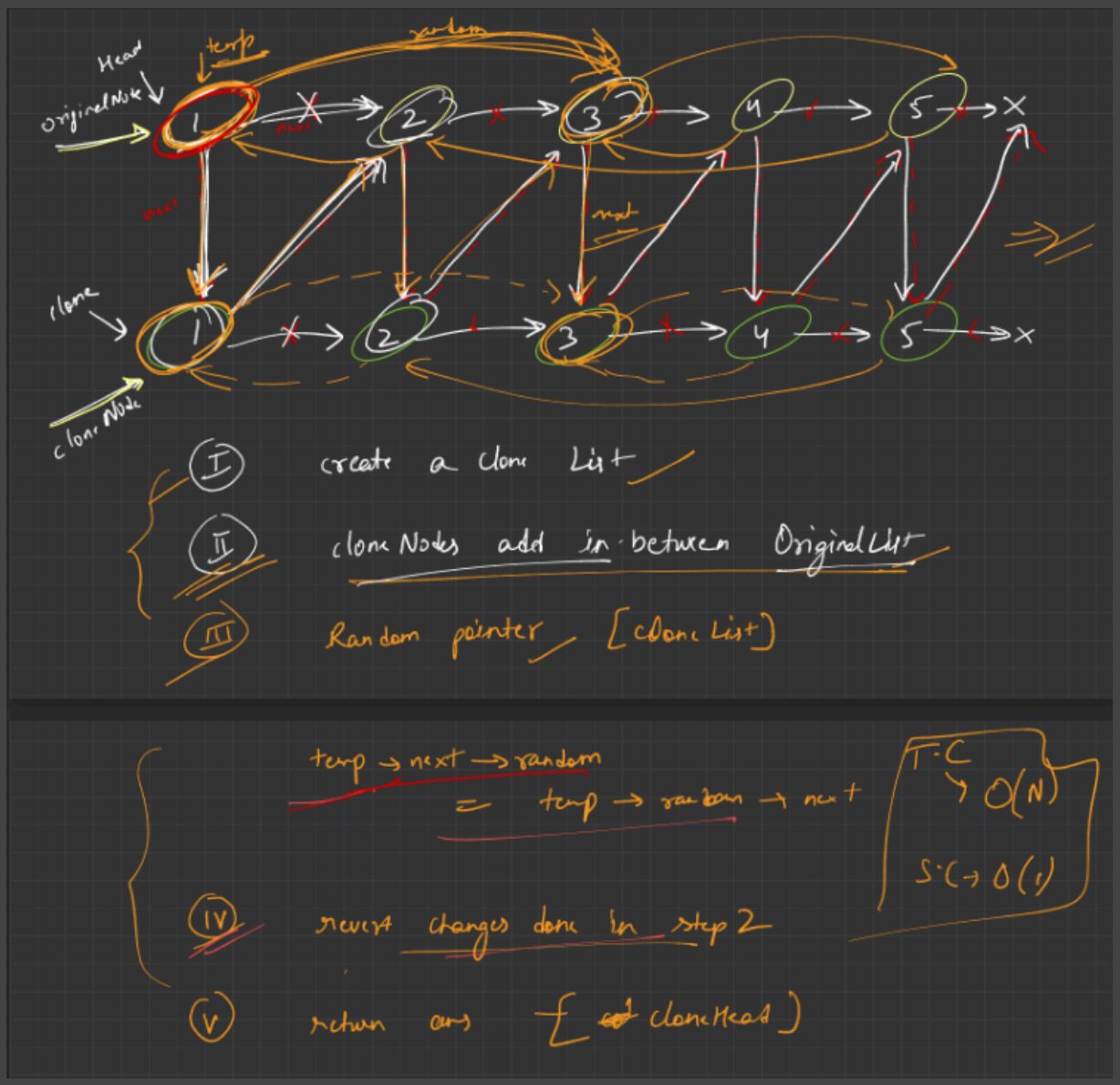
Add Two Numbers Represented By Linked List:

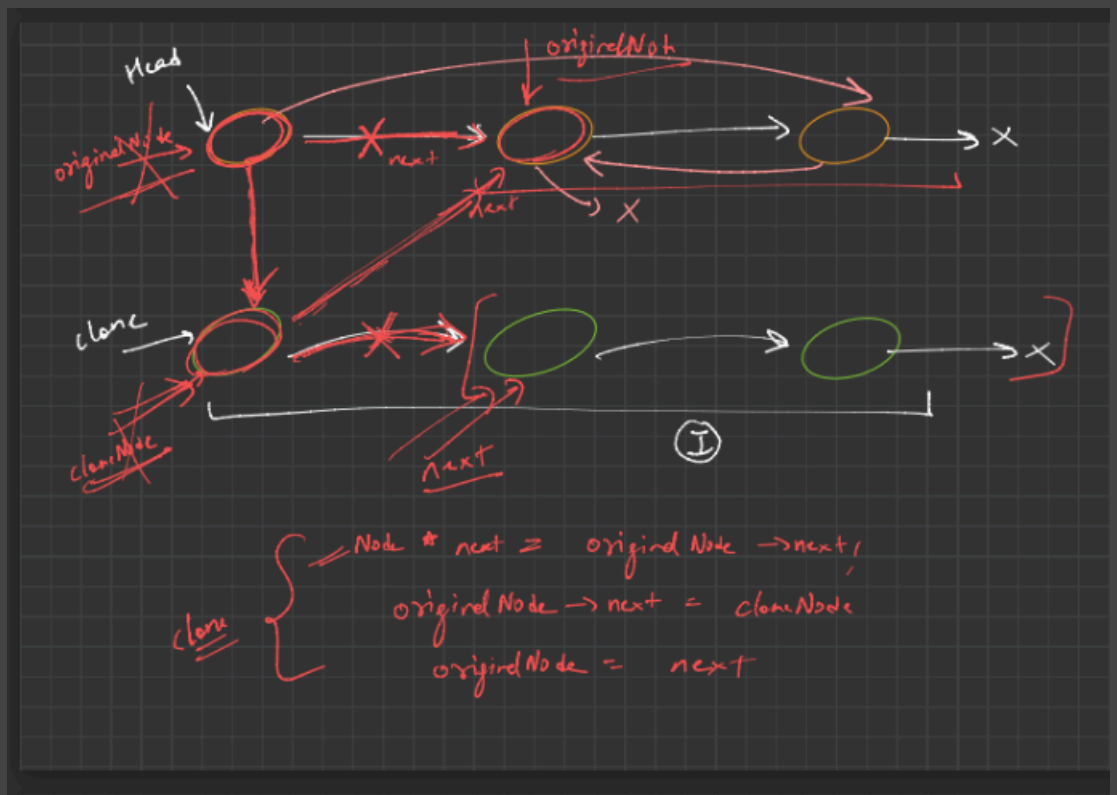


Clone A Linked List Approach-I:



Clone A Linked List Approach-II:

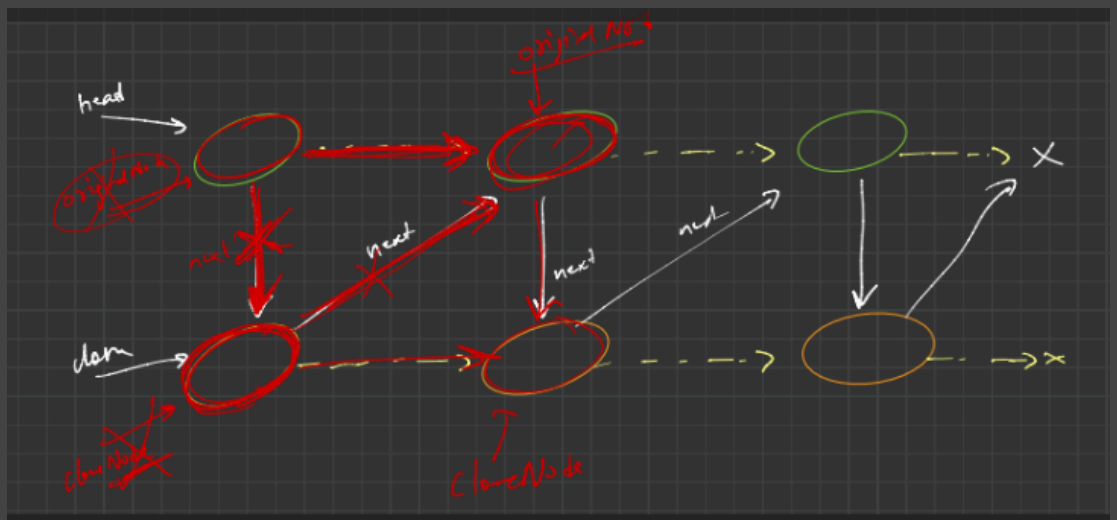





```
clone {
    Node * next = originalNode->next;
    originalNode->next = cloneNode
    originalNode = next
```

(I)

```
    next = cloneNode->next;
    cloneNode->next = originalNode
    cloneNode = next
```

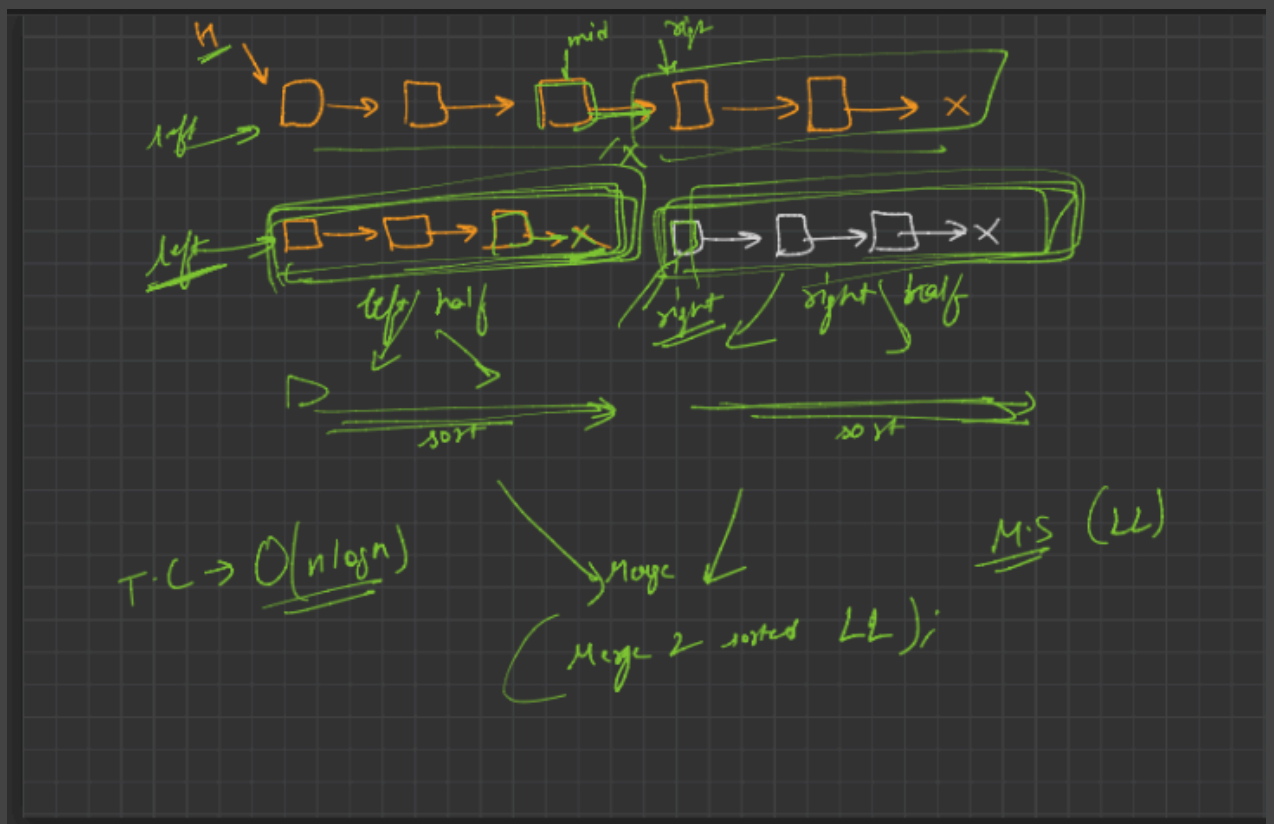


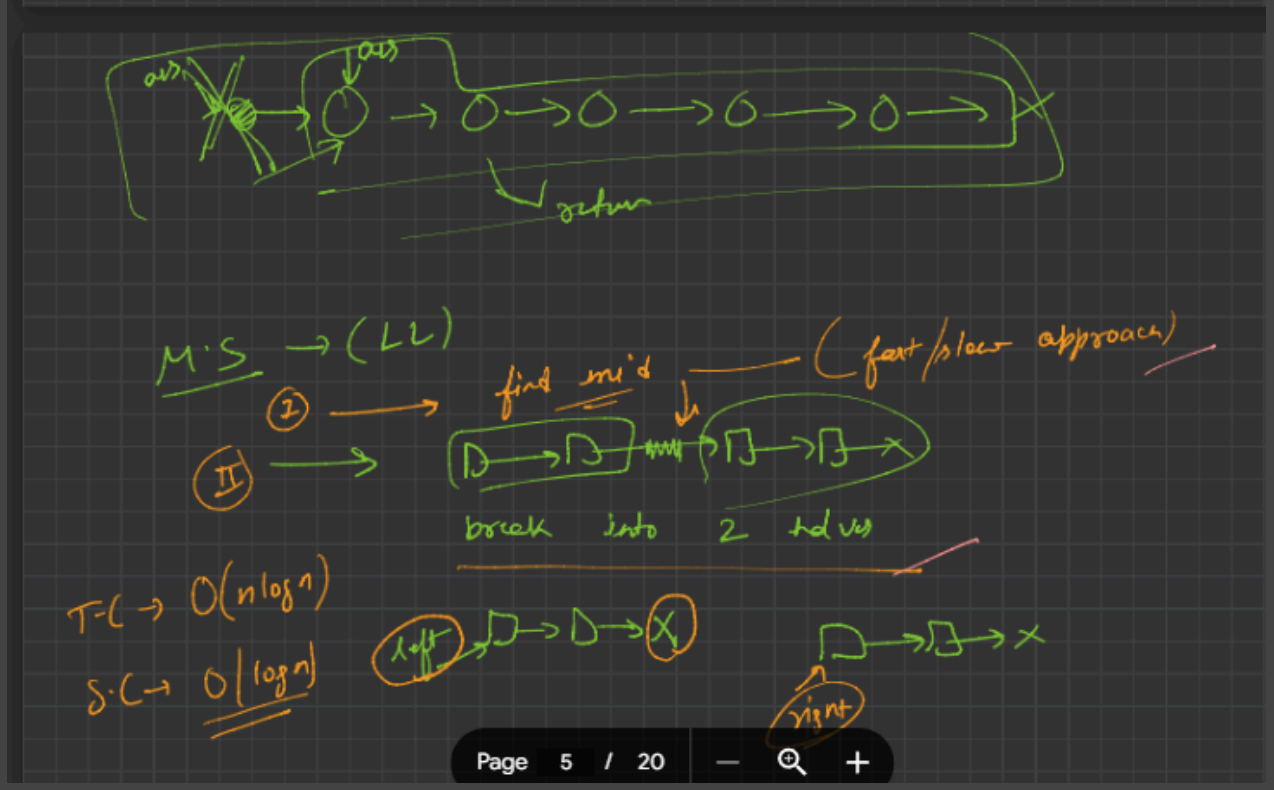

```
clone {
    originalNode->next = cloneNode->next
    originalNode = originalNode->next

    cloneNode->next = originalNode->next
    cloneNode = cloneNode->next
```

(II)

Merge Sort In Linked List:



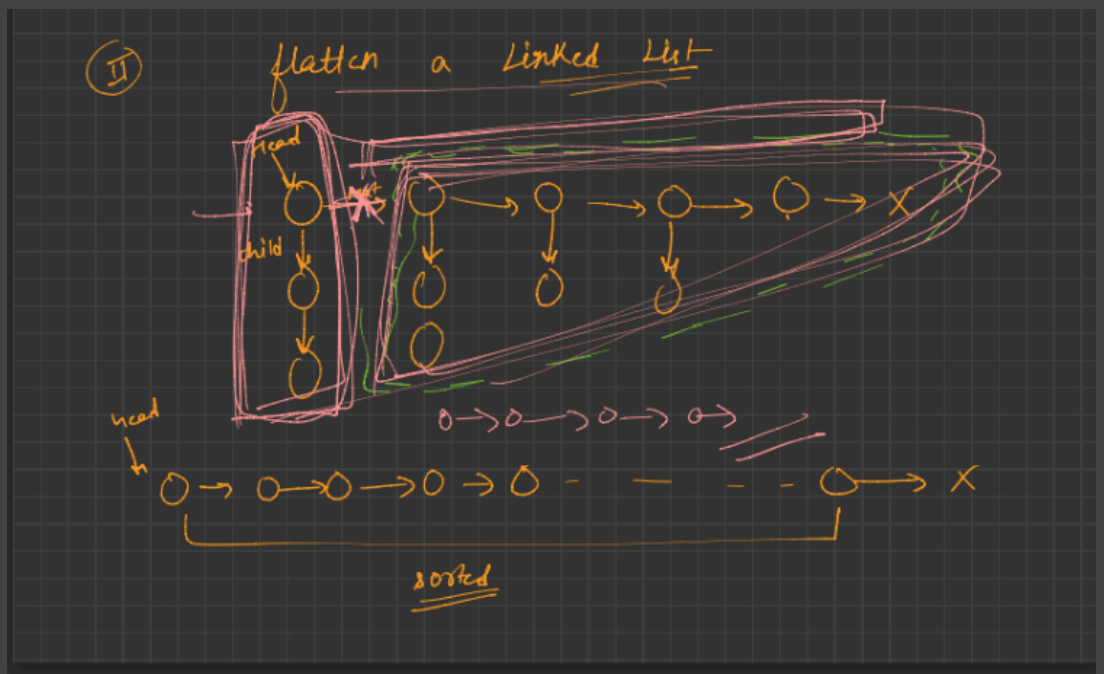


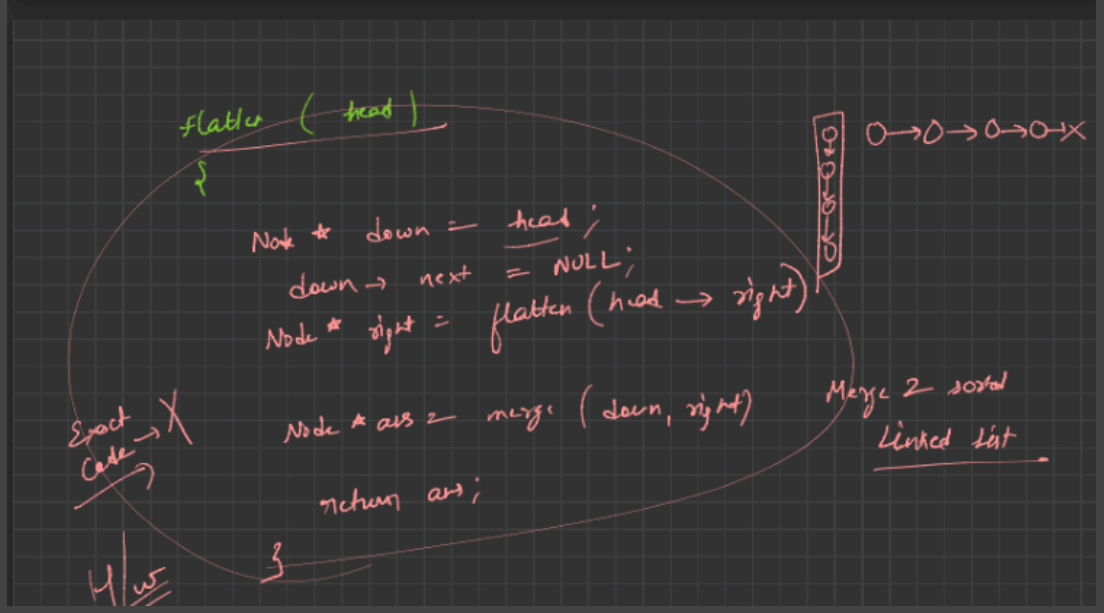
III Recursively sort left/right halves

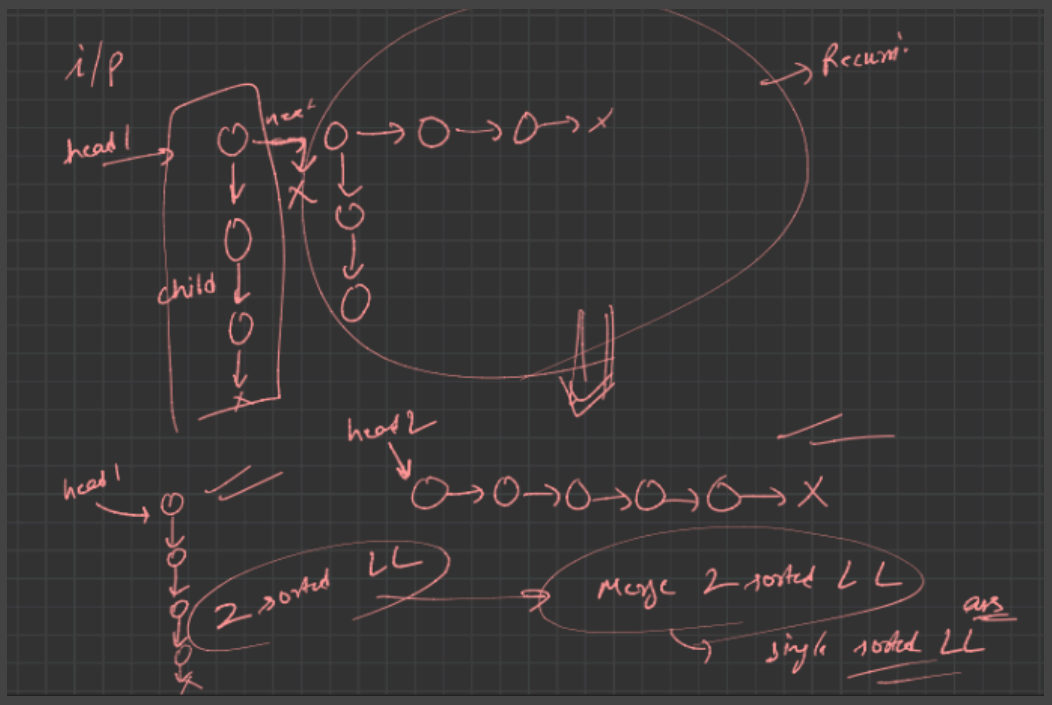
IV merge left/right sorted halves → (Merge 2 sorted LL)

V return merged list

Flatten Linked List:

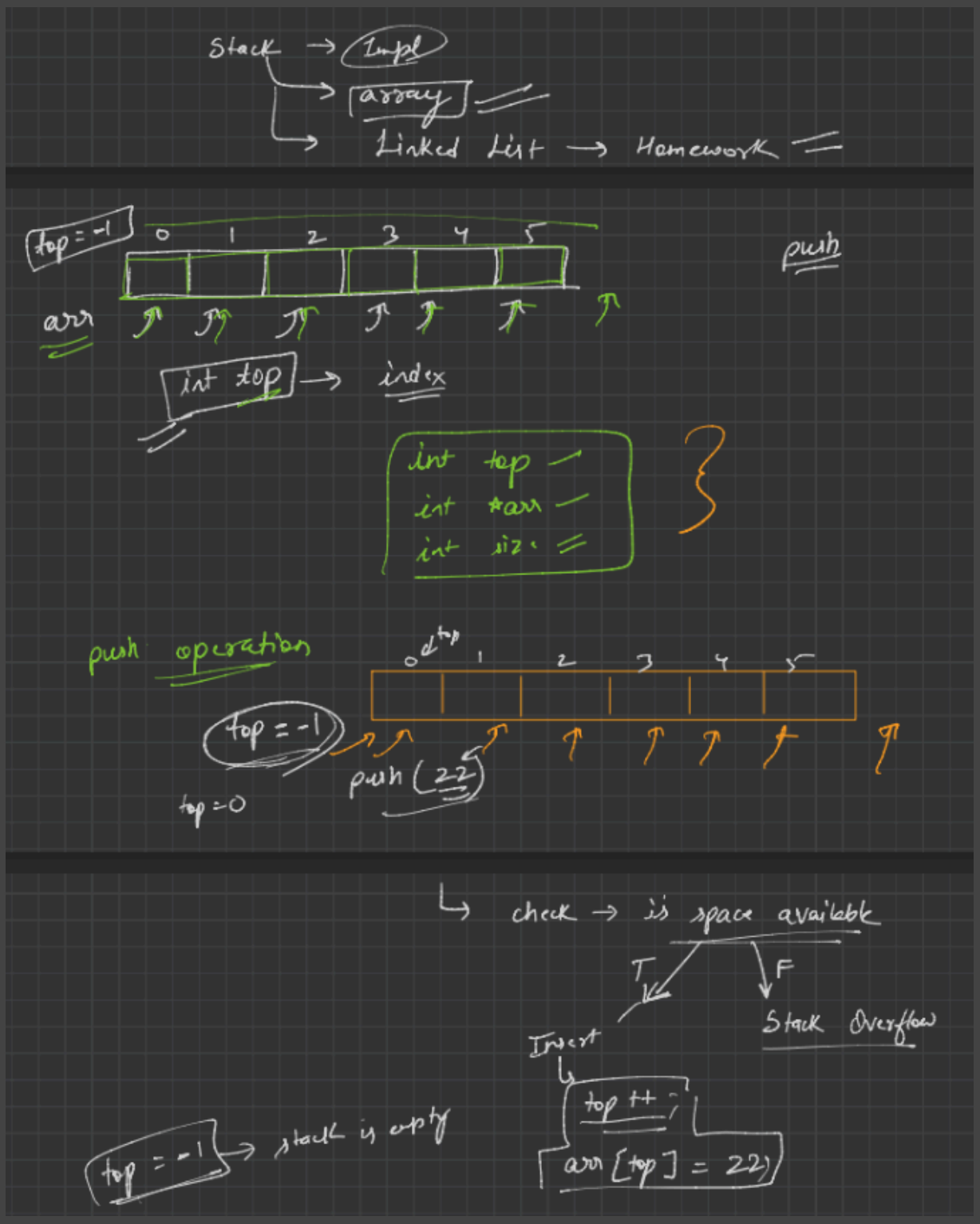




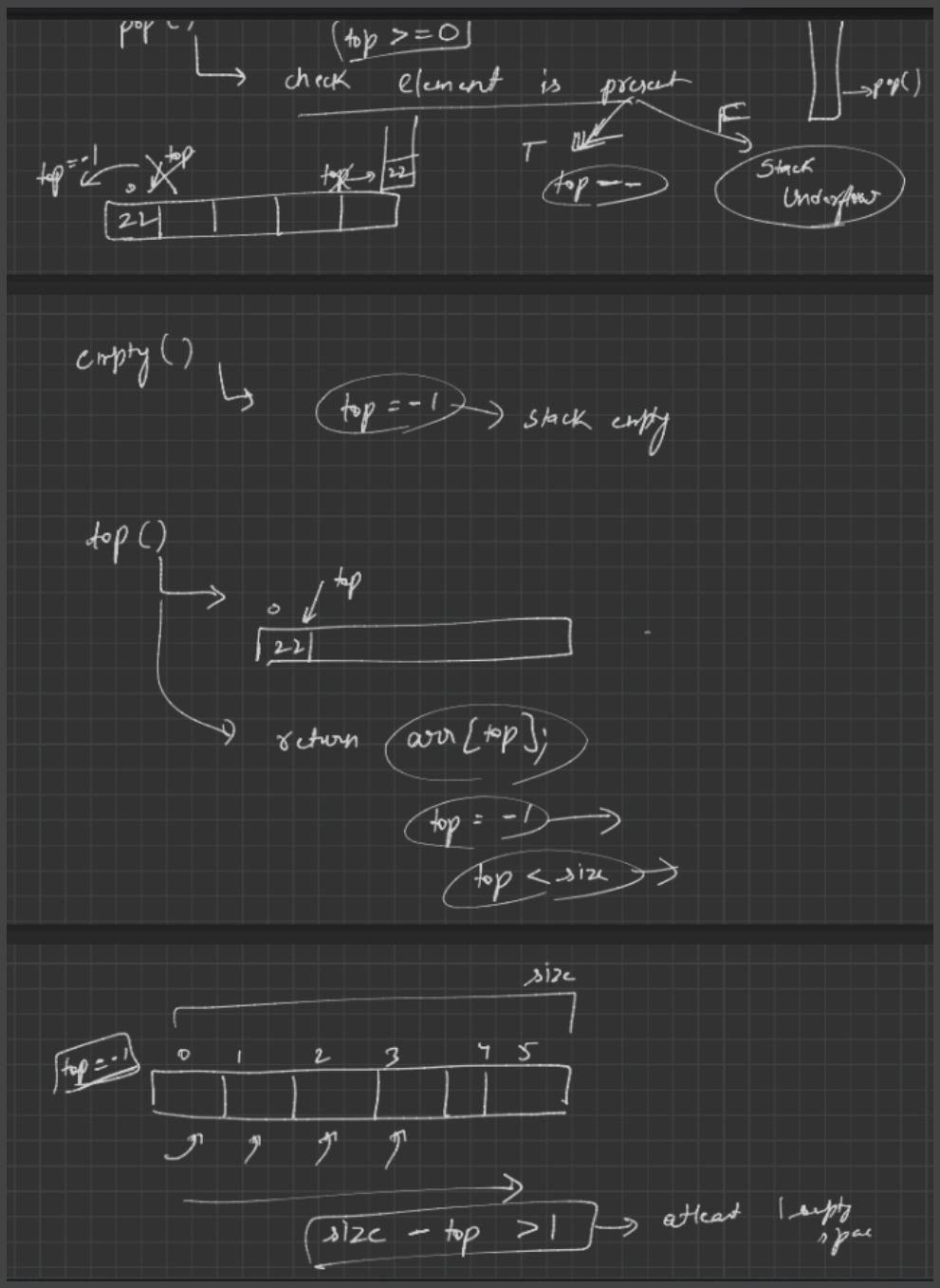


Stack

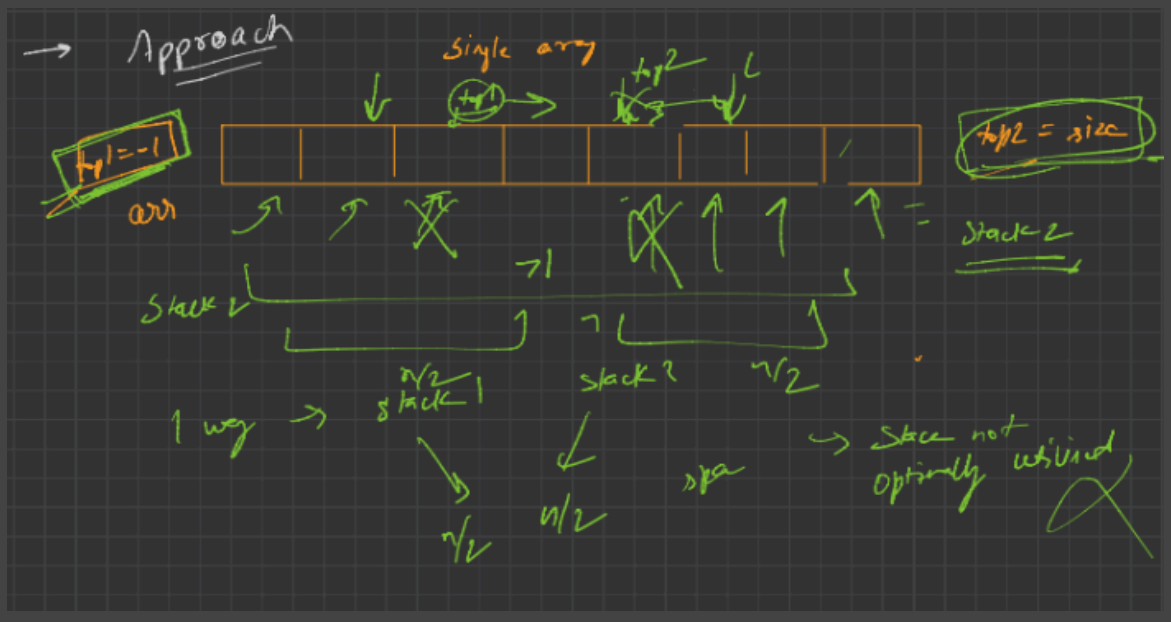
Create A Stack Using An Array:



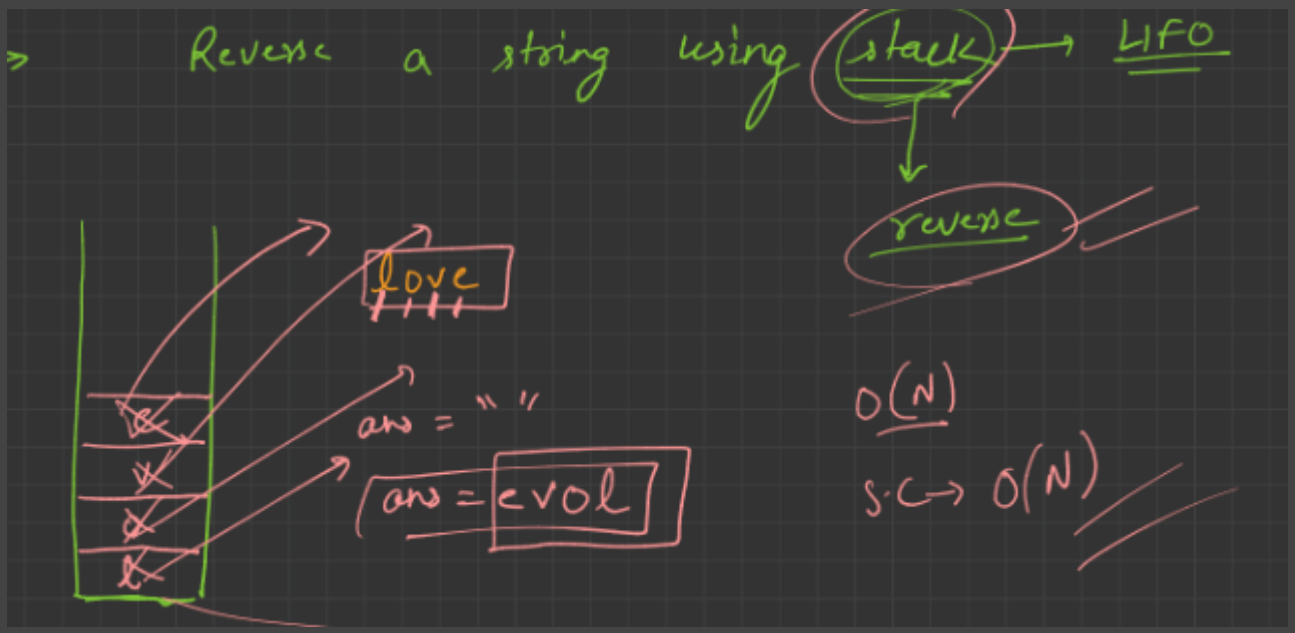
Create A Stack Using A Linked List:



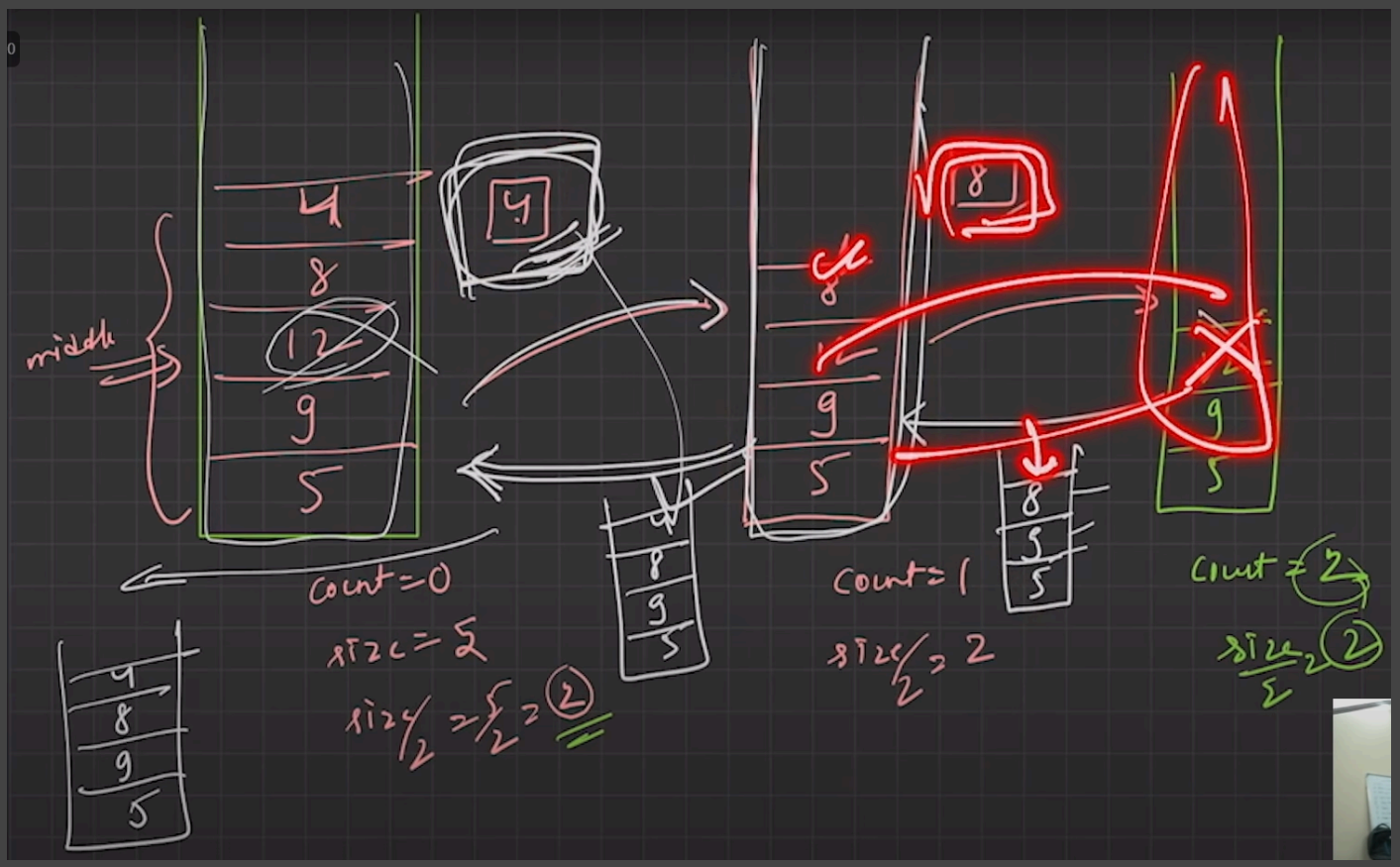
Create Two Stacks Using An Array:



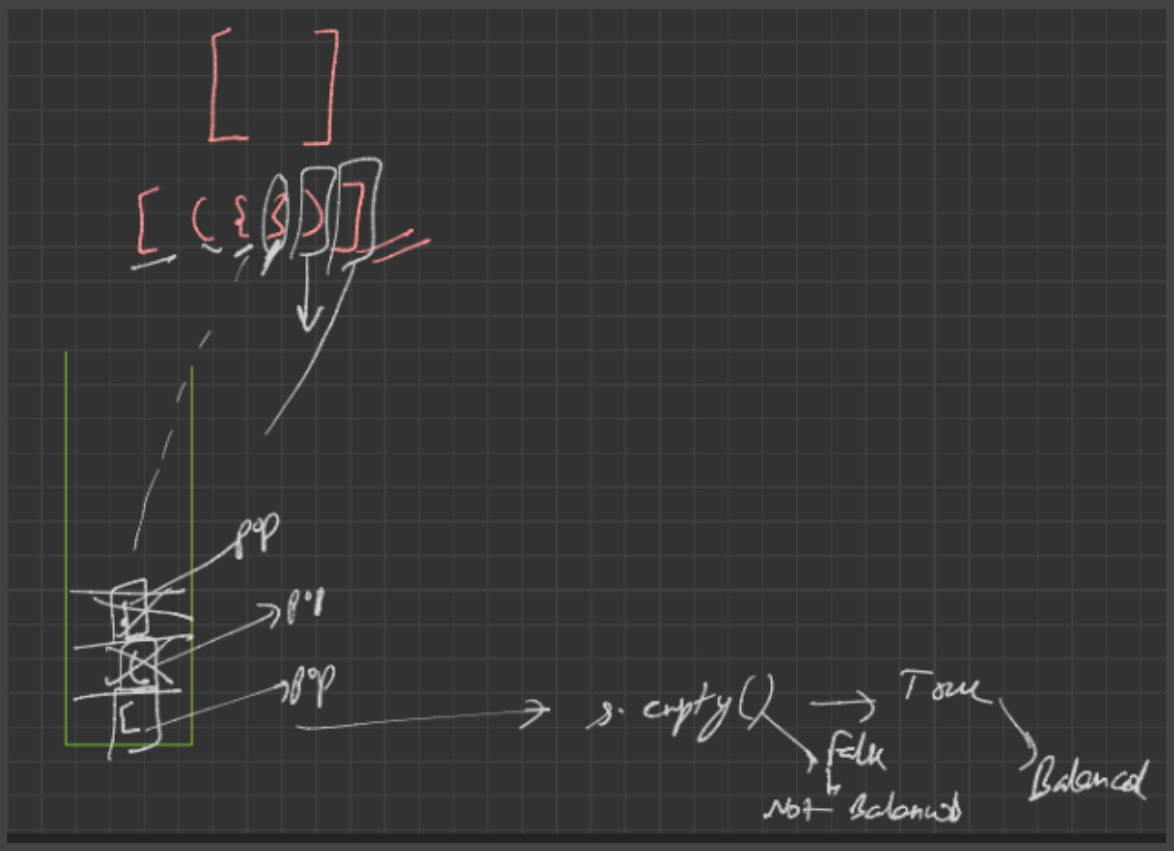
Reverse A String Using Stack:



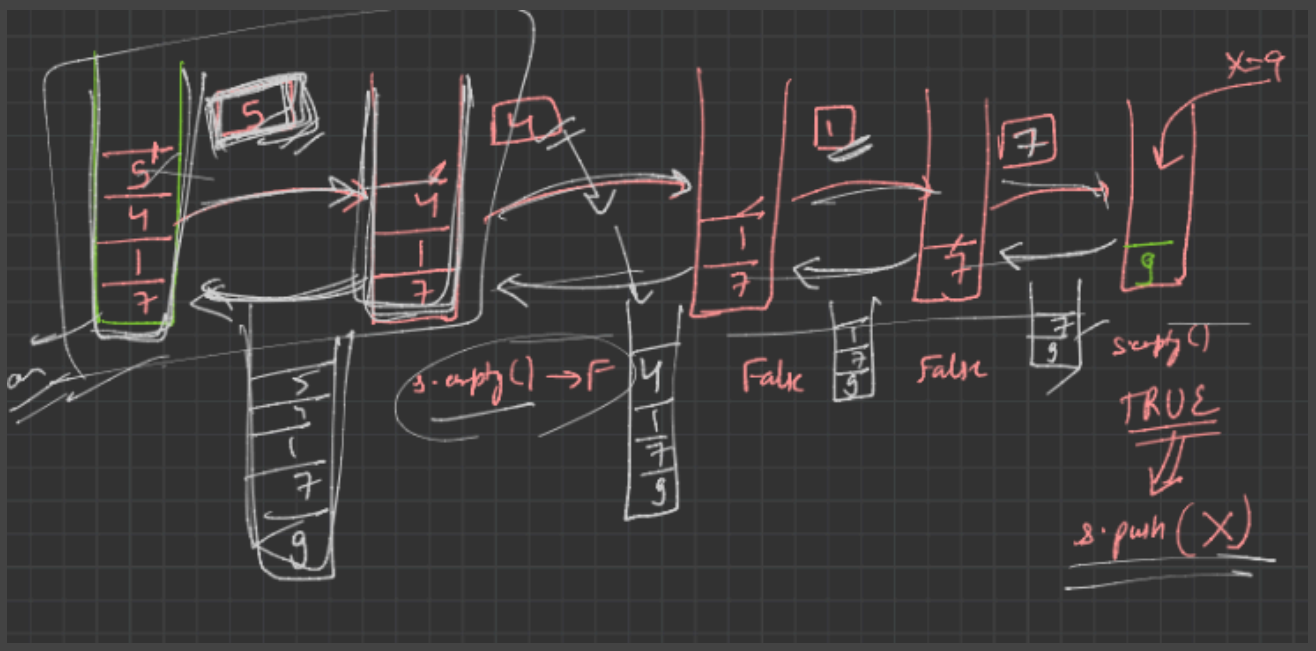
Delete MIddle Element In A Stack:



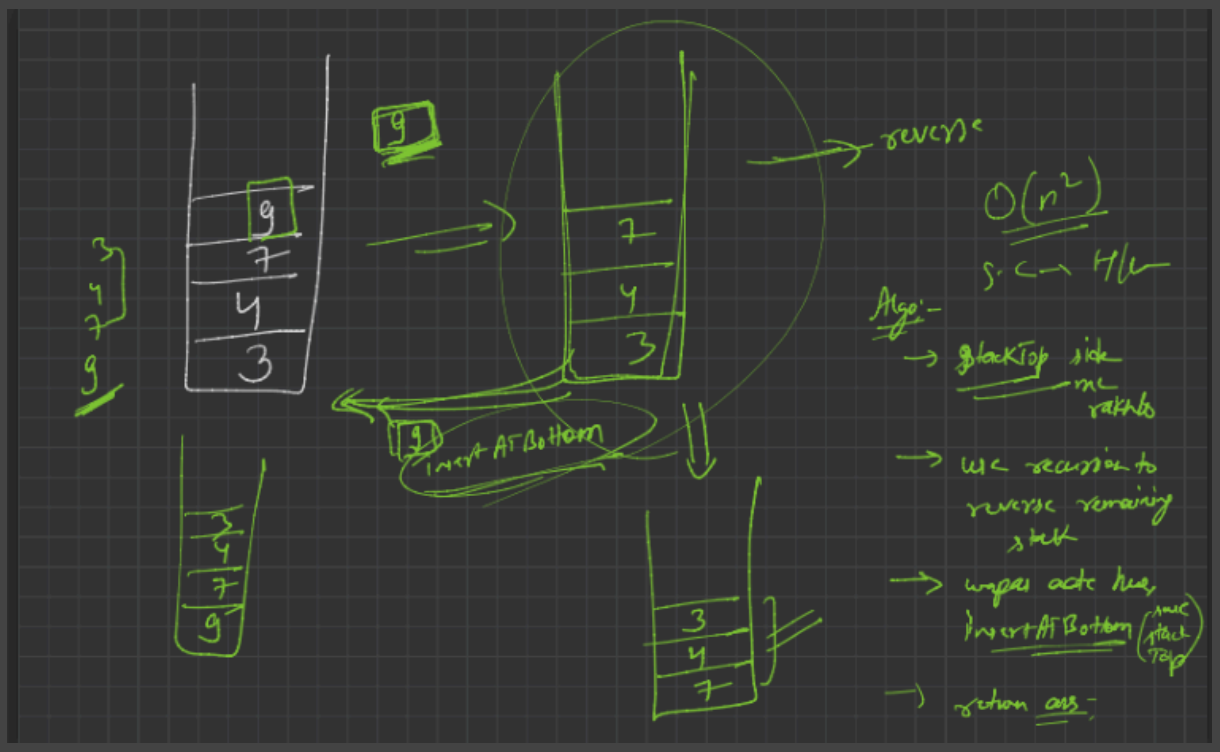
Check Whether the Parenthesis Are Valid:



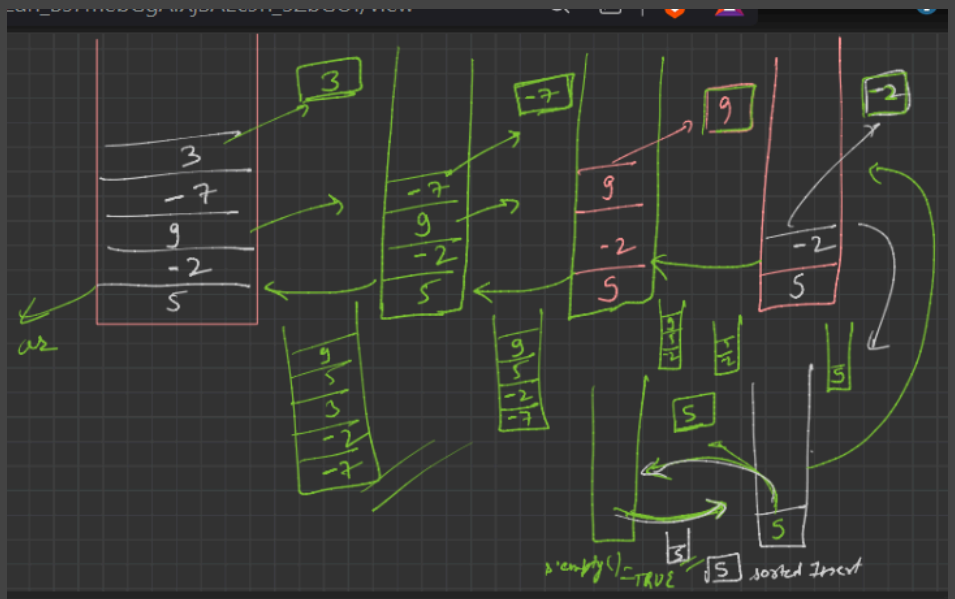
Insert An Element At Its Bottom In A Given Stack:

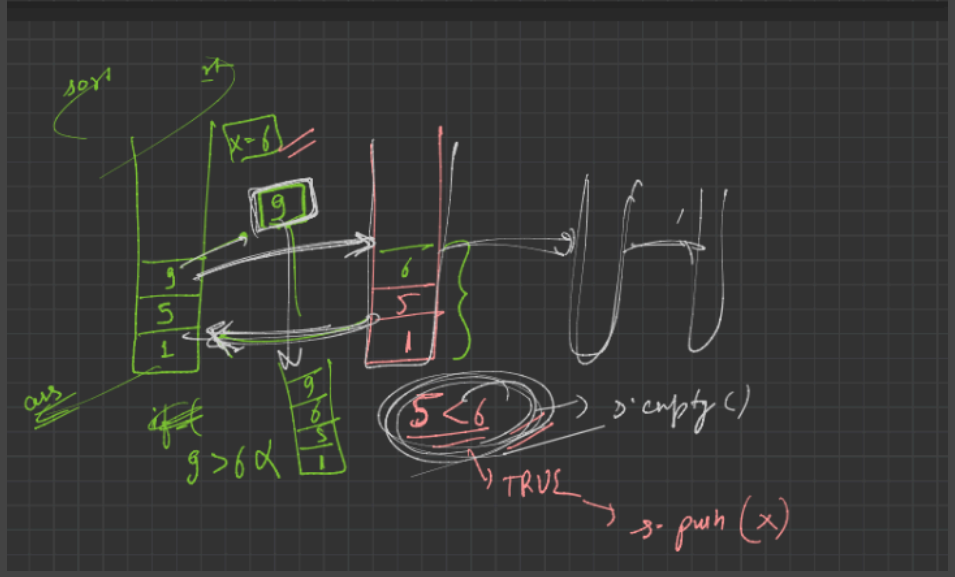


Reverse A Stack:

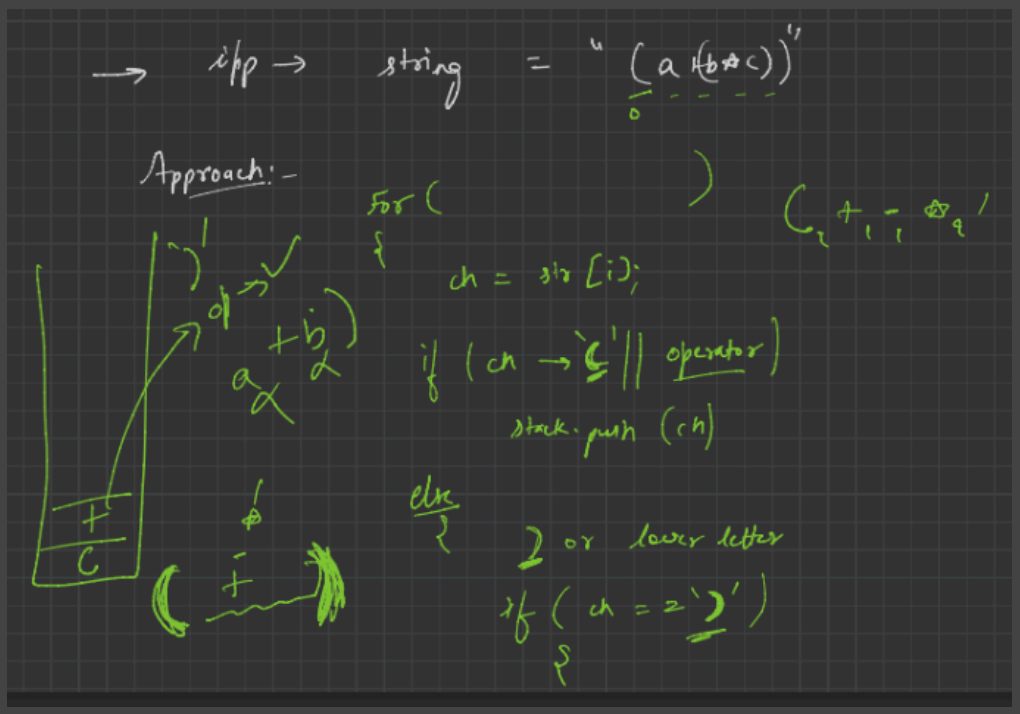


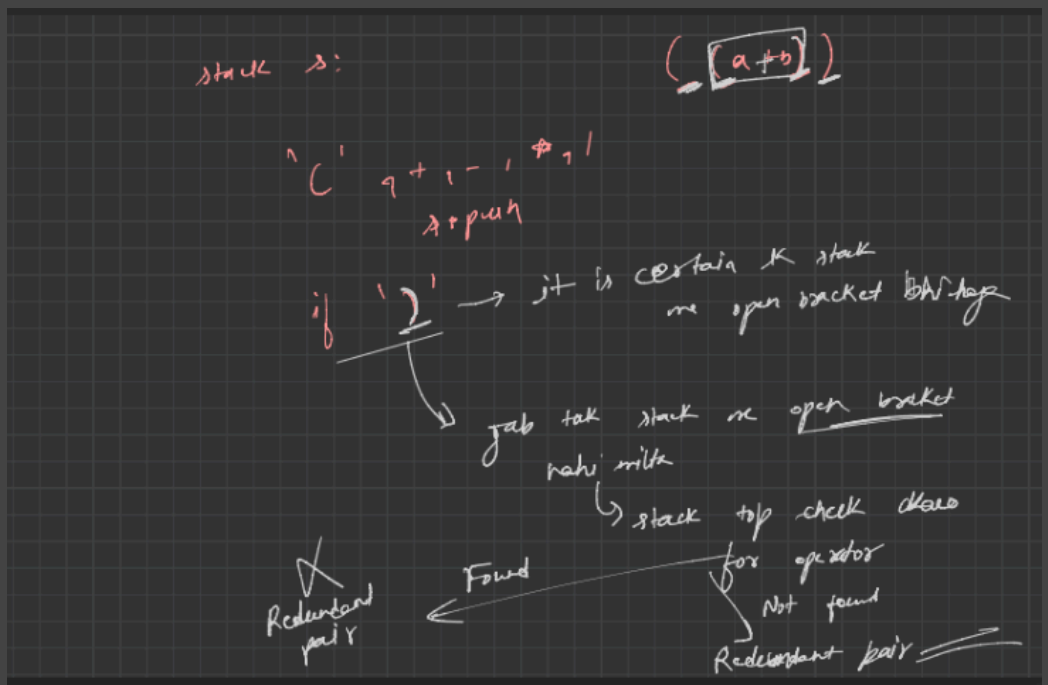
Sort A Stack:

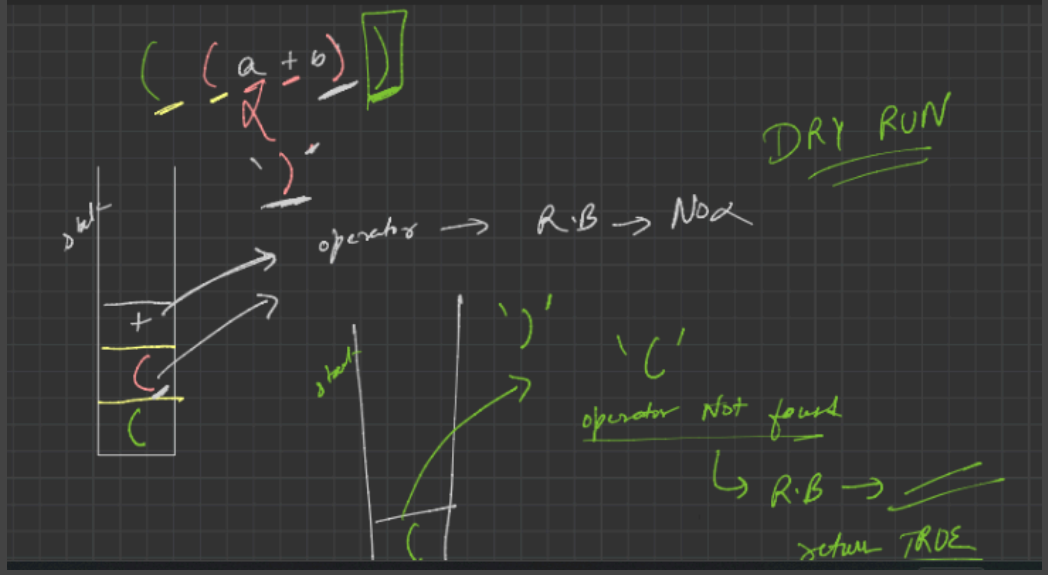




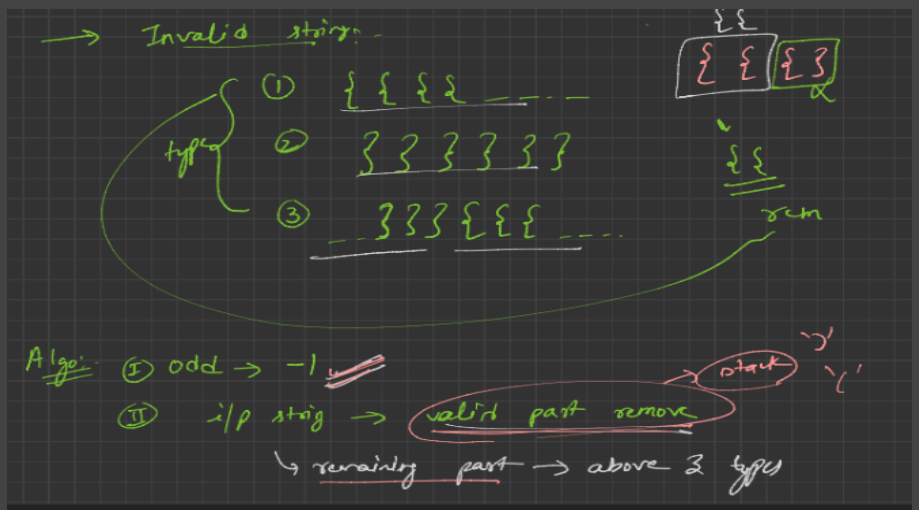
Check Whether There Are Redundant Brackets Or Not:

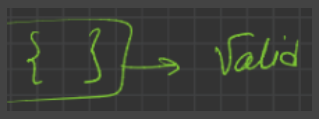


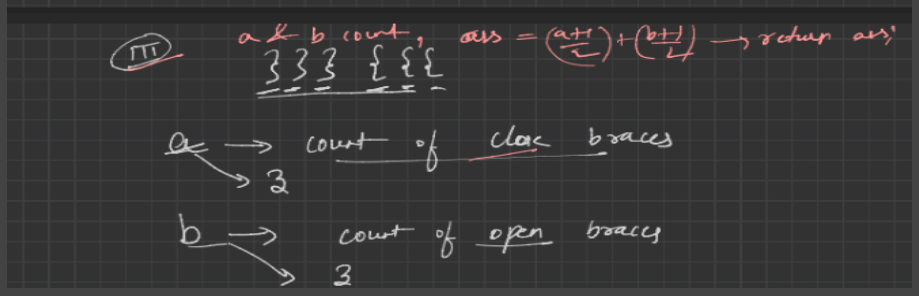


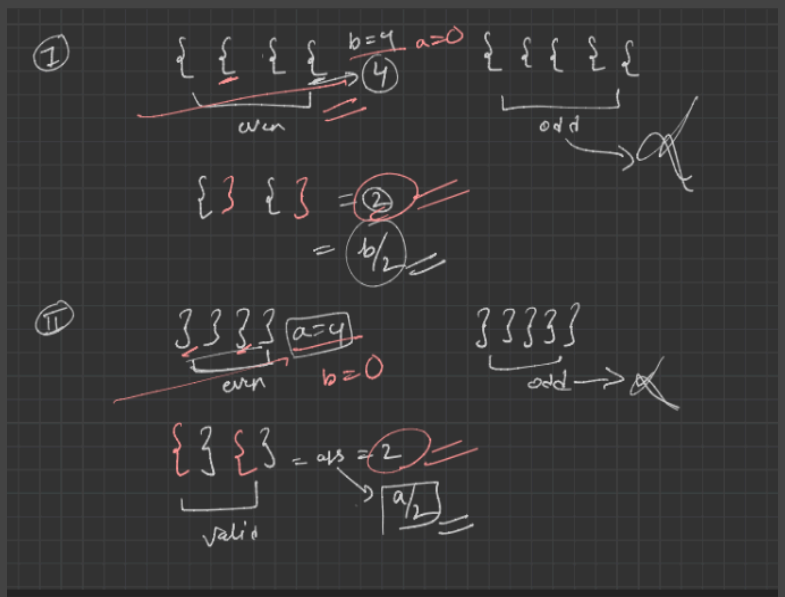


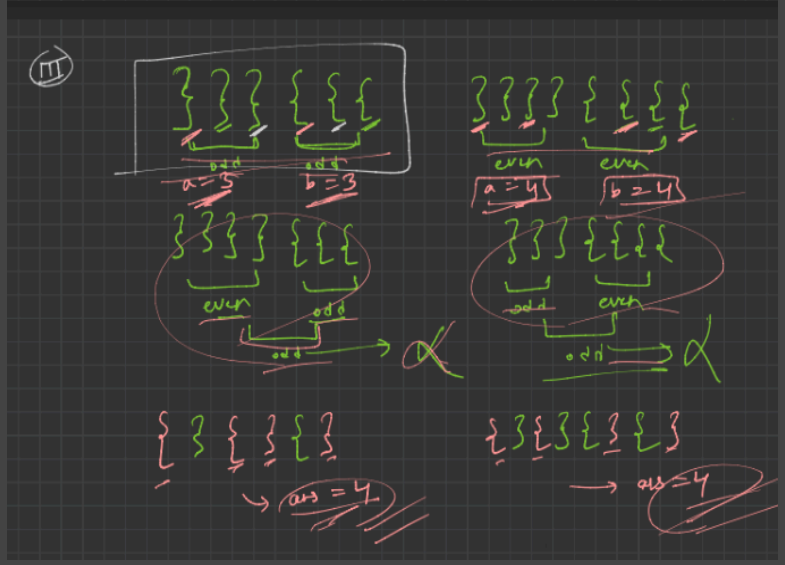
Minimum Number Of Operations To Make A Valid Bracket String:

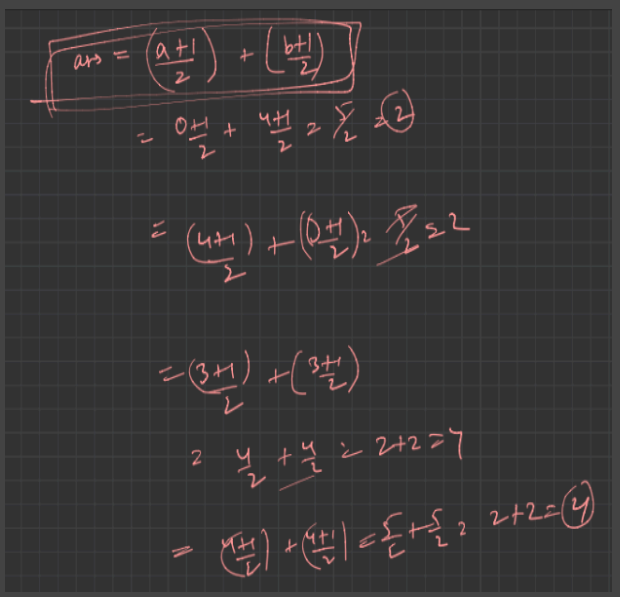




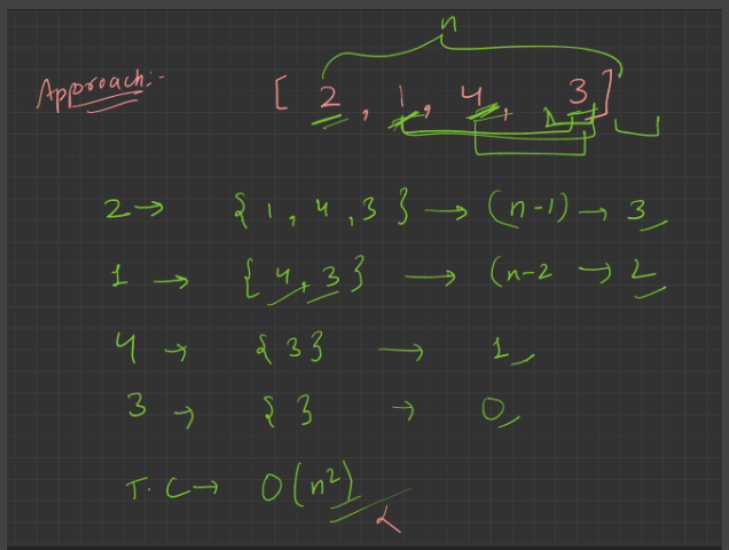


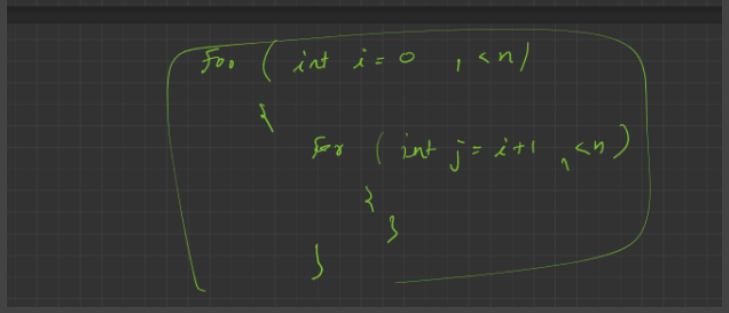




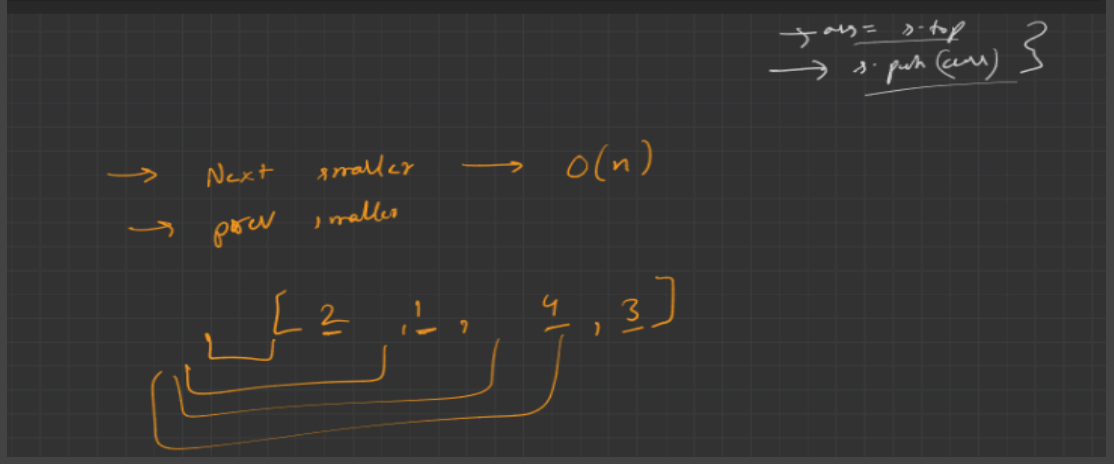


Next Smaller Element Approach-I:





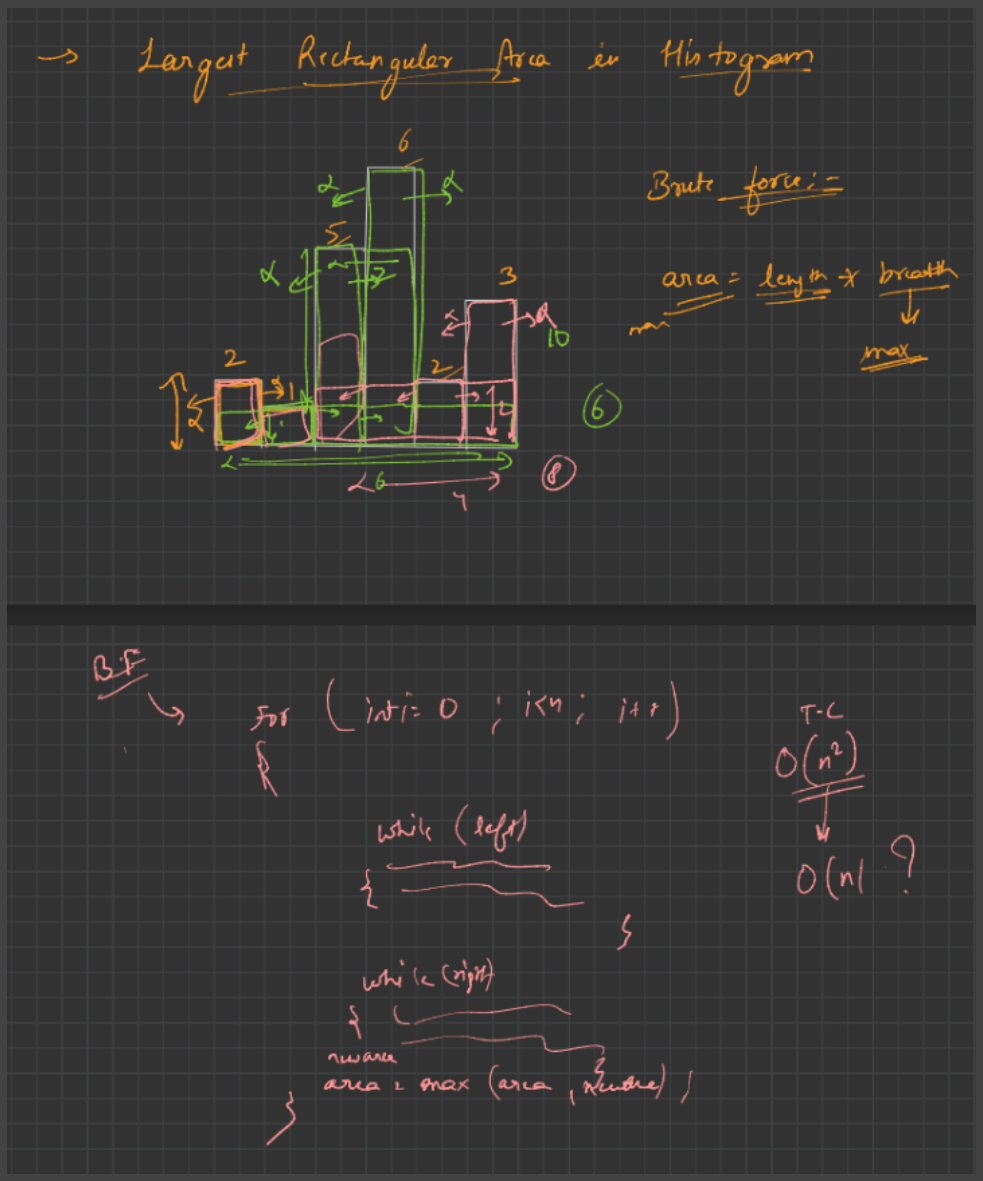
Next Smaller Element Approach-II:



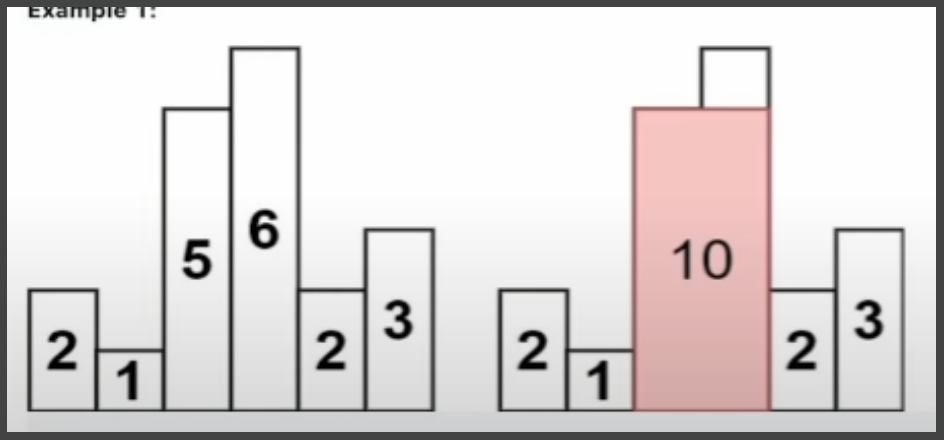
Previous Smaller Element:

If you want to find the previous smaller element for each element in the array, you would need to modify the code to iterate from left to right instead of from right to left.

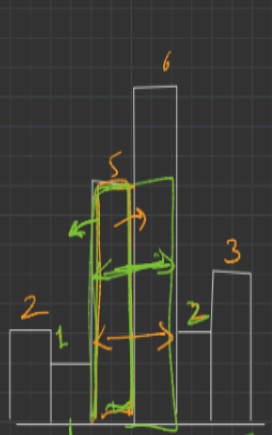
Largest Rectangle In Histogram Approach-I:



Largest Rectangle In Histogram Approach-II:



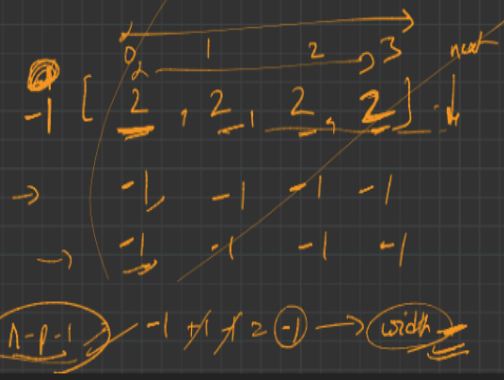
O(n) → Approach #2



width = n-p-1

area = $\frac{\text{base}}{3} \times h$ = 4-1-1 = 2

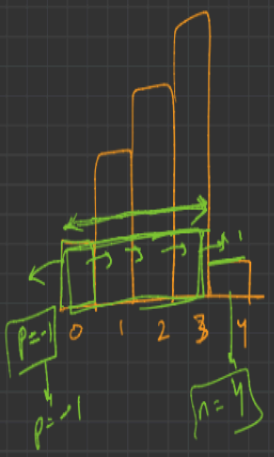
2 2 2 2



next → -1 -1 -1 -1

prev → -1 -1 -1 -1

n-p-1 = -1 ≠ 1 ≠ 2 → -1 → width



width = n-p-1

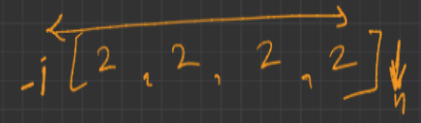
= 4-(-1)-1

= 4 ≠ 1 ≠ 1

= 4

next → next smaller element index

prev → prev



next = -1 -1 -1 -1 if(next == -1

prev = -1 -1 -1 1

4

n-p-1

= -1 ≠ 1 ≠ -1 → break/width = -1

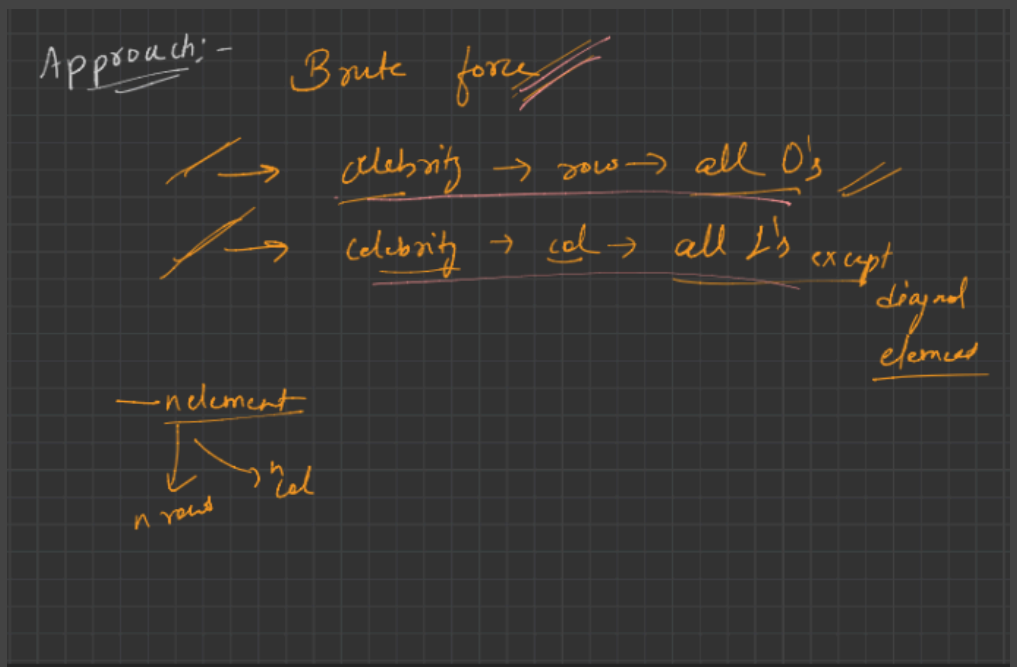
area = -ve

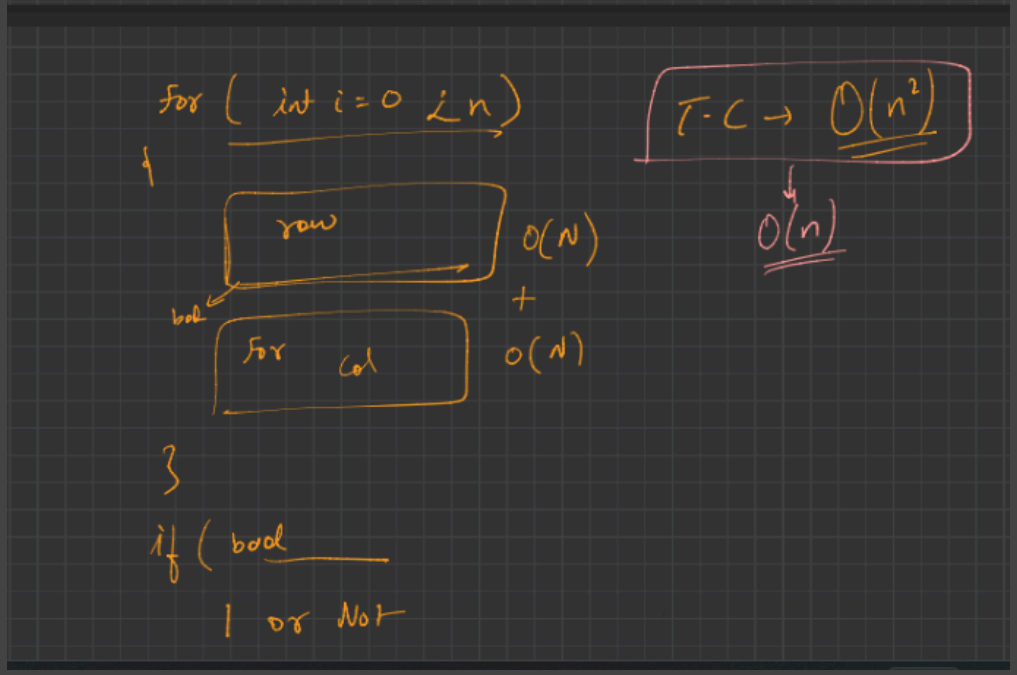
T.C → O(N)

S.C → O(N)

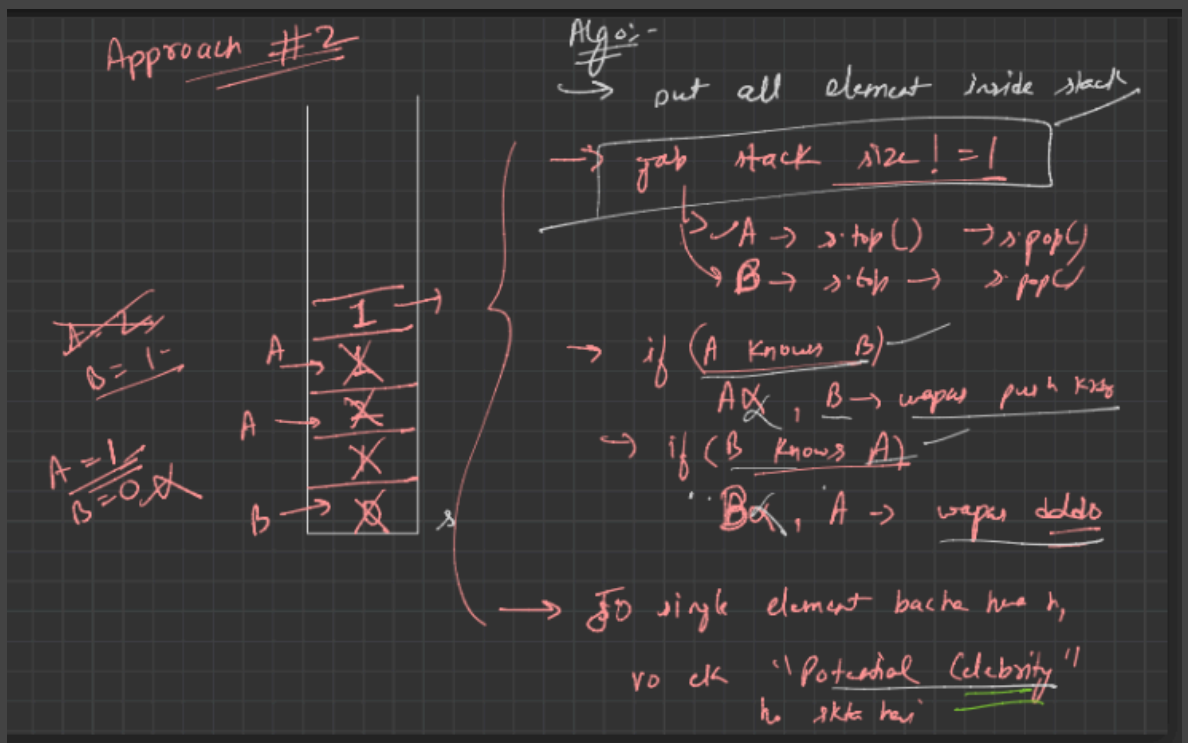
Hoza aagya

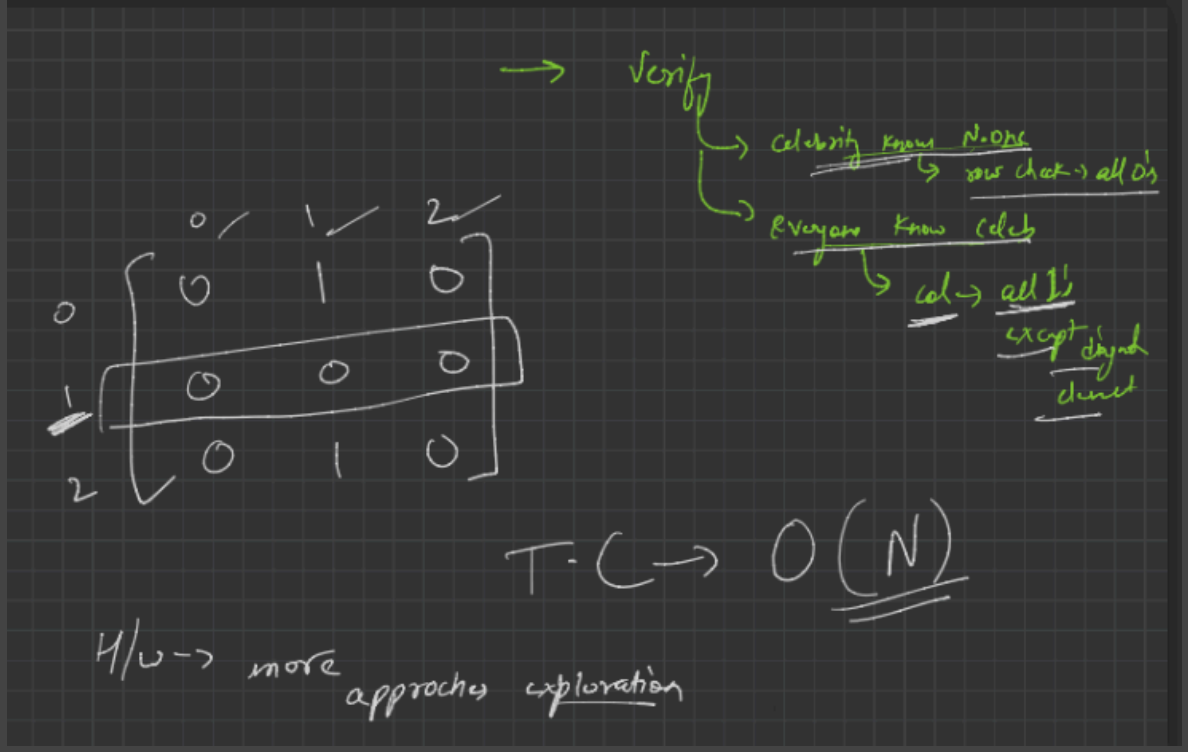
Celebrity Problem Approach-I:



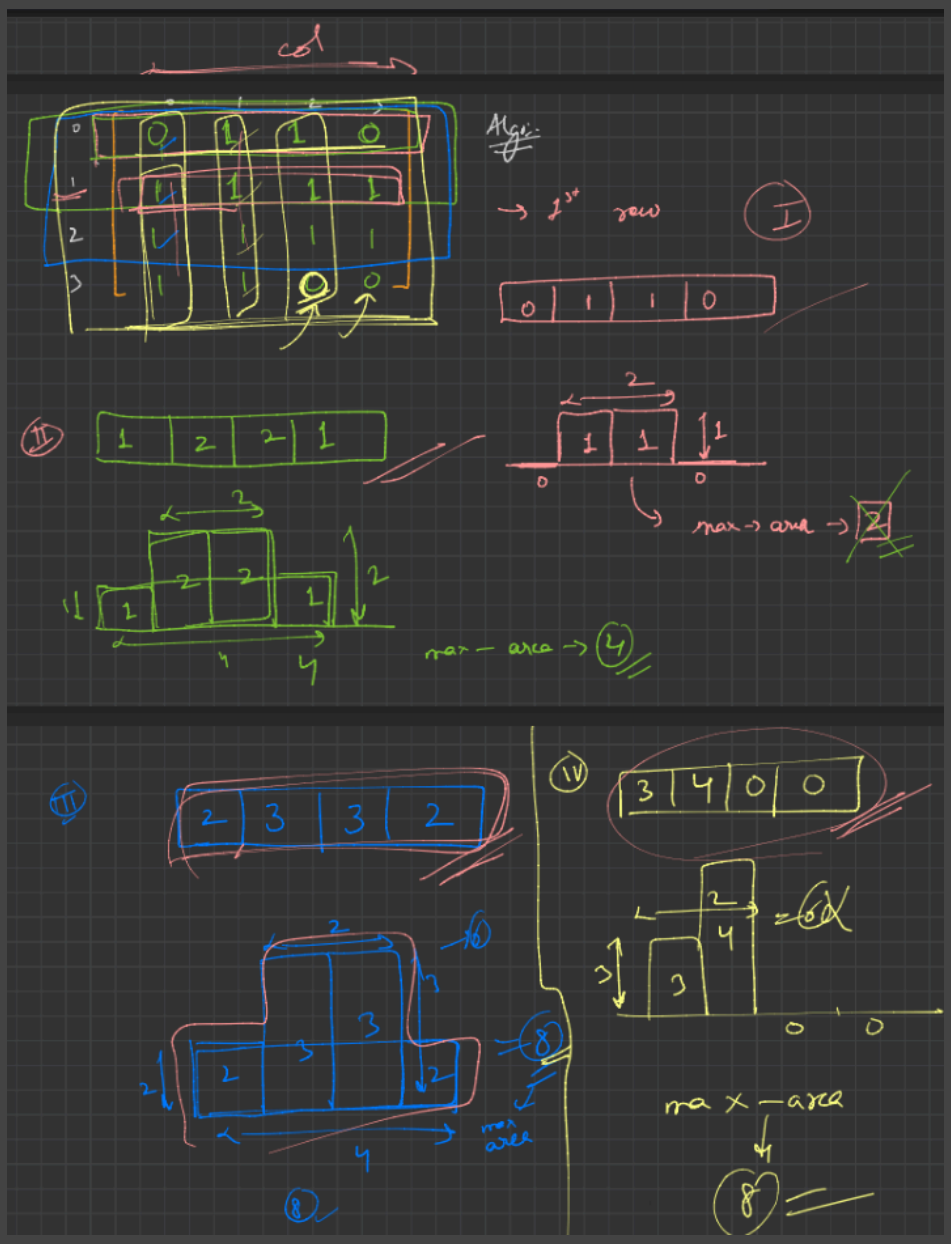


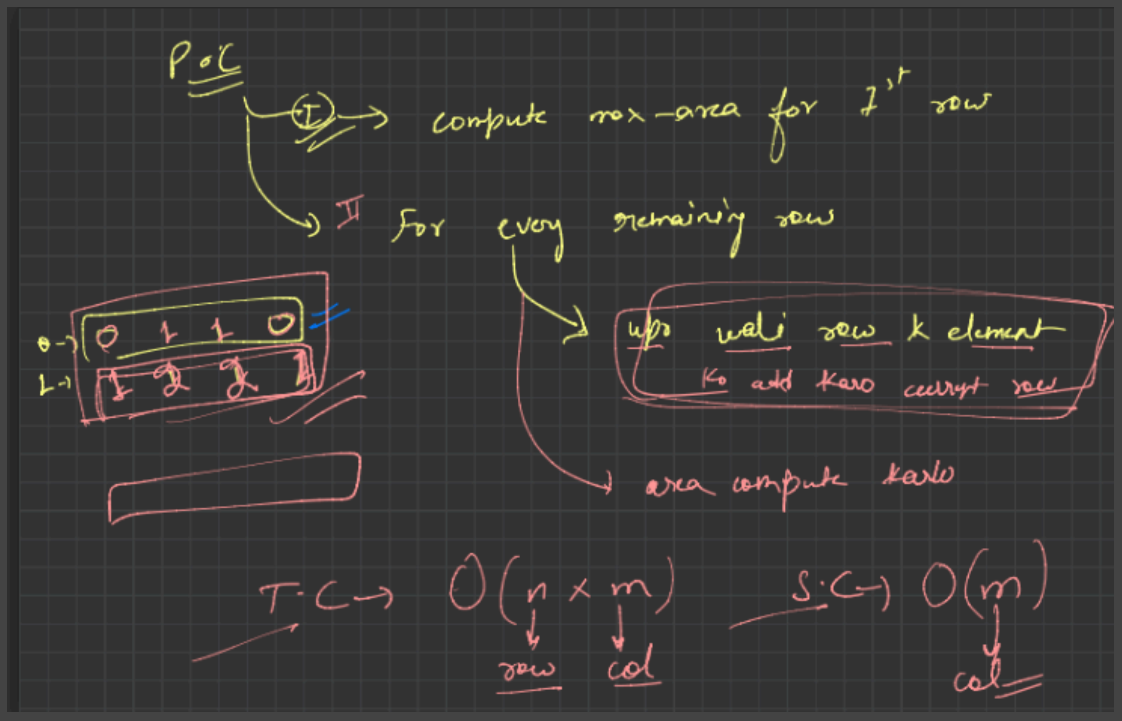
Celebrity Problem Approach-II:





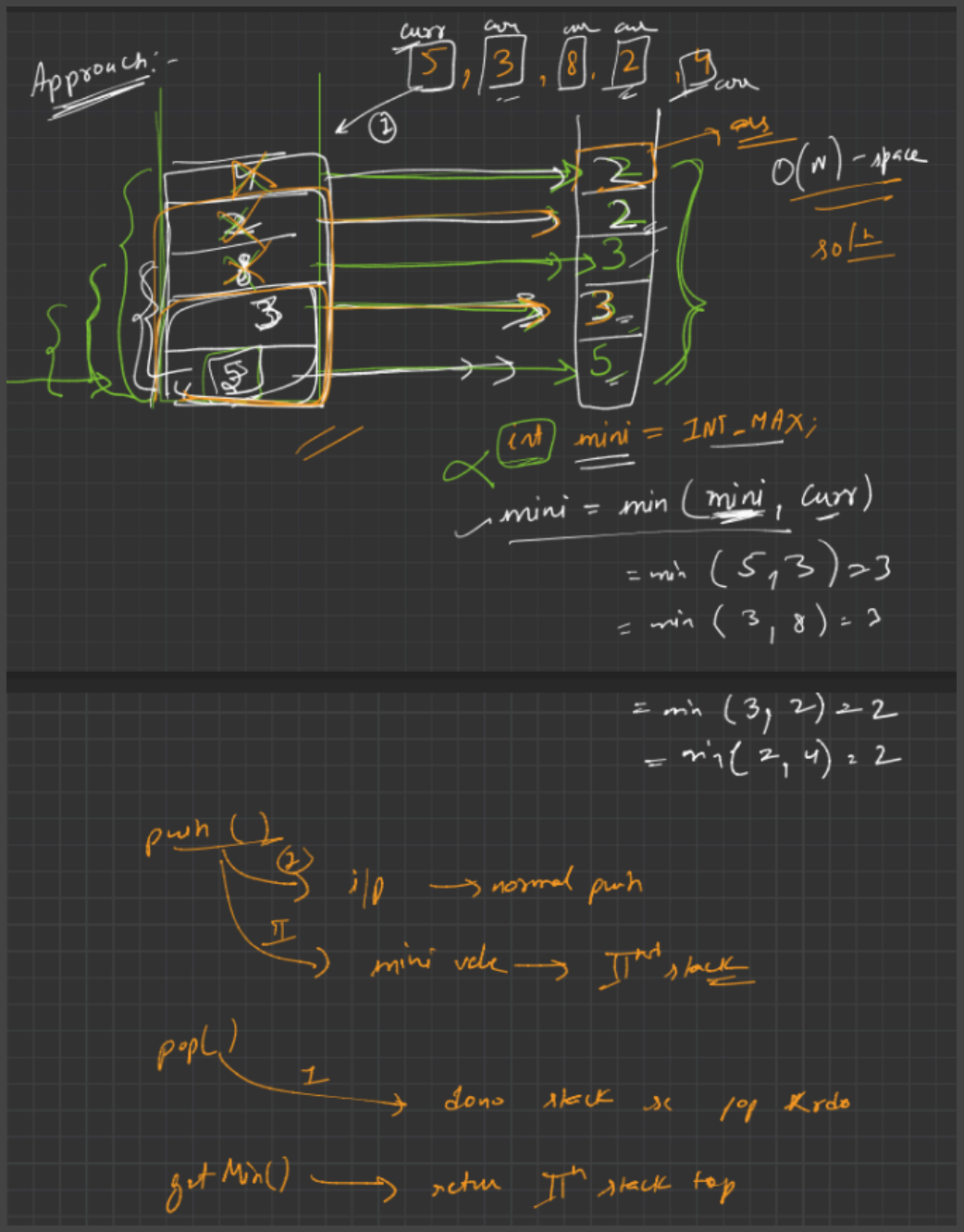
Maximum Rectangle In Binary Matrix:

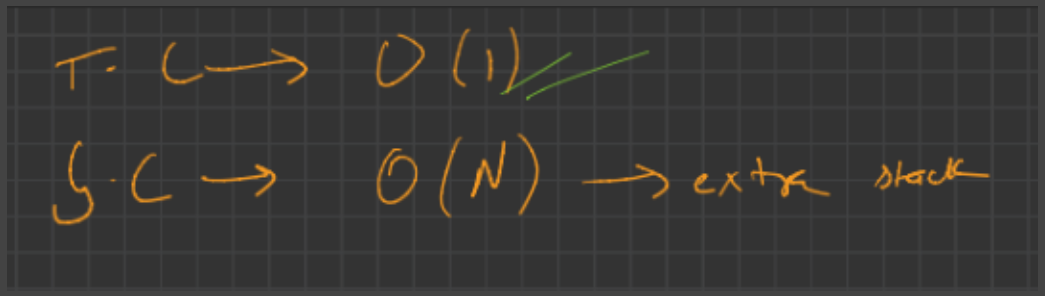




Design A Stack That Supports getMin in O(1) Time Complexity and O(1) Space Complexity:

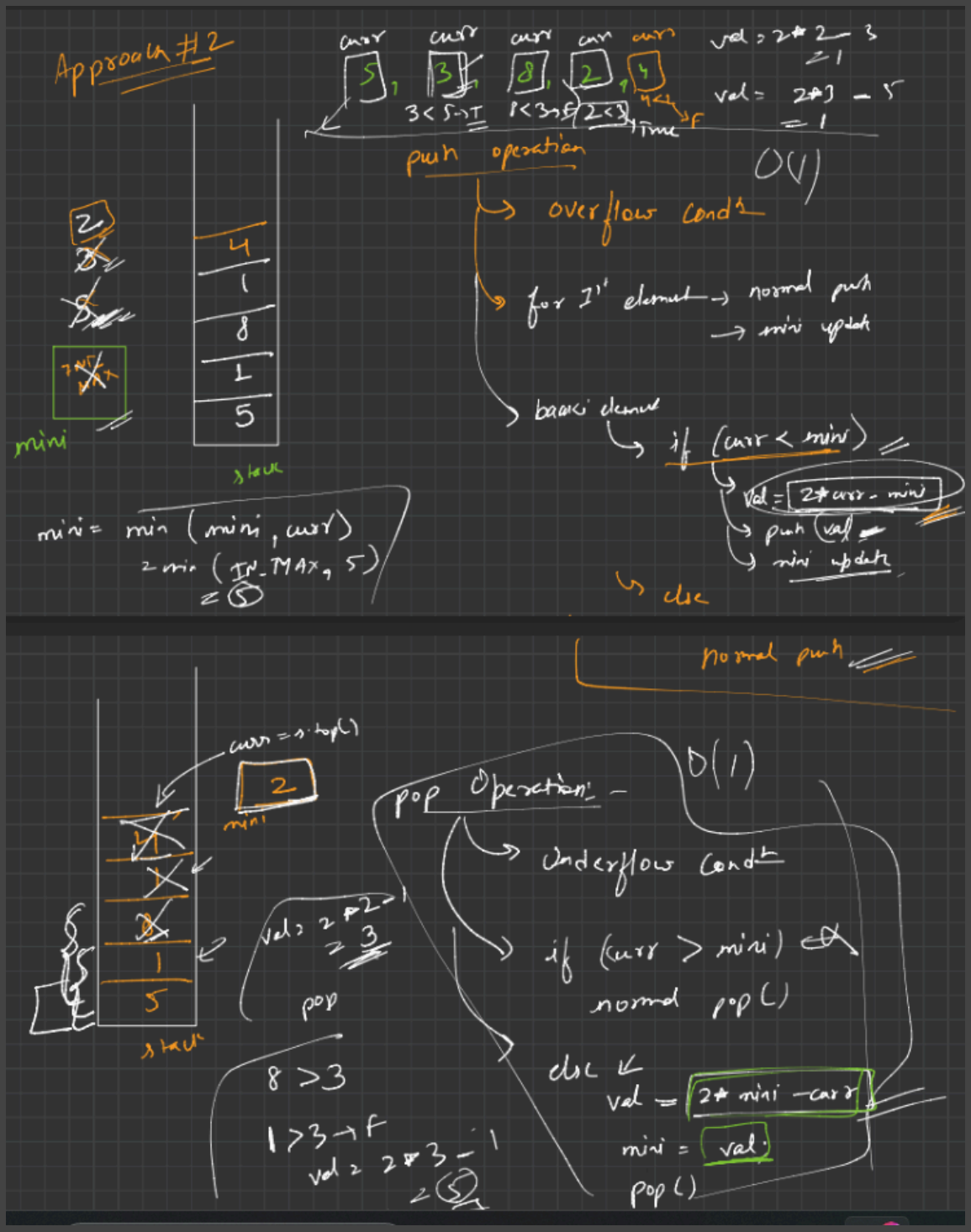
Approach -I:





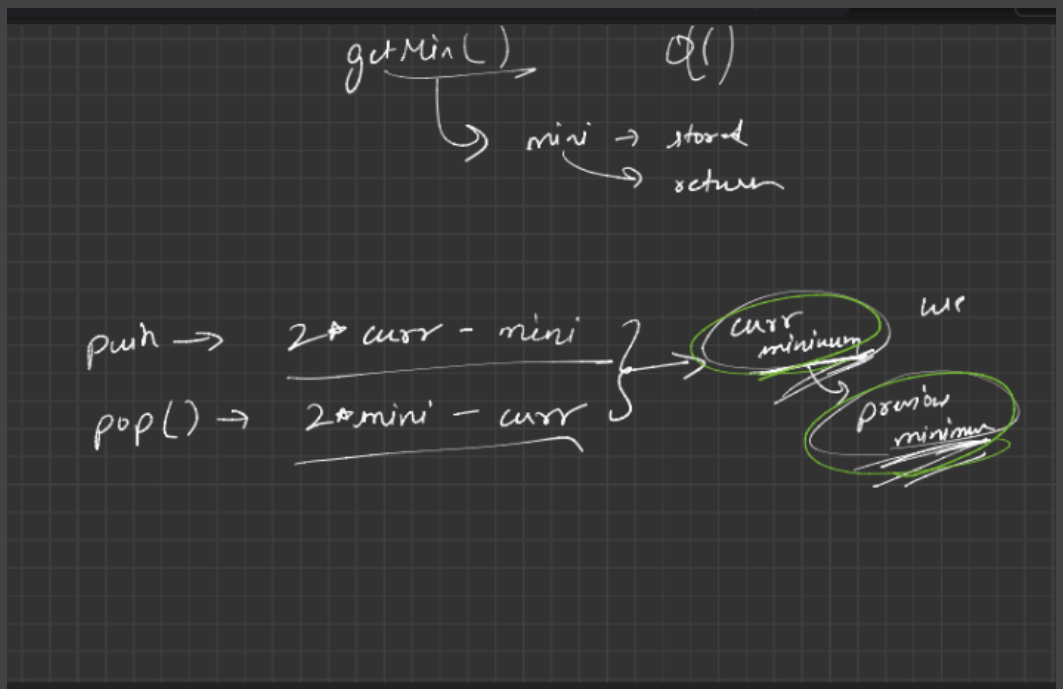
Design A Stack That Supports getMin in O(1) Time Complexity and O(1) Space Complexity:

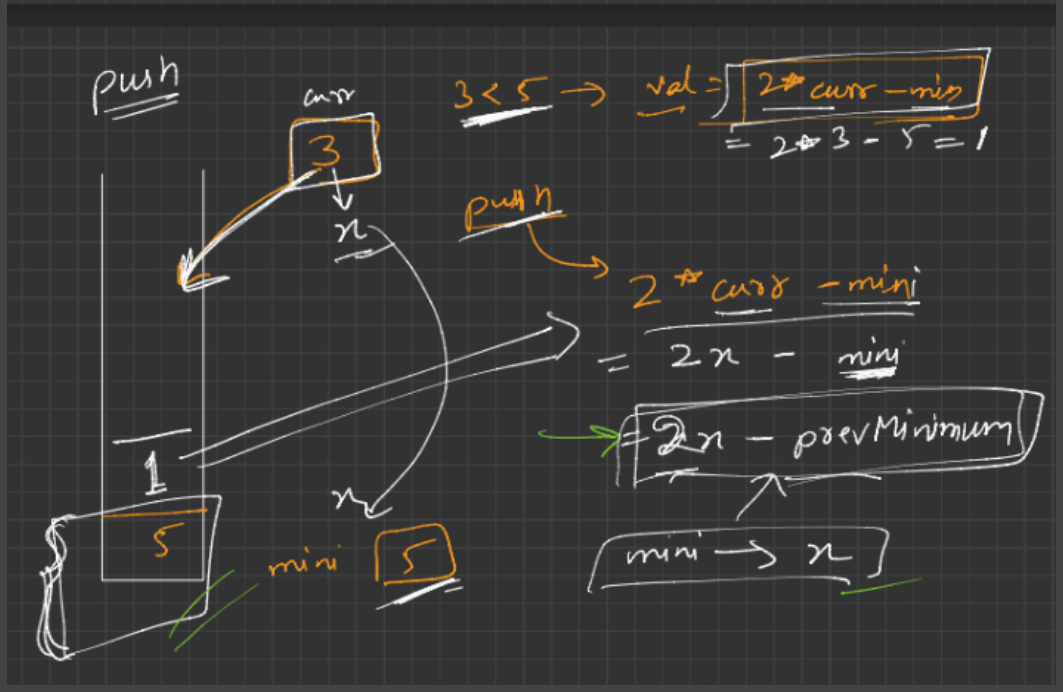
Approach -II:

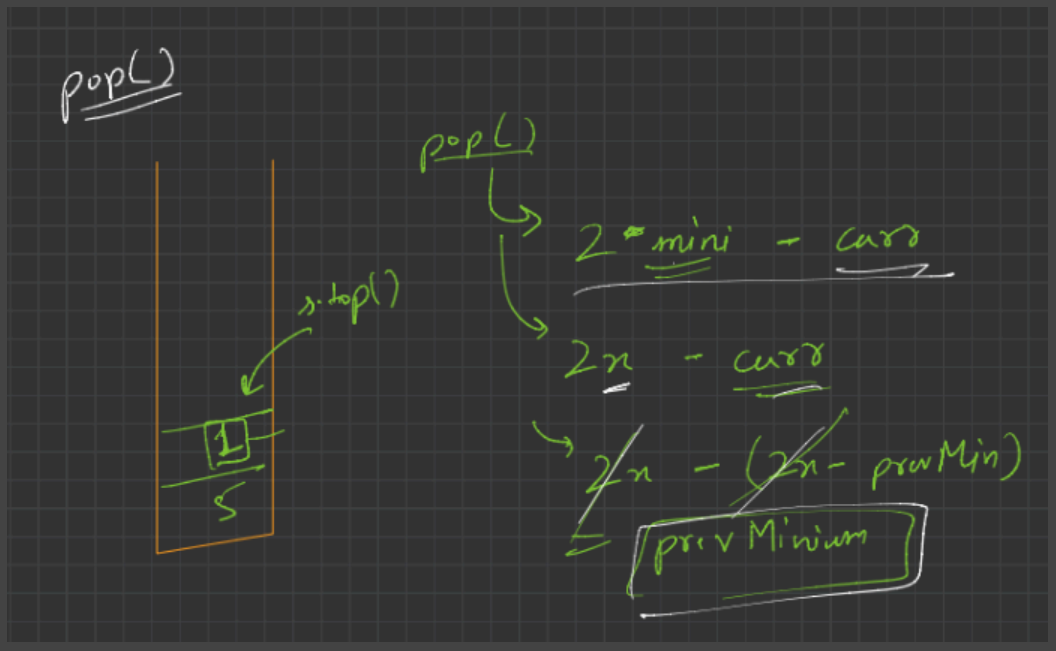


The SpecialStack class uses a single stack s to store the elements. The mini variable is used to keep track of the minimum element. When a new element is pushed onto the stack, if it is less than the current minimum, the stack stores a value derived from the new element and the current minimum (specifically, 2*data - mini). This derived value is not an additional element, but a transformation of the new element being pushed. Therefore, it does not count as additional space.

When an element is popped from the stack, if it is less than the current minimum, the minimum is updated using the derived value and the current minimum (specifically, 2*mini - curr). Again, this does not use additional space, as it's a calculation performed on existing values.

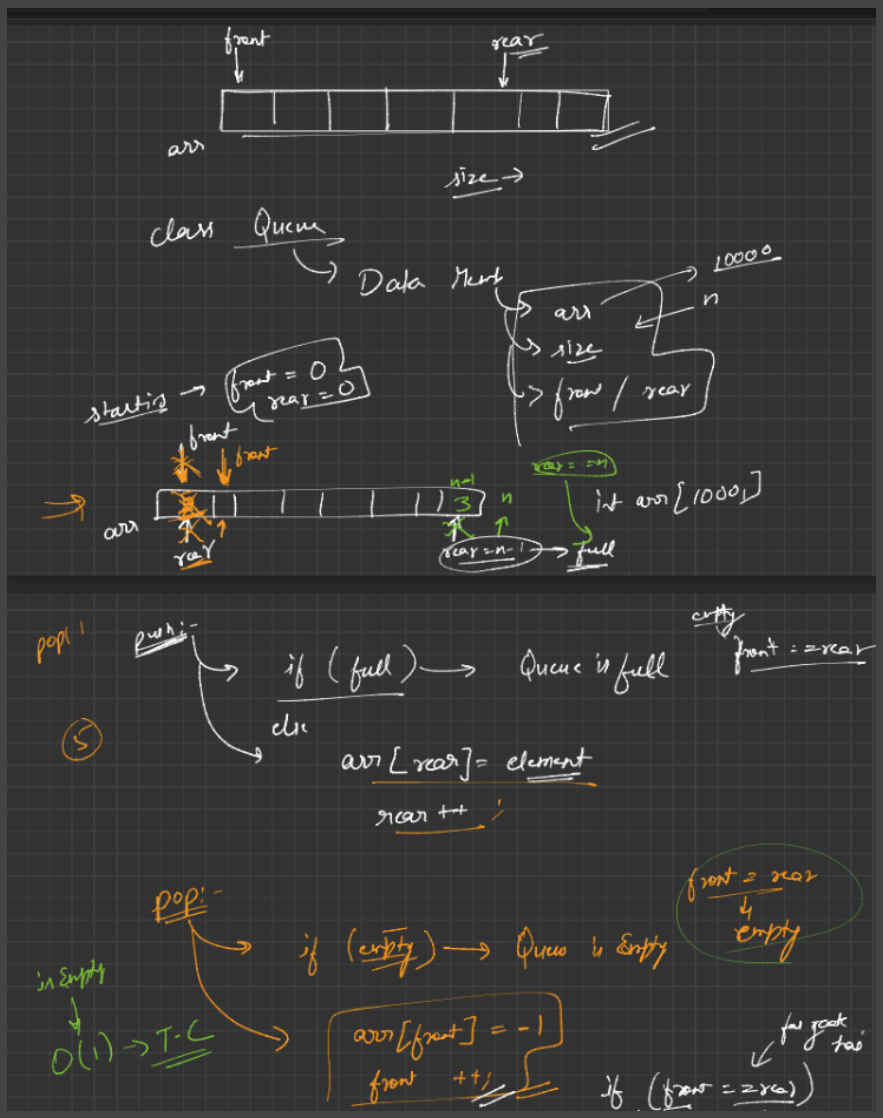

getMin()
pop()
mini → stored
actual
push → 2*curr - mini
pop() → 2*mini - curr
curr minimum
we
previous minimum


push
curr
3
3 < 5 → val = 2*curr - mini
= 2*3 - 5 = 1
push n
2*curr - mini
= 2n - mini
= 2n - prevMinimum
mini → n
1
5
mini 5


pop()
s.top()
1
5
pop()
2*mini - curr
2n - curr
2n - (2n - prevMin)
prevMinimum

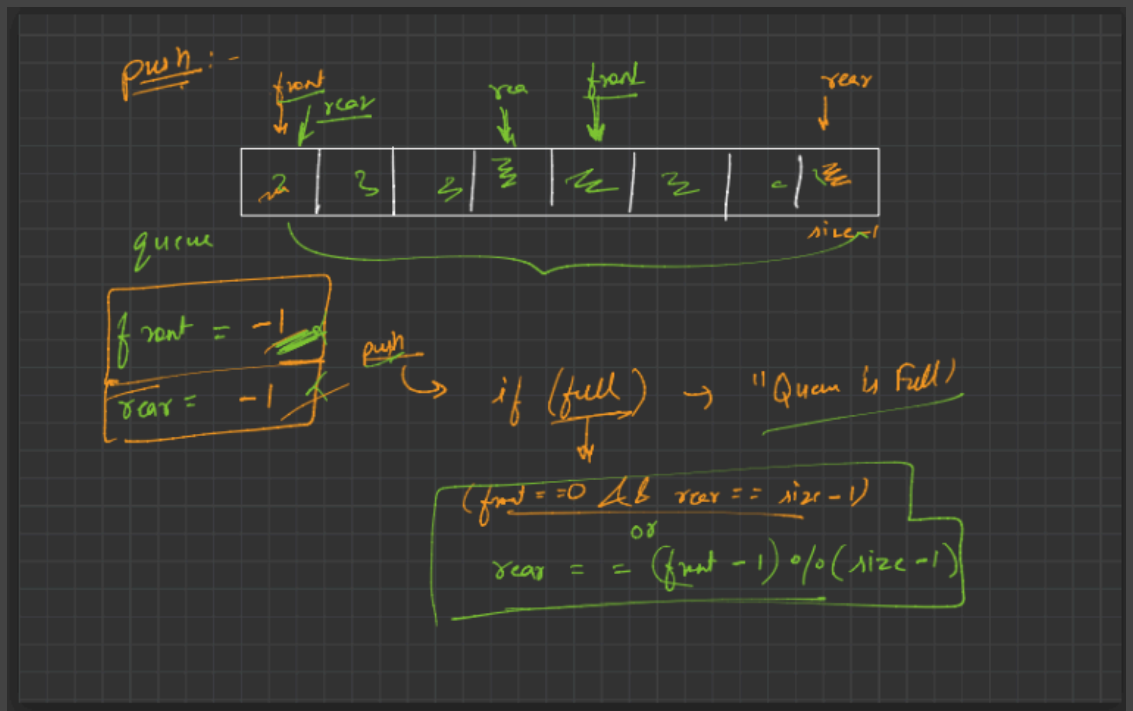
Queue

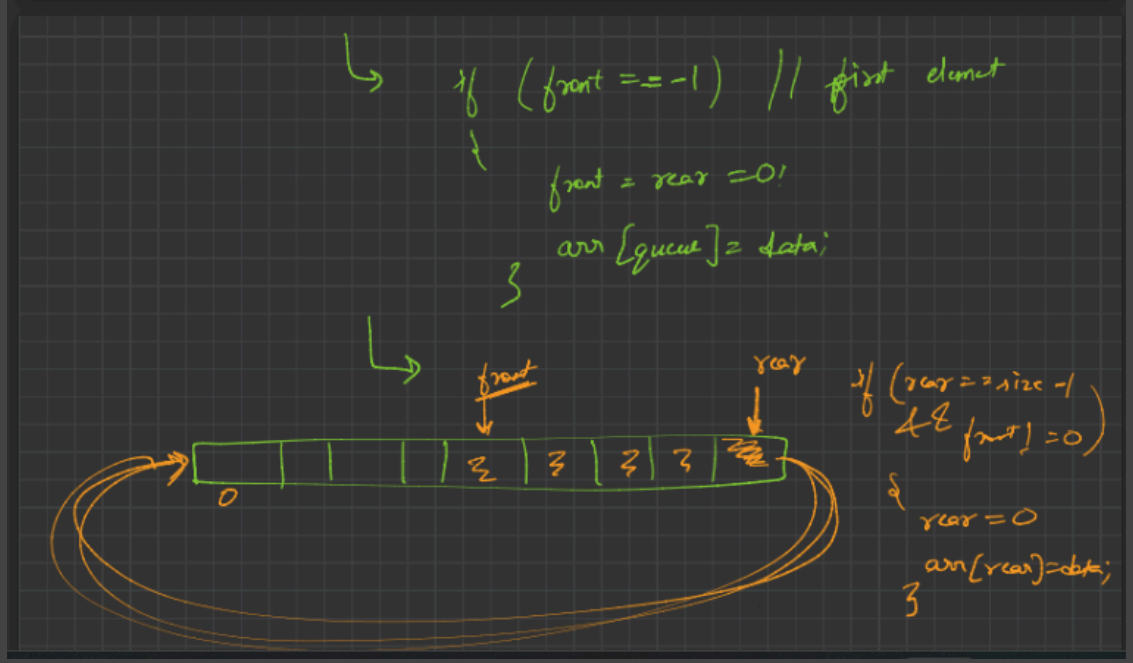
Implementing Queue Using Arrays:

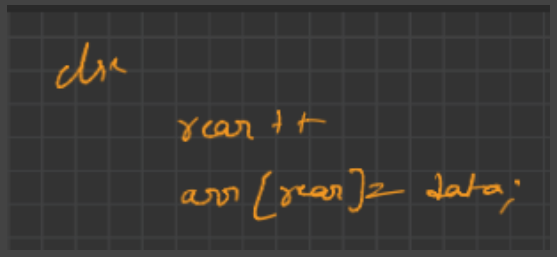


Implementing Queue Using Linked-List:

Circular Queue:







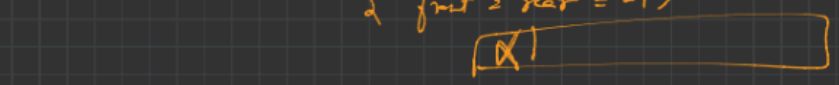
Dequeue / pop :-

→ if (empty) → Queue is Empty

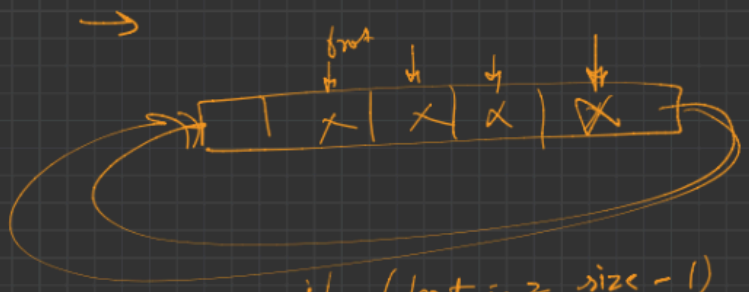
front == -1

→ // single element in queue

if (front == rear)
{ front = rear = -1;



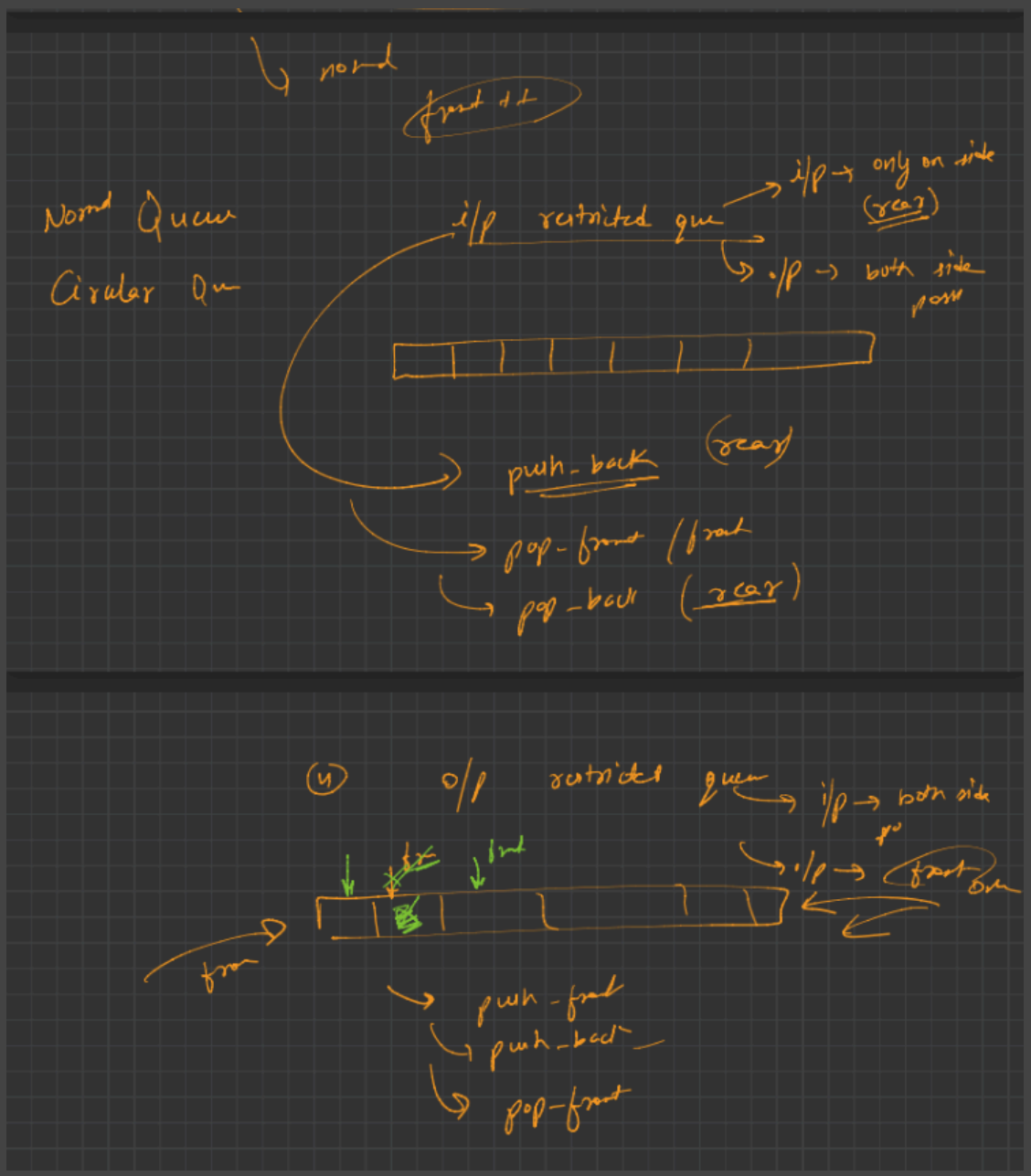
→



if (front == size - 1)
{ front = 0;

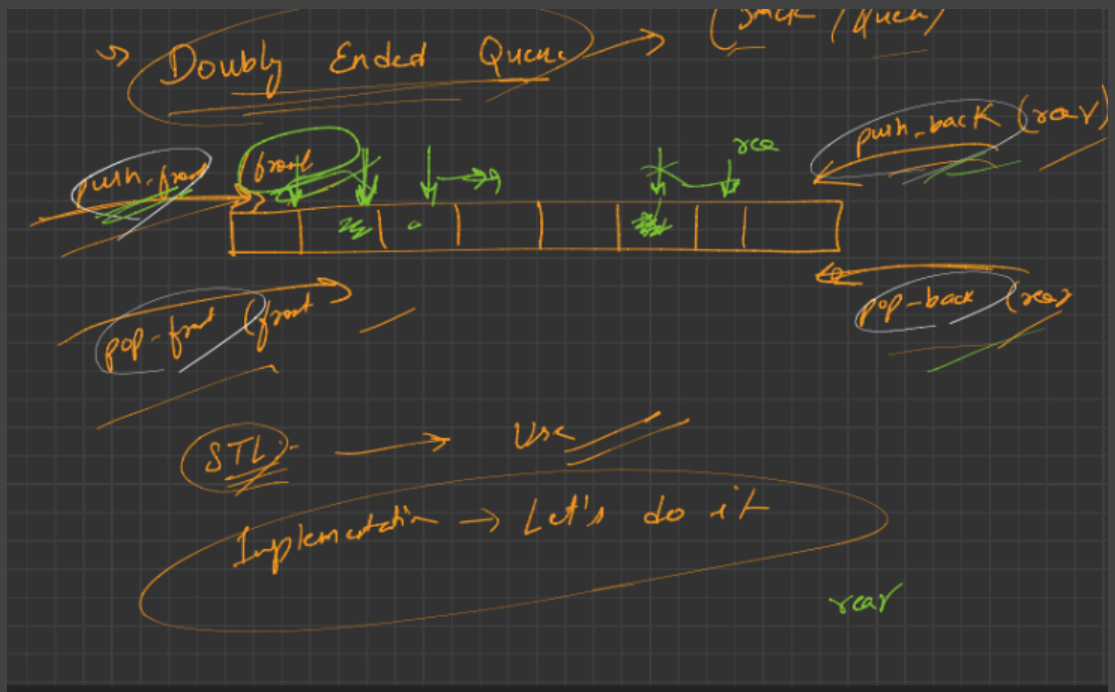
→ front ++;

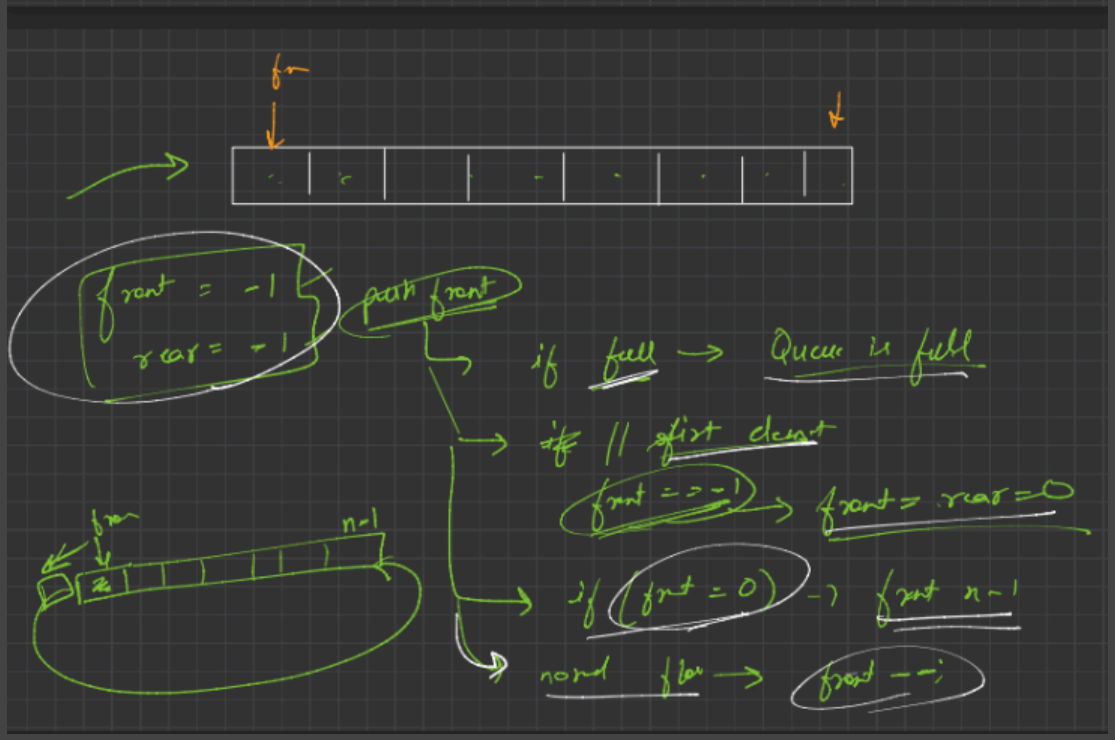
Input Restricted And Output Restricted Queue:

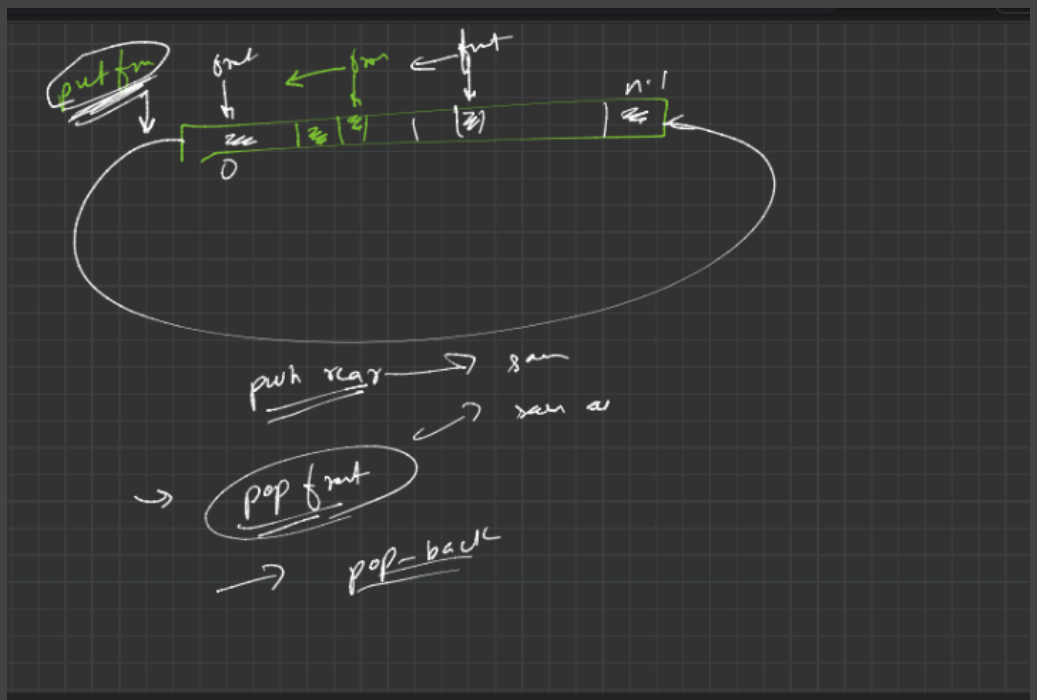


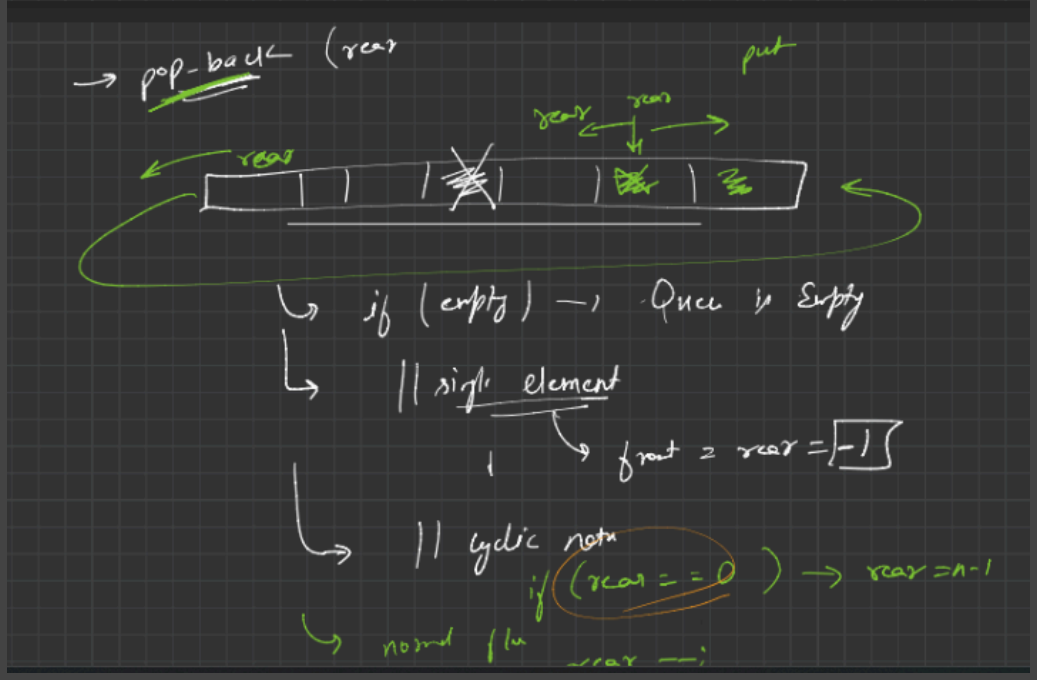
Doubly Ended Queue(Deque) STL Code:

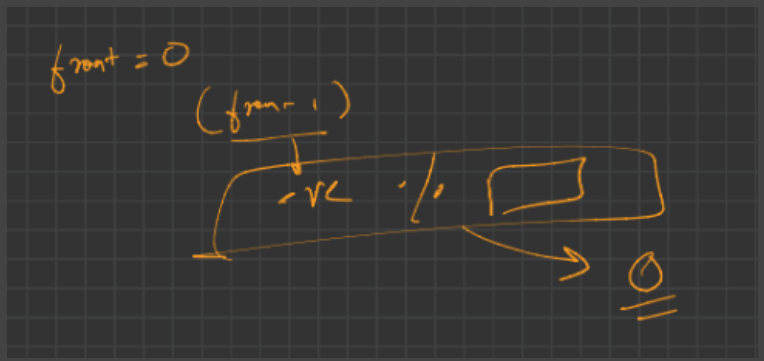
Doubly Ended Queue(Deque) Implementation:



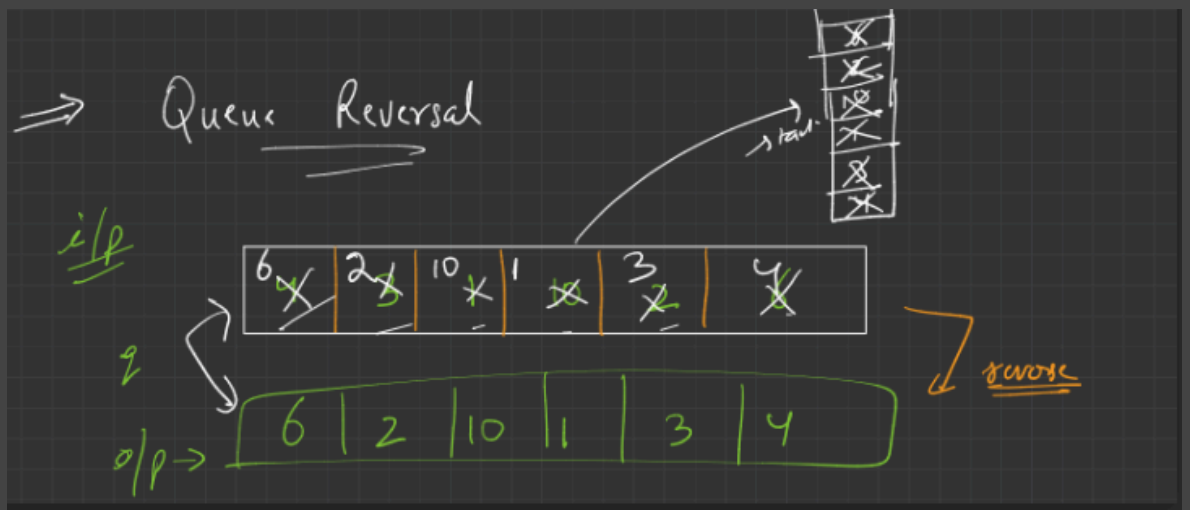


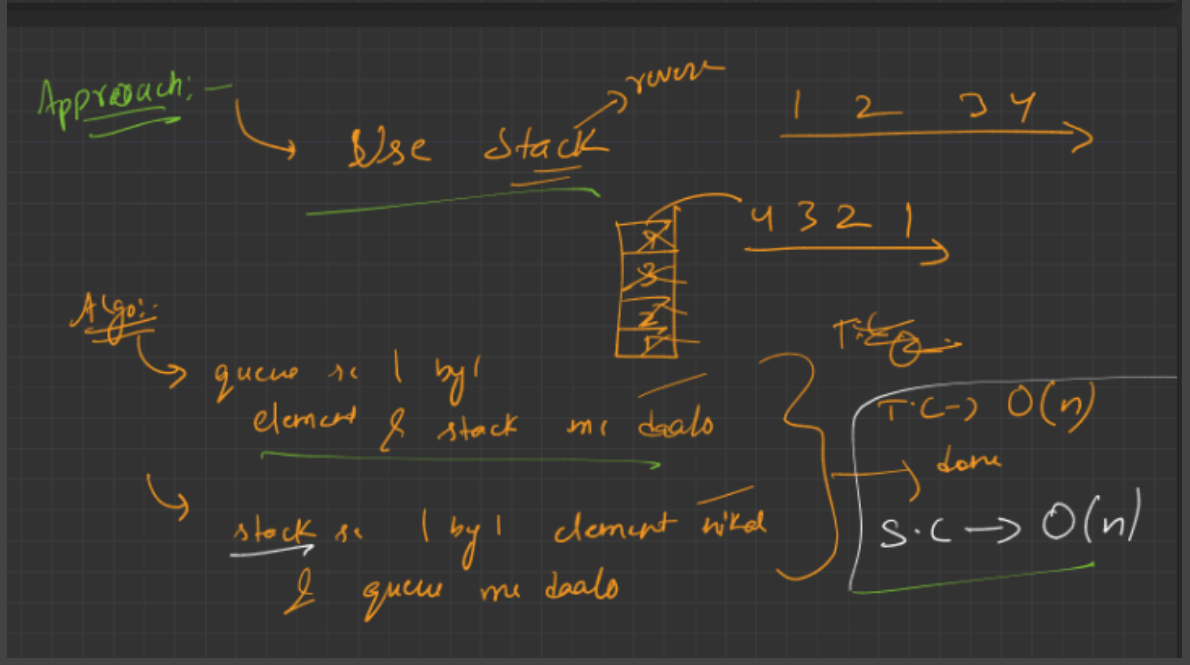

front
front
front
n-1
0
push rear → same
→ pop front
→ pop-back


→ pop-back (rear
rear
rear
rear
front
→ if (empty) → Queue is Empty
→ || single element
front = rear = -1
→ || cyclic note
if (rear == 0) → rear = n-1
→ normal flow
rear --;

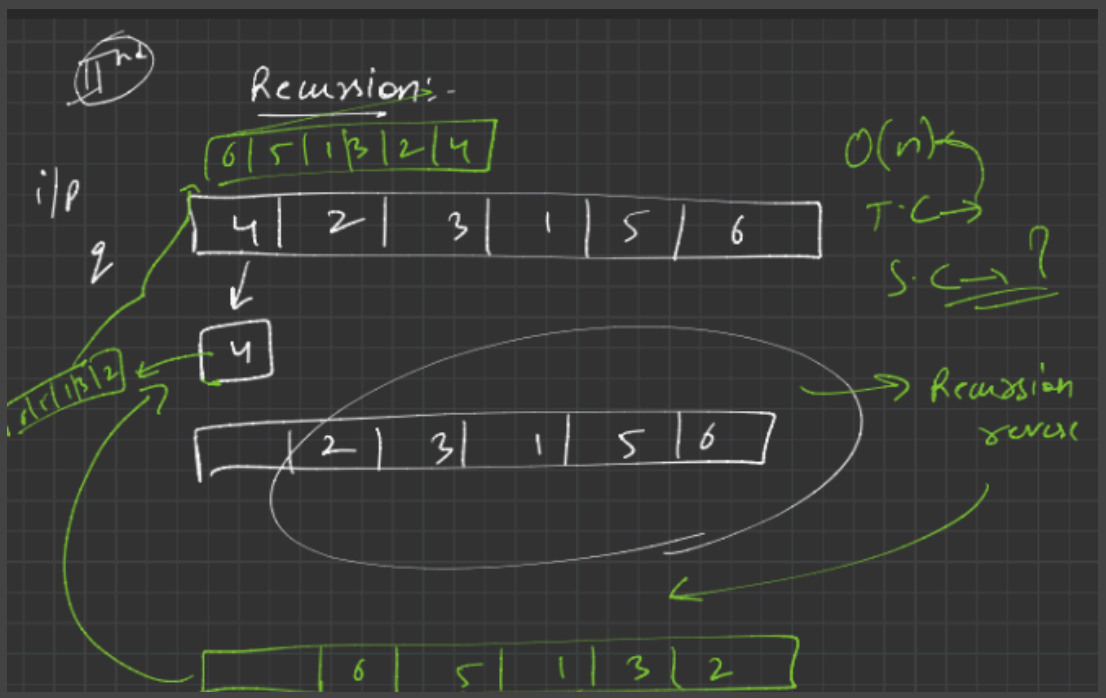

front = 0
(front - 1)
-ve
0

Reverse A Queue Approach-I:

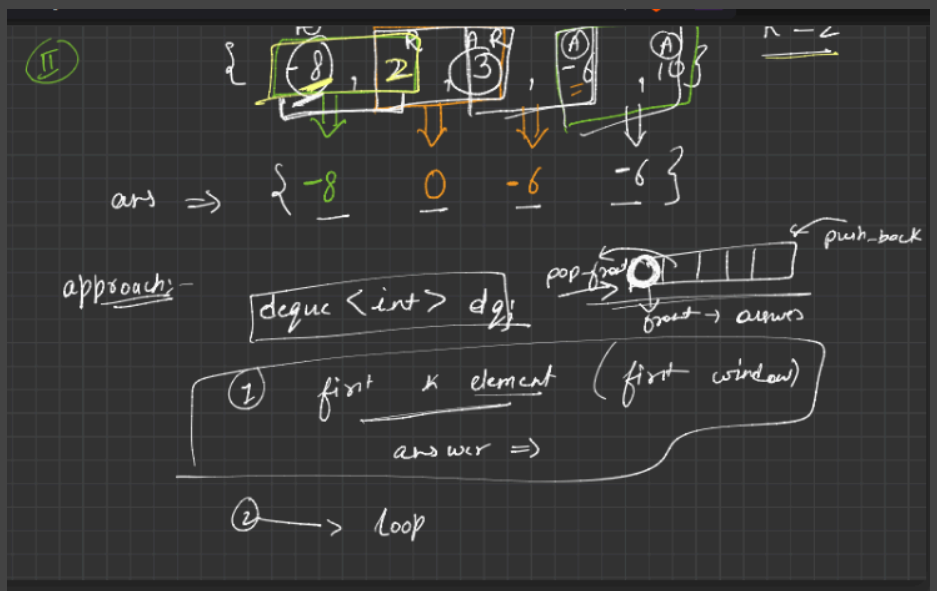


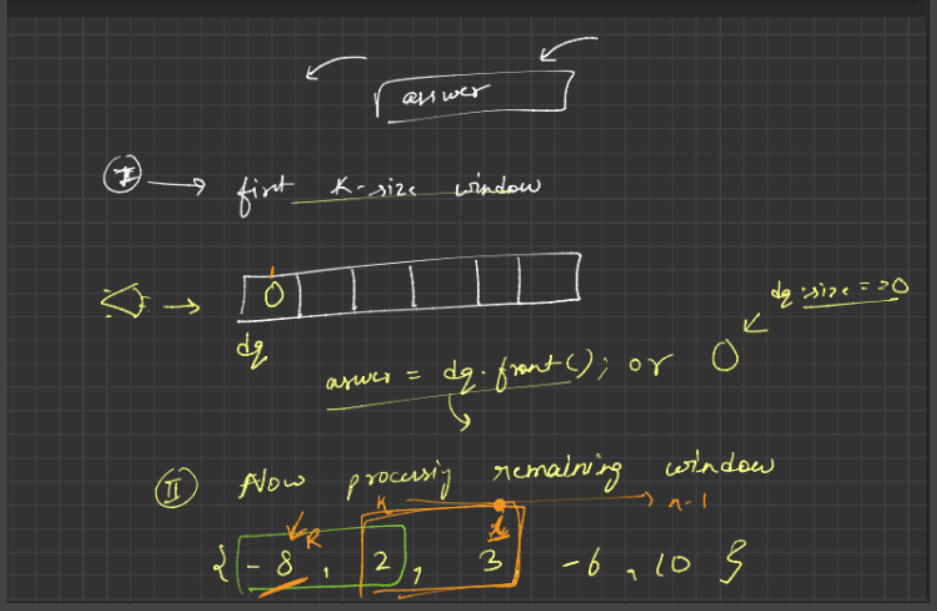


Reverse A Queue Approach-II:



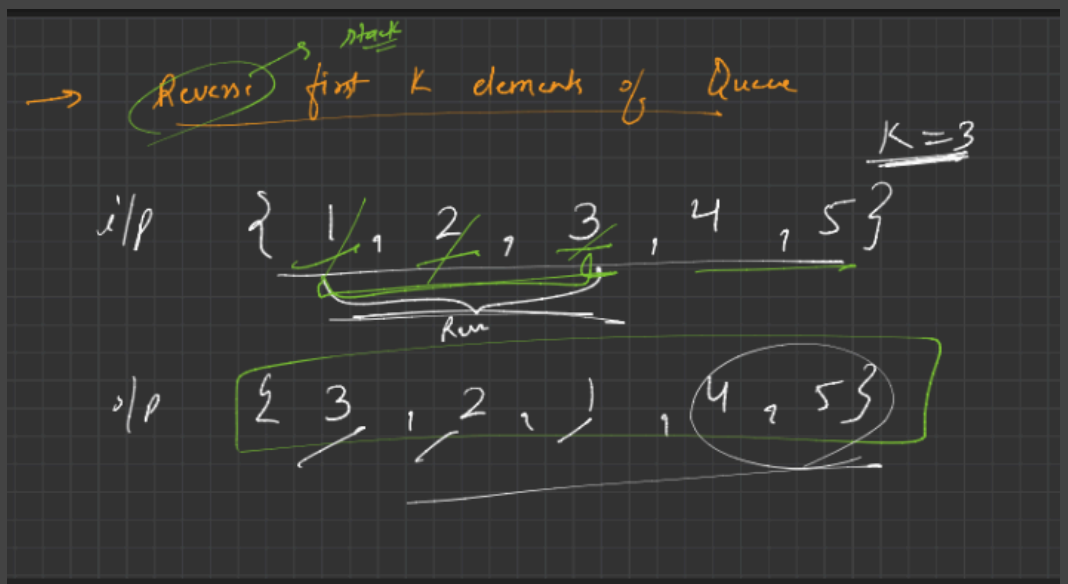
First Negative Integer In Every Window Of Size K:

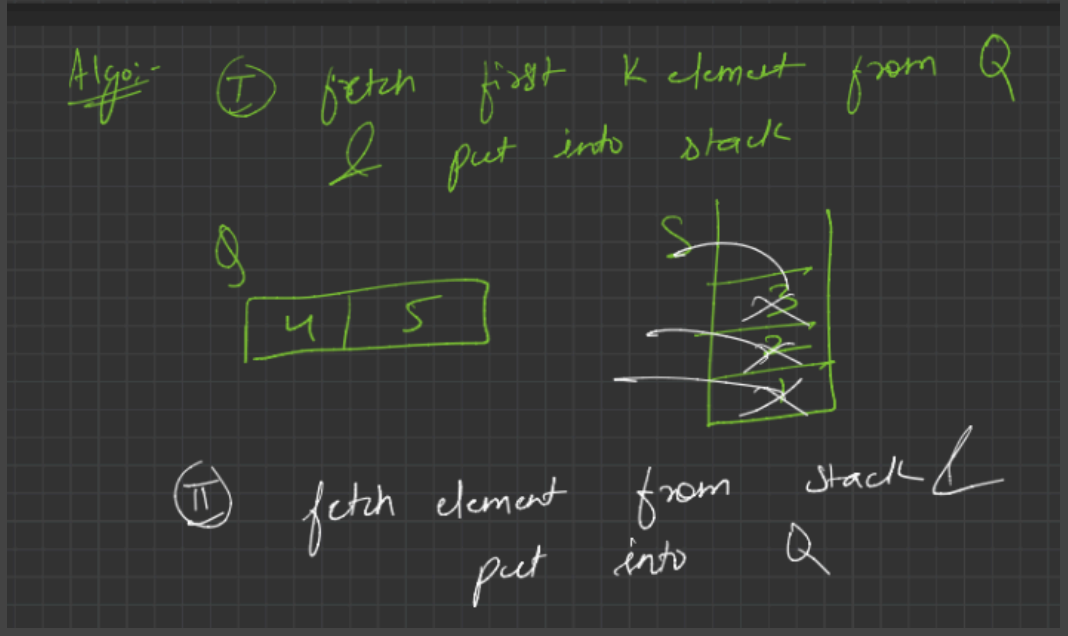


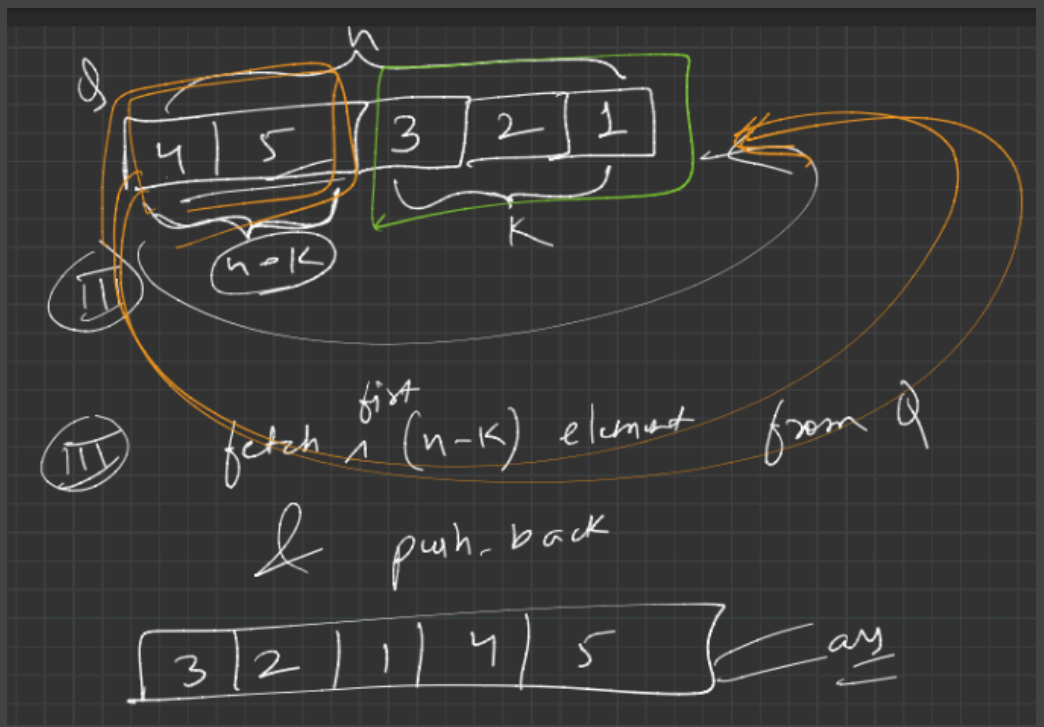



```
// Removal
for ( i -> <n )
{
    // Removal
    if ( dq.front() - i >= k)
        dq.pop_front();
    // addition
    if (arr[i] < 0)
        dq.push_back(i);
ans  dq.size()>0  ans -> dq.front
     size=0       0
```

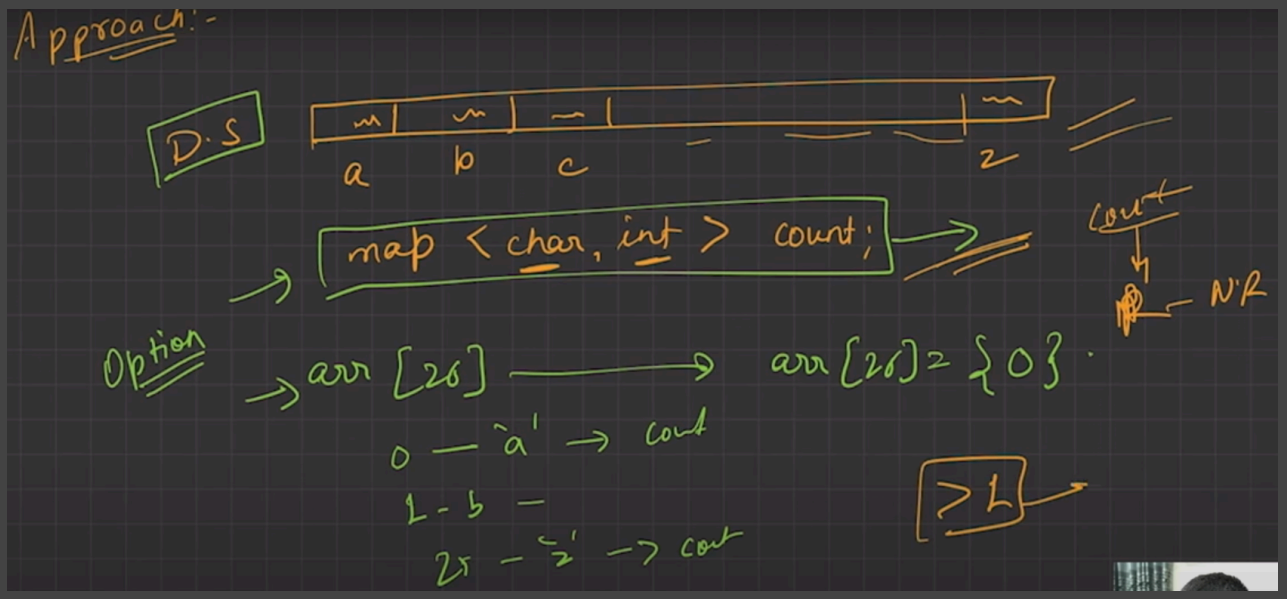
Reverse First K Elements In A Queue:

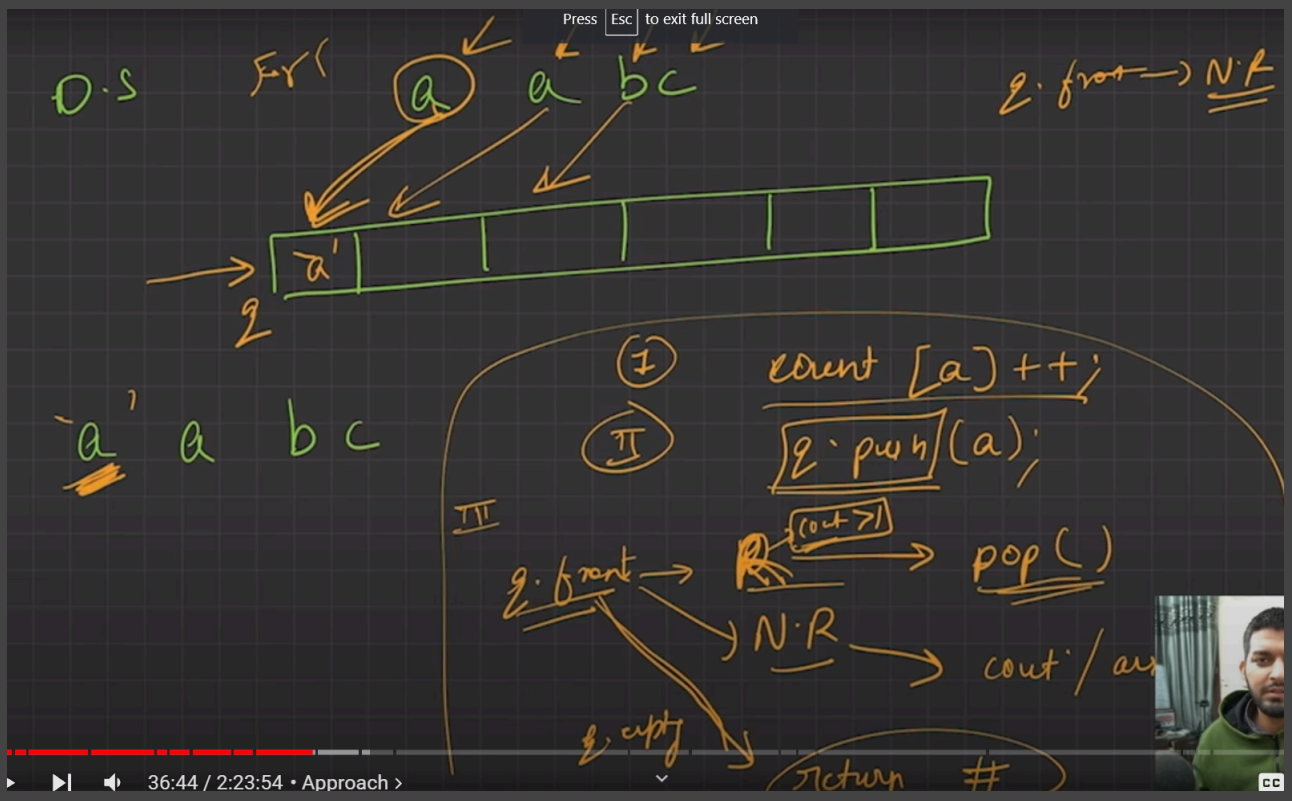




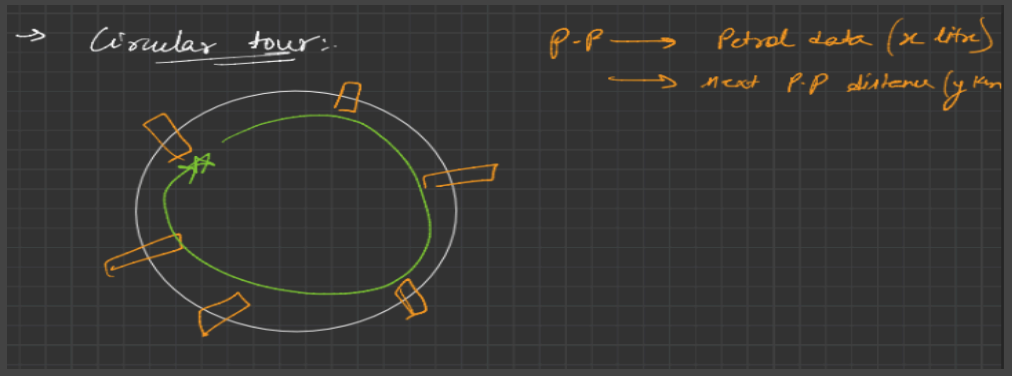


First Non-Repeating Character In A Stream:





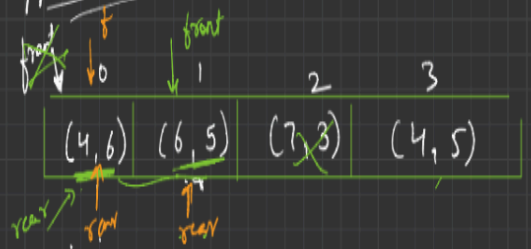
Circular Tour:



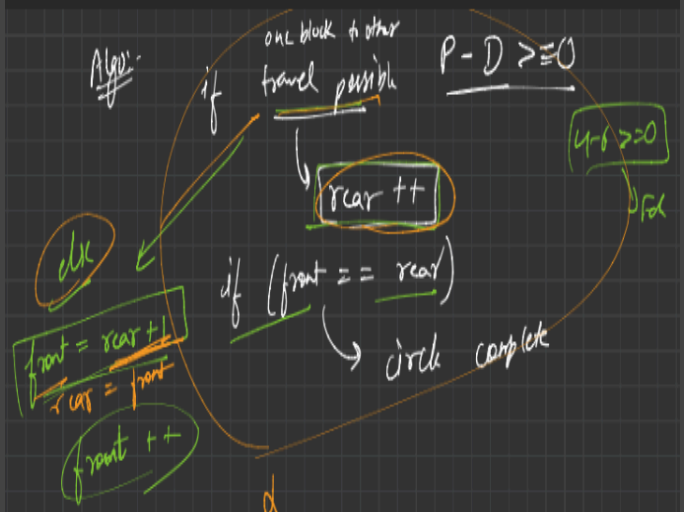
approach #1 :- B.F

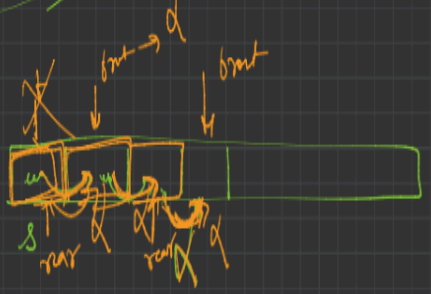
$\rightarrow O(n^2)$

approach #2 :-



Algo :-





balance >=0

Algo :- start block to next block

if (travel possible)

rear ++

else

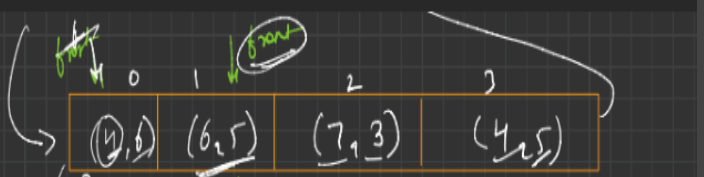
front = rear + 1

start = front

rear = front

B.C $\rightarrow$ if (front == rear)

$\rightarrow$ cycle complete



(1) P - D >= 0 False

4 - 6 = -2 < 0

front = rear + 1

(I) P - D >= 0 !

8 - 5 = 3 >= 0

balance = 4

balance = 1 + 7 - 8 - 3 = 5 >= 0 T

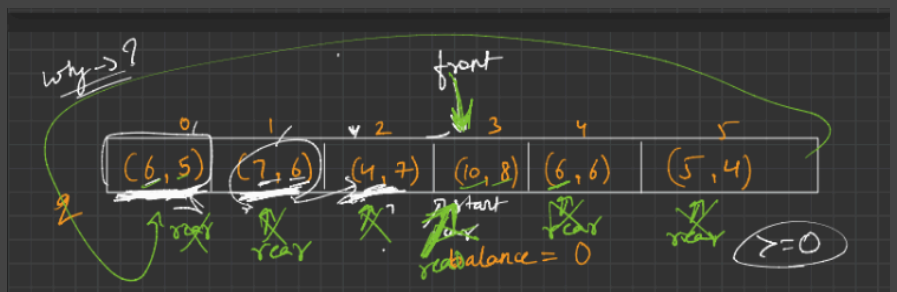
balance = 5 + 4 = 9 - 5

= 4 >= 0 T

balance = 4 + 4 - 6

= 8 - 6

= 2 >= 0 T



why -> front ++ ∝

front = rear + 1

balance = balance + P - D

= 0 + 6 - 5 = 1 >=0 -> TRUE

balance = 1 + 7 - 6

= 8 - 6 = 2 >=0 -> TRUE

balance = 2 + 4 - 7

= 6 - 7 = -1 >=0 -> false

worst case

Block -> 2 visit

at

single visit

balance = 0

balance = 0 + 10 - 8

= 2 >=0 TRUE

balance = 2 + 6 ≠ 6

2 (2) >=0 TRUE

= 2 + 5 - 4

= 7 - 4

= 3 >=0 -> TRUE

= 3 + 6 - 5

= 9 - 5 = 4 >=0 T

= 3 + 6 - 5

= 9 - 5 = 4 >=0 T

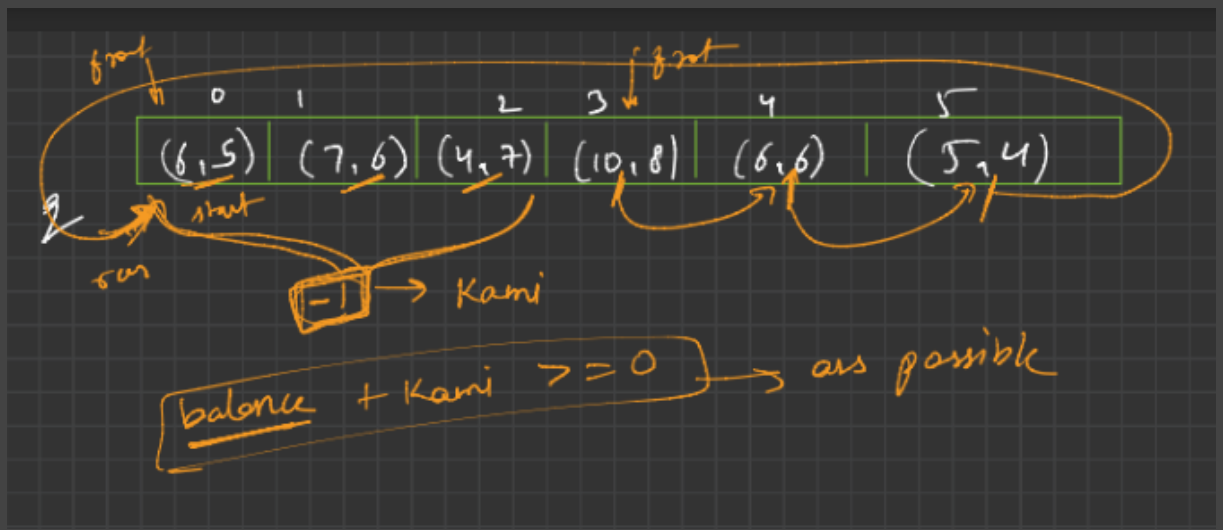
= 4 + 7 - 6

= 11 - 6 = 5 >=0 TRUE

= 5 + 4 - 7

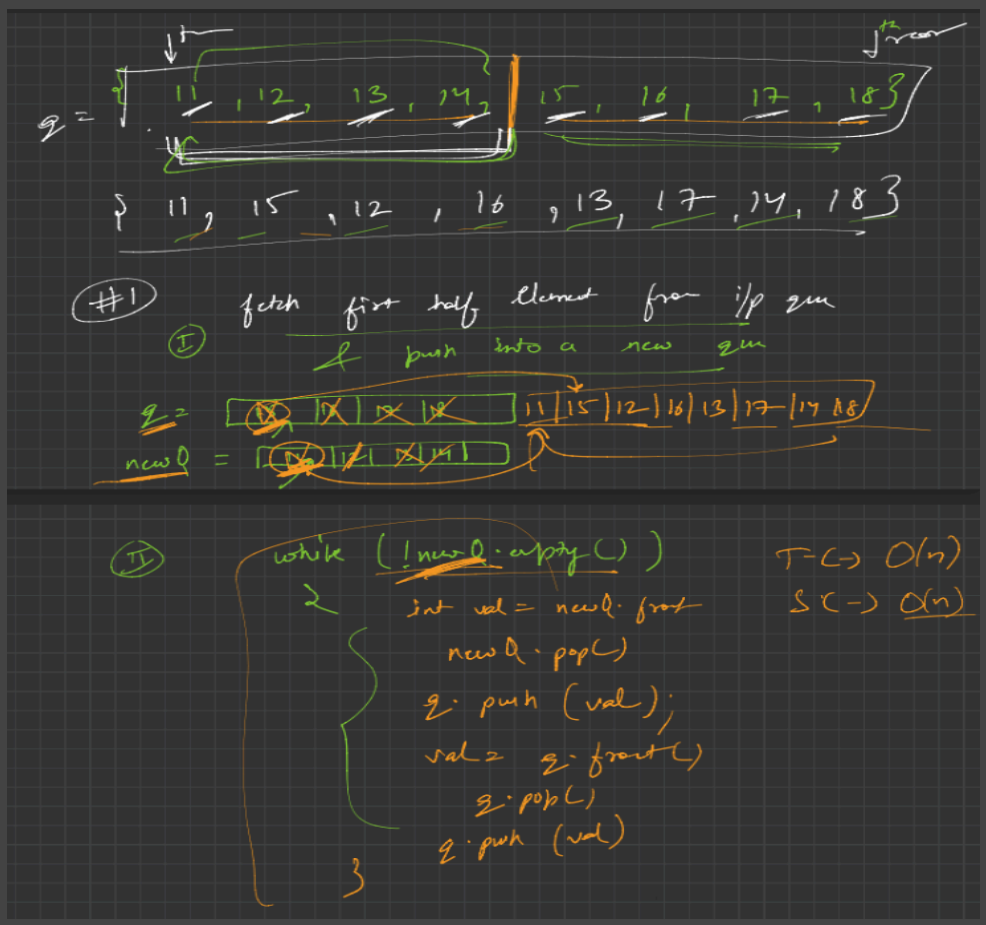
= 9 - 7 = 2 >=0

TRUE

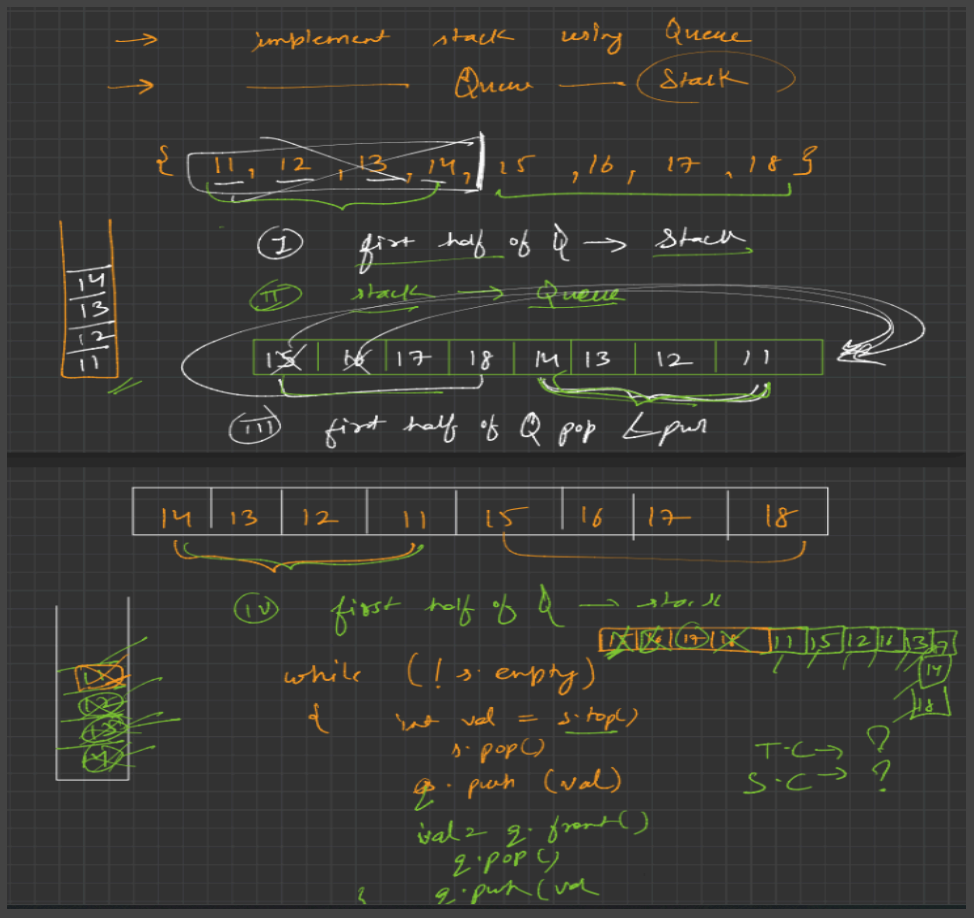


balance + Kami >=0 -> as possible

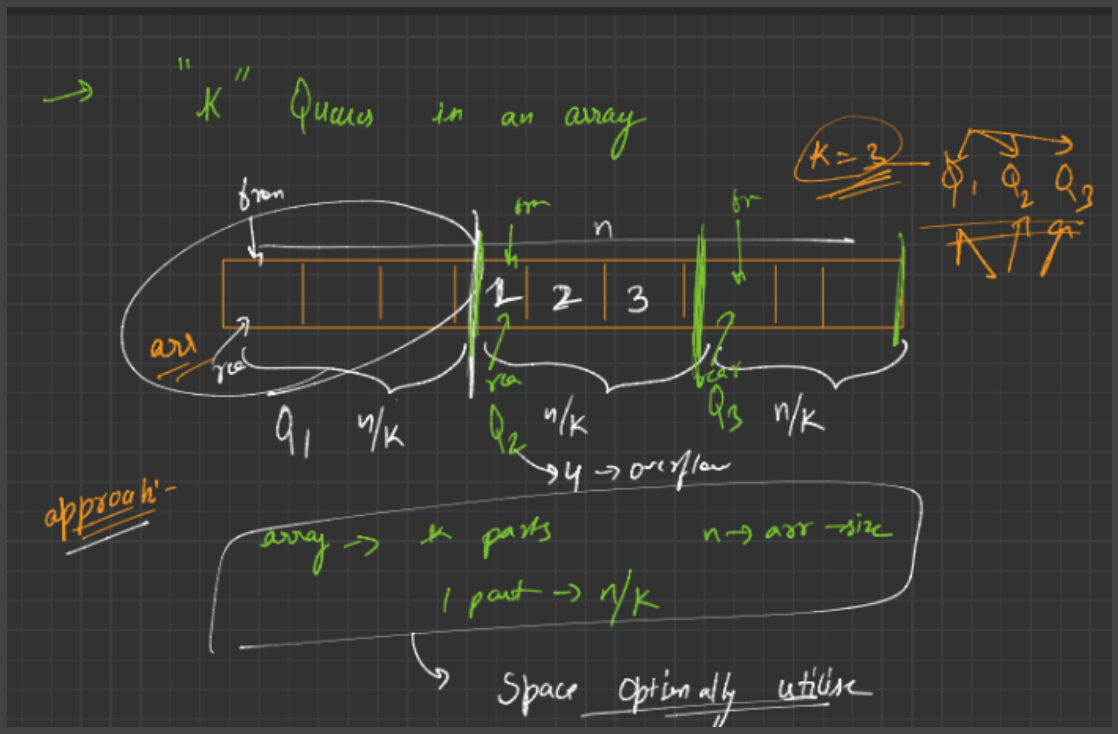
Interleave First and Second Halves Of A Queue:



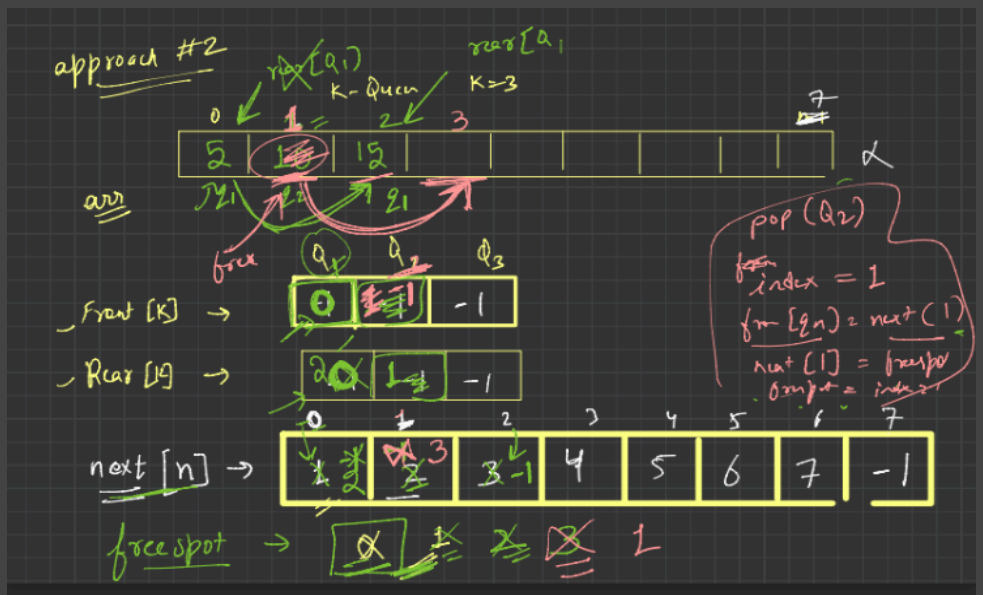
Interleave First and Second Halves Of A Queue Using Stack:

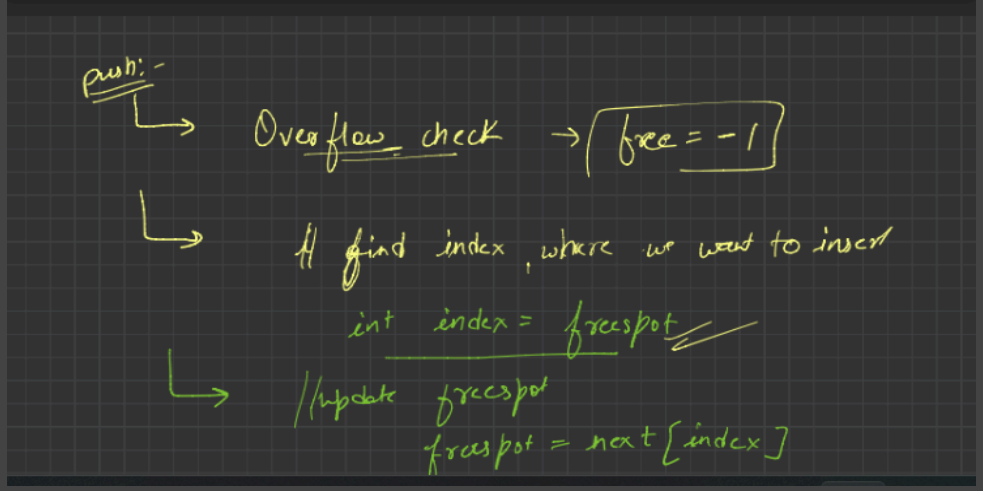


Insert K Queues In An Array Approach-I:

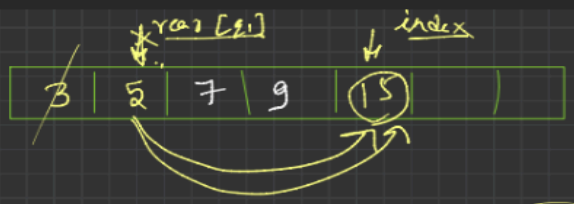


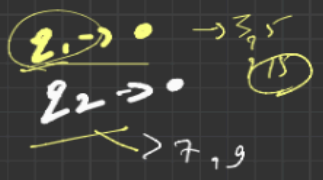
Insert K Queues In An Array Approach-II:





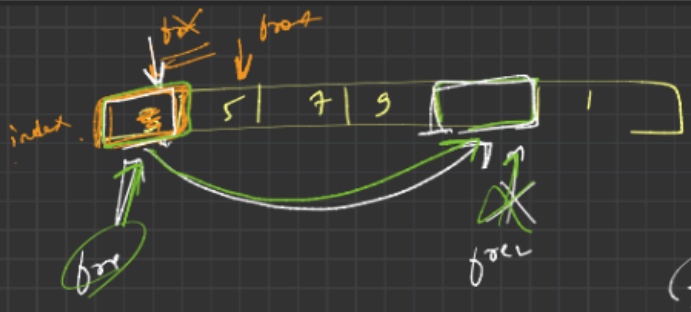
```
↳ // if first element
   if (front[qn] == -1)
       front[qn] = index;
   else
   {
       next[rear[qn]] = index;
   }
```





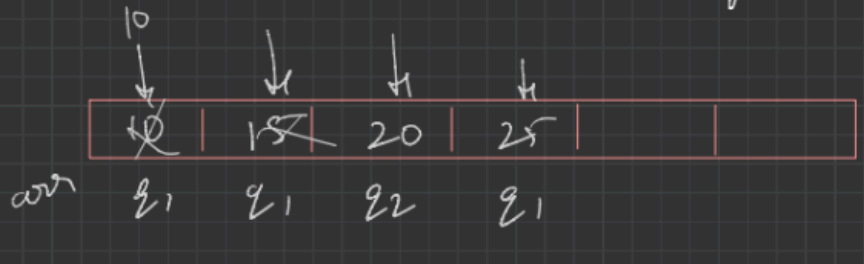
next[rear[qn]] = index;

```
↳ next[index] = -1;
   // point rear to index
↳ rear[qn] = index;
↳ // push element
   arr[index] = x;
```

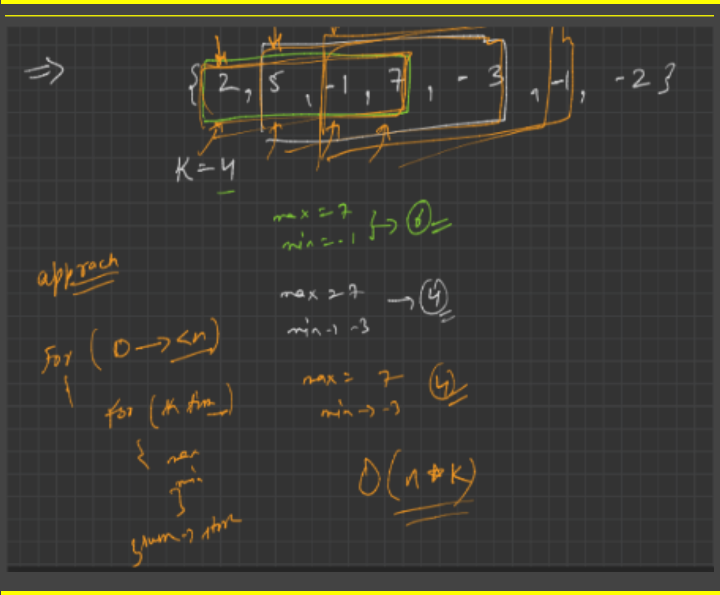


next[index] = free

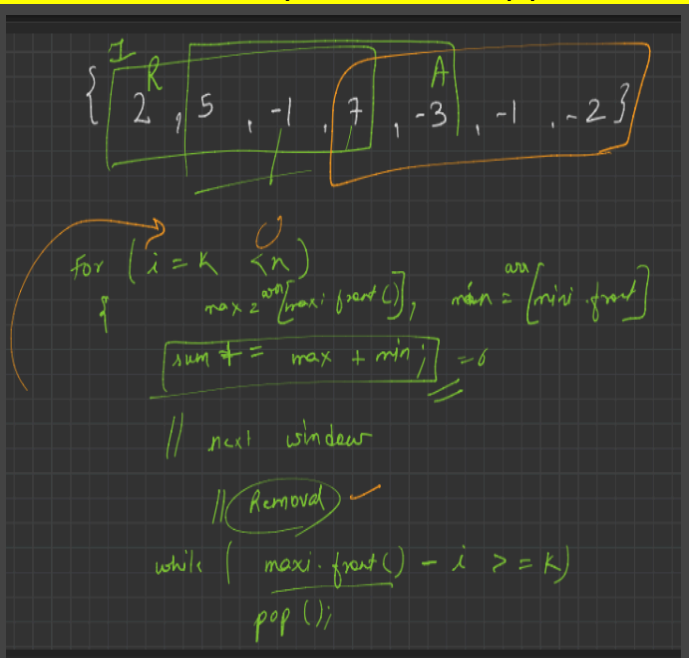
free = index

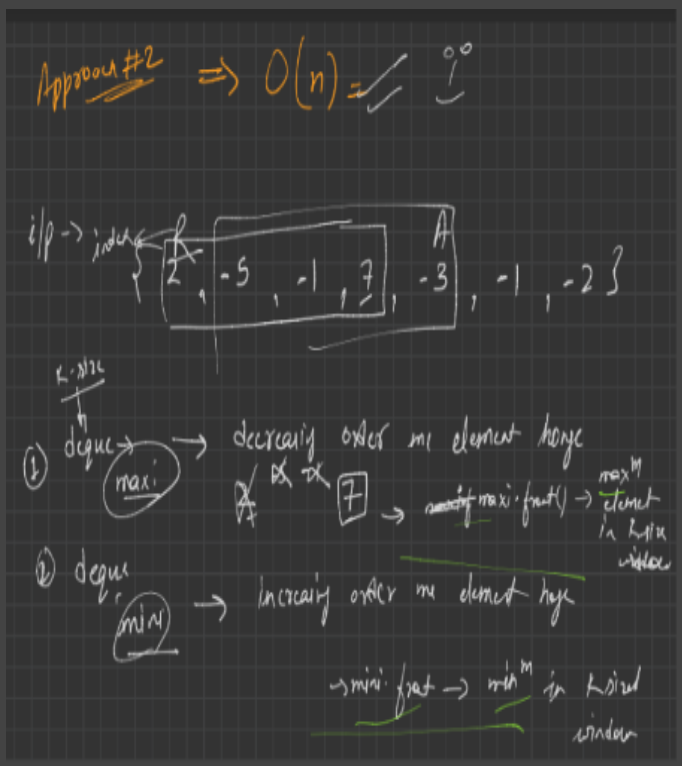


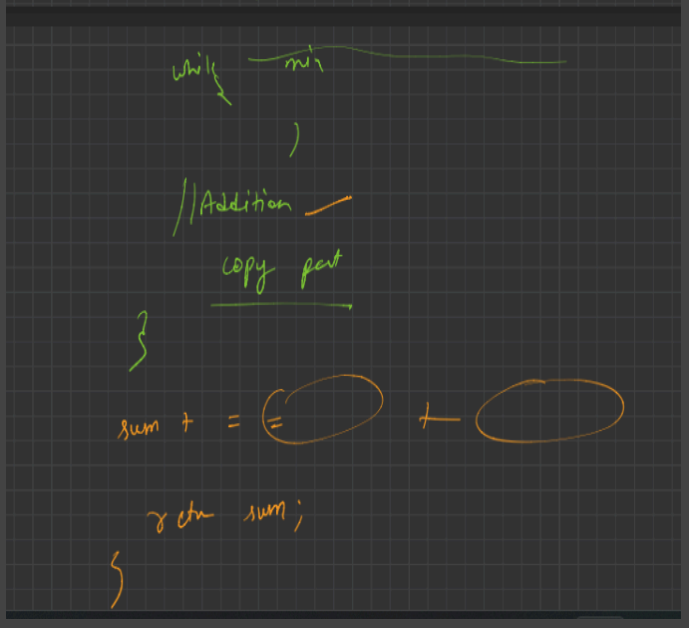
Sum Of Minimum And Maximum Elements Of All Subarrays Of Size K Approach-I:

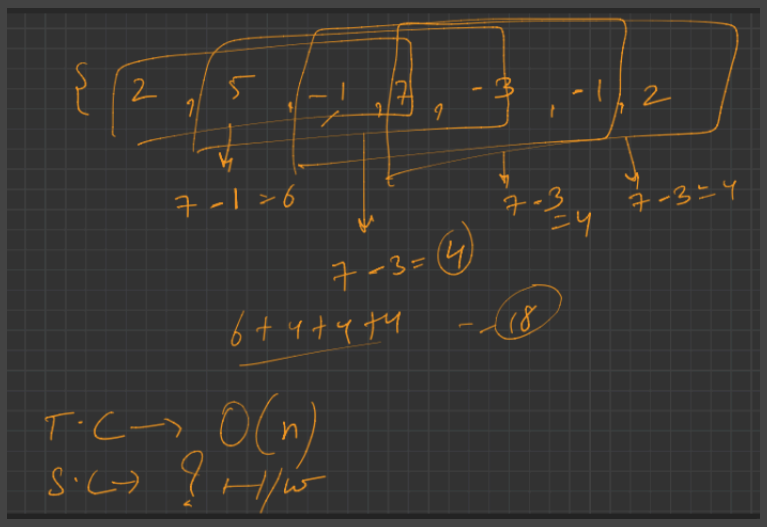


Sum Of Minimum And Maximum Elements Of All Subarrays Of Size K Approach-II:









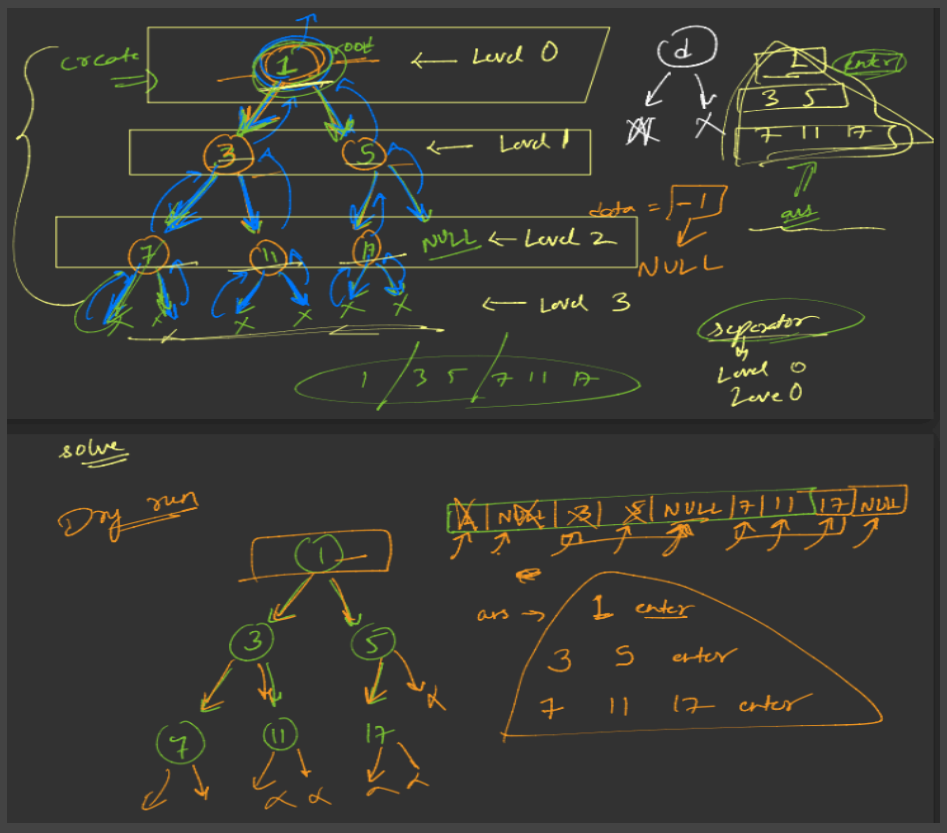
1. For the first k elements: The code enters the first for loop and processes the first k elements of the array. Each element is added to the deques maxi and mini after removing elements from the back of the deques that are not needed. This process takes constant time for each element, so for k elements, it takes O(k) time.
2. For the remaining elements: The code enters the second for loop and processes the remaining n-k elements of the array. For each element, it first removes elements that are out of the current window from the front of the deques. Then it adds the current element to the deques after removing elements from the back of the deques that are not needed. This process also takes constant time for each element, so for n-k elements, it takes O(n-k) time.

Since the code processes k elements in the first loop and n-k elements in the second loop, the total time complexity is O(k) + O(n-k) = O(n), where n is the total number of elements in the array. This is why we say each element in the array is processed only once in a single pass. The operations inside the loops are all constant time operations, so they do not change the overall time complexity.

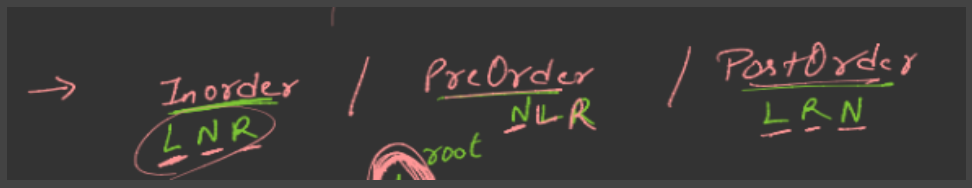
Binary Tree

Basic Terms Of Binary Tree:

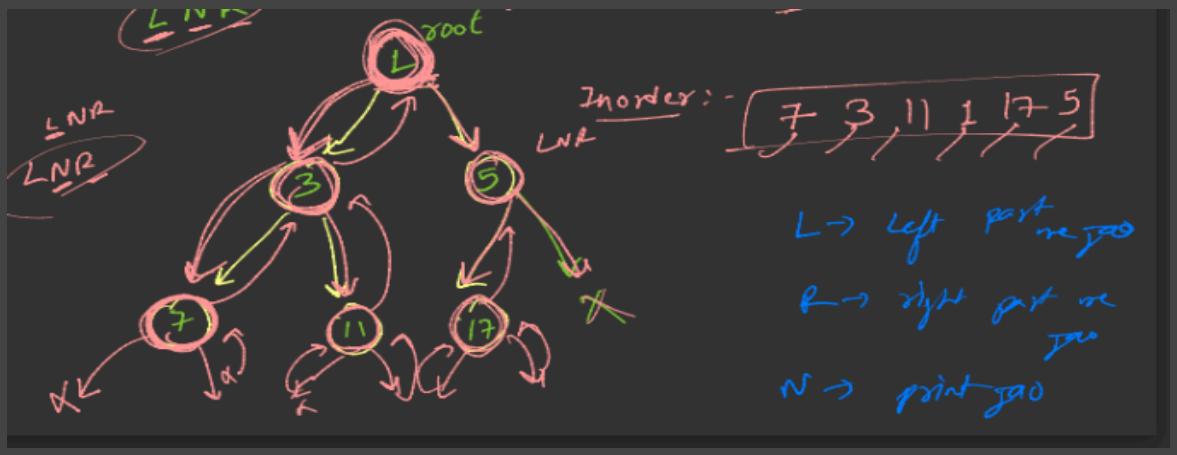
Creation Of Binary Tree & Level Order Traversal(LOT):



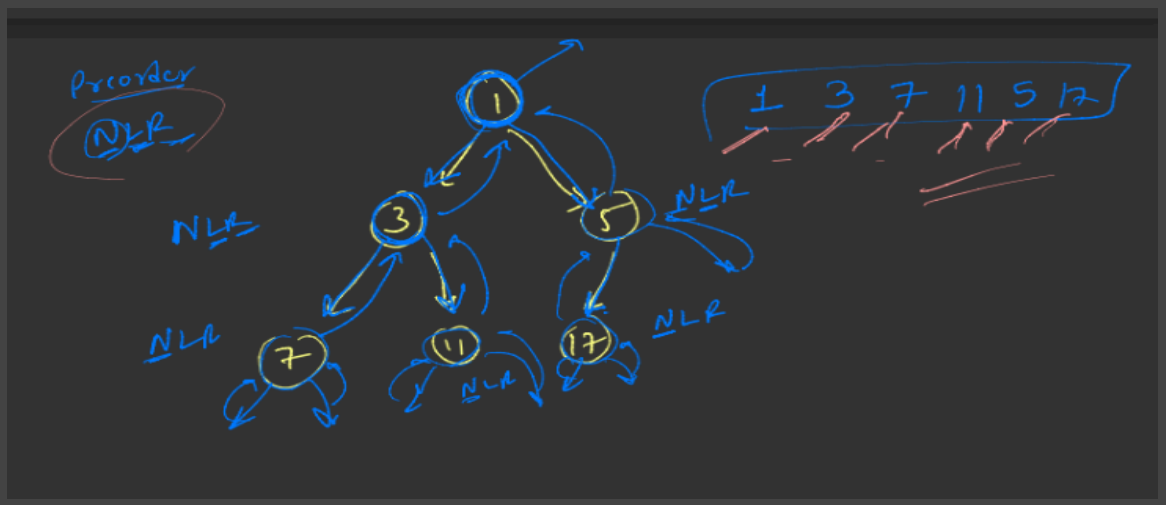
Reverse Level Order Traversal:



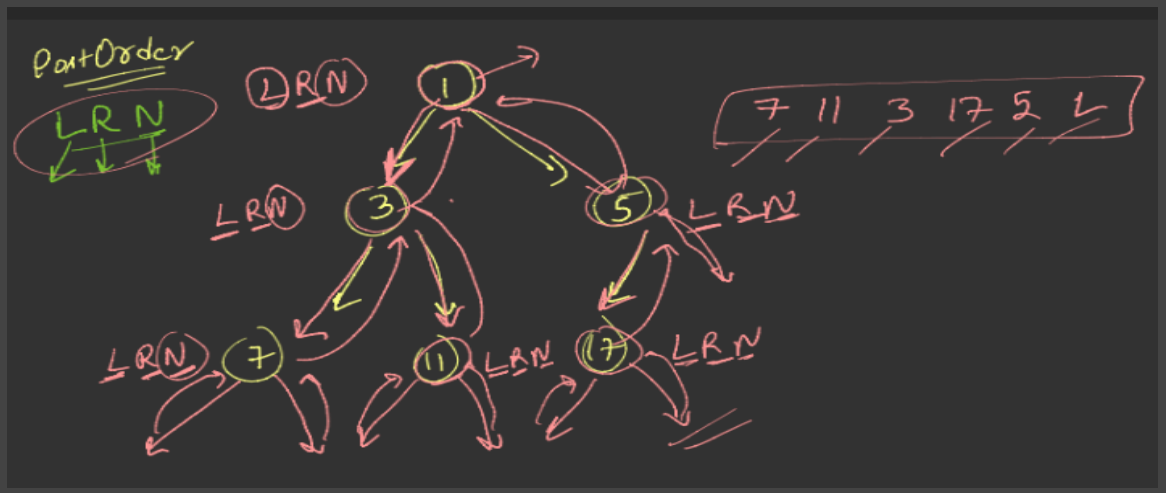
Inorder Traversal:



Preorder Traversal:



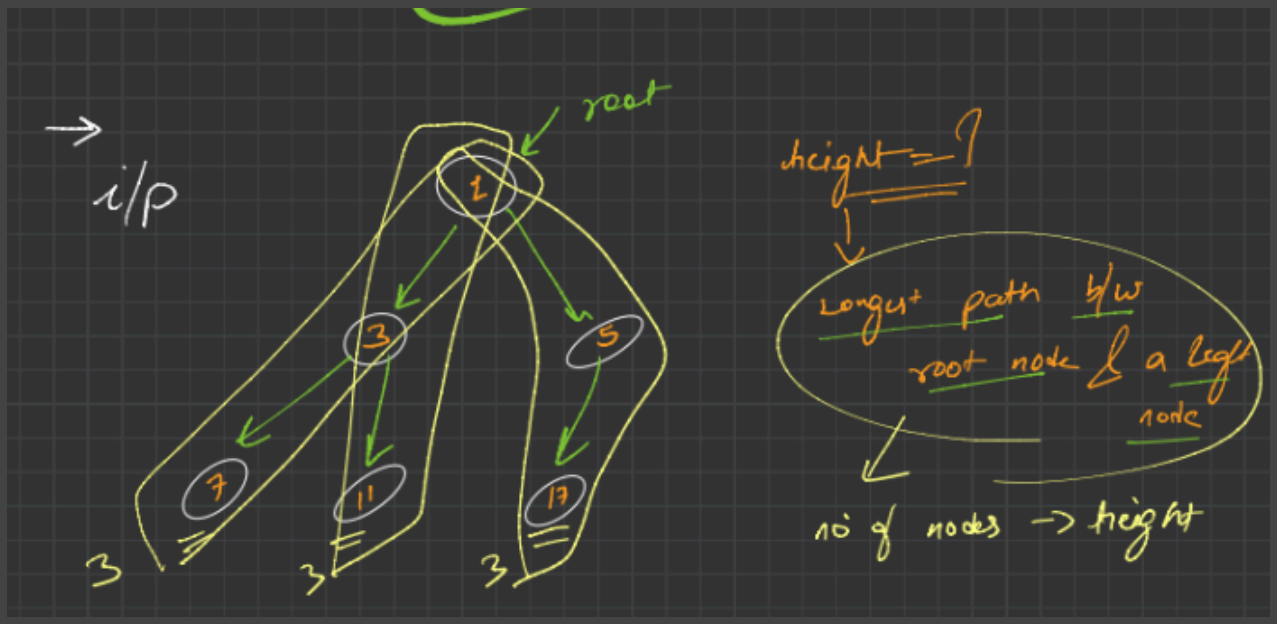
Postorder Traversal:

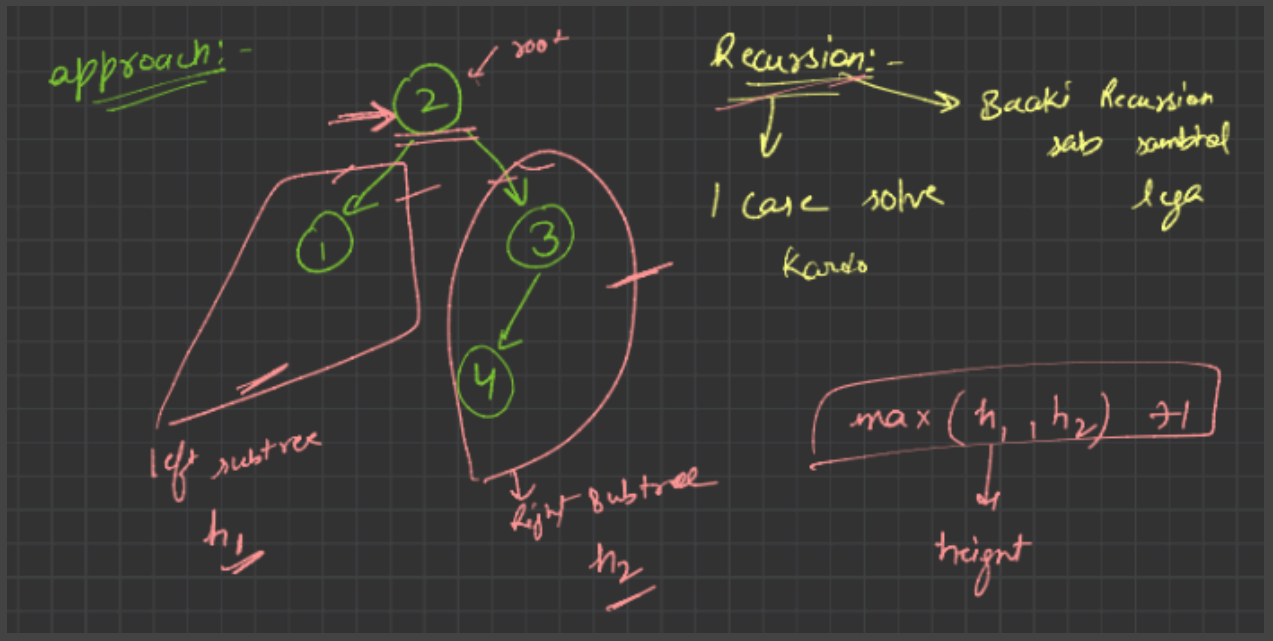


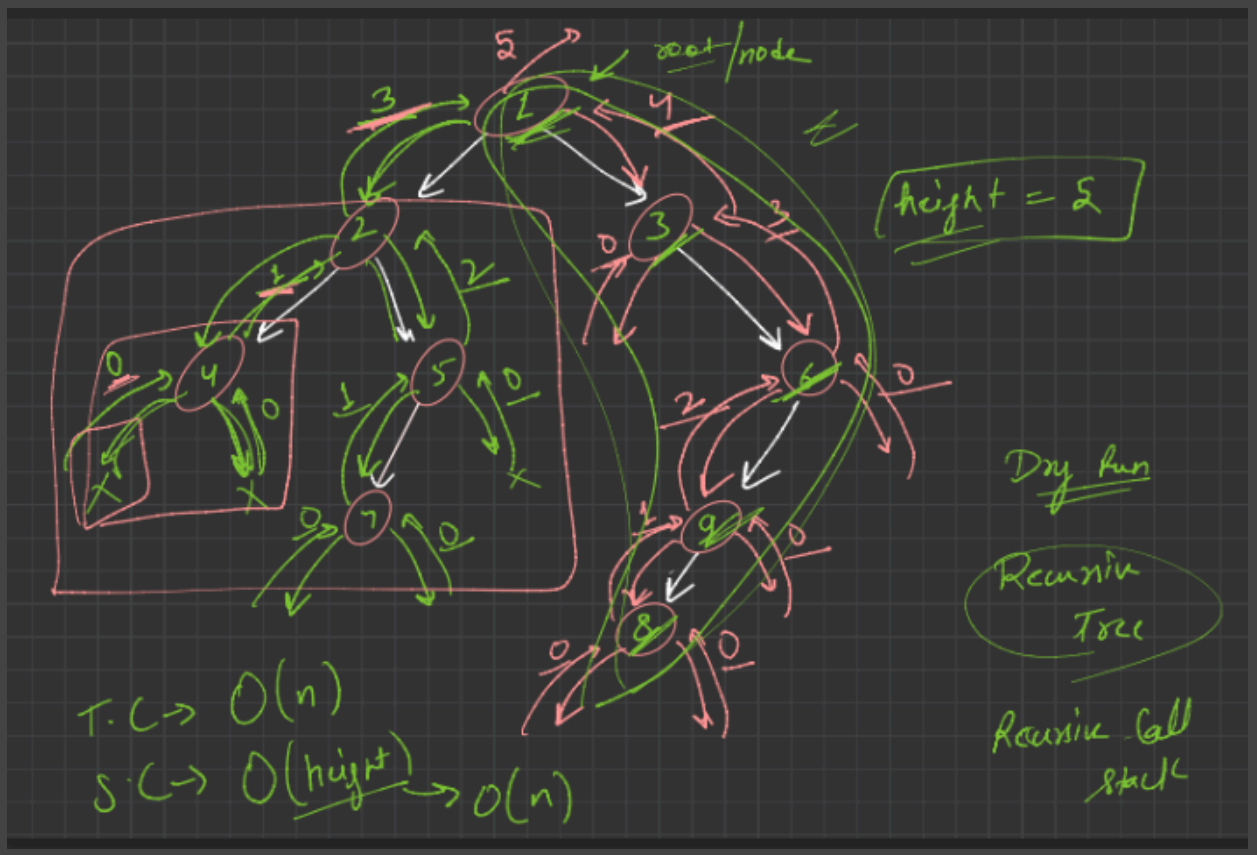
Build Binary Tree From Level Order:

Count Leaf Nodes:

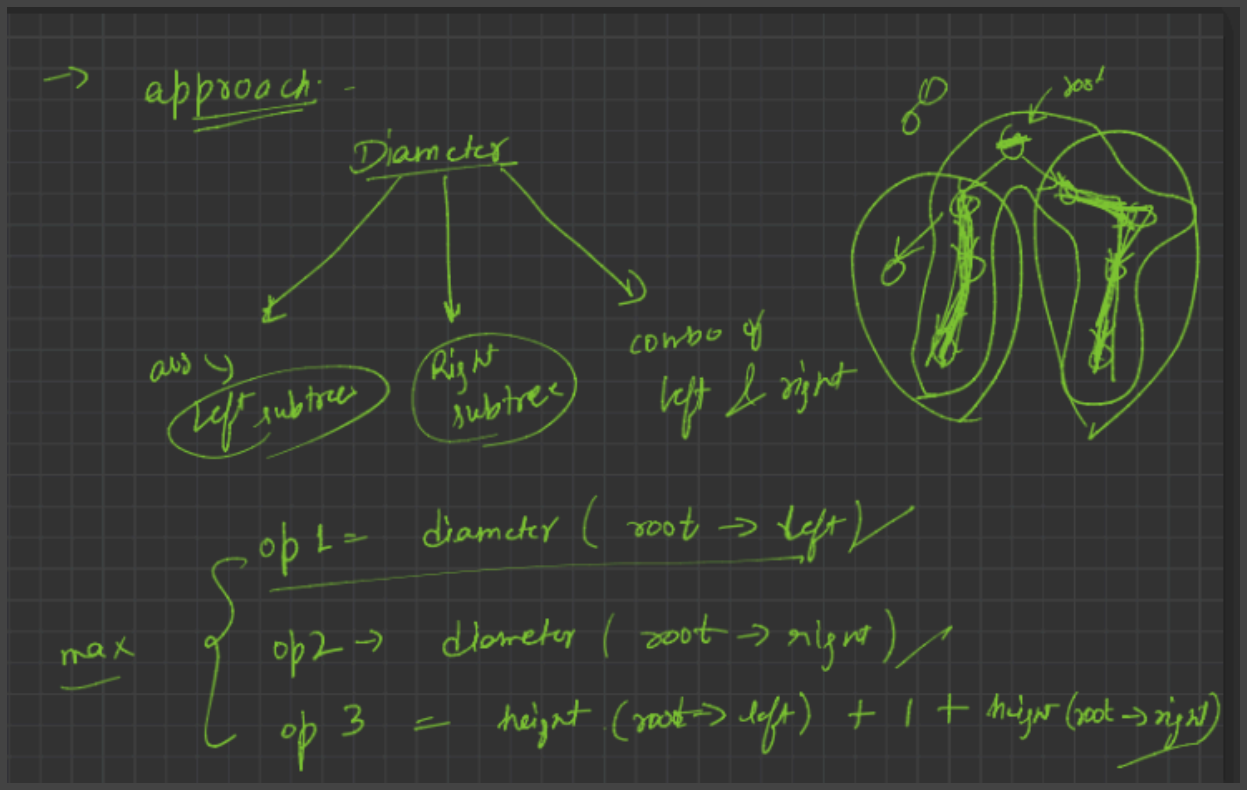
Height Of A Binary Tree:



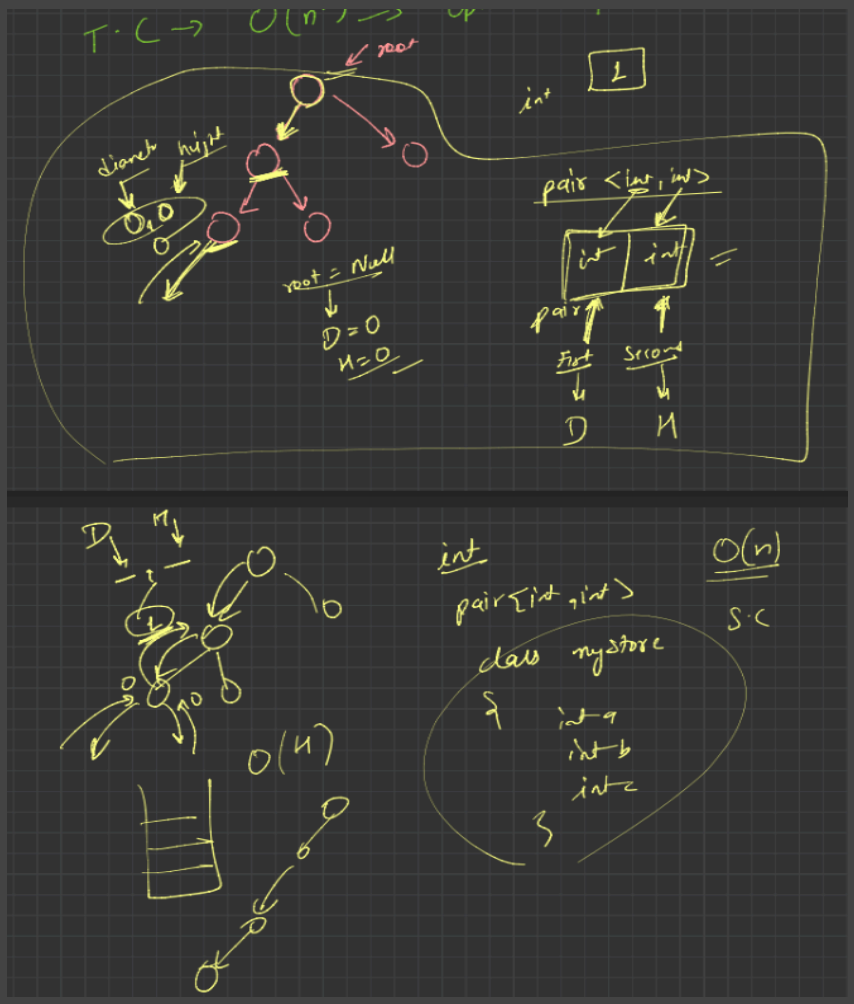




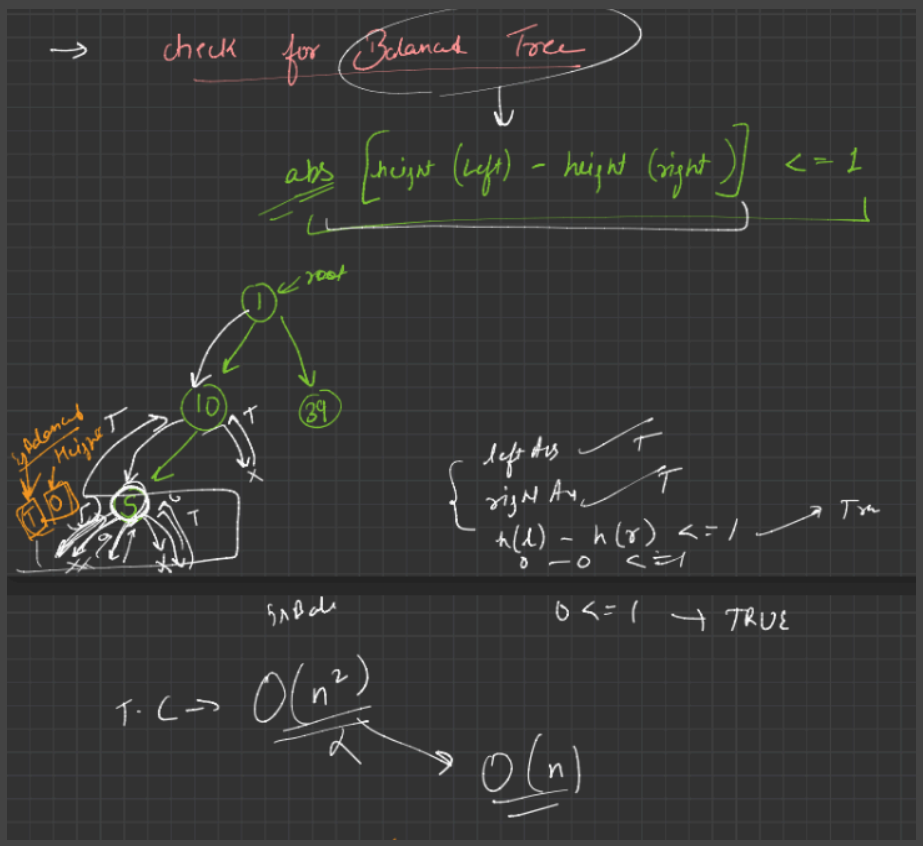
Diameter Of Binary Tree Approach-I:



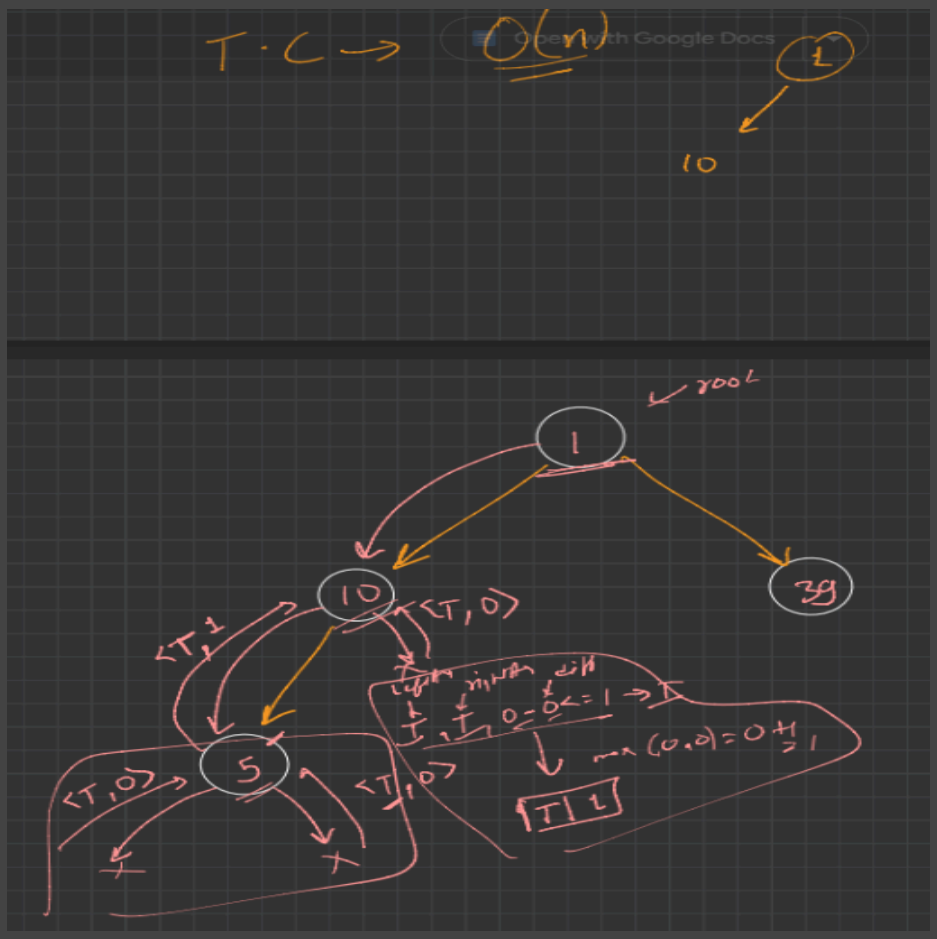
Diameter Of Binary Tree Approach-II:



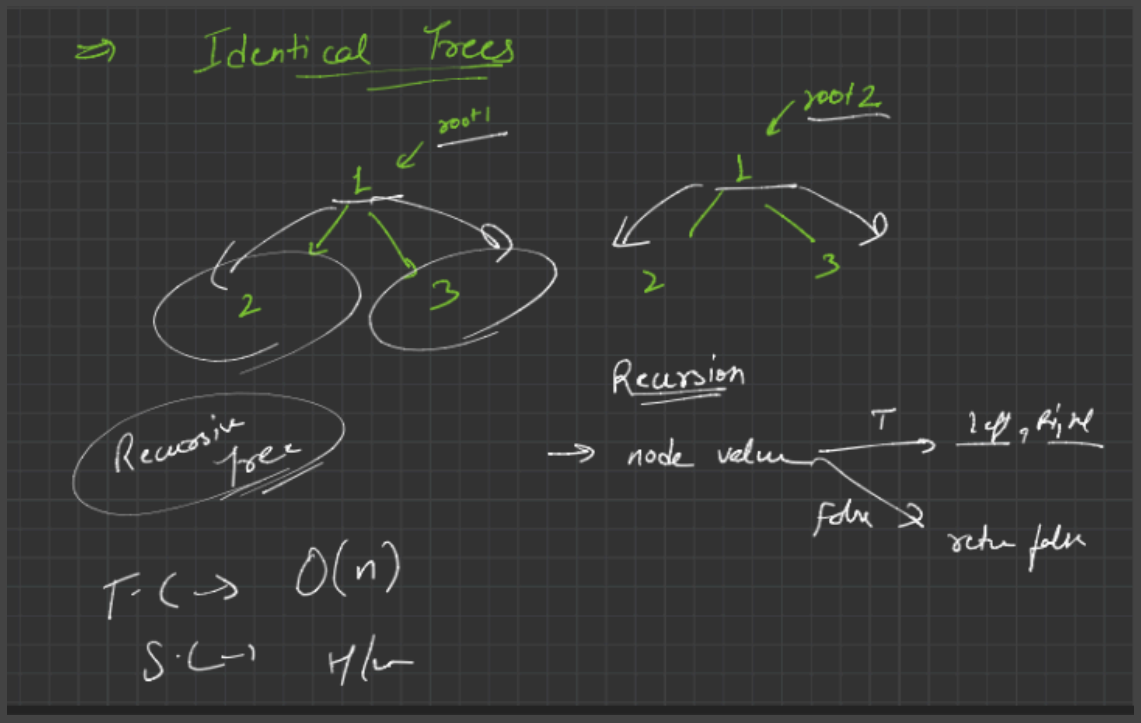
Check If The Tree Is Balanced Or Not Approach-I:



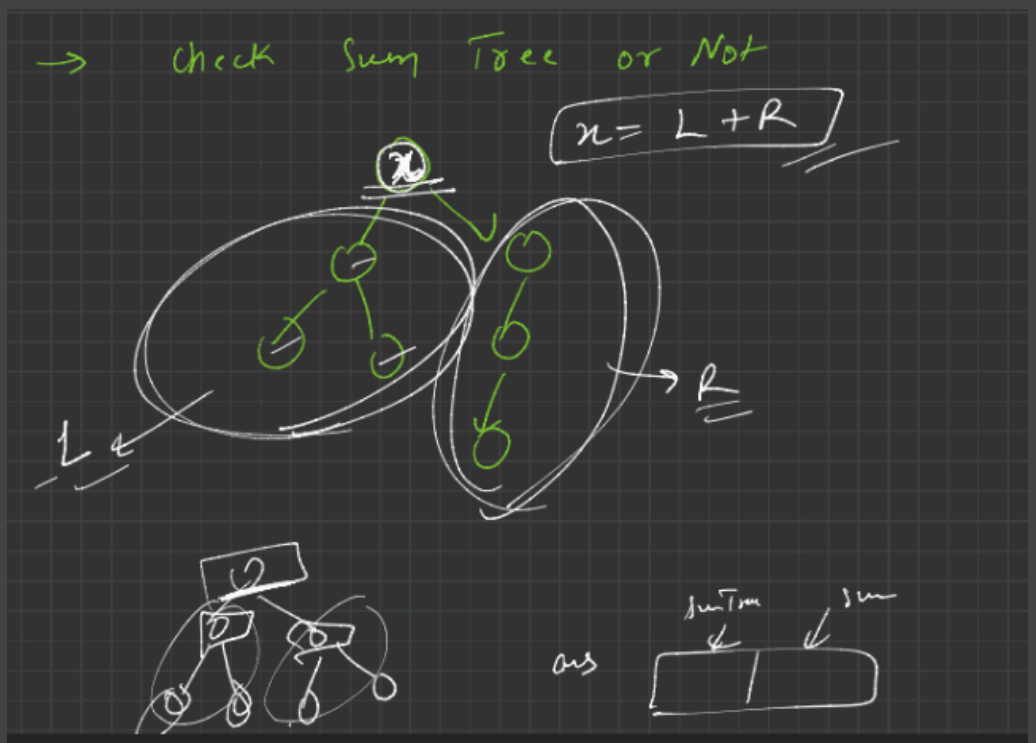
Check If The Tree Is Balanced Or Not Approach-II:

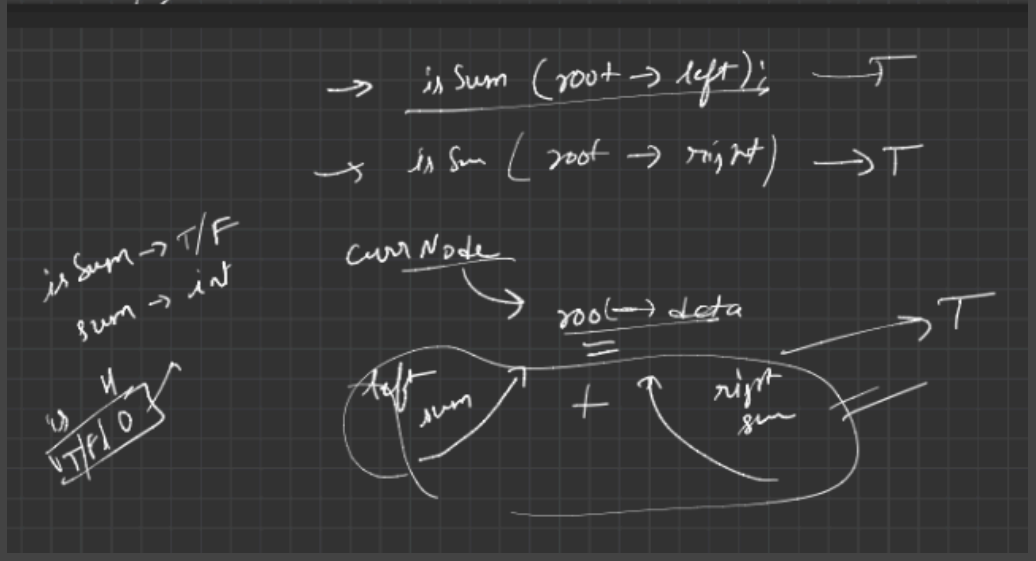


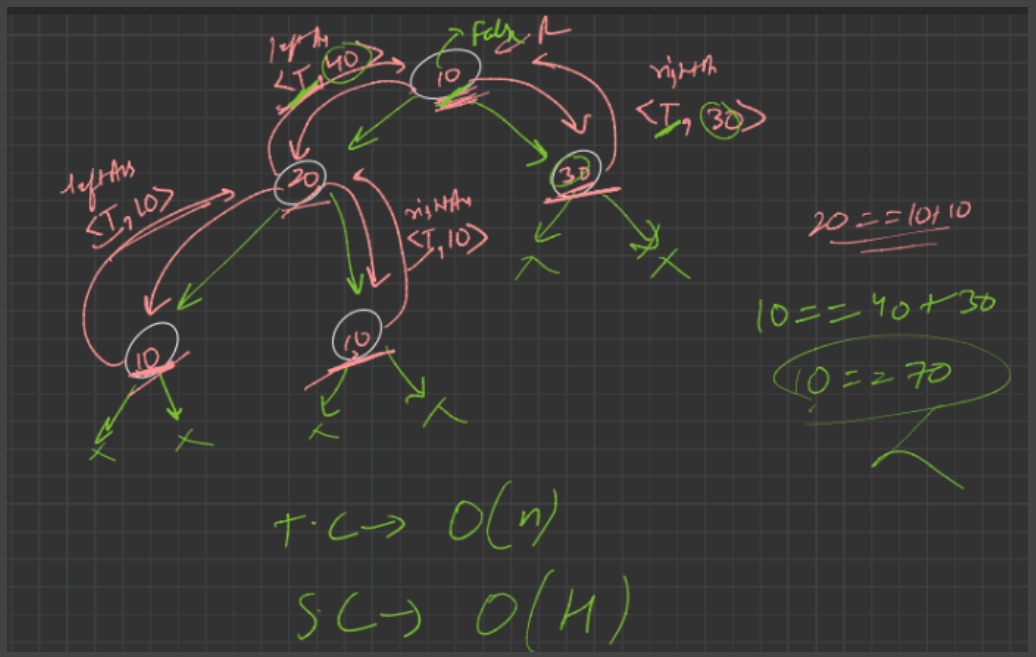
Check Whether The Two Trees Are Identical Or Not:



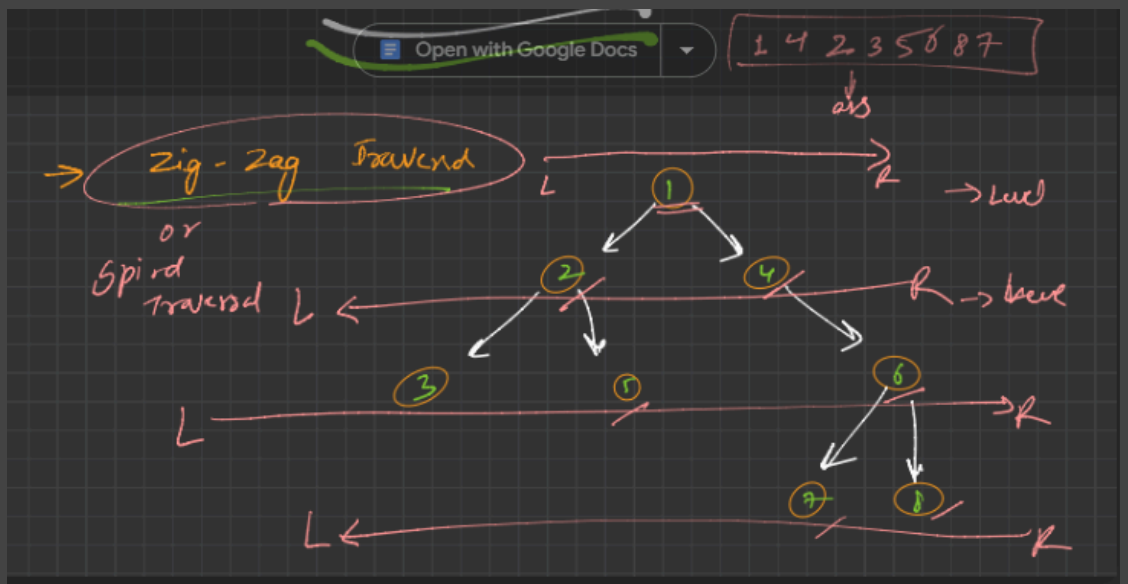
Sum Tree:

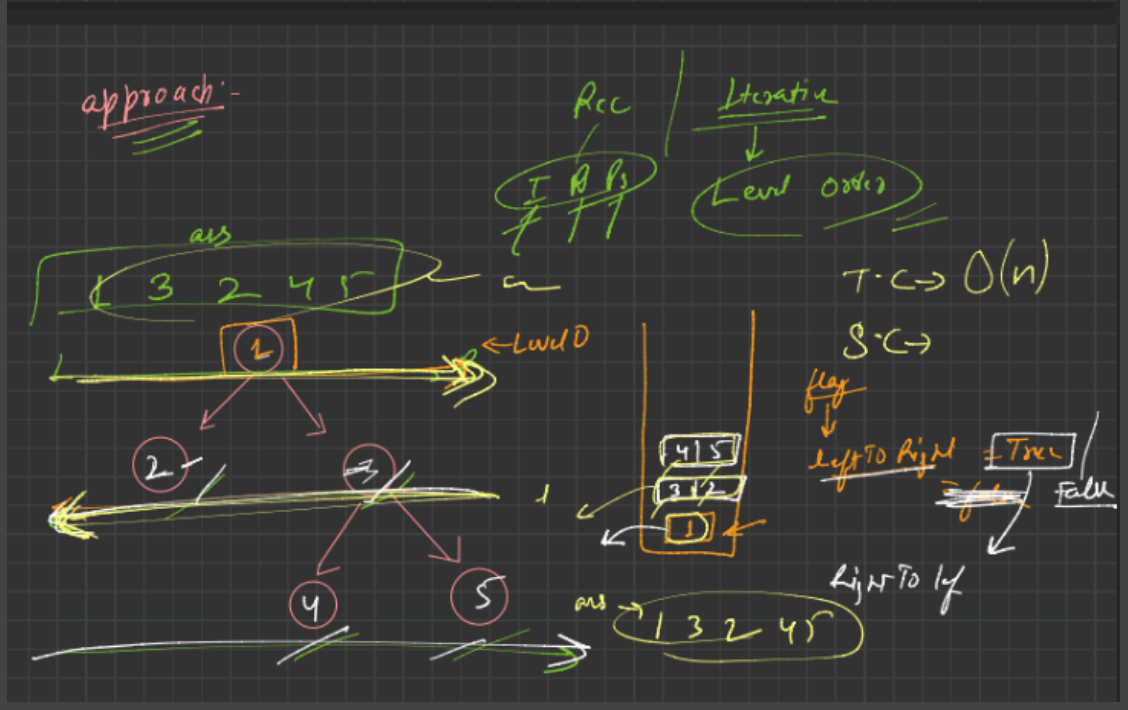

Check Sum Tree or Not
x = L + R
L
R
sumTree
sum
or

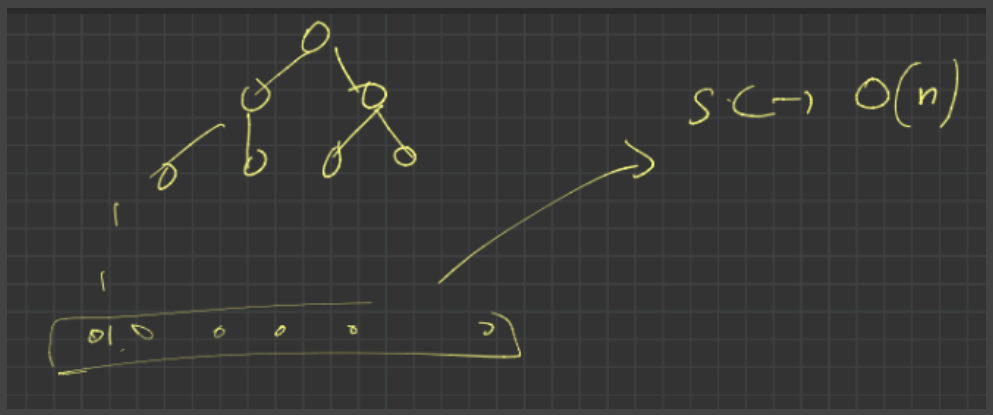

isSum (root → left);
isSum (root → right)
T
isSum → T/F
sum → int
currNode
root → data
left sum
+
right sum


leftNN
rightNN
20 == 10 + 10
10 == 40 + 30
10 == 70
T.C → O(n)
S.C → O(H)

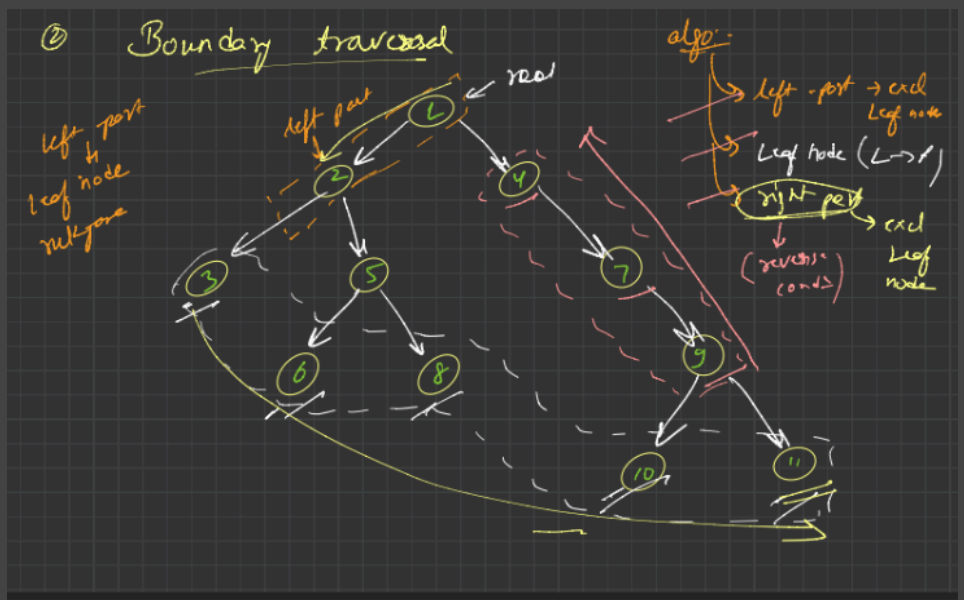
Zigzag Traversal:

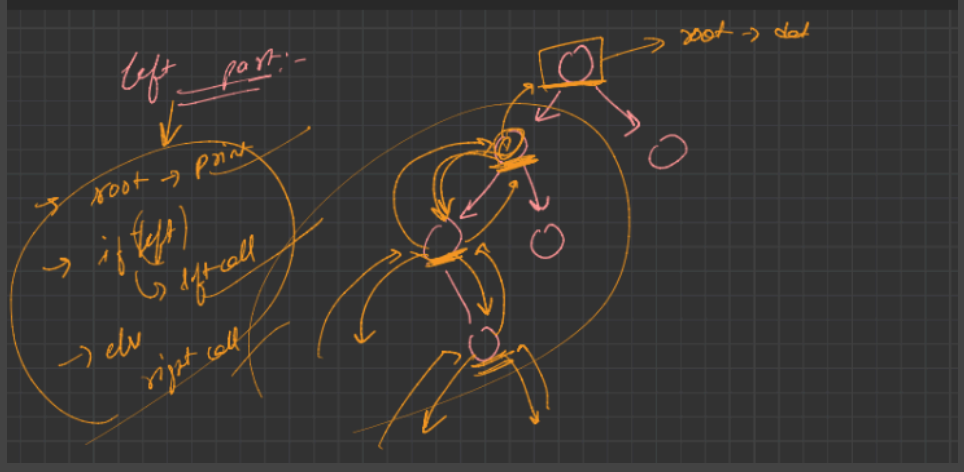


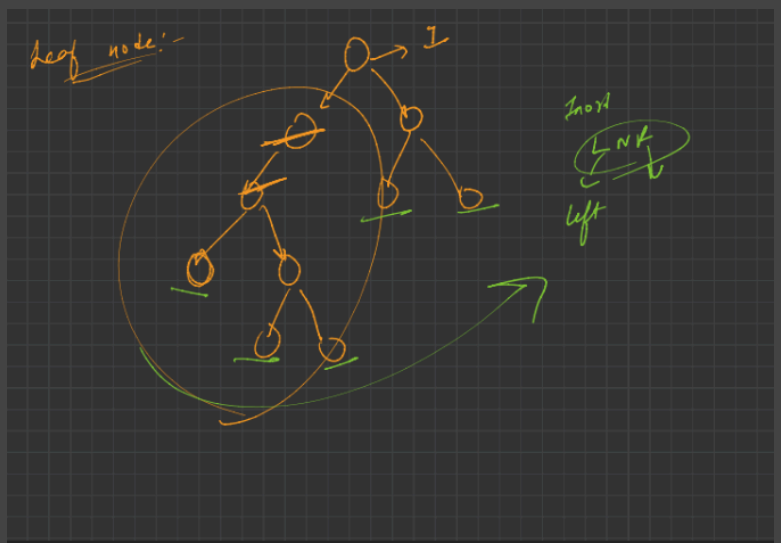


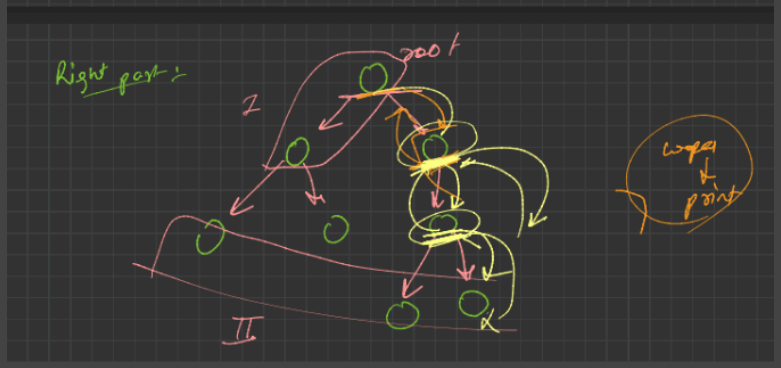


Boundary Traversal:

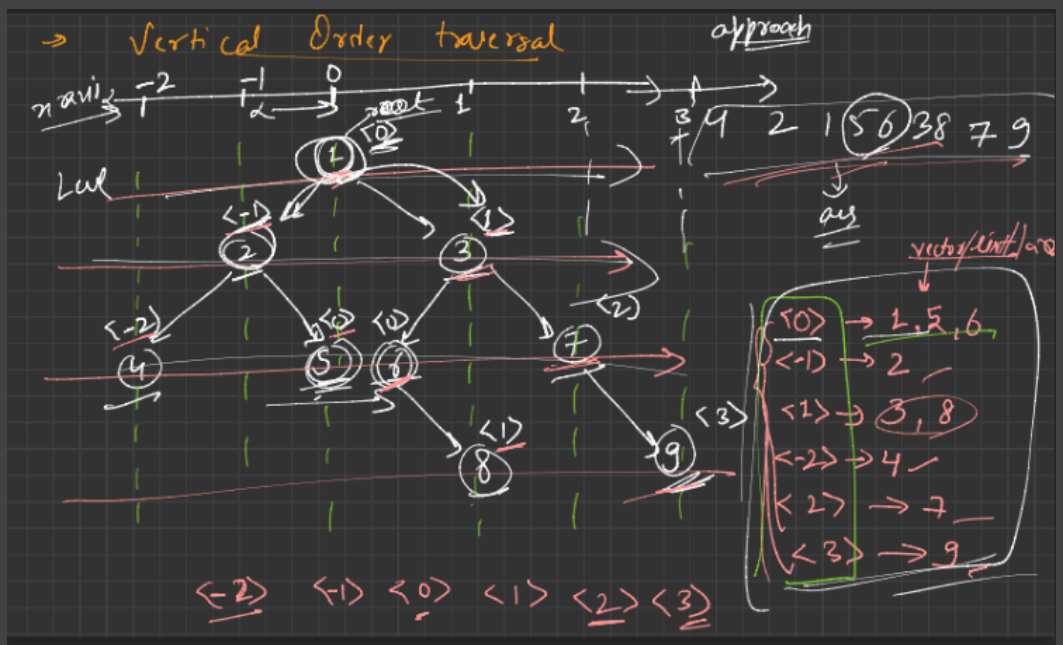


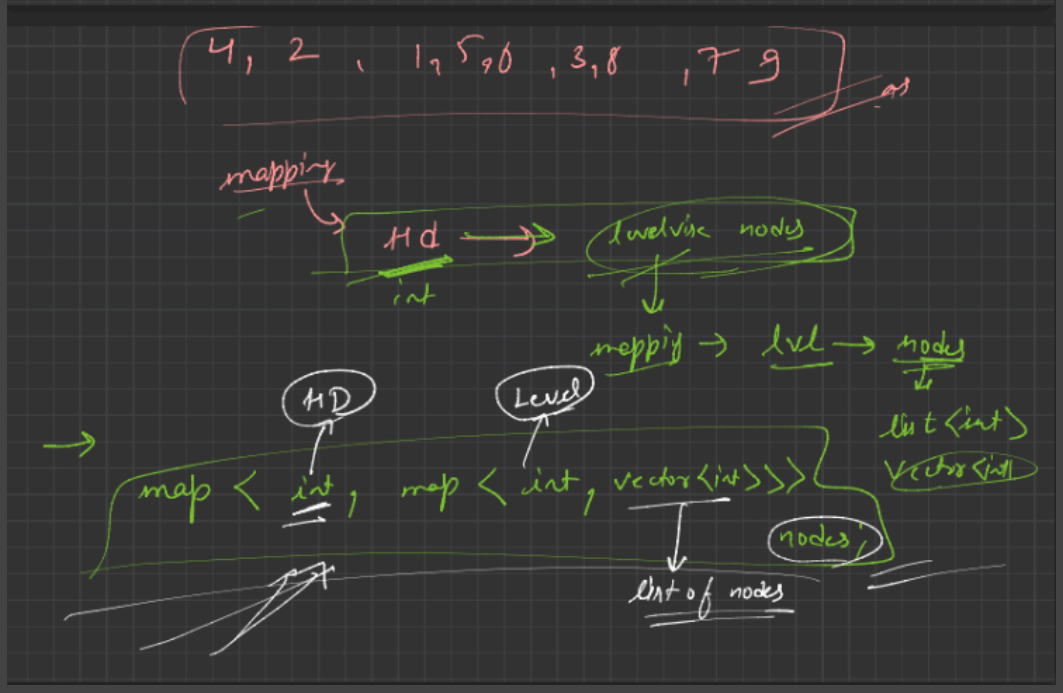


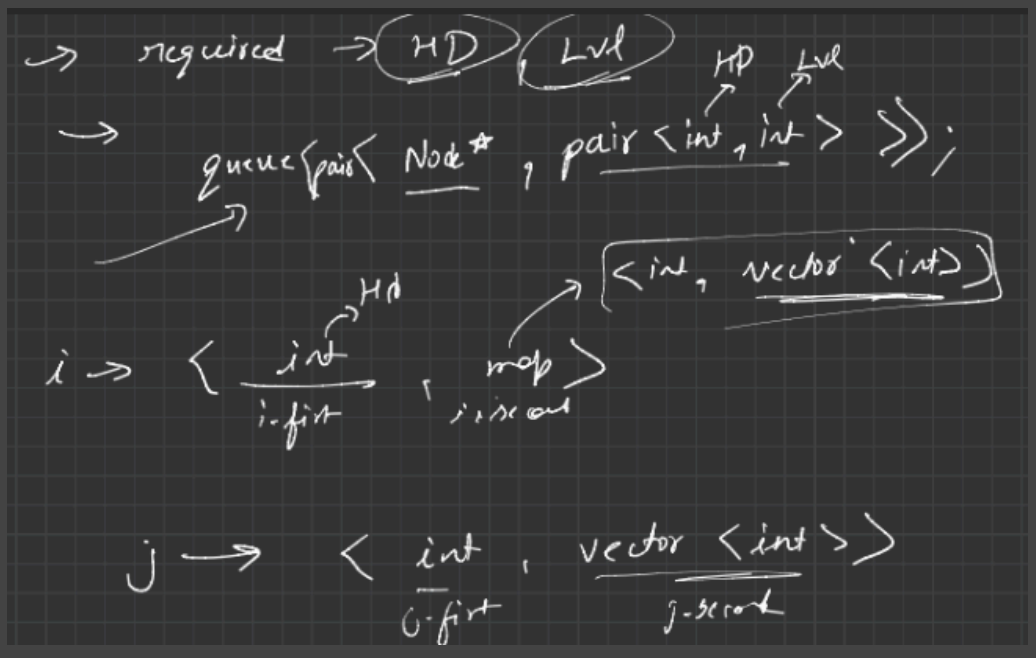




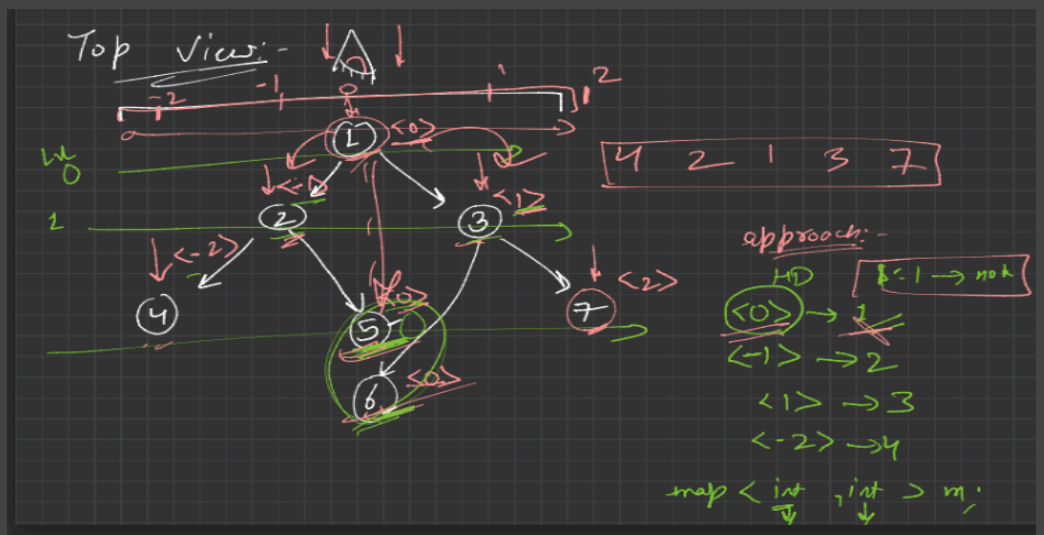
Vertical Traversal:

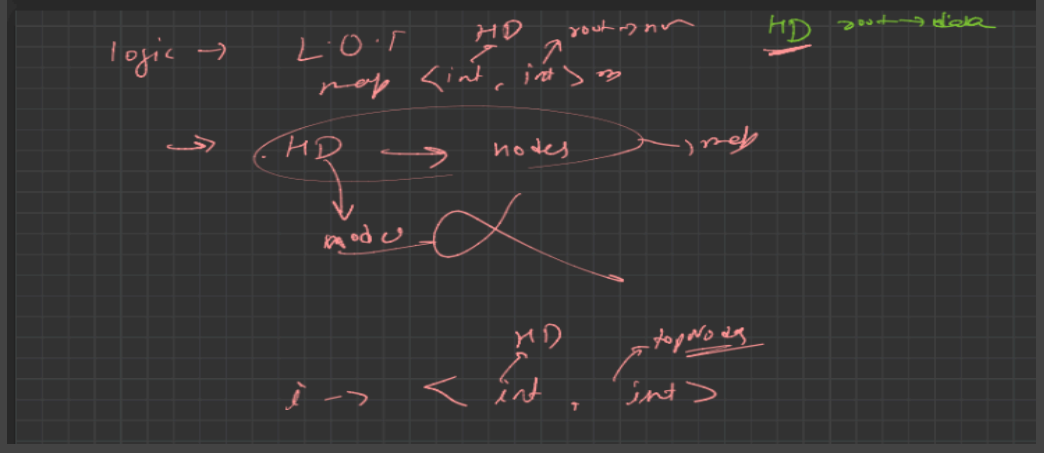




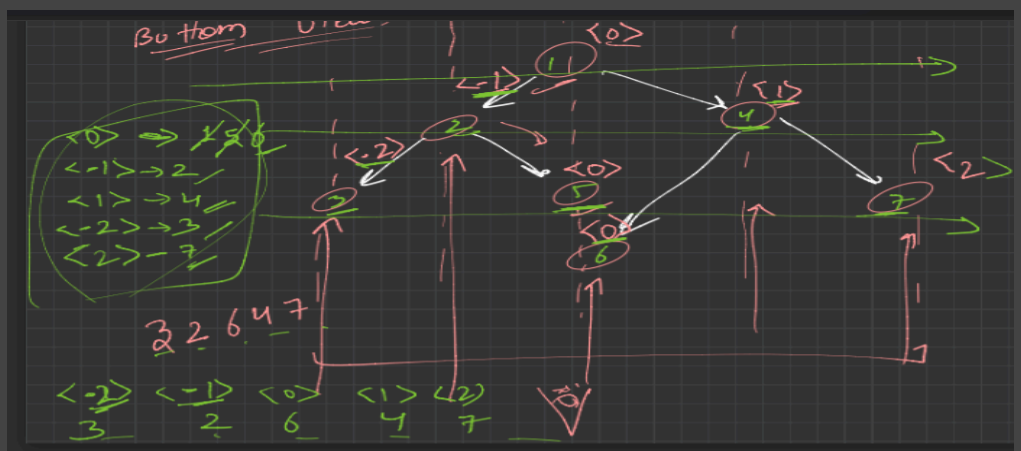


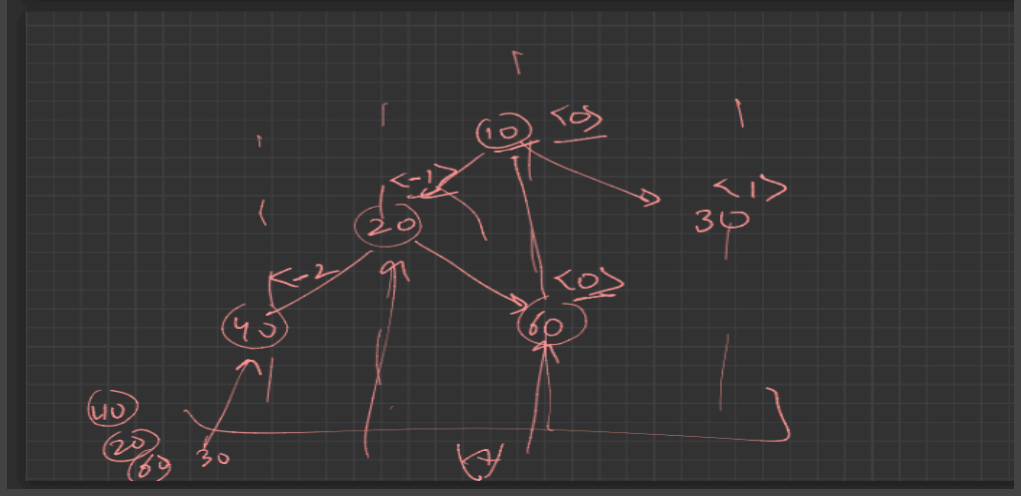
Top View:



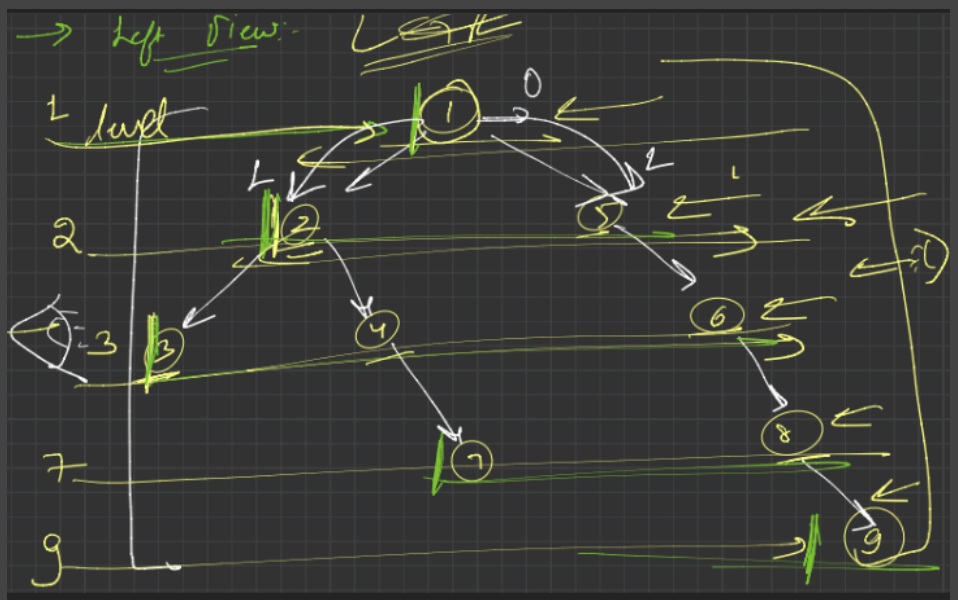


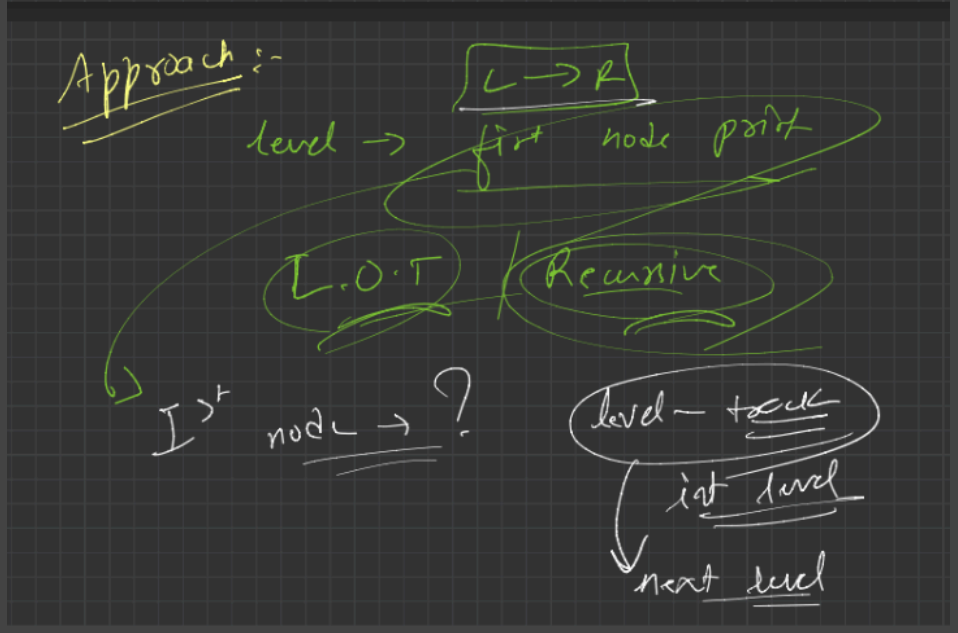
Bottom View:

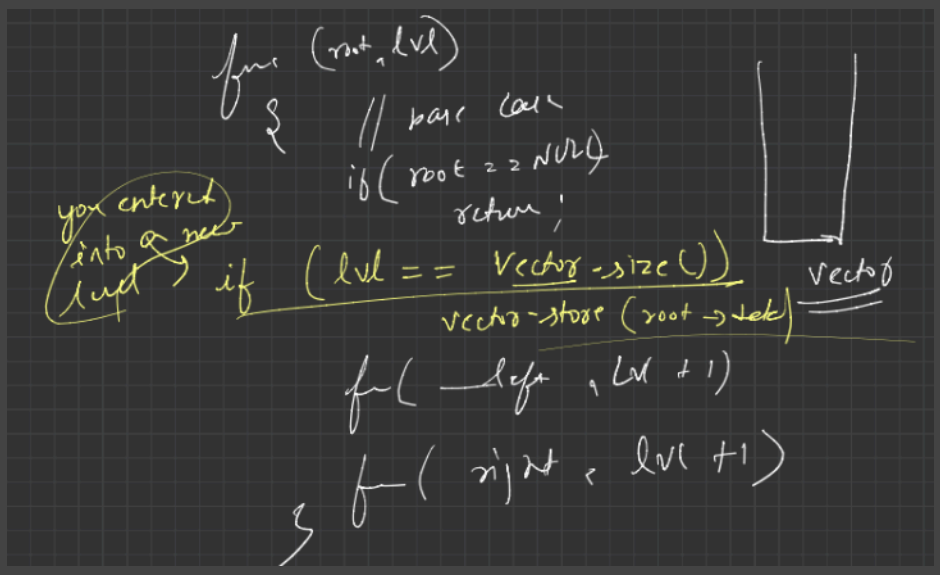




Left View:





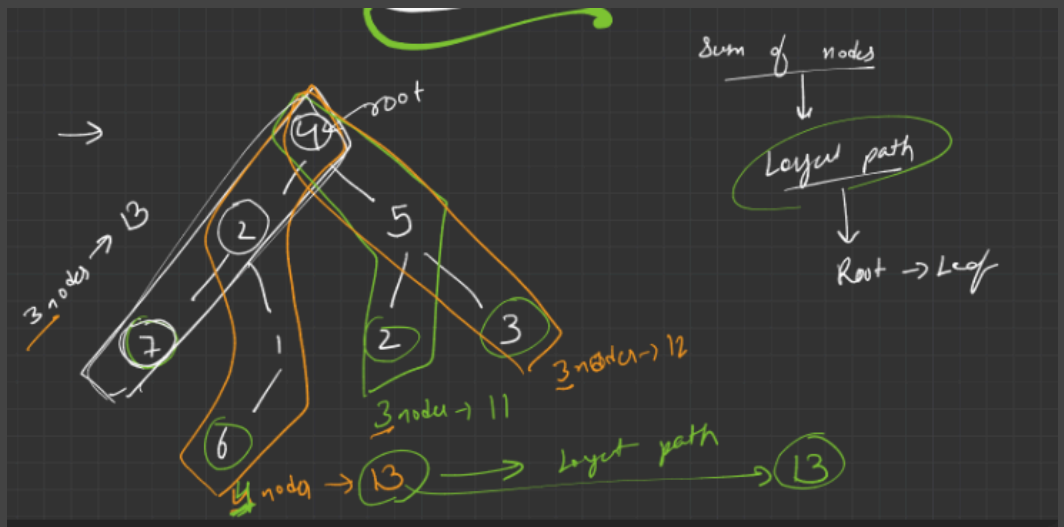


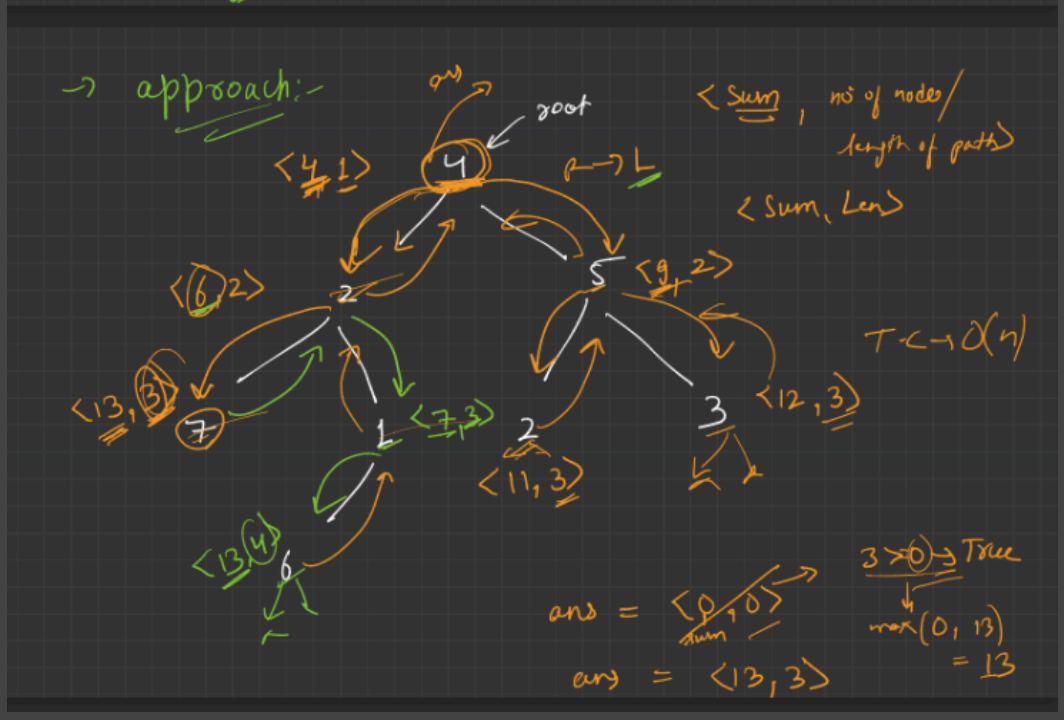
Right View:

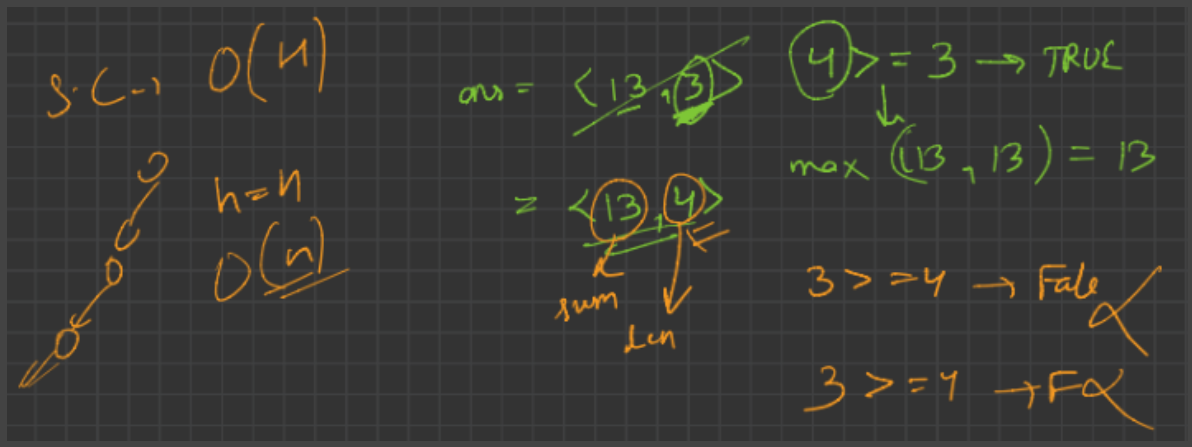
Just make the recursive call for right first and then left.

Diagonal Traversal:

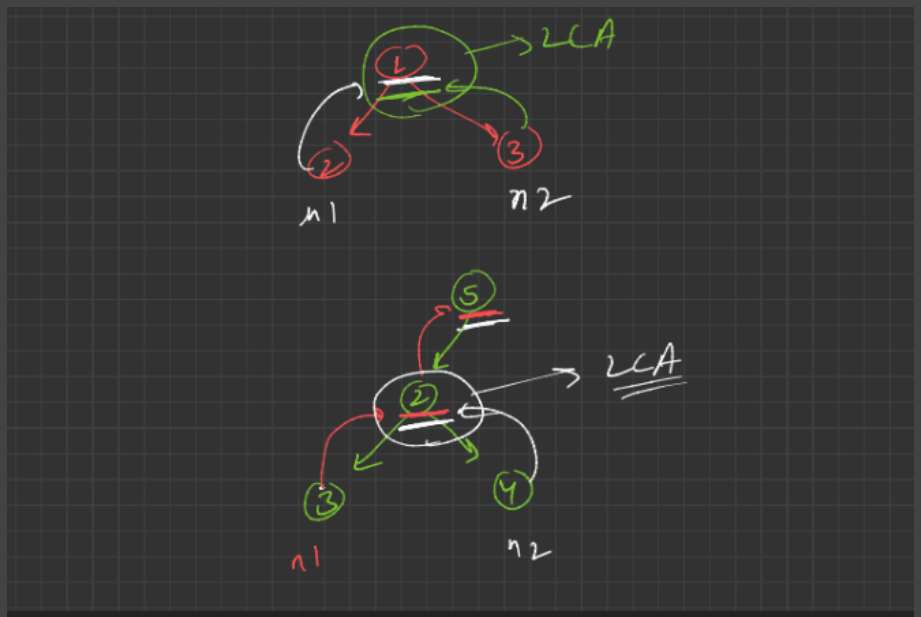
Sum Of The Largest Bloodline Of A Tree(Sum Of Nodes On The Longest Path From Roof To Leaf Node):

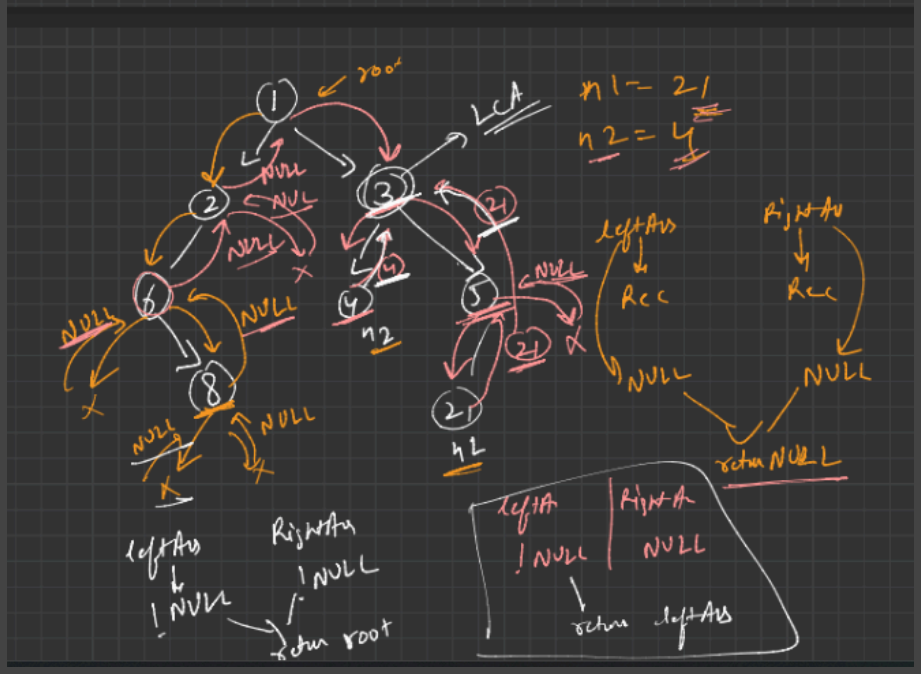


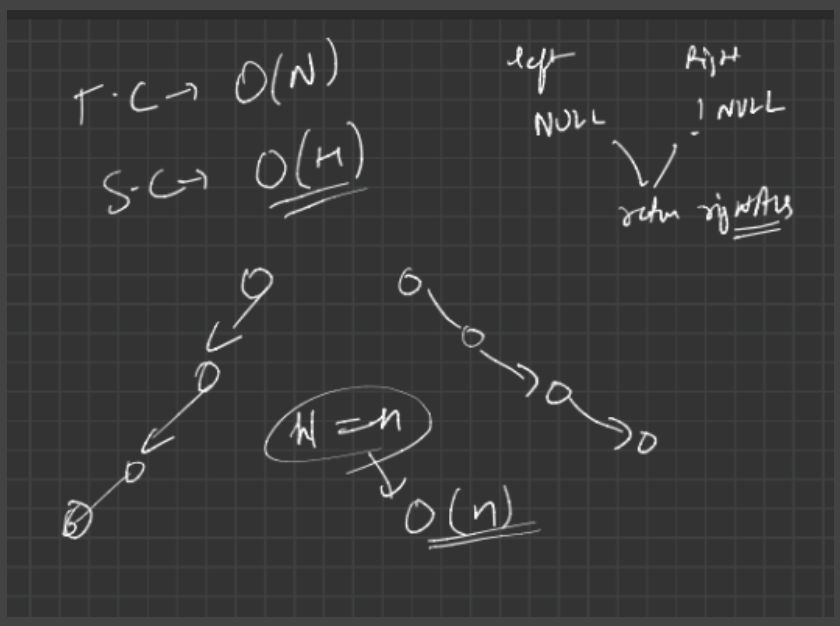




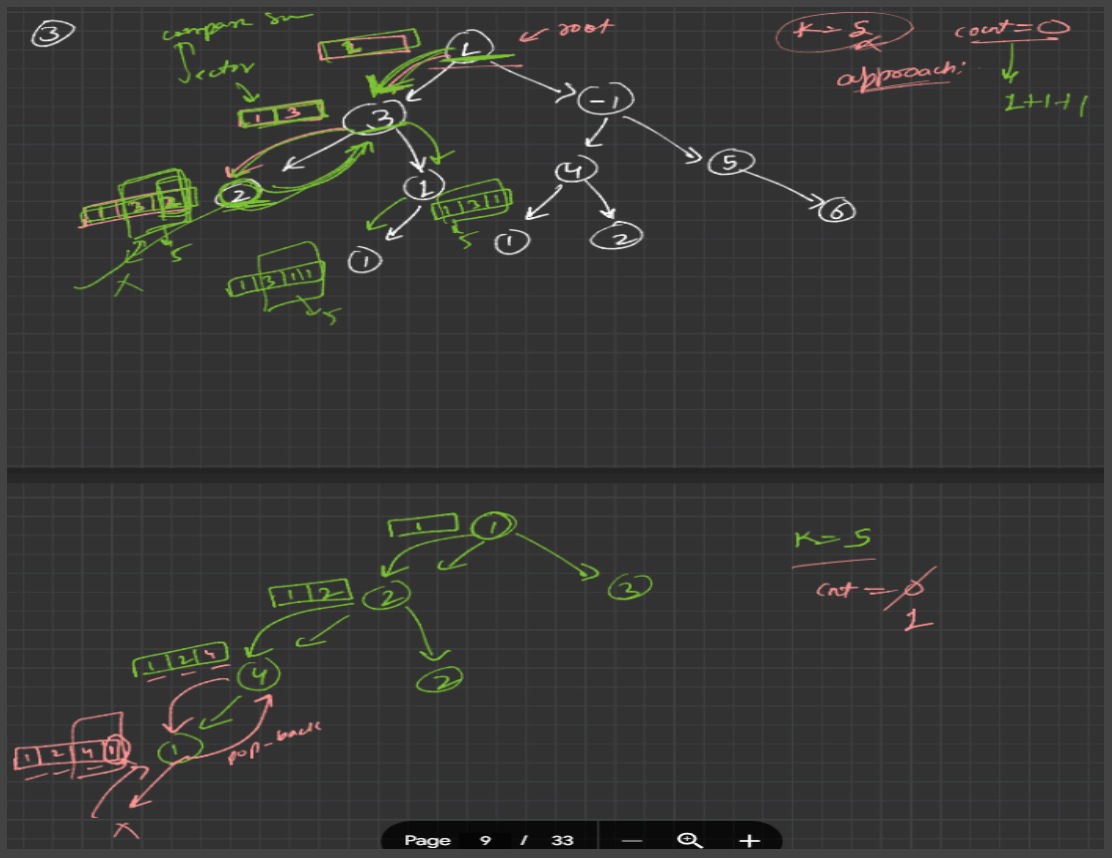
Lowest Common Ancestor:



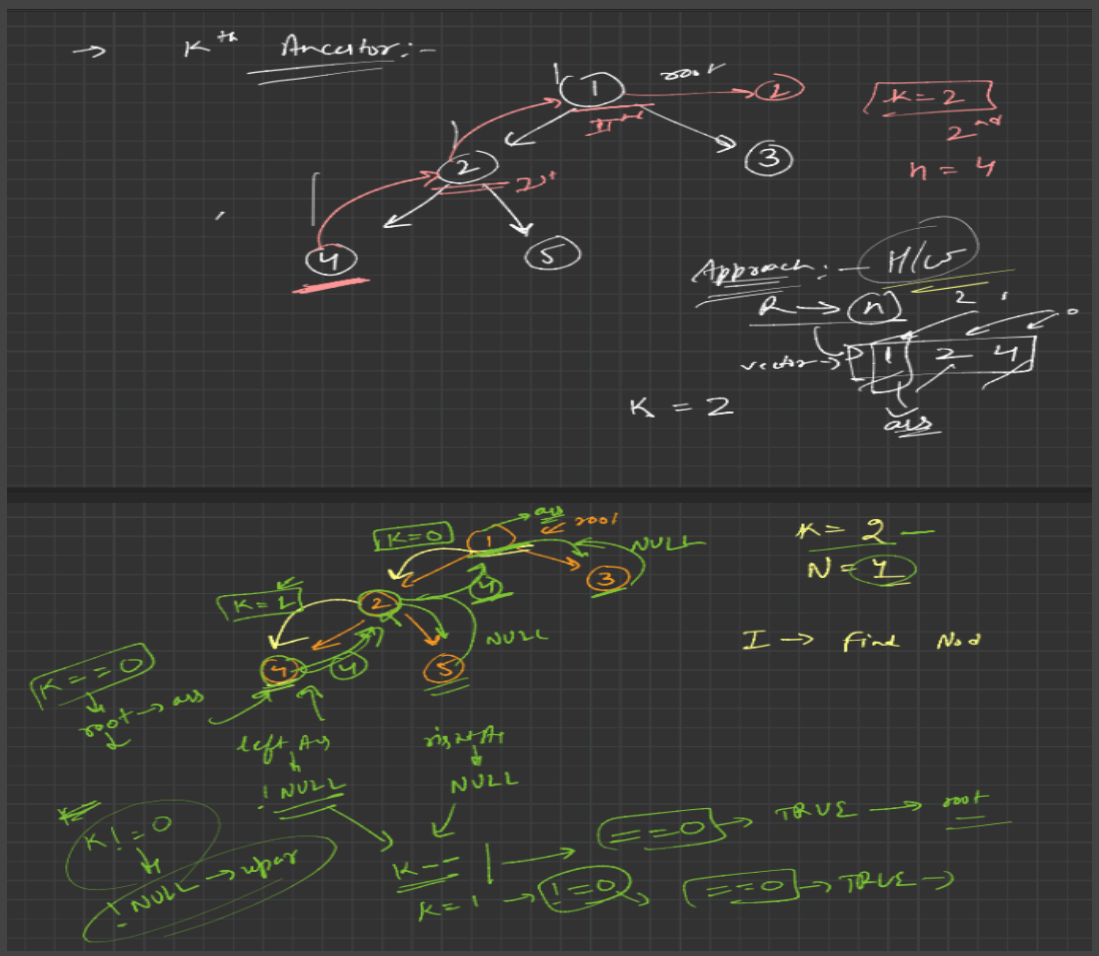


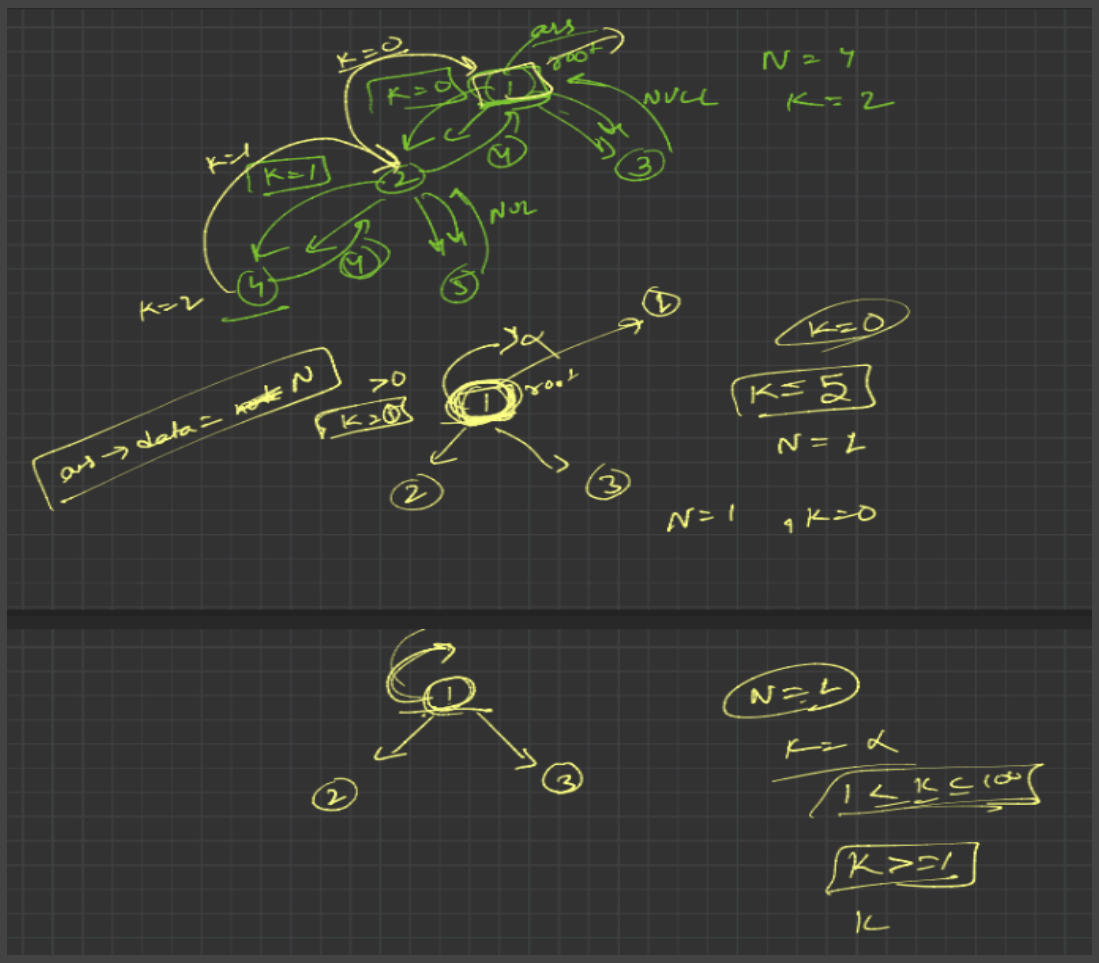


K Sum Paths:

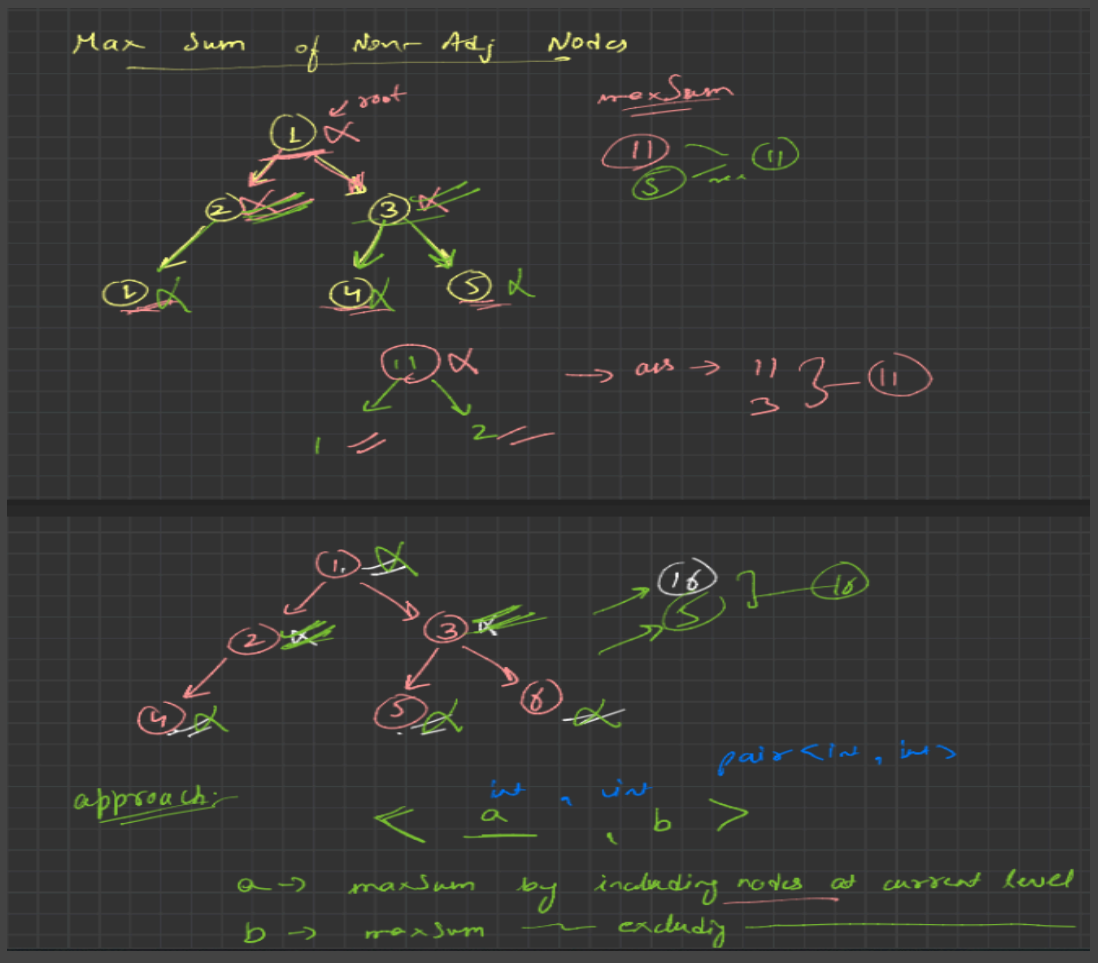


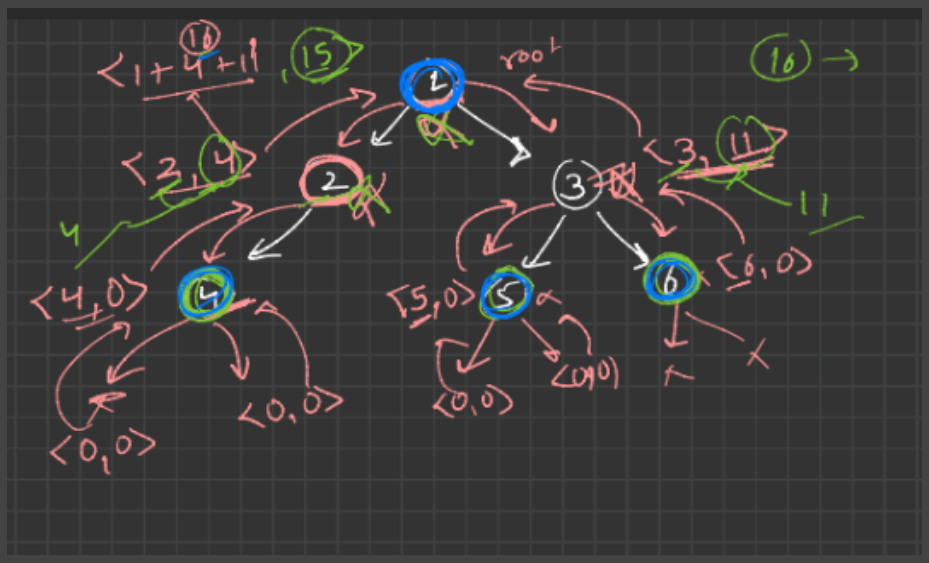
Kth Ancestor In A Tree:



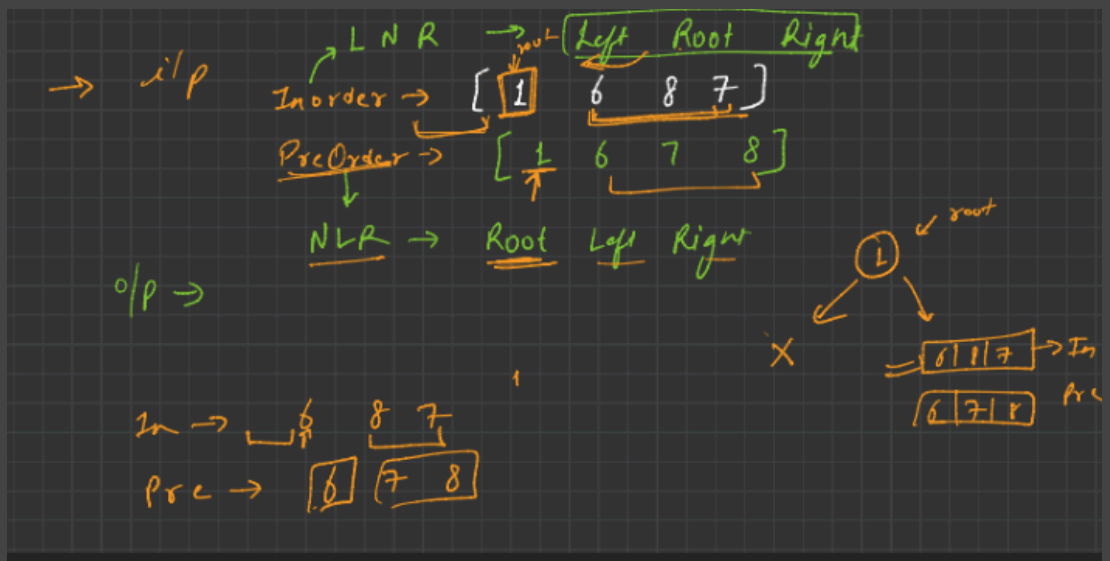


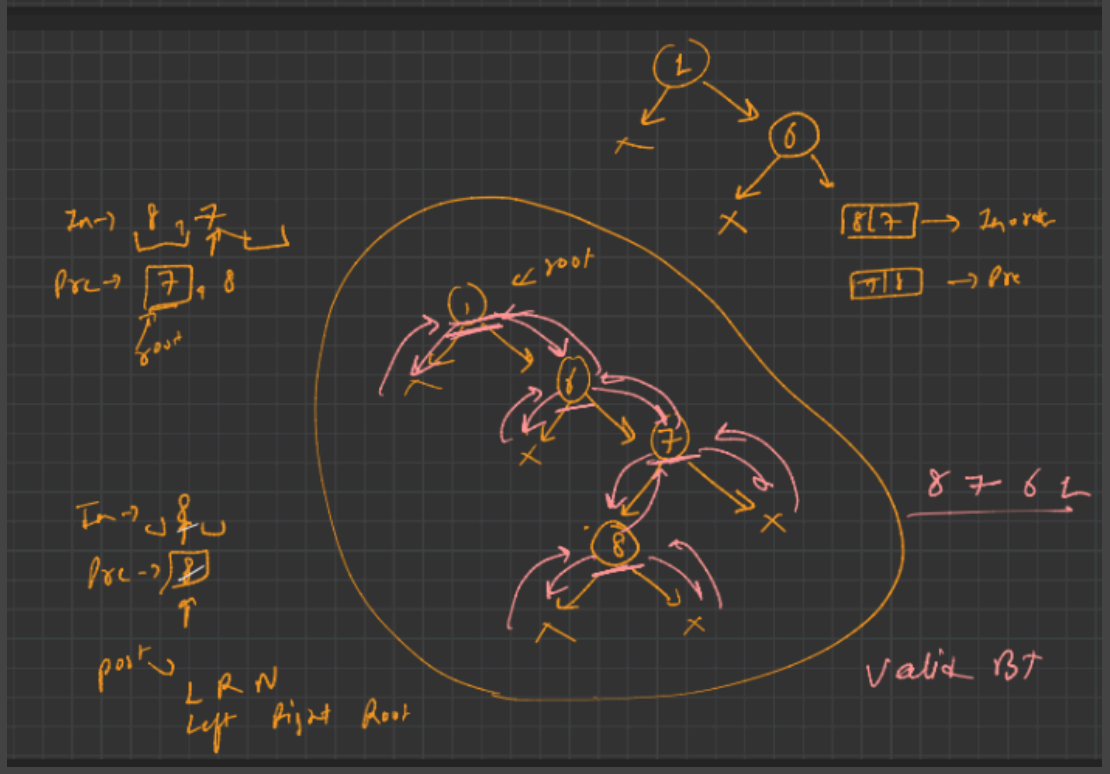
Maximum Sum Of Non-Adjacent Nodes:

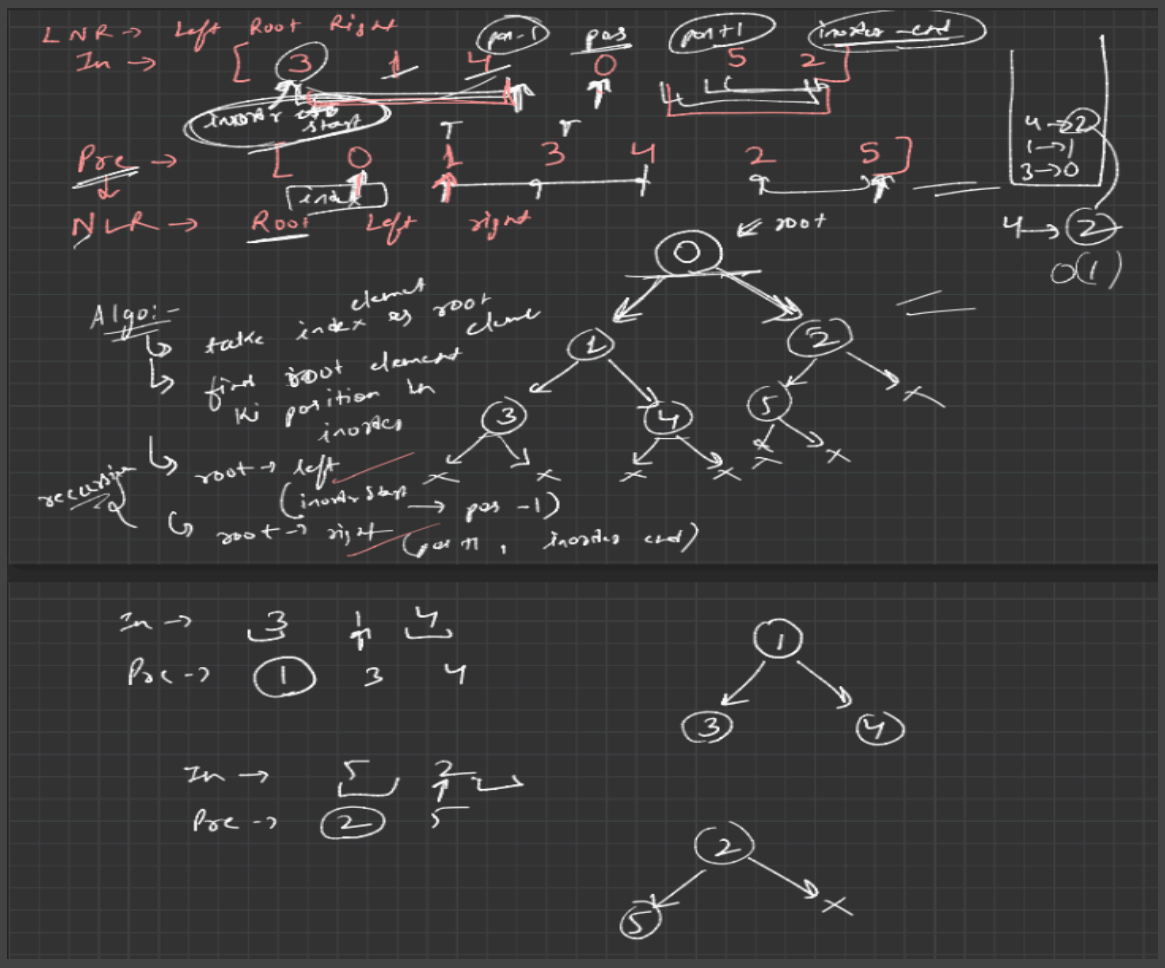




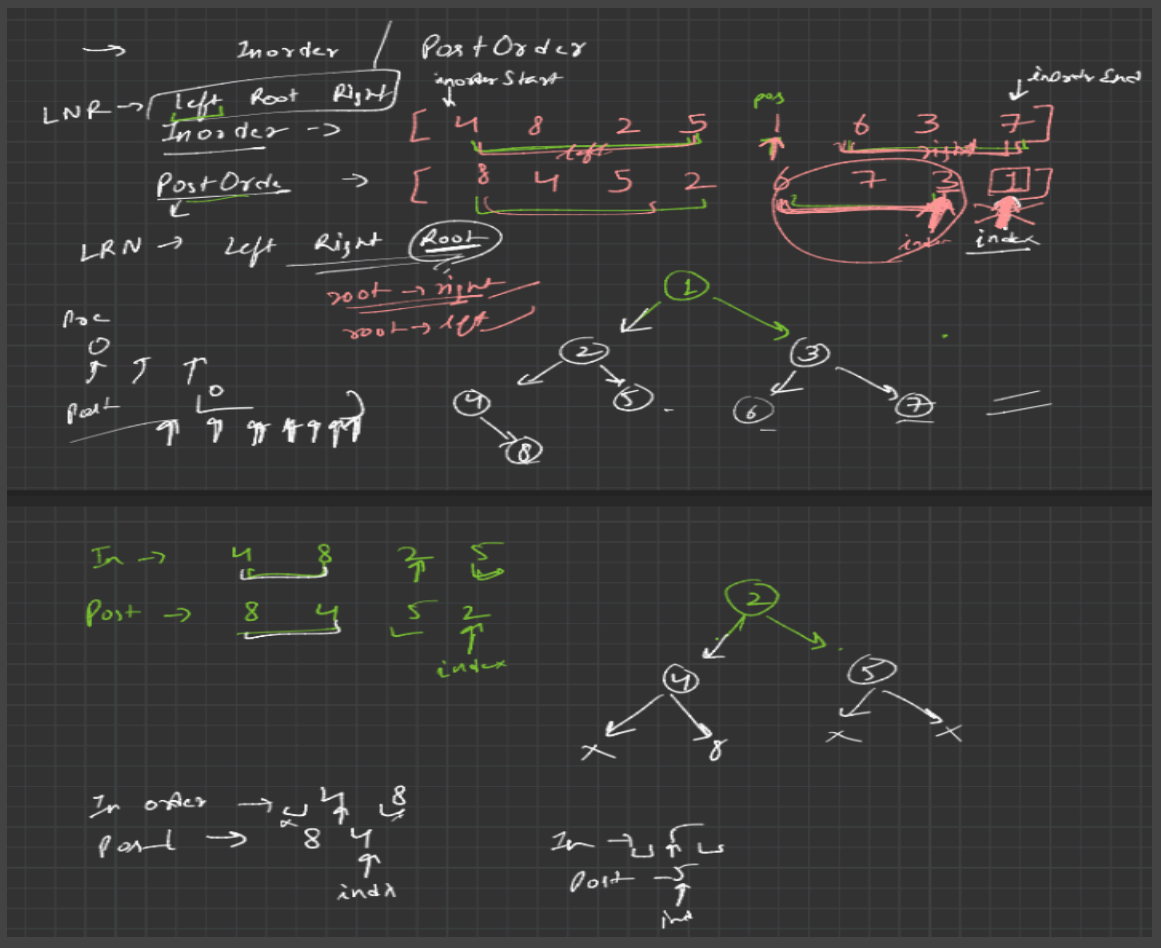
Construct Tree From InOrder And PreOrder:

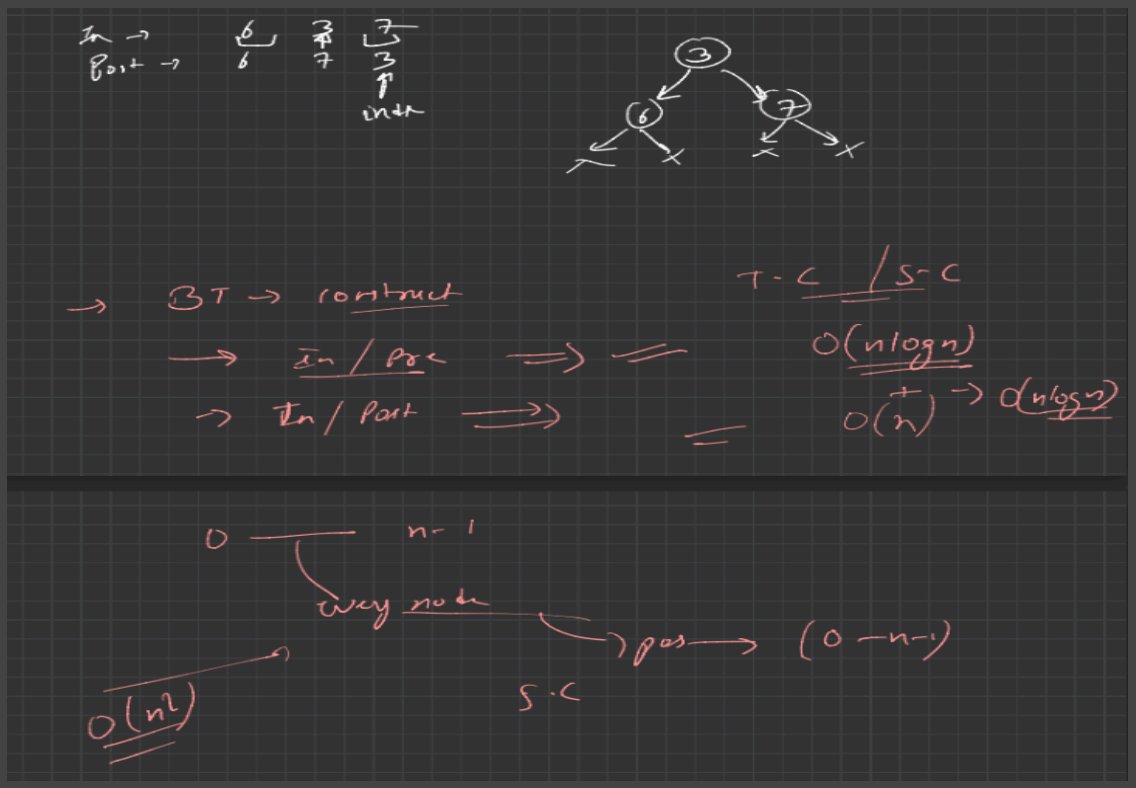




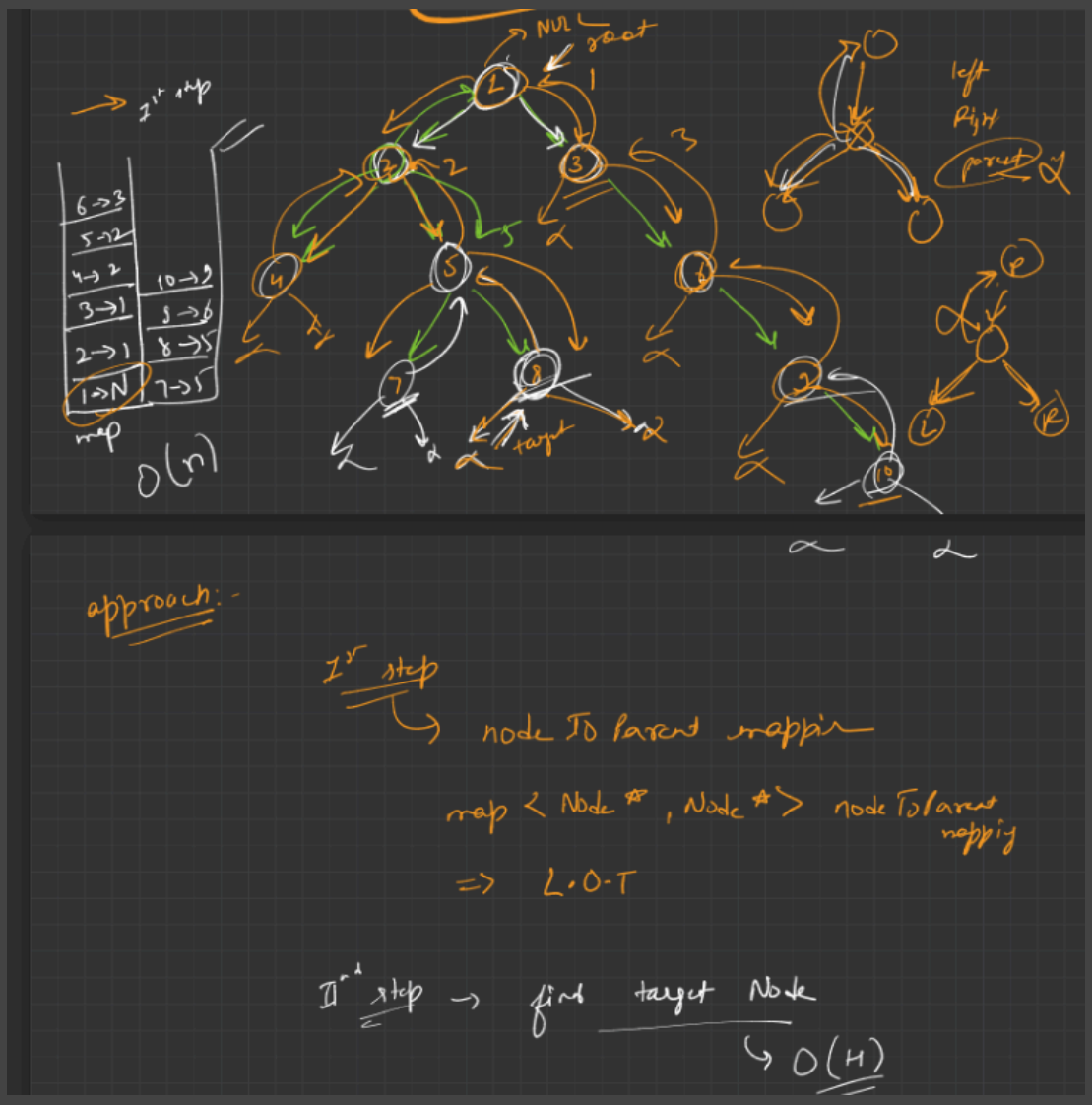


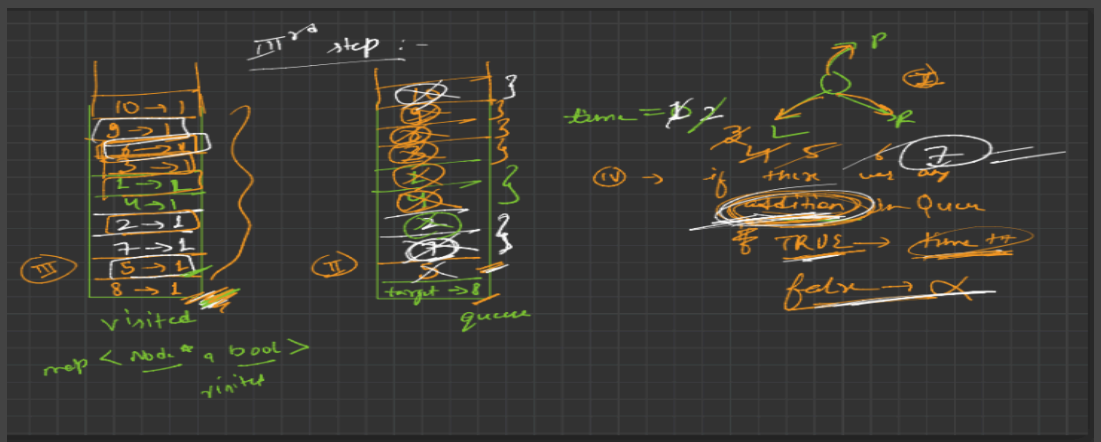
Construct Tree From InOrder And PostOrder:

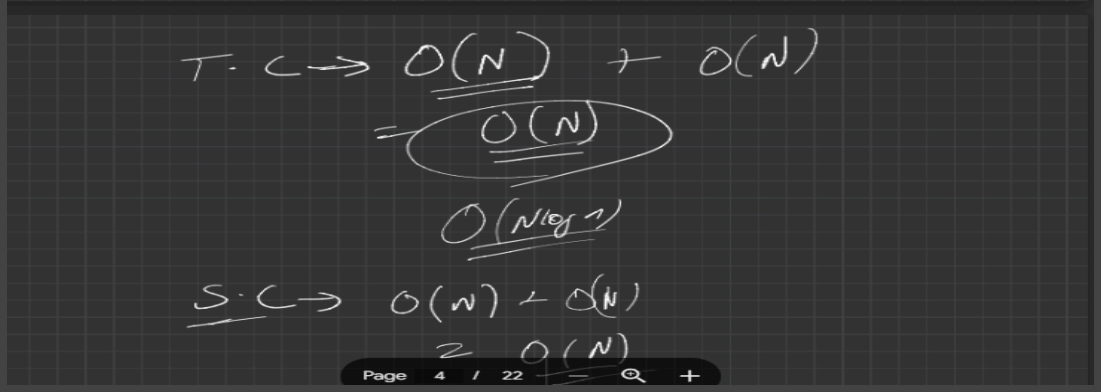




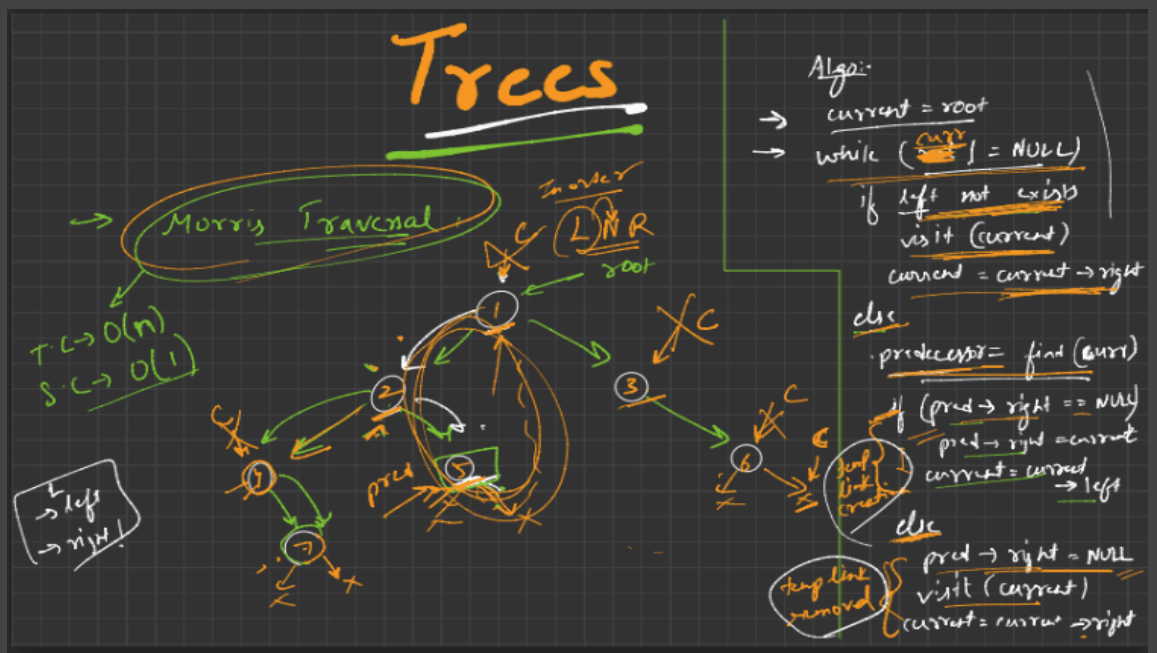
Minimum Tree To Burn The Entire Binary Tree:

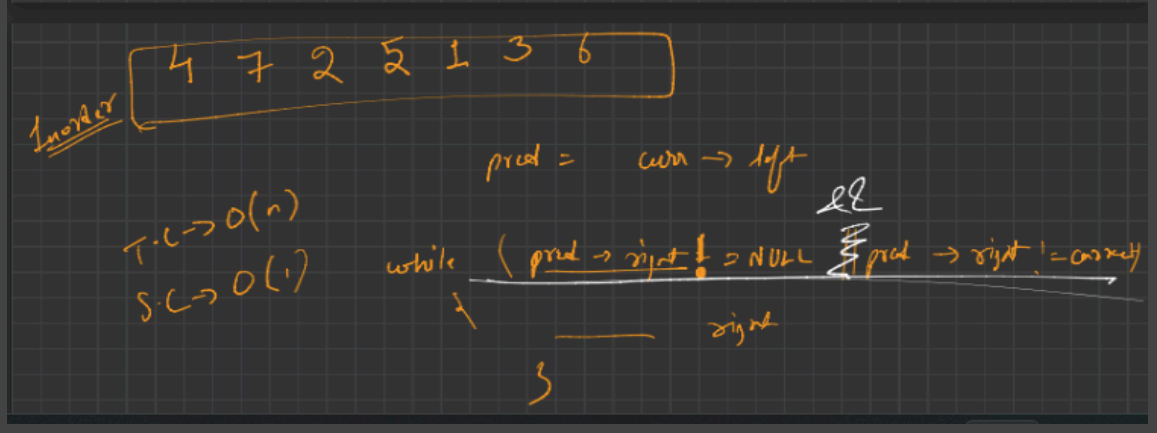




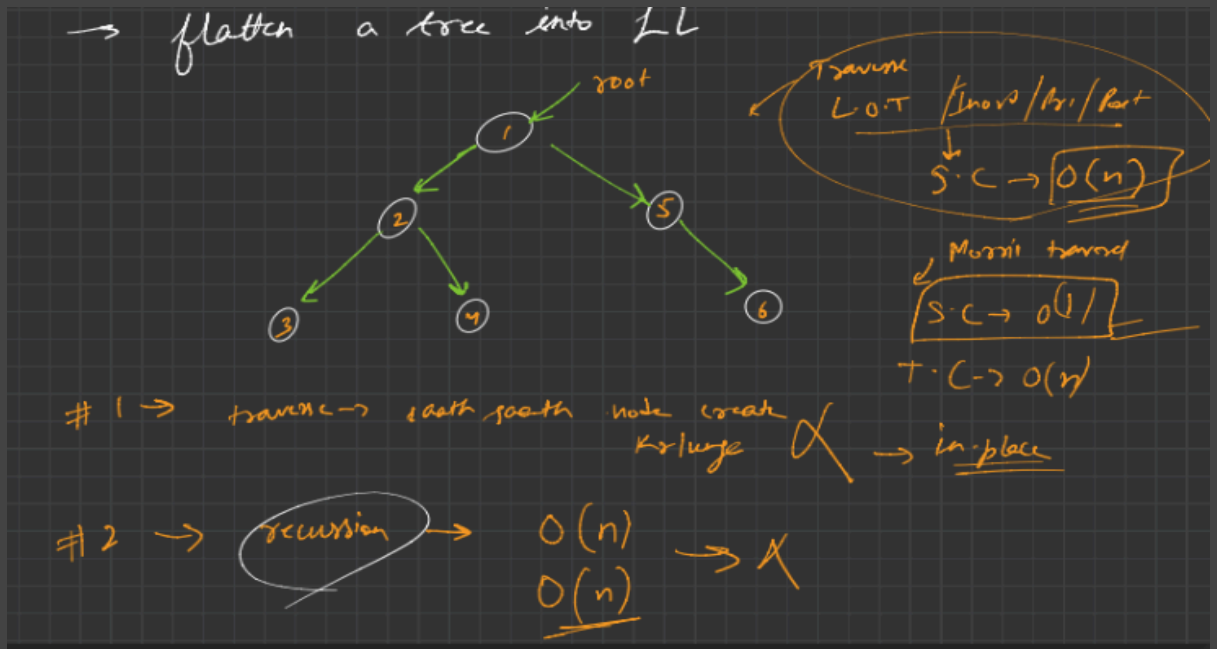


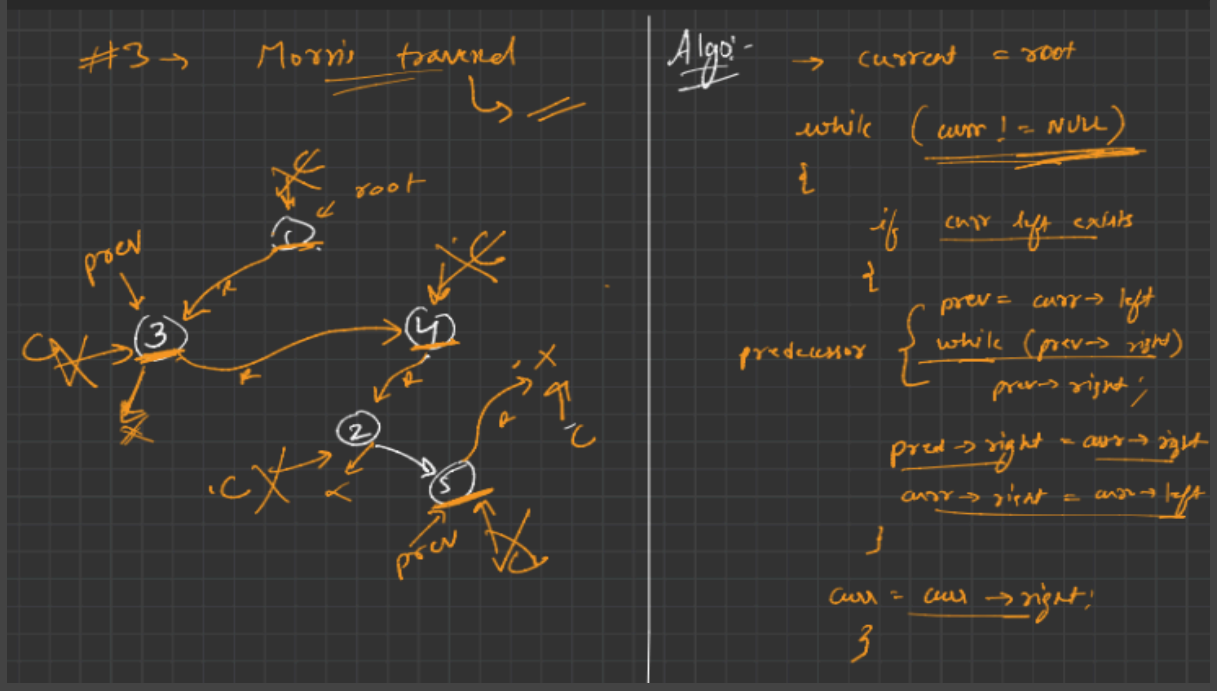
Morris Traversal:

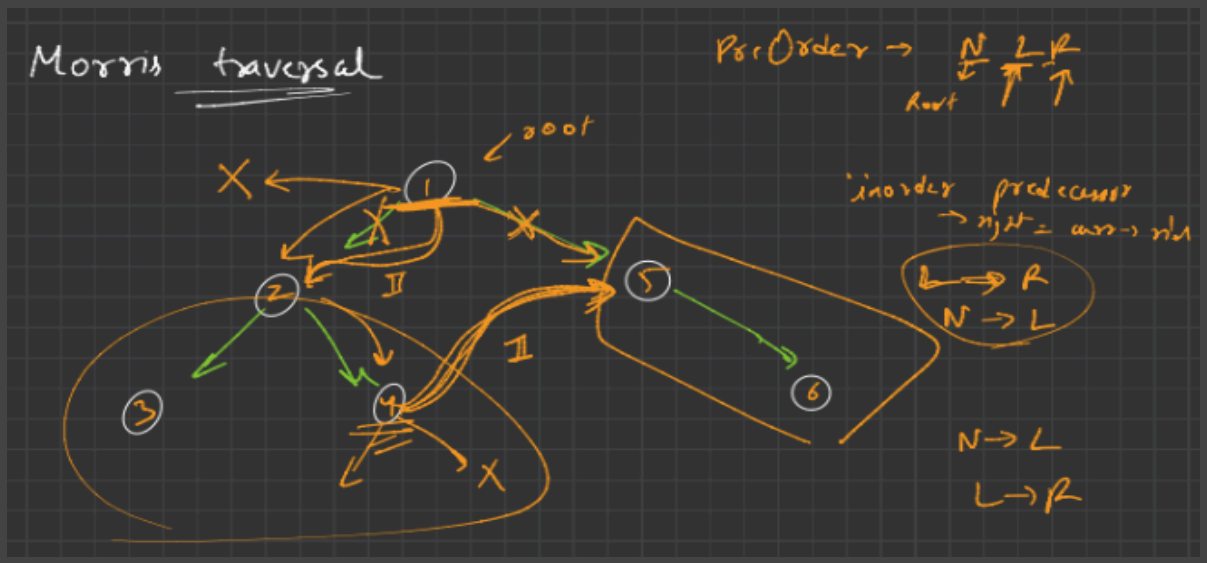




Flatten A Binary Tree To Linked List:

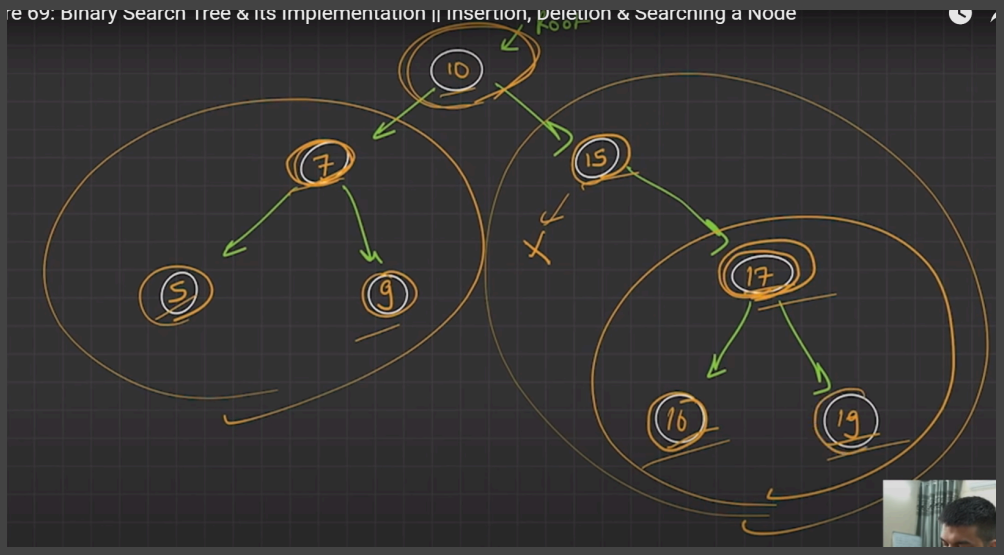


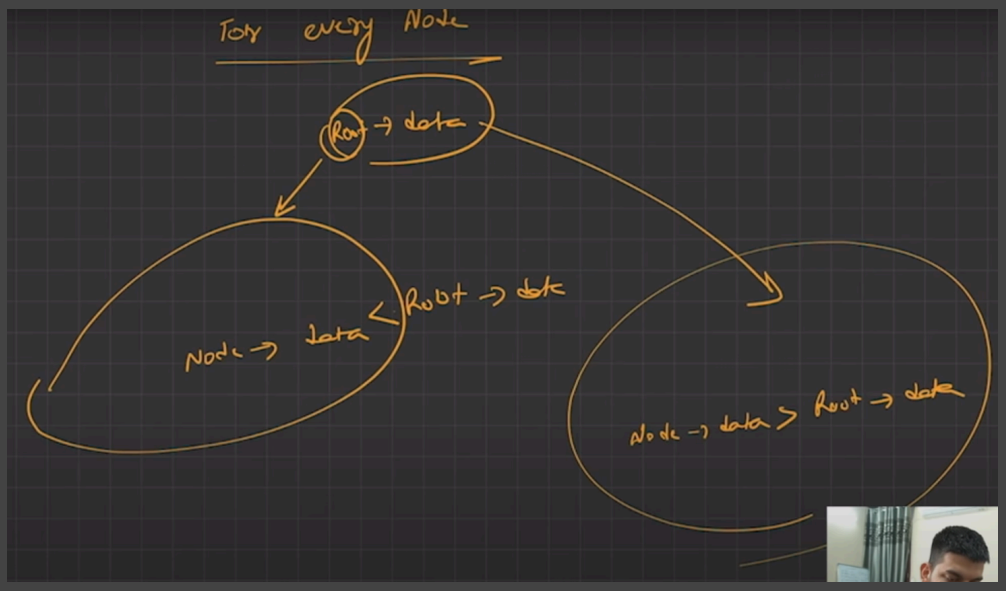


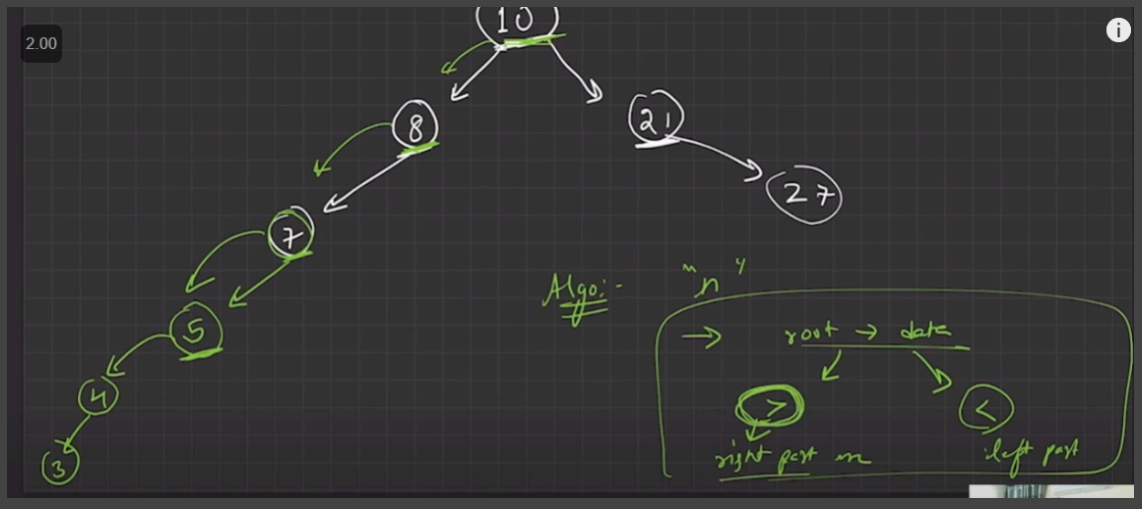


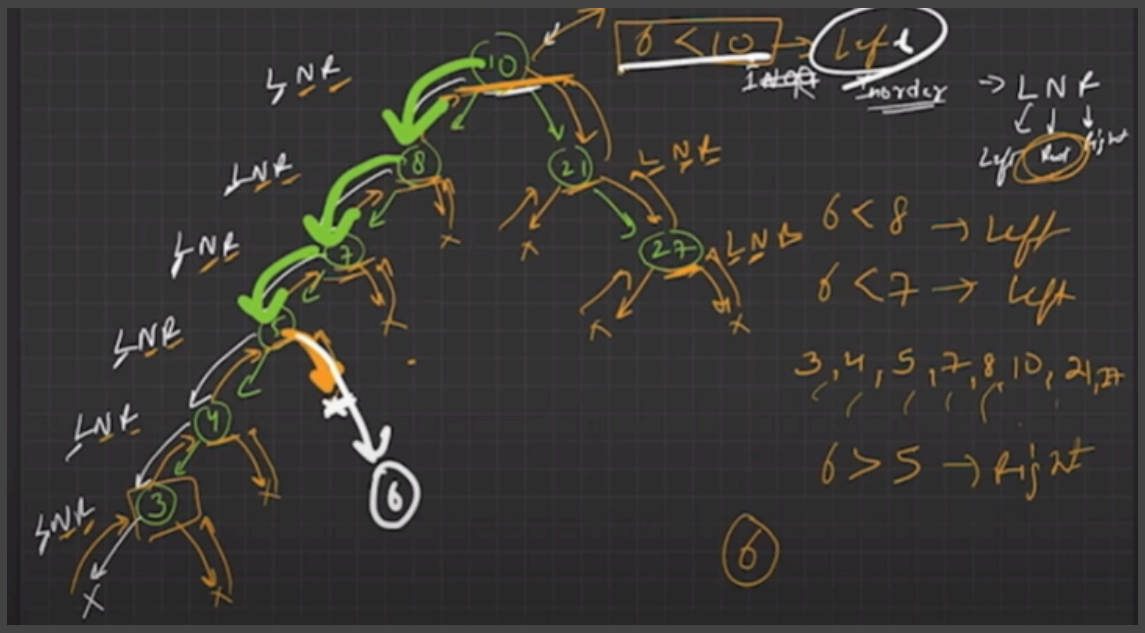
Binary Search Tree(BST)

Insert A Node In BST:

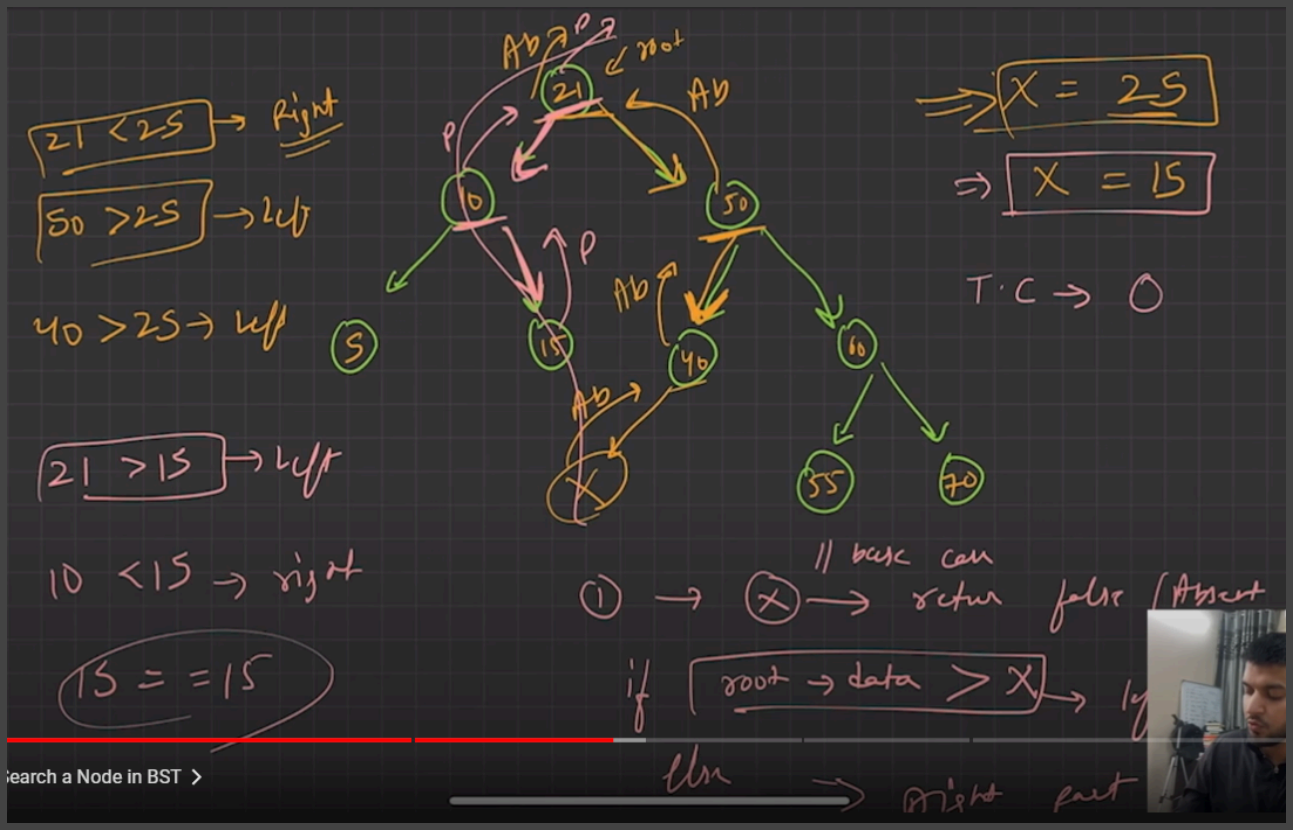




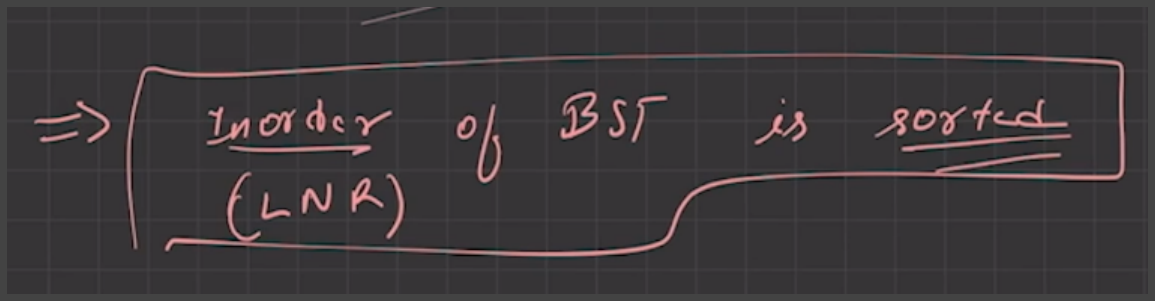


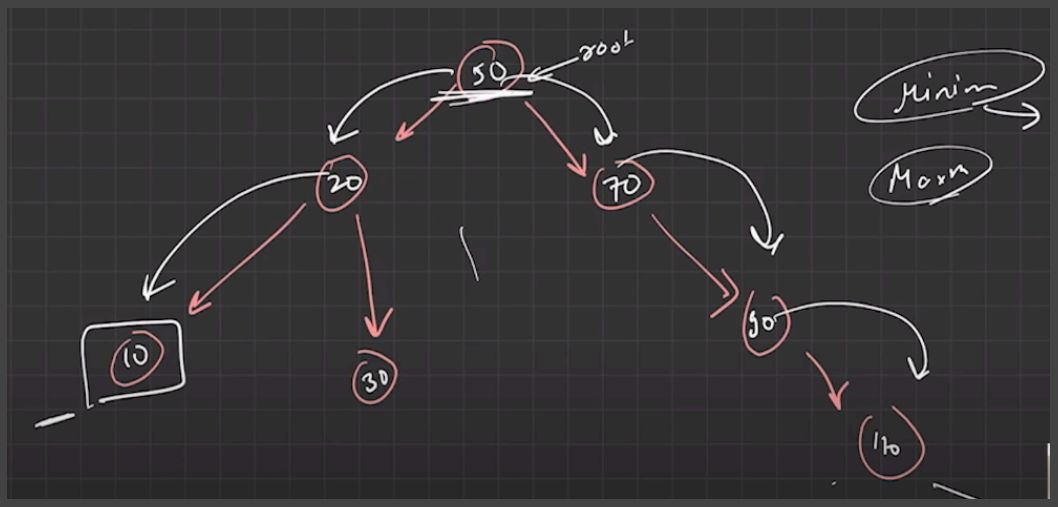


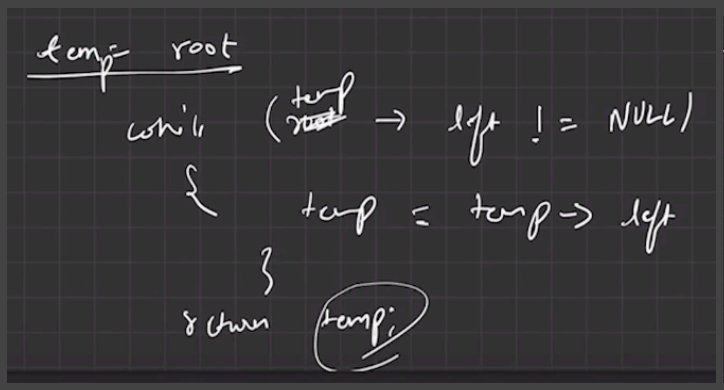
Search A Node In BST:

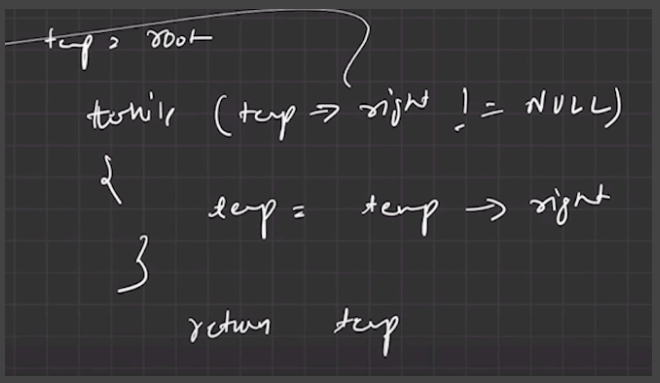


Minimum/Maximum Value In BST:

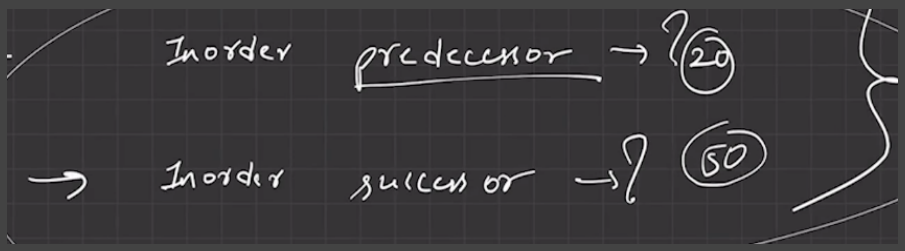








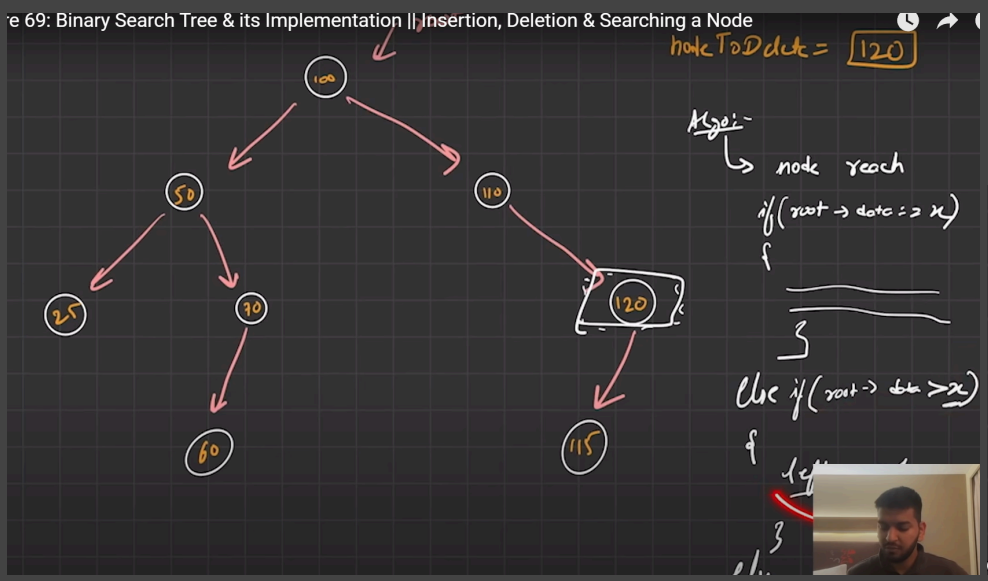
Inorder Predecessor In BST:

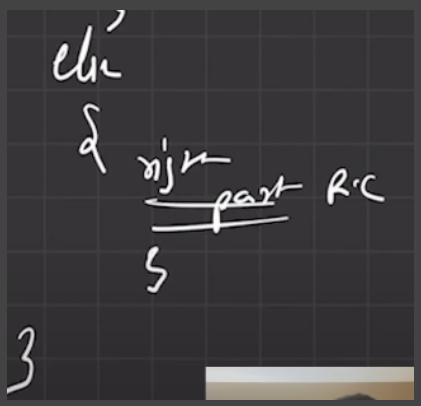


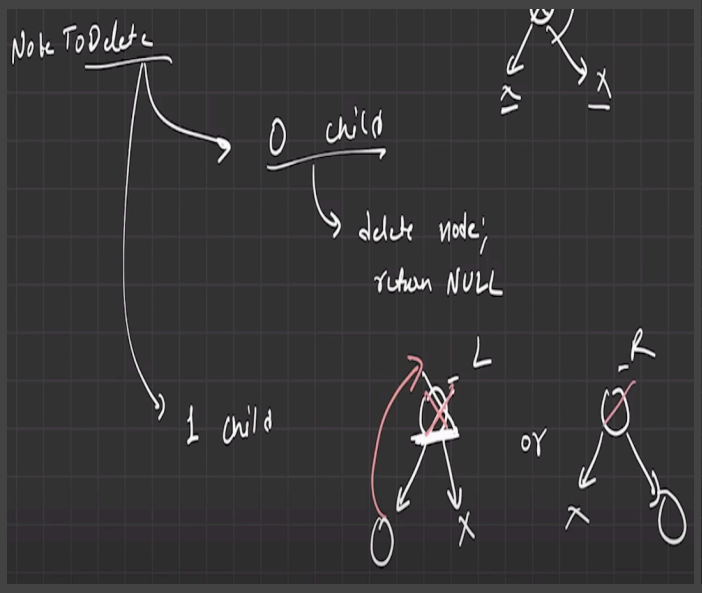
Inorder Successor In BST:

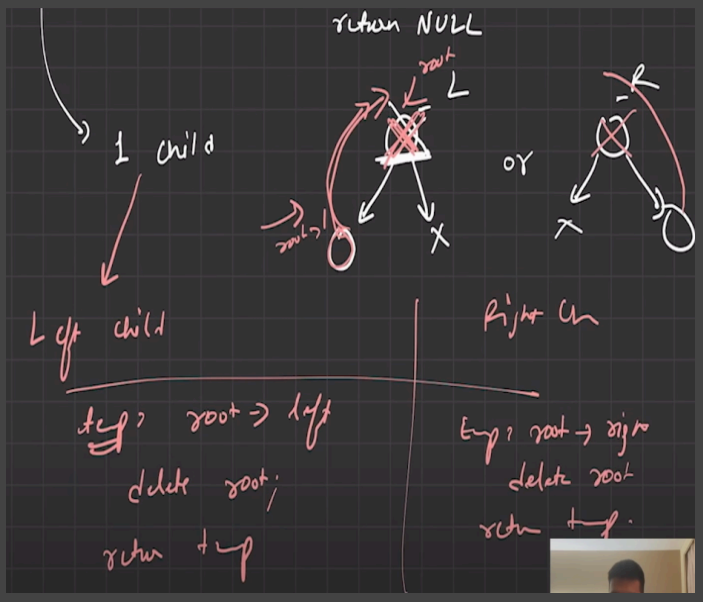
```
110
Printing Inorder
10 20 30 50 70 90 110
Printing Preorder
```

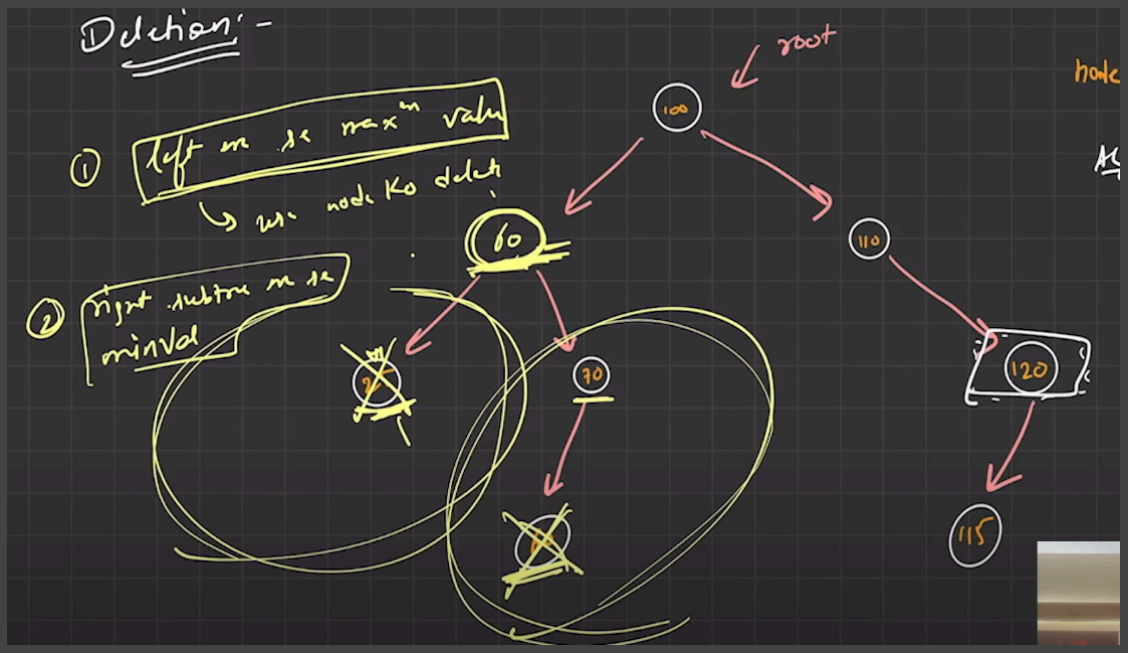
Delete A Node In BST:



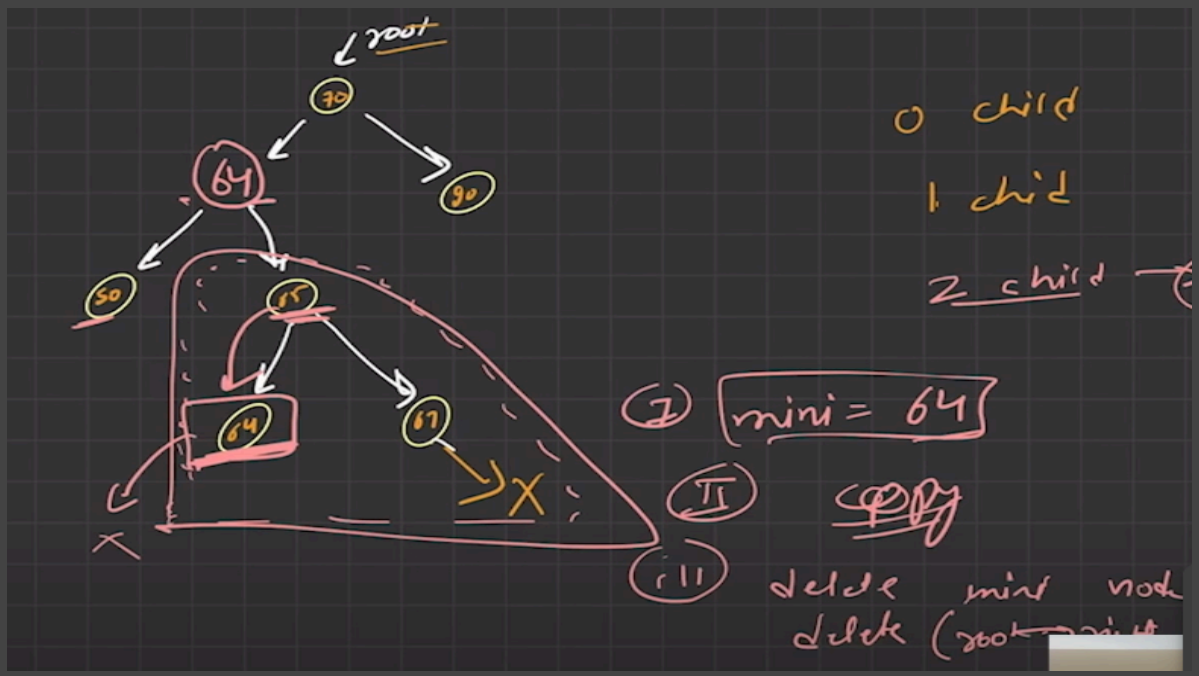



Note ToDelete
0 child
delete node;
return NULL
1 child
L
R
or
X

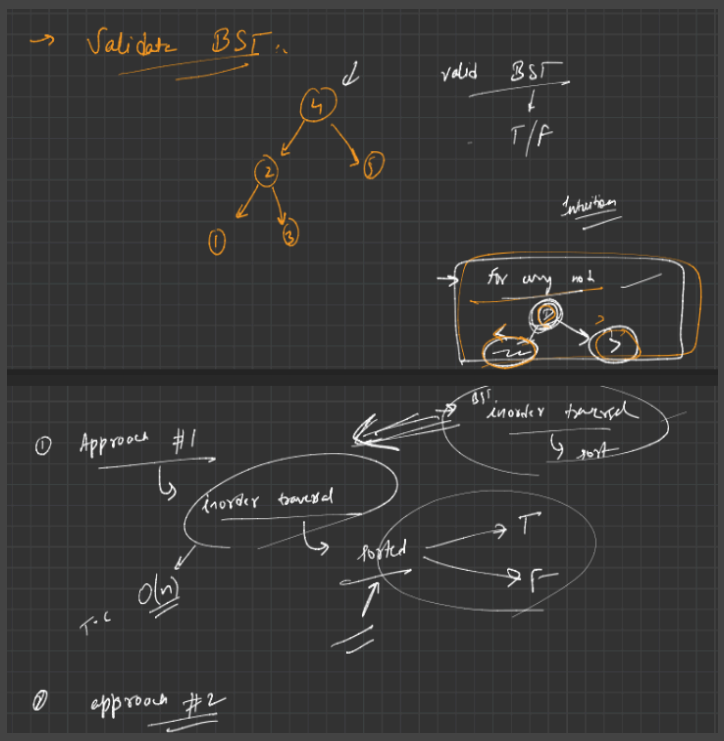

return NULL
root
L
R
1 child
or
X
Left child
Right Ch
temp = root -> left
delete root;
return temp
temp = root -> right
delete root
return temp.

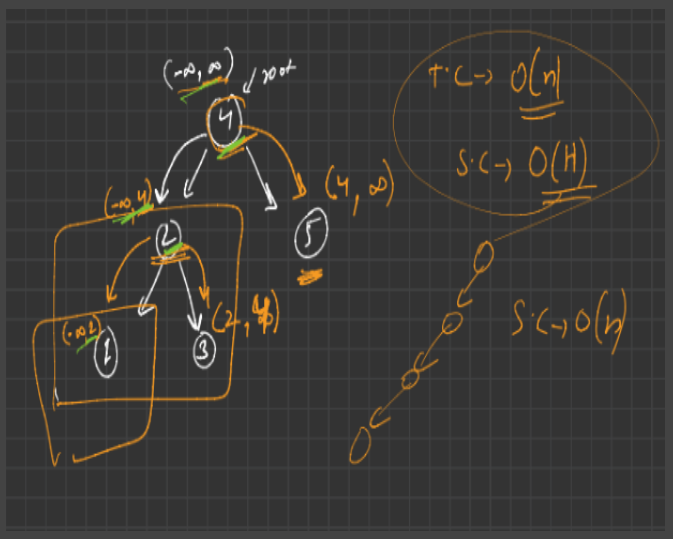

Deletion:-
root
node
1 left se max^m value
2 right subtree se
minVal
100
60
110
70
120
115

Any one 1 or 2

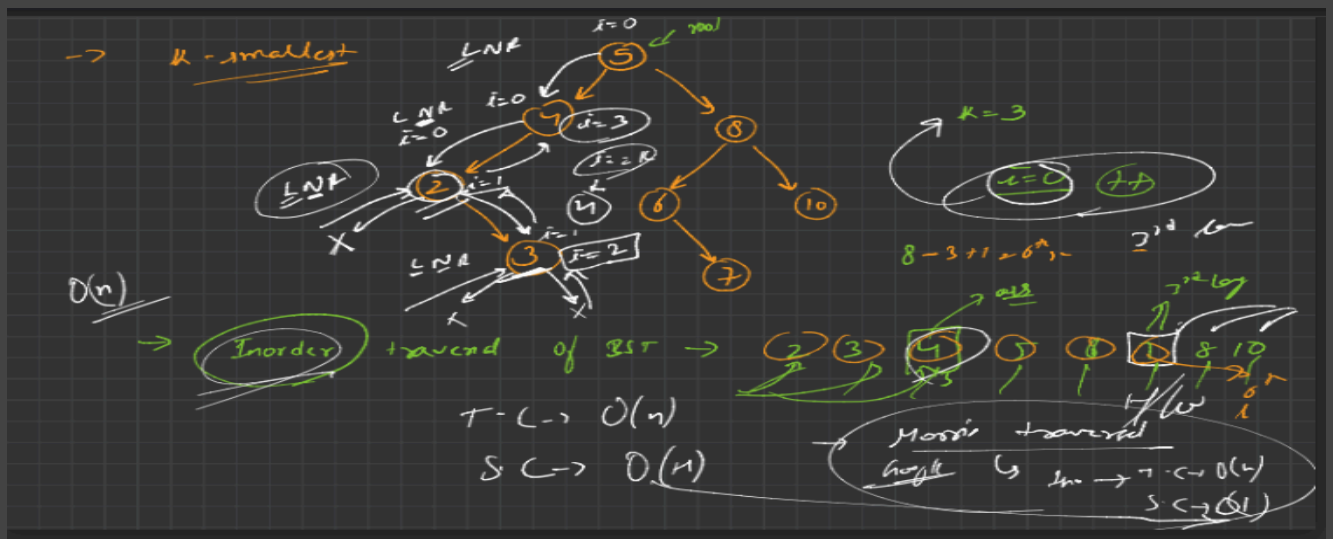

root
70
64
90
50
65
64
67
X
0 child
1 child
2 child
2 mini = 64
delete mini node
delete (root

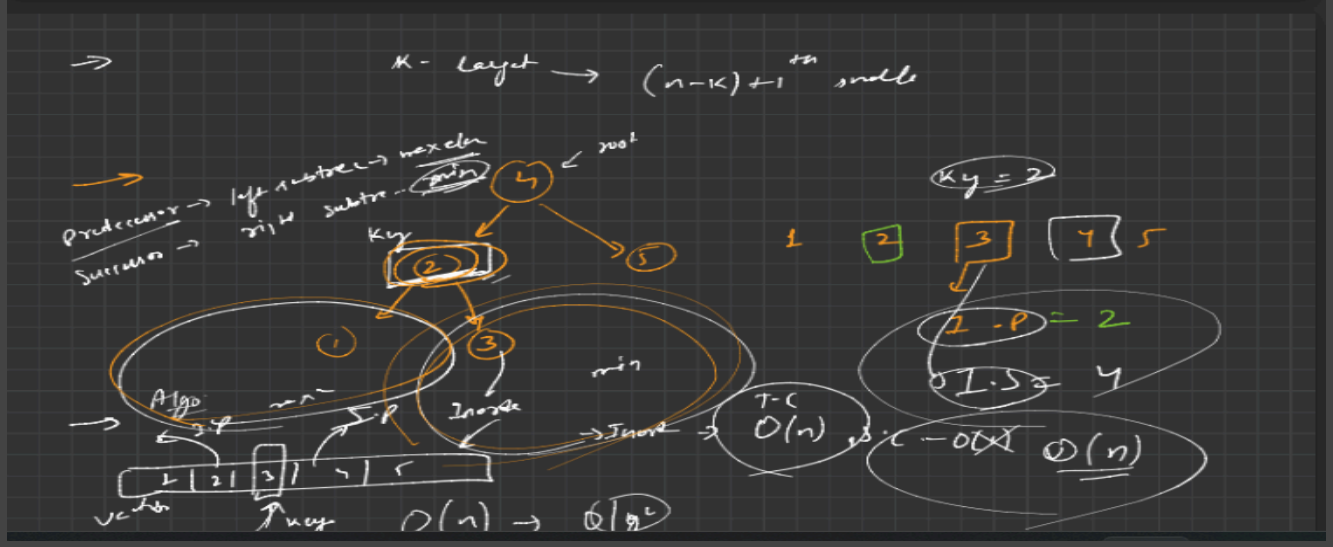
Validate BST:

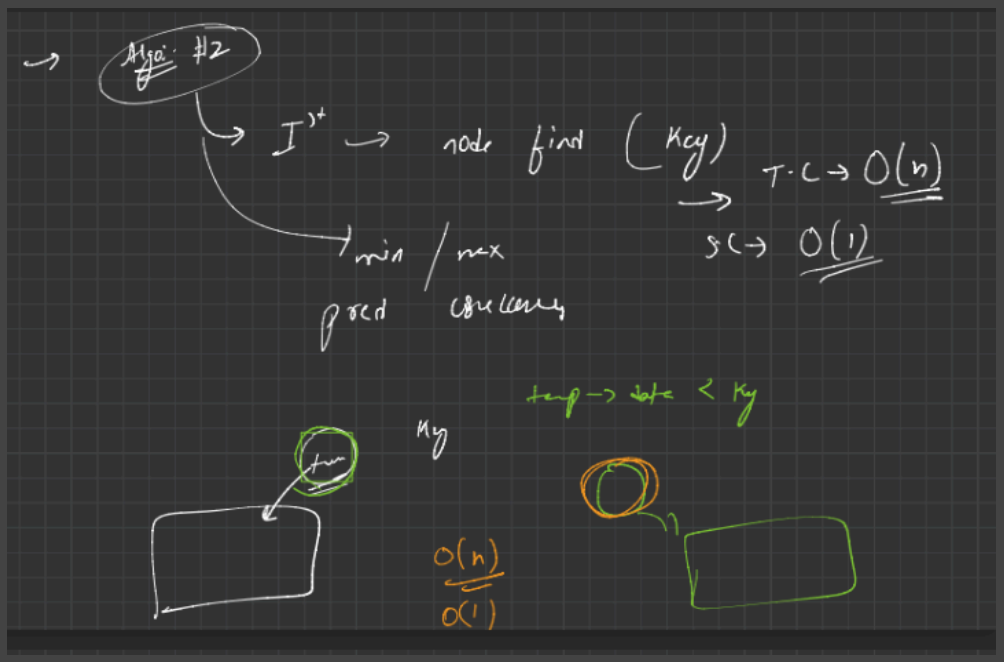




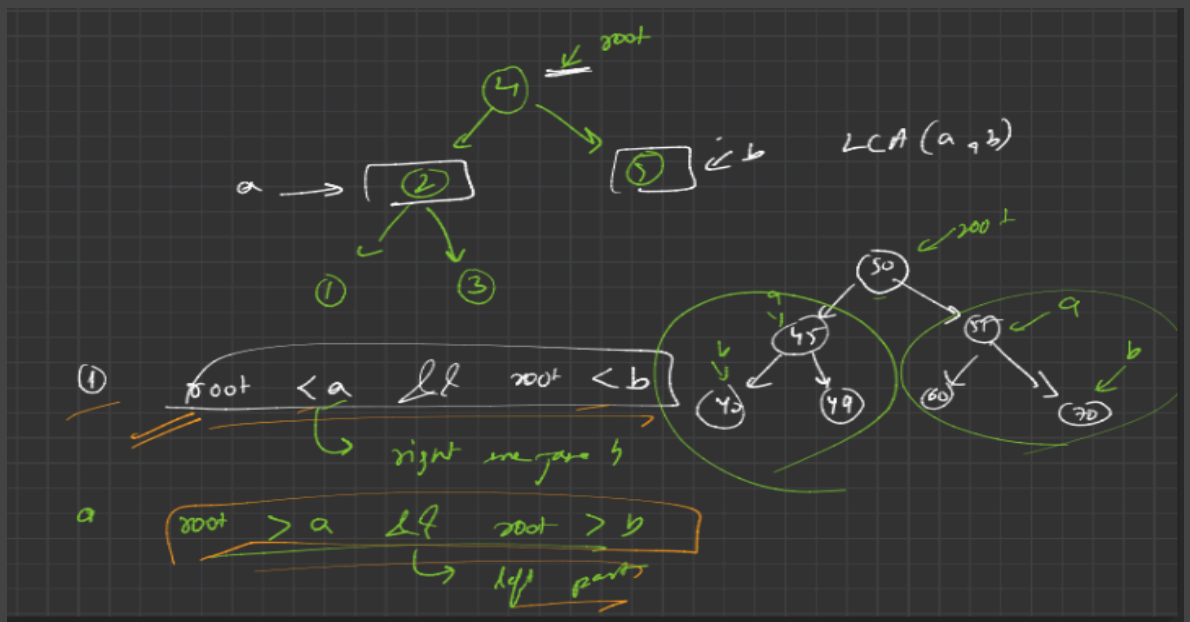
Kth Smallest/Largest Element In BST:

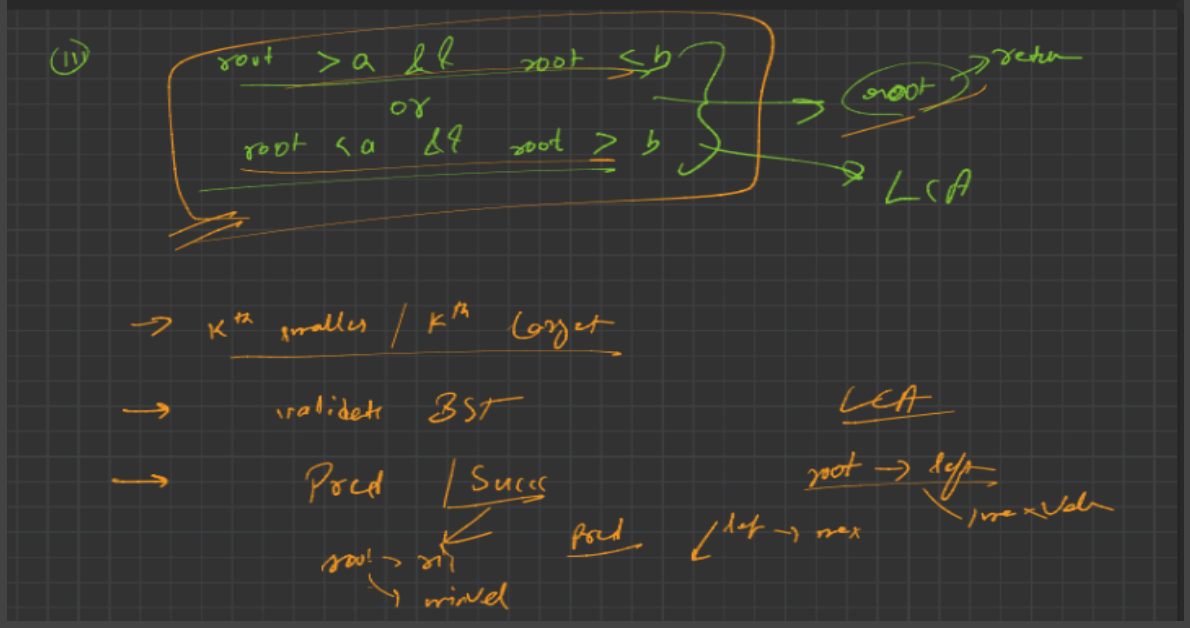




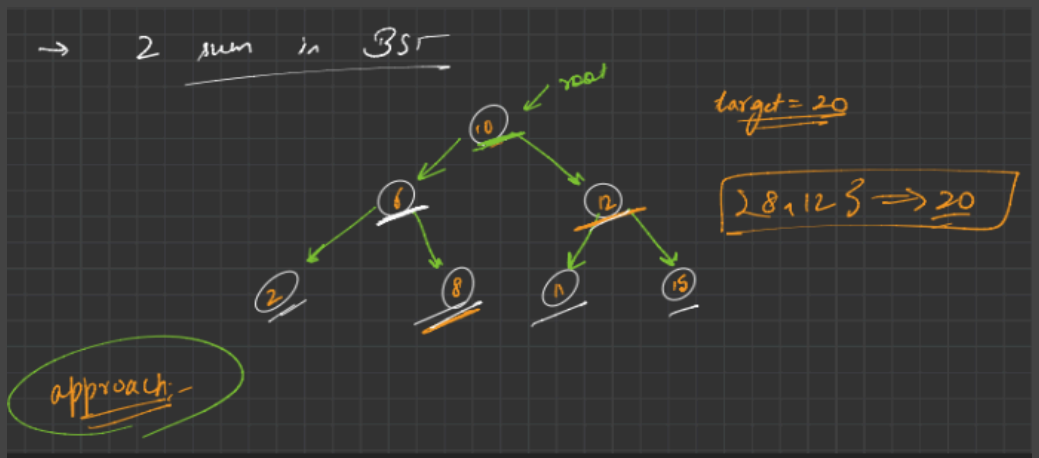


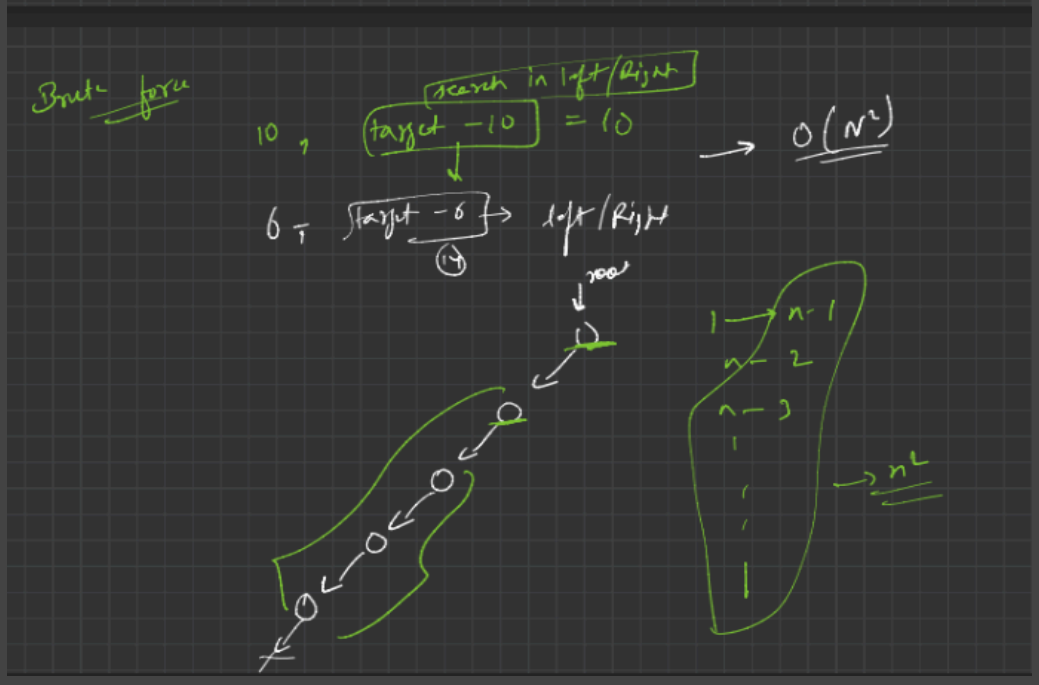
Lowest Common Ancestor In BST:

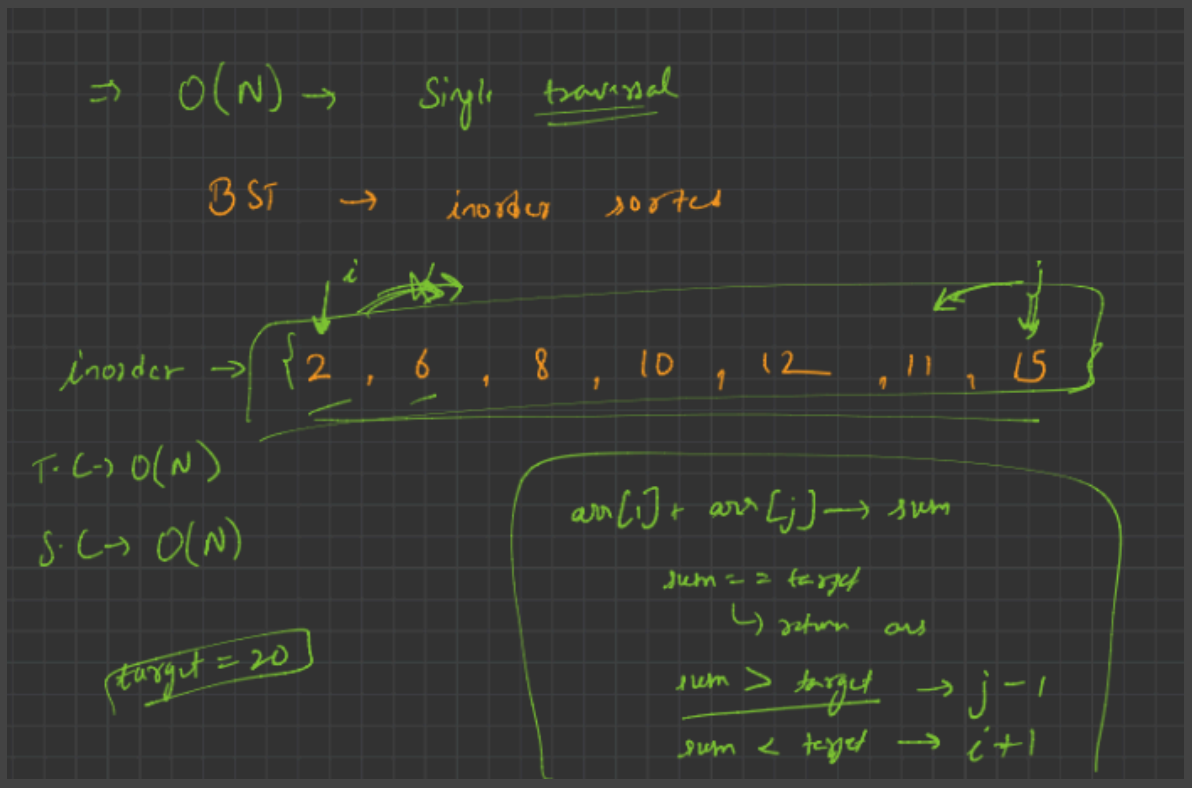




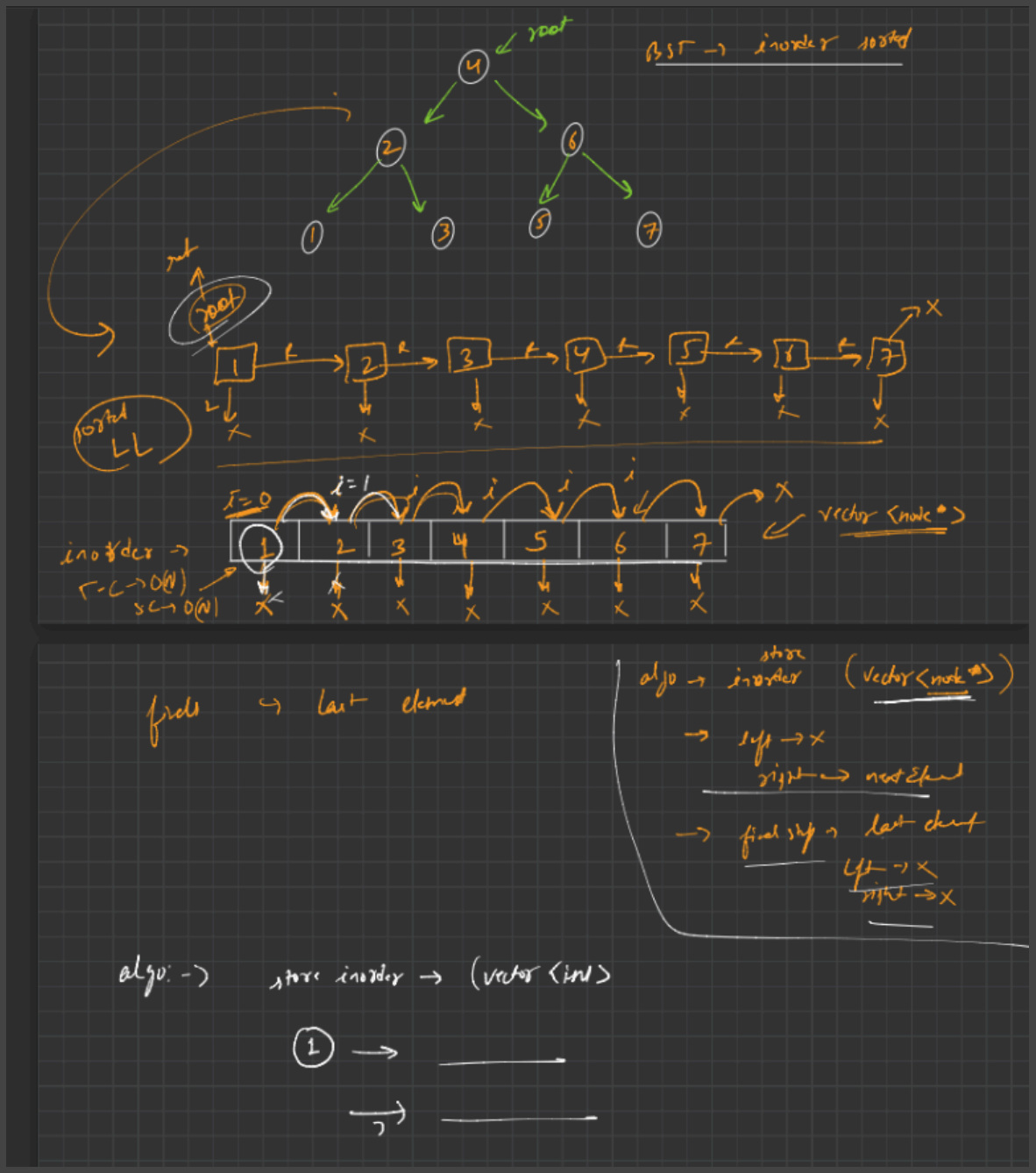
Two Sum In BST:



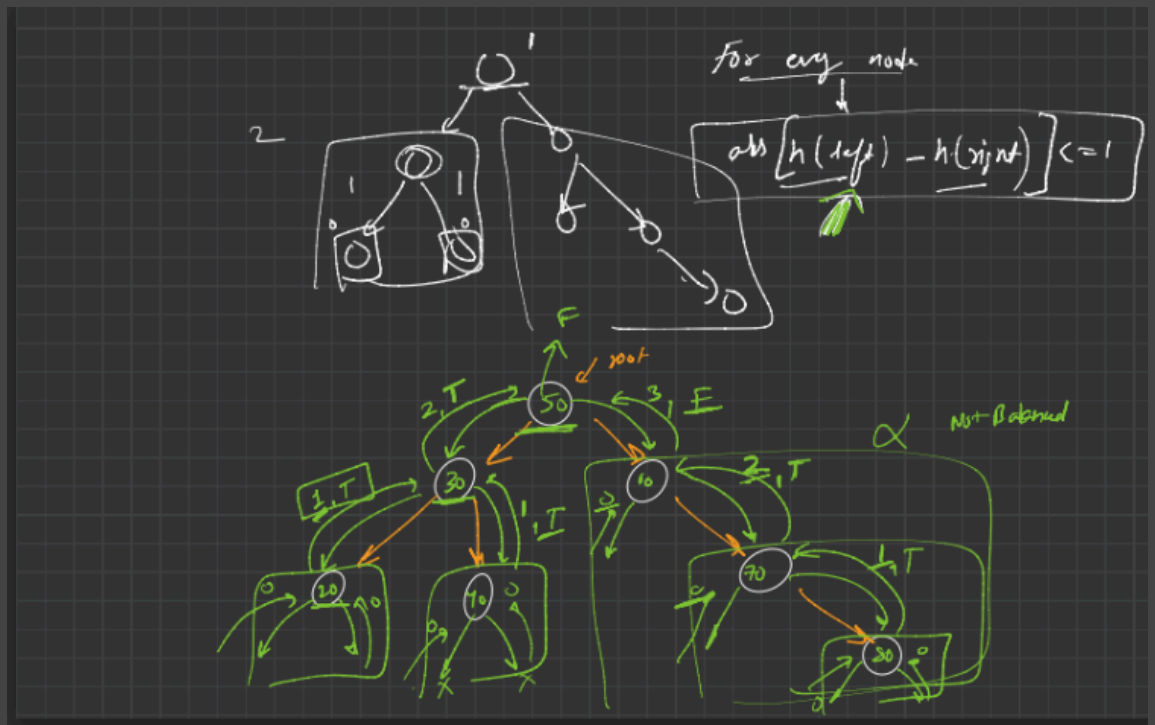


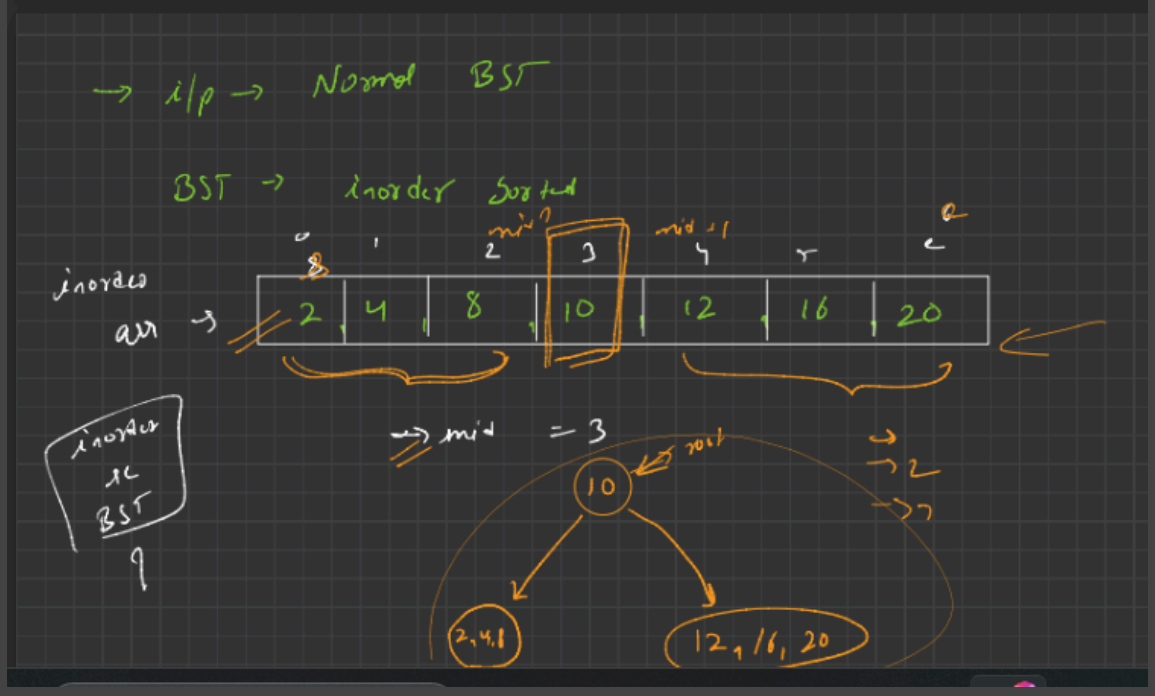


Flatten BST To A Sorted List:

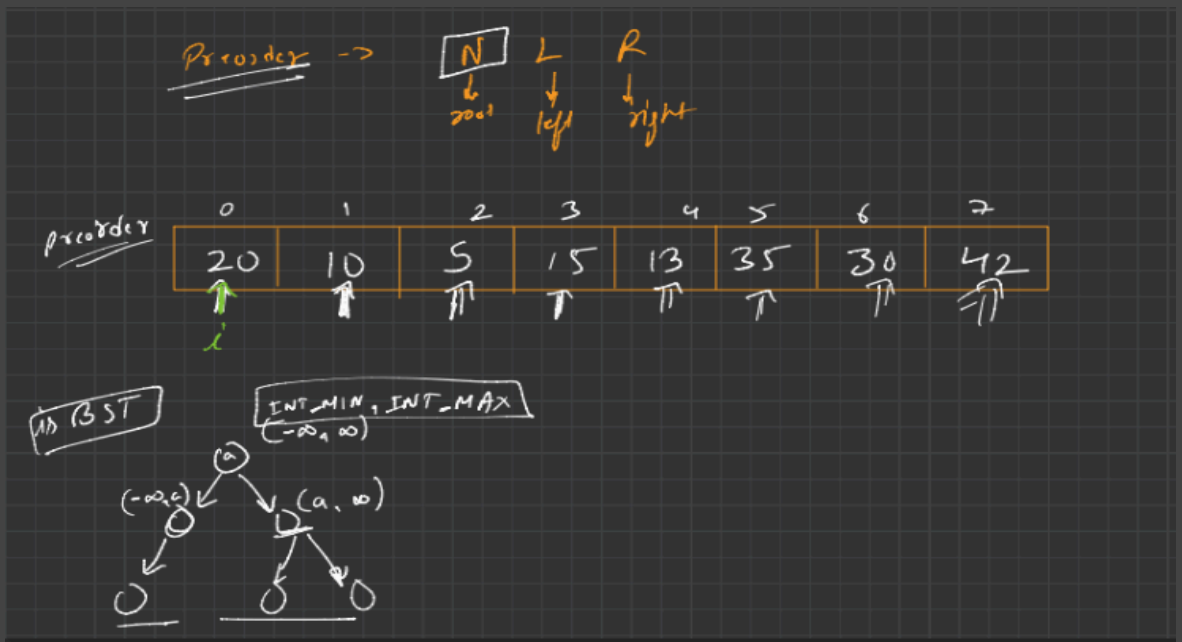


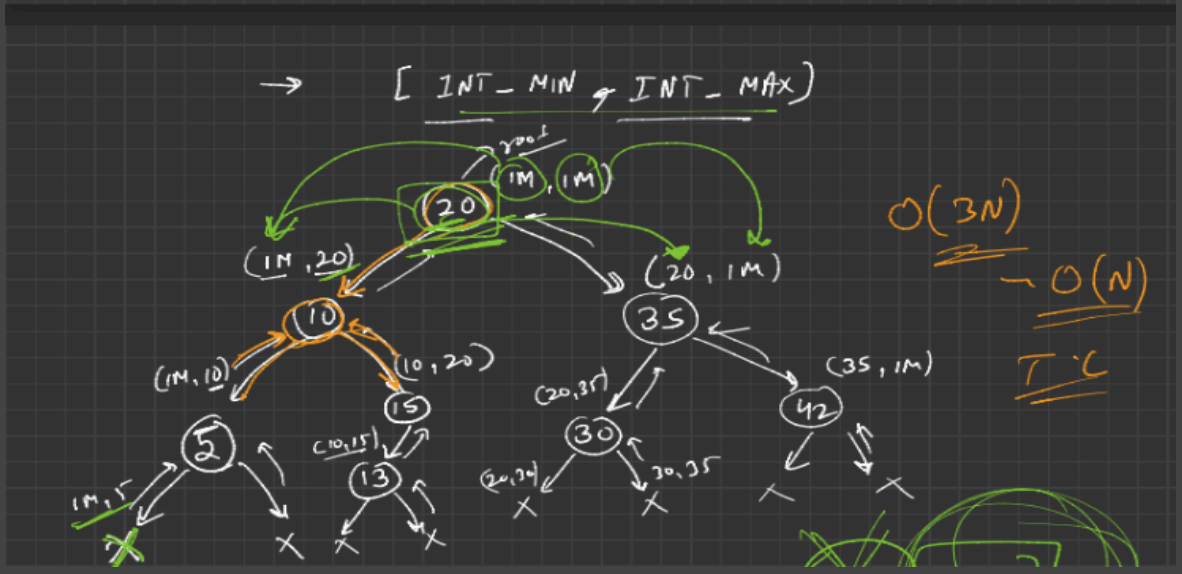
Normal BST To Balanced BST:



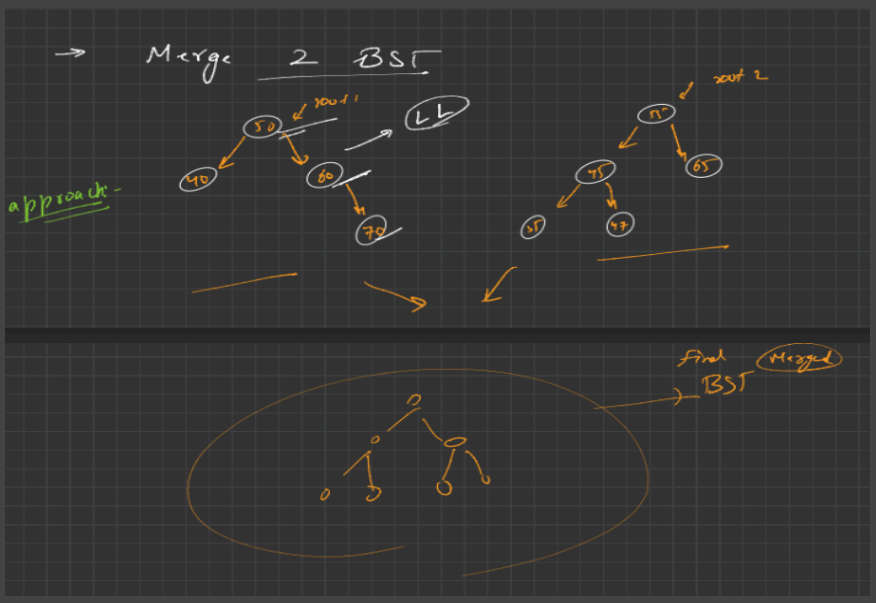


BST From Preorder Traversal:





Merge Two BST:



approach #1

Inorder To BST

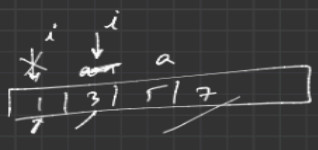
↳ i/p → BST 1 , BST 2

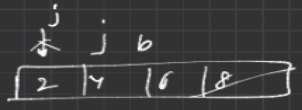
step 1 → vector → inorder 1 (m) , inorder 2 (n) [store inorder]

step II → Merge 2 sorted arrays → arr3 = [sorted, m+n]

step III → inorder to BST → ans

T.C → O(m+n)
S.C → O(m+n)





ans (a+b)

⇒ O(m+n) ✗

S.C →

S.C → O(h1 + h2)

Approach #2

↳ i/p → BST 1 Root 1 , BST 2 Root 2

convert BST into sorted LL step 1

sorted LL , sorted LL

T.C → O(m) O(n)
S.C → O(h1) O(h2)

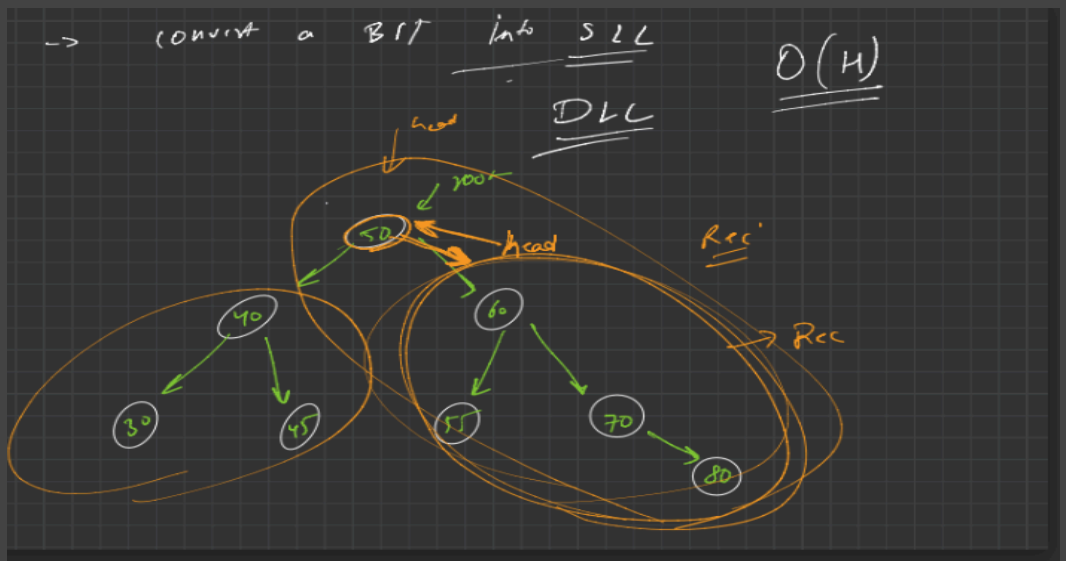
step 2 → merge 2 sorted Linked List → T.C → O(m+n), S.C → O(1)

1 sorted LL

step 3 → sorted LL → BST → S.C → O(h1+h2)

Balanced BST

① convert a BST into sorted LL → O(n) → O(h)
② Merge 2 sorted Linked List
③ sorted LL → BST

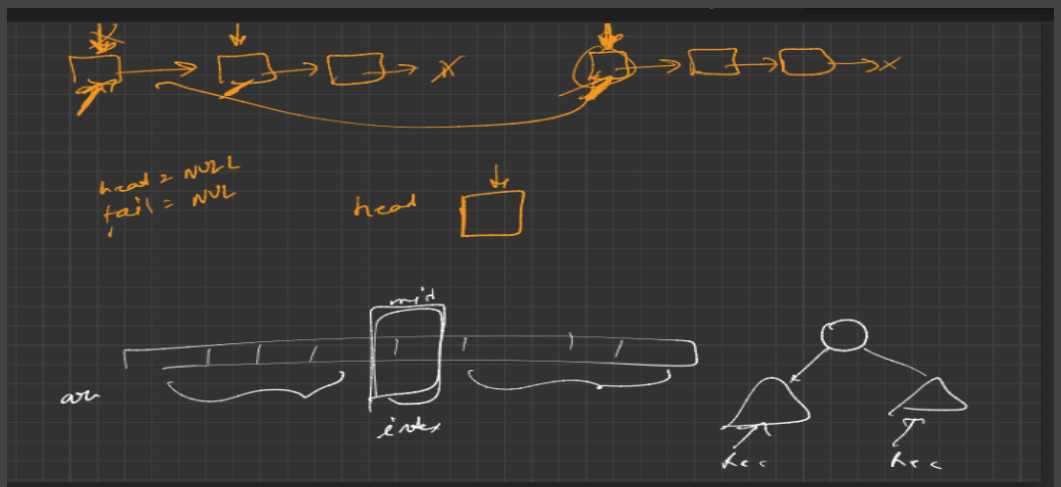


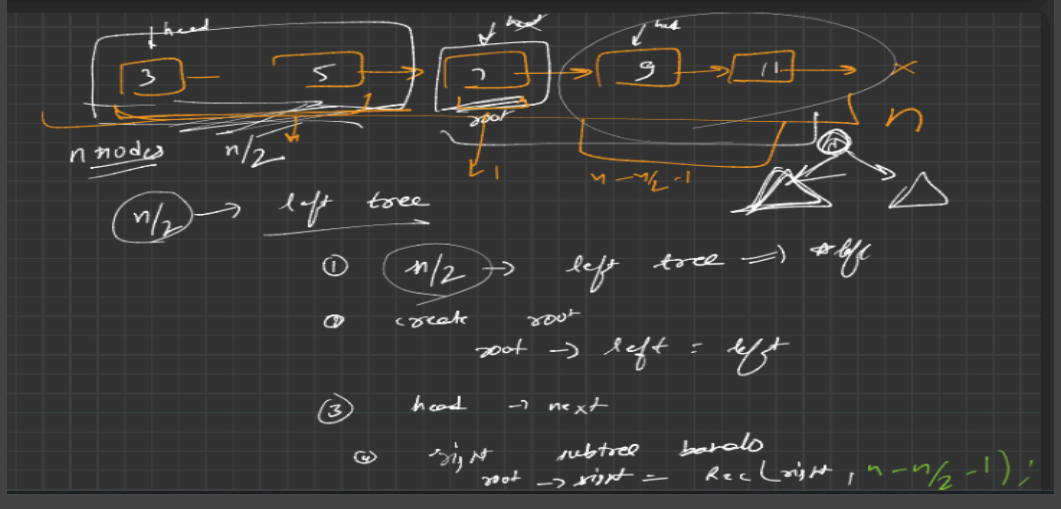
algo -> convert right subtree into LL -> (head of LL)

-> root -> right = head
head -> left = root

-> head = root

-> convert left subtree into LL -> head of LL



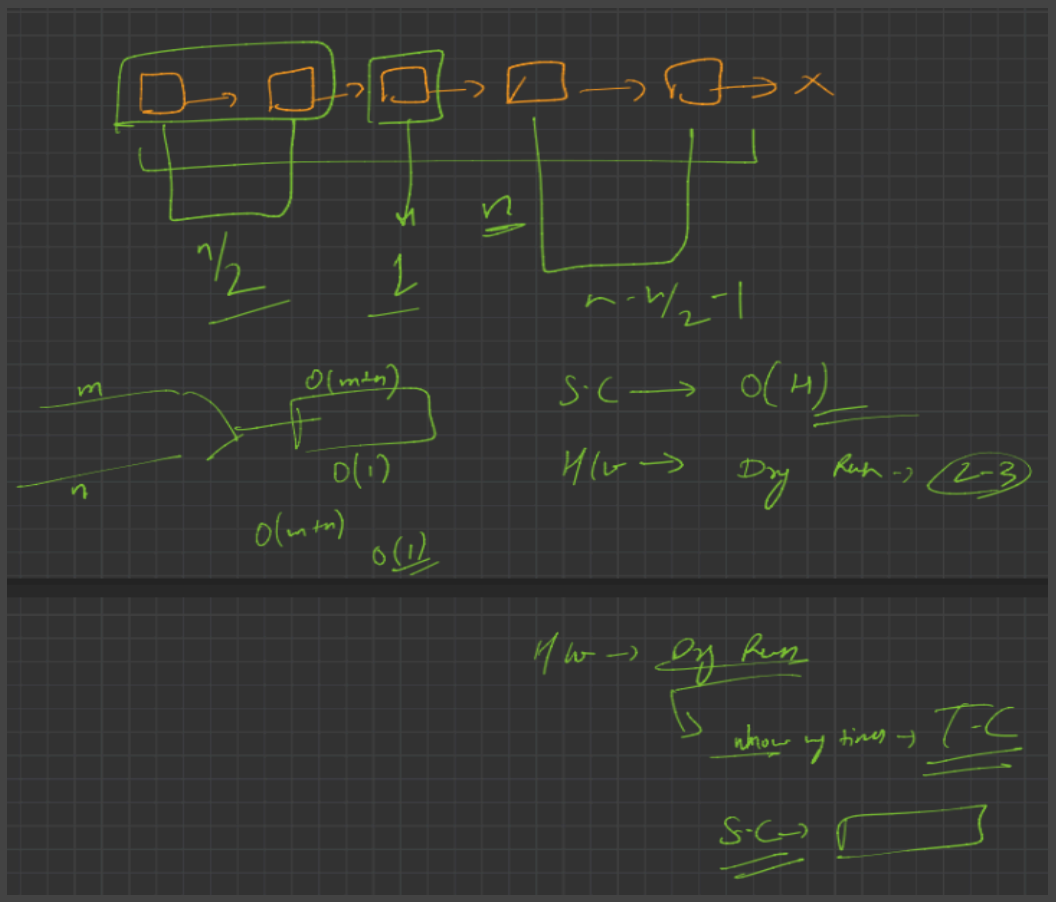


① n/2 -> left tree => left

② create root
root -> left = left

③ head -> next

④ right subtree banaLO
root -> right = Rec(right, n - n/2 - 1);



H/w → Dry Run

↳ whose ty time → T.C

S.C →