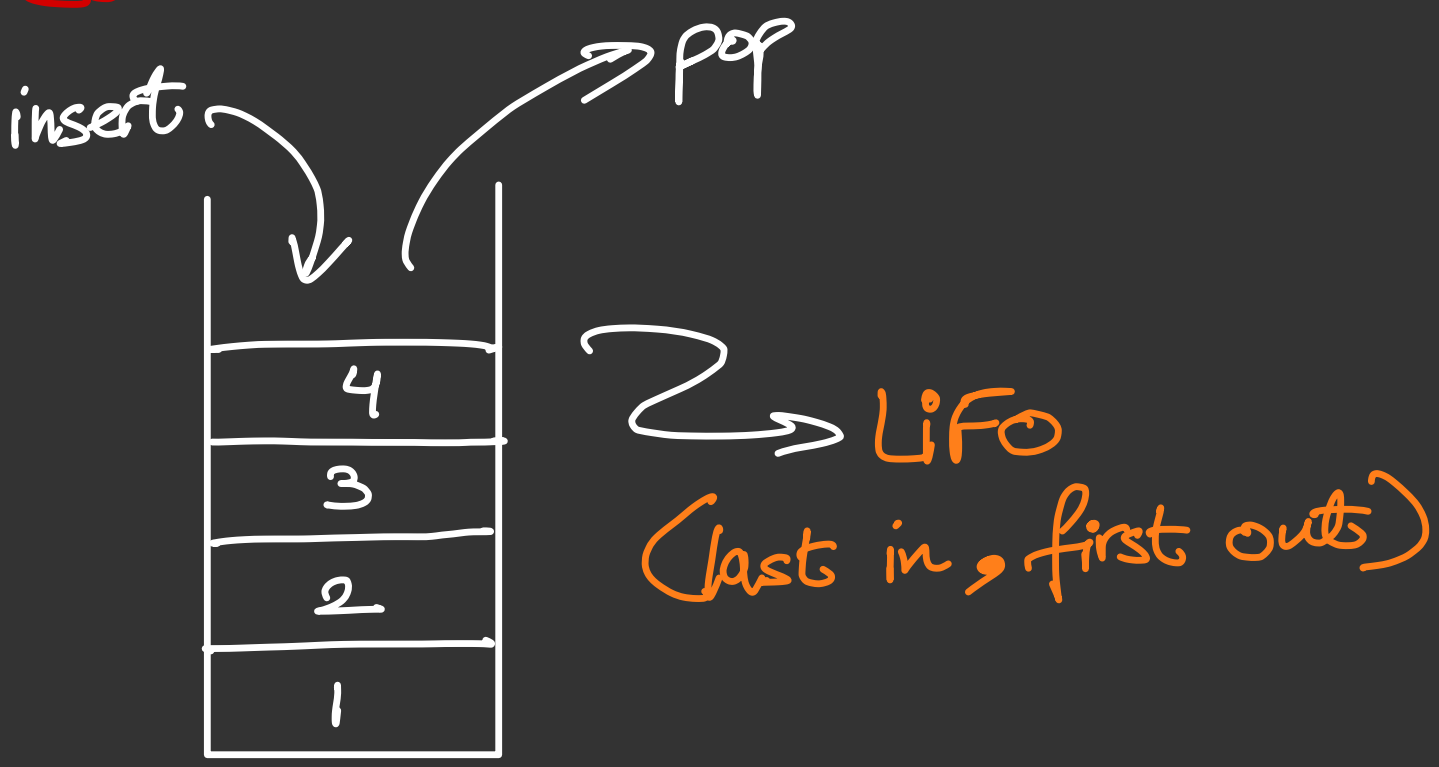
Stacks

Lecture 54



Operations

- push / insert
- pop / delete
- peak (top element)

→ stacks.cpp

Creation using STL

```
stack <int> s;
s.push(2);
s.pop();
s.top();
```

→ returns top element

`s.empty()` → True / False

✱ So the last element you inserted, will be the first one to be deleted (or) popped.

`s.size();` → length of the stack (or) number of elements.

Stack Implementation without STL

- Can be done in two ways
  - Arrays
  - Linked List

Using Arrays

properties
- int top
- int *arr
- int size

push()
- check for space, else Stack Overflow
- if space available → top++; arr[top] = element

pop()

↳ check element is present_
if not → Empty Stack

⟶ top--;

⊗ for element to be present

↳ top >= 0

top()

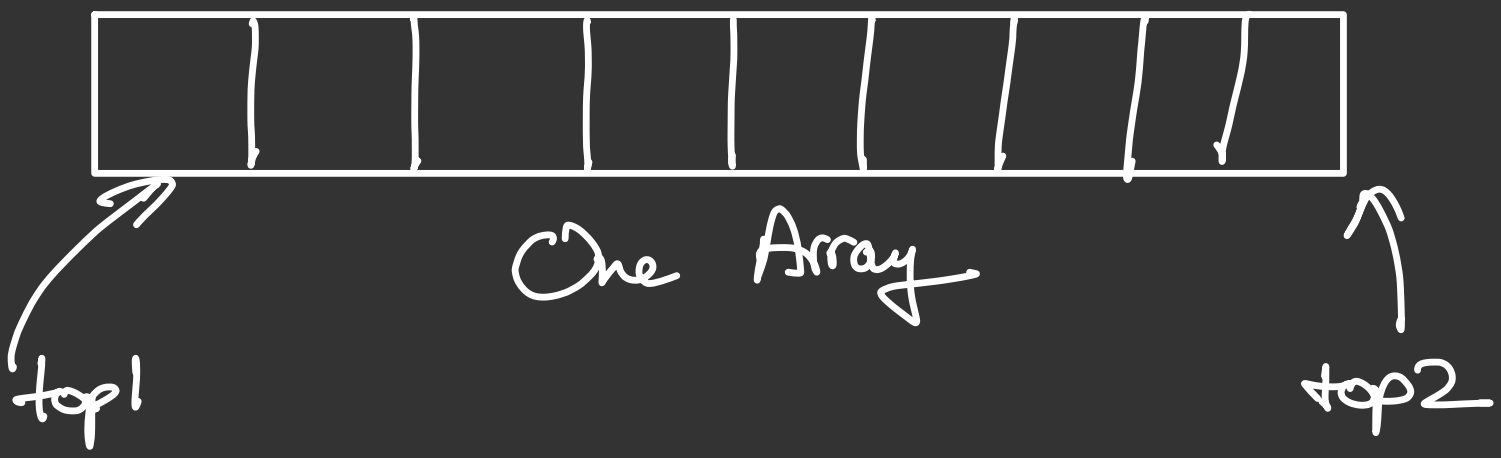
↳ arr[top]

⟶ stackImplementation_Arrays.cpp

Two Stacks using one array

Approach ⟶

```
int *arr;
int top1;
int top2;
int size;
```



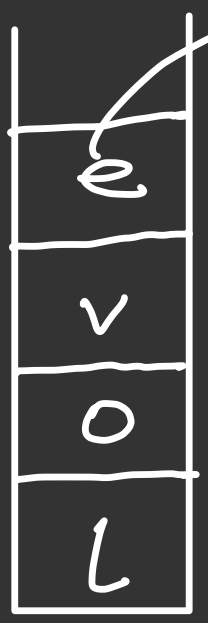
for stack 1 insertion : left to right
stack 2 insertion : right to left

Lecture 55

☆ By default whatever you insert in stack, you will get it back in REVERSE.

example:

love



ans = "evol"

→ reverseString_using_Stack.cpp

↳ SC = O(N)

Pop the middle element

→ for traversal in stack we need to pop top elements.

→ Thus, in order to get to the middle and presove all the top elements, we use RECURSION.

1. Store the top element
2. Pop it and then call recursive call
3. After the call, push the element again.

→ removeMiddleElement.cpp

Valid Paranthesis

Approach 1 ⟶

counting and checking

☆ But this approach does not say false for this

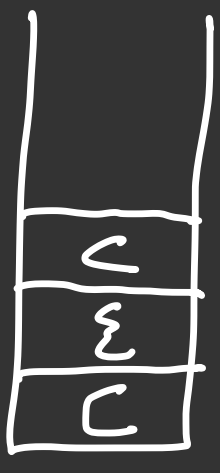
([)] ⟶ this is wrong

Approach 2 ⟶

Using Stack

example:

[{()}]



* Whenever a closing bracket comes check the current top with its opening and if true pop.

So, basically just keep on pushing opening brackets but as soon as a opening brackets comes check with current top.

——> validParenthesis.cpp

Insert element at the Bottom of the Stack

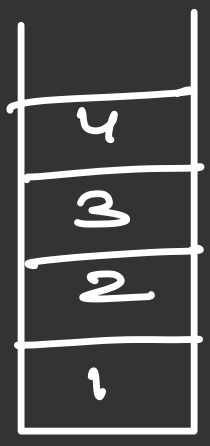
→ Again we have to traverse without deleting the elements, thus RECURSION

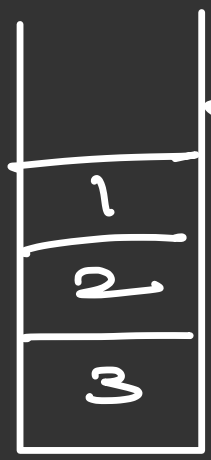
→ insertAtBottom_ofStack.cpp

Reverse Stack using Recursion

Key property of recursion

↳ you solve one case, rest recursion will do





→ this is already there

Now, we just have to insert [4] at the bottom.

→ Pick each element and place it at the bottom.

→ But this happens after the recursive call, because we want the element to be added in the bottom from current stack's bottom to up.

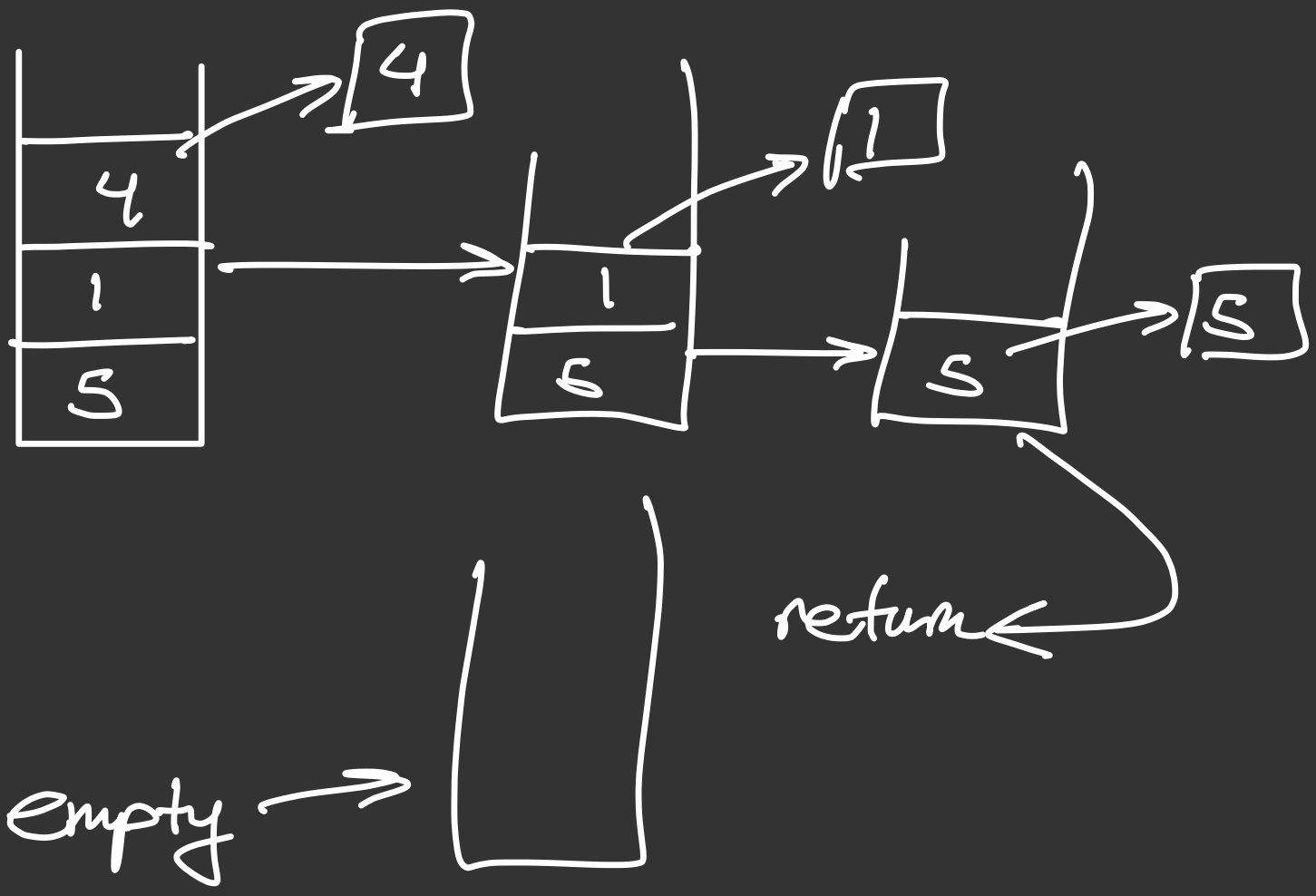
→ reverse_Stack_using Recursion.cpp

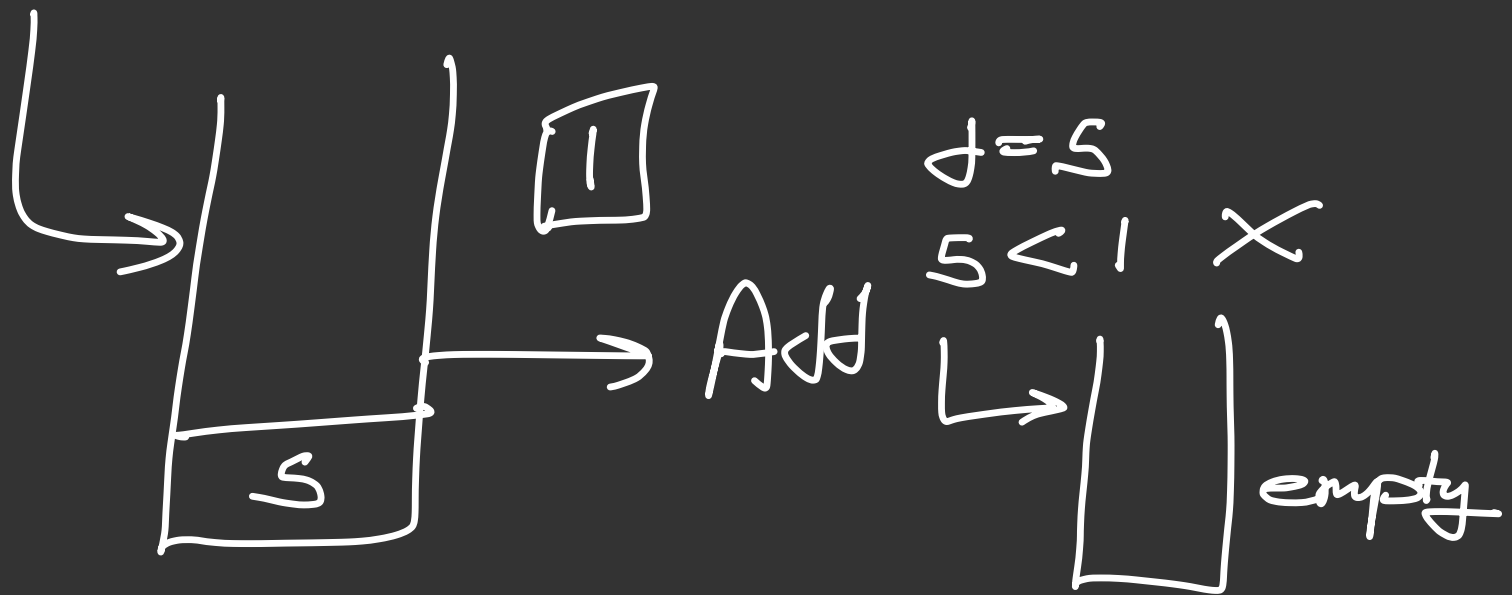
TC = $O(n^2)$

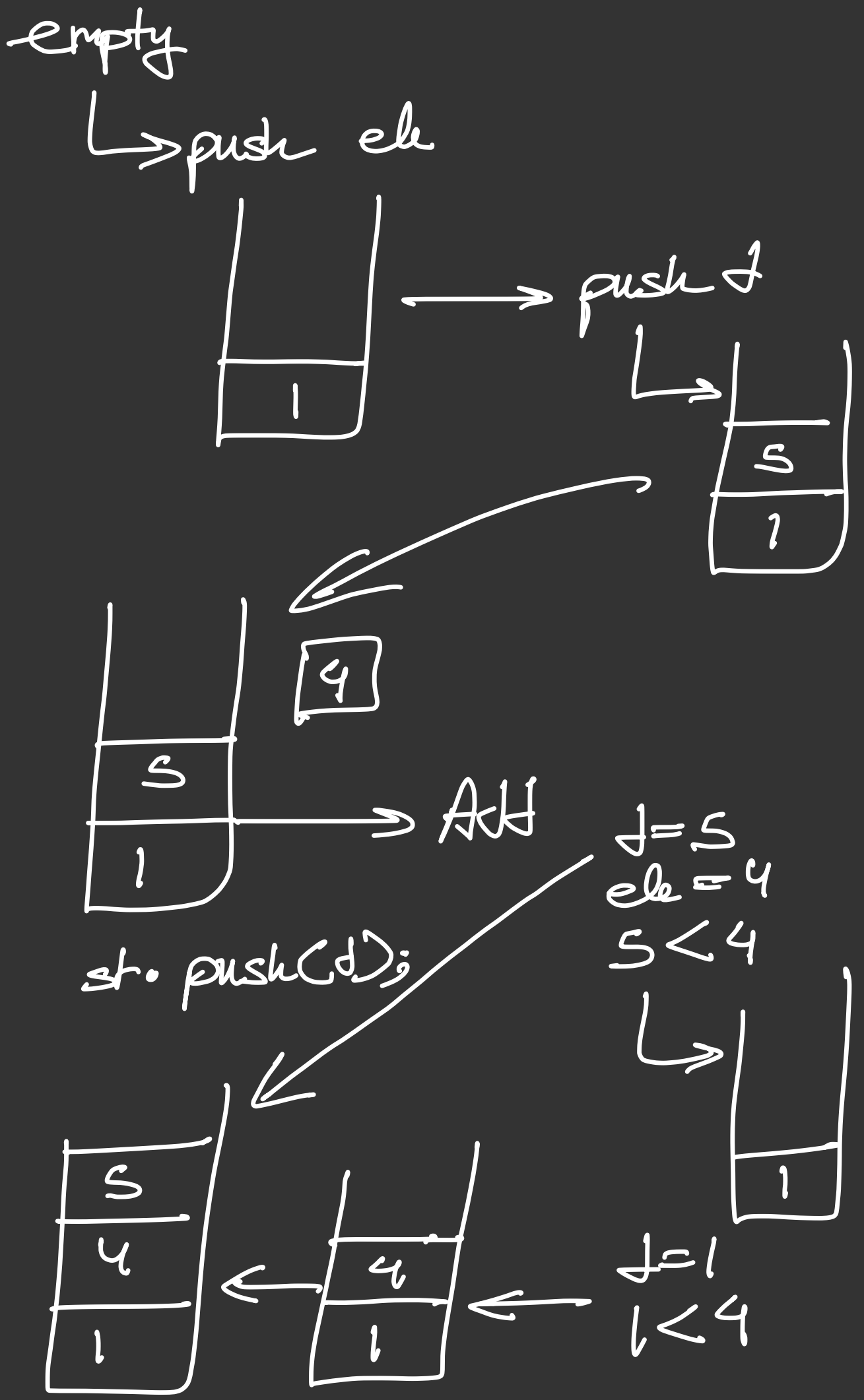
Sort Using Recursion

→ sortStack_usingRecursion.cpp

TC = $O(n^2)$






empty
push ele
1
push 4
5
1
4
5
1
AH
4=5
ele=4
5<4
st.push(4);
1
4
1
4
1
5
4
1
4=1
1<4

(1) Go till the last element in sort

(2) If stack is empty
add the ele

Else,
Add

(i) Check if stack is empty
push ele, return;

(ii) If top < ele
push ele, return;

(iii) Thus if top > ele
store top
pop top element
recursive call

(iv) Then push top element

* Whenever doing pop or fetching top element, make sure Stack is not Empty.

Redundant Parenthesis

⟶ considering all characters other than '(' @ ')' as 'w'

Problems ⟶

(w)

((w))

()

because I am popping 

as soon as this appears in Stack, so anything other than this also the redundant.

⟶ redundant_brackets.cpp

Minimum Cost to Make String Valid.

☆ if odd number of characters — (directly.

(1) if stack is empty and '}' is coming → push

(2) if '{' is coming always push

(3) (i) if '}' is coming and on top is stack is '{' just pop that it is valid, thus nullifies.

(ii) if '}' and stack top is '{' then by just reversing } → { we can make them valid then count++; and pop.

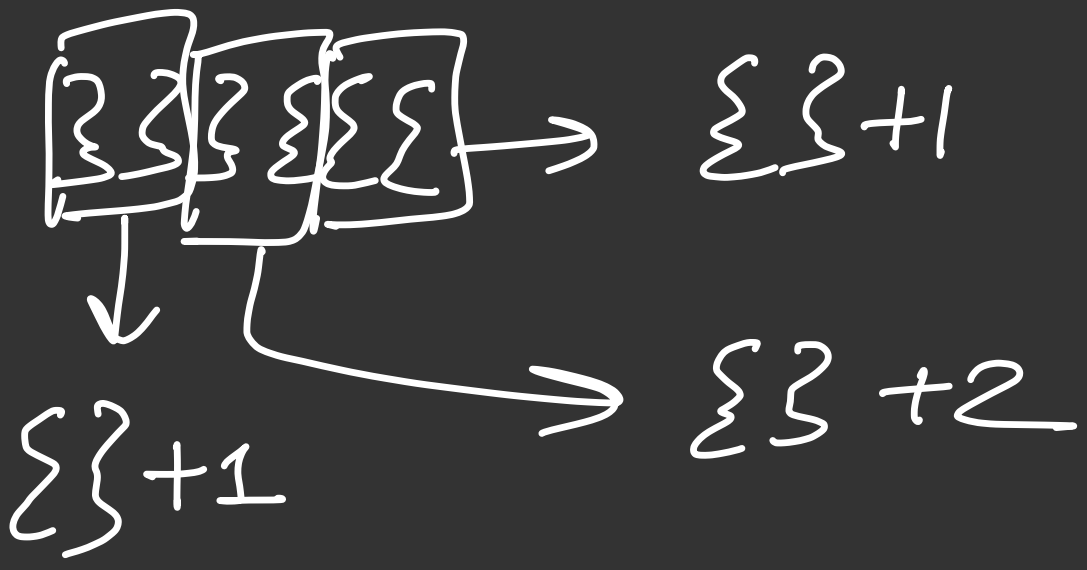
* Meanwhile, adding number of open and close.

Finally, only even number of characters will be left but if they both are in even numbers

}}{{{{

(open/2 + close/2) move cost.

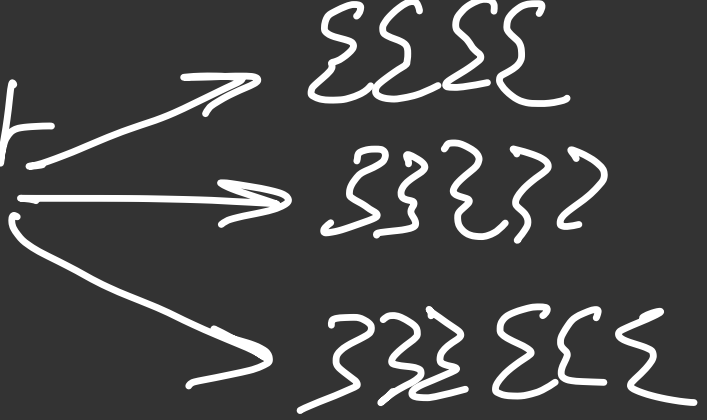
But if both of them are odd



Thus, subtract one from both and add 2 to the cost, then each divide by 2 and added to the total cost

⟶ minCost_validString.cpp

Remaining part will be



Approach 2 ⟶
(Optimised One)

✡ $ans = \left(\frac{open+1}{2}\right) + \left(\frac{close+1}{2}\right)$

(1) Remove the valid brackets

(2) Else keep adding to stack and keep the count

⟶ keep adding 'ξ'

⟶ if 'ξ' check if top = 'ξ' then pop else push 'ξ'

Lecture 56

Next Smaller Elements

i/p → [2, 1, 4, 3]

↓ ↓ ↓ ↓

{1 -1 3 -1}

→ return next smallest element

→ if no smaller element present, return -1.

Approach 1 →

```
for(i=1 to n)
{
    for(j=i+1 to n)
    {
    }
}
```

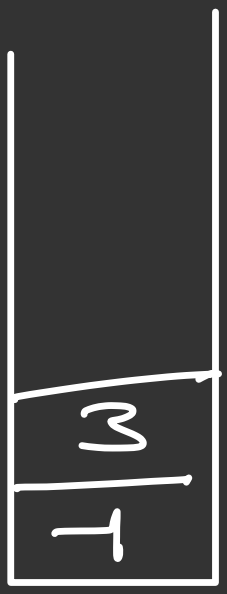
3 3

TC = O(n²)

Approach 2 →

Since we need the smallest part in the right, we should traverse also from right to left.

ele = ~~3~~ 4



→ return -1 for first ele

→ if stack top < ele
  ↳ store stack top

→ if stack top > ele

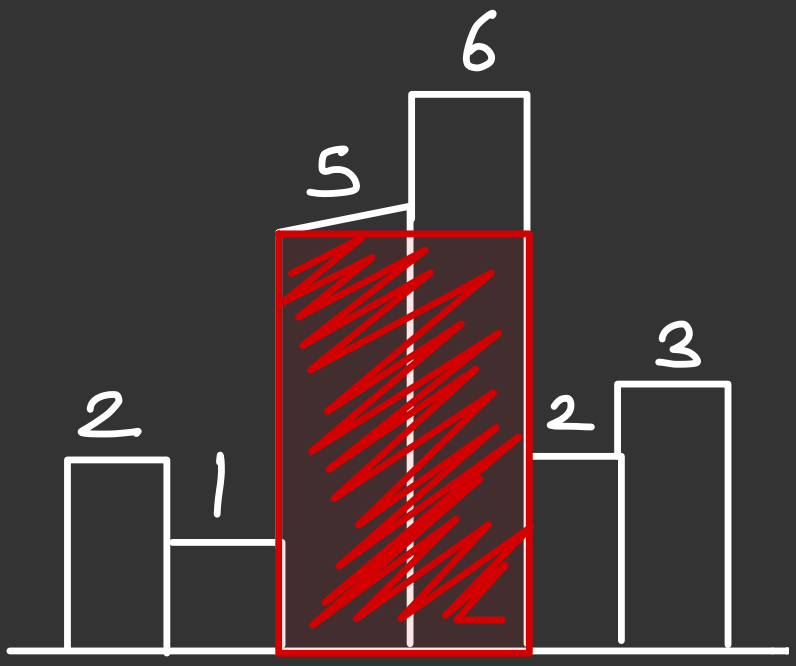
```
while (top > ele)
{
    pop();
}
store top
push(ele);
```

→ as all the elements in stack were greater than ele, thus we popped them, and ele will be the smallest now.

→ nent_smaller_element.cpp

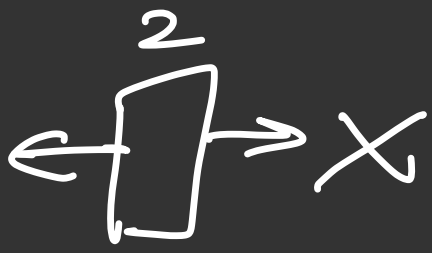
TC = O(n)

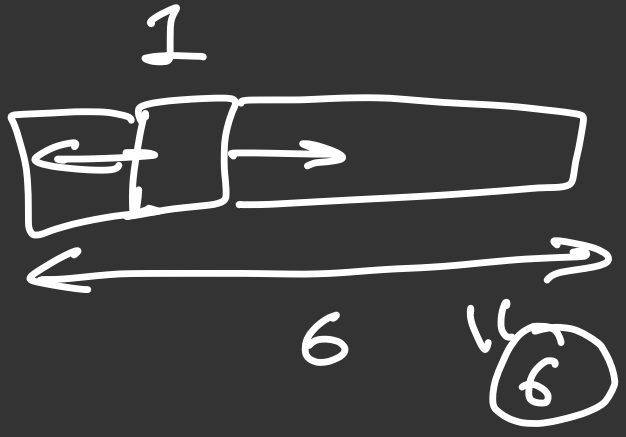
Largest Rectangular Area in Histogram

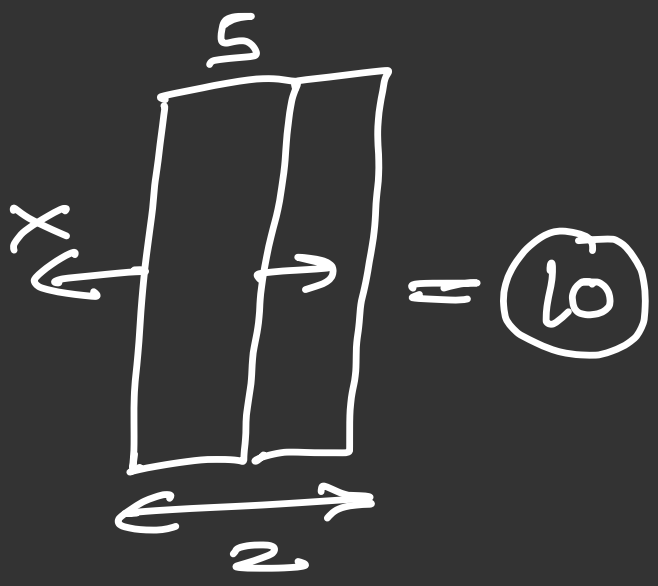


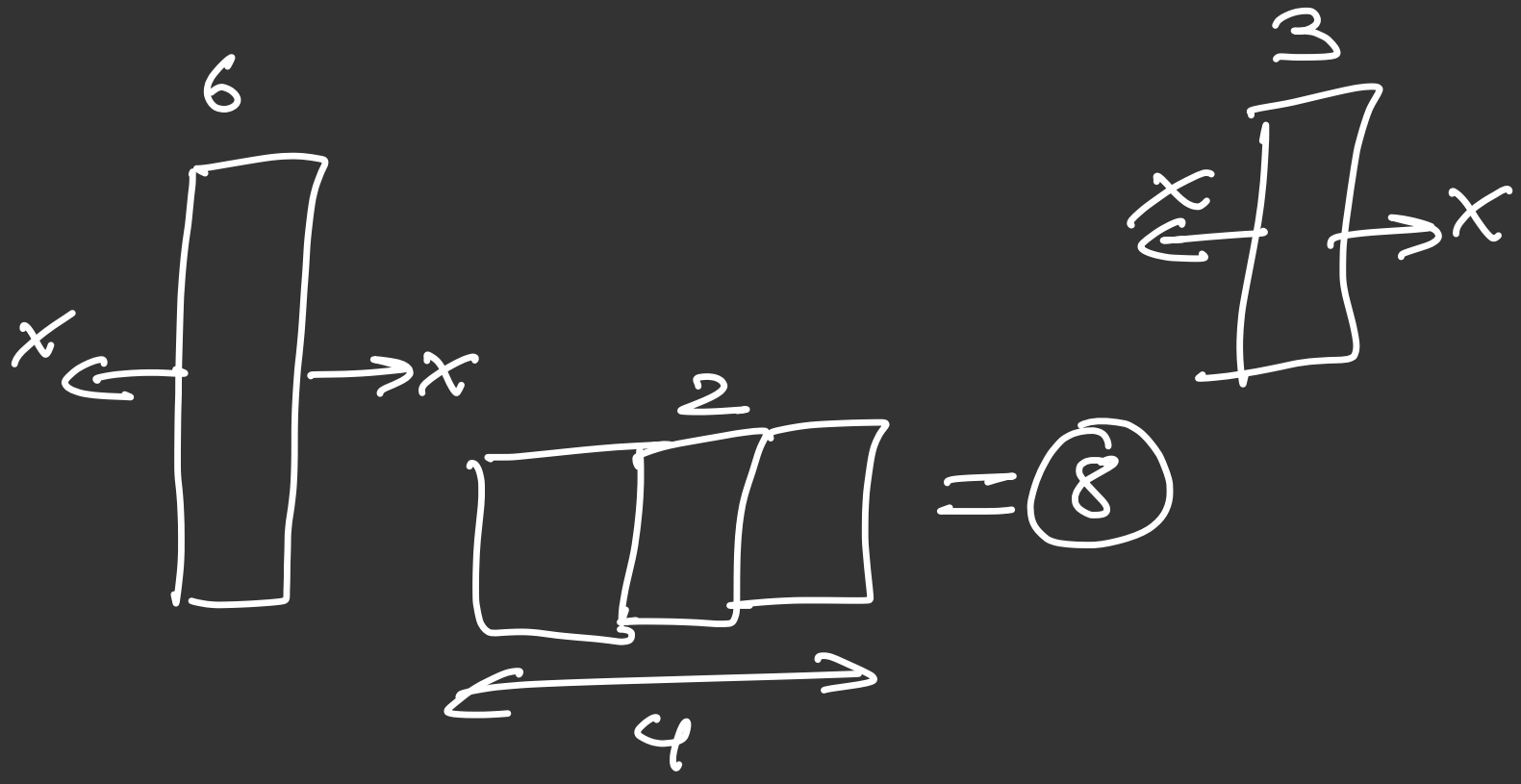
area = length * breadth;

⟶ length is constant, and we need to maximize the breadth









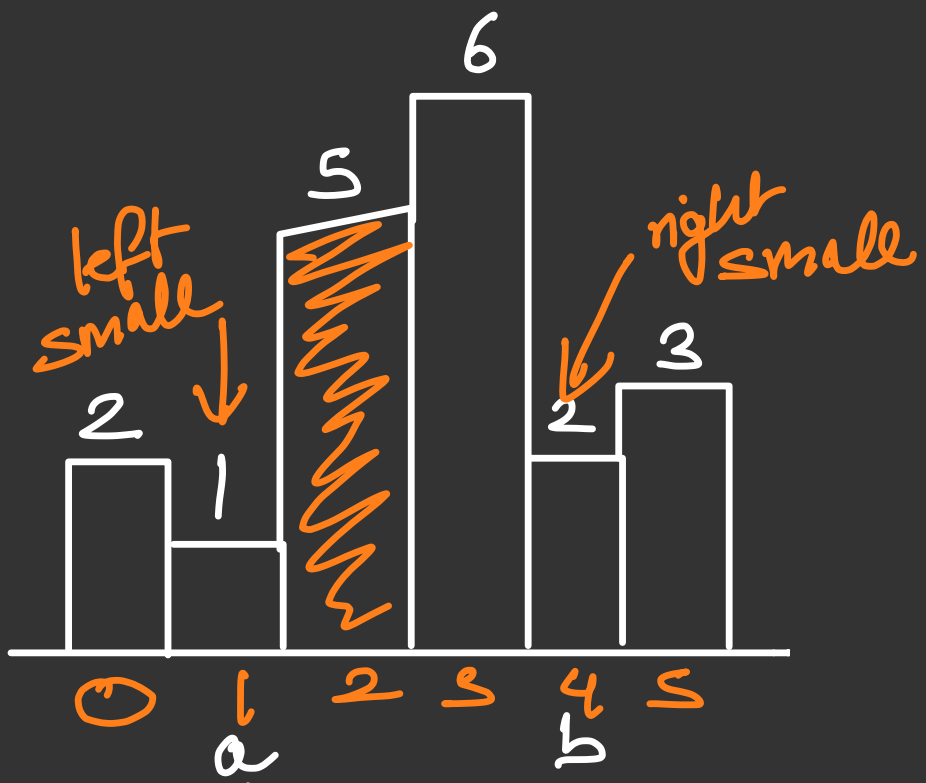
→ We are checking if they can be extended or not in breadth.

```
for (i=0 to n)
{
    while (left)
    ==
    while (right)
    new_area =
    area = max (area, new_area);
}
```

→ largest_rectangle_in_Histogram.cpp

Approach 2 →

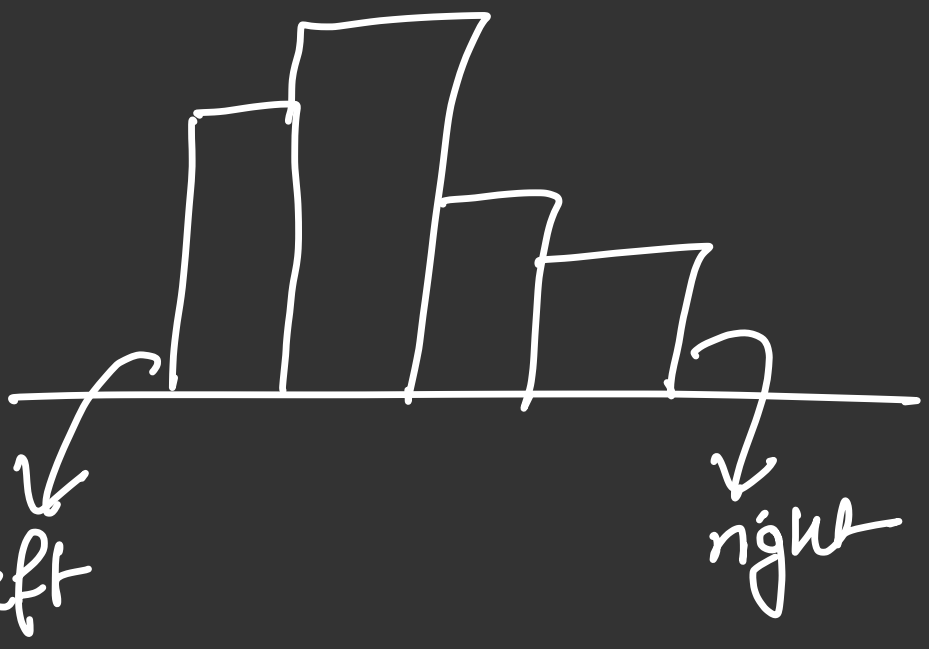
☆ Breadth can be extented till all the elements near by are greater than the element. thus stops before right small and left small.



↳ width = b − a − 1;
= 4 − 1 − 1
= 2

Exception Case:

	2	2	2	2
left =	-1	-1	-1	-1
right =	-1	-1	-1	-1



right = -1

→ means no element in the right is smaller than ele, thus right = n in formula.

Find n^{th} Number → GFG Problems

→ should only contain 1, 3, 5, 7, 9

11 — 31
12 — 33
13 — 35
14 — 37
15 — 39
16 — 51
17 — 53
18 — 55
19 — 57
20 — 59
21 — 61

5 * 2's → 3
* 3's → 5

```
long long ans=0, num=1;
vector<int> v={9,1,3,5,7};
while(N!=0)
{
  ans= ans + v[N%5]*num;
  if(N%5 == 0)
  {
    N=N/5-1;
  }
  else
  {
    N=N/5;
  }
  num= num*10;
}
```

Lecture 57

Celebrity Problem

→ everyone knows celebrity but does not knows anyone.

① Ignore diagonal

Thus, for celebrity

→ row will be all zero

→ column will be all 1 except (i,i)

Approach 1 →

for each element check the two conditions

```
for (i = 0 ——— n-1)
{

    check (row)    O(N)
```

+

check (column) O(N)

if (both true)

cout << i << endl;

}

TC = O(n²)

Approach 2 ⟶

(1) Put all elements in stack

(2) Pick 2 elements

A ⟶ s.top() ⟶ s.pop()

B ⟶ s.top() ⟶ s.pop()

if (A knows B)

A is not a celebrity

if (B knows A)

B is not a celebrity

(3) Do this stack size != 1

Still need to check the element for celebrity.

"POTENTIAL CELEBRITY"?

→ findingCelebrity.cpp

TC = O(N)

Max Rectangle in Binary Matrix with all ones

Hint: like Max area in histogram

example:

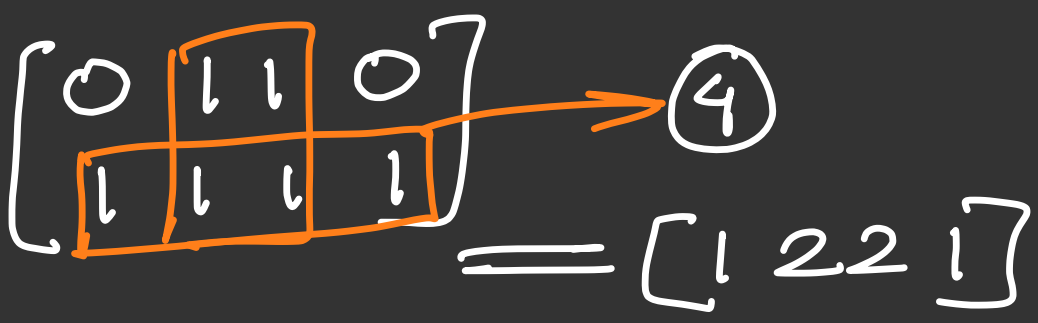
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

⟶ Ist row

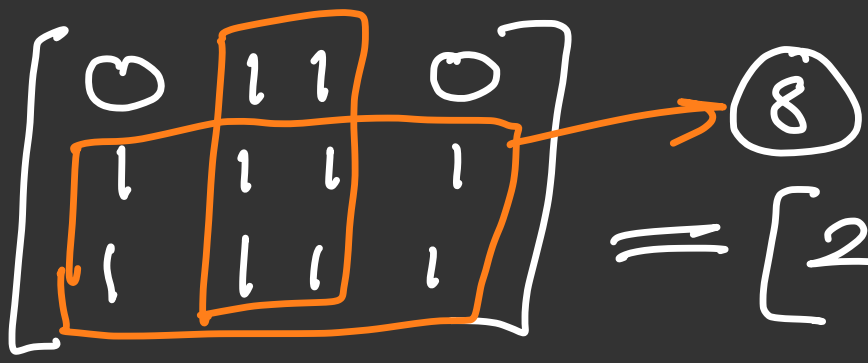
[0 1 1 0]

max area = 2

⟶ Ist two rows

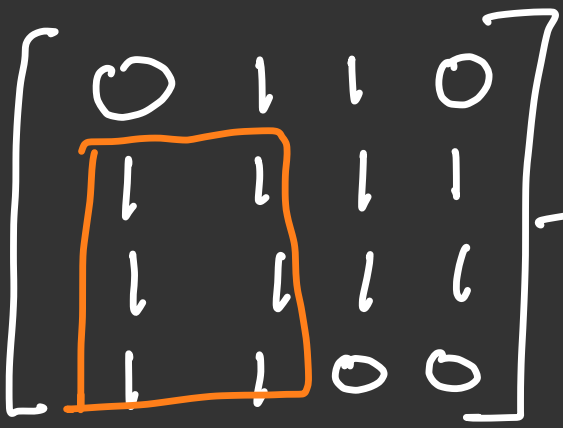


→ Ist three rows



= [2332]

→ Ist four rows



= [3400]

important

Place = 8

Logic →

1. Max Area for 1st row
2. For every remaining row
   ↳ add the above rows.

(1) Max area of histogram for each row from top to that row.

(i) Calculate the smaller element left and right

(ii) As for the inter breadth can be till the elements >= to the current element.

(2) Finally you get the answer.

→ maxRectangle_binaryMatrix.cpp

TC = O(n x m)

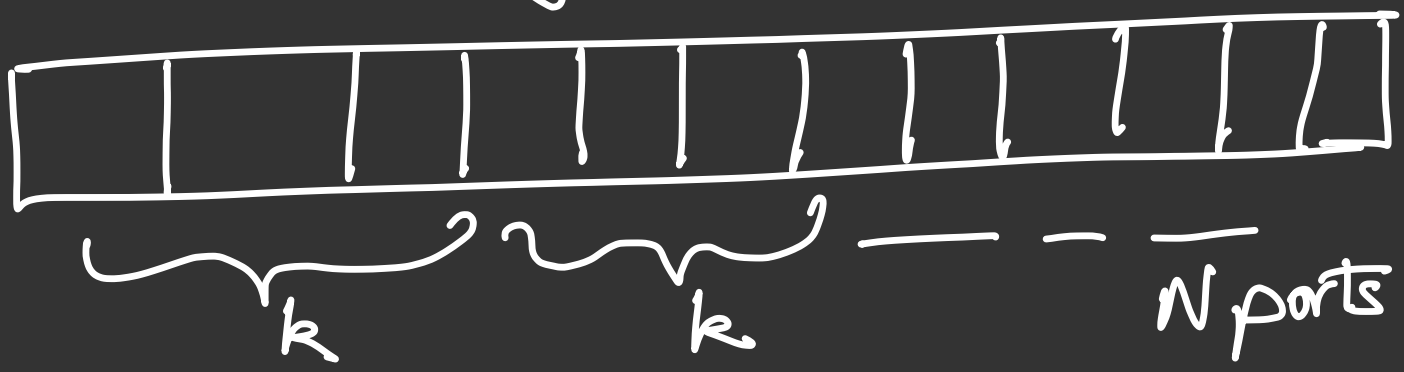
SC = O(m)

Lecture 58

N Stacks in an array

Approach 1 ⟶

divide array into N parts



$n/N = k$

⟶ But space will not be optimised

Because, one stack might get filled and one we try to add another element in that branch, but space available in array.

Approach 2 ⟶

two things

- top[] — represent index of top element
- next[]
  - if arr[i] has an element ↓ point to next element after stack top (to keep a chain)
  - ⟶ else, point to next free space

N ⟶ no. of stacks

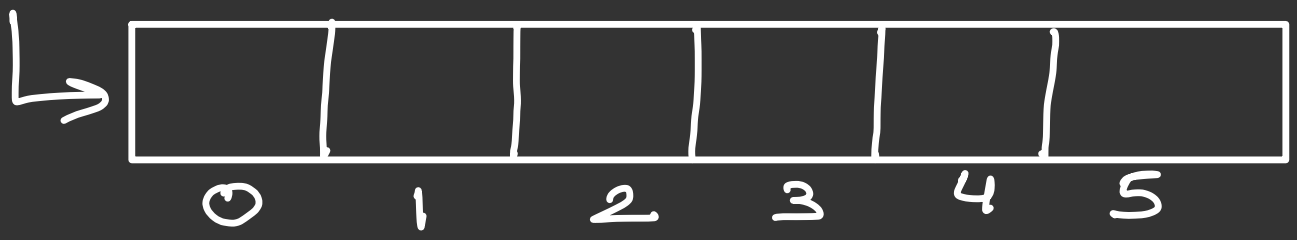
S ⟶ size of the array

example:

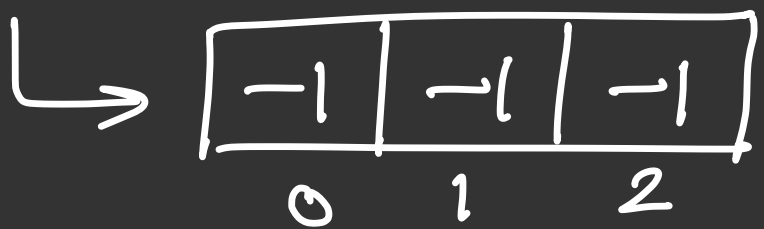
$N = 3$

$S = 6$

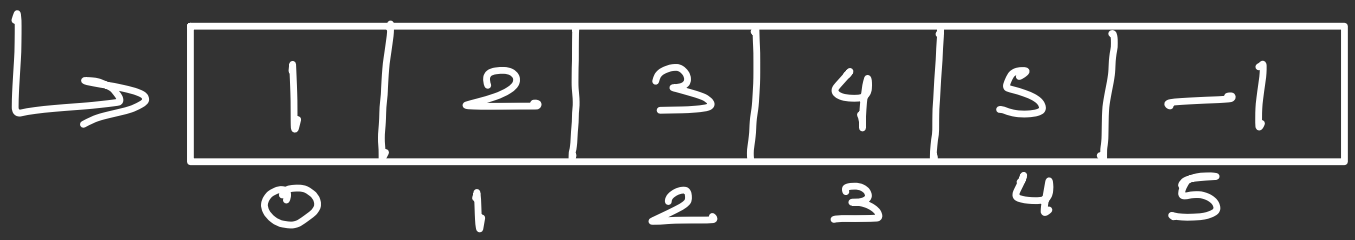
arr



top[3];



next[6];



→ Initially all top = -1 and next points to the next empty space.

freespot = 0

↳ the first free empty space

PUSH →

```
if (freespot == -1)
    return false;
```

(i) find index

```
int index = freespot;
```

(ii) update freespot

```
freespot = next[freespot];
```

(iii) insert in array

```
arr[index] = x;
```

(iv) update next

```
next[index] = top[n-1];
```

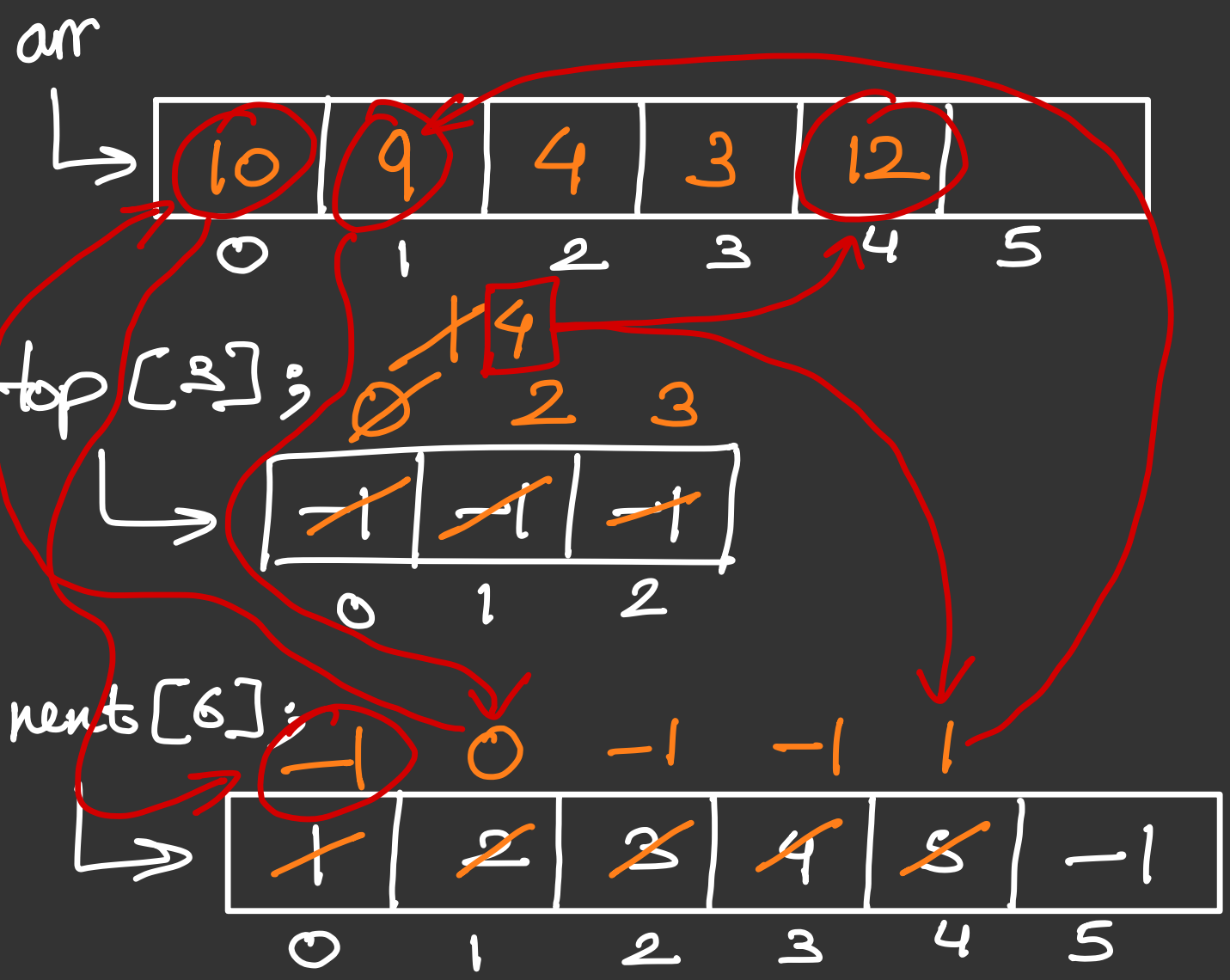
(v) update top

```
top[n-1] = index;
return true;
```

POP →

```
if (top[m-1] == -1)
    return -1;
```

(i) `int index = top[m-1];`
(ii) `top[m-1] = next[index];`
(iii) `next[index] = freespot;`
(iv) `freespot = index;`
`return arr[index];`



freespot = ~~0 1 2 3 4 5~~

index = ~~0 1 2 3~~ 4

push(10, 1)
push(9, 1)
push(4, 2)
push(3, 3)
push(12, 1)

☆ top → you store the index where the top element of the stack is store.

next → at next of same index we first store the current top

☆ First store top[m-1] in next, and then update top[m-1] as current index

TRAVERSAL

1st element

index = top[n-1]

element = arr[index]

next element

index1 = next[index]

2nd element

index = index1

element = arr[index]

next element

index1 = next[index]

Do till next[index] == -1

While popping

1. top element index
2. Now, the next element index is at the top, updates top
3. What freespot is now should be the next empty space, the
   next[index] = freespot (current)

   Now update freespot
   freespot = index;
4. return the element

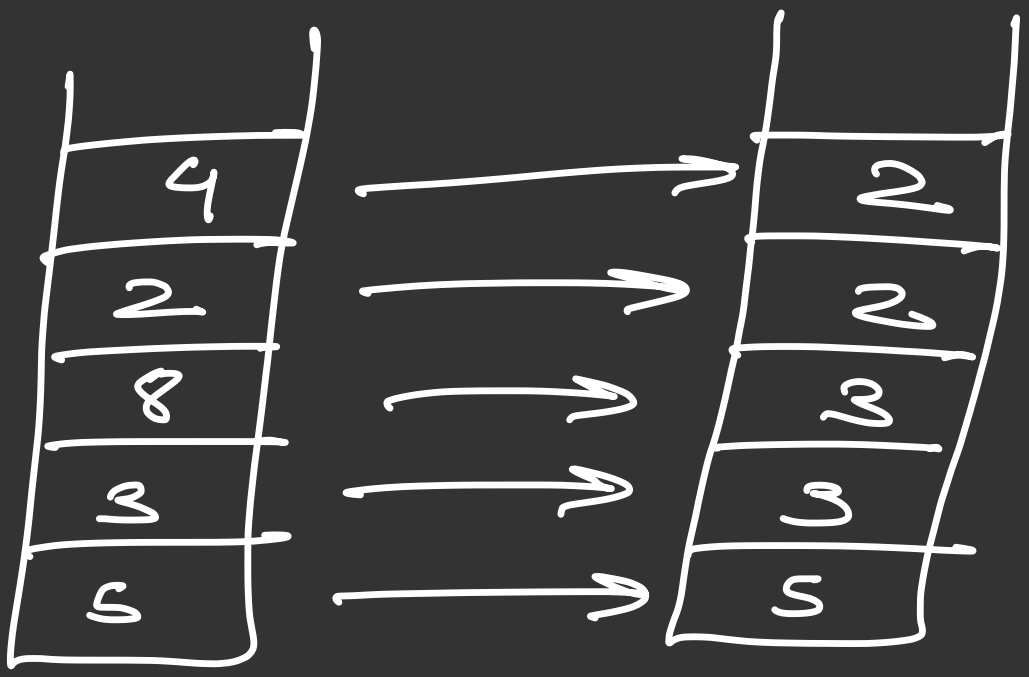
→ N_Stacks.cpp

Lecture 59

Design a stack that supports getMin() in O(1) → TC and SC

Approach 1 →

☆ A seperate to store the minimum till now and at pop() both stacks get popped.



SC = O(n)

Approach 2 →

* for O(1) space complexity, you have to use a variable.

int mini = INT_MAX;

PUSH

→ Check overflow condition

→ for 1st element
  ↳ normal push
  ↳ update mini

→ for other elements
  → if (cur < min)
    val = (2 * cur) - mini;
    push (val);
    update mini;

→ else
normal push

pop

→ check for underflow
→ if (s.top() > mini)
↳ normal pop

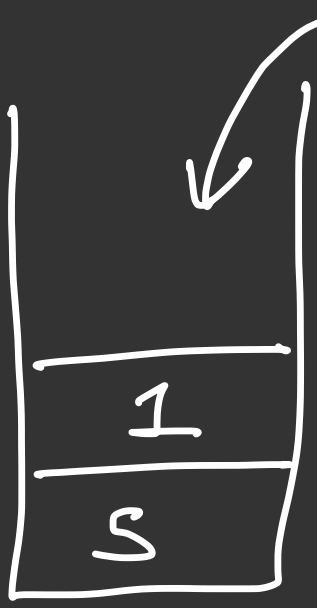
else
↳ update mini
val = 2*mini − curr
mini = val
pop();

getMini()
↳ return mini;

Why $(2 * cur) - mini$

$(2 * mini) - cur$?

⟶ to use current minimum to find previous minimum



$(2 \times 3) - 5 = 1$

$= 2 \times n - 5 = 1$

new_mini = 1

for prev_min $= (2 \times 3) - 1$

$= 5$

$(2 \times n) - 1 = 5$

$- 5 = 1$

→ getMin_O(1)complexity.cpp