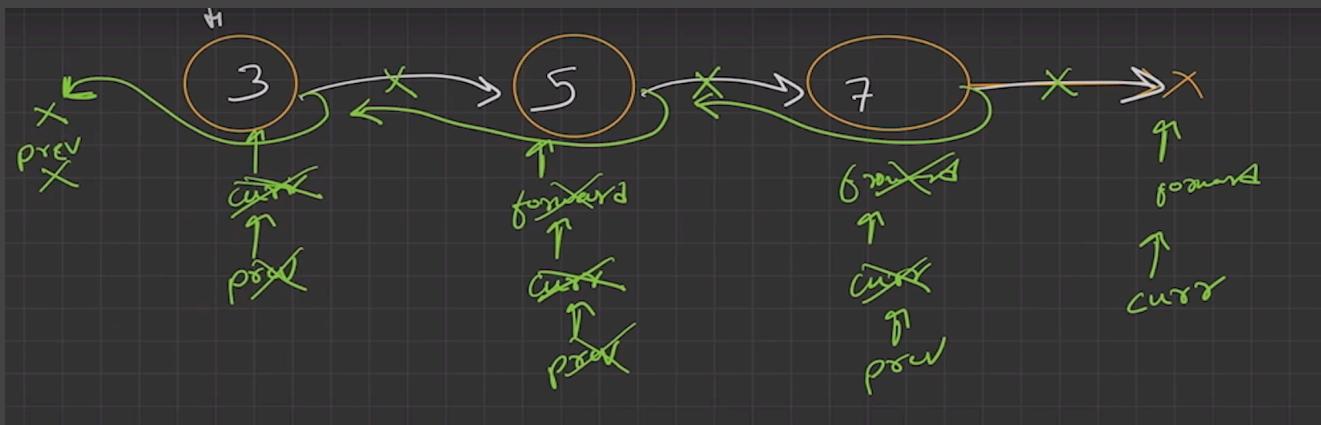
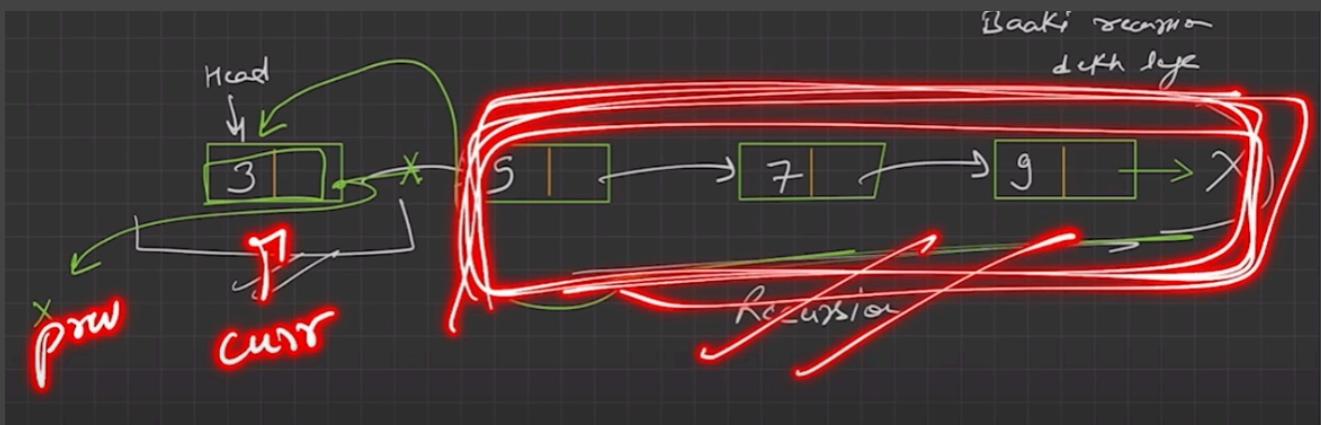


Linked-List

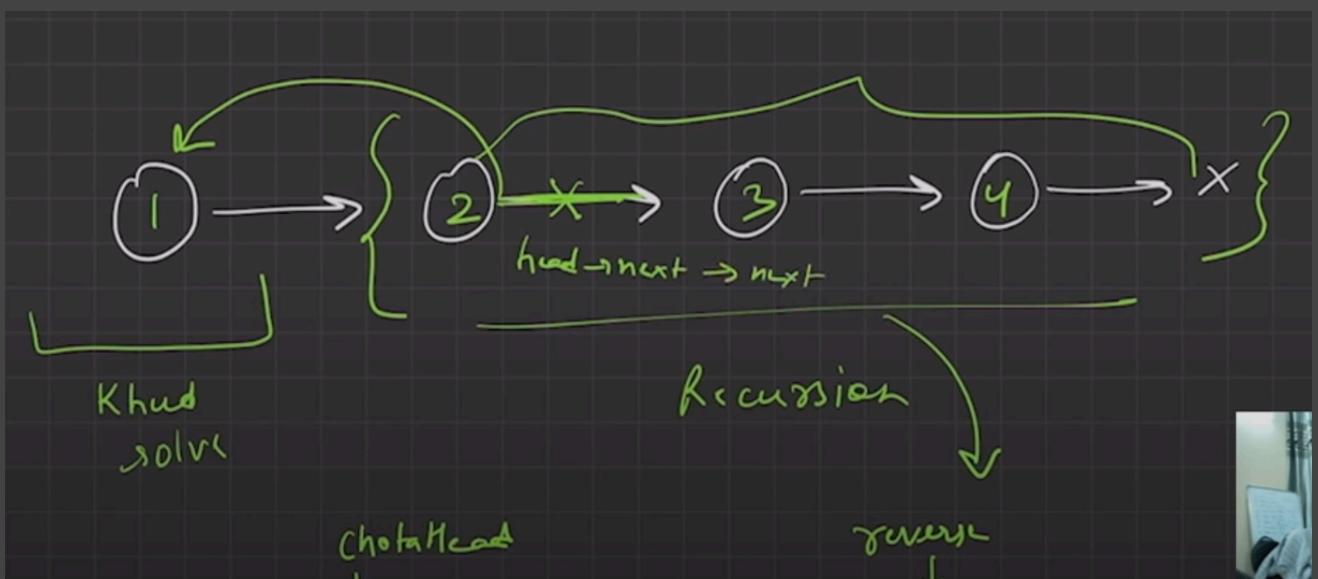
Reverse A Singly Linked List Approach-I:



Reverse A Singly Linked List Approach-II:

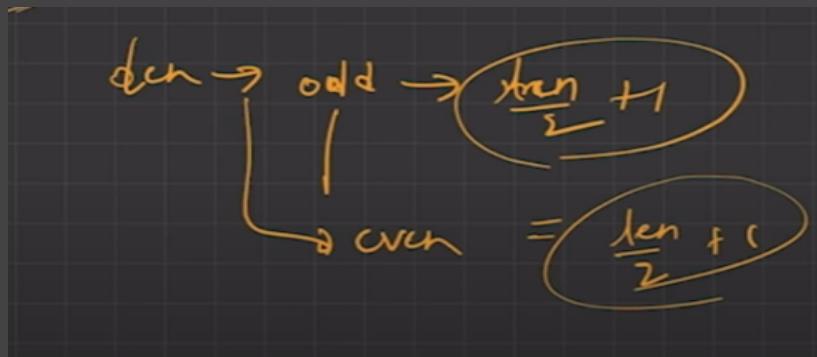
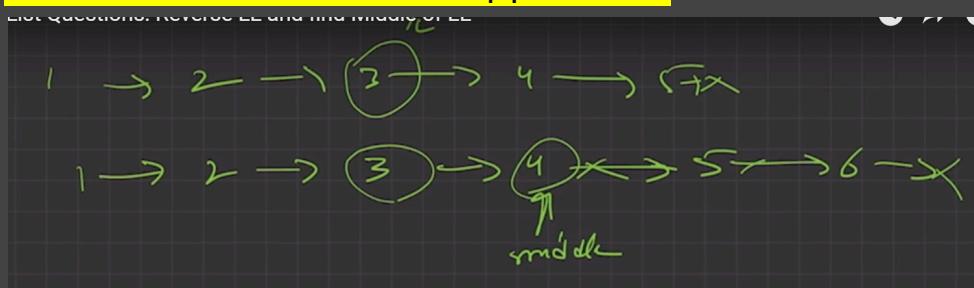


Reverse A Singly Linked List Approach-III:

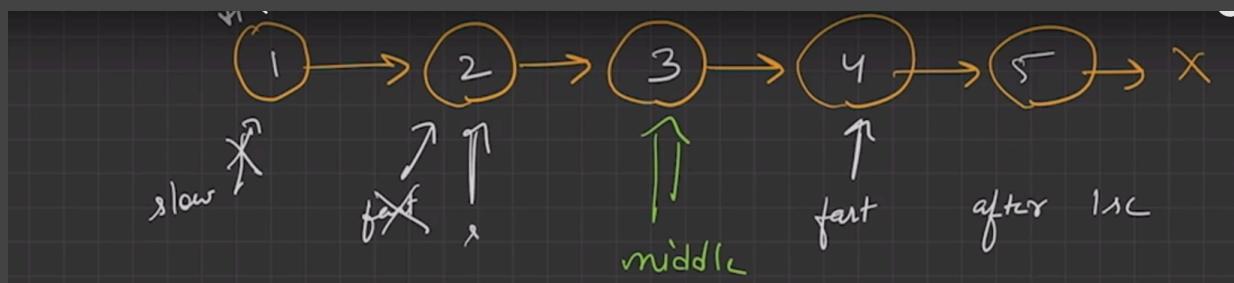


Reverse A Doubly Linked List

Find Middle In A Linked List Approach-I:



Find Middle In A Linked List Approach-II:



① if \rightarrow empty list \rightarrow return `NULL`



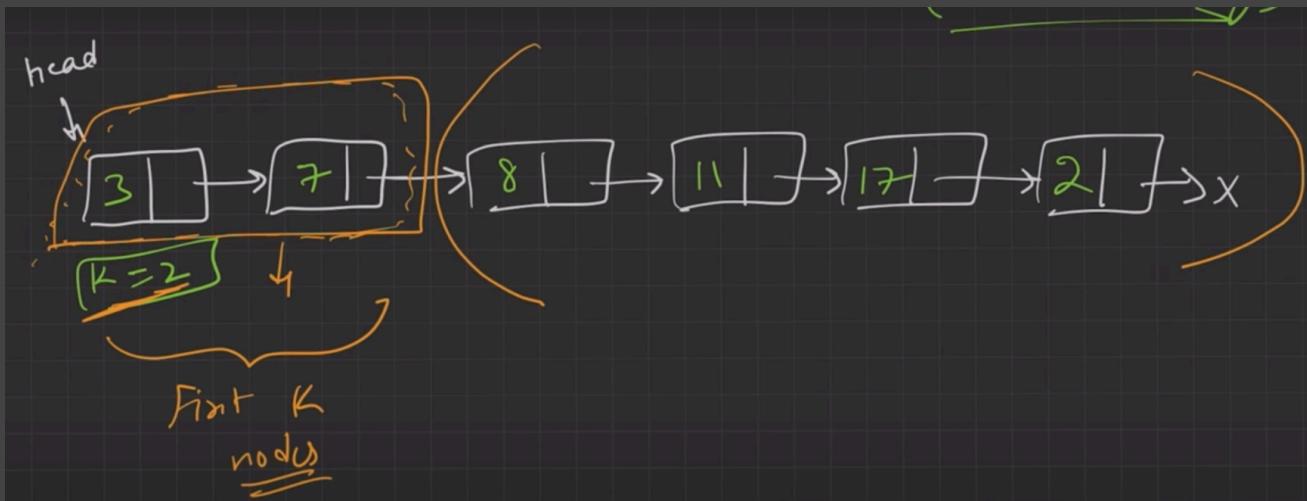
② 1 node \rightarrow head

③ 2 nodes \rightarrow head \rightarrow next

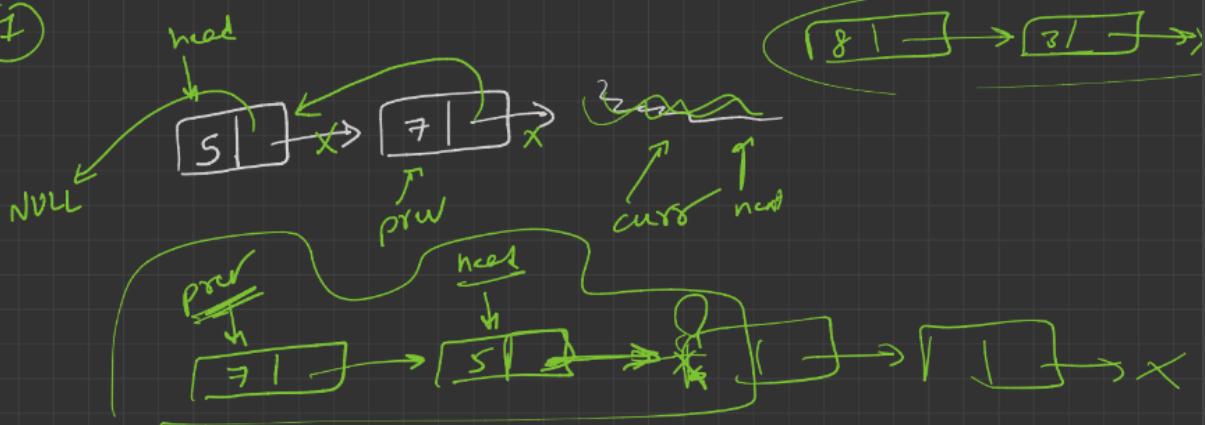
④ Algo



Reverse Linked List In 'K' groups:



(i)



(ii)

$head \rightarrow next \rightarrow$ Recursion call

(iii)

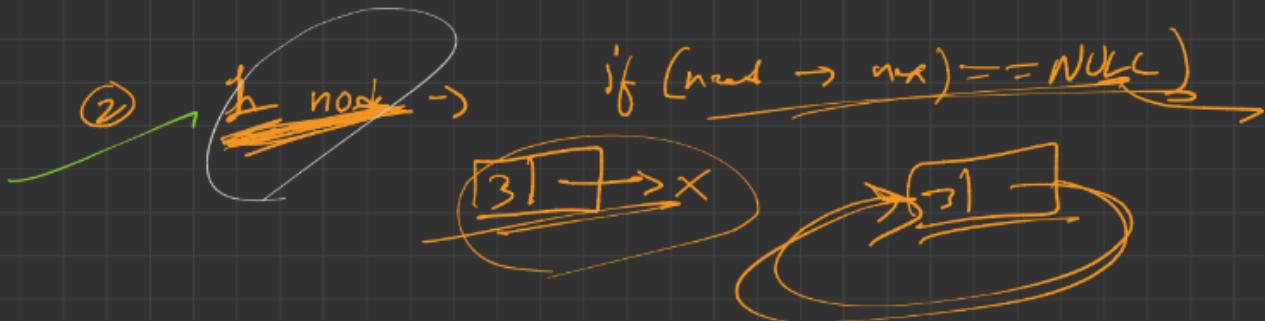
return prw

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

Approach 1 :-



(1) Empty List \rightarrow if (head == NULL)
return True



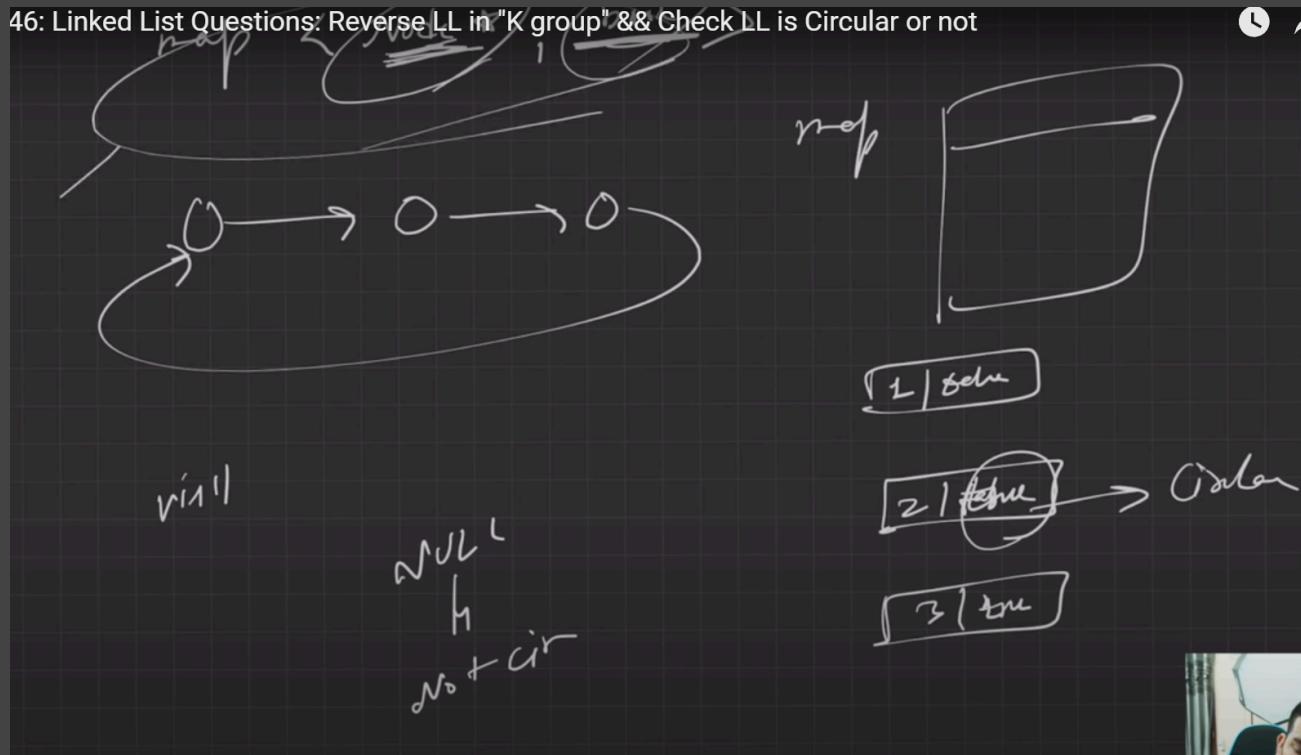
(2) Non-empty List \rightarrow if (head->next == head)
return True



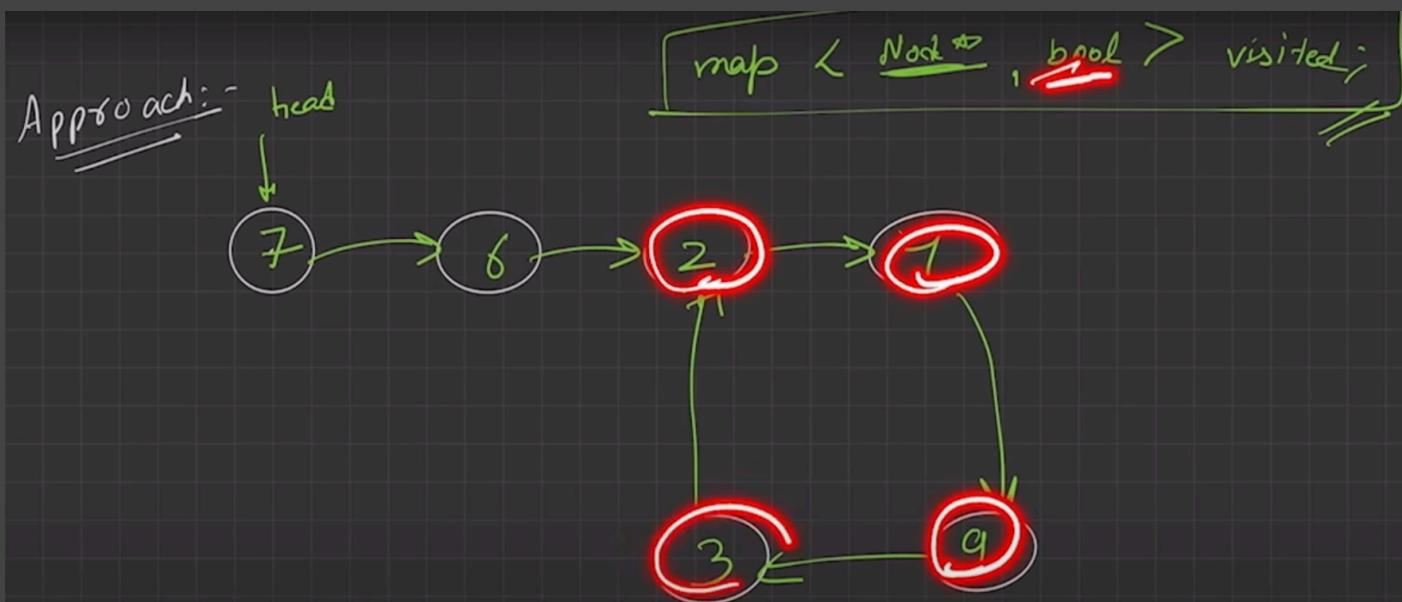
L
 \rightarrow n

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

46: Linked List Questions: Reverse LL in "K group" && Check LL is Circular or not

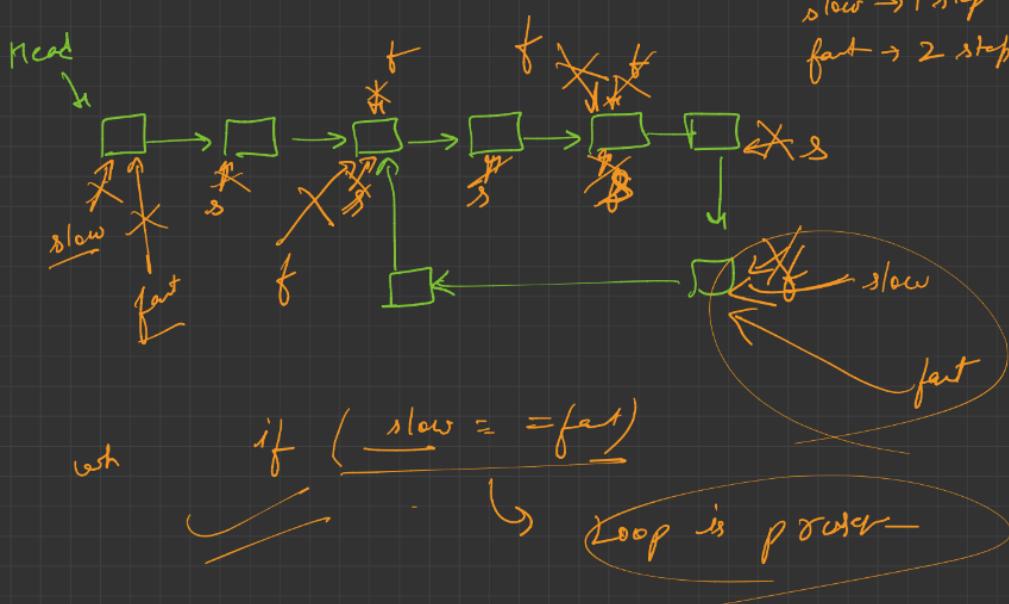


Detect Whether A Linked List Forms a Loop or Not:

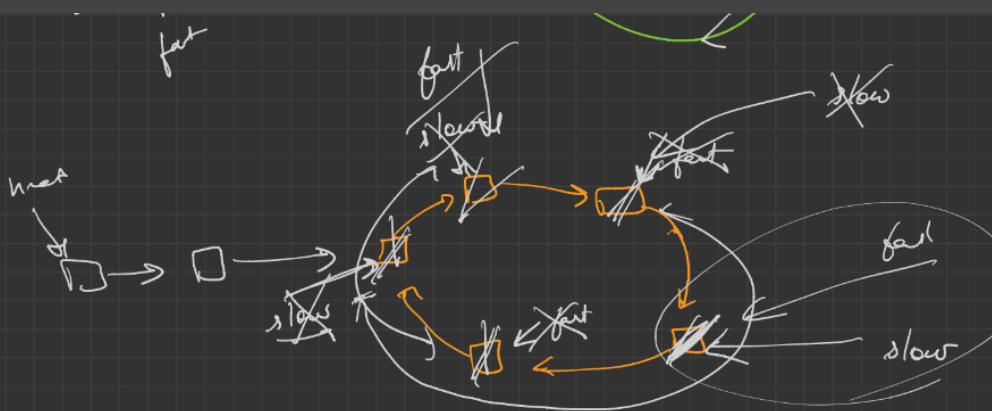


Floyd's Cycle Detection:

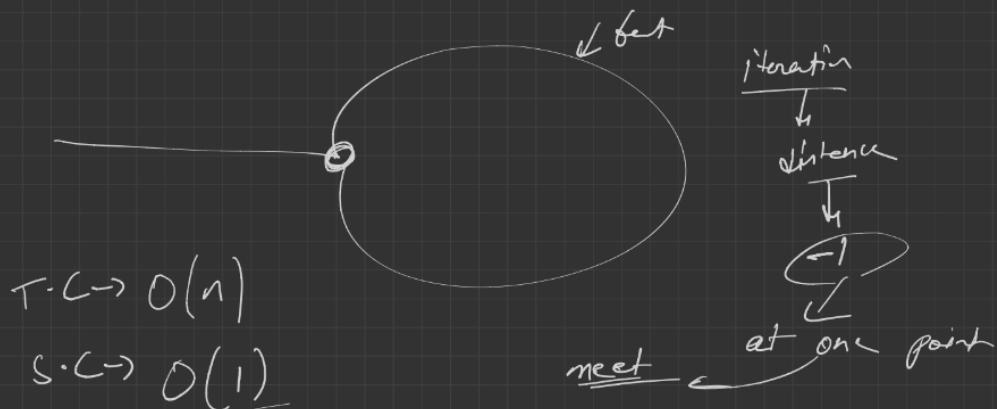
① Floyd's cycle detection algo.



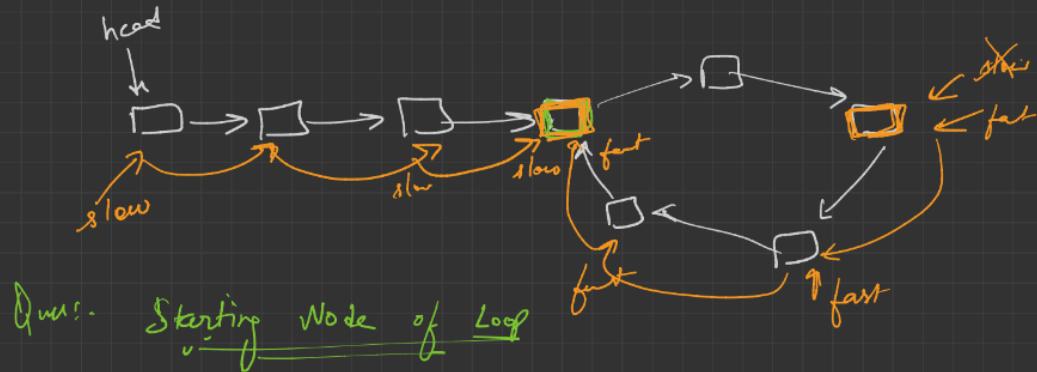
② fast = NULL → No loop



Distance: 4, 3, 2 1



Find The Starting Node Of The Loop:



Approach:-

I \rightarrow FCD Algo \rightarrow Point of Intersection

II \rightarrow $slow = head$
 $slow, fast \rightarrow$ same pace
 / / /

when ($slow == fast$)

\hookrightarrow [start] point of Loop

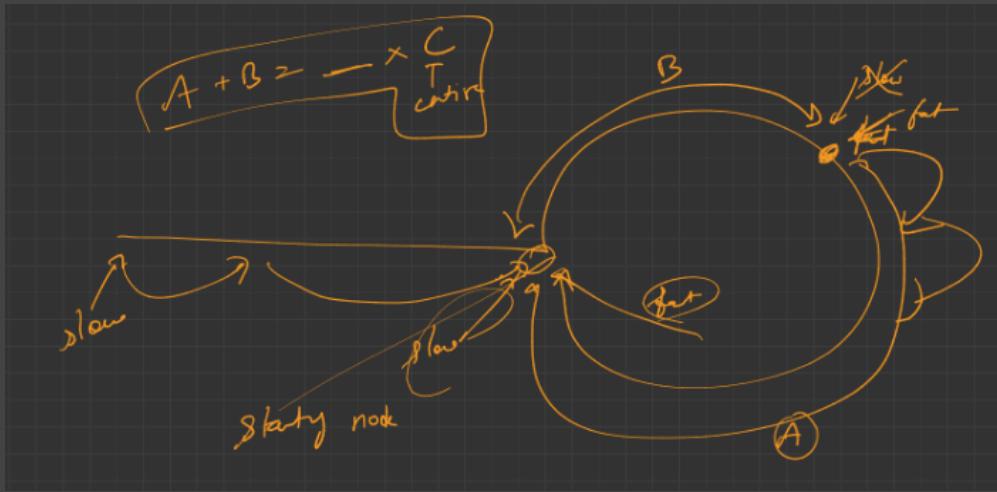
Distance by $= 2 \star$ Distance by slow pointed
 fast pointer

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

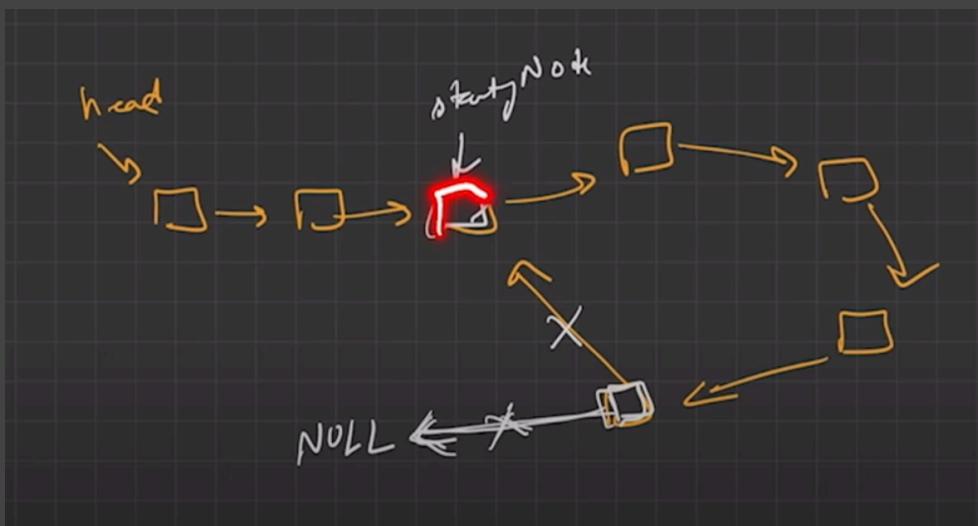
$$A + nC + B = 2A + 2yC + 2B$$

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

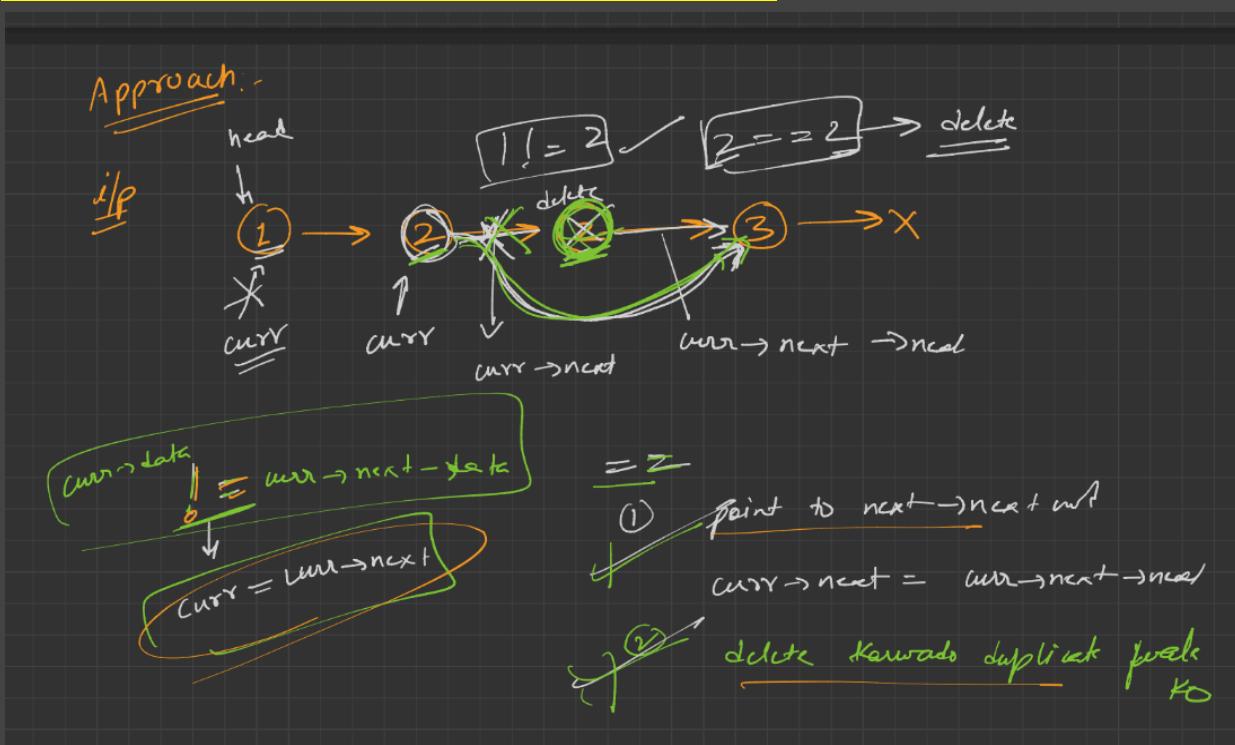
$$\frac{A + B}{K} = \frac{K \text{ times } C}{K}$$



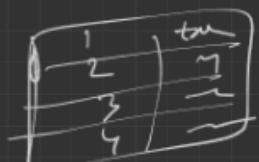
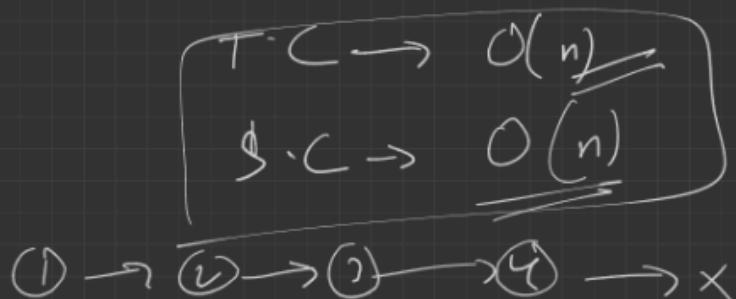
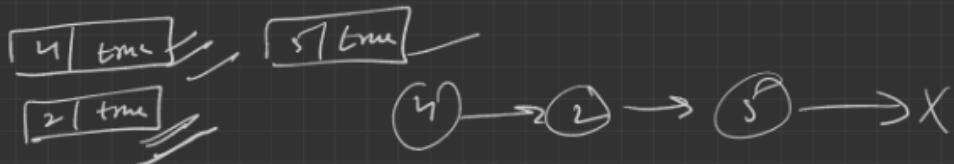
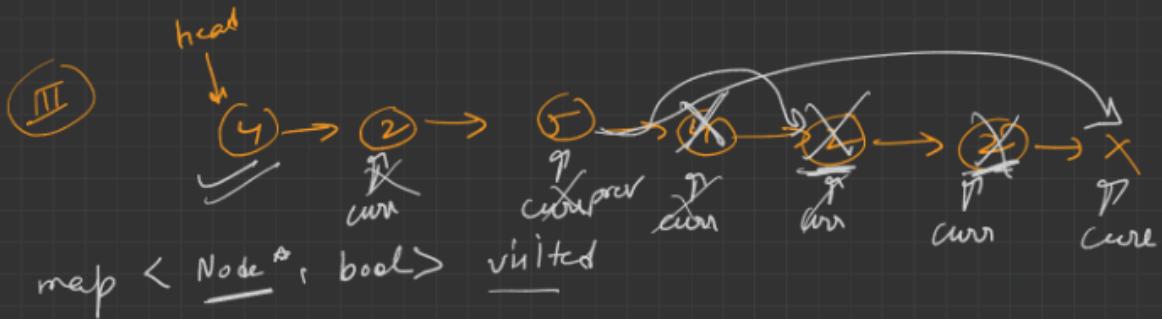
Remove The Loop:



Remove Duplicates From A Sorted Linked List:



Remove Duplicates From A Unsorted Linked List:

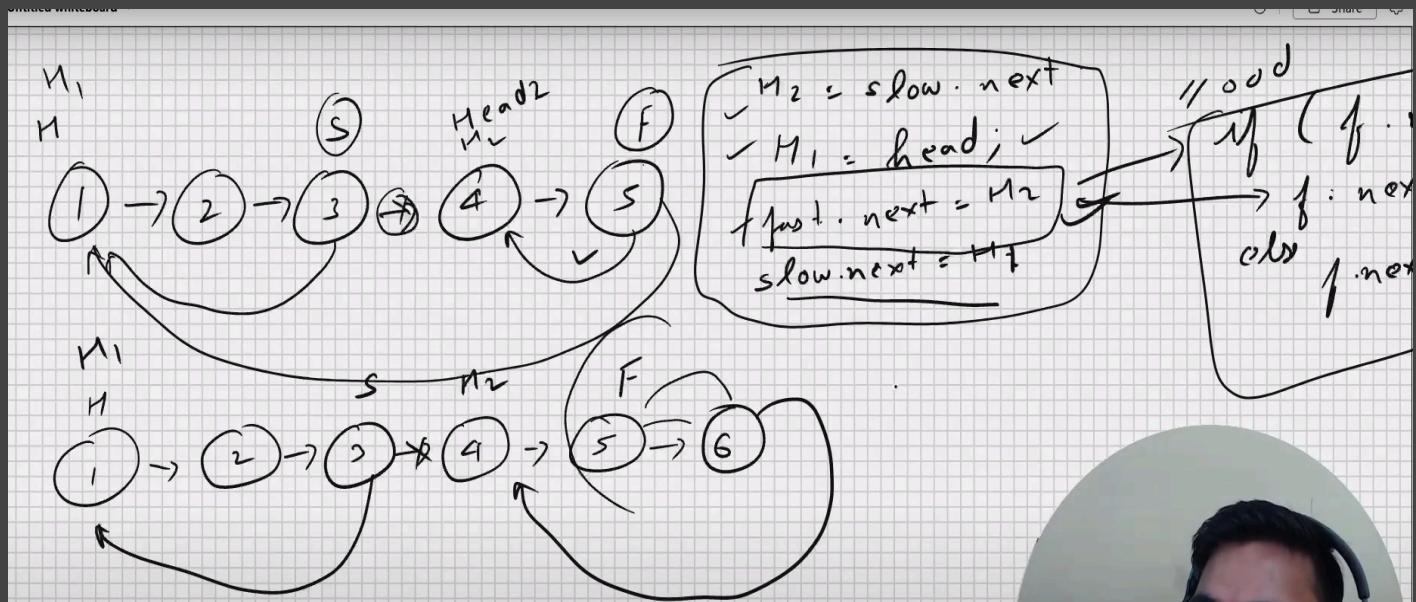


Remove Duplicate from Unsorted LL

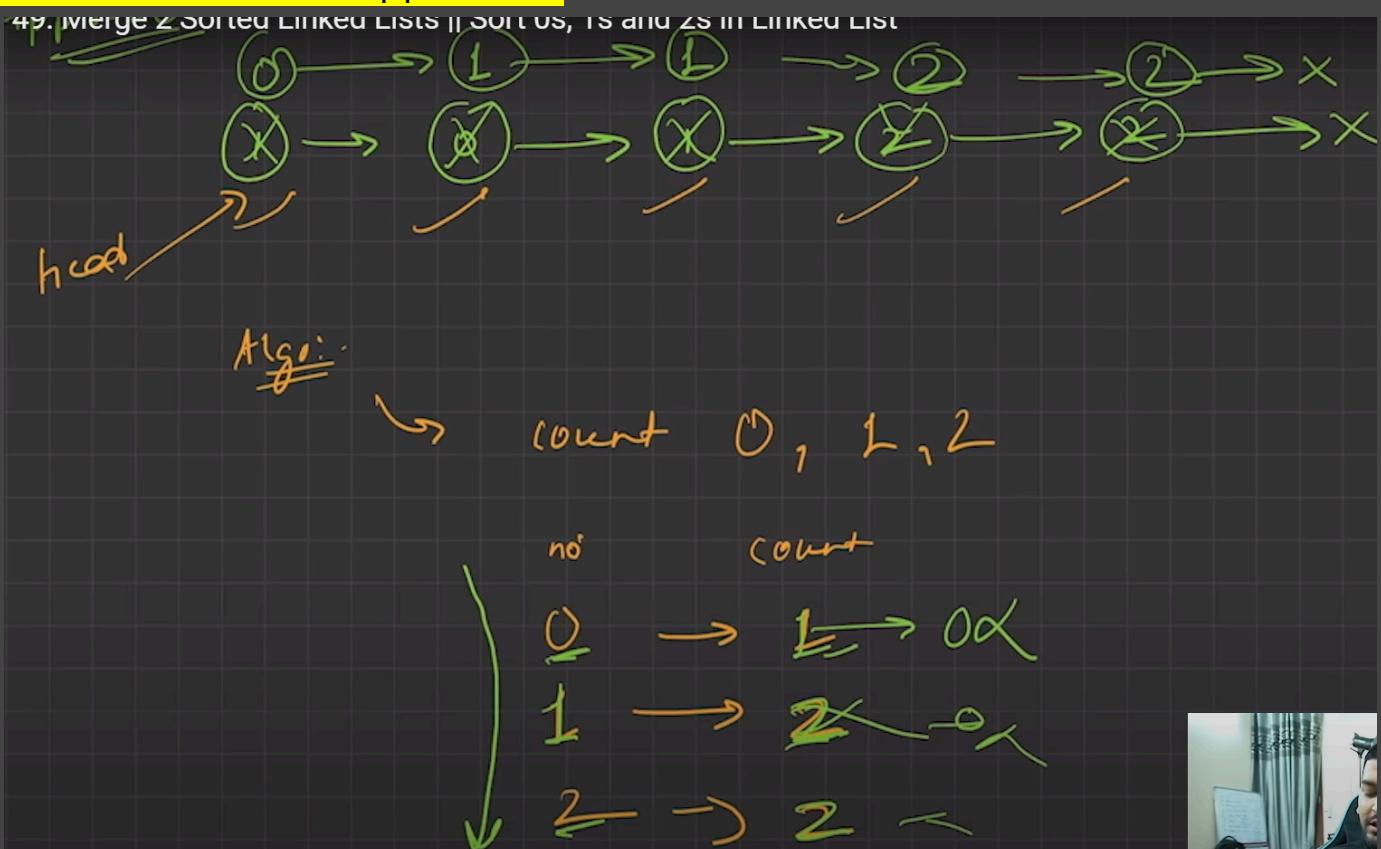
Homework

$O(n)$ \rightarrow 2 Loops
 $O(n \log n)$ \rightarrow sort \rightarrow $O(n)$ when
 \Rightarrow map \rightarrow $O(n)$

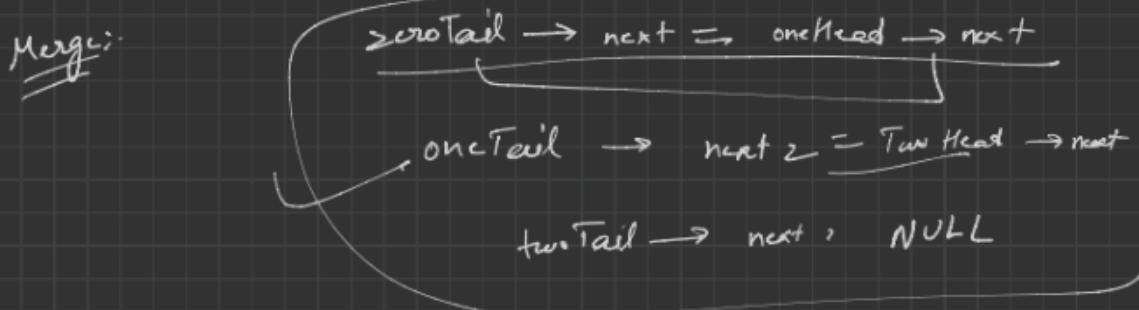
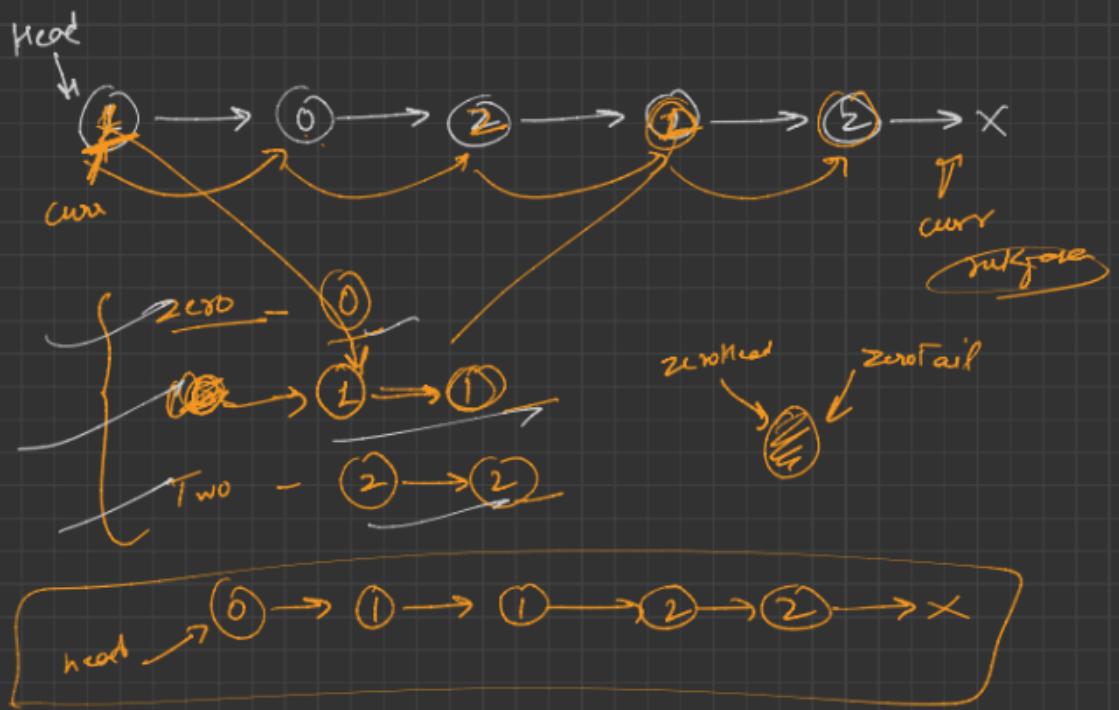
Split A Circular LL Into Two Halves:



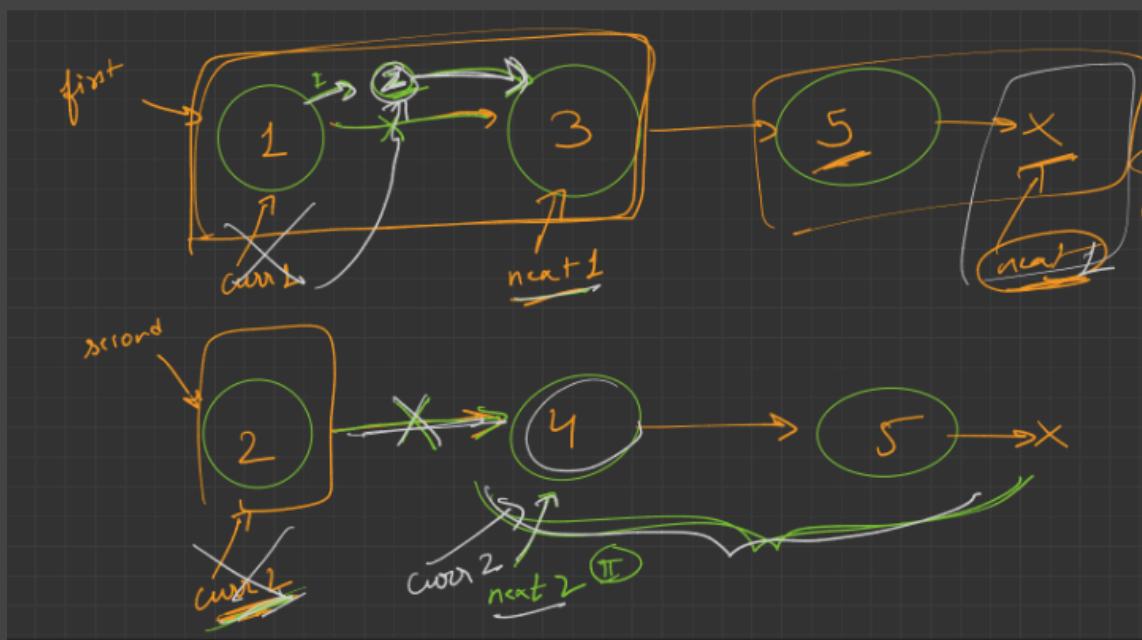
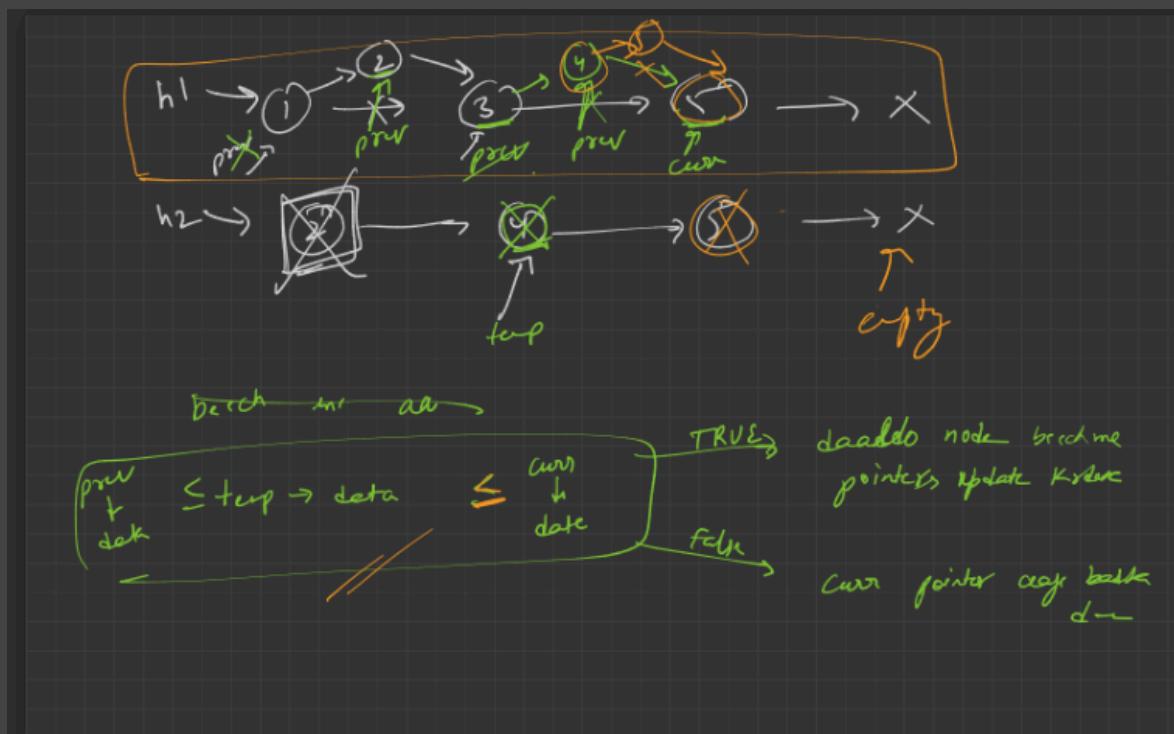
Sort 0s, 1s & 2s In A LL Approach-I:



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



Merge Two Sorted LL:



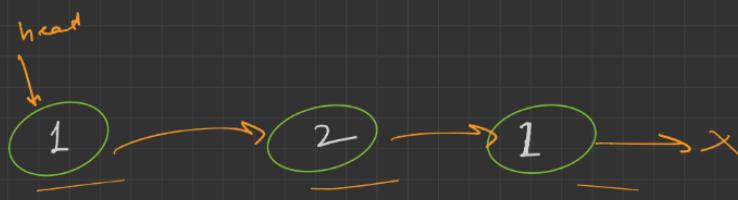
$$\left. \begin{array}{l}
 I \rightarrow curr1 \rightarrow next = curr2 \\
 II \quad next = curr2 \rightarrow next \\
 III \quad curr2 \rightarrow next = next - 1
 \end{array} \right\}$$

$$(N) \Rightarrow curr1 = curr2$$

$$(v) = curr2 = nc$$

Check Palindrome In LL Approach-I:

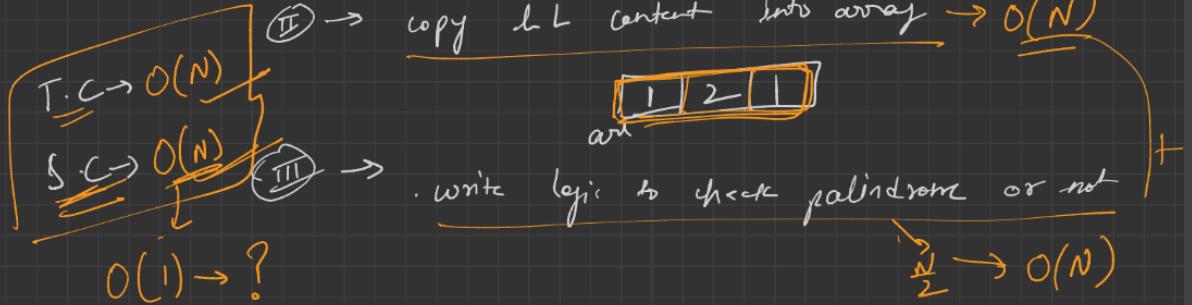
Approach #1



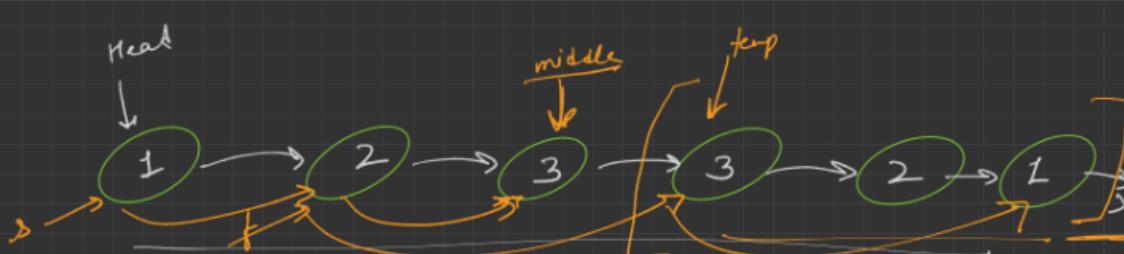
Algo:-

① → create an array

② → copy h L content into array $\rightarrow O(N)$



Check Palindrome In LL Approach-II:



Algo:-

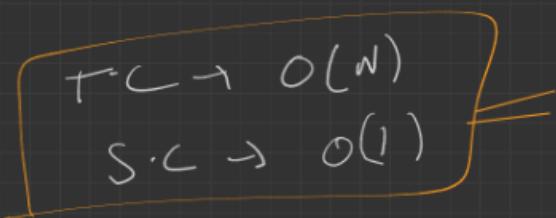
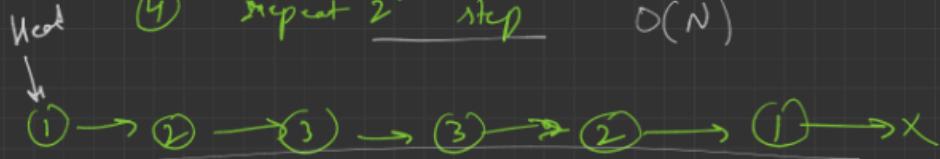
① Middle (find) $\rightarrow O(N)$

head ② reverse LL after it $O(N)$

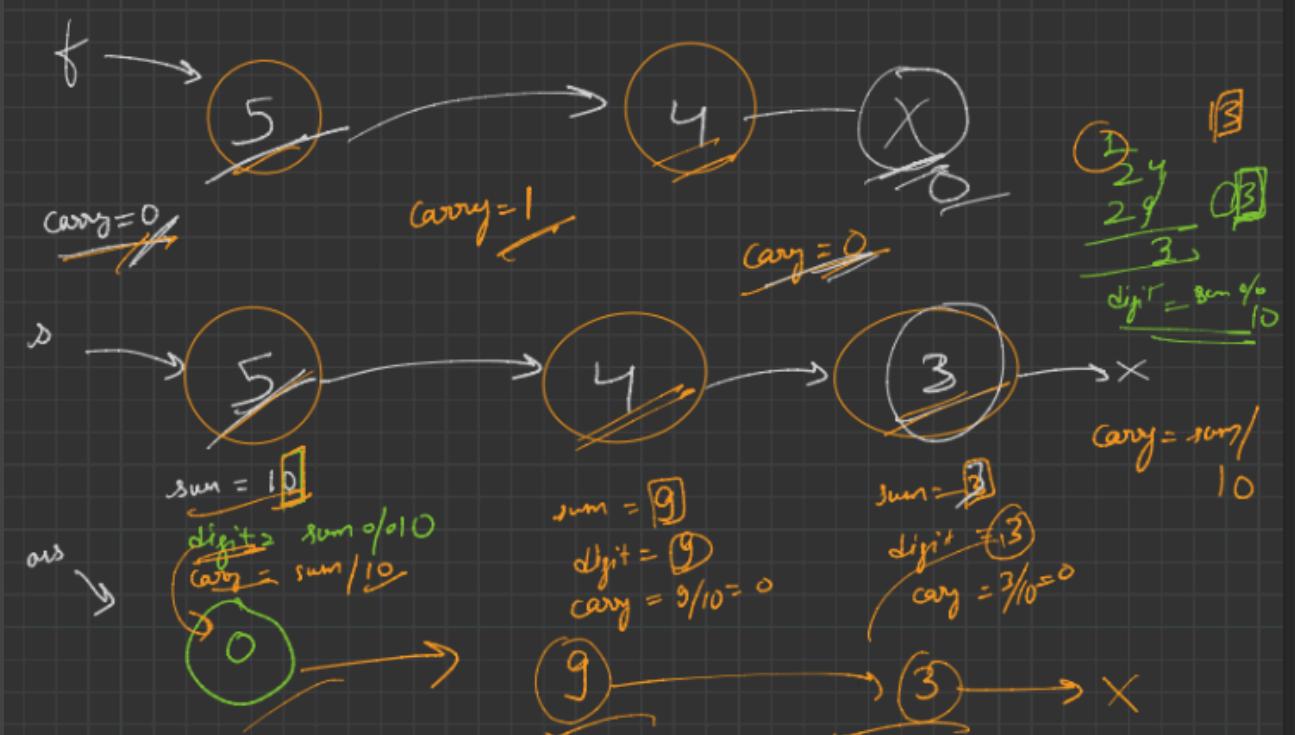
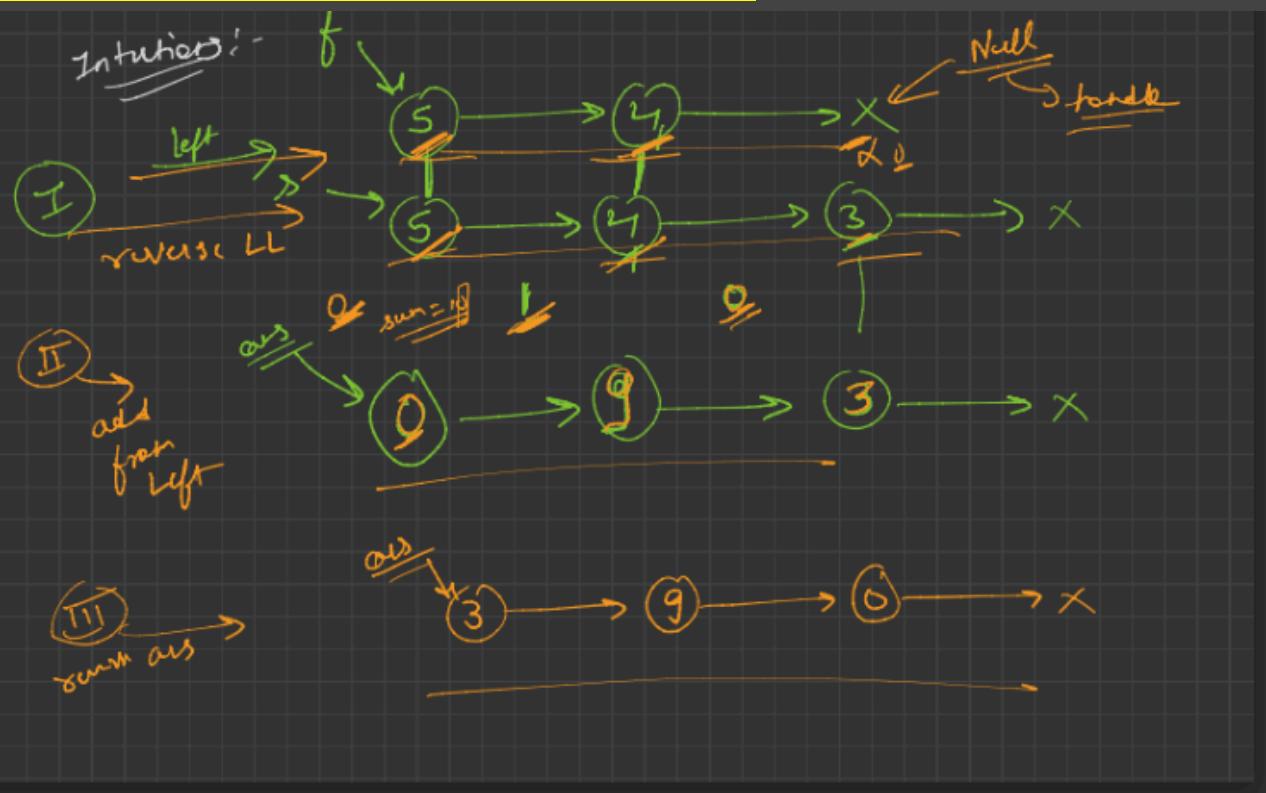


③ compare both halves PL or not $O(N)$

④ repeat 2ⁿ⁻¹ step $O(N)$



Add Two Numbers Represented By Linked List:

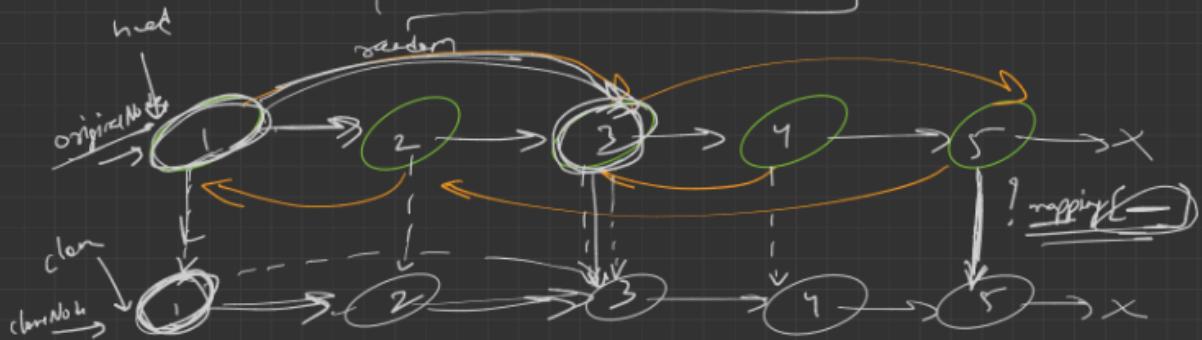


Clone A Linked List Approach-I:

Approach #2

→ (I) → create a clone List (using next ptr of Original List)

→ (II) Random ptr (copy) →



(II)

clone Node → random = ?

mapping [original Node → random]

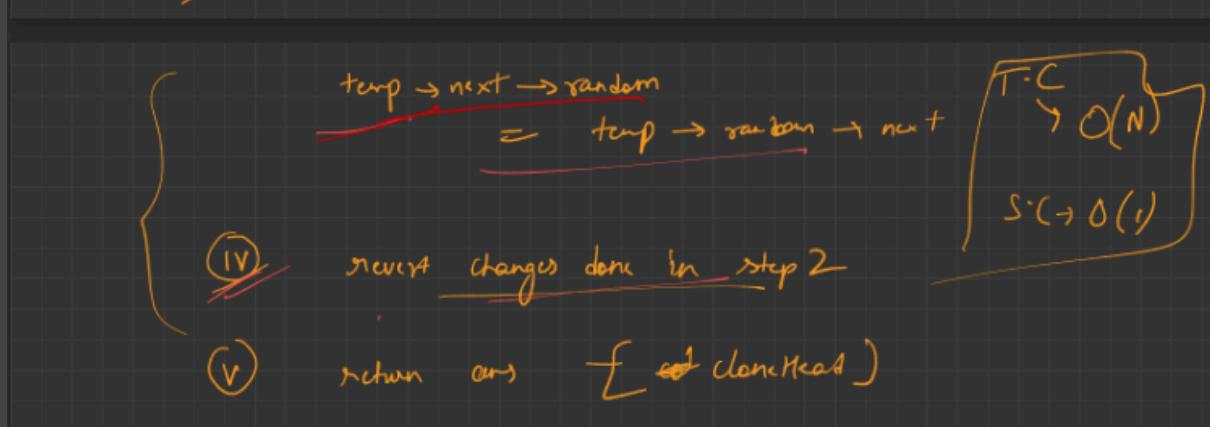
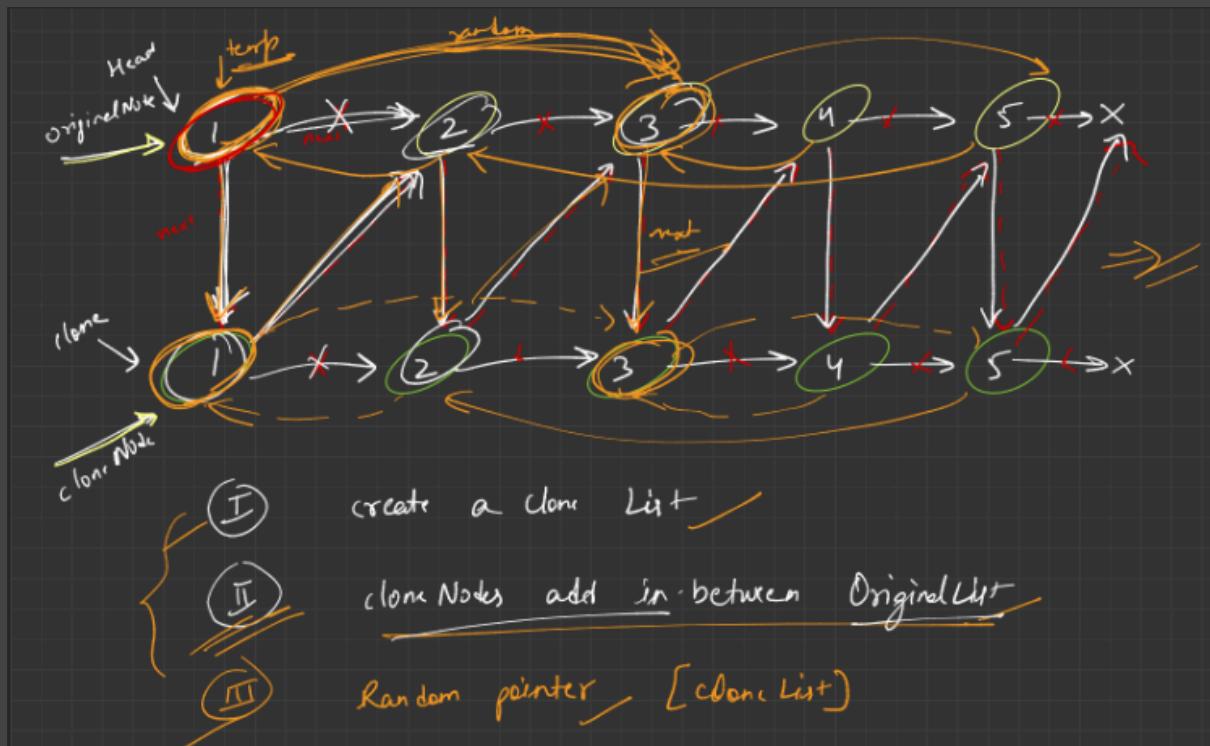
T.C → O(N)

S.C → O(N)

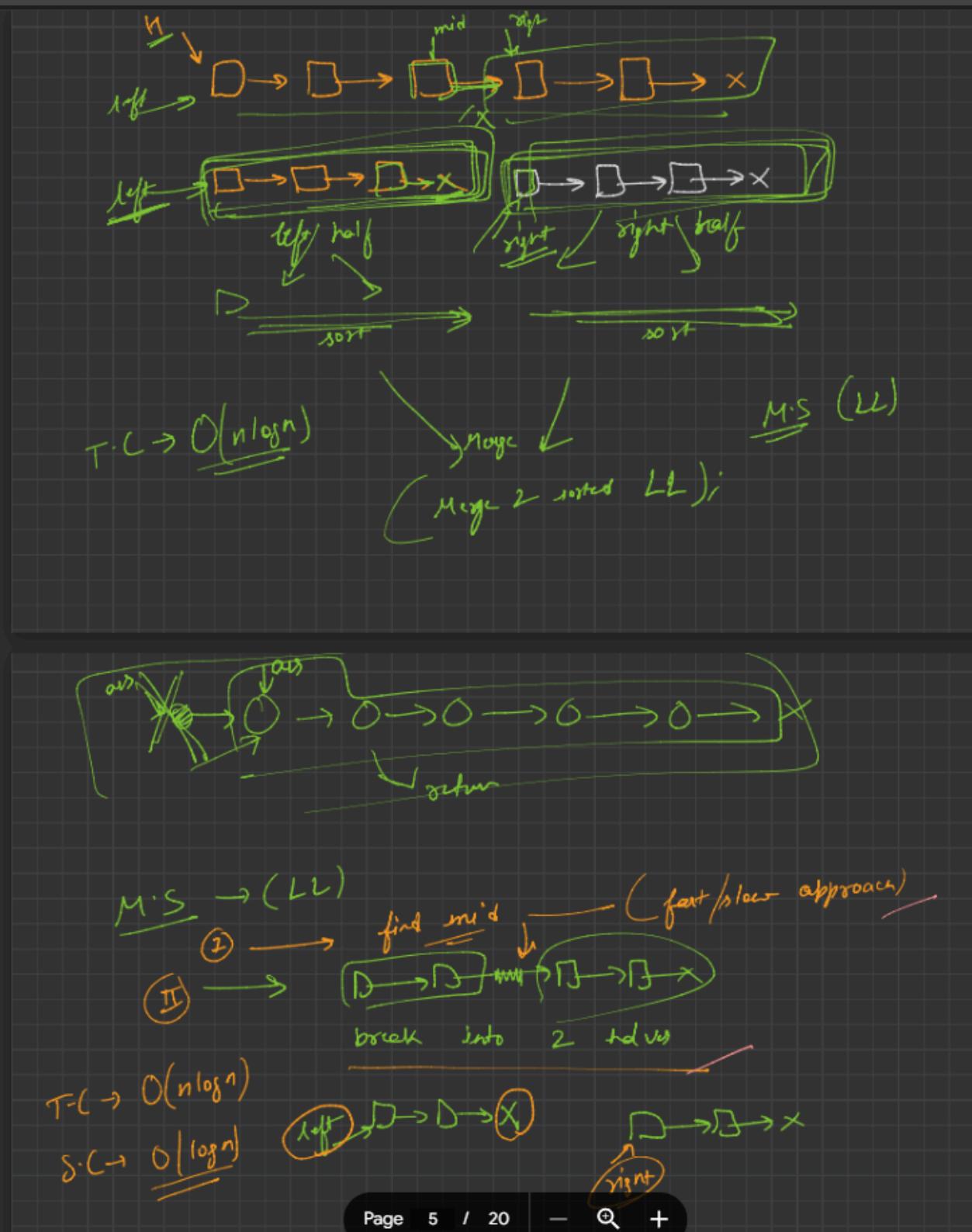
map ↘

O(1)

Clone A Linked List Approach-II:



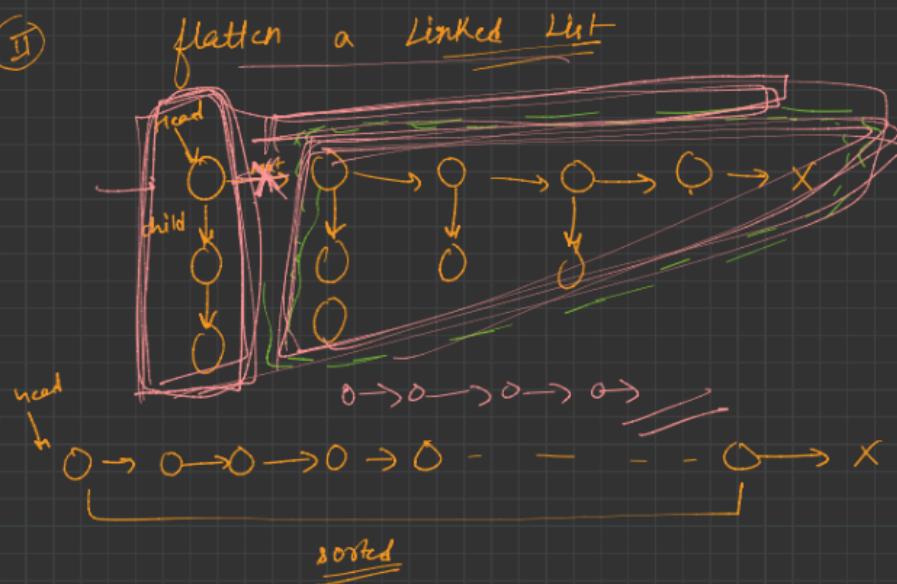
Merge Sort In Linked List:



- (III) Recursively sort left/right halves
- (IV) merge left/right sorted halves \rightarrow Merge 2 sorted LL
- (V) return merged list

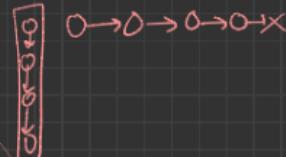
Flatten Linked List:

(II)



flatten (head)

Node * down = head;
down → next = NULL;
Node * right = flatten (head → right)



Merge 2 sorted
linked list

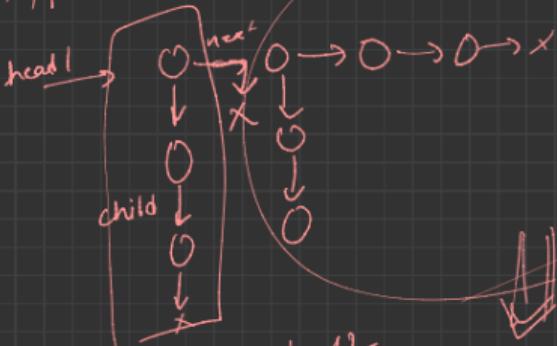
Exact
Order ↗

Node * ans = merge (down, right)

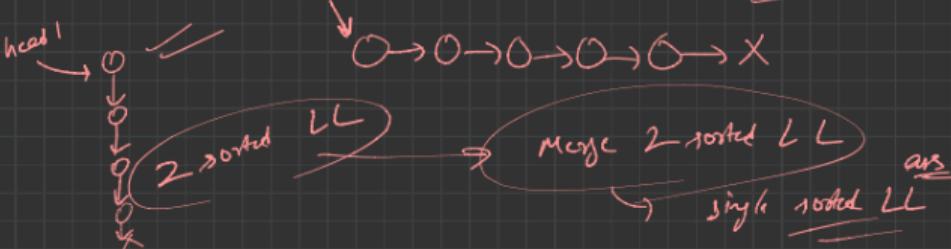
return ans;

1 | 2 | 3

i/p

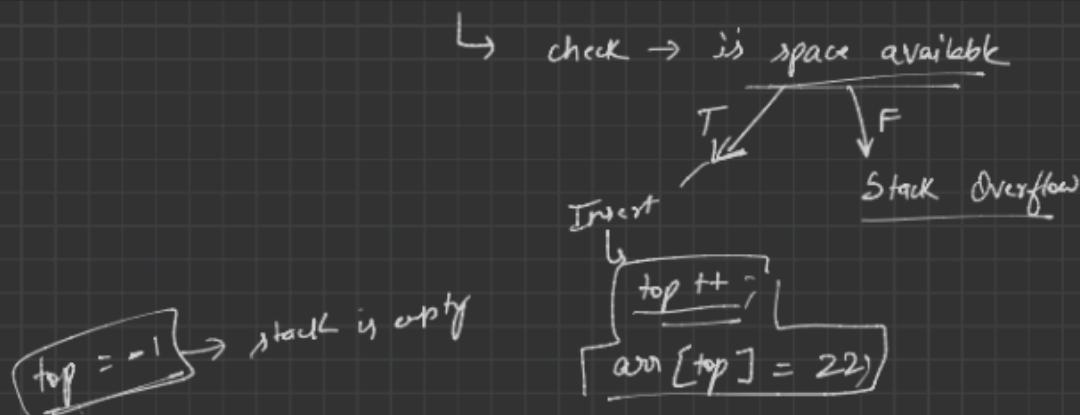
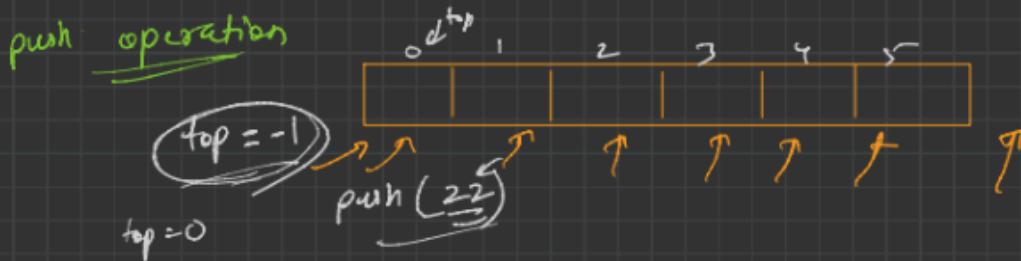
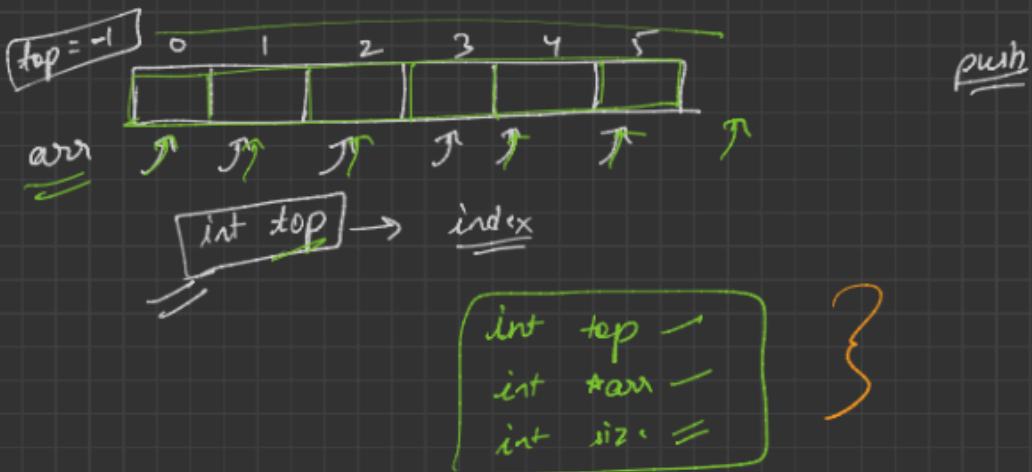
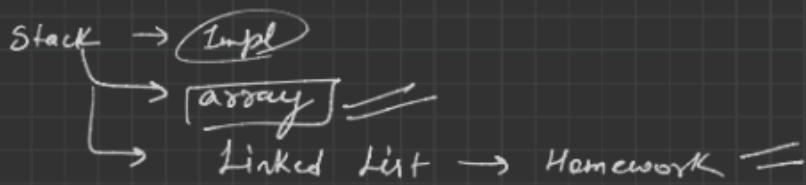


Recurr.

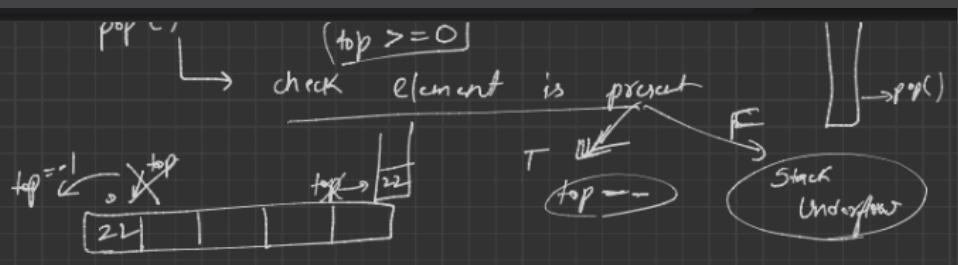


Stack

Create A Stack Using An Array:



Create A Stack Using A Linked List:



$\text{empty}()$

$\text{top} = -1 \rightarrow \text{stack empty}$

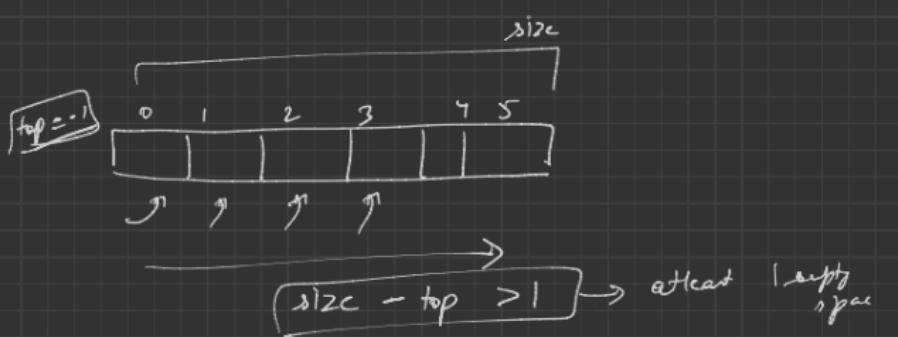
$\text{top}()$

$\text{top} \rightarrow 0$

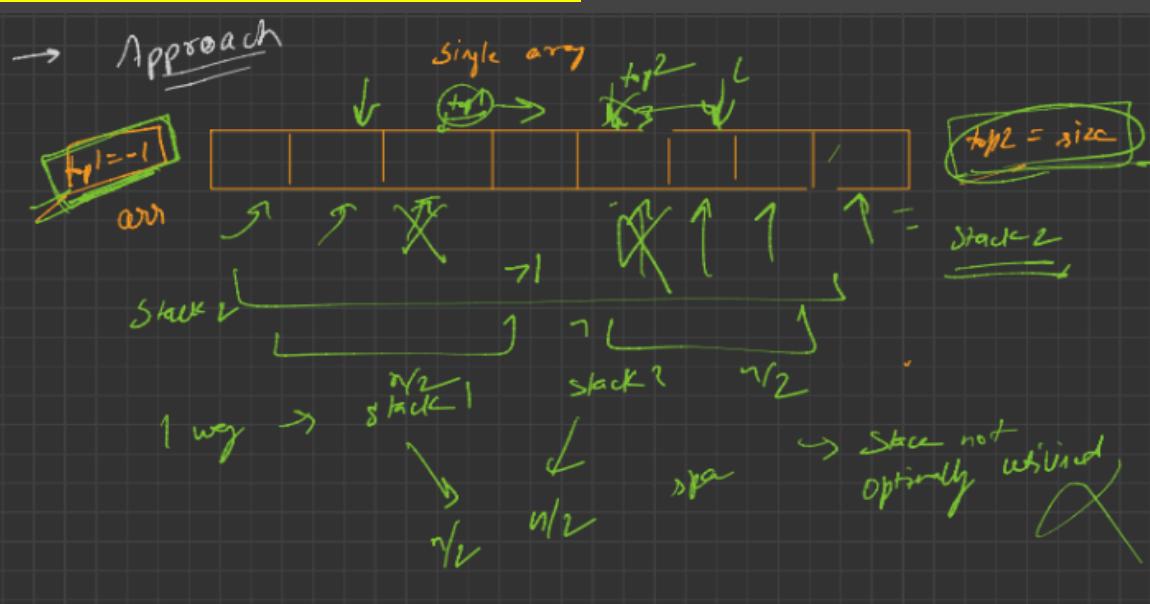
$\Rightarrow \text{return arr}[\text{top}]$

$\text{top} = -1 \rightarrow$

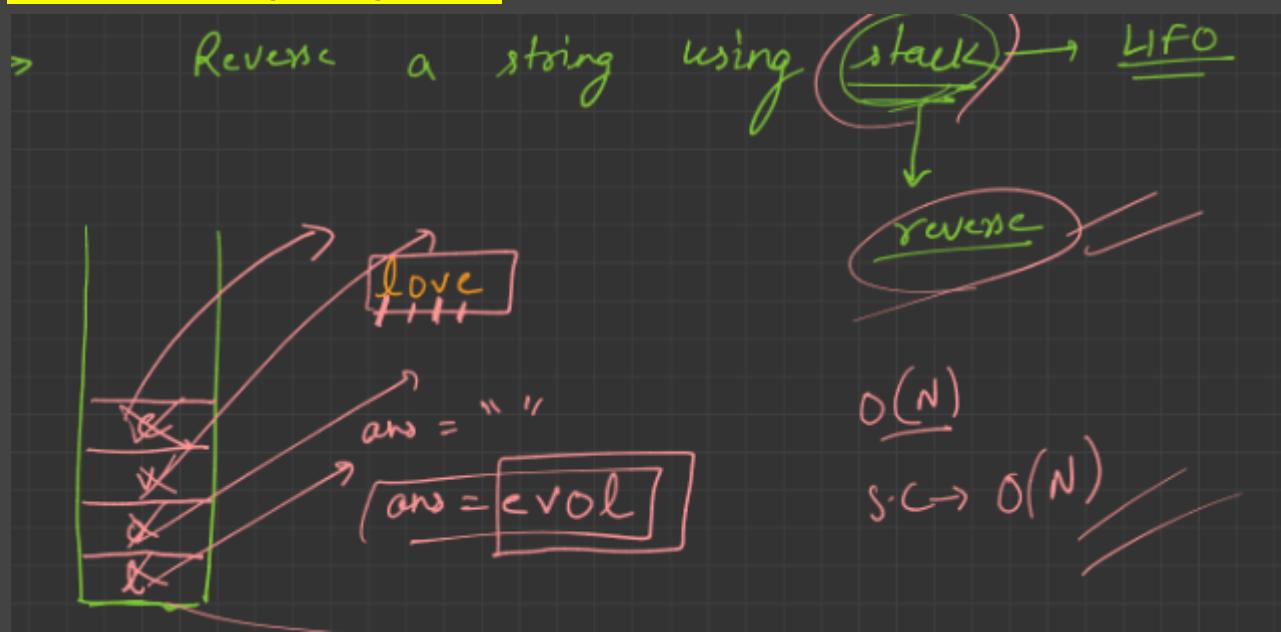
$\text{top} < \text{size} \rightarrow$



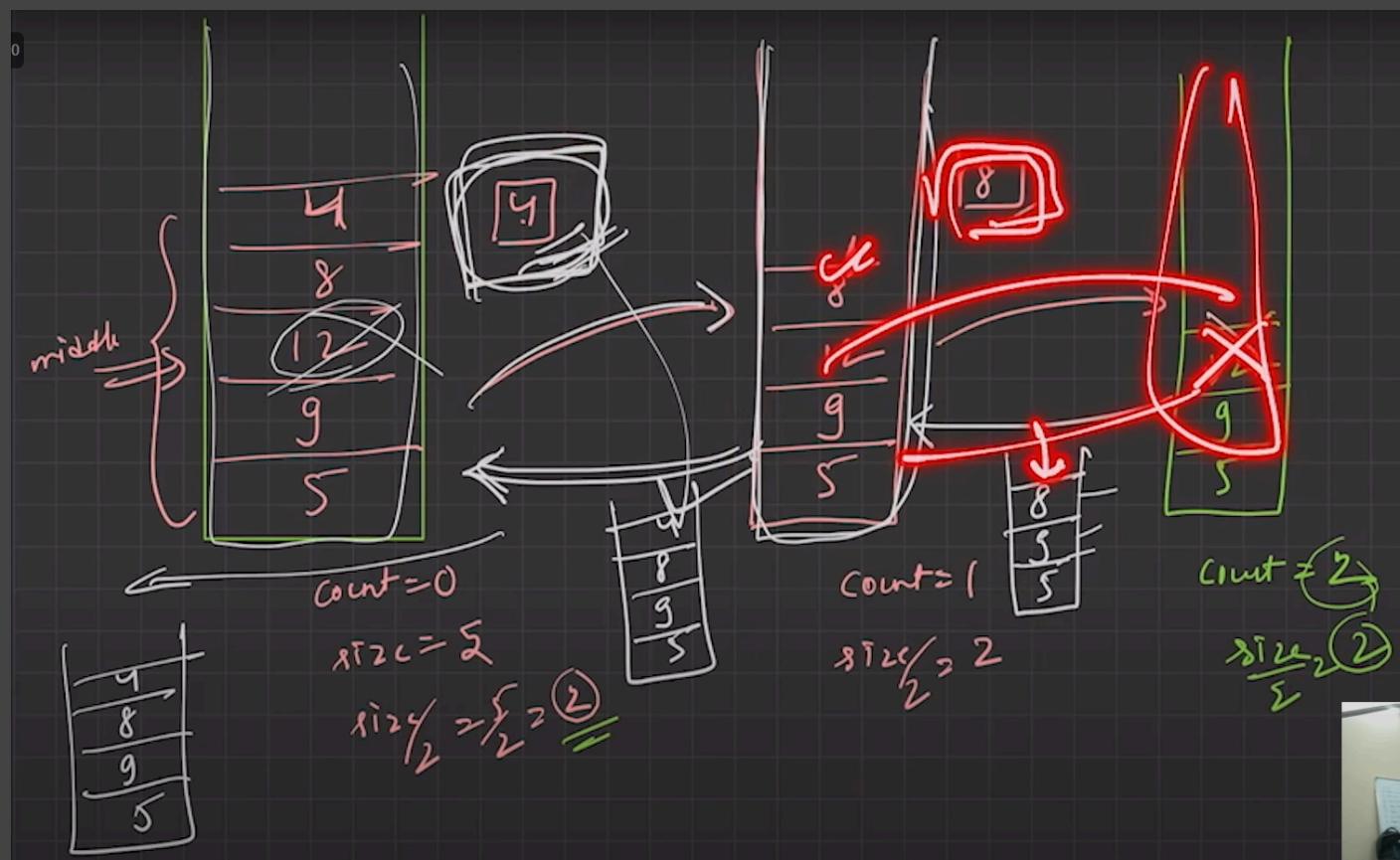
Create Two Stacks Using An Array:



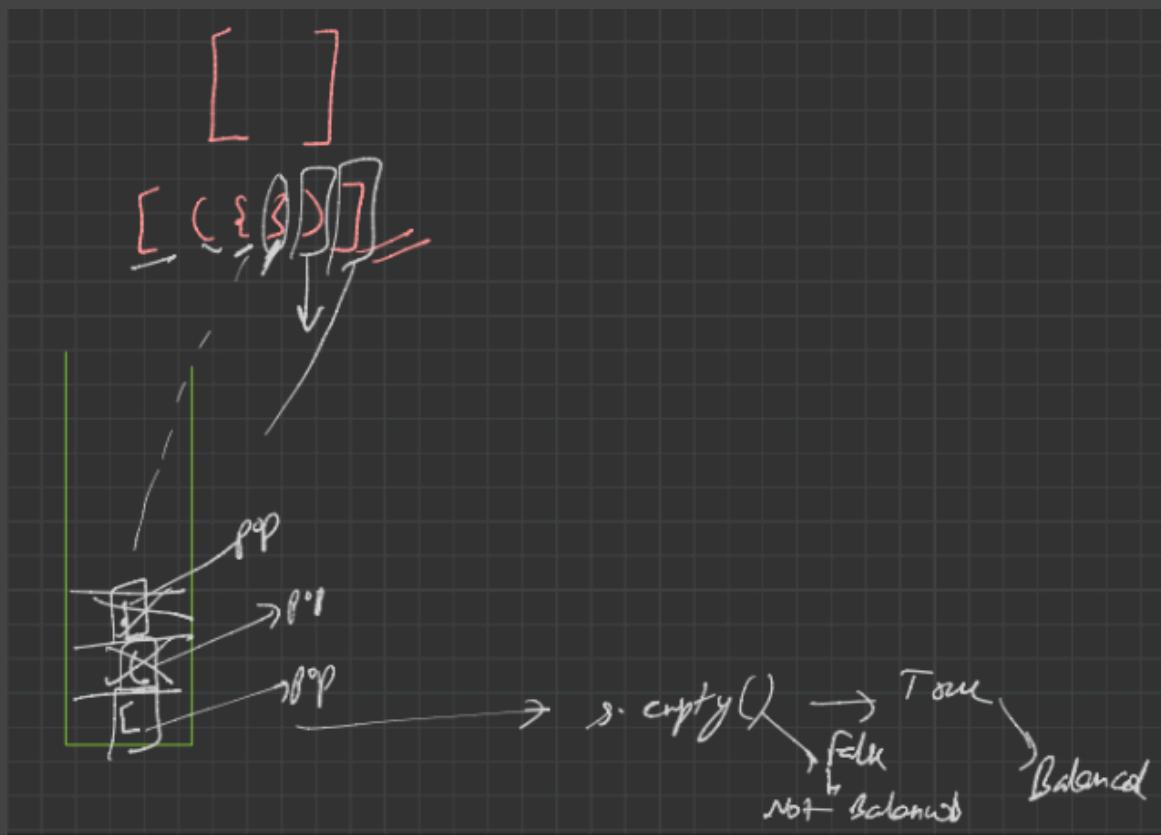
Reverse A String Using Stack:



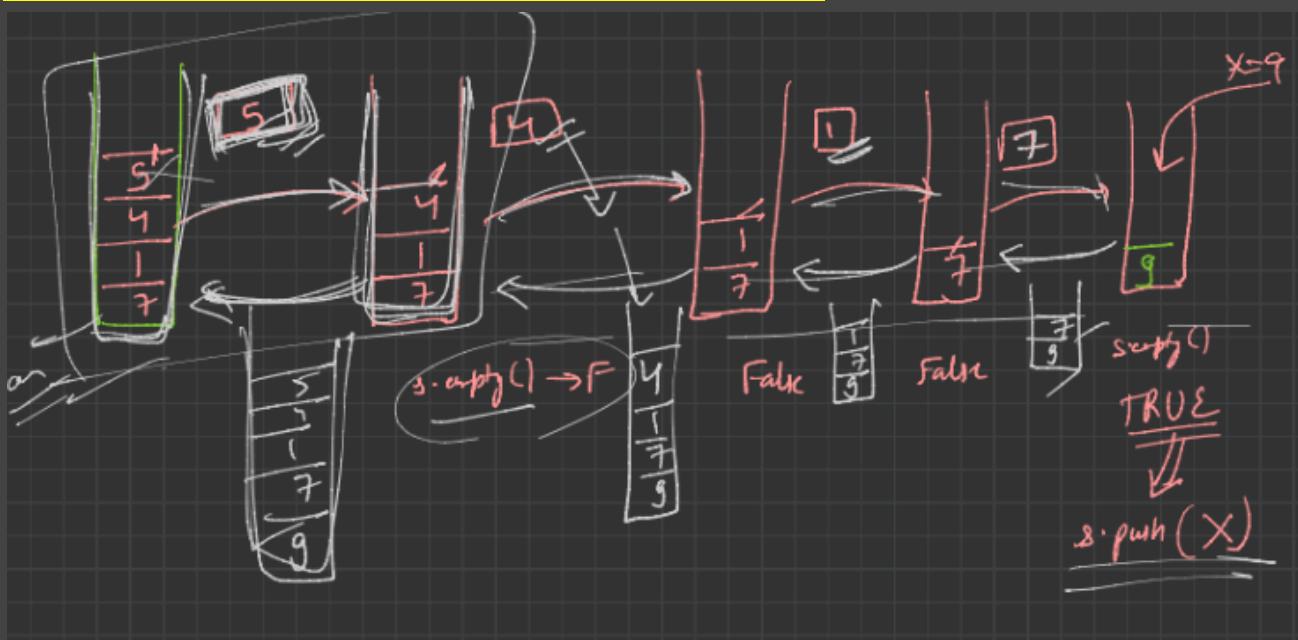
Delete Middle Element In A Stack:



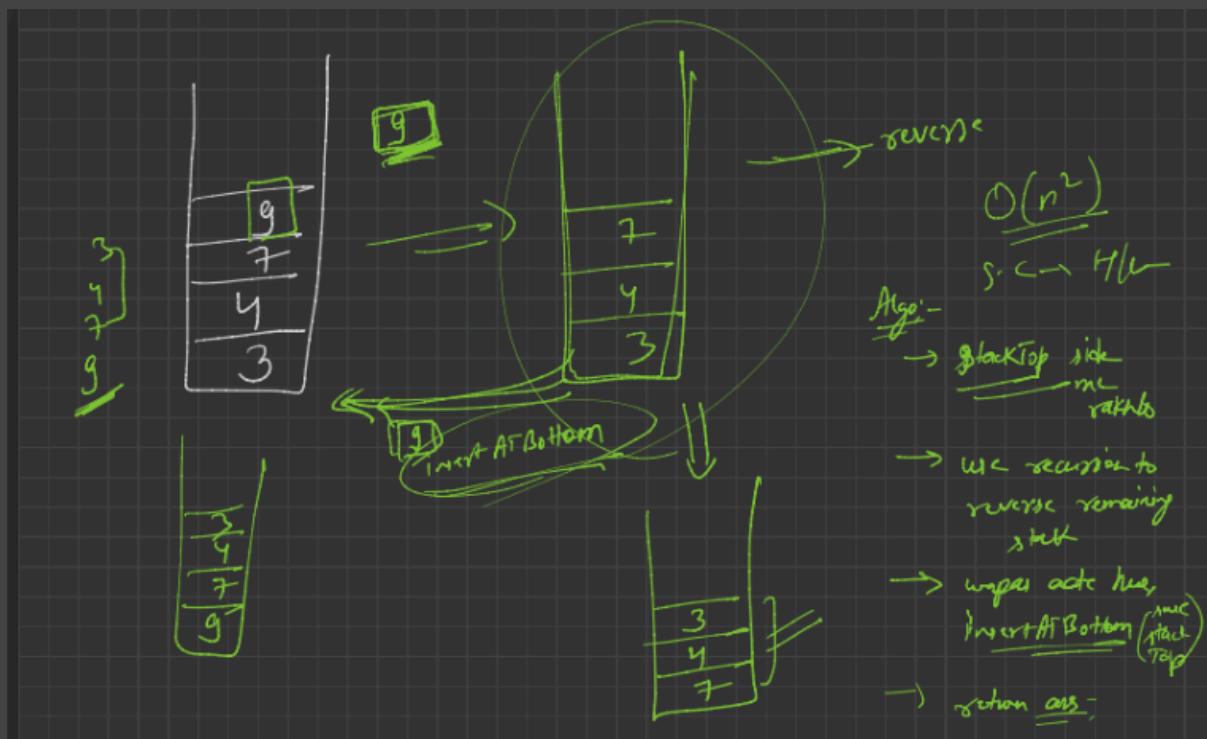
Check Whether the Parenthesis Are Valid:



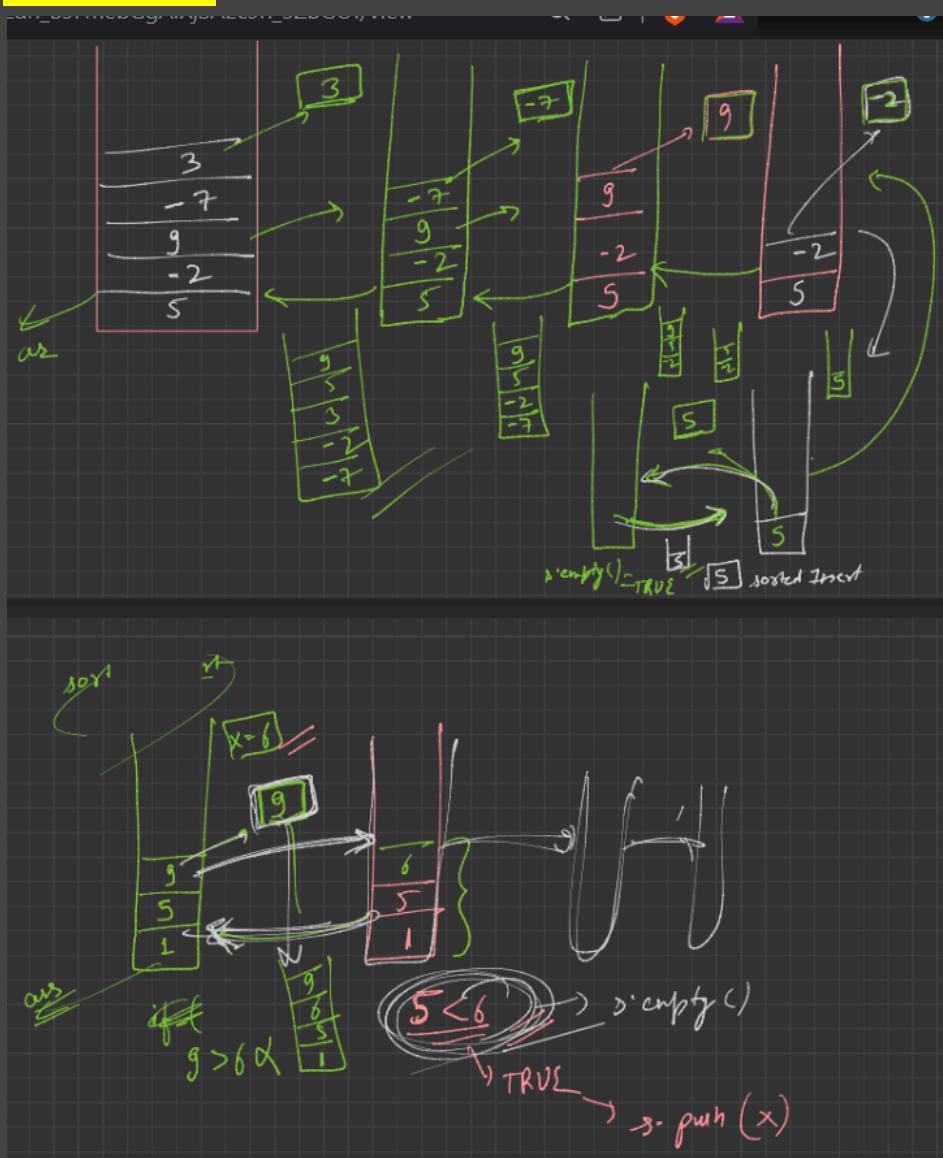
Insert An Element At Its Bottom In A Given Stack:



Reverse A Stack:



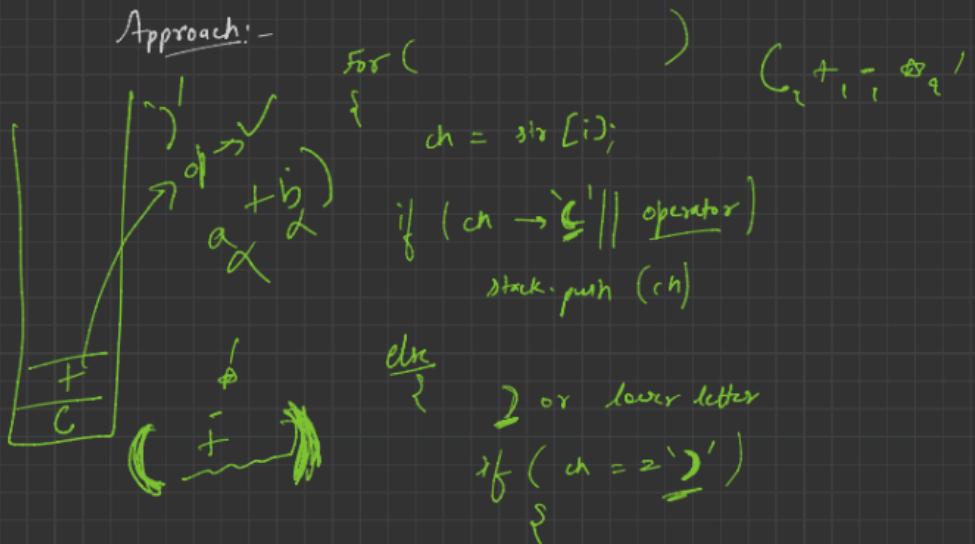
Sort A Stack:



Check Whether There Are Redundant Brackets Or Not:

$$\rightarrow \text{ifp} \rightarrow \text{string} = "(\underline{a+b})"$$

Approach:-



stack :-

([a+b])

"C" q + i - 1 & 1
A.push

if ')' → it is certain in stack
me open bracket b/w tags

jab tak stack me open bracket
nahi milta

stack top check karo

Redundant pair

Found

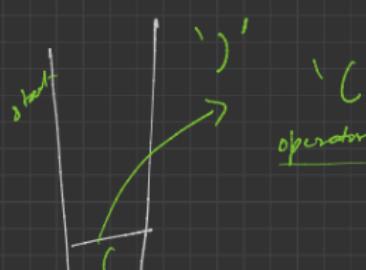
for operator
Not found

Redundant pair ==

([a+b])

DRY RUN

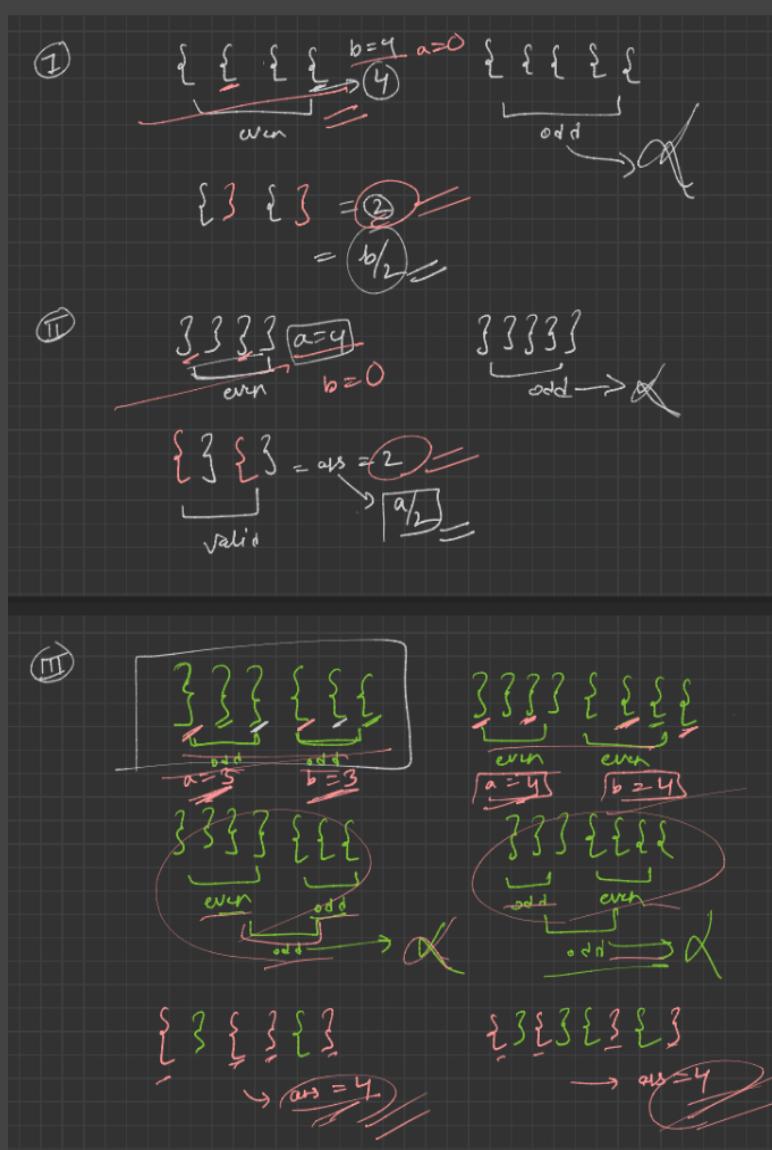
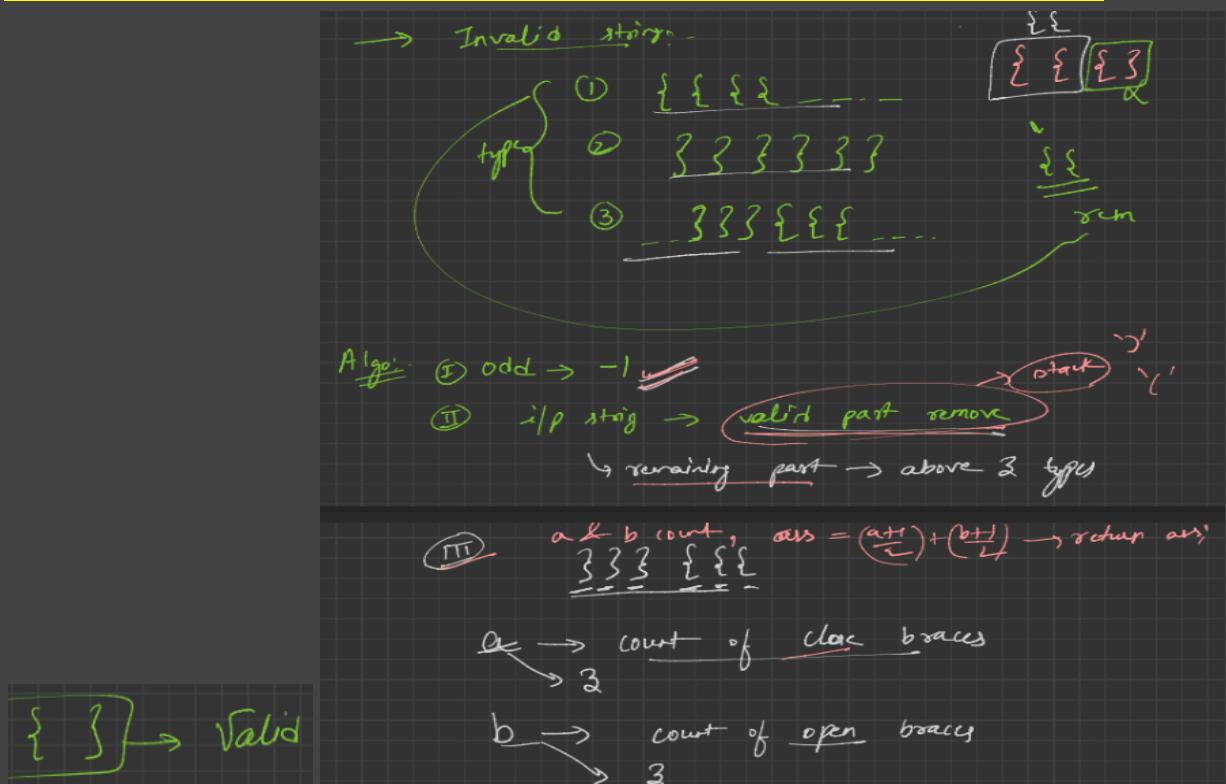
operator → R.B → Node



operator Not found

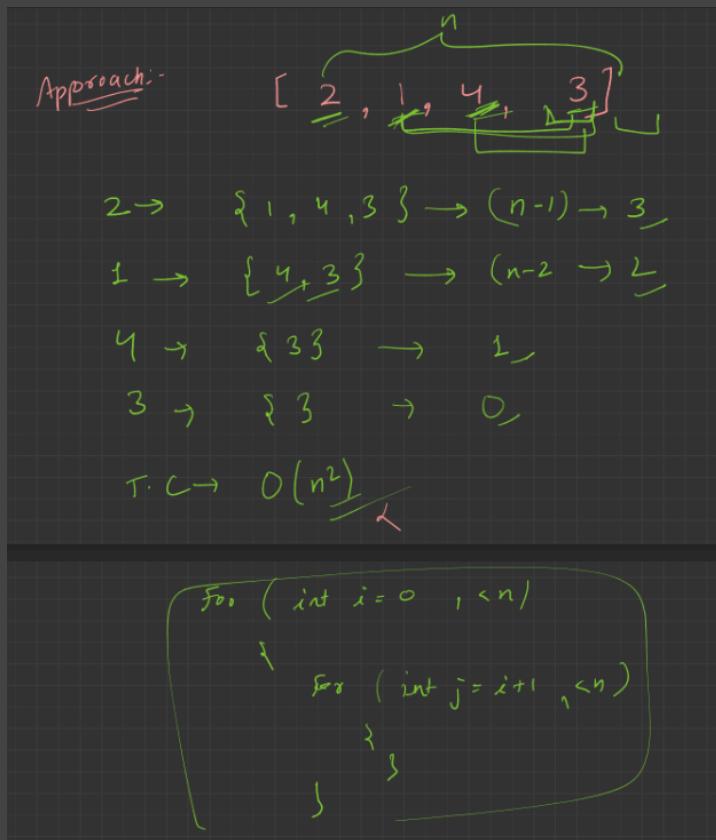
R.B →
return TRUE

Minimum Number Of Operations To Make A Valid Bracket String:



$$\begin{aligned}
 \text{ans} &= \left(\frac{a+1}{2} \right) + \left(\frac{b+1}{2} \right) \\
 &= \frac{0+1}{2} + \frac{4+1}{2} = \frac{5}{2} = 2.5 \\
 &= (4+1) + (0+1) = \frac{5+1}{2} = 3 \\
 &= (3+1) + \left(\frac{3+1}{2} \right) \\
 &= \frac{4}{2} + \frac{4}{2} = 2+2 = 4 \\
 &= \left(\frac{4+1}{2} \right) + \left(\frac{4+1}{2} \right) = \frac{5}{2} + \frac{5}{2} = 2+2 = 4
 \end{aligned}$$

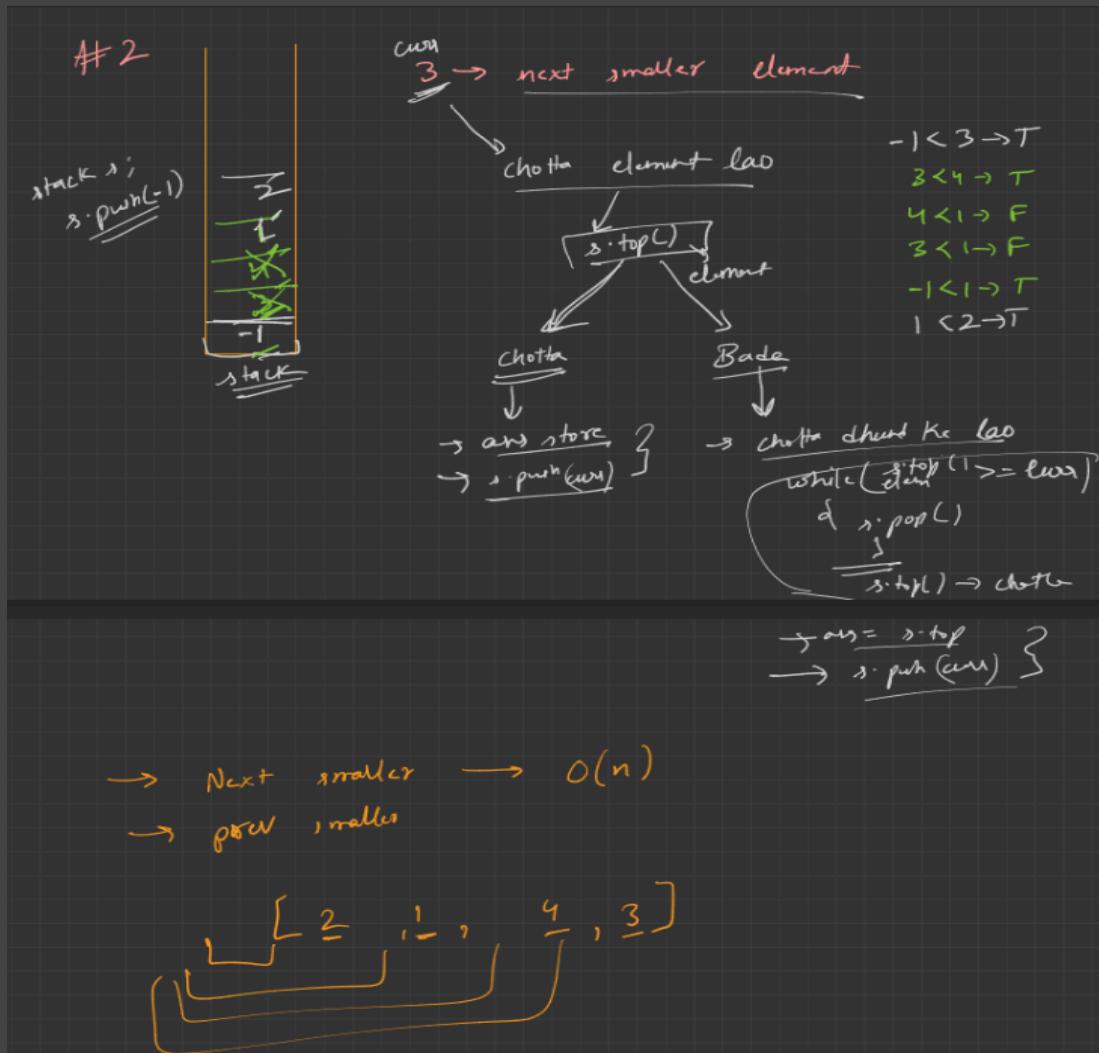
Next Smaller Element Approach-I:



```

    Foo ( int i=0 , <n)
    {
        for ( int j=i+1 , <n )
        {
        }
    }
  
```

Next Smaller Element Approach-II:

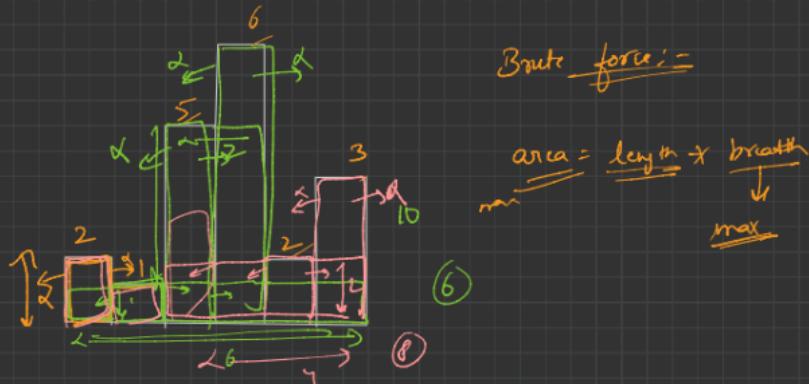


Previous Smaller Element:

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 Rectangular Area in Histogram



BF

for (int i = 0; i < n; i++)

{

 while (left)

 {

 while (right)

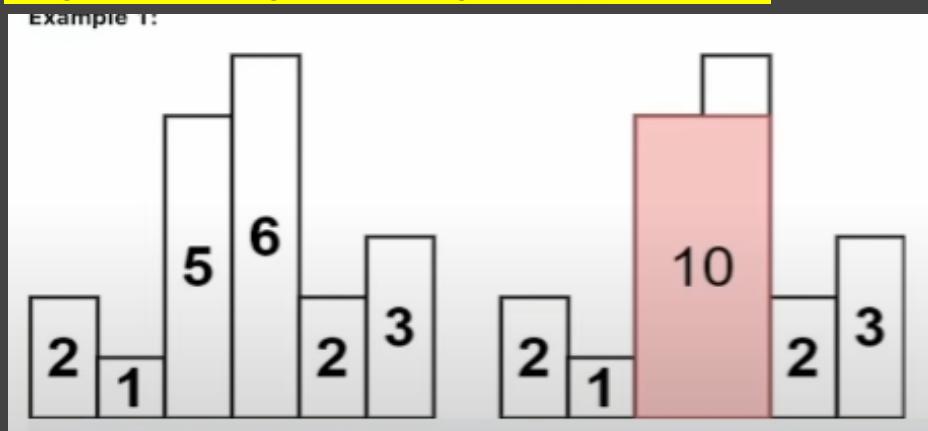
 {

 area = max (area, min (heights));

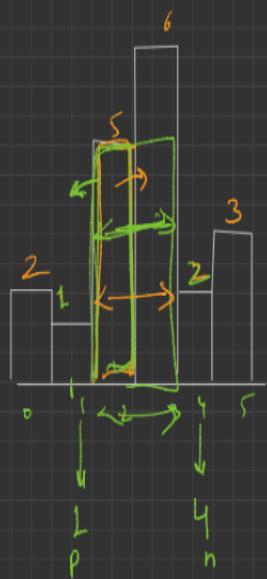
T-C
 $O(n^2)$

$O(n)$?

Largest Rectangle In Histogram Approach-II:



$O(n)$ → Approach #2

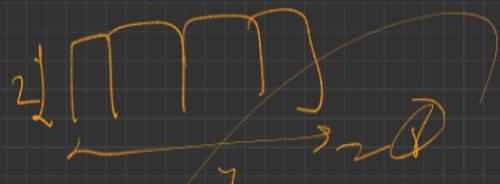


$$\text{width} = \frac{n-p-1}{2}$$

$$\text{area} = \frac{1}{2} \times n \times (p-1) = \frac{1}{2} \times n \times (n-p-1) = \frac{1}{2} n(n-p-1)$$

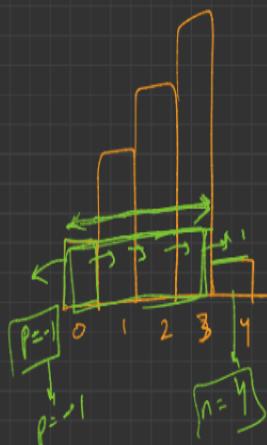


2 2 2 2



$$\text{next} \rightarrow [2, 2, 2, 2] \quad \text{prev} \rightarrow [-1, -1, -1, -1]$$

$$n-p-1 = -1 + 1 = 0 \rightarrow \text{width} = 0$$



$$\text{width} = \frac{n-p-1}{2} = \frac{4-(-1)-1}{2} = \frac{4+1}{2} = 2.5$$

next → next smaller element (index)
prev → prev —————

$$\text{next} = -1 \quad -1 \quad -1 \quad -1 \quad \text{if } (\text{mod} = 0 - 1)$$

$$\text{prev} = -1 \quad -1 \quad -1 \quad -1 \quad 1$$

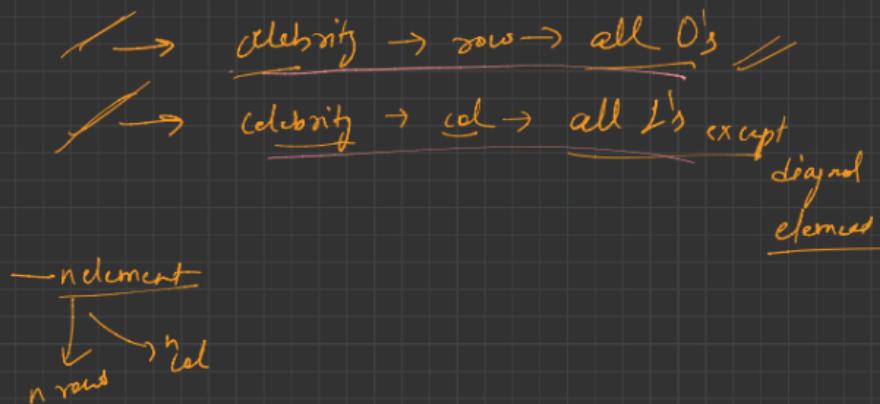
$$n-p-1 = -1 + 1 = 0 \rightarrow \text{trapezoid width} = 0$$

$$\text{area} = 0$$

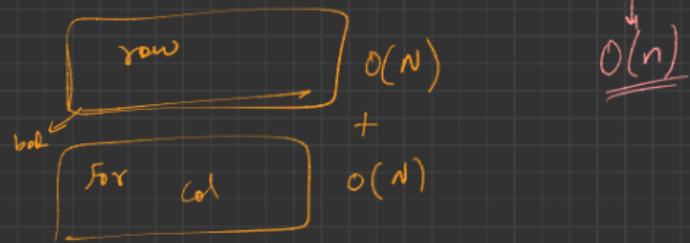
$T.C \rightarrow O(N)$ $S.C \rightarrow O(N)$ More efficient

Celebrity Problem Approach-I:

Approach:- Brute force ~~Brute force~~



for (int i = 0 i < n) T.C $\rightarrow \mathcal{O}(n^2)$

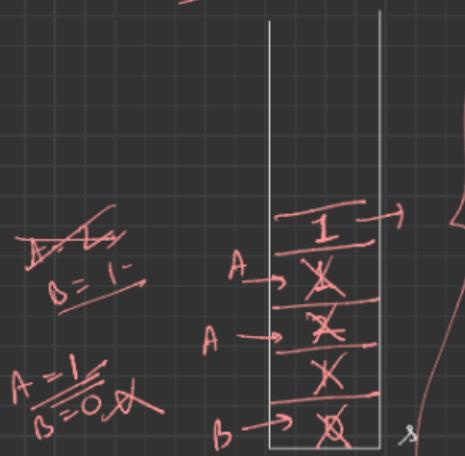


}

if (bool
 | or Not

Celebrity Problem Approach-II:

Approach #2



Algo:-

put all element inside stack

→ jab stack size != 1

→ A → s.top() → s.pop()

→ B → s.top() → s.pop()

→ if (A Knows B)

A → , B → wapas push karo

→ if (B Knows A)

B → , A → wapas add do

→ jo single element bache hua h,

vo ck "Potential Celebrity"

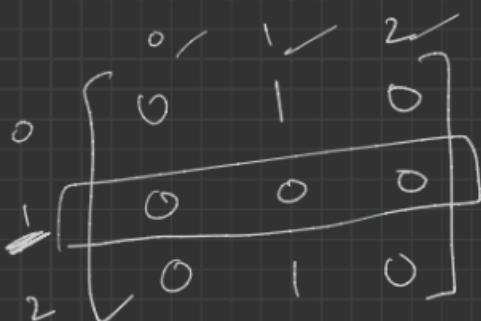
ho skta hain

→ Verify

→ Celebrity knows N.One
→ now check → all O's

Everyone Knows celeb

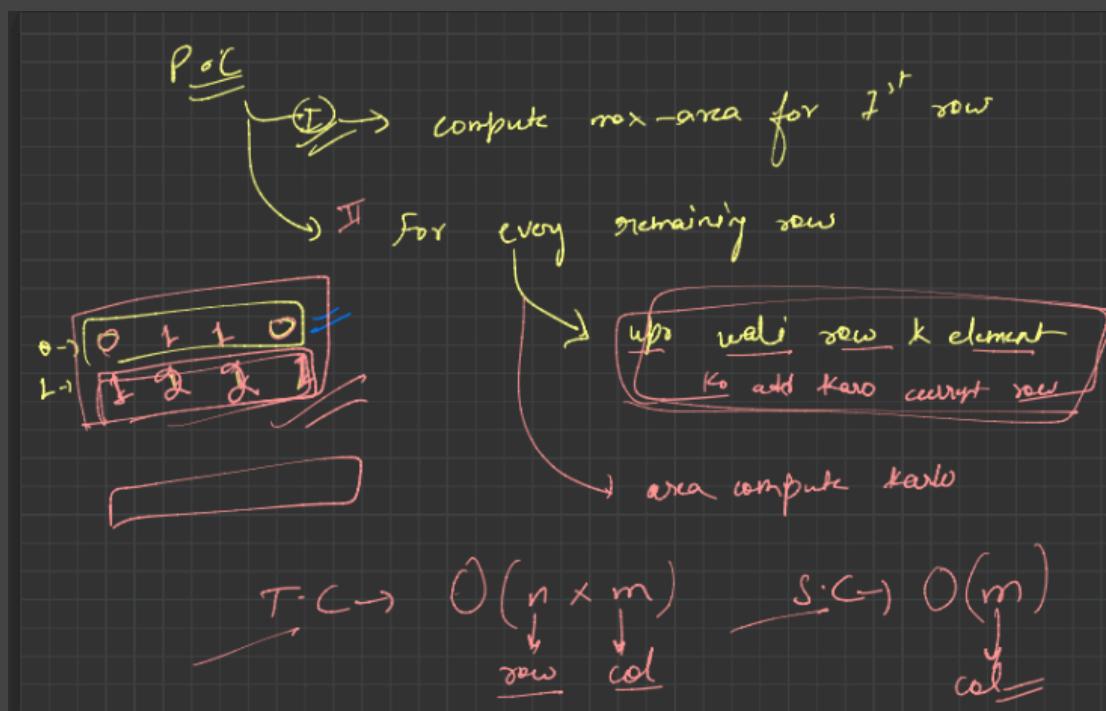
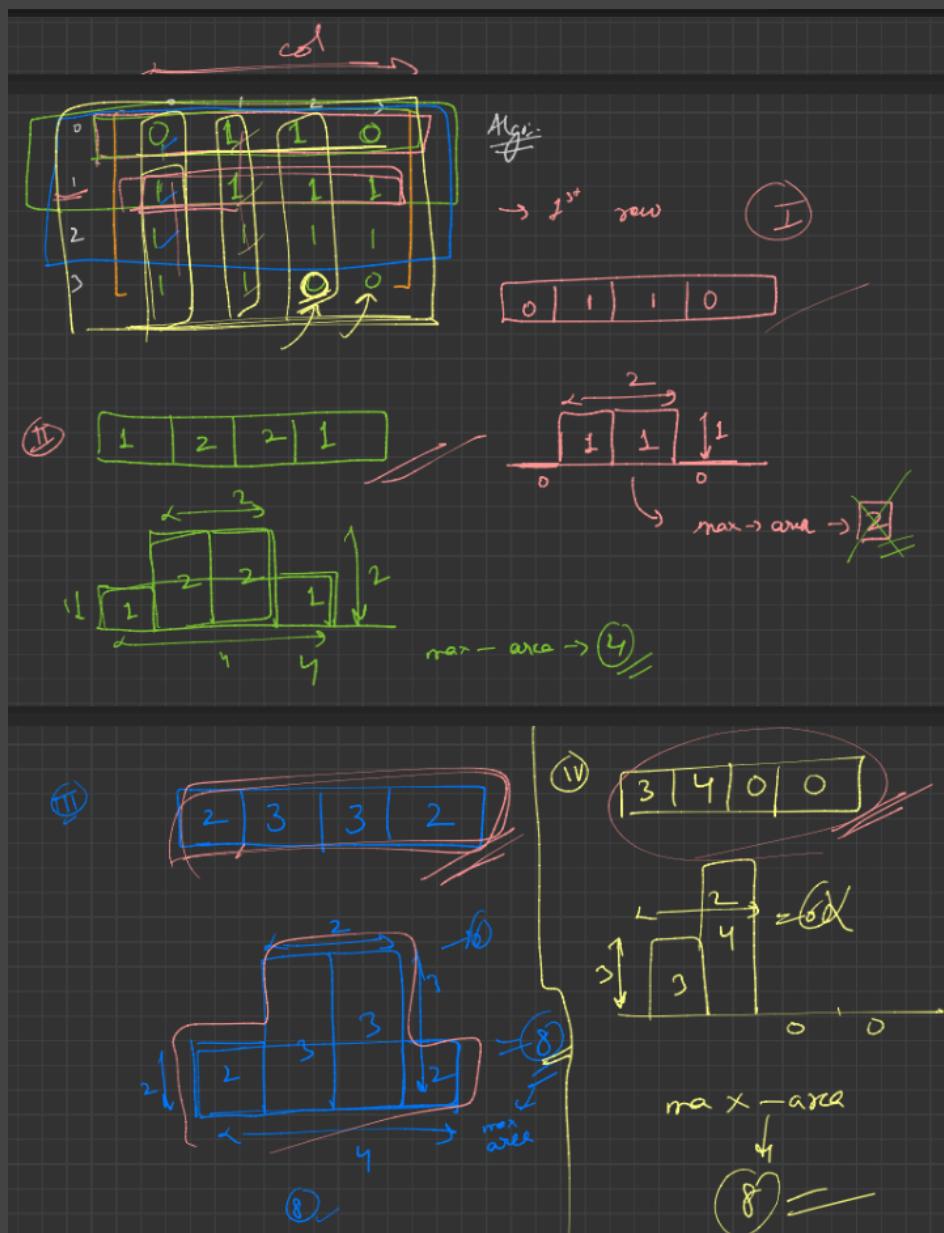
→ col → all 1's
except diagonal
diagonal
direct



T.C → O(N)

H/W → more approaches exploration

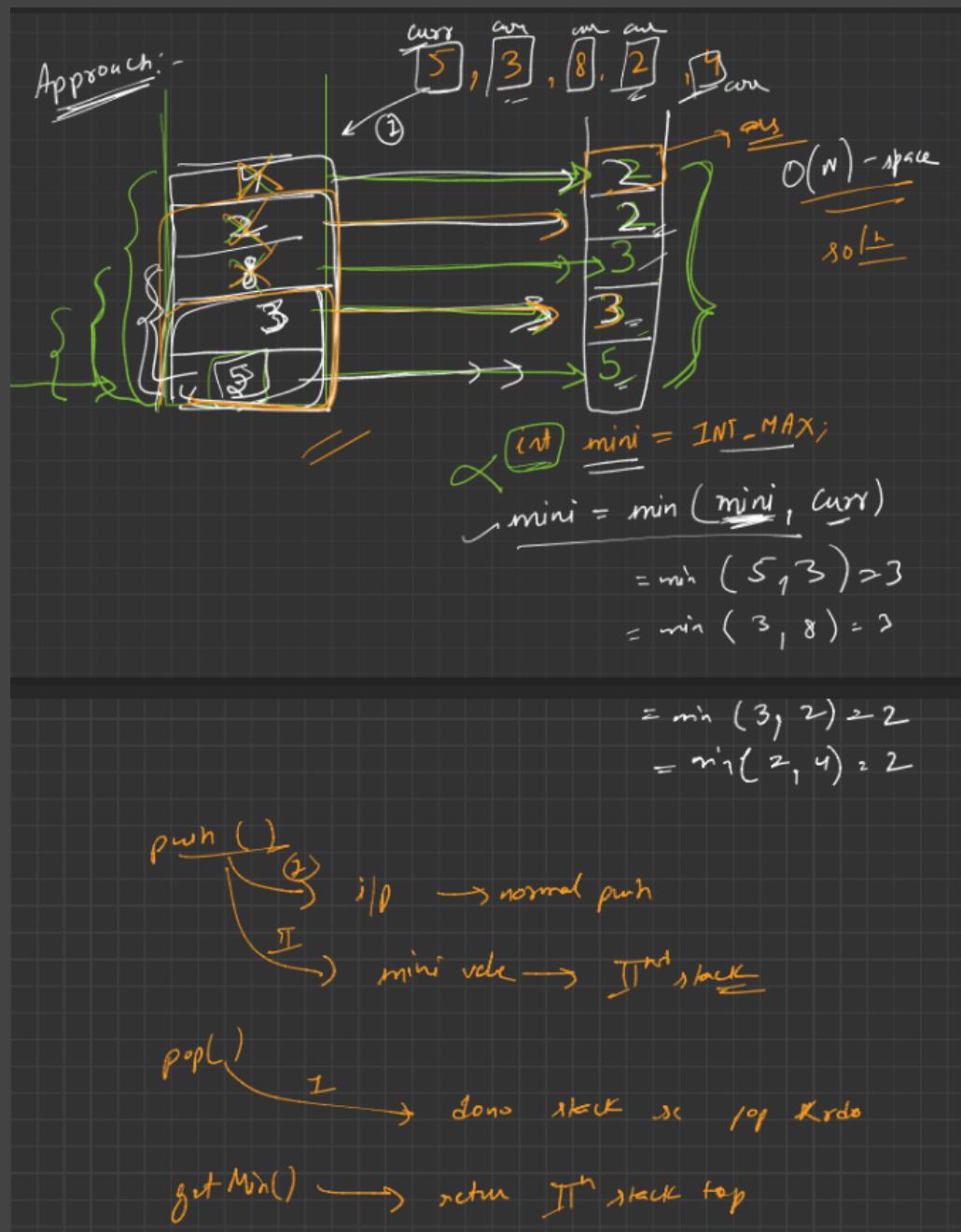
Maximum Rectangle In Binary Matrix:



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

Complexity:

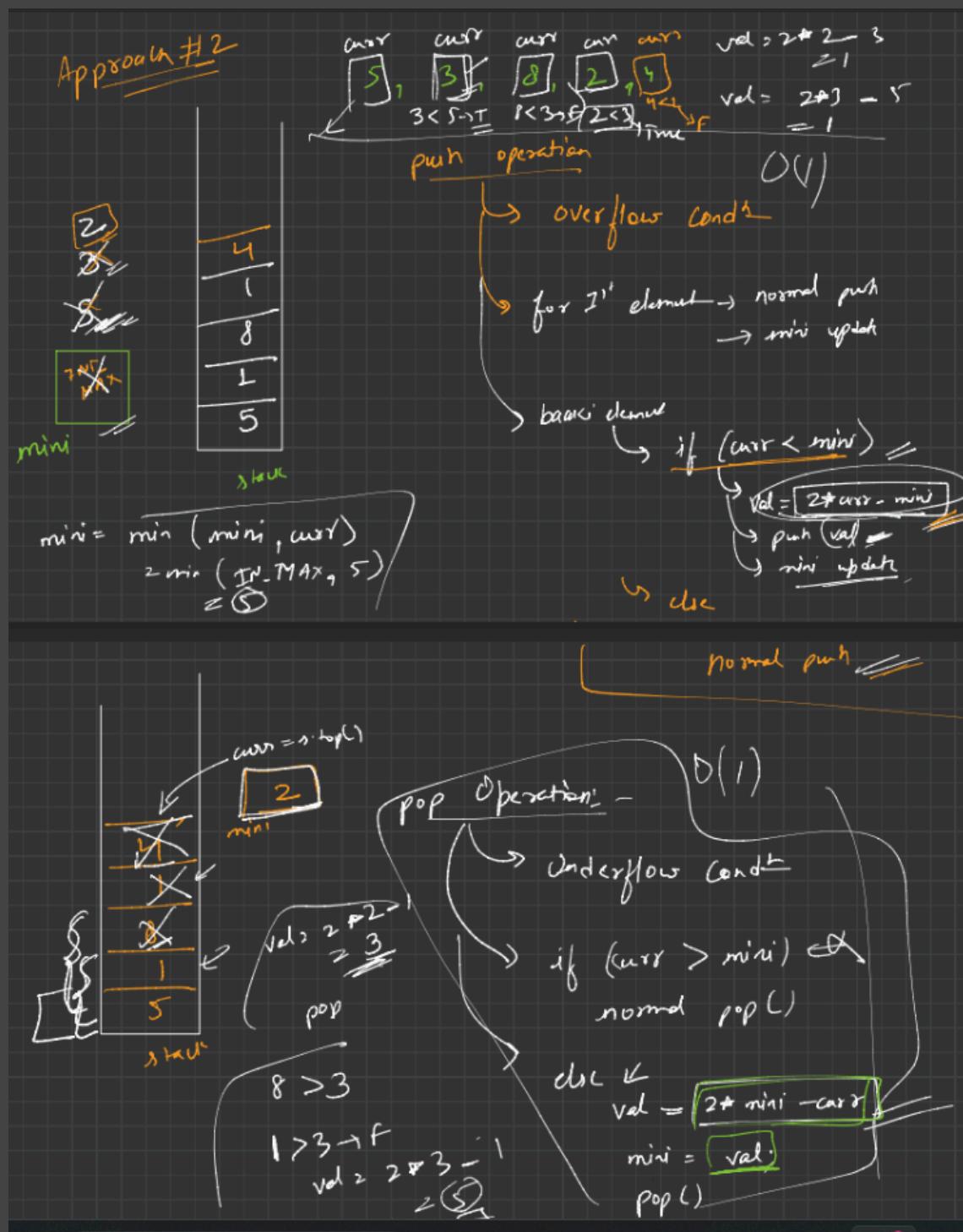
Approach -I:



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() $O(1)$

min → stored
return

push → $2 * \text{curr} - \text{mini}$

pop() → $2 * \text{mini} - \text{curr}$

curr minimum
prev minimum

push

$3 < 5 \rightarrow \text{val} = \boxed{2 * \text{curr} - \text{mini}}$
 $= 2 * 3 - 1 = 5$

$2 * \text{curr} - \text{mini}$
 $= 2 * 3 - 1$

$\boxed{2 * n - \text{prevMinimum}}$
 $\boxed{\text{mini} \rightarrow n}$

pop()

pop()

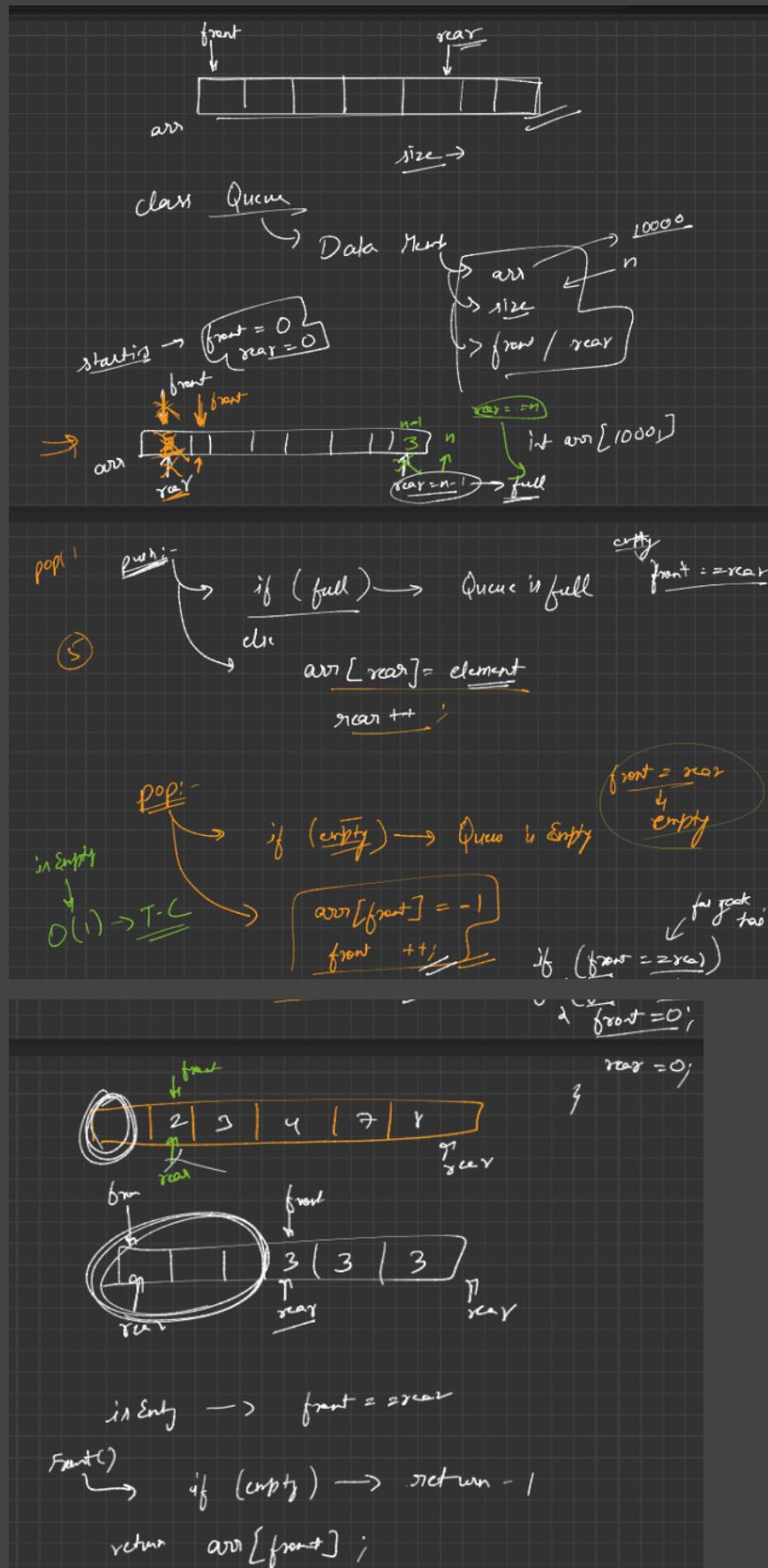
$2 * \text{mini} - \text{curr}$
 $2 * 1 - 5$

$2 * n - \text{curr}$
 $2 * 1 - 5$

$2 * n - (2 * n - \text{prevMin})$
 $\boxed{\text{prev Minimum}}$

Queue

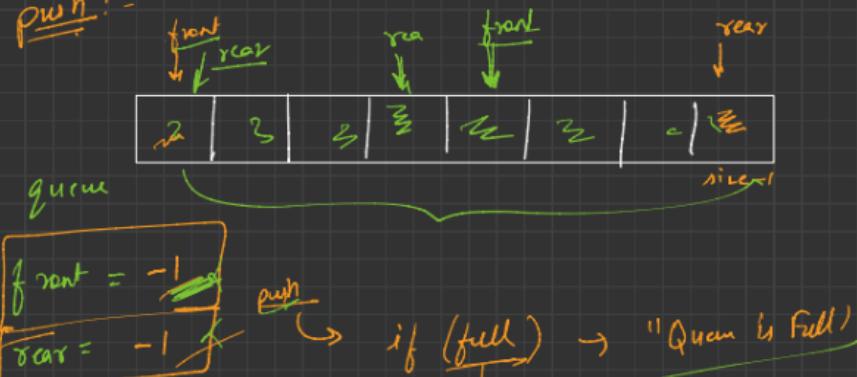
Implementing Queue Using Arrays:



Implementing Queue Using Linked-List:

Circular Queue:

push:

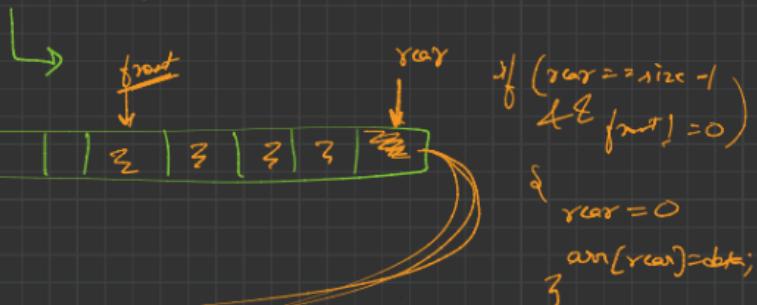


$\hookrightarrow \text{if } (\text{front} == -1) \quad // \text{first element}$

\downarrow
 $\text{front} = \text{rear} = 0!$

$\text{arr}[\text{queue}] = \text{data};$

$\}$

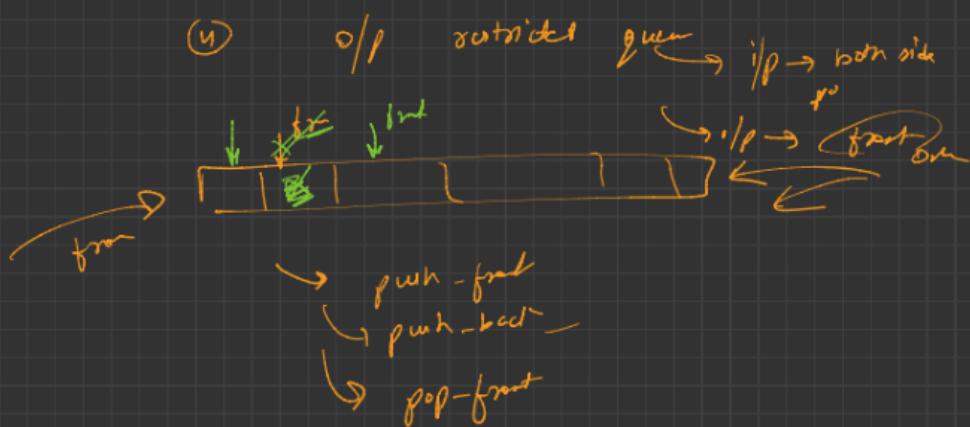
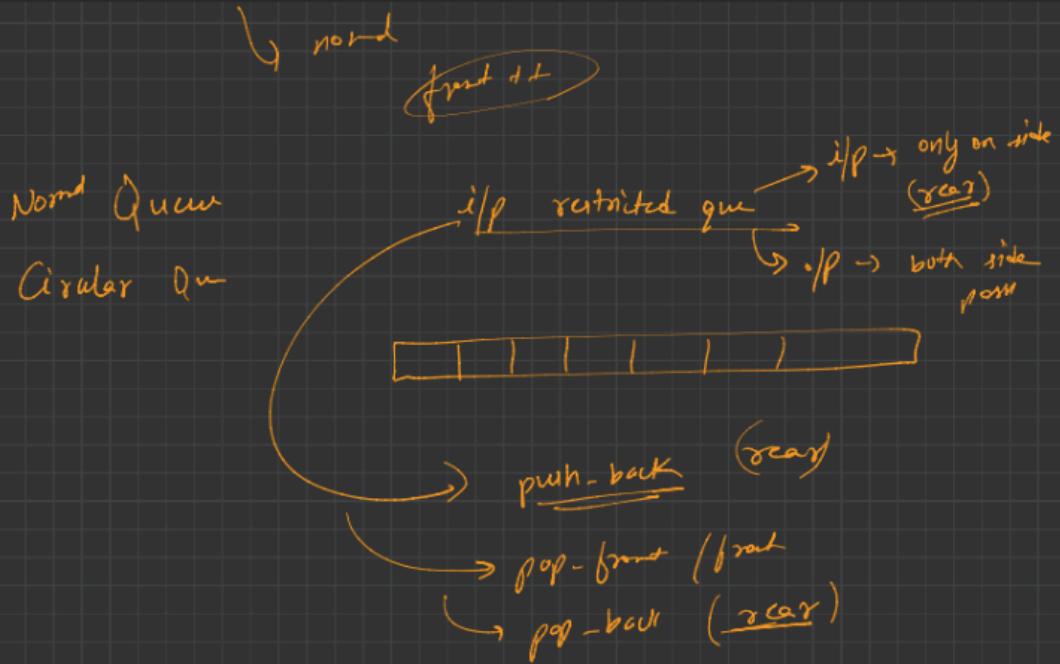


else

$\text{rear} + 1$

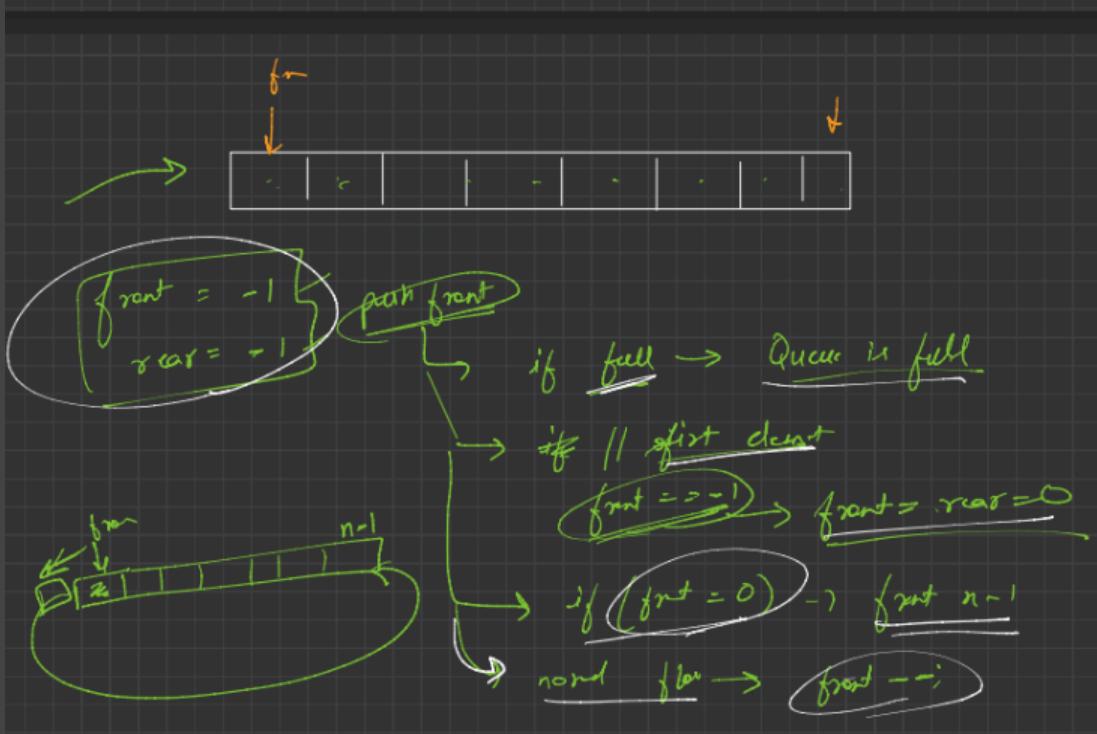
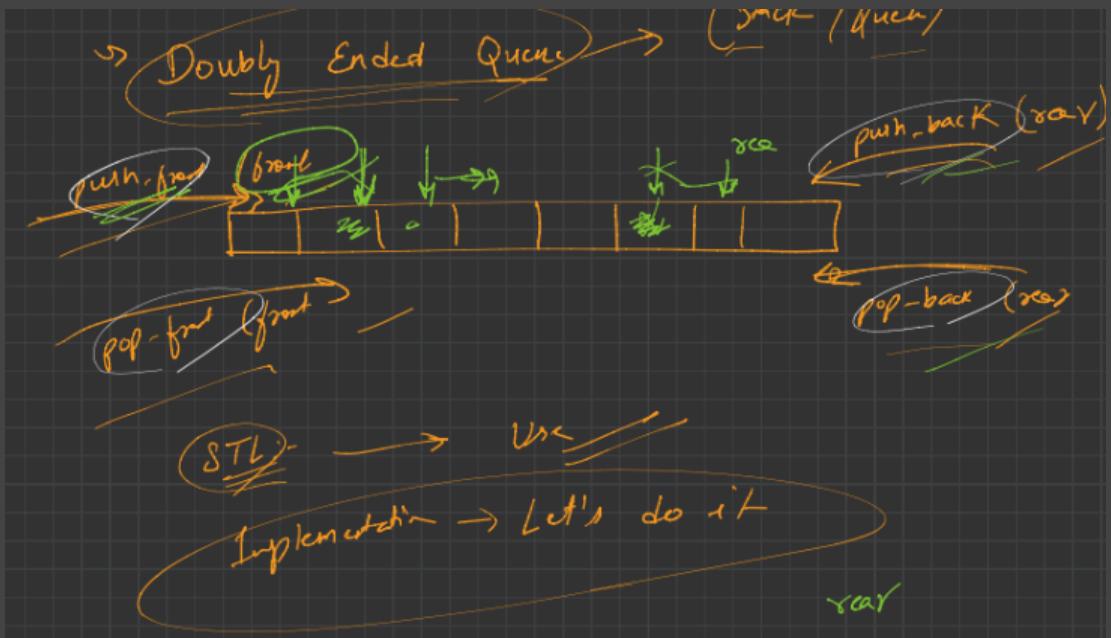
$\text{arr}[\text{rear}] = \text{data};$

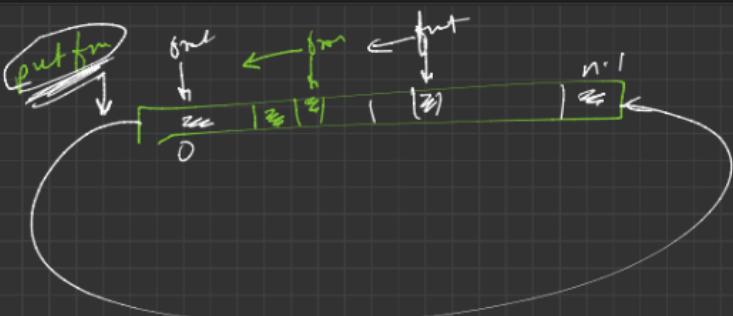
Input Restricted And Output Restricted Queue:



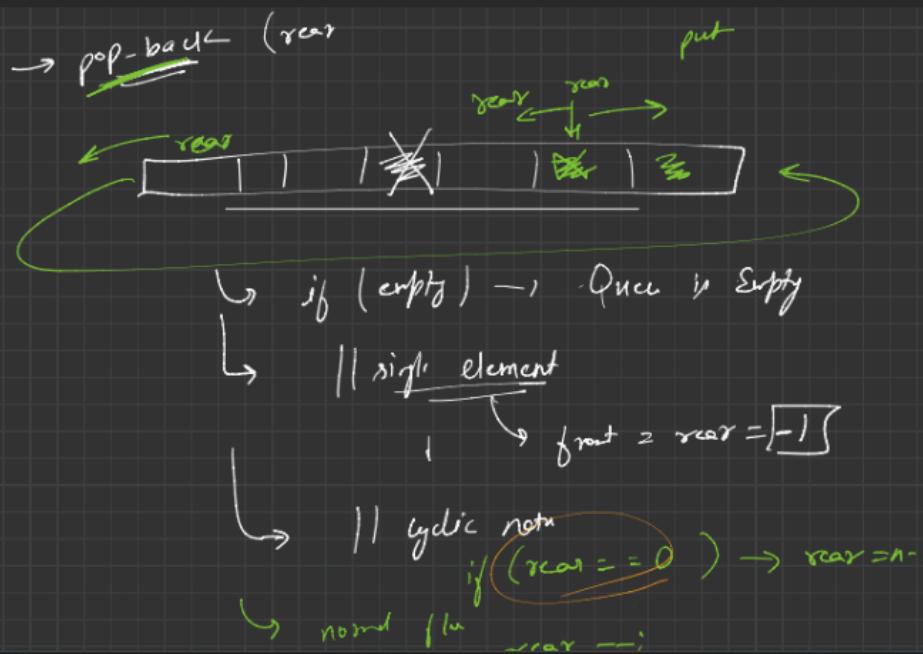
Doubly Ended Queue(Deque) STL Code:

Doubly Ended Queue(Deque) Implementation:

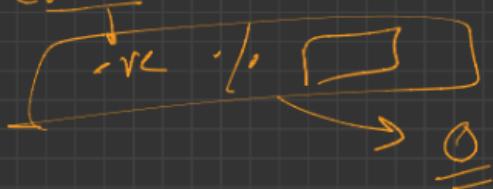




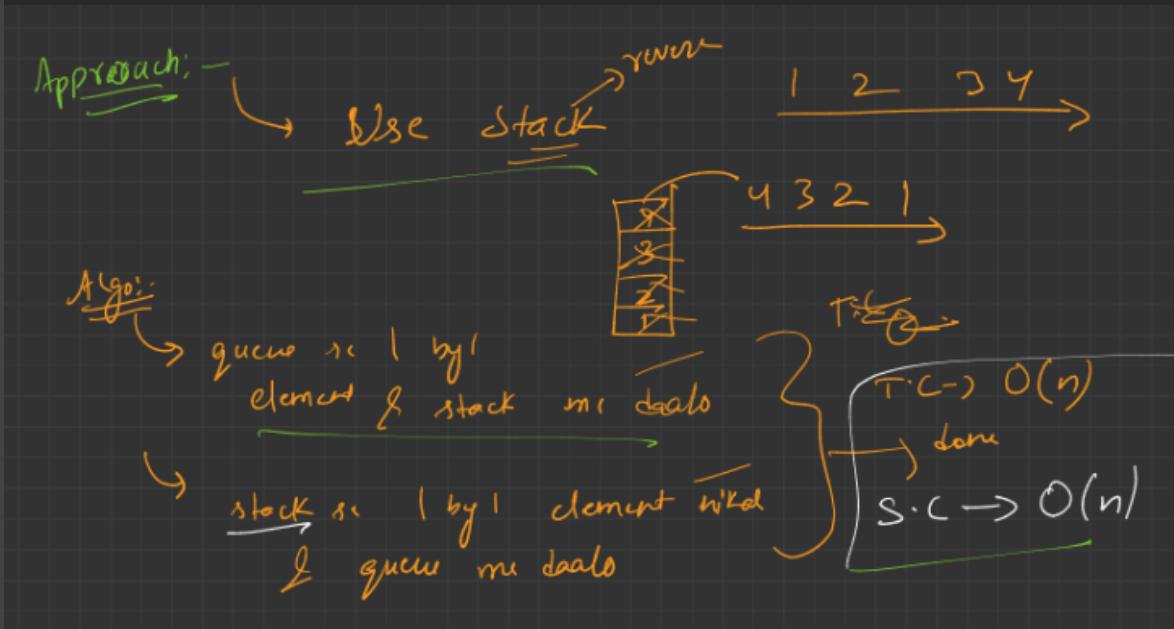
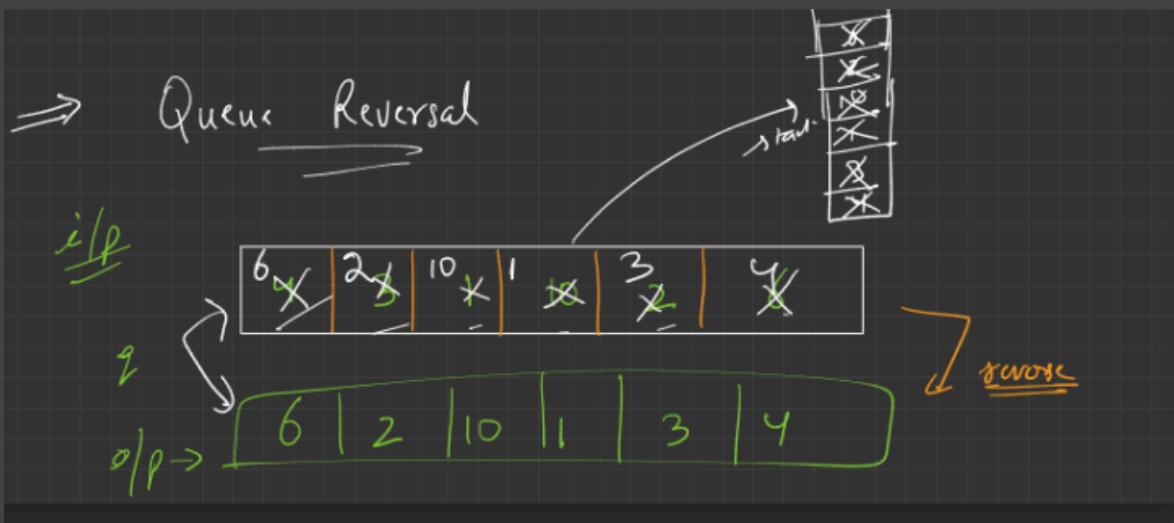
push rear → 8
rear → num →
→ pop front
→ pop-back



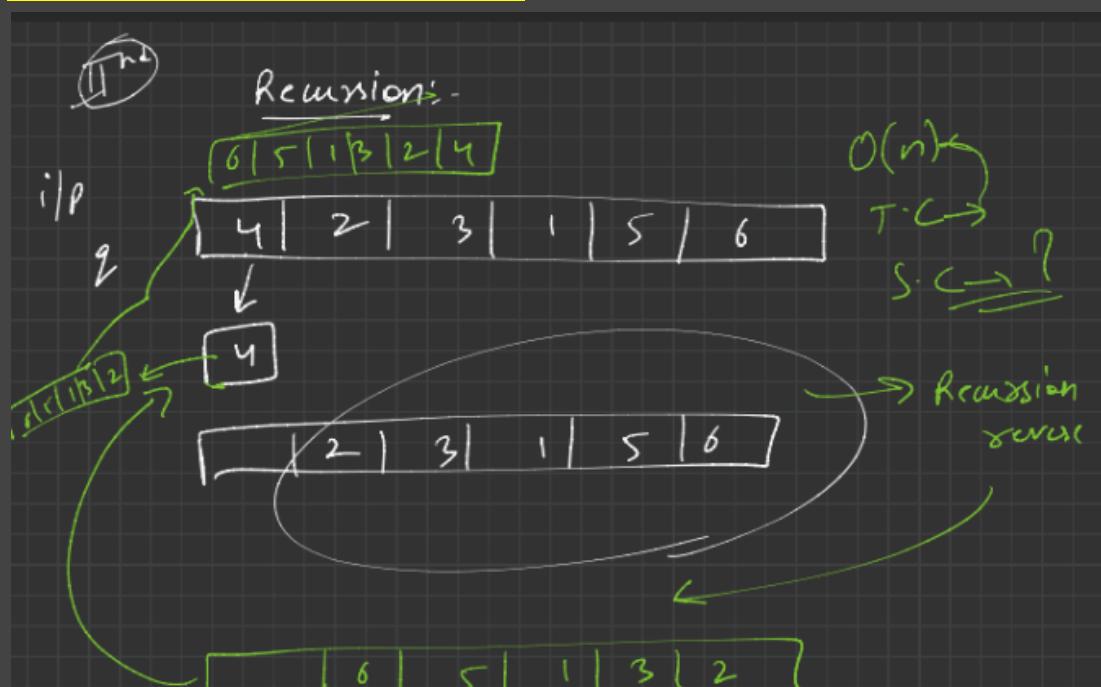
front = 0
(front - 1)



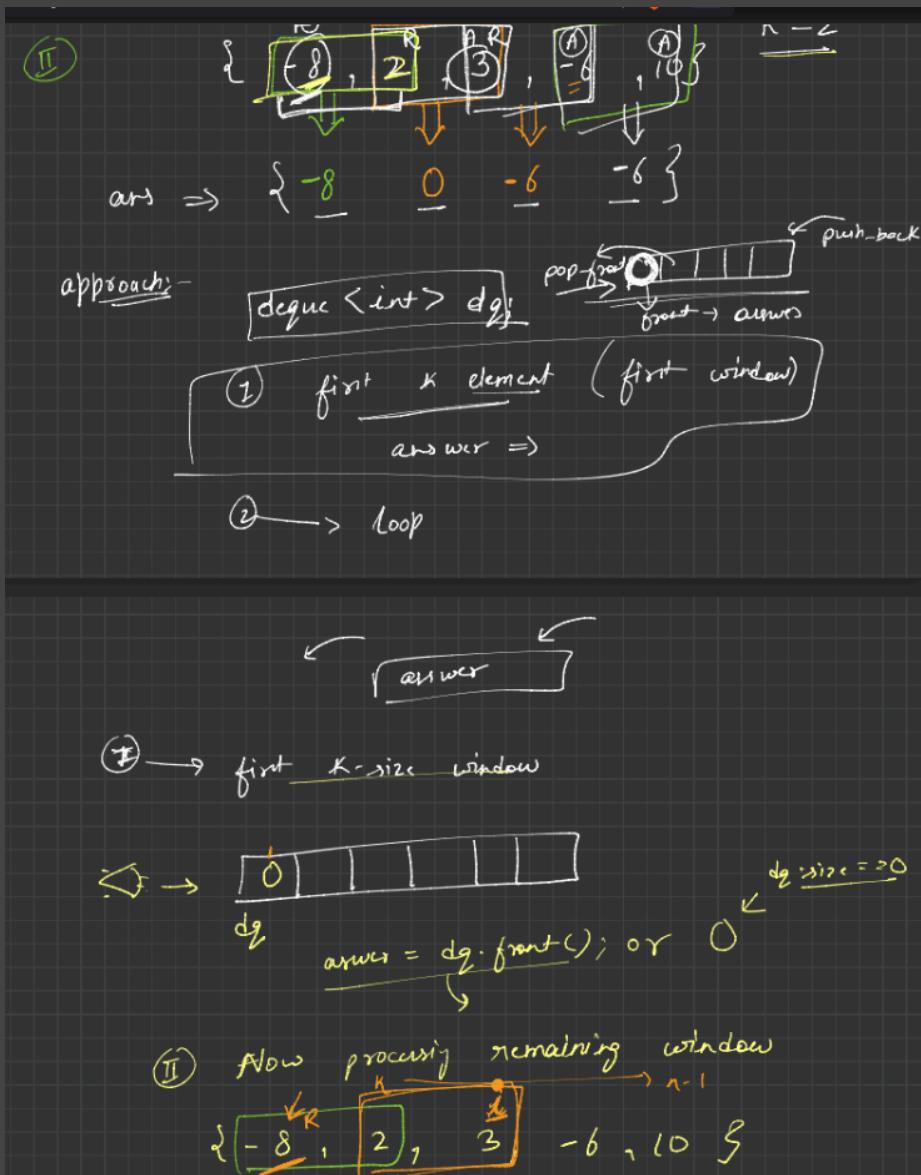
Reverse A Queue Approach-I:



Reverse A Queue Approach-II:



First Negative Integer In Every Window Of Size K:

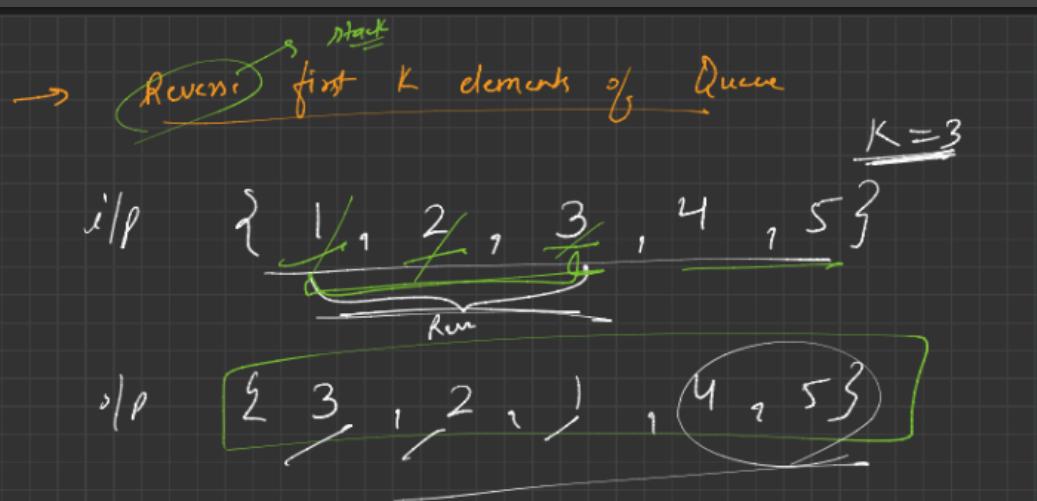


```

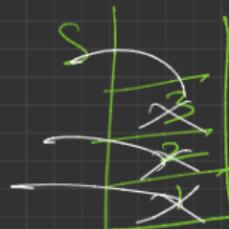
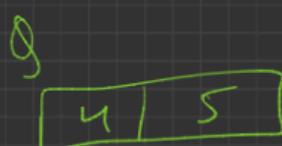
// Removal
for ( i → n )
{
    // Removal
    if ( dq.front() - i ≥ k )
        dq.pop_front();
    // addition
    if ( arr[i] < 0 )
        dq.push_back(i);
    ans ← dq.front();
    size = 0;
}

```

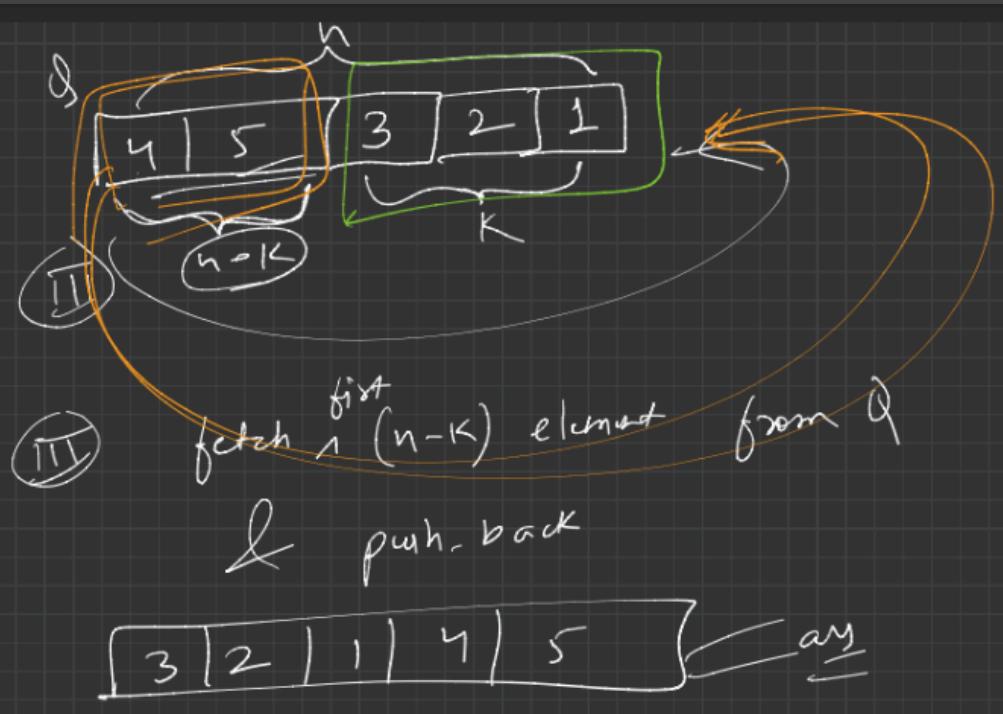
Reverse First K Elements In A Queue:



Algo:- (I) fetch first K element from Q
 ↳ put into stack

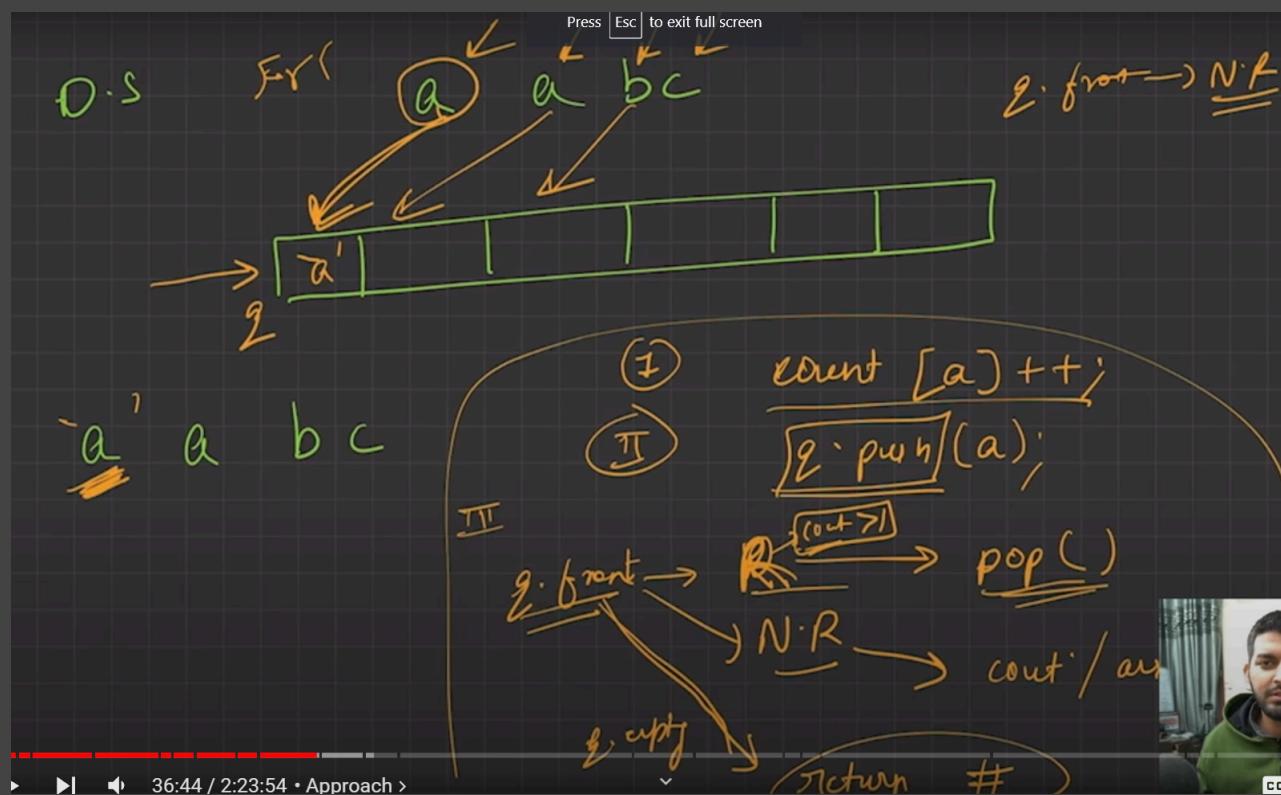
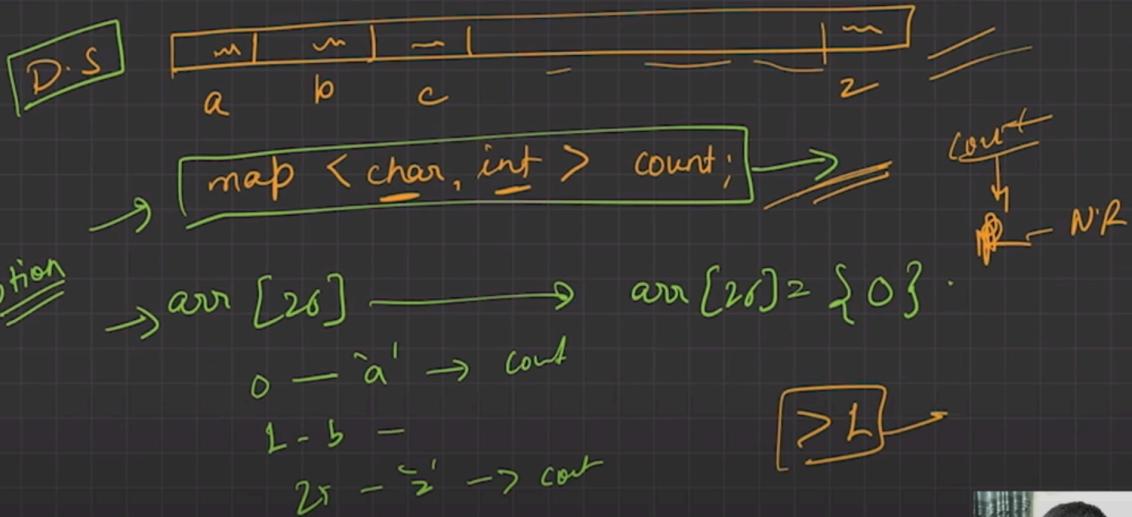


(II) fetch element from stack
 ↳ put into Q



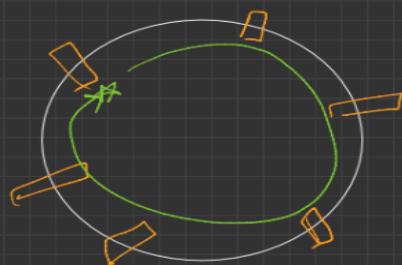
First Non-Repeating Character In A Stream:

Approach:-



Circular Tour:

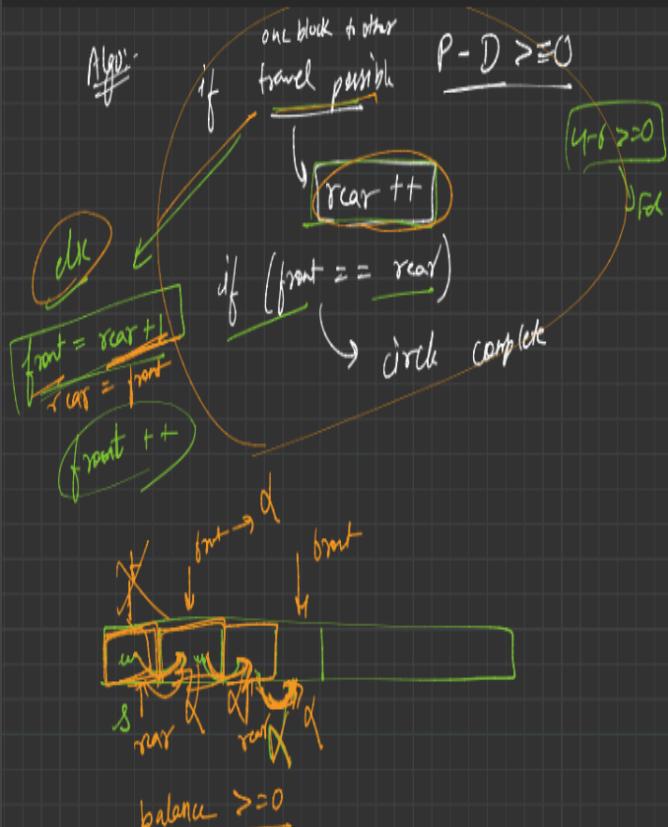
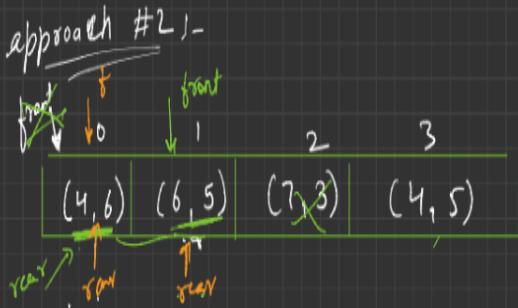
→ Circular tour:



P-P → Patrol data (in litres)

→ mean P-P distance (y km)

approach #1 - B.F

 $\rightarrow O(n^2)$


Algorithm

start block to block

if (travel possible)

rear ++

else

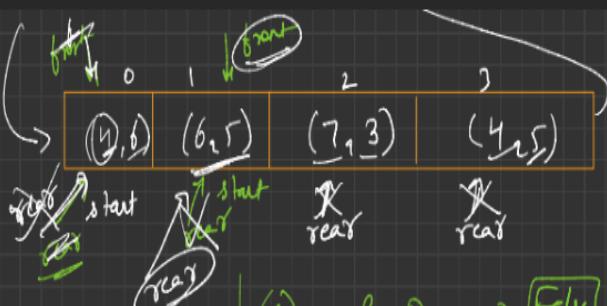
front = rear + 1

start = front

rear = front

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

cycle complete



(I) $P - D \geq 0$ [False]

$$4 - 6 = -2 < 0$$

$front = rear + 1$

(II) $P - D \geq 0$ [True]

$$8 - 5 = 3 \geq 0$$

balance = 3

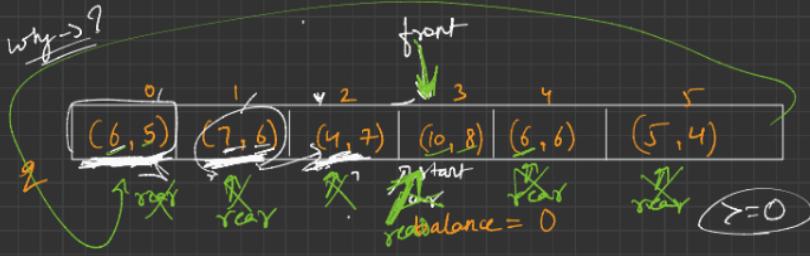
balance = 1 + 7 - 8 - 3 = 5 - 11 = -6

balance = 5 + 4 = 9 - 5

$$= 4 \geq 0$$

balance = 4 + 4 - 6

$$= 8 - 6 = 2 \geq 0$$



\Rightarrow $front++ \times$ $balance = balance + p - 1$
 $\boxed{front = rear + 1}$ $= 0 + 6 - 5 = 1 \geq 0 \rightarrow \text{TRUE}$

$balance = 1 + 7 - 6$
 $= 8 - 6 = 2 \geq 0 \rightarrow \text{TRUE}$

$balance = 2 + 4 - 7$
 $= 6 - 7 = -1 \not\geq 0 \rightarrow \text{False}$

Want last
Block \rightarrow 2 visit
single visit

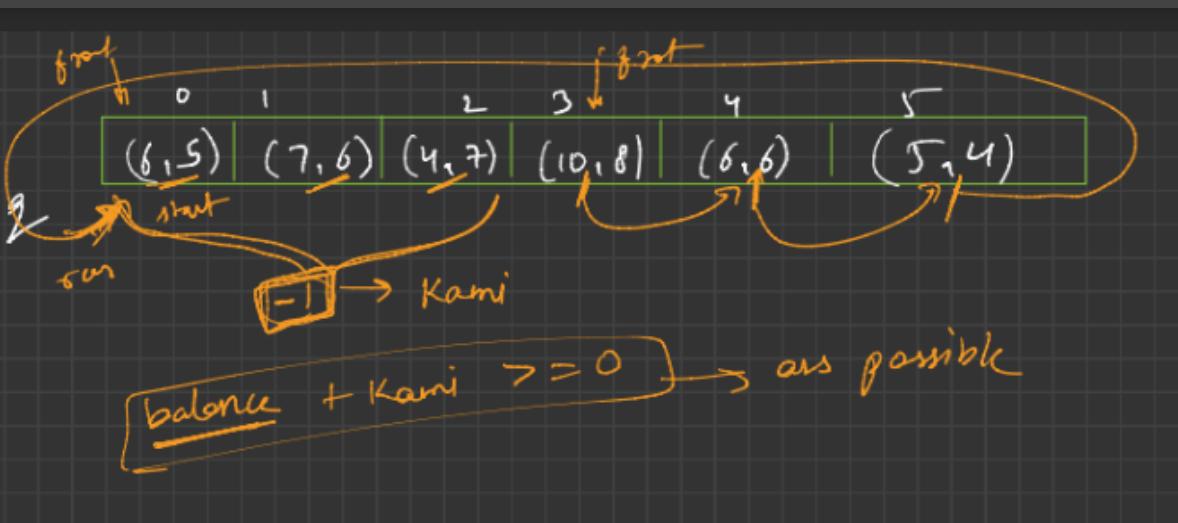
$balance = 0$
 $balance = 0 + 10 - 8$
 $\Rightarrow 2 \geq 0 \rightarrow \text{TRUE}$

$balance = 2 + 8 \not\geq 0$
 $2 \not\geq 0 \rightarrow \text{TRUE}$

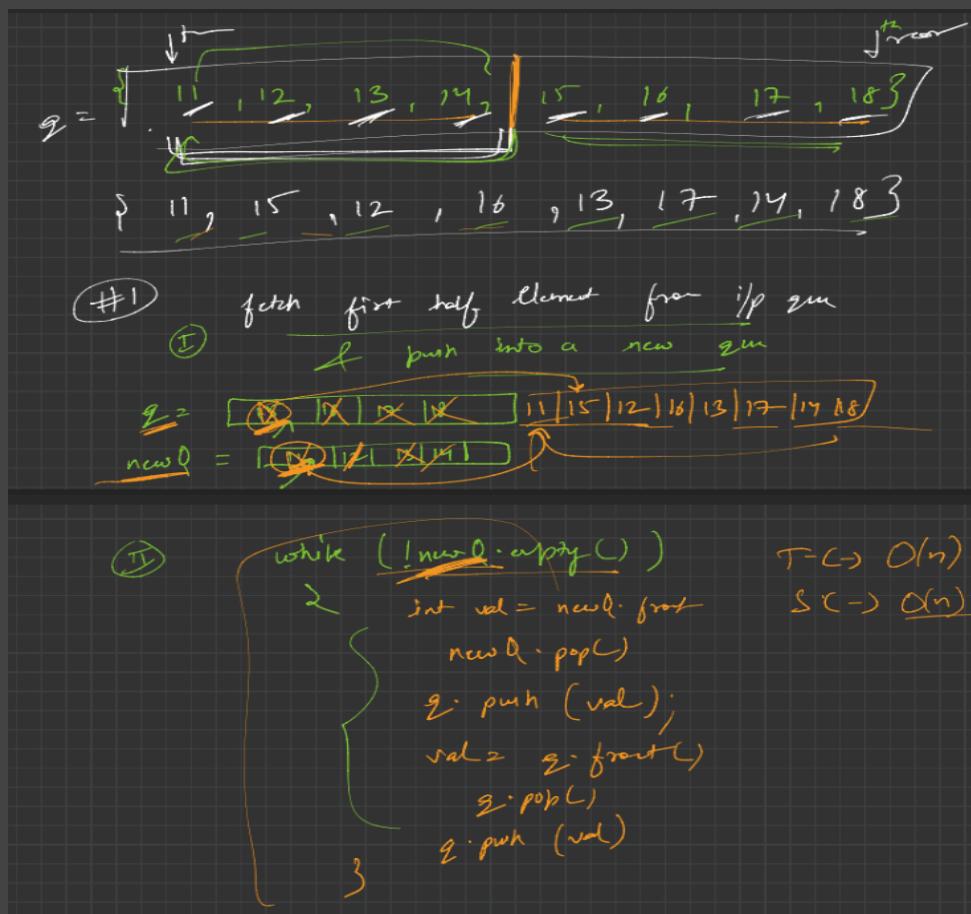
$\neg 3 + 6 \neg 5$
 $\neg 9 - 5 = 4 \geq 0 \rightarrow \text{True}$

$= 4 + 7 - 6$
 $= 11 - 6 = 5 \geq 0 \rightarrow \text{TRUE}$

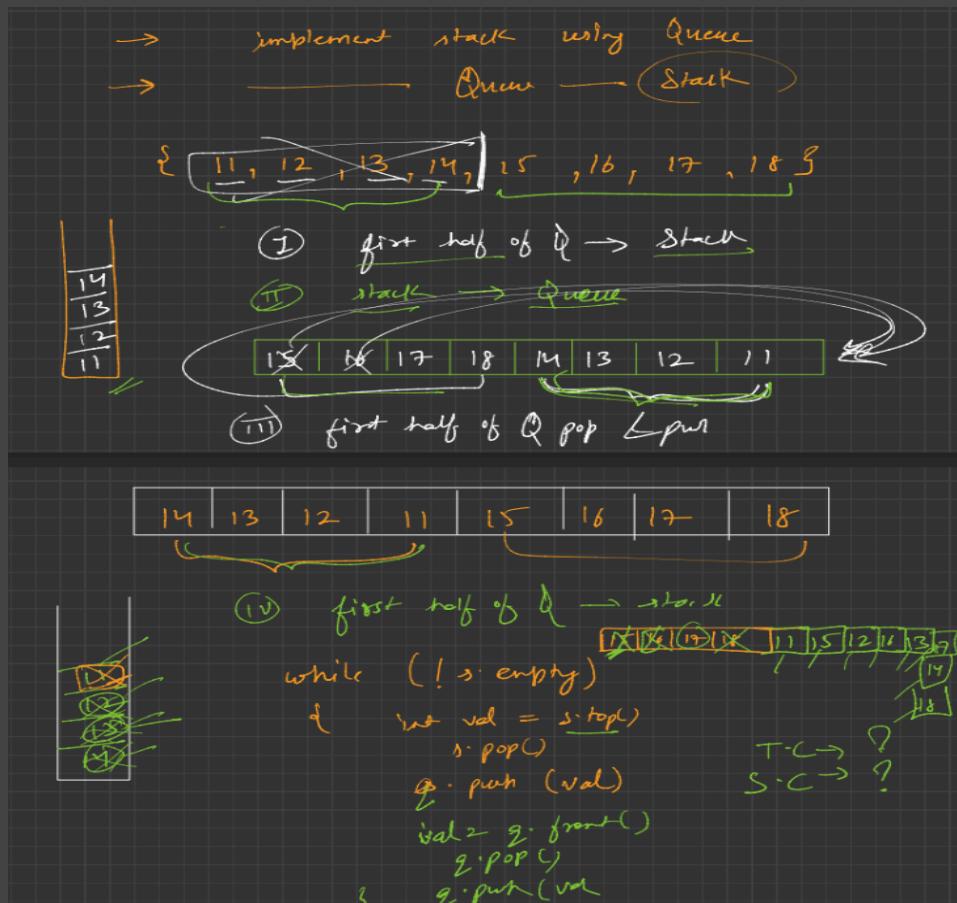
$\neg 5 + 4 - 7$
 $\neg 9 - 7 = 2 \geq 0 \rightarrow \text{TRUE}$



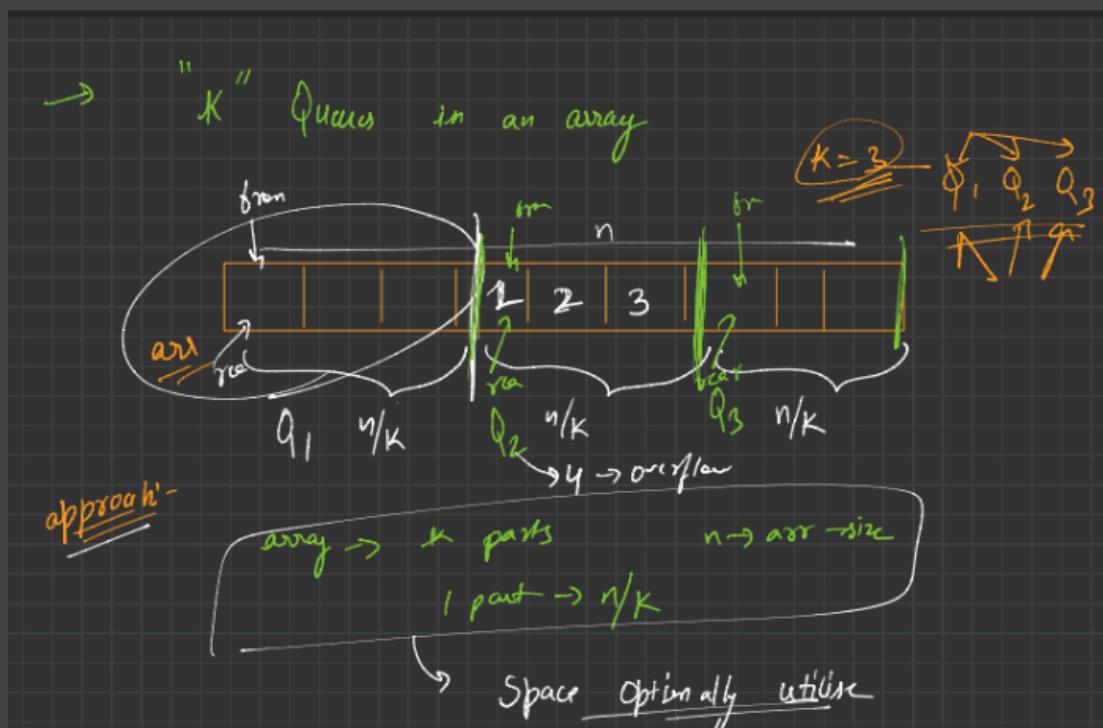
Interleave First and Second Halves Of A Queue:



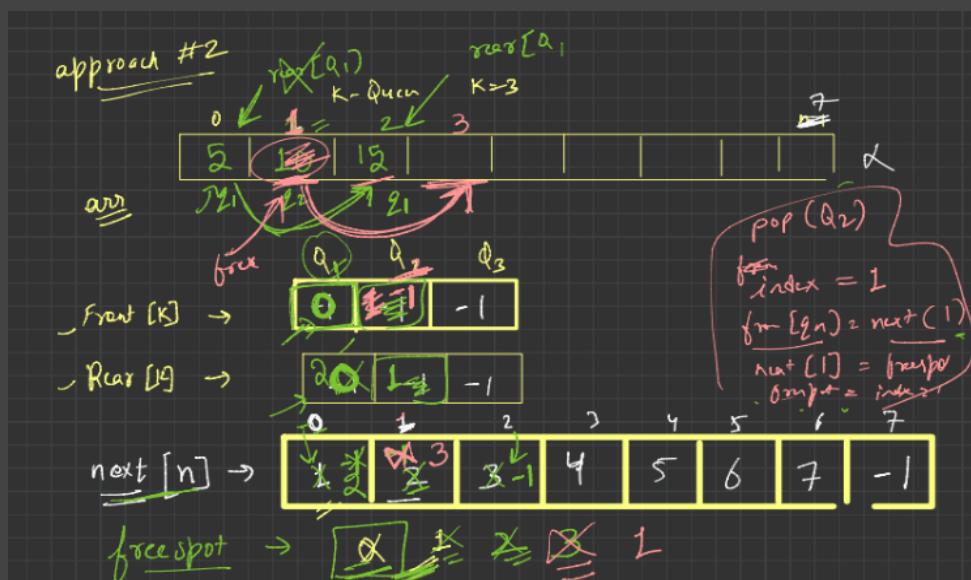
Interleave First and Second Halves Of A Queue Using Stack:



Insert K Queues In An Array Approach-I:



Insert K Queues In An Array Approach-II:



push:-
 → Overflow check → $\boxed{\text{free} = -1}$

→ // find index, where we want to insert

int index = freespot

→ // update freespot
 $\text{freespot} = \text{next}[\text{index}]$

↳ // if first element

if (front [gn] == -1)

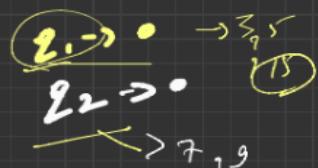
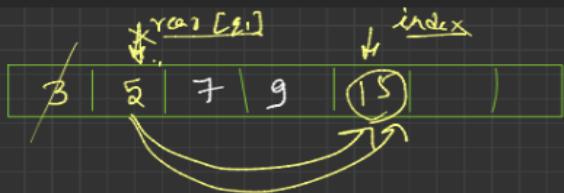
front [gn] = index;

else

{

next [rear [gn]] = index;

}

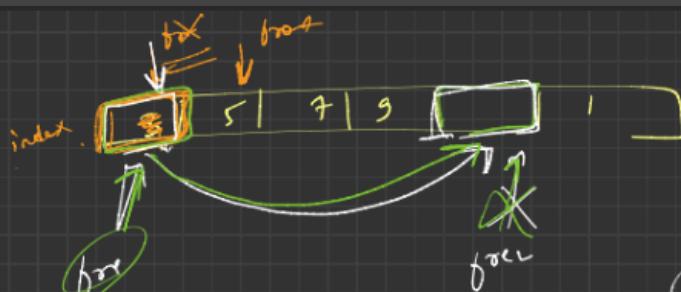


next [rear [gn]] = index

↳ next [index] = -1 ;

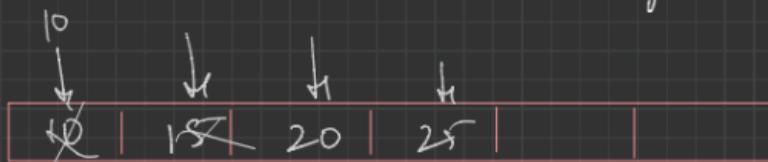
↳ // point rear to index
 rear [gn] = index;

↳ // push element
 arr [index] = n;



next [index] = free

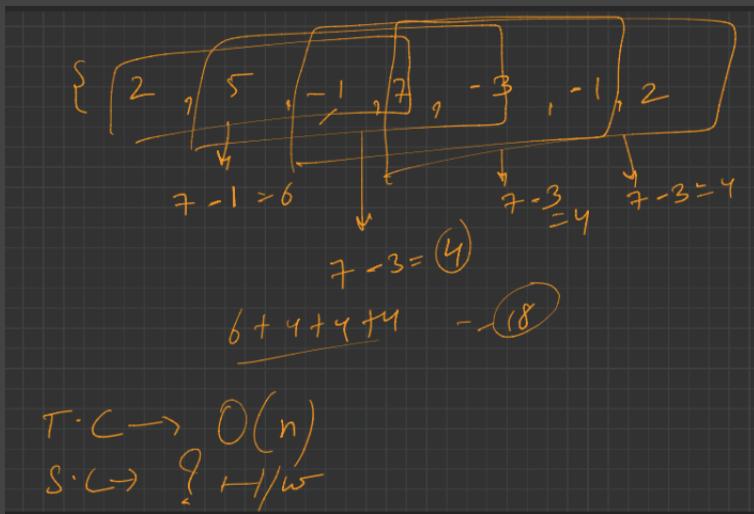
free = index



arr q1 q1 q2 q1

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 deque's maxi and mini after removing elements from the back of the deque 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 deque. Then it adds the current element to the deque after removing elements from the back of the deque that are not needed. This process also takes constant time for each element, so for $n-k$ elements, it takes $O(n-k)$ time.

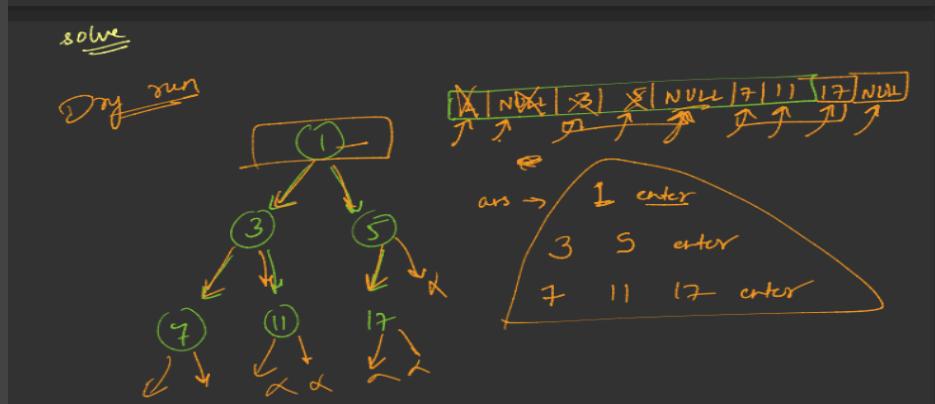
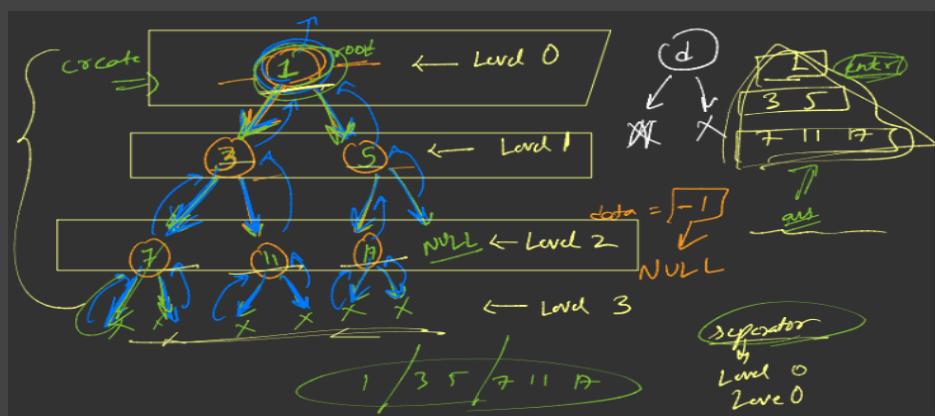
Since the code processes k elements in the first loop and $n-k$ elements in the second loop, the total time complexity is $O(k) + O(n-k) = O(n)$, where n is the total number of elements in the array. This is why we say each element in the array is processed only once in a single pass. The operations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

Binary Tree

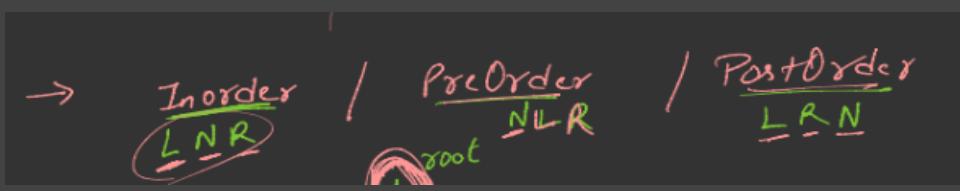
Basic Terms Of Binary Tree:



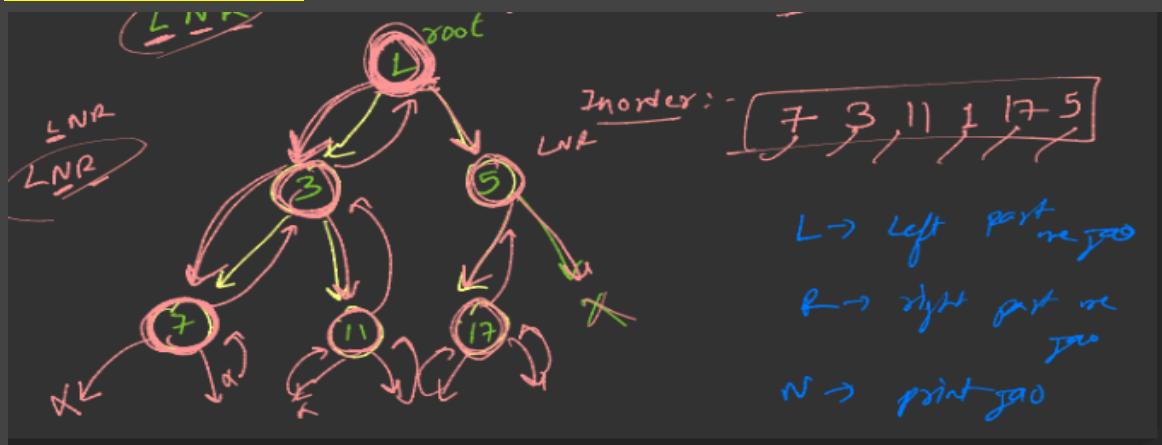
Creation Of Binary Tree & Level Order Traversal(LOT):



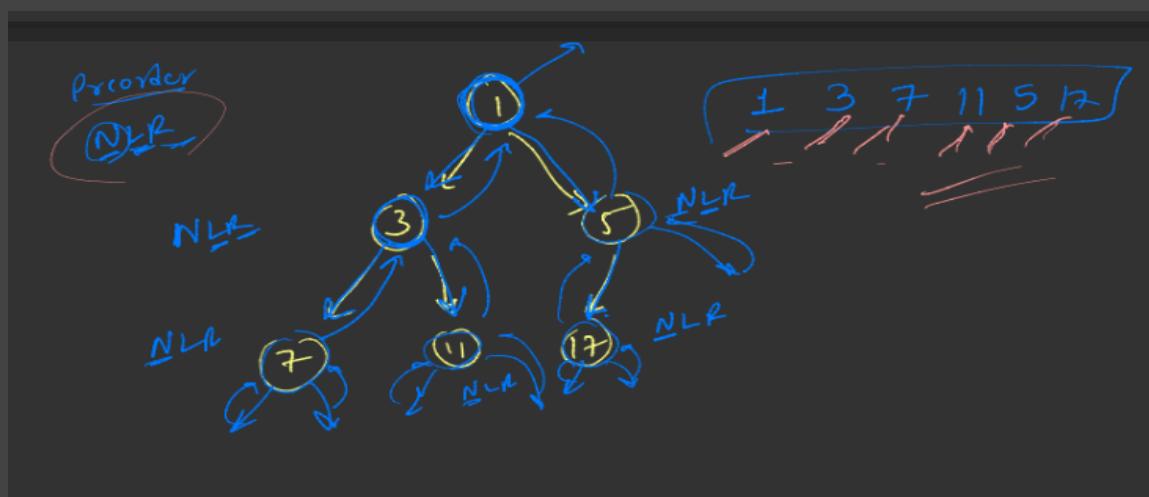
Reverse Level Order Traversal:



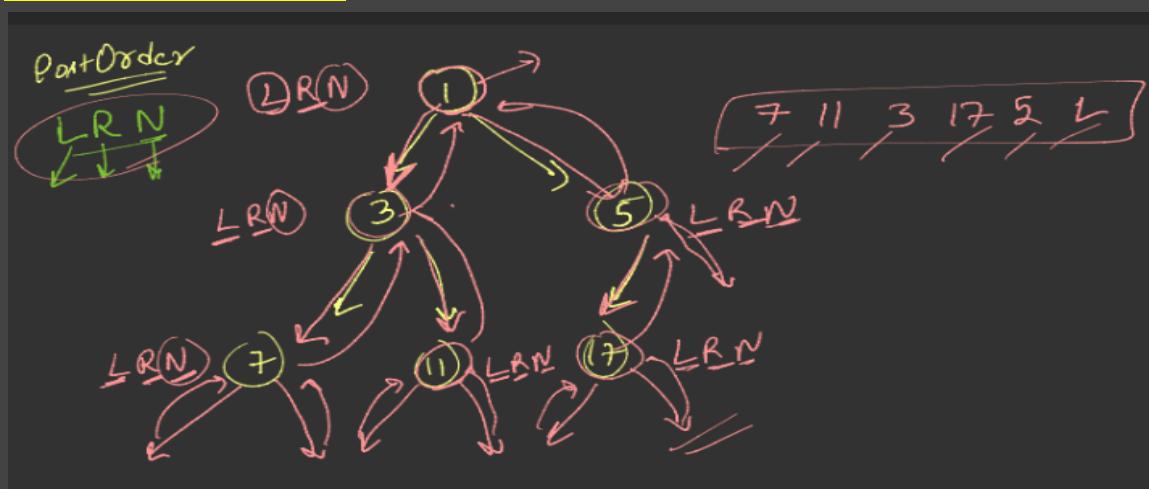
Inorder Traversal:



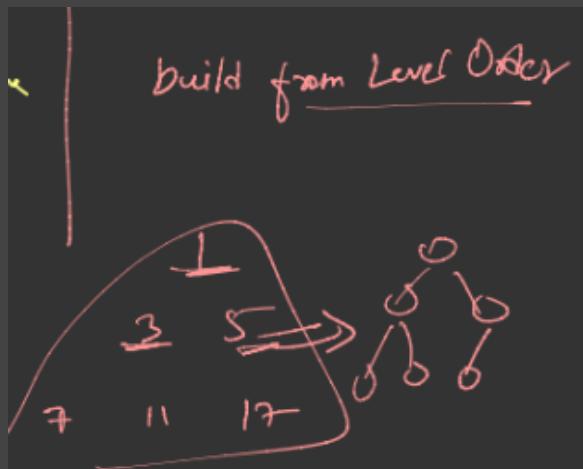
Preorder Traversal:



Postorder Traversal:

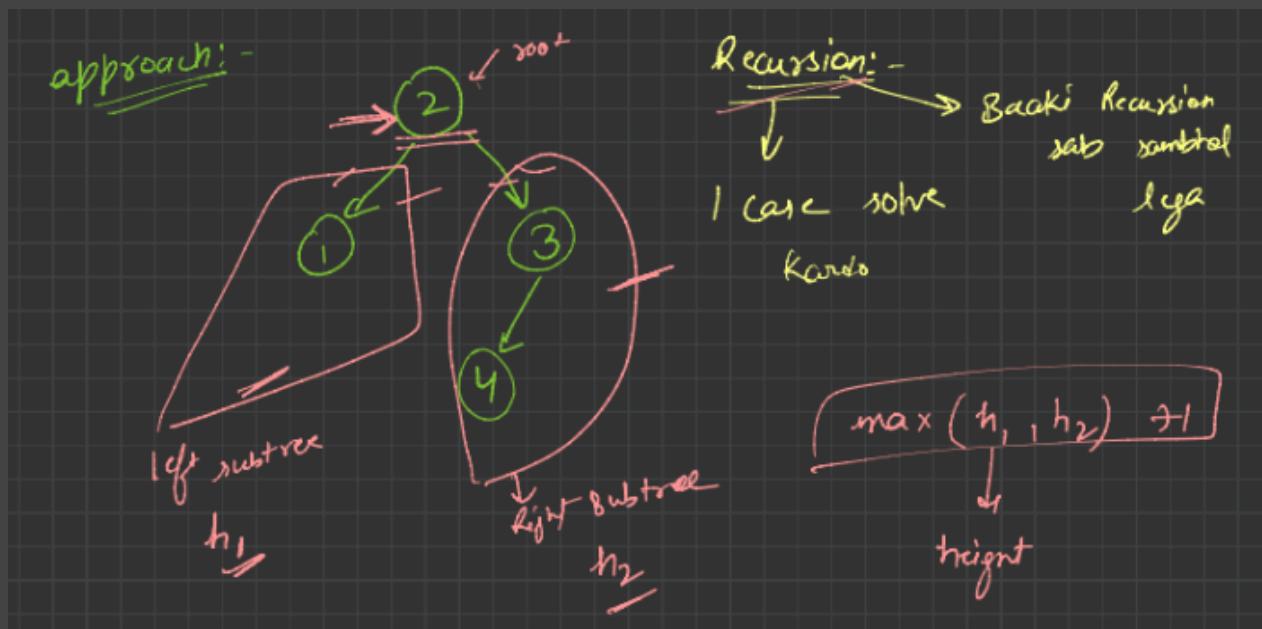
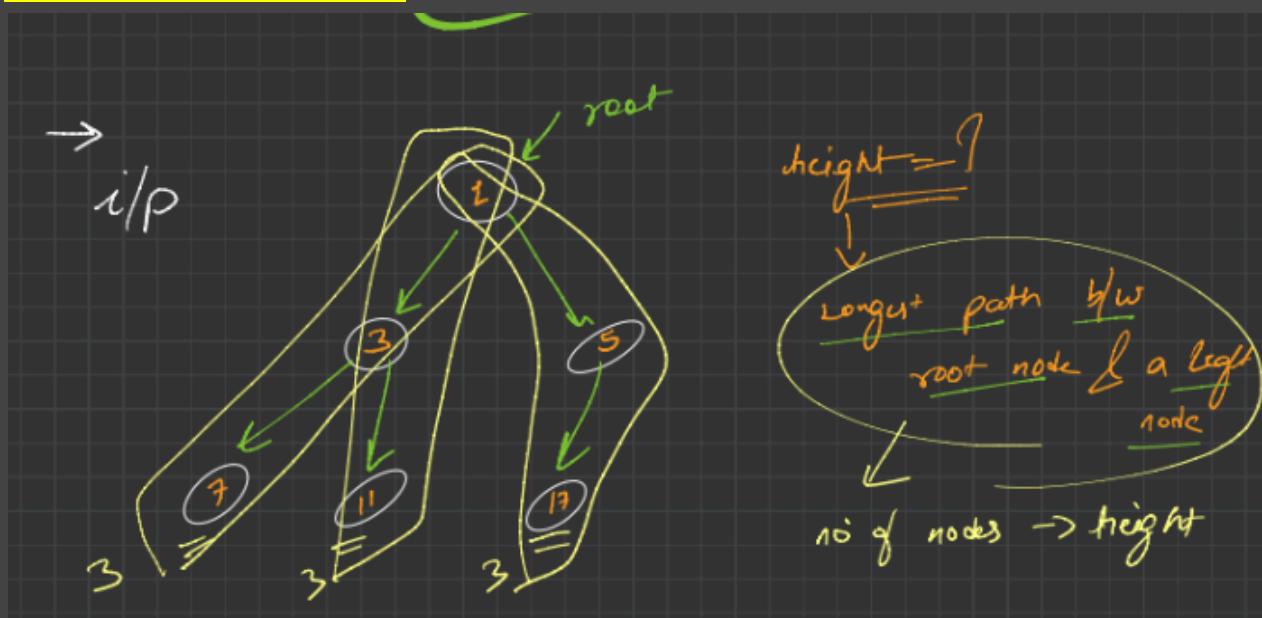


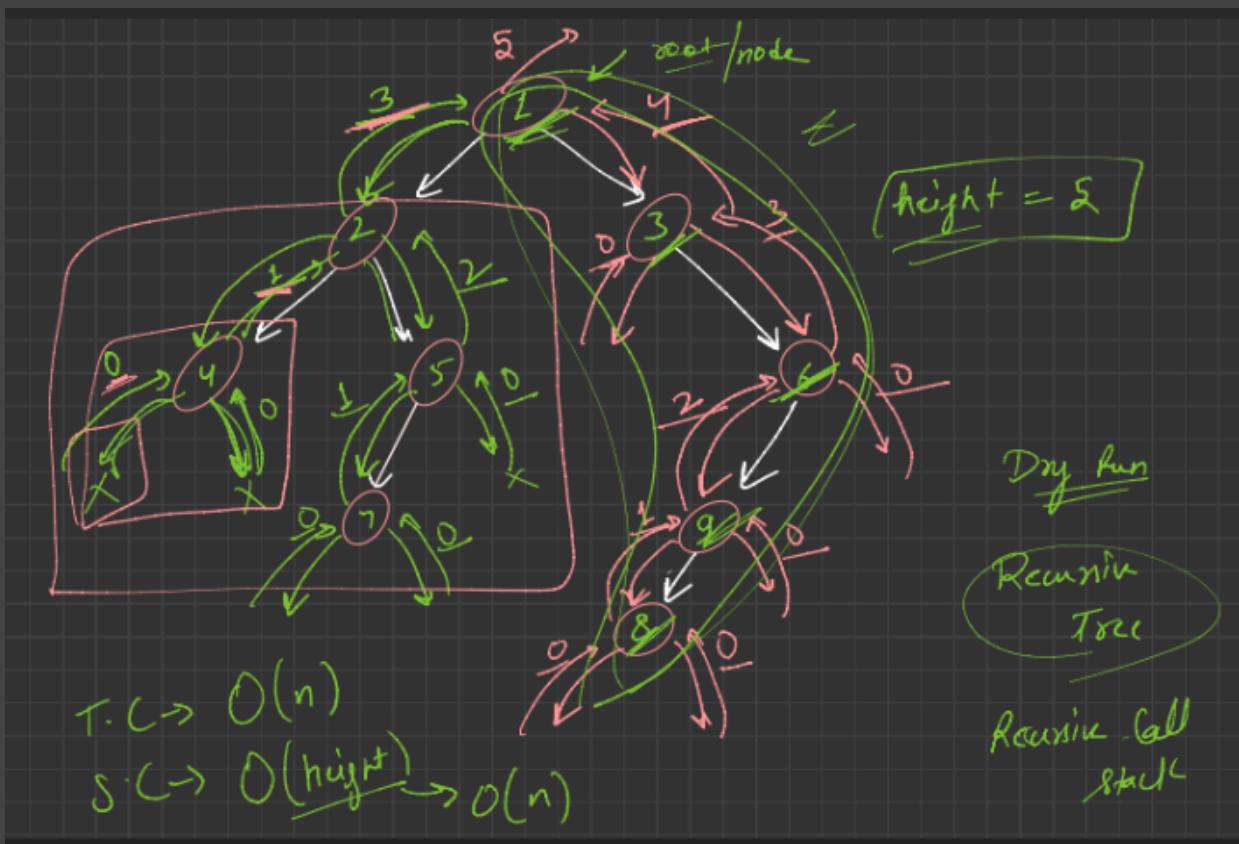
Build Binary Tree From Level Order:



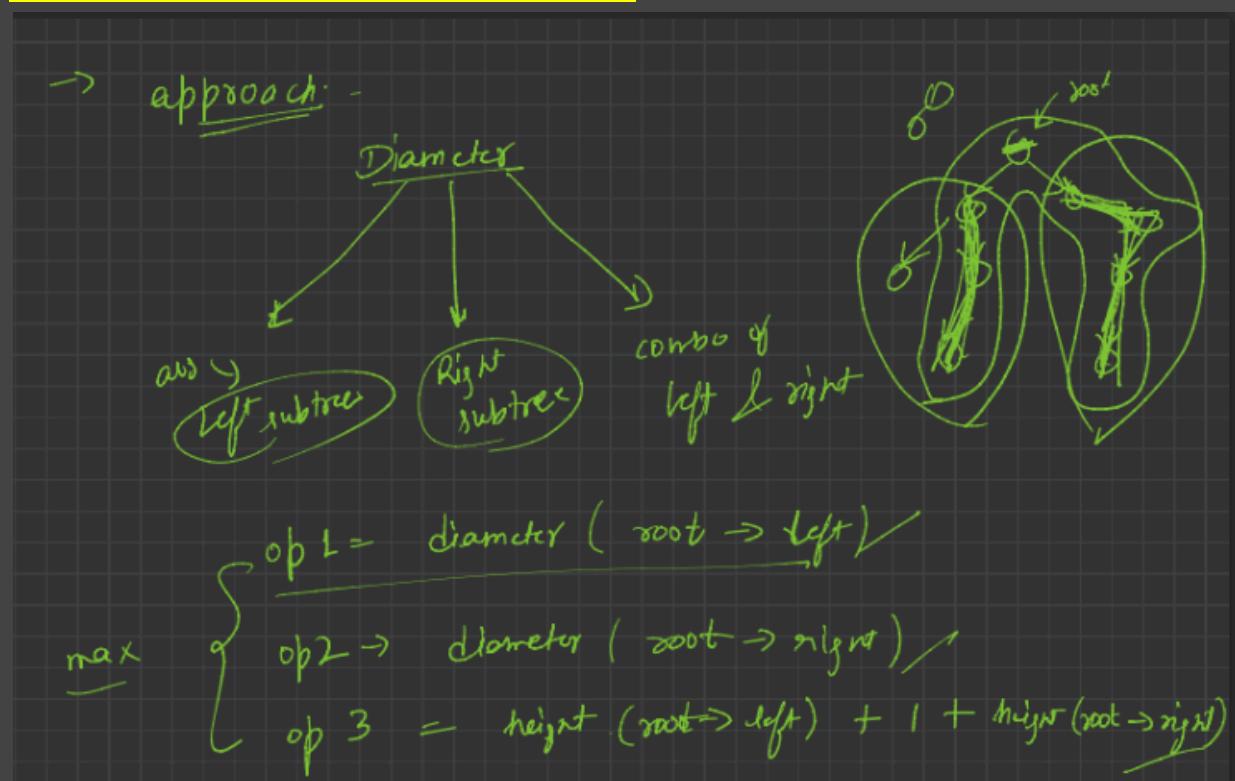
Count Leaf Nodes:

Height Of A Binary Tree:

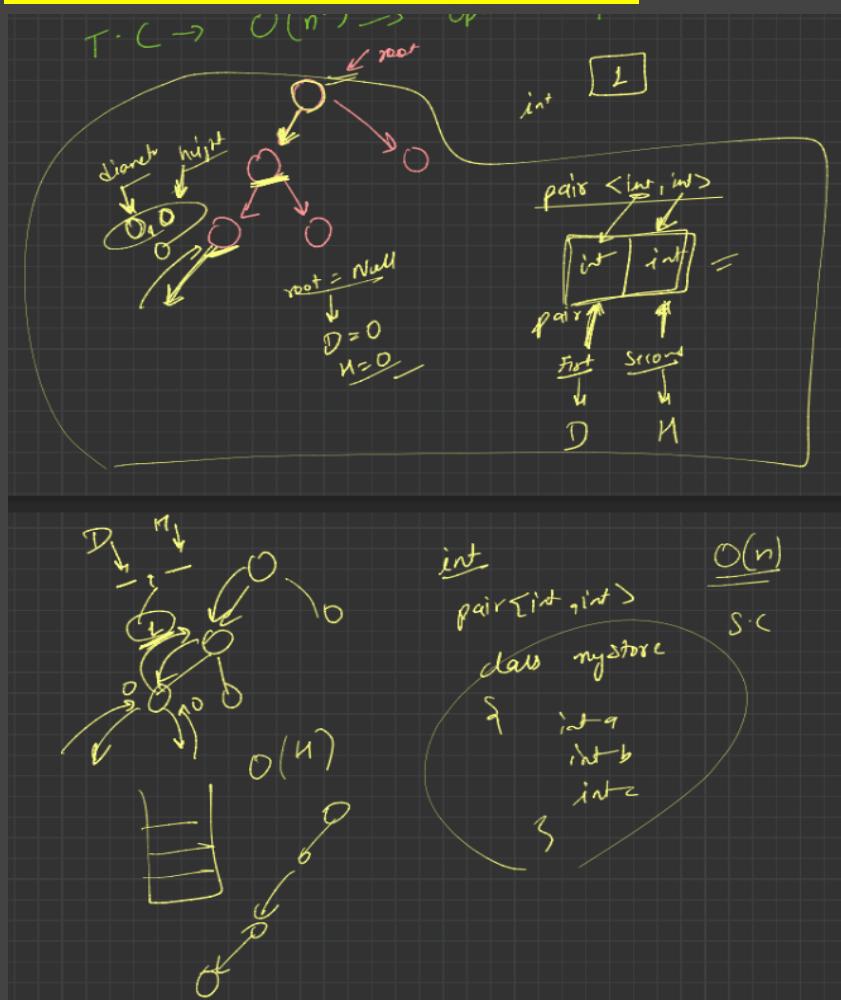




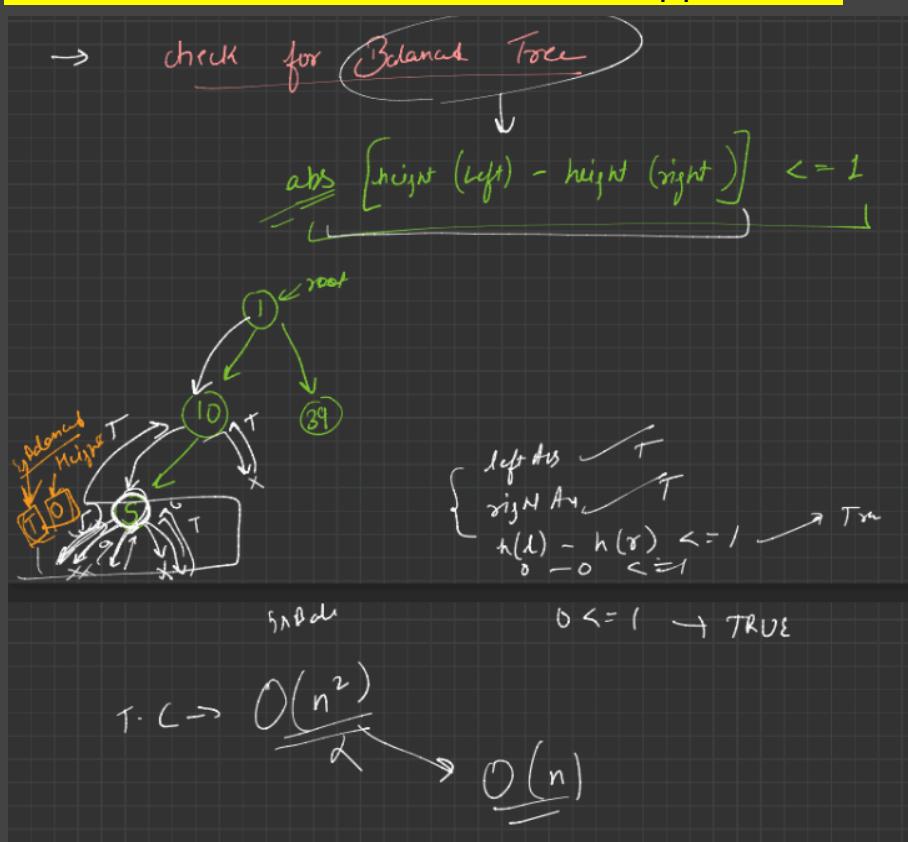
Diameter Of Binary Tree Approach-I:



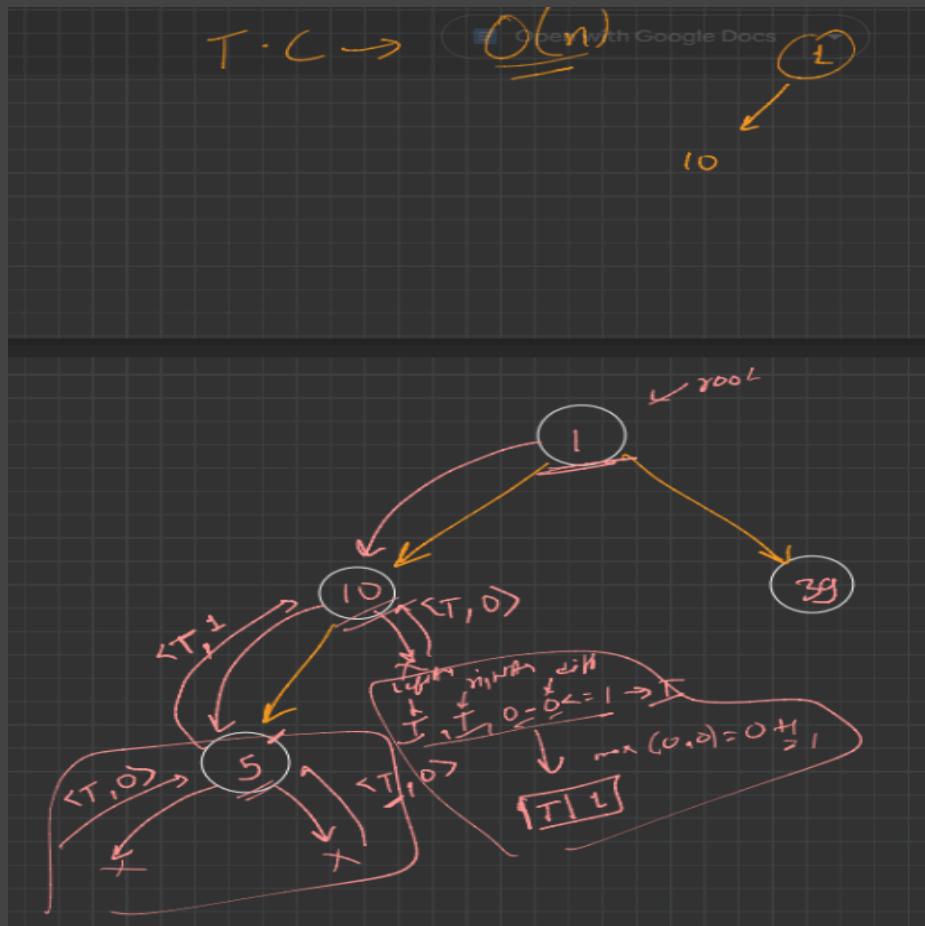
Diameter Of Binary Tree Approach-II:



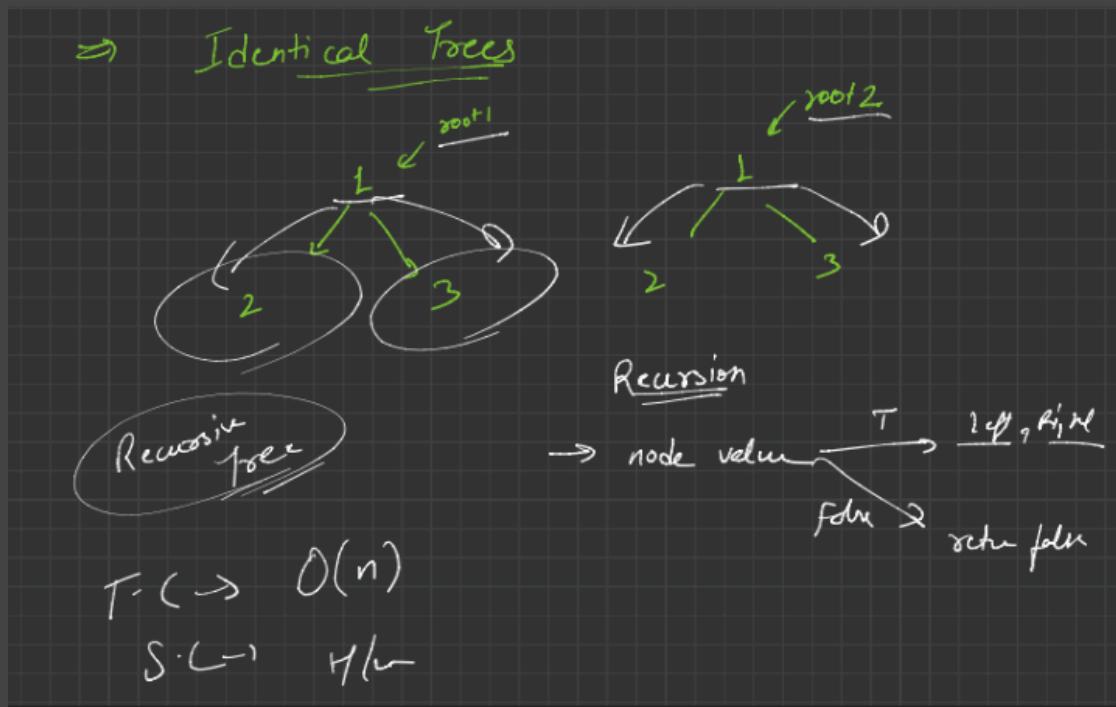
Check If The Tree Is Balanced Or Not Approach-I:



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

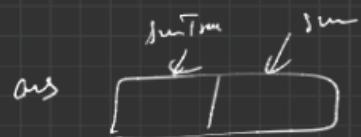
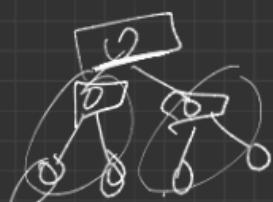
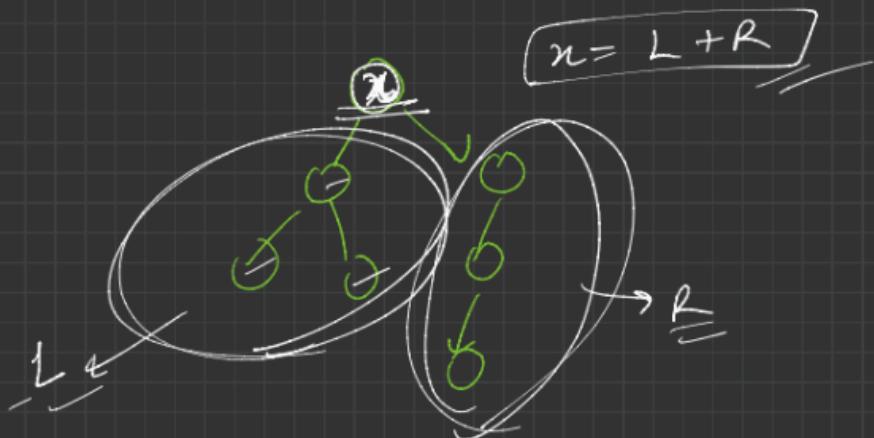


Check Whether The Two Trees Are Identical Or Not:



Sum Tree:

→ check Sum Tree or Not



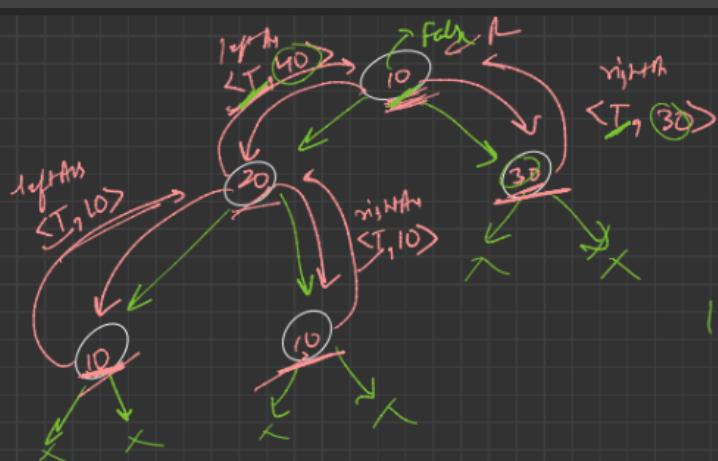
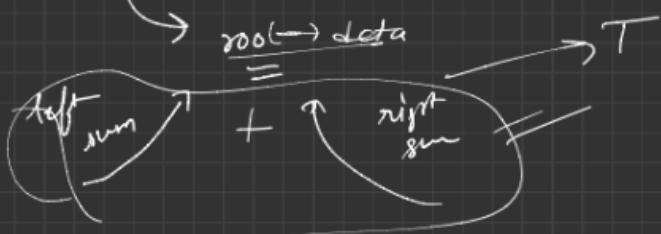
→ isSum ($\text{root} \rightarrow \text{left}$) → T

→ isSum ($\text{root} \rightarrow \text{right}$) → T

isSum → T/F
sum → int

$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

currNode



$$20 = 10 + 10$$

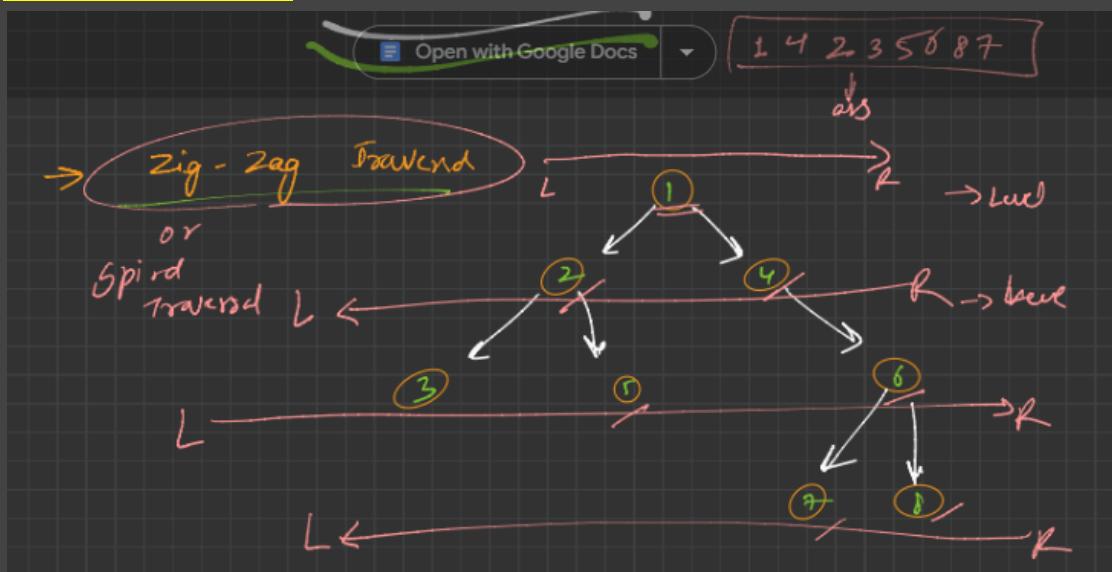
$$10 = 40 + 30$$

$$10 = 70$$

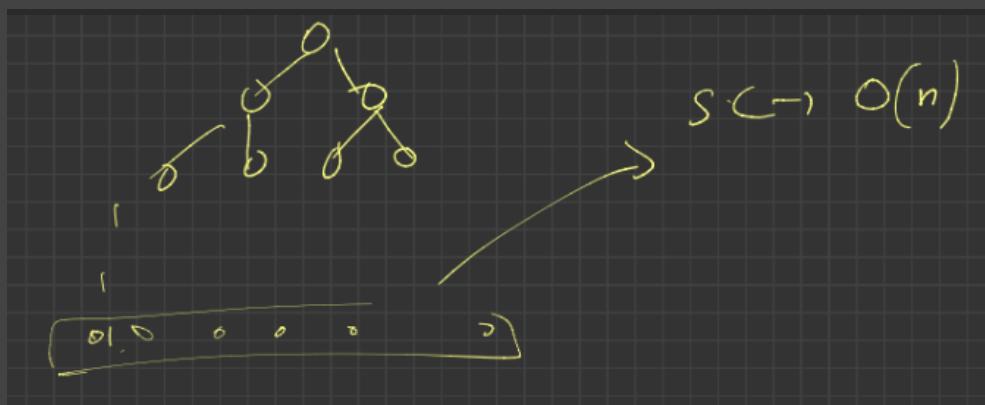
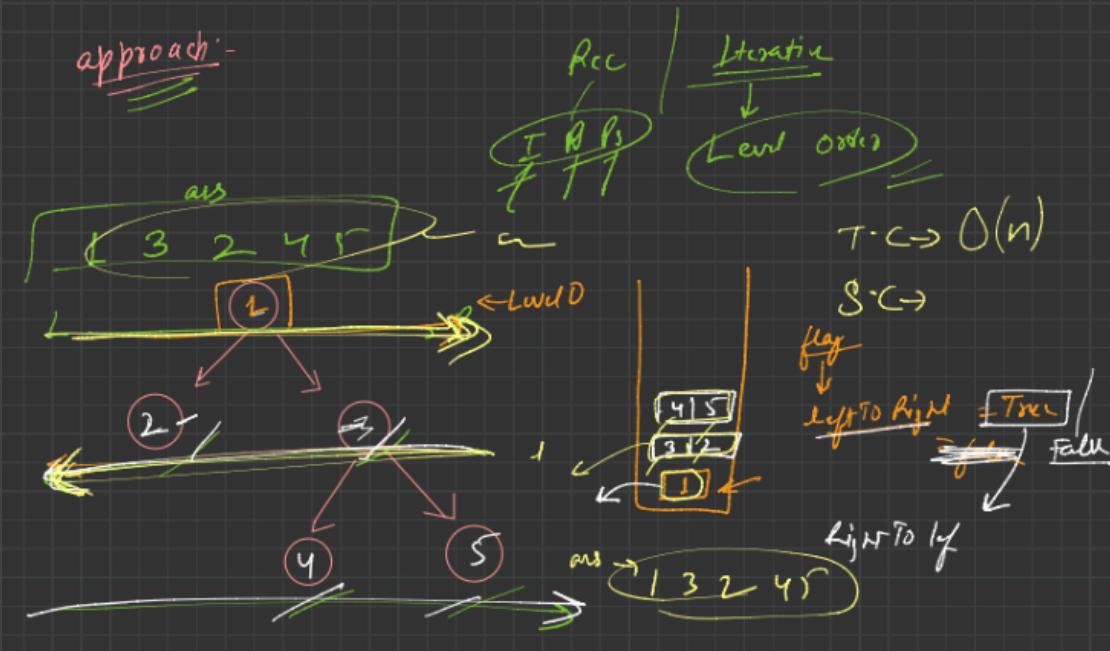
T.C → O(n)

S.C → O(H)

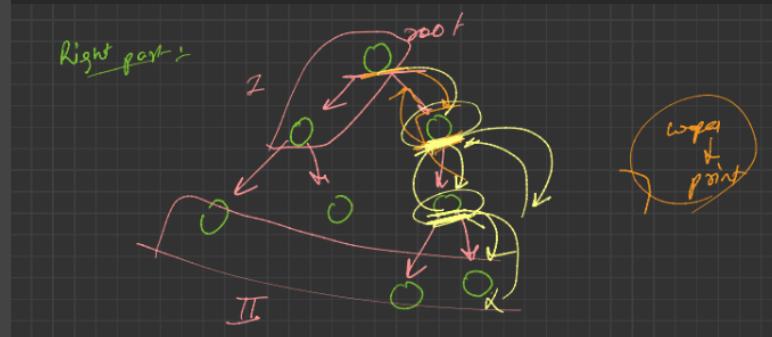
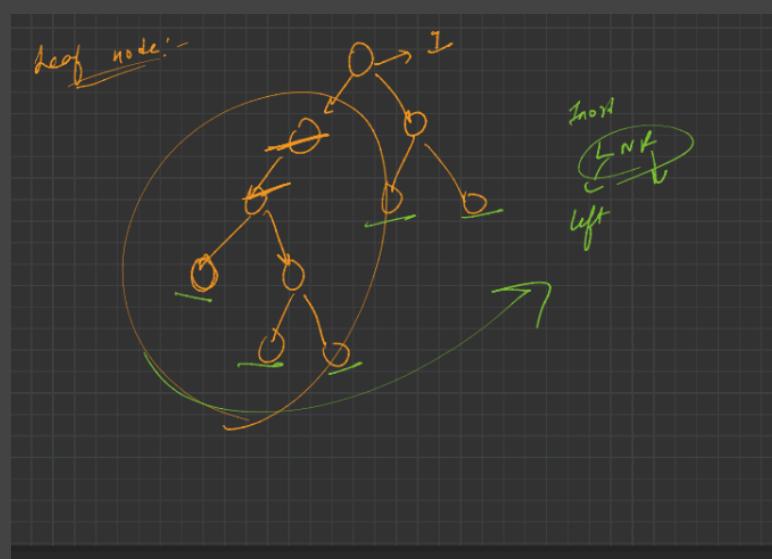
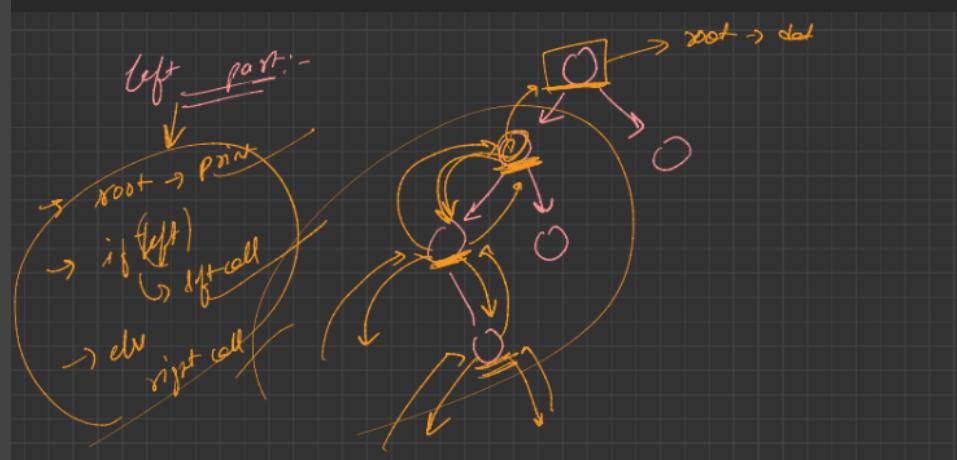
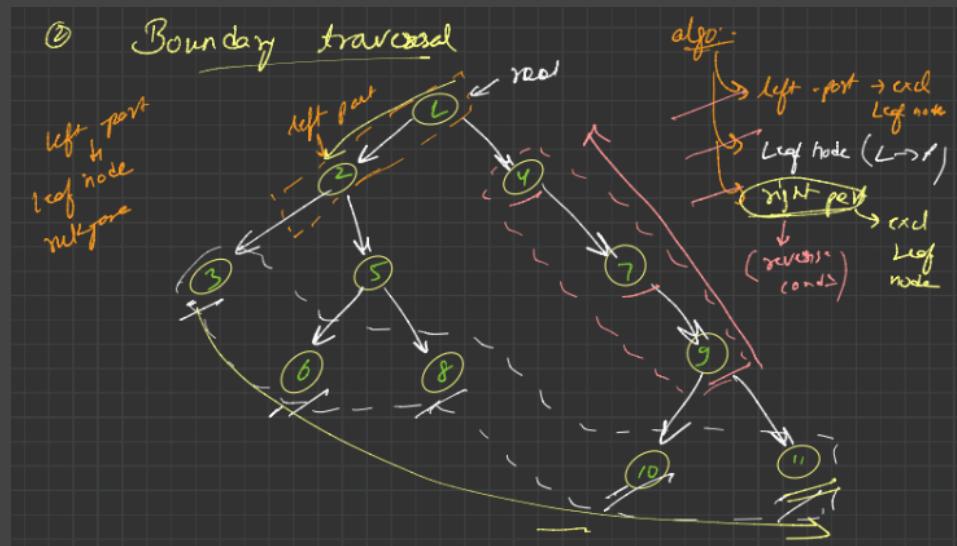
Zigzag Traversal:



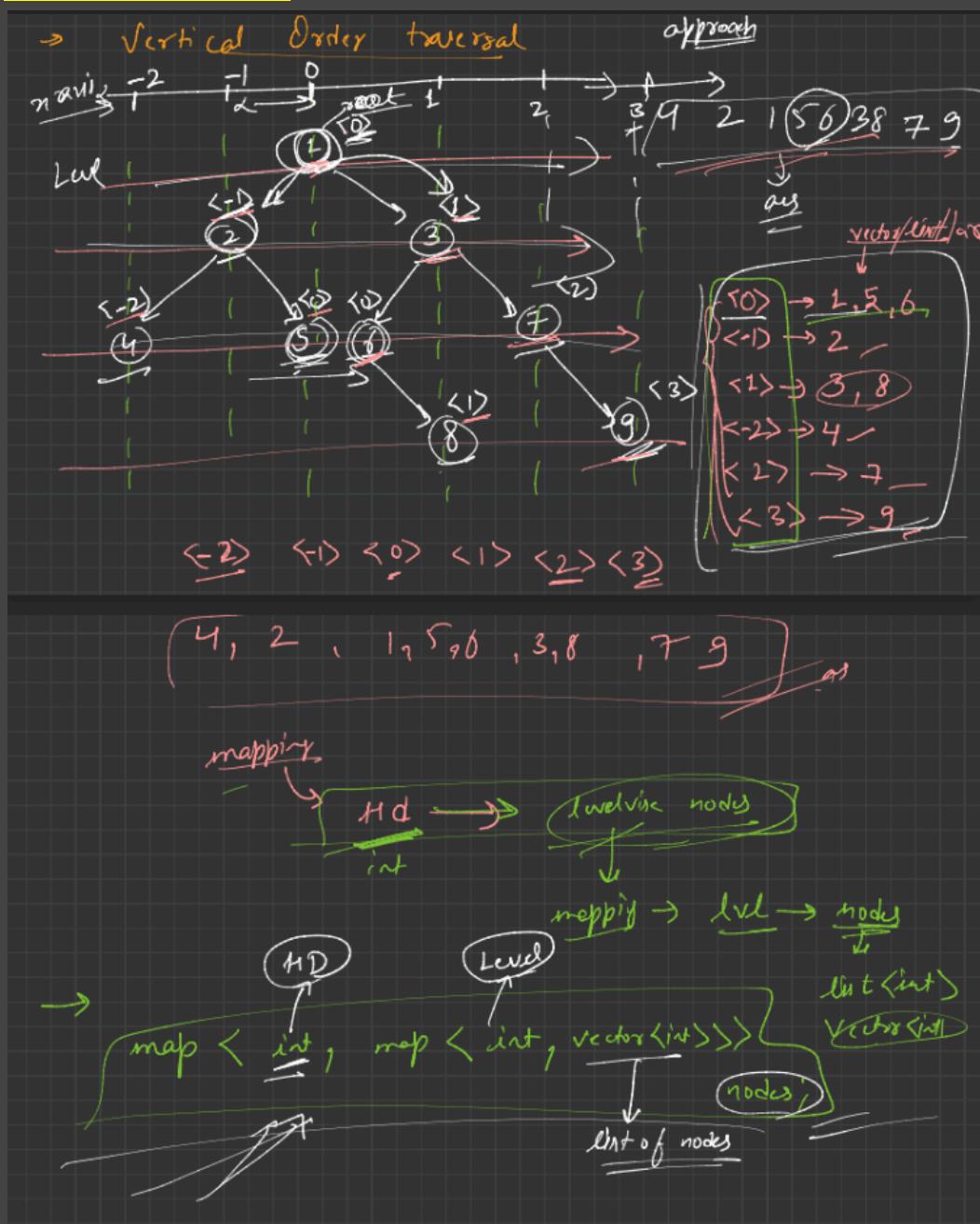
approach:-



Boundary Traversal:



Vertical Traversal:



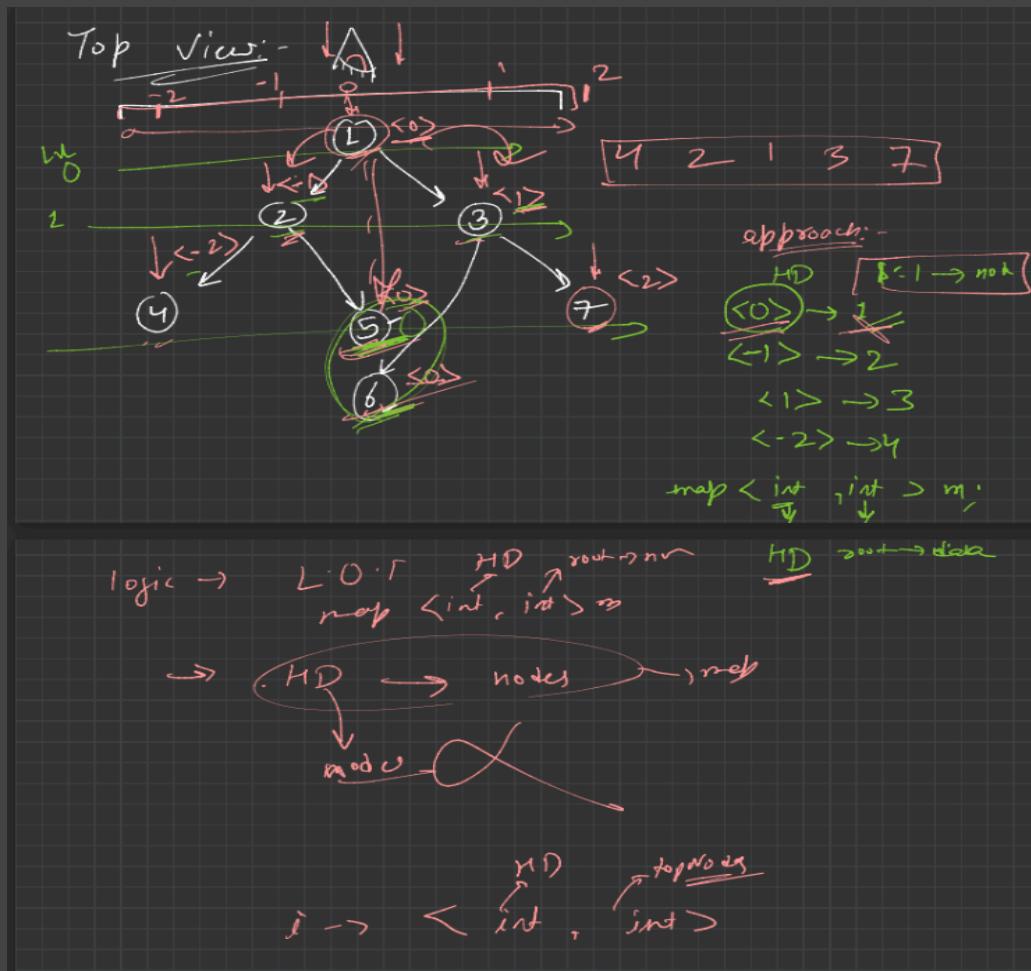
→ required \rightarrow (HD, Lvl)

→ queue<pair<Node*, pair<int, int>>>;

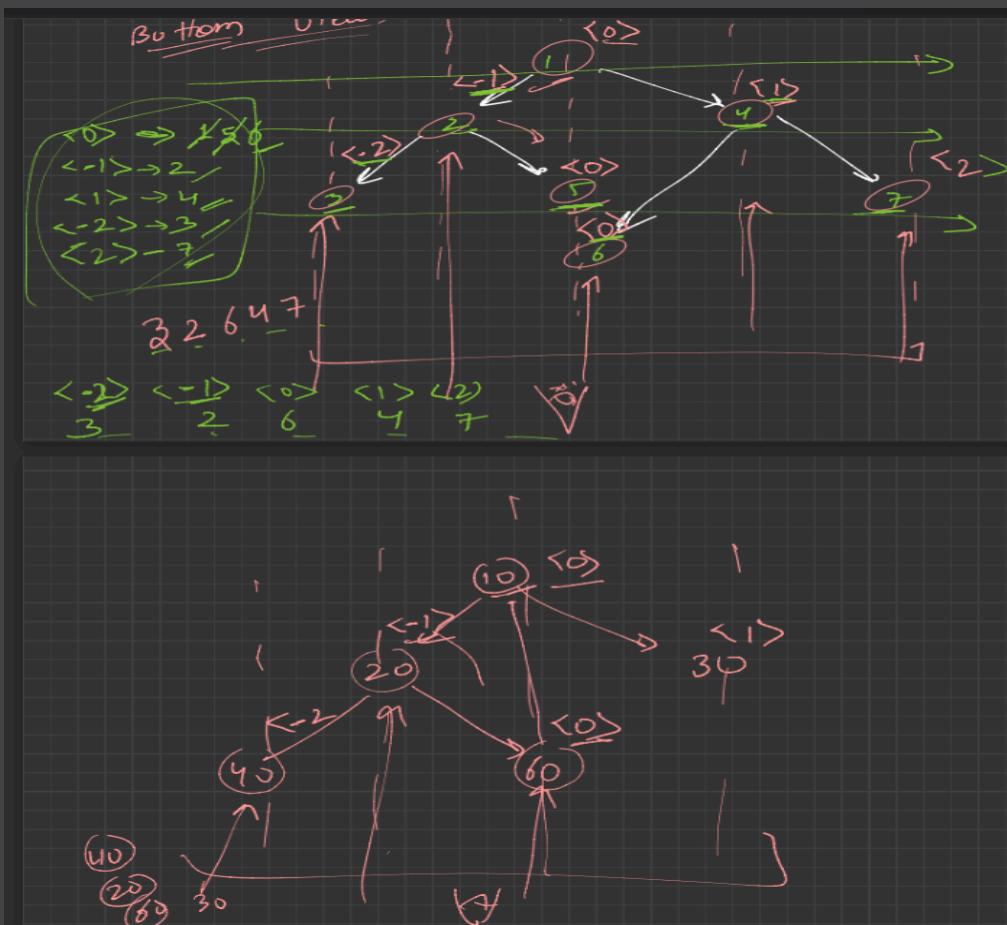
i \rightarrow < int, map >

j \rightarrow < int, vector<int>>

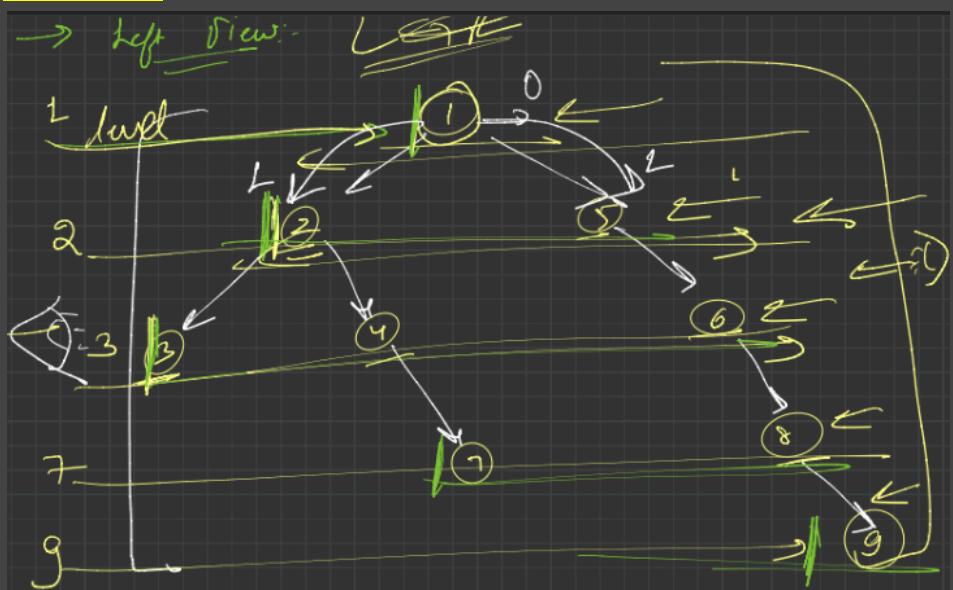
Top View:



Bottom View:



Left View:



Approach :-

$L \rightarrow R$

level → first node print

L.O.T

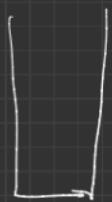
Recursive

I^{th} node → ?

level → stack
int level
next level

func (root, lvl)

{ // base case
if (root == NULL)
return;



you enter it
into & never
exit

if (lvl == Vector.size())
Vector
vector-store (root->val)

f(left, lvl + 1)

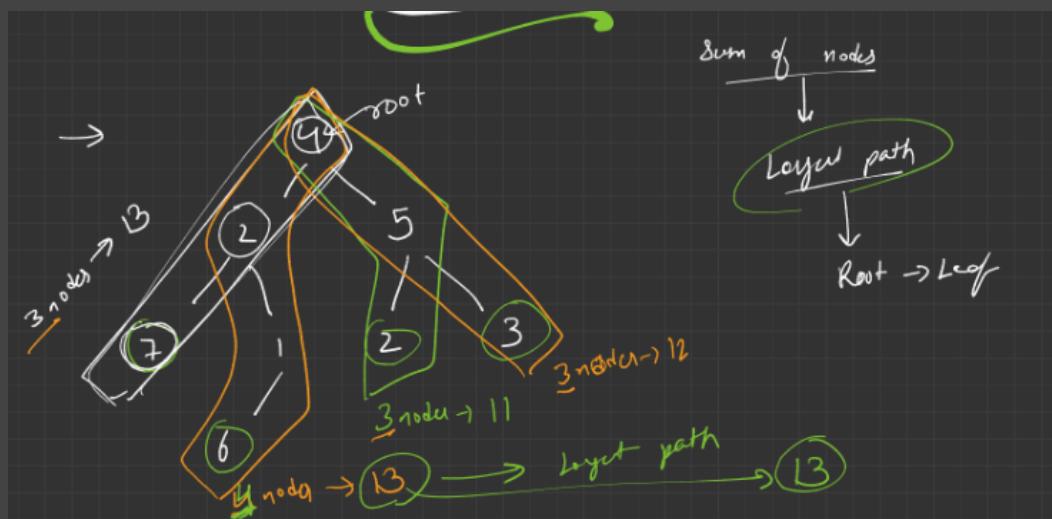
f(right, lvl + 1)

Right View:

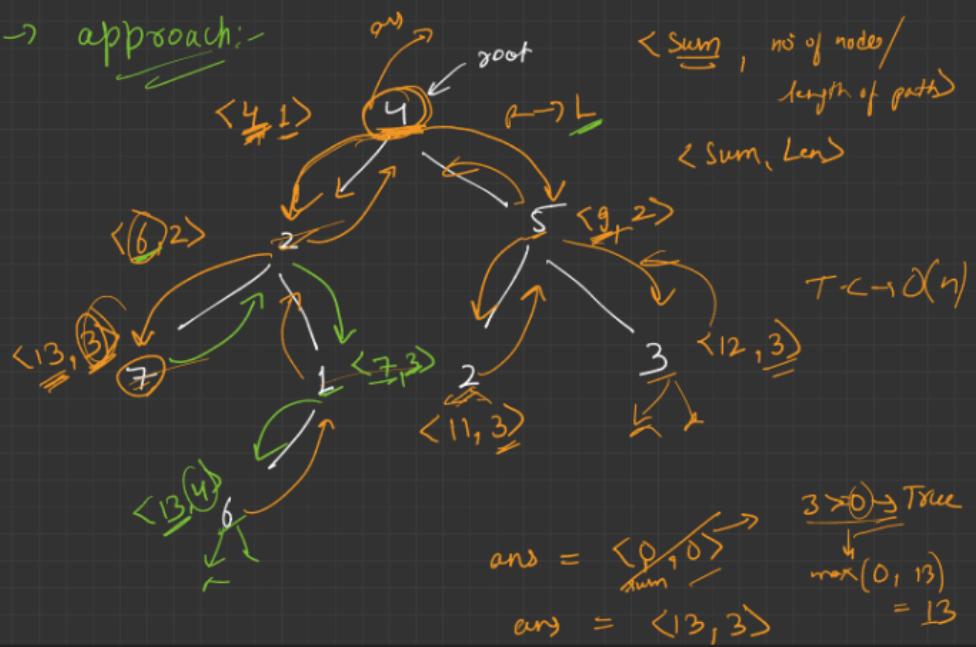
Just make the recursive call for right first and then left.

Diagonal Traversal:

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

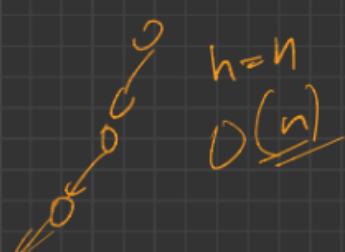


→ approach:-



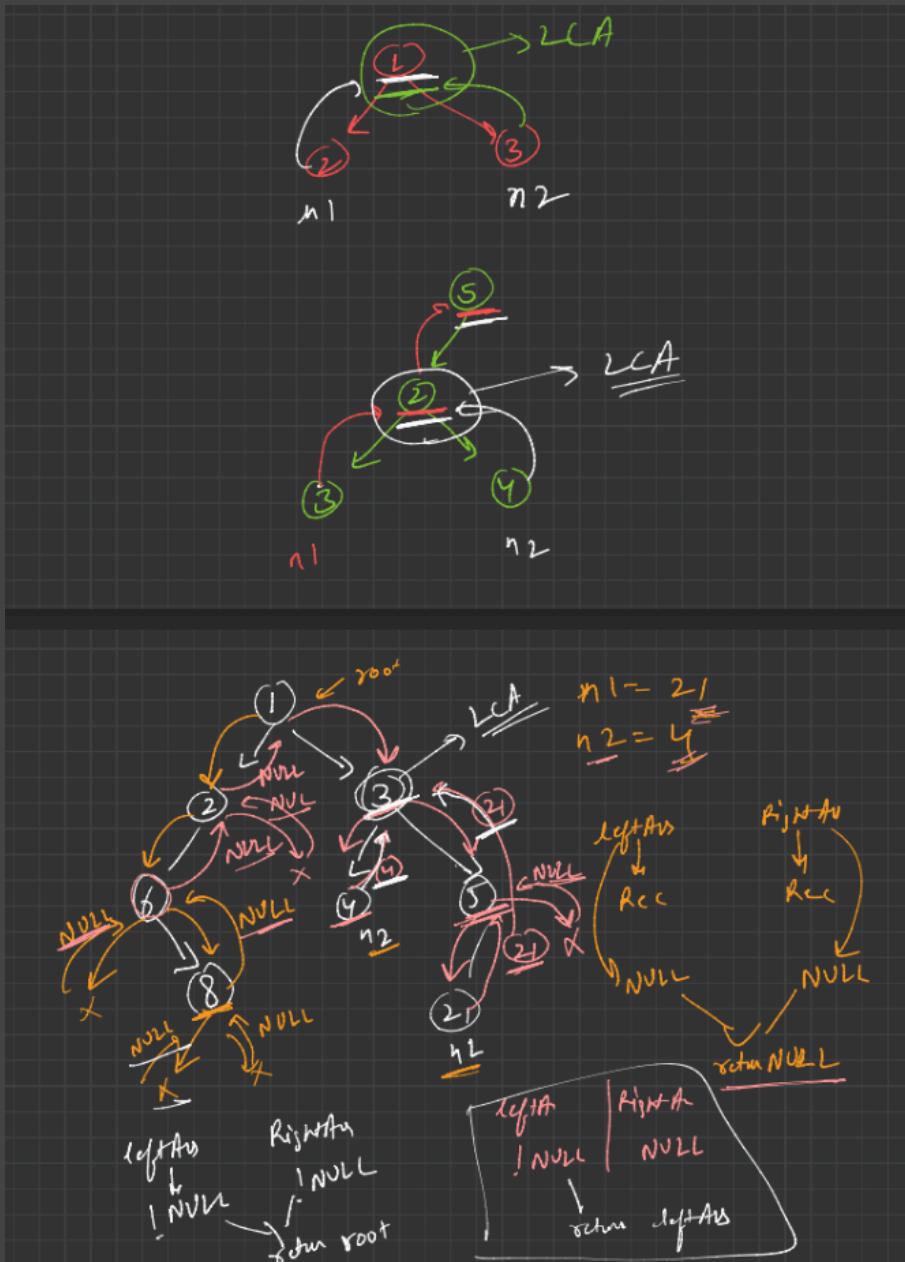
$$\text{ans} = \begin{cases} <0, 0> & 3 > 0 \rightarrow \text{True} \\ \max(0, 13) & = 13 \end{cases}$$

$S.C \sim O(n)$

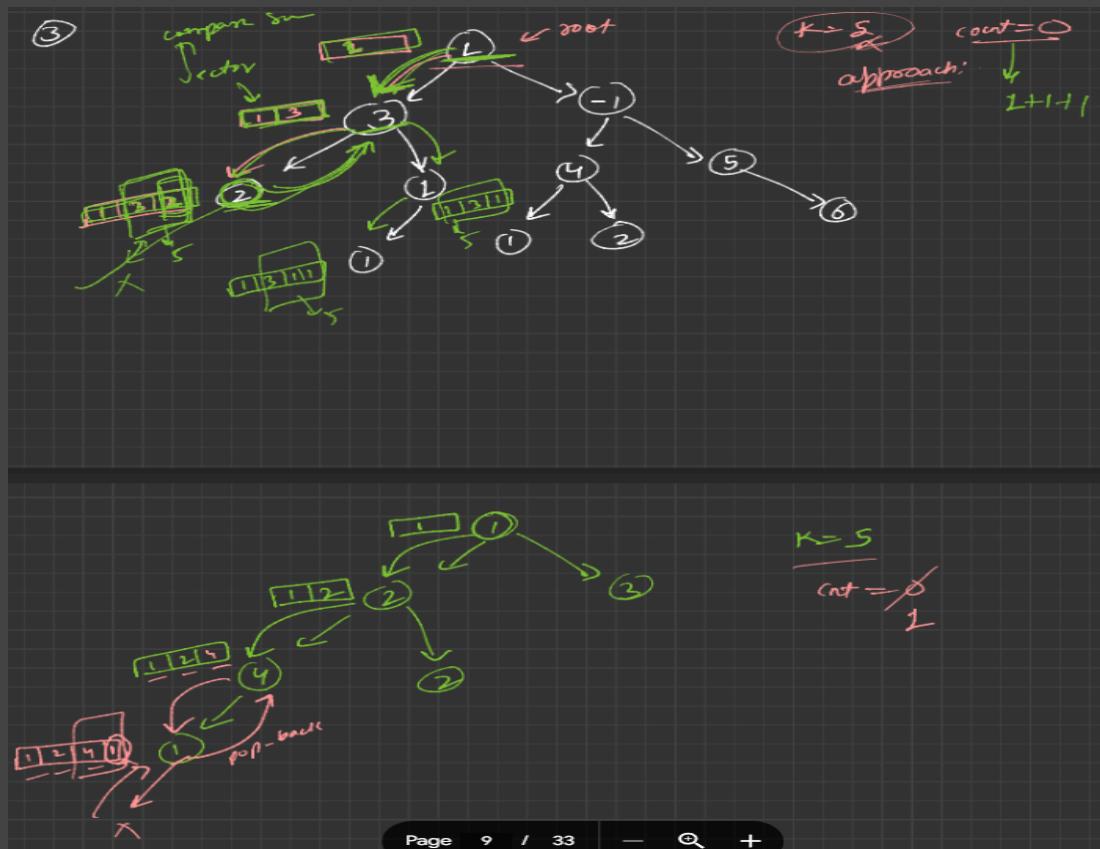


$$\begin{aligned} \text{ans} &= <13, 3> & 4 &= 3 \rightarrow \text{TRUE} \\ &= <13, 4> & \max(13, 13) &= 13 \\ && \text{sum} & \\ && \text{len} & \\ 3 > &= 4 \rightarrow \text{False} \\ 3 > &= 4 \rightarrow \text{False} \end{aligned}$$

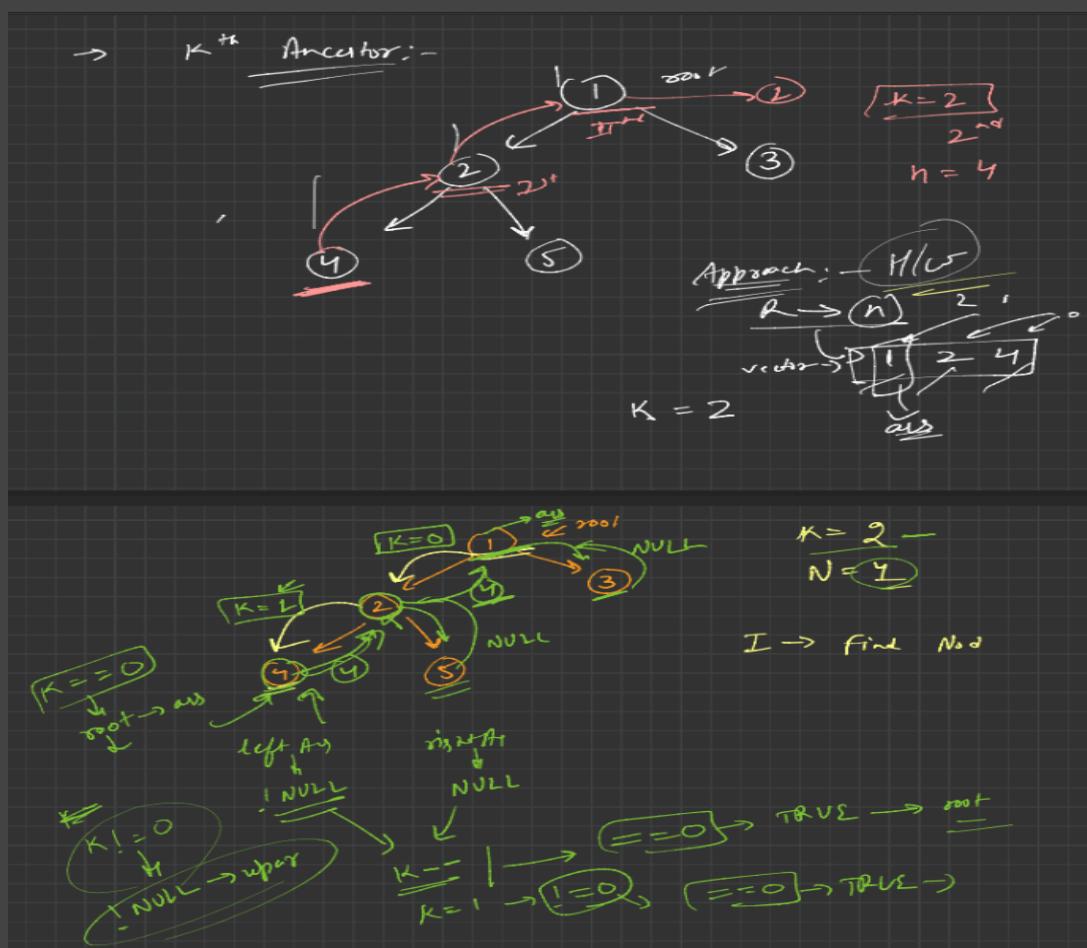
Lowest Common Ancestor:

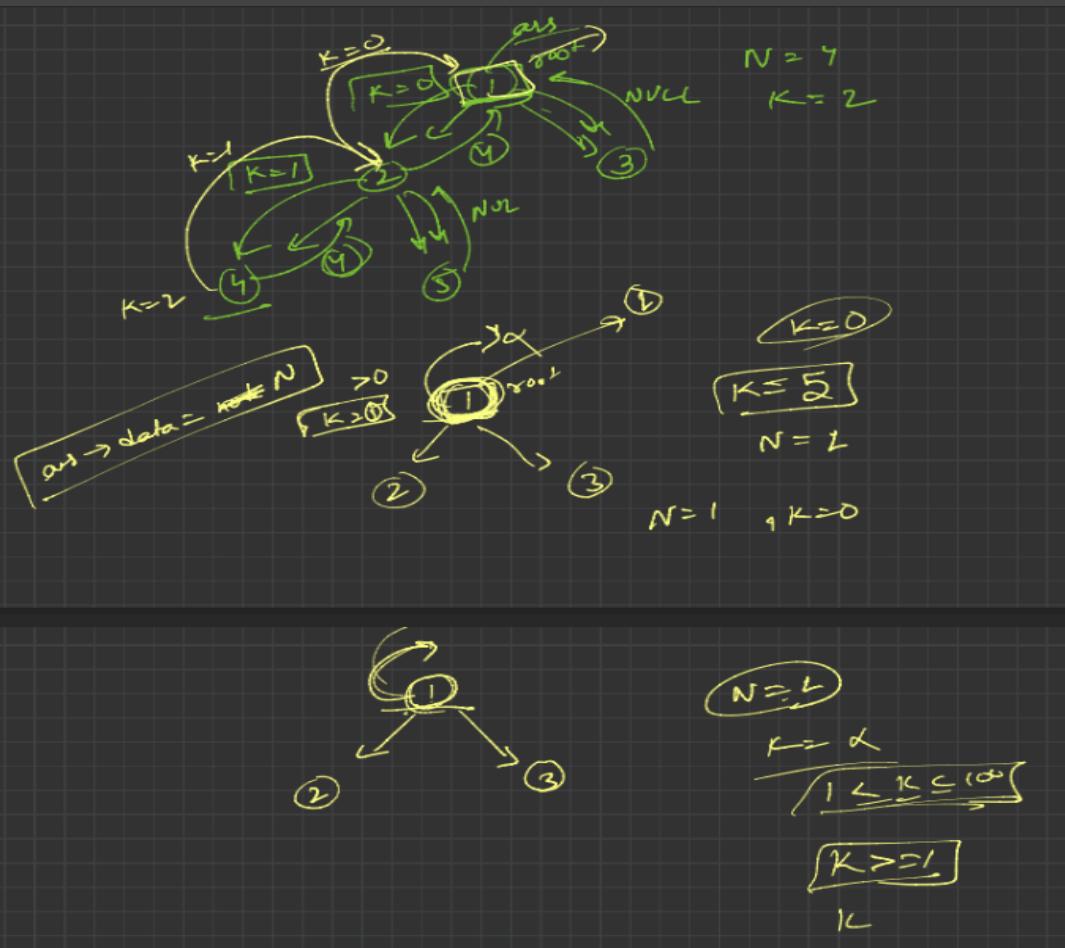


K Sum Paths:

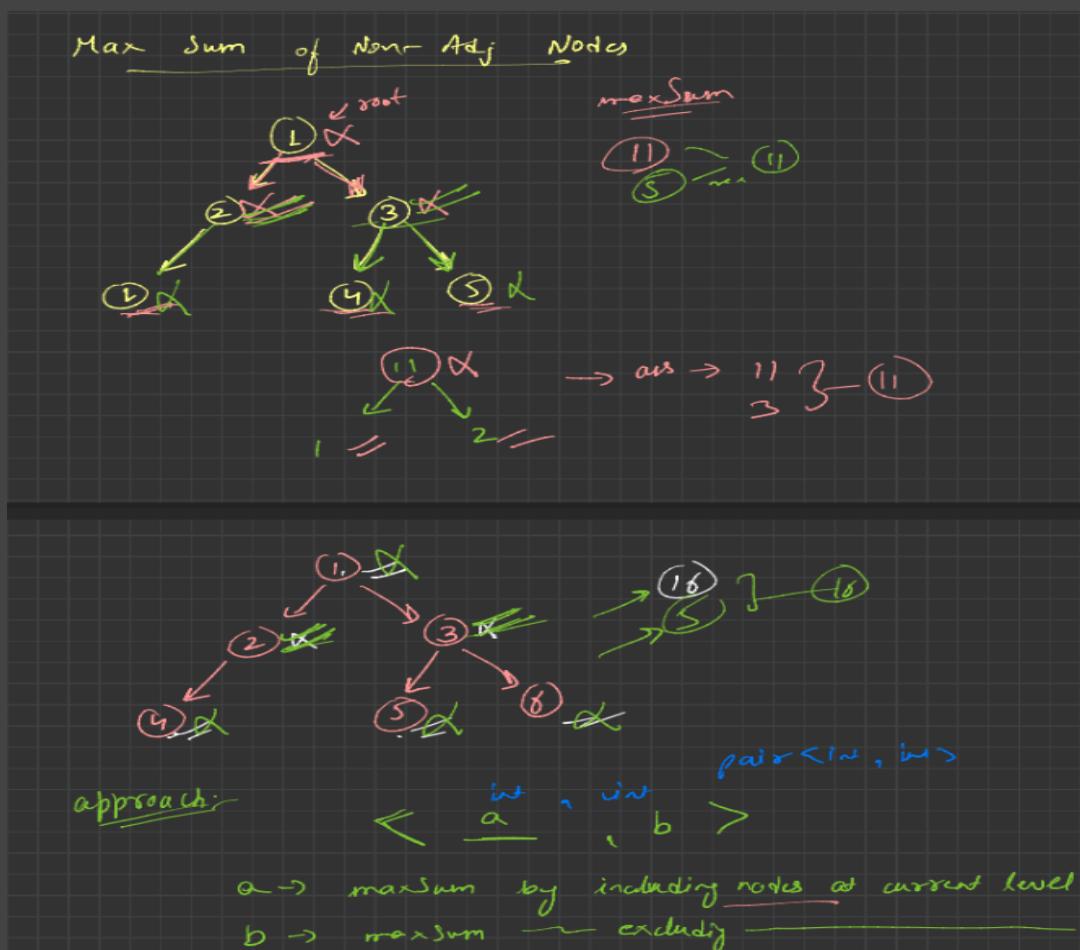


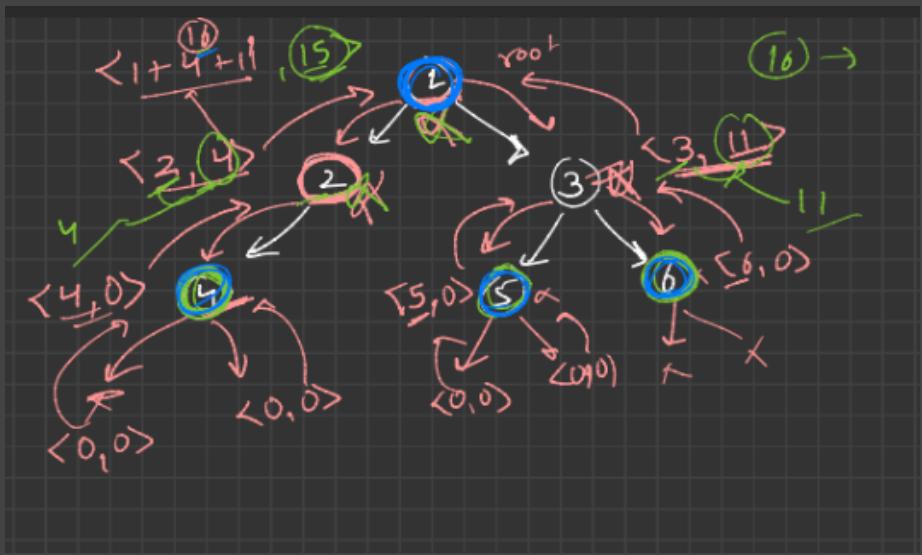
Kth Ancestor In A Tree:



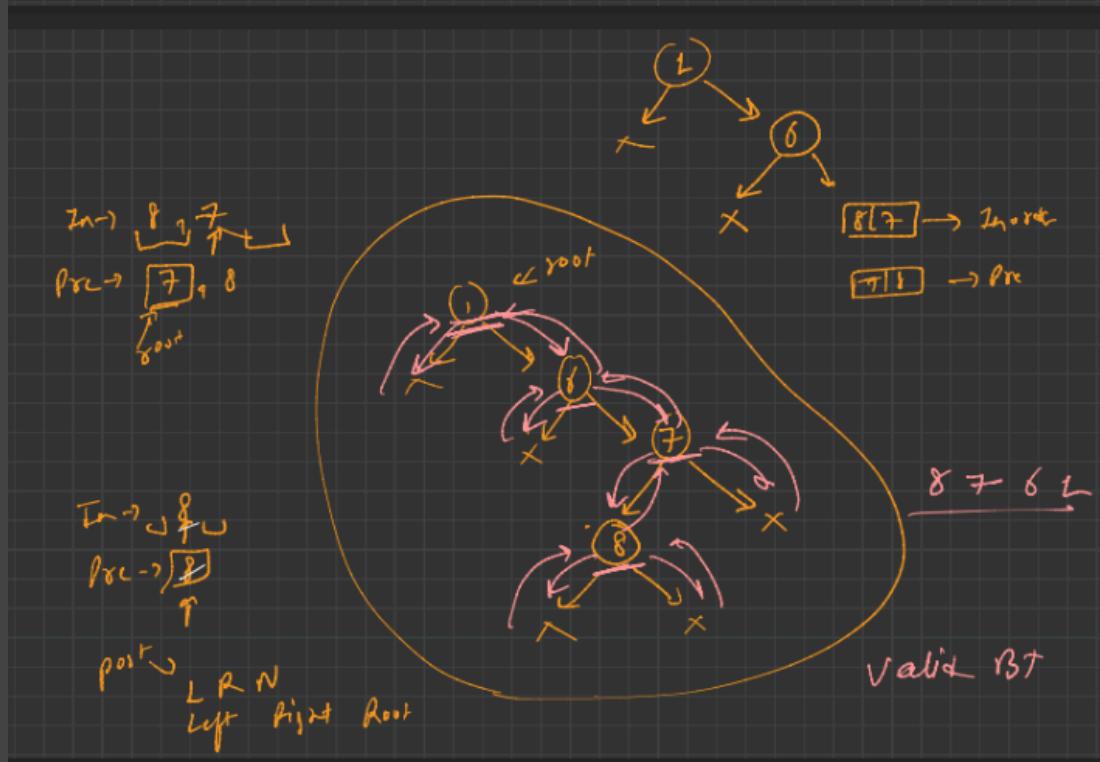
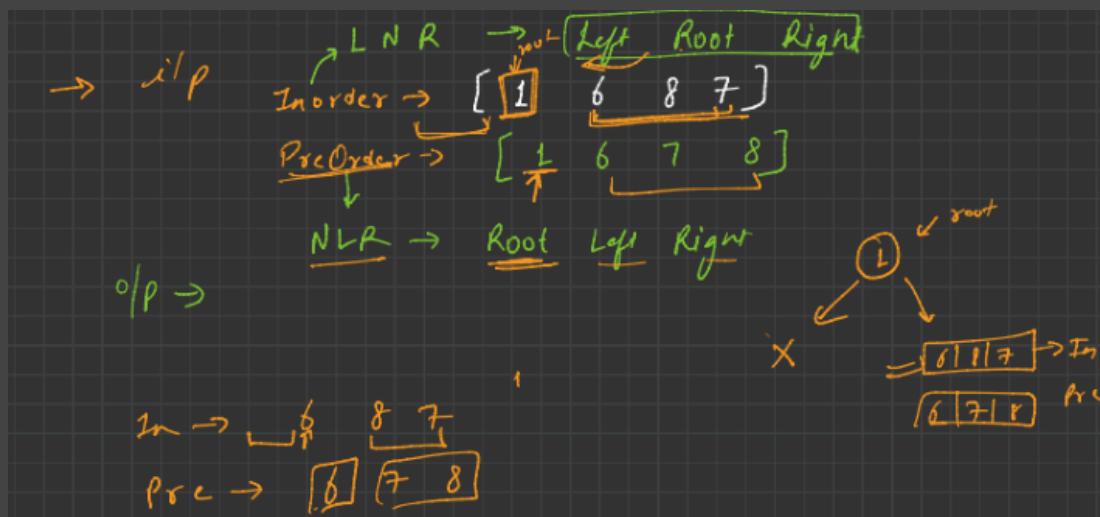


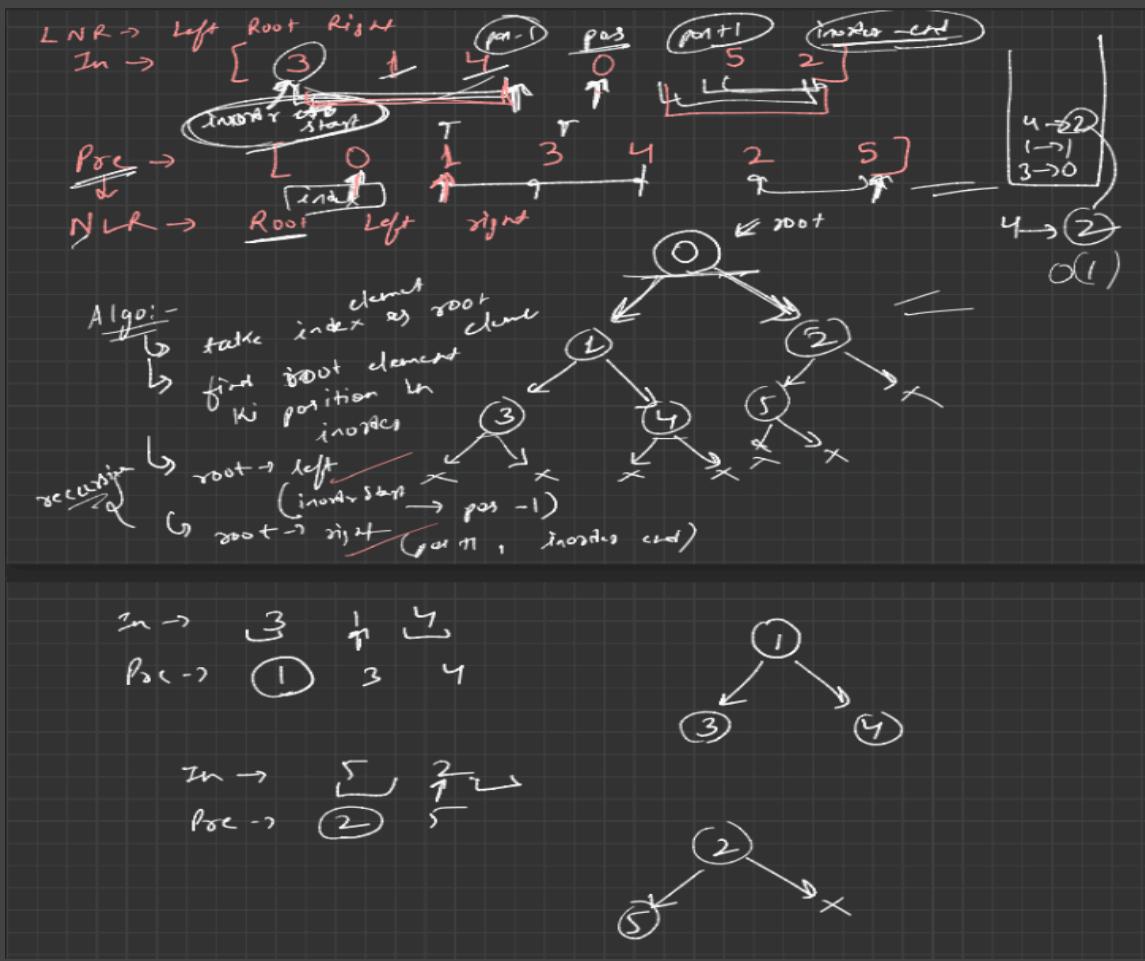
Maximum Sum Of Non-Adjacent Nodes:



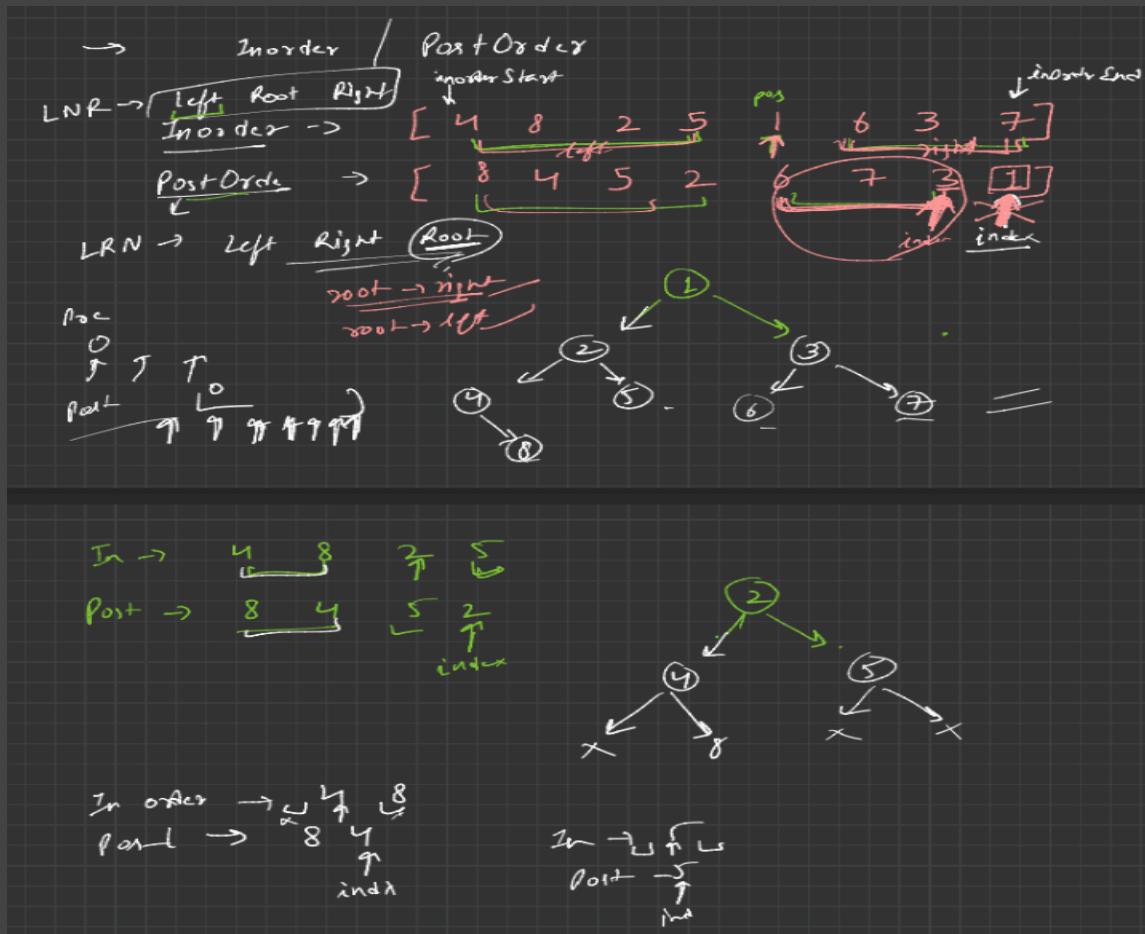


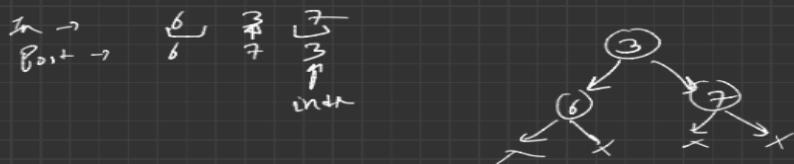
Construct Tree From InOrder And PreOrder:





Construct Tree From InOrder And PostOrder:





\rightarrow BT \rightarrow construct

$\rightarrow \underline{\text{in}} / \text{pre} \Rightarrow \equiv$

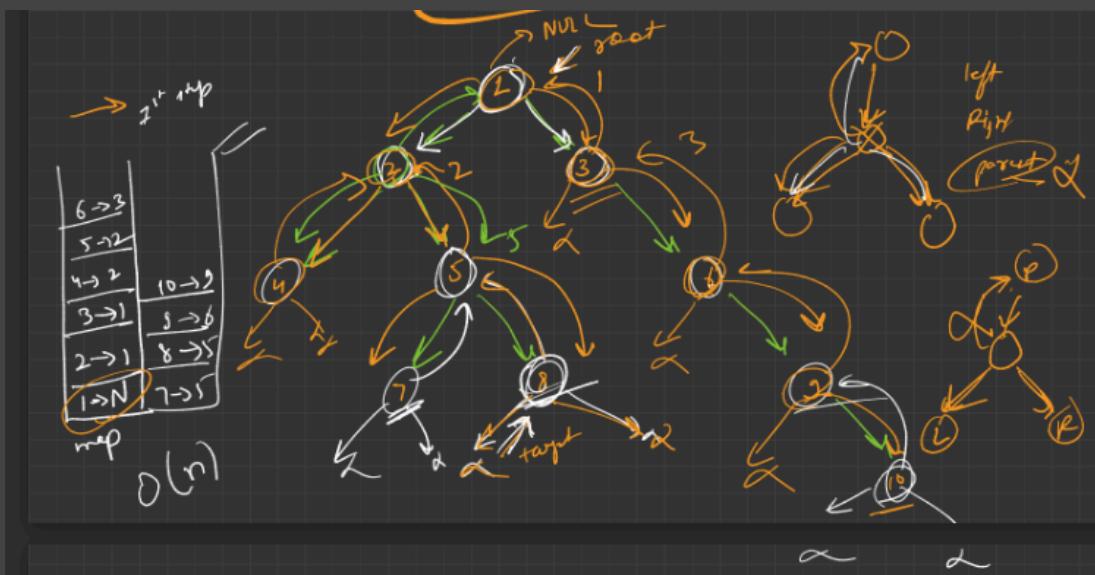
→ In / Part \Rightarrow

$$T - C \quad \underline{\underline{S - C}}$$

$$\mathcal{O}(\underline{n \log n})$$

$$O(n^+) \rightarrow \underline{O(n \log n)}$$

Minimum Tree To Burn The Entire Binary Tree:



approach:-

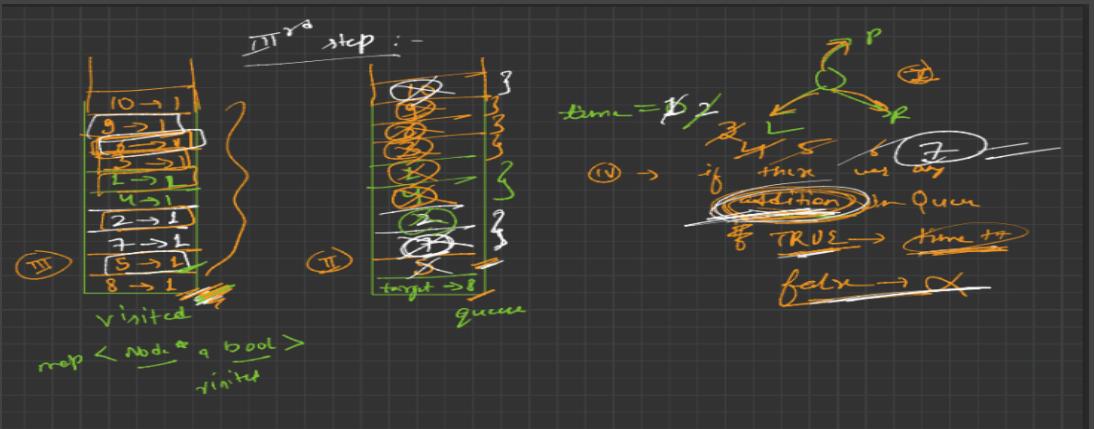
Ist step

node ID parent mapping

map < Node*, Node* > nodeToParent
napping

\Rightarrow L.O.T

$\overline{II^r}$ step \rightarrow find target Node
 $\hookrightarrow O(H)$



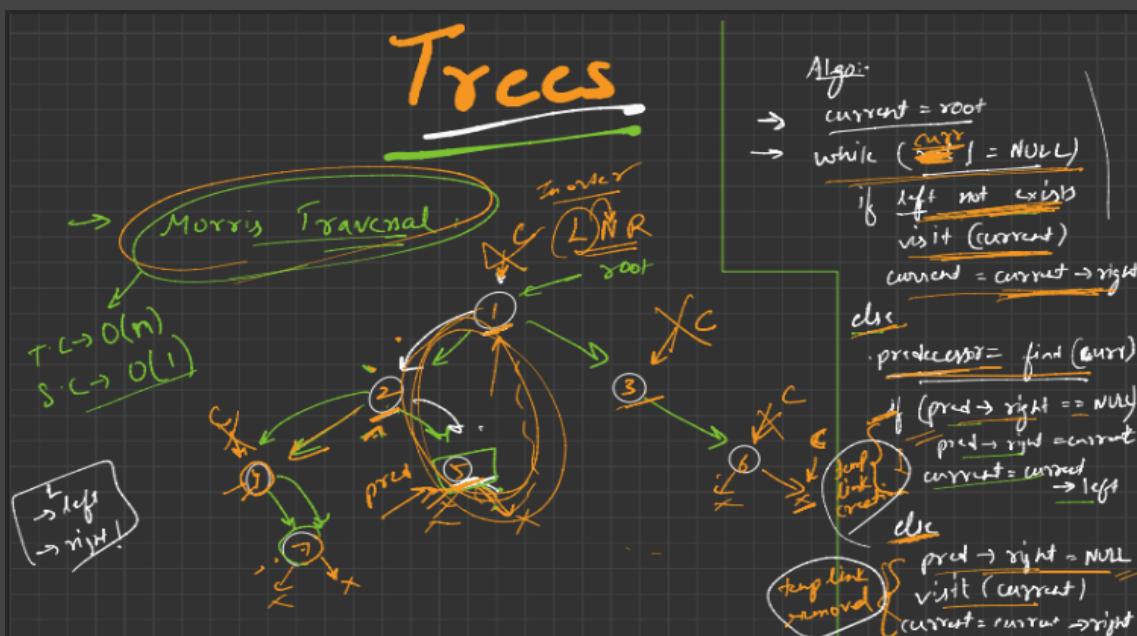
$$T \cdot \subset \Rightarrow \underline{\underline{O(N)}} + \underline{\underline{O(n)}}$$

= 

$\underline{\underline{O(N^{\log n})}}$

S.C $\rightarrow O(n) + O(N)$

Morris Traversal:



1 mark 4 7 2 5 1 3 6

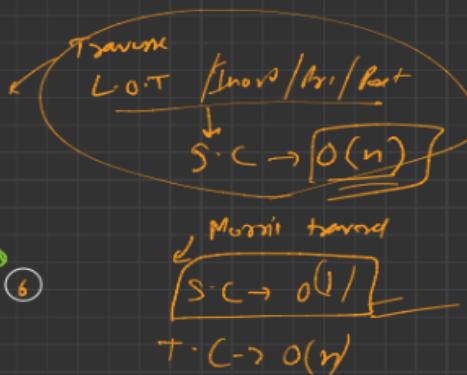
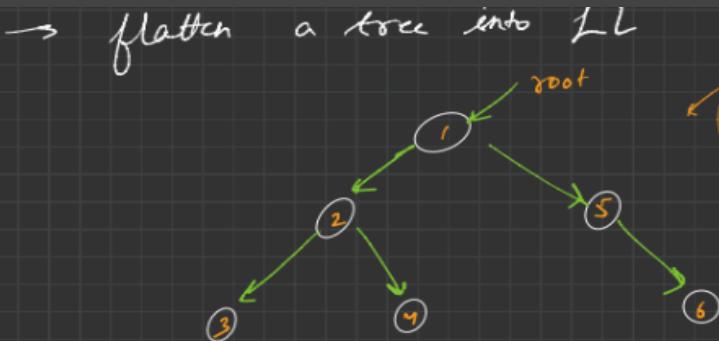
$\rho_{red} = \text{curr} \rightarrow \text{left}$

$$I \rightarrow O(n)$$

$\mathcal{O}(1)$

while $\text{pred} \rightarrow \text{right}^! = \text{NULL}$ $\text{pred} \rightarrow \text{right}^! = \text{correct}$

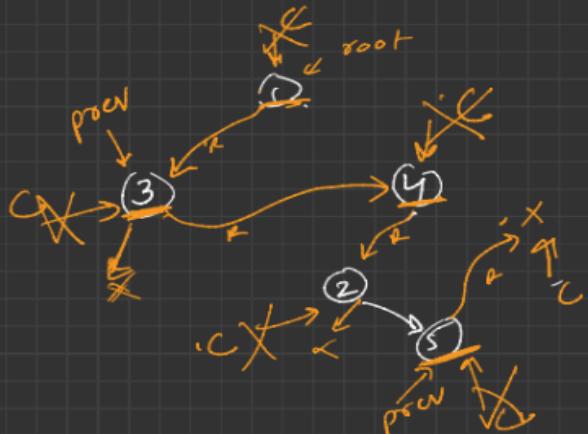
Flatten A Binary Tree To Linked List:



1 → traverse → each goeth node create
 Kriluge \times → in-place

2 → recursion → $O(n)$
 $O(n)$ $\rightarrow \times$

3 → Morris traversal



Alg - \rightarrow current = root

while (curr != NULL)

i

if curr left exists

i

predecessor {
 while (prev > right)
 prev = right;

prev > right = curr > right

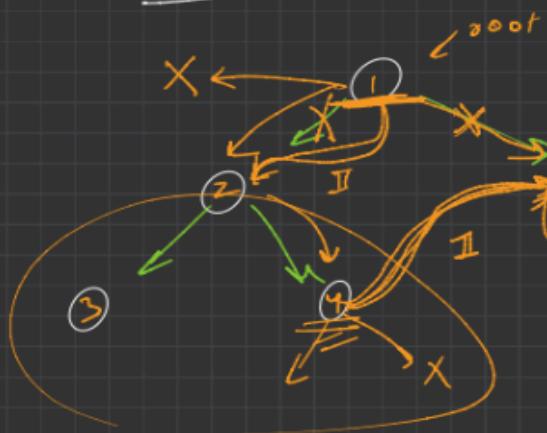
curr > right = curr > left

}

curr = curr > right;

}

Morris traversal



PerOrder \rightarrow N $\xrightarrow{\text{Root}}$ L $\xrightarrow{\text{R}}$

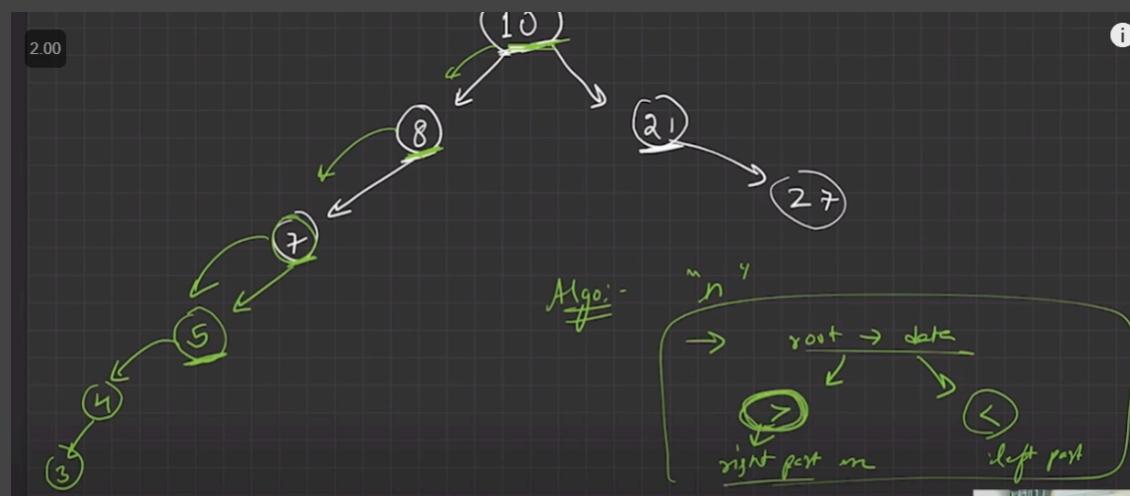
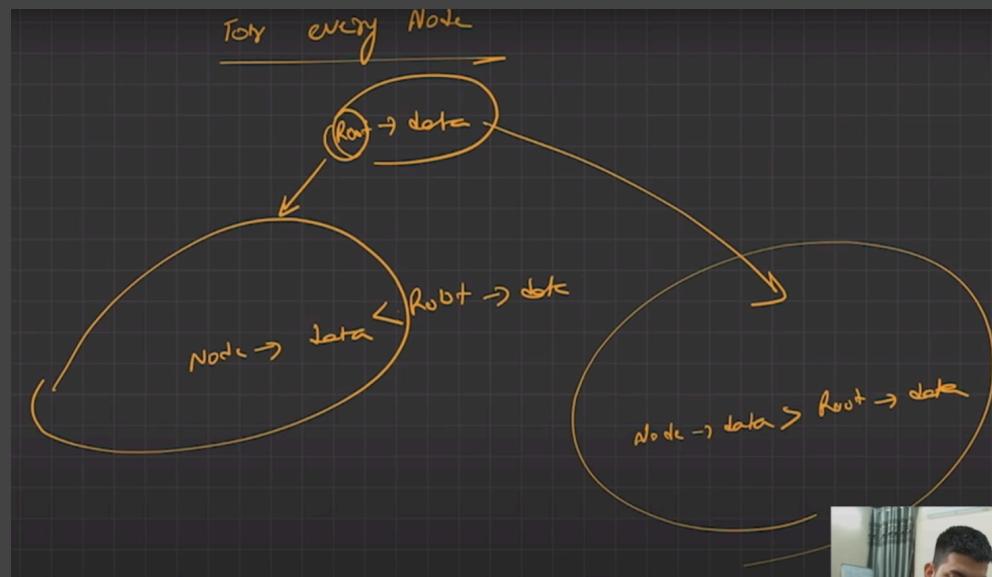
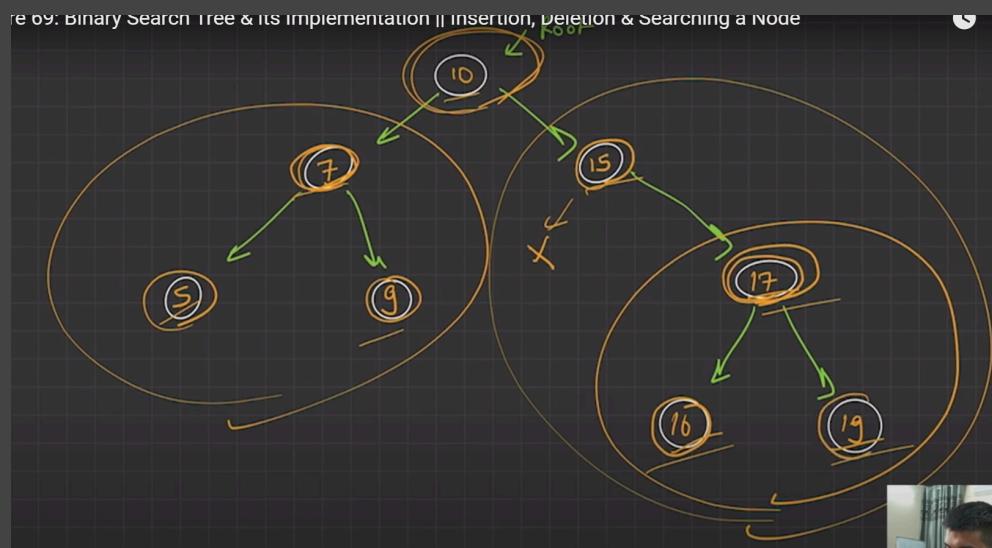
inorder predecessor
 \rightarrow right = curr > right

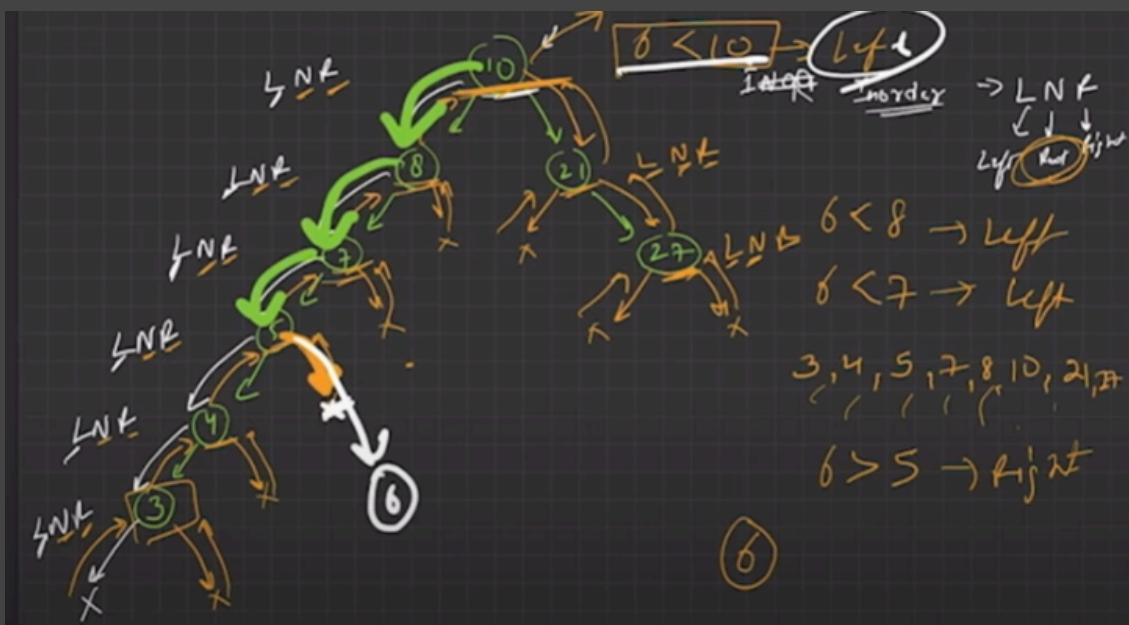
L \rightarrow R
 N \rightarrow L

N \rightarrow L
 L \rightarrow R

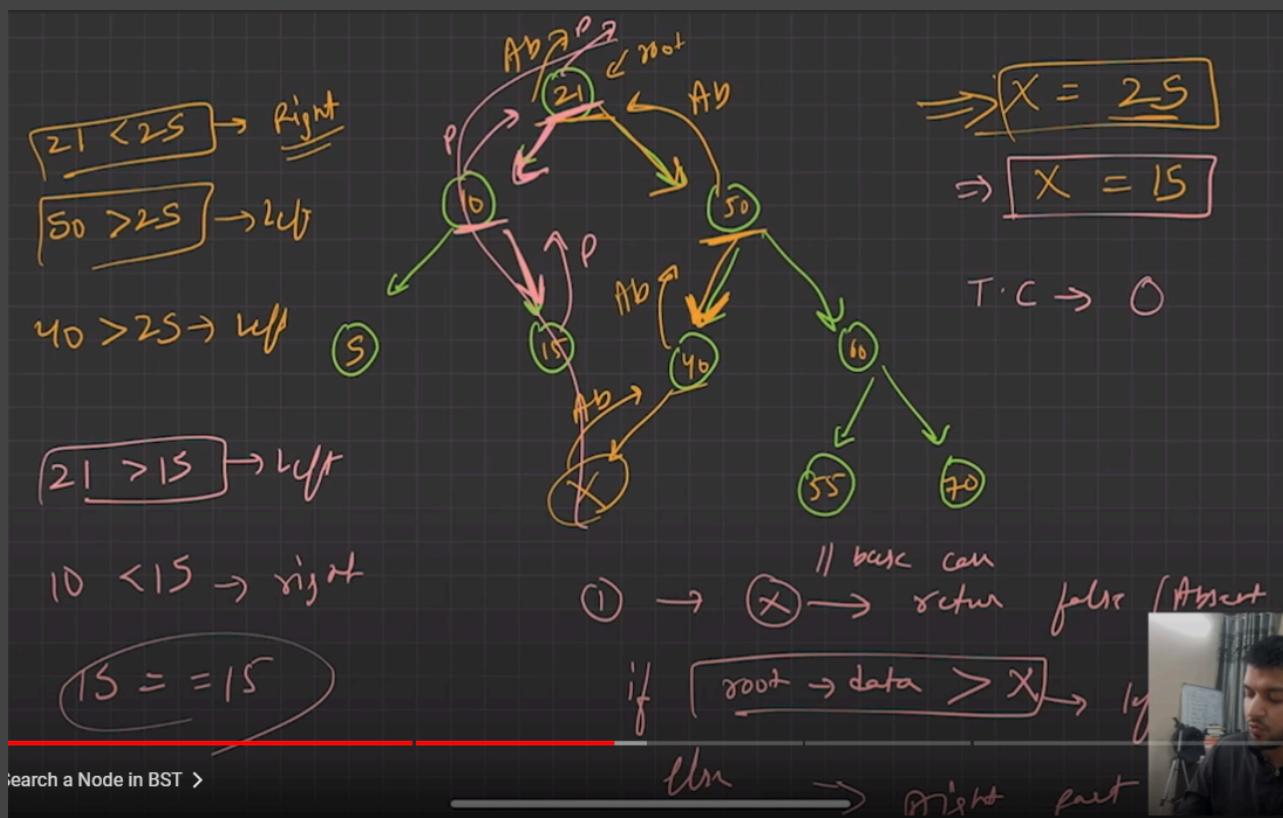
Binary Search Tree(BST)

Insert A Node In BST:



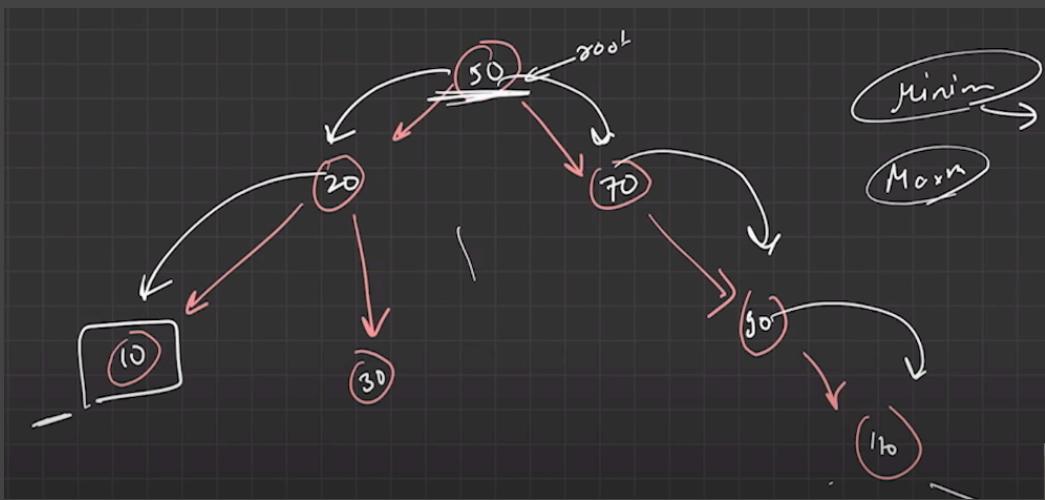


Search A Node In BST:



Minimum/Maximum Value In BST:

\Rightarrow Inorder of BST is sorted
(LNR)

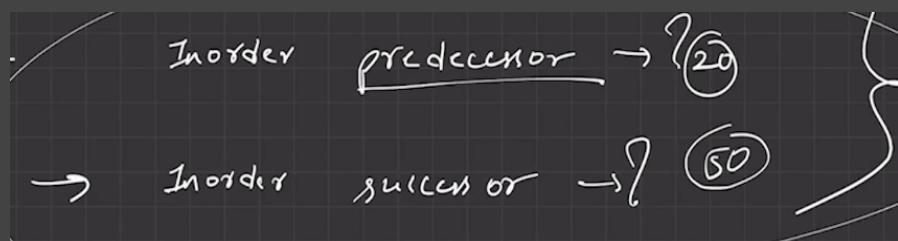


temp = root
 while (~~temp~~ → left != NULL)
 {
 temp = temp → left
 }
 return temp

temp = root
 while (~~temp~~ → right != NULL)
 {
 temp = temp → right
 }
 return temp

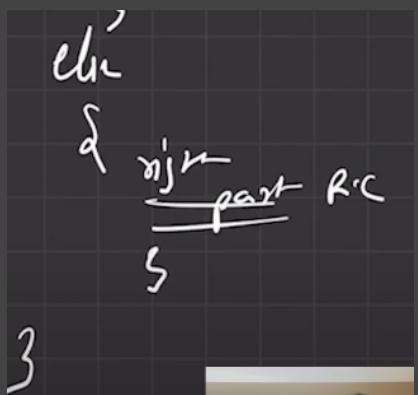
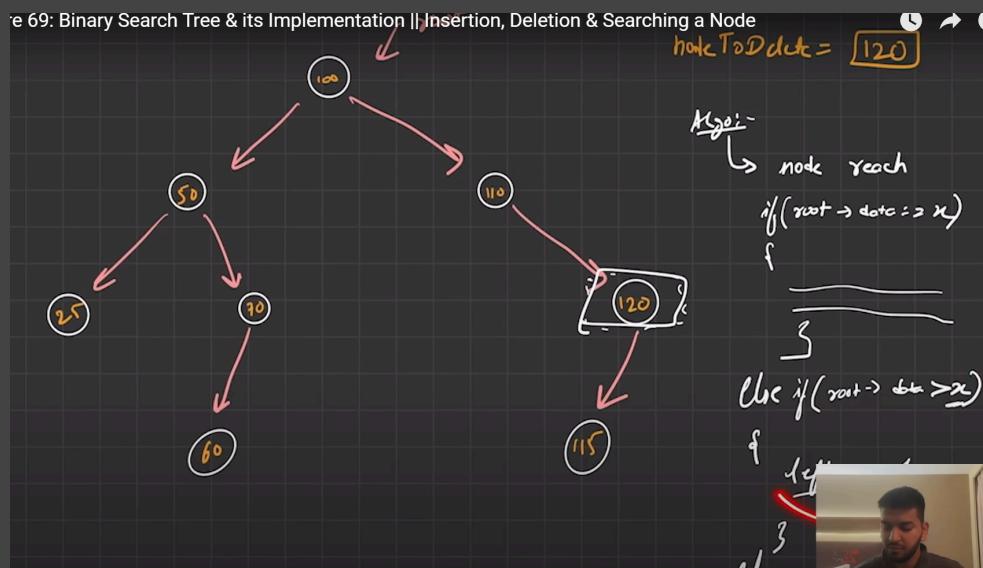
Inorder Predecessor In BST:

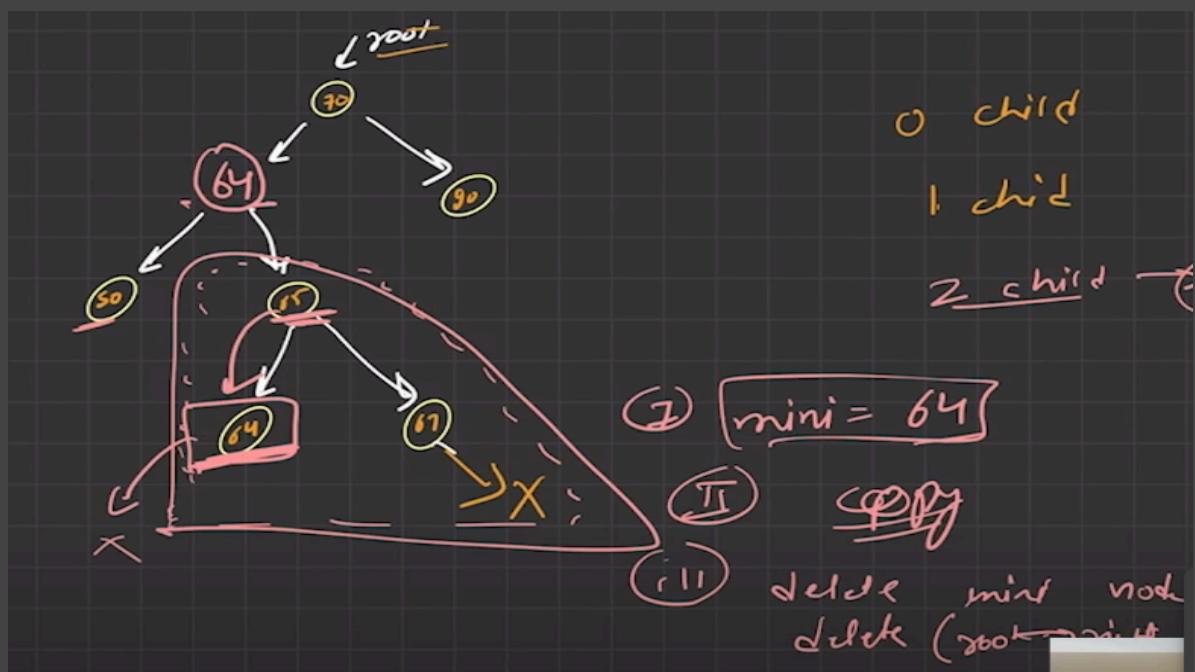
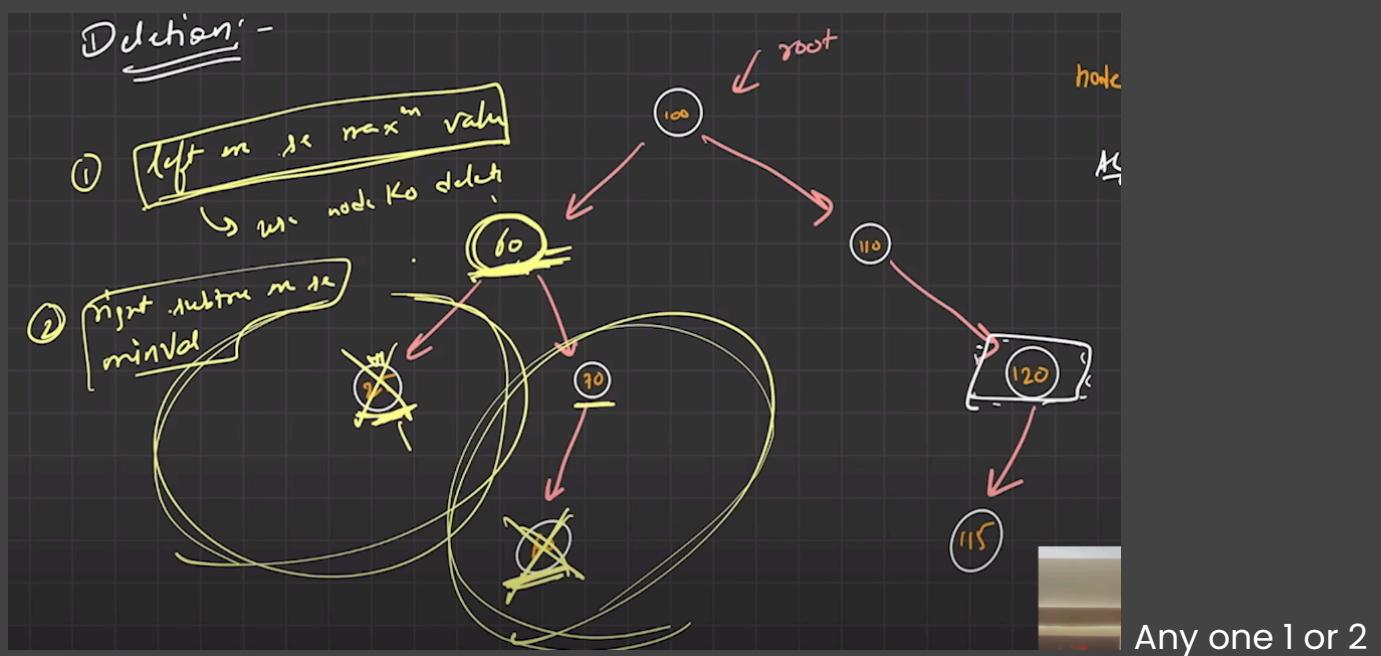
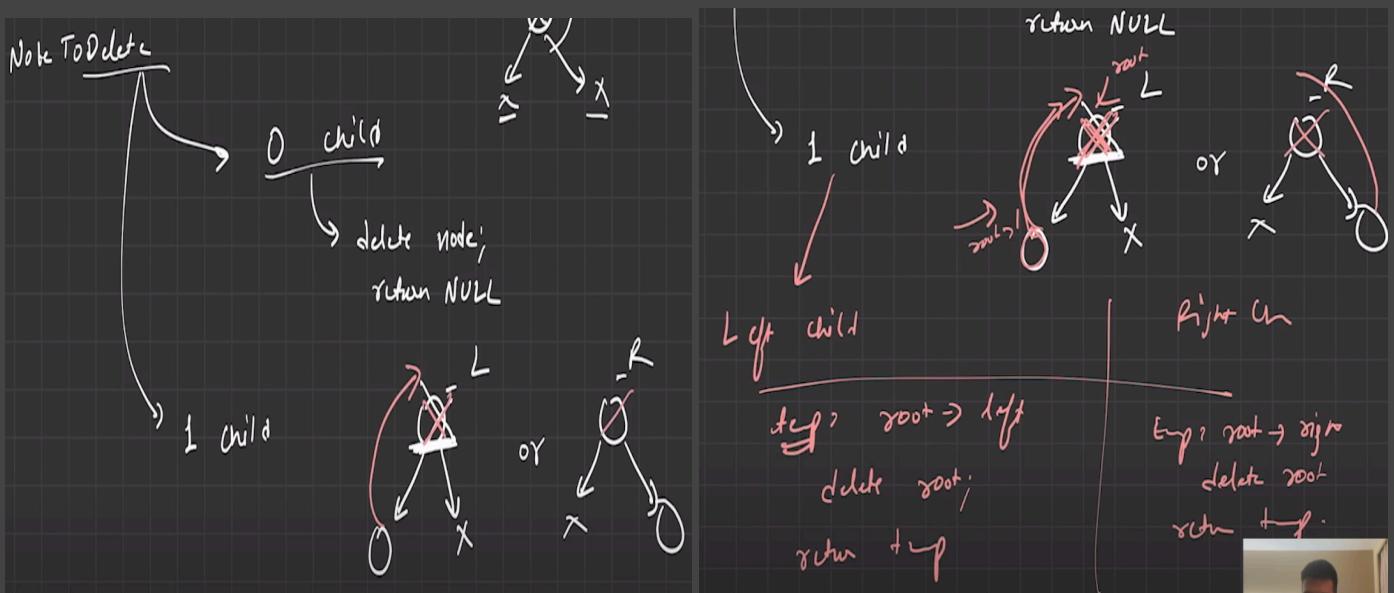
Inorder Successor In BST:



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

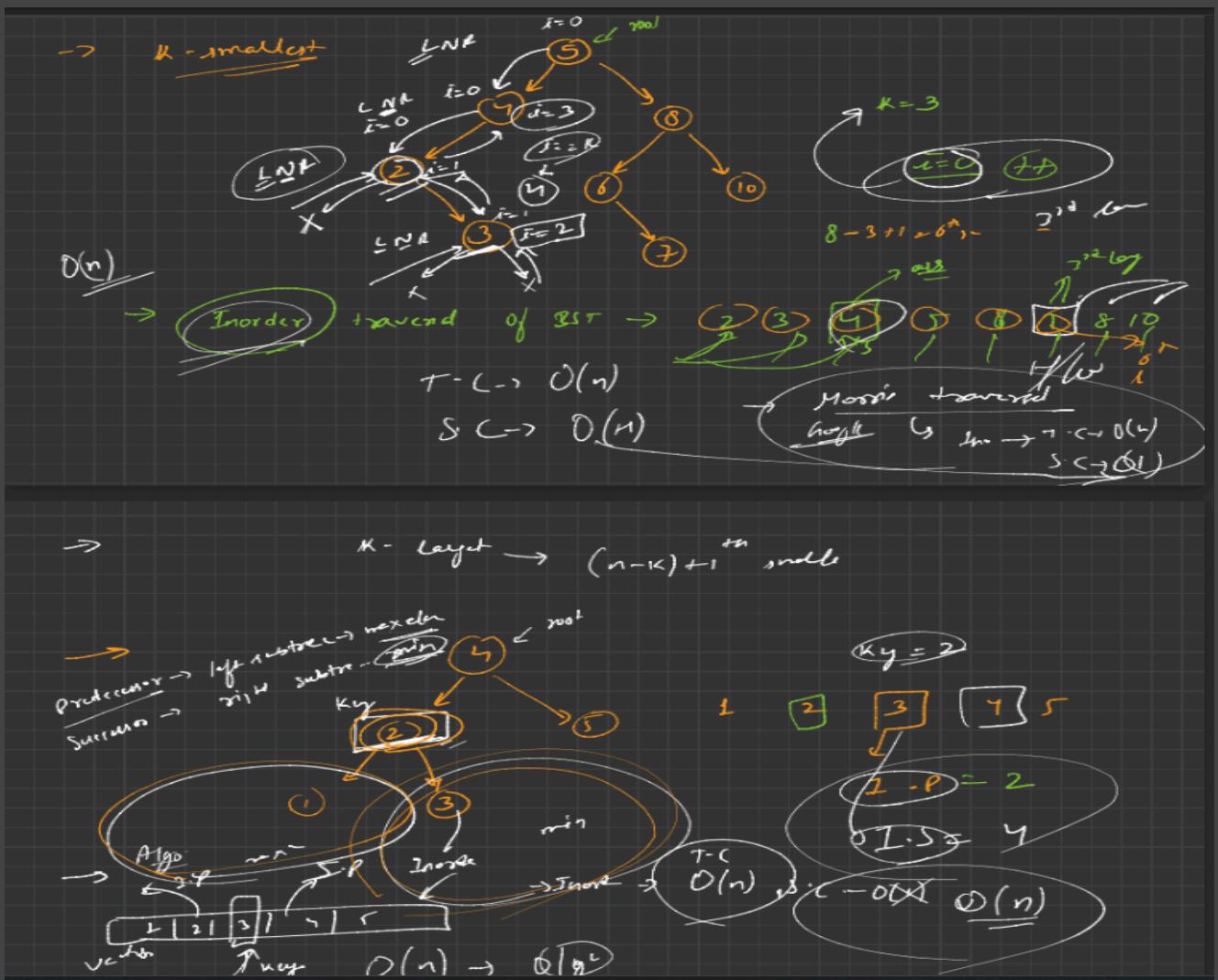
Delete A Node In BST:

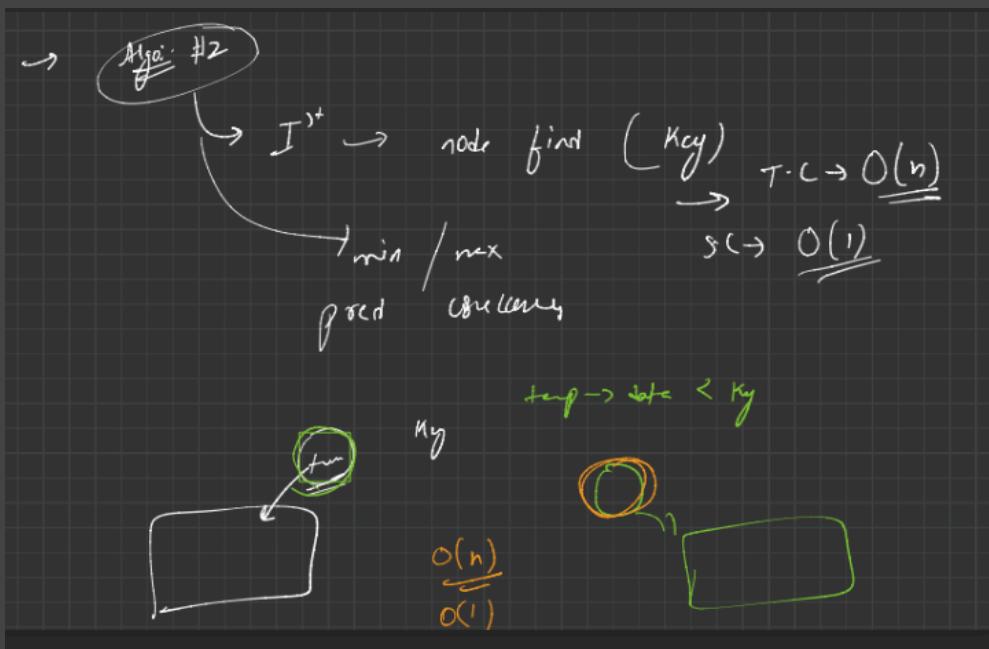




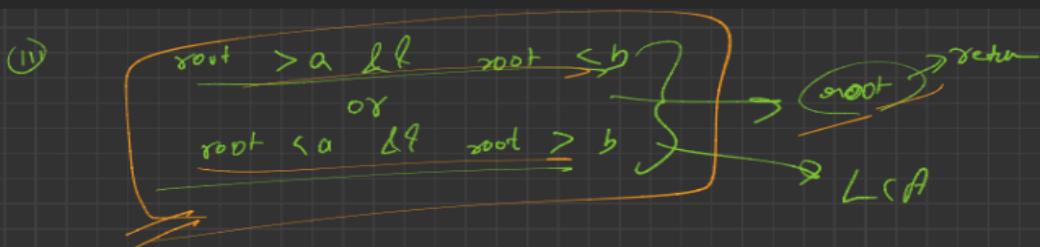
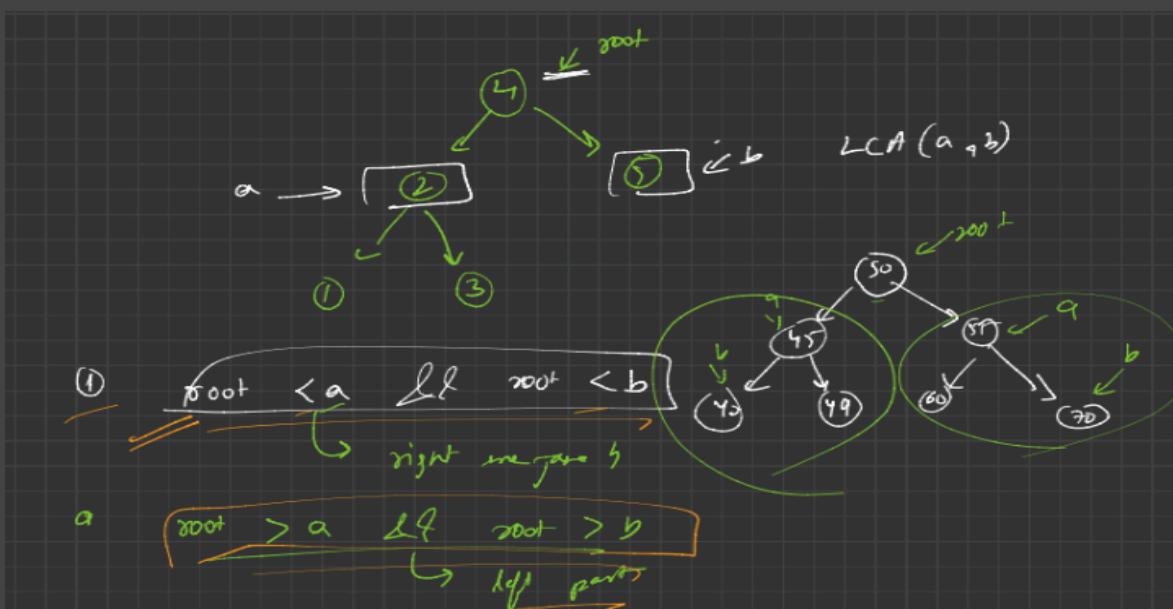
Validate BST:

Kth Smallest/Largest Element In BST:



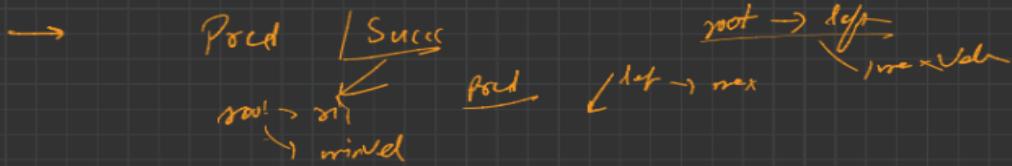


Lowest Common Ancestor In BST:



$\rightarrow K^m$ smaller / K^n longer

\rightarrow invalid BST



Two Sum In BST:

Brute force

search in left/Right

$$10, \quad (\text{target} - 10) = 10 \rightarrow O(N^2)$$

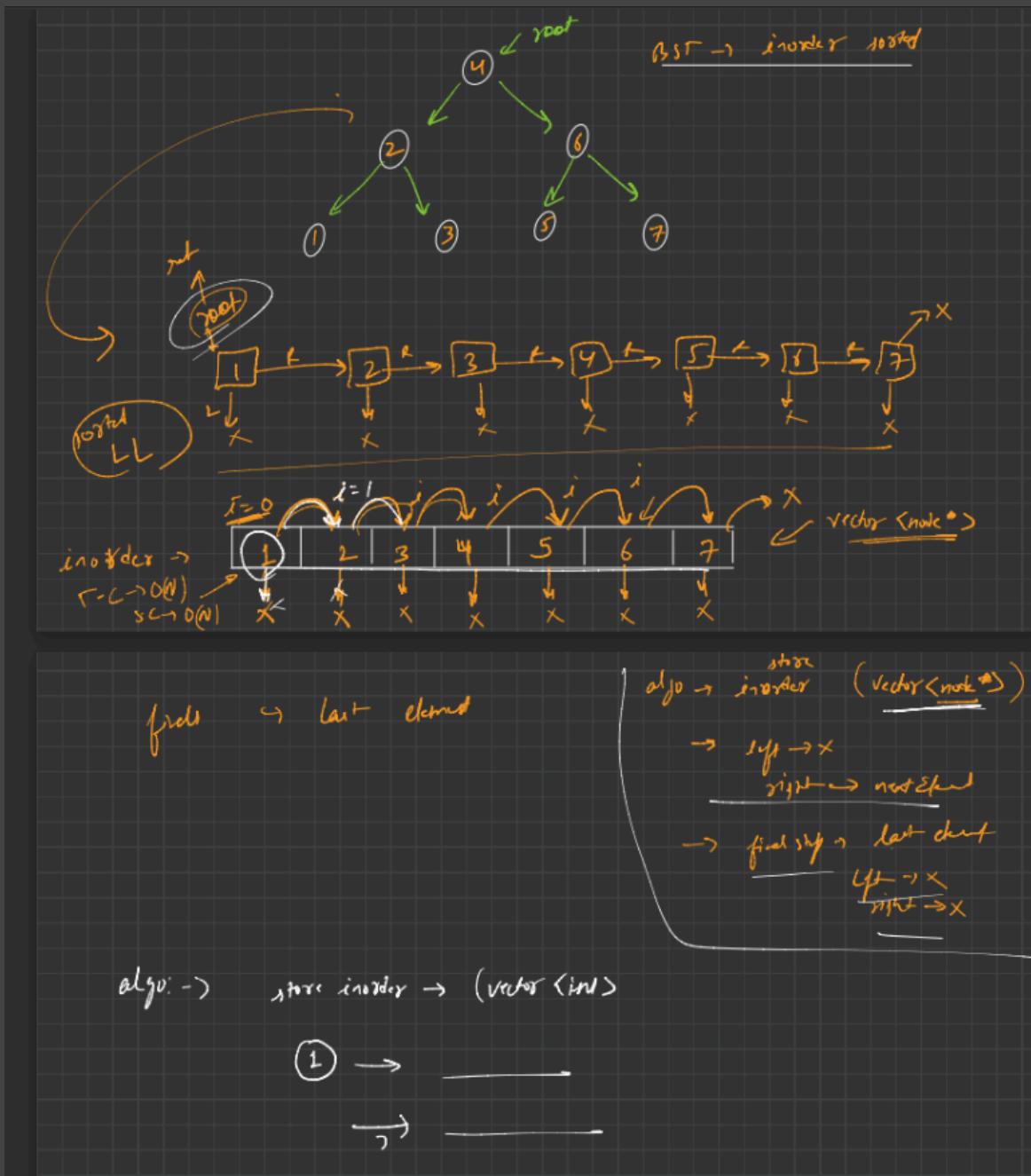
$$6, \quad (\text{target} - 6) \xrightarrow{(4)} \text{left/Right}$$

1000
1
n-1
n-2
n-3
1
1
 $\rightarrow n^2$

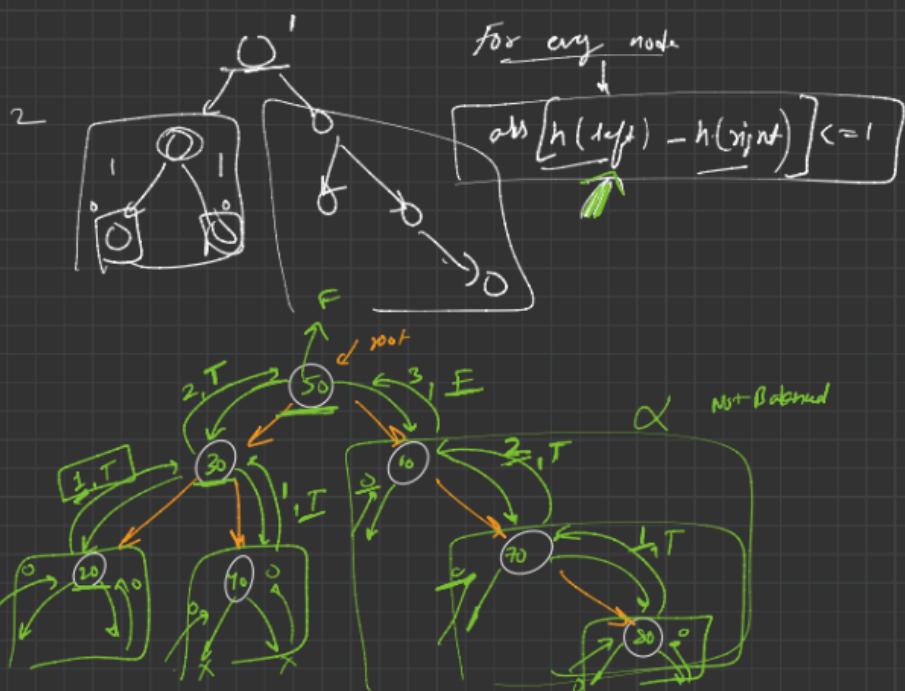
$\Rightarrow O(N) \rightarrow$ Single traversal

$BST \rightarrow$ inorder sorted

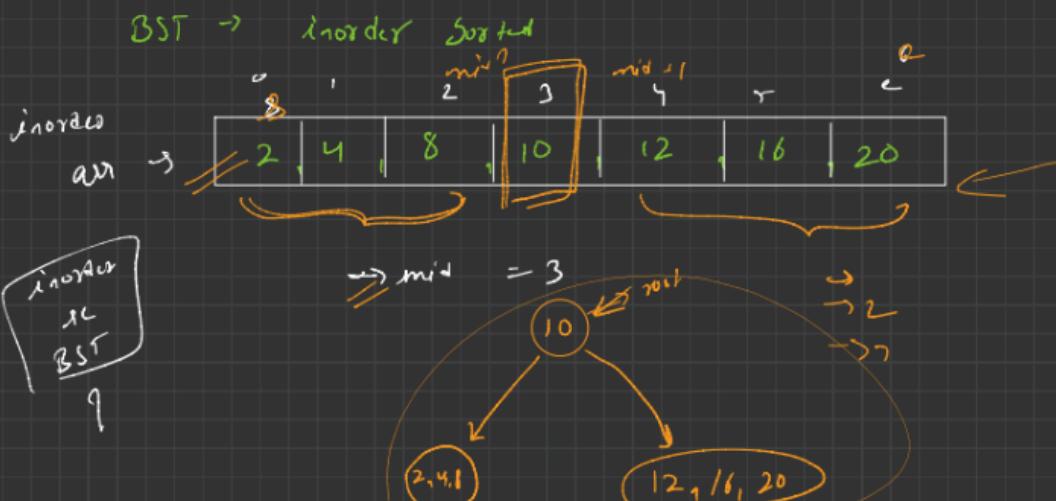
Flatten BST To A Sorted List:



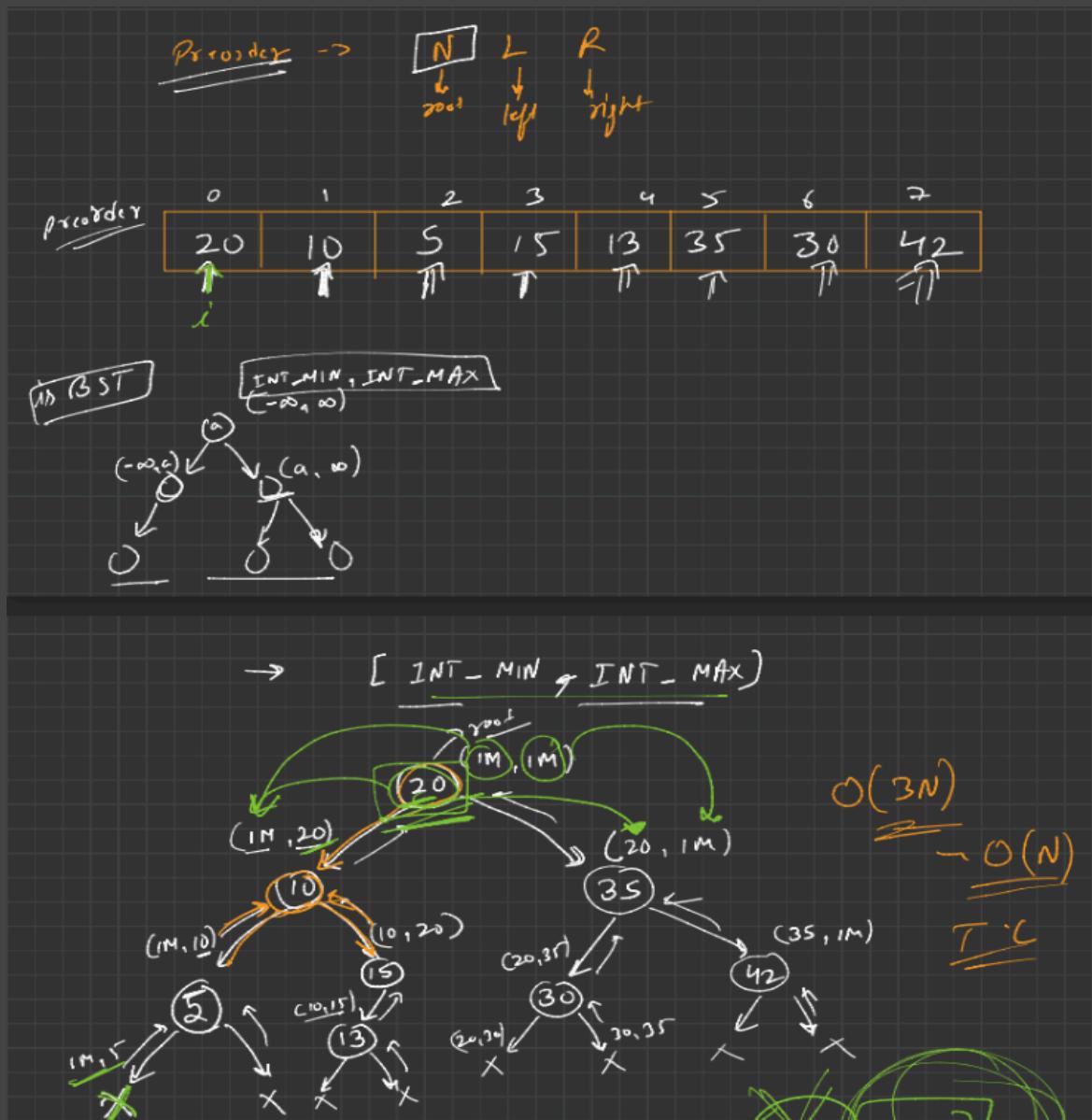
Normal BST To Balanced BST:



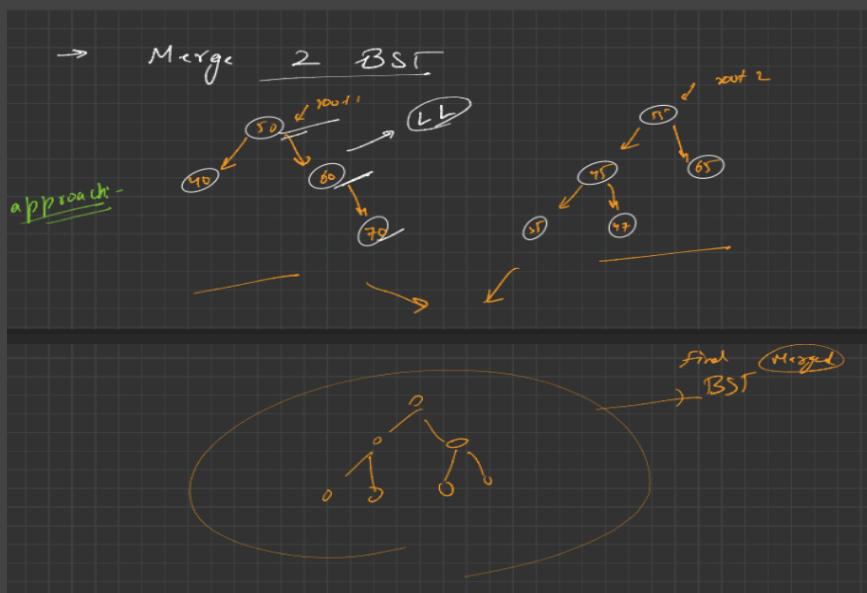
$\rightarrow i/p \rightarrow \text{Normal BST}$



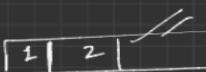
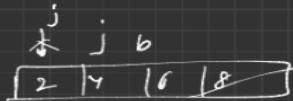
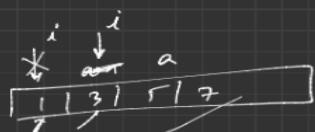
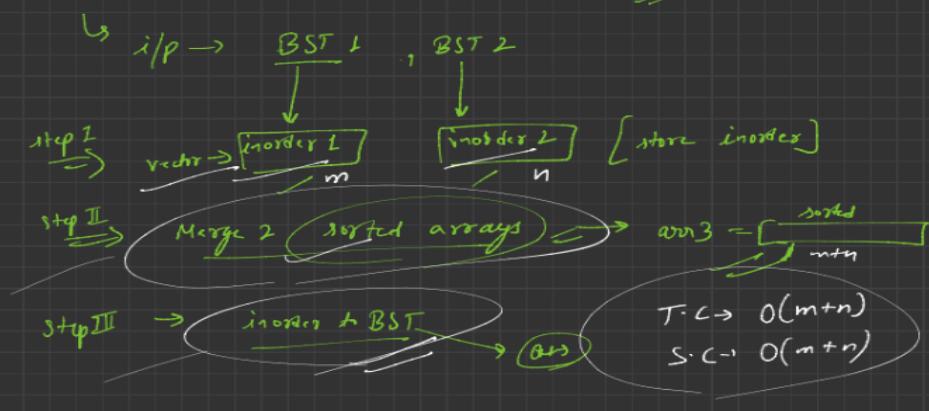
BST From Preorder Traversal:



Merge Two BST:



Approach #1

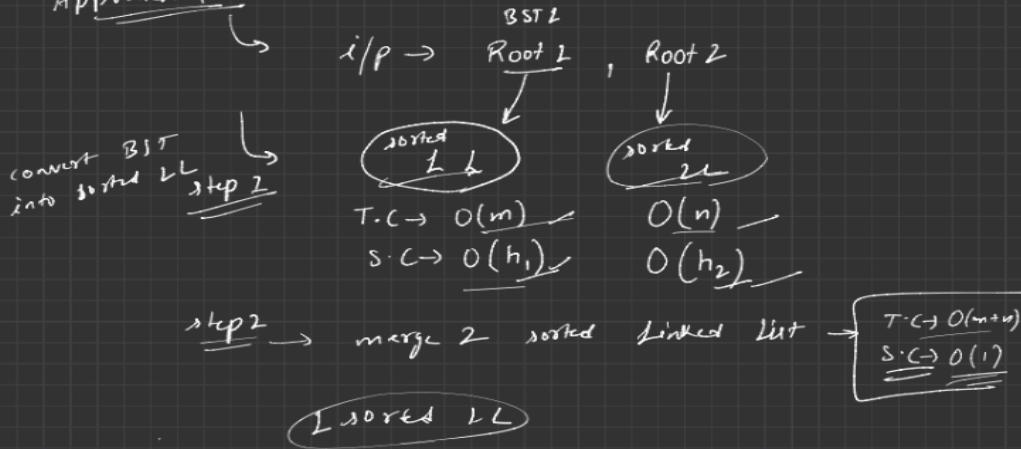


and $(a+b)$

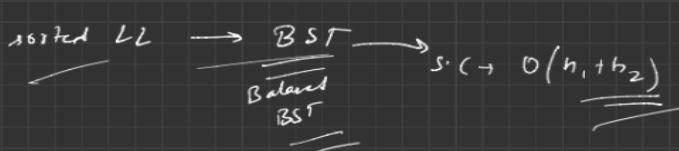
$\Rightarrow T.C \rightarrow O(m+n)$

$S.C \rightarrow O(h_1 + h_2)$

Approach #2

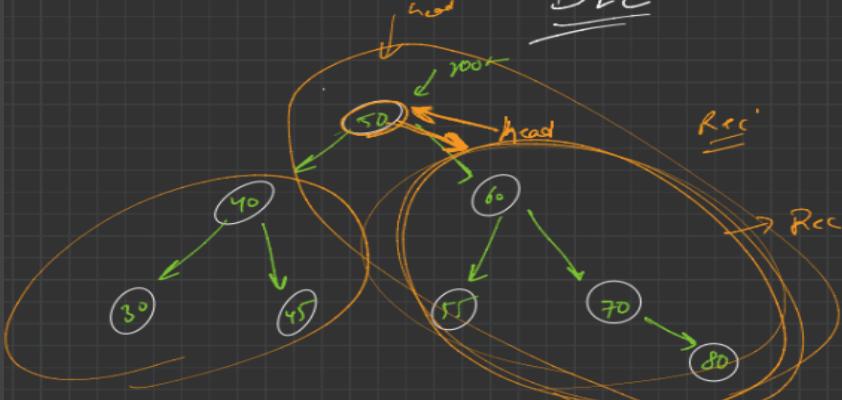


Step 3



- ① (convert a BST into sorted LL) $\rightarrow O(n)$
 $O(h)$
- ② Merge 2 sorted linked lists
- ③ sorted LL \rightarrow BST

→ convert a BST into DLL $\underline{\underline{O(H)}}$

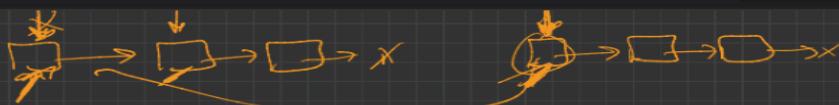
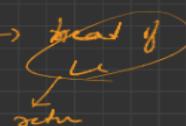


clgo → convert right subtree into LL → head of LL

→ root → right = head
head → left = root

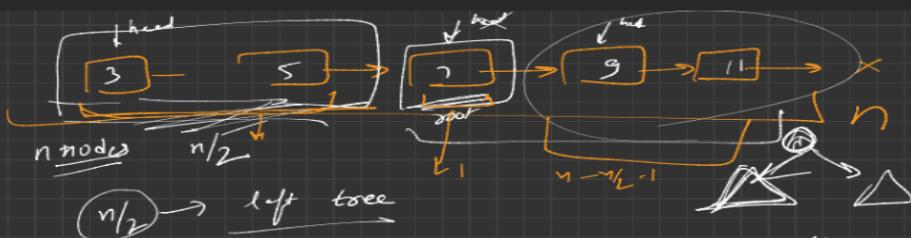
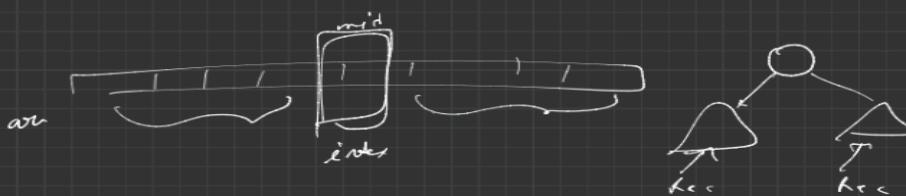
→ head = root

→ convert left part into LL → head of LL



head = NUL
tail = NUL

head



$n/2 \rightarrow$ left tree

① $n/2 \rightarrow$ left tree \Rightarrow *left

② create root
root \rightarrow left = left

③ head \rightarrow next

④ right subtree \rightarrow tail
 $root \rightarrow right = Rec(right, n - n/2 - 1);$

