STACKS AND QUEUES

STACK

```
class Stack {
    int size = 100
    top = -1

void push(int n) {
    top++;
    if (top == size-1) return;
    top++;
    arr[top] = n;
}

void pop() {
    int n = arr[top];
    top--;
    return n;
}
```

QUEUE

```
class Queue {
    arr[]
    start = -1
    end = -1
    currsize = 0;

push (element) {
    if (currsize == maxsize) return

    if (end == -1) {
            start = 0
            end = 0
    } else {
            end = (end + 1) % maxsize;
    }

    arr[end] = element
    currsize++;
}
```

PUSH

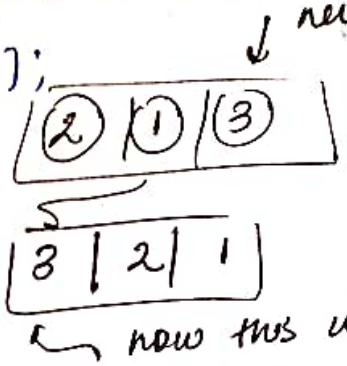
STACK USING QUEUE

Approach

→ push(n) → push the element in the queue

use a for loop of size() - 1, remove element from queue and again push back to the queue, hence the most recent become the most former element.

```
class Stack {
    Queue<Integer> q = new LinkedList<>();

    void push(int n) {
        q.add(n);
        for (int i = 0; i < q.size() - 1; i++) {
            q.add(q.remove());
        }
    }

    int pop() {
        return q.remove();
    }

    int top {
        return q.peek();
    }
}
```



↳ now this will get removed first recent one LIFO

... returns null when empty...

```
public int pop() {
    if (start == -1) return
    int popped = arr[start];
    if (currSize == 1) {
        start = -1;
        end = -1;
    } else {
        start = (start + 1) % maxSize;
    }
    currSize--;
    return popped;
}

public int top() {
    if (start == -1) { return }
    return arr[start];
}
}
```

...QUEUE USING STACK...

Approach 1

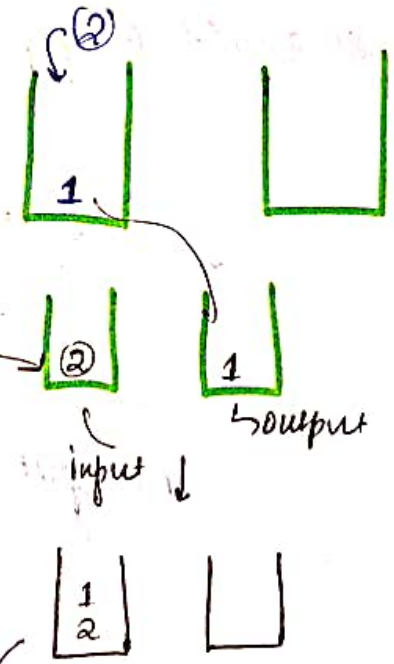
class Queue{

Stack<Integer> input = new Stack<>();
Stack<Integer> output = new Stack<>();

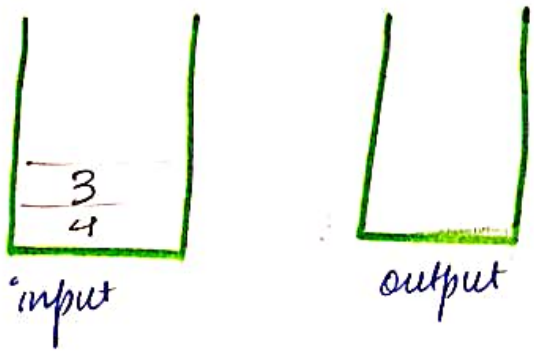
O(n)

void push(x){
while(input.empty() == false){
output.push(input.peek());
input.pop();
}
input.push(x);
while(output.empty() == false){
input.push(output.peek());
output.pop();
}
}
}

int pop(){
if(input.empty()){ return }
int val = input.peek()
input.pop();
return val;
}

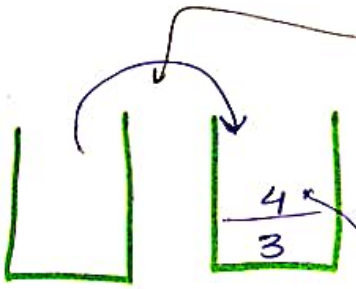


APPROACH : 2



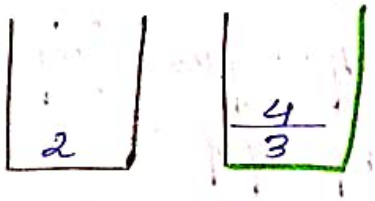
pop : 4 should be popped first :-

two way



New put

O(n)

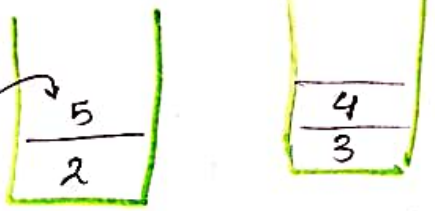


when already there in output queue just pop

O(1)

depends on if element is there or not in

push



just push O(1)

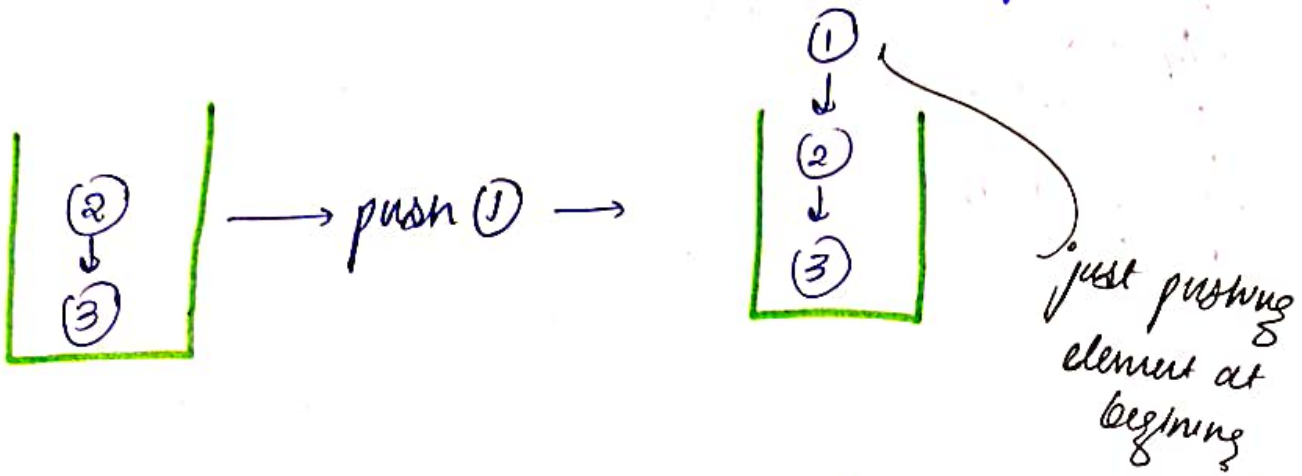
Everything happens in pop only.. Or

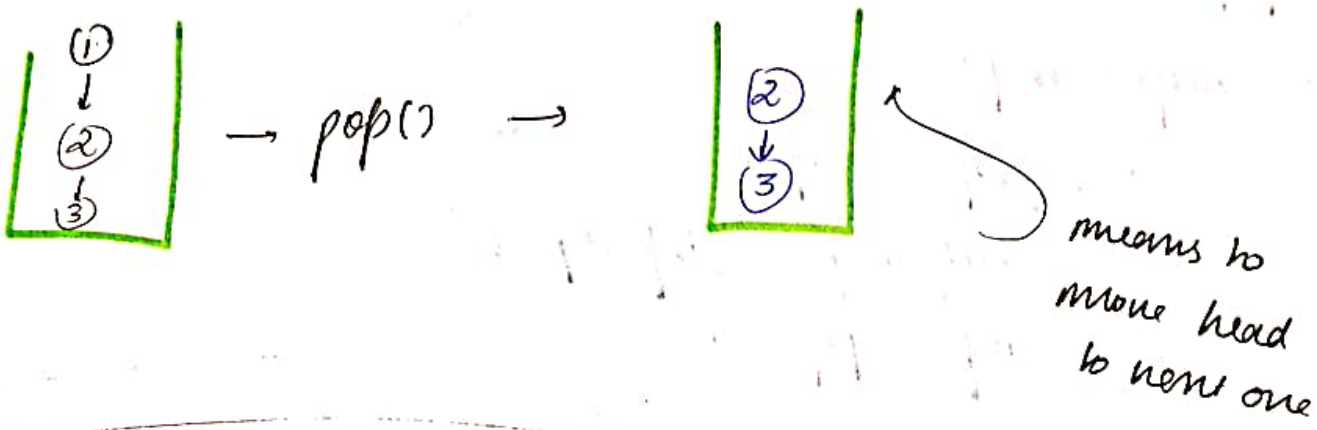
```
MyQueue {
    input }  ⟶ 2 stacks
    output
void push(int n) {
    input.push(n);
}

int pop() {
    if (output.empty()) {
        while (input.empty == false) {
            output.push(input.peek())
            input.pop()
        }
    int n = output.peek()
    output.pop()
    return n;
}

int peek() {
    if (output.empty()) {
        shift input element to
            output
    }
    return output.peek()
}

int size() {
    return output.size() + input.size();
}
```

O(n) or O(1)

O(n) or O(1)

...STACK USING LINKED LIST...

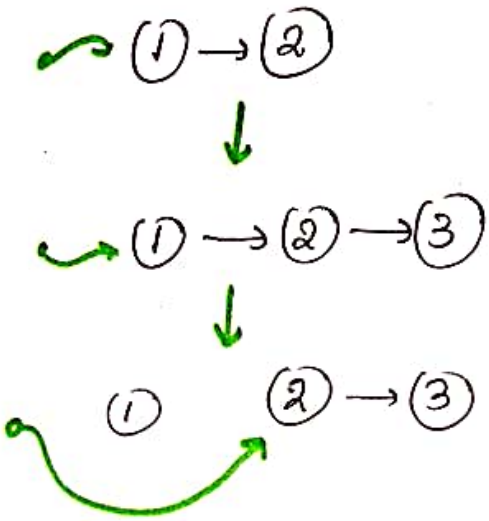





```
StackPush (n) {
    StackNode element = new StackNode(n);
    element.next = top;
    top = element;
    size++;
}
```

```
StackPop(){
  if (top == null) return -1;
  int topData = top.data;
  StackNode temp = top;
  top = top.next;
  return topData;
}
```

QUEUE USING LINKED LIST




```
Node Queue {
    front = = rear = null;
    size = 0
}

void enqueue (value) {
    QueueNode temp;
    temp = new QueueNode (value);
    if (temp = = null) {
                        } full
    }

    else {
        if ( front = null ) {
                front = rear = temp;
        }
        else {
            rear.next = temp;
            rear = temp
        }
    }
                    size ++
}
```

```
void dequeue(){
  if(front == null){
        empty
    }
  else{
      QueueNode temp = front;
       front = front.next;
       size--;
    }
}
```

INFIX TO POSTFIX

✔ Algorithm steps

→ initalize an empty stack (for operators)

→ initalize an empty result (postfix expression)

→ scan expression from left to right

→ for each token:

- operand → add directly to postfix expression
- left '(' → push onto stack
- right ')' → pop until '(' is encountered and add to postfix and discard both paranthises.

- operaters (+, -, *, /, ^):

if precedence of current operater ≤ precedence of top of stack
pop and top should not be '('
push operater onto stack

- after expression pop all remaining operators from stack to prefix postfix

			Associativity
^	→ 3	→	right to left
*/%	→ 2	→	left to right
+ -	1	→	left to right.

Associativity

A - B - C

↓ first two then

A - C

↳ this (left to right evaluation when precedence equal)

A 2 ^ 3 ^ 4

↓ first two

2 3 4 ^ ^ or not ~~2 ^ 3 ^ 4~~ 2 3 ^ 4 ^ X

2 ^ 81

↳ then two (right to left evaluation when precedence is equal)

* otherwise wrong output

INFIX TO PREFIX

$(A+B) * C - D + F$

Reverse the given infix

Do infix to postfix conversion

↳ do not pop when precendence is equal

★

↳ we only pop when right associative operator

Reverse the answer

$(A+B) * C - D + F$

↳ $F + D - C *) B + A ($ opening ⟷ closing

↳ $F + D - C * (B + A)$

↓ postfix

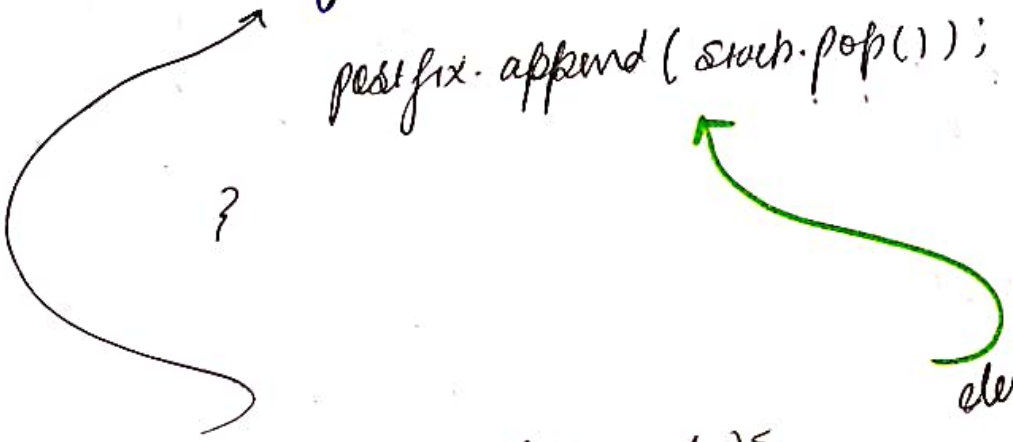
→ $FDCBA+*-+$

↓ reverse

$+ - * + A B C D F$

★ Associativity check

```
while (!stack.isEmpty() && precedence(curr) <=
        precedence(stack.peek()) &&
        isLeftAssociative(curr)) {
    postfix.append(stack.pop());
}
```



element pop only when left to right associativity

```
isLeftAssociative(char ch) {
    return ch != '^';
}
```

O(n) — time

O(n) — SPACE

```
while (!stack.isEmpty() &&
        precedence(stack.peek()) > precedence(curr) ||
(precedence(stack.peek()) == precedence(curr) &&
    isRightAssociative(currentOp)) {
        result.append(stack.pop());
}
```

when == then associativity should be right

why?

we scan reverse the expression

∴ associativity gets flipped

left (+, −, *, /) becomes right

right (^) become left

2^3^4 → we do not pop as exp evaluates to right to left

↓

4^3^2 → we need to pop as ~~right~~ associativity changes and needs to evaluate 4^3 first.

2^3^4

↓

4^3^2 → 43^2^

↳ ^2^34 ✓

but when not poped

2^3^4

↓

4^3^2 → 432^^ ↷ ^^234 ✗

drawing evaluation

this will be evaluated first wrong

time - $O(\frac{N}{2}) + O(\frac{N}{2}) + O(2N)$

(worse)

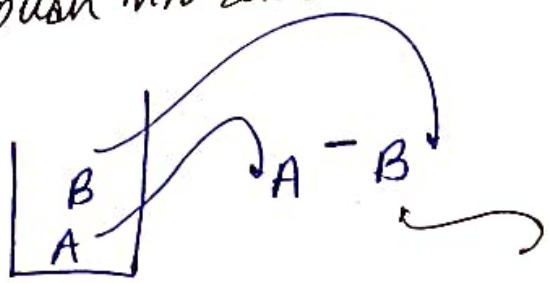
SPACE - $O(N)$

POSTFIX TO INFIX

AB - DE + F * /

#Scan left to right

- operand are pushed onto stack
- when operator can pop two operand combine and push into stack



first popped should be on right

A B - D F + F Y /

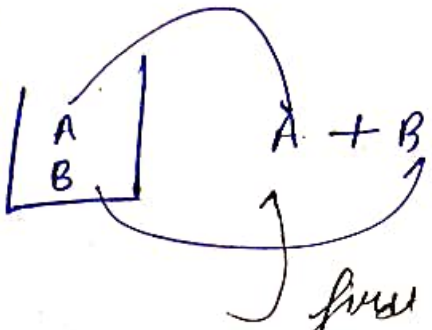
	St
A	A
B	AB
-	(A - B)
D	(A - B), D
E	(A - B), D, E
+	(A - B), (D + E)
F	(A - B), (D + E), F
*	(A - B), ((D + E) * F)
/	((A - B) / ((D + E) * F))

empty stack

PREFIX TO INFIX

Scan from right to left
Same as before

but A



first opopped will come in front.

POSTFIX TO PREFIX

AB - DE + f * /

pattern = (operator) (top 2) (top 1)

		st
A	———	A
B	———	AB
-	——→	-AB
D	———	-AB, D
E	———	-AB, D, E
+	———	-AB, +DE
f	———	-AB, +DE, f
*	———	-AB, *+DEf
/	——→	/-AB*+DEf

PREFIX TO POSTFIX

/-AB*+DEF

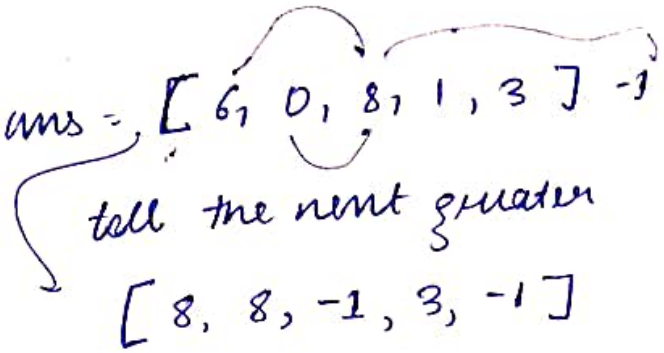
scan from right to left

pattern = (top 1) (top 2) (operator) ★

	St
A	A
F	F
E	F, E
D	F, E, D
+	F, DE+
*	DE+F*
B	DE+F*, B
A	DE+F*, B, A
-	DE+F*, AB-
/	AB-DE+F*/

NEXT GREATER ELEMENT

MONOTONIC STACK → when elements are stored in specific order.



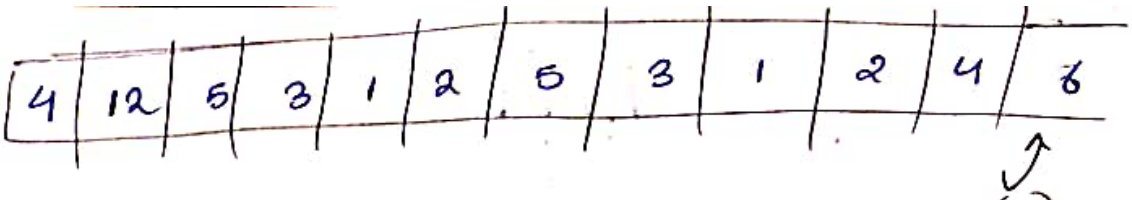
BRUTE

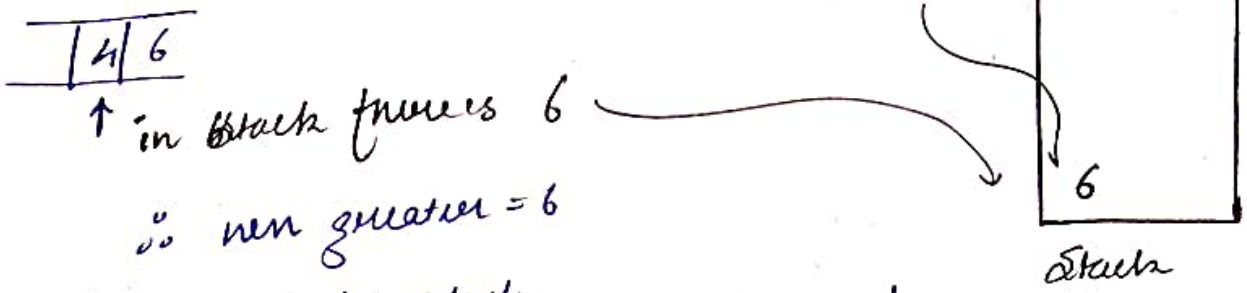
iterate and find next greater $O(n^2)$

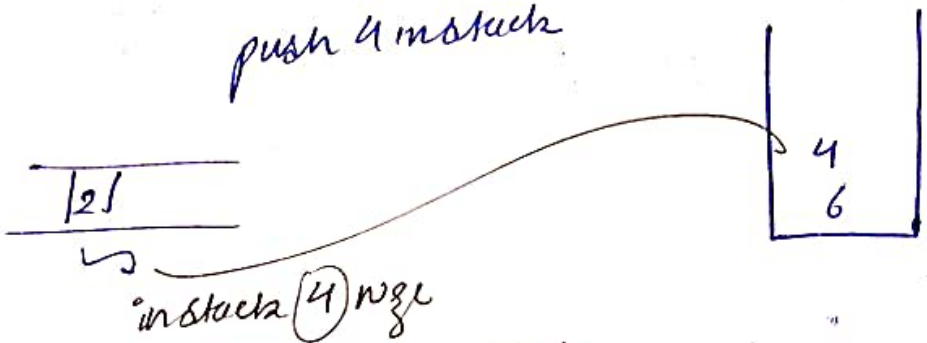
```
for (i=0 ⟶ n-1) {
    for (j=i+1) ⟶ {
        if (arr[j] > arr[i])
```

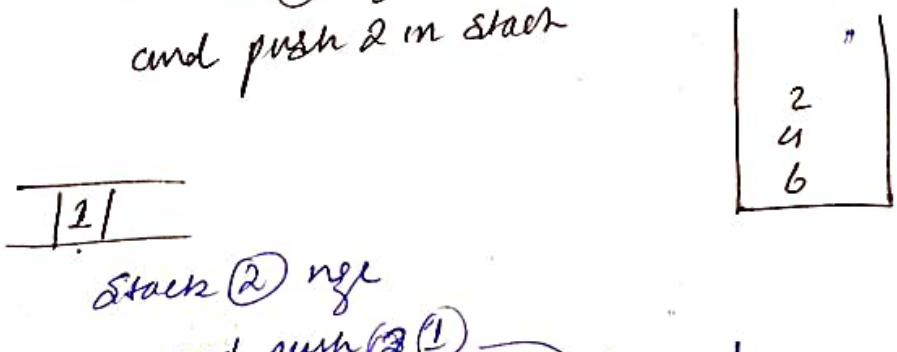
OPTIMAL

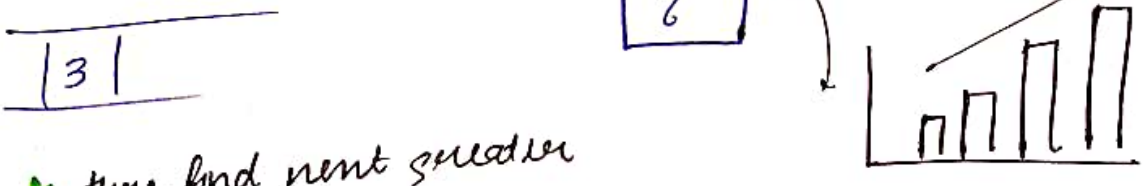
traverse from back



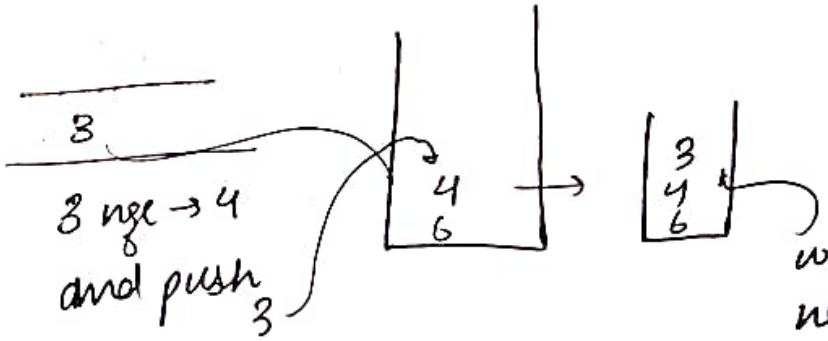






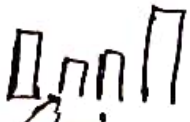


if we insert 3 directly then



mean specific order

we do not need middle



smaller one

to find nge

O(n)

```
findNGE ( ) {
    for (i = n-1 ⟶ 0) {
        while (!st.empty() && st.top() <= arr[i]) st.pop();
        if (st.empty()) nge[i] = -1
        else nge[i] = st.top()
        st.push(arr[i])
    }
    return nge
}
```

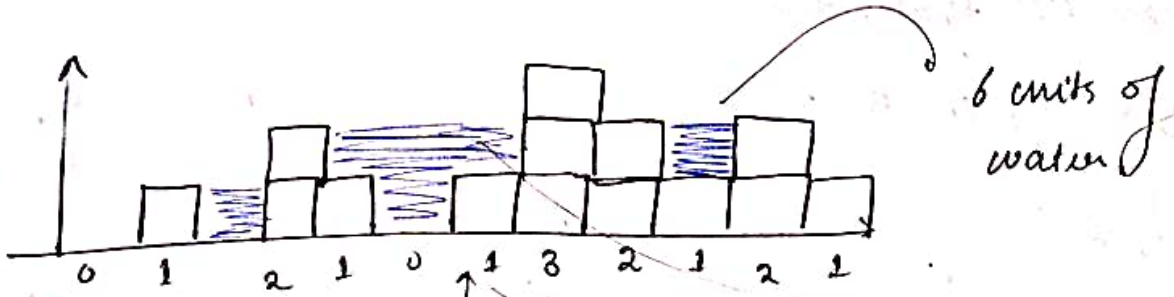
Time - O(2N)

when last element is biggest one and at most only n element can be removed overall

∴ N + n = 2N

SPACE → O(n)

TRAPPING RAIN WATER



BRUTE

for every (index)

Σ min (leftMax, rightMax) − arr[i] = 1 unit

< arr[i] < arr[i]

```
total = 0
for (i = 0 ⟶ n-1) {
    if (arr(i) < leftmax && arr(i) < rightmax) {
        total += min( , ) - arr(i)
    }
}
return total
}
```

leftmax = prefixMax[i]

rightmax = suffixMax[i]

prefixMax[n]

↑

max till that index

arr = [2, 1, 0, 5, 3]

prefix = [2, 2, 2, 5, 5]

→ [2, 2, 2, 5, 5]

we have got max to left of index

prefix[0] = arr[0]

for (i = 1 → n-1) {

prefix[i] = max(prefix[i-1], arr[i])

}

time → O(3N) → O(N)

SPACE → O(2N)

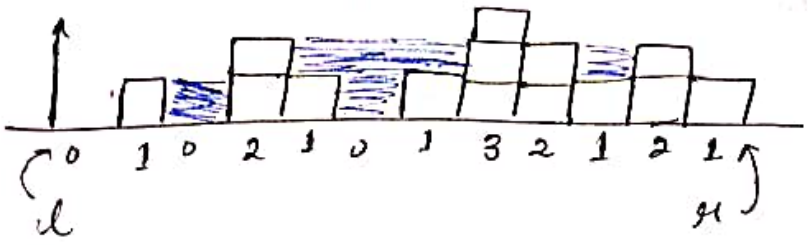
suffix[n]

arr = [2, 1, 0, 5, 3]

suffix = [5, 5, 5, 5, 3]

[5, 5, 5, 5, 3]

OPTIMAL



now we only need length of smaller one.

lmax = 0

↑ left max biggest building

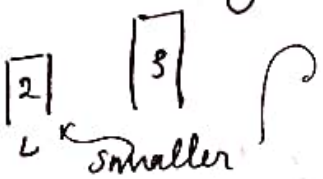
rmax = 0

↳ right max biggest building

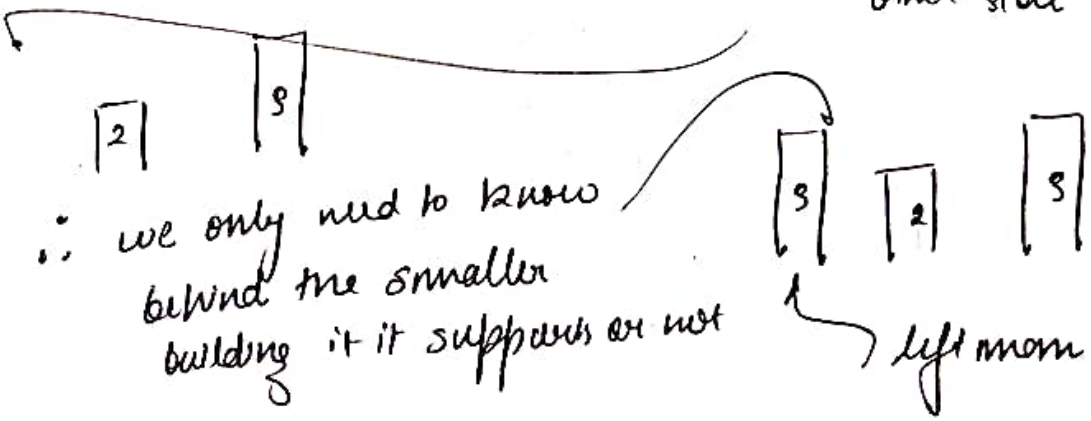
(l) (r)

current two building we are processing

now during traversal one of them would be smaller so would know that there is surely a taller building on the other side



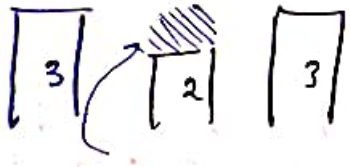
∴ we know we have the support from other side



∴ we only need to know behind the smaller building if it supports or not

∴ leftmom ≥ smaller one

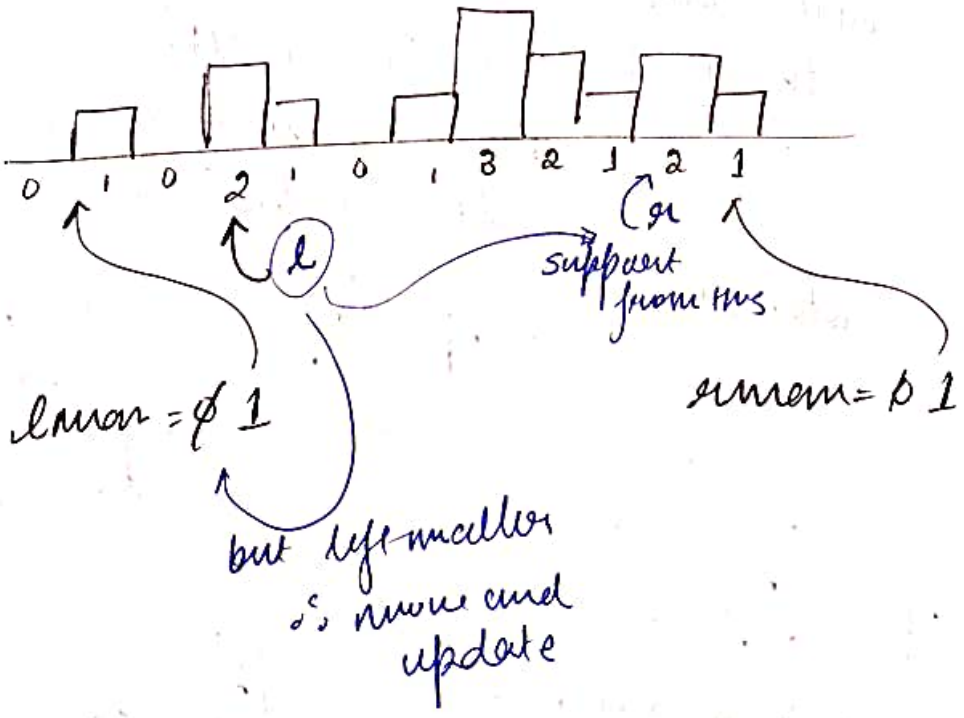
∴ it can store water



concept

traverse two building figure out smaller one we know opposite side supports us ∴ o check behind if true it supports or not ∴ cacluate water unit is supported

after this update right or left mom according to L or R building



..SUM OF SUBARRAY MINIMUM..

arr = {3, 1, 2, 4}

sum all the min elements in all subarray

{3} {1} {2} {4} {3,1} {3,1,2} {3,1,2,4}
{1,2} {1,2,4} {2,4}

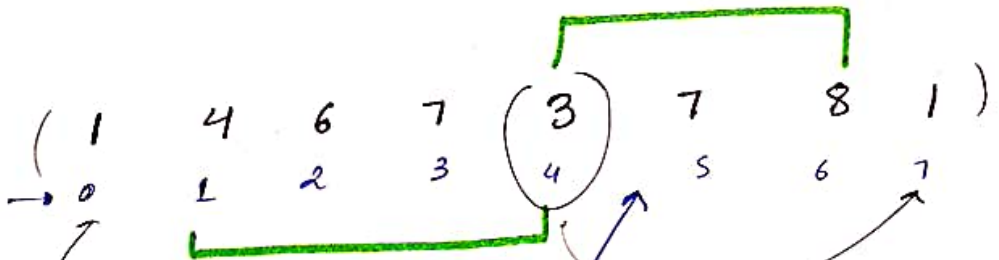
3+1+2+4+1+1+1+1+1+2 = (17)

BRUTE

→ enumerate all subarray
→ figure out minimum while extending array

↑ $O(n^2)$

OPTIMAL



previous smaller index = (0)
4-0=4 element that can form subarray

next small index = (7)
∴ 7-4=(3) element from can form subarray

total subarray including (3) = (4×3) = 12×3 = (36)
↑ element ↑ to add

```
int findtotal(arr)
lmax=0, rmax=0, total=0, l=0, r=r-1
while(l<r){                    // support from right
  if(arr[l]<=arr[r]){
    if(lmax>arr[l]){
      total = lmax - arr[l]
    }
    else lmax = arr[l]
    l=l+1
  }
  else{
    if(rmax>arr[r]){
      total = rmax - arr[r]
    }
    else rmax = arr[r]
    r=r-1
  }
}
return total
}
```

O(n)

Edge case when element equal

arr = [1 , 1]
0 1

nse → 2 2

pse → -1 -1

0 inden = left = 0 − (−1) = 1
right = 2 − 0 = 2
1 × 2 = ②

[1 .]
[1, 1]

but

1 inden left = 1 − (−1) = 2
right = 2 − 1 = 1
2 = 1
[1 , 1] ★

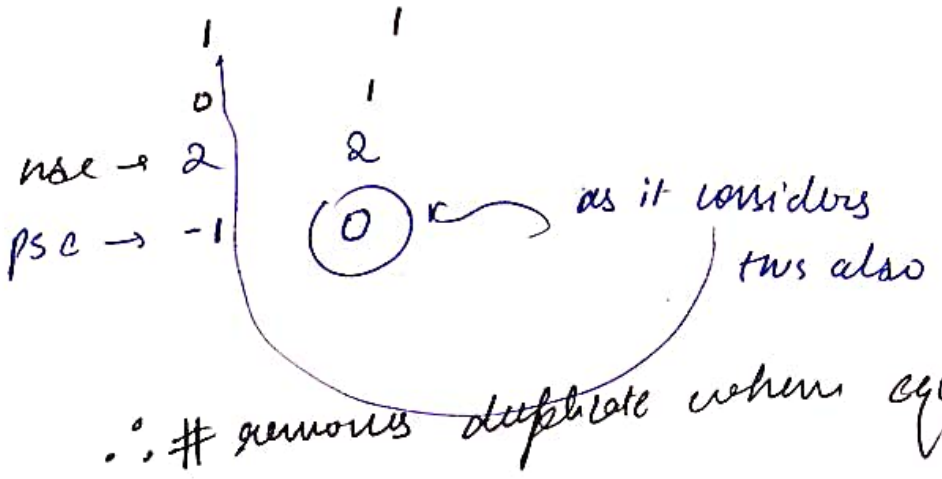
this included twice

∴ only consider left one or right one

∴ do not consider on both

when pse → do not look < but include = (equal) also

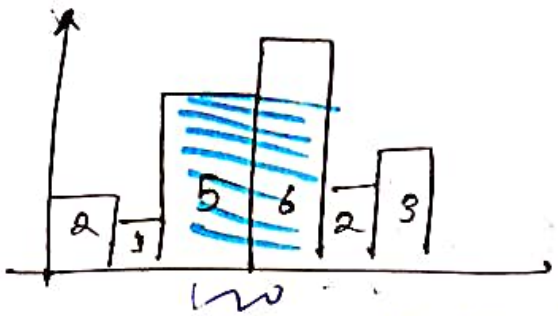
∴ psmaller or equal



∴ # removes duplicate when equal

```
int sum(arr){
    nse = findNse(arr)
    psee = findPSee(arr)
    total=0 , mmod = (int)(1e9+7)

for(i=0 → n-1){
    left = i - psee[i]
    right = nse[i]-i;
    total = (total + (right*left * arr[i])%mmod)%mmod

}
return total
```

... LARGEST RECTANGLE IN HISTOGRAM -



5+5 = 10 being the largest

BRUTE

go to every index

2, 1, 5, 6, 2, 3

↓

it cannot go left or right ∴ 2×1 = 2

2, 1, 5, 6, 2, 3

↑

cannot go to all ∴ 1×6 = 6

2, 1, 5, 6, 2, 3 ∴ 5×2 = 10

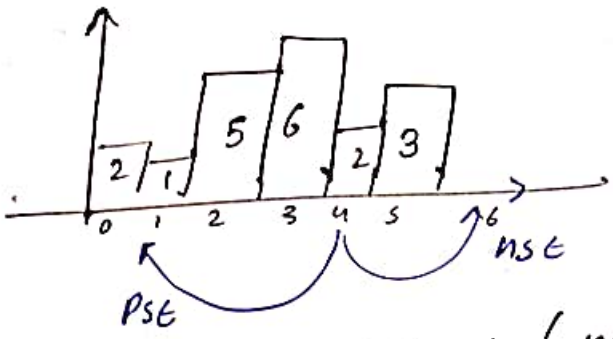
↑

cannot go to 6 only ∴

similarly other

∴ we can go the smaller element only

∴ we need (pse, nse)



arr[i] × (nse − pse − 1)

↑ height ↑ width

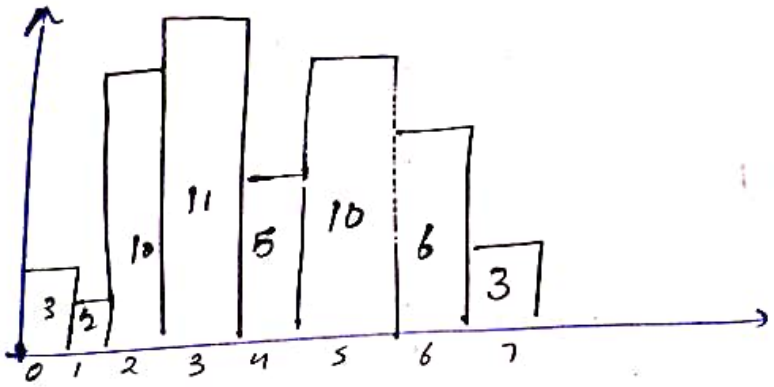
pse, nse

-1 n

```
fun(arr)
    nse = findNSE(arr)  ⟶ 2×n
    pse = findPse(arr)
    mom = 0
    for(0 ⟶ n-1){                    O(n)
        area = arr[i] × (nse(i) − pse(i) − 1)
        mom = Math.mom(area, mom);
    }
    return mom;
}
```

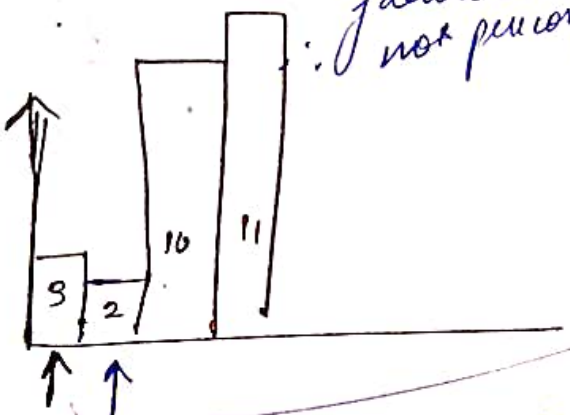
time - O(n)

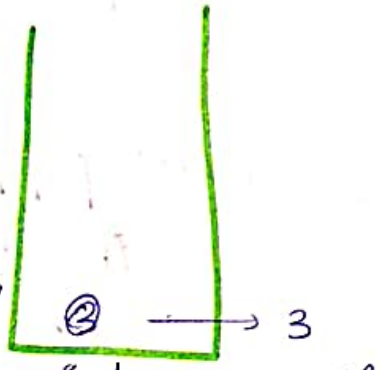
space → O(n)



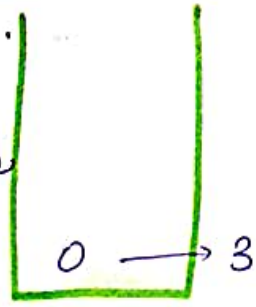
area[i] x (nse - pse - 1)

this can be computed while going forward not per computation





index val



will pop (3) as it is smaller

so, when item will be popped then we definitely know that popped nse element is current one "2" as it is smaller the top

popped pse is popped element below in the stack if empty then '-1'

∴ 3 nse → 1 ← current's indx

popped

3 pse → -1 ← as stack empty

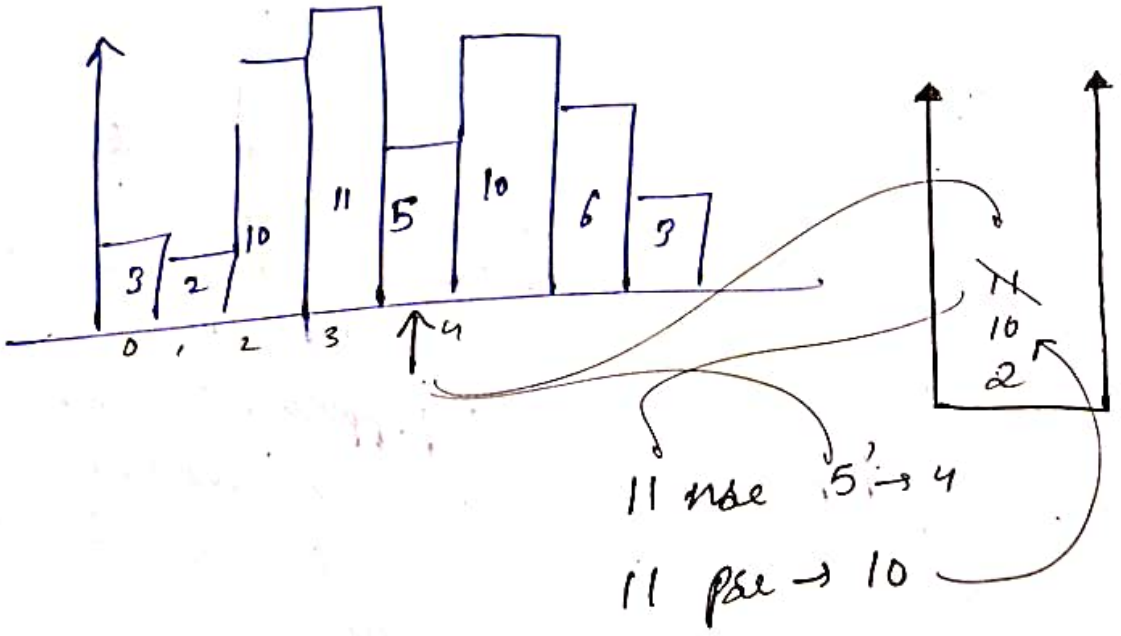
put in formula

area(i) × (nse − pse − 1) = area

3 × (1 − (−1) − 1) = area

3 × (1) = (3)

∴ while popping / coming back we know that the nse and pse ∴ compute only then



when element still in stack

then nse is "N" as no next smaller one

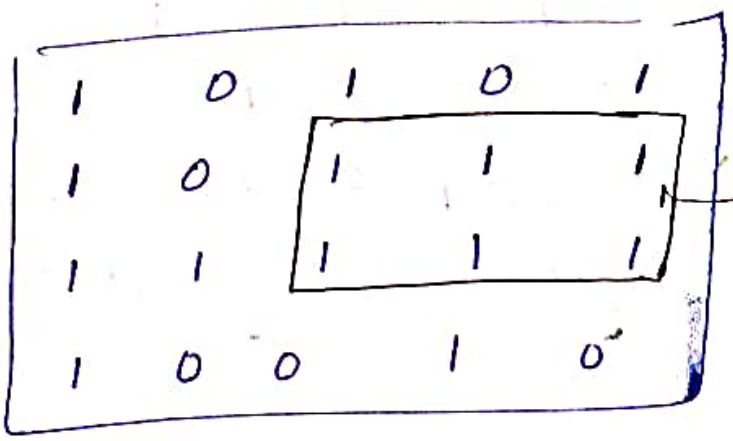
and as PSE is stack empty the "-1"

```
fun(arr){
    Stack st, maxArea = 0                    O(n)
    for(j=0 ⟶ n-1){                          inelem
        while(!st.empty() && arr[st.top] > arr[i]){
            element = st.top()
            st.pop()
            nse = i
            pse = st.empty ? -1 : st.top
            area = element x (nse - pse - 1)
            max = max(area, max)
        }                                    popping throughout
        st.push(i)                           O(n)
    }
    while(!st.empty()){
        nse = n
        pse = st.empty() ? -1 : st.top
        area
        and update
    }
return maxArea
}
```

time - O(2N)

space - O(N)

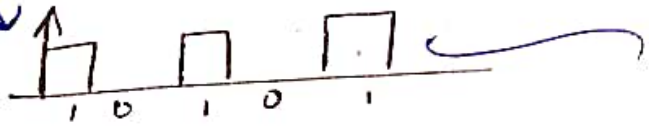
MAXIMAL RECTANGLE



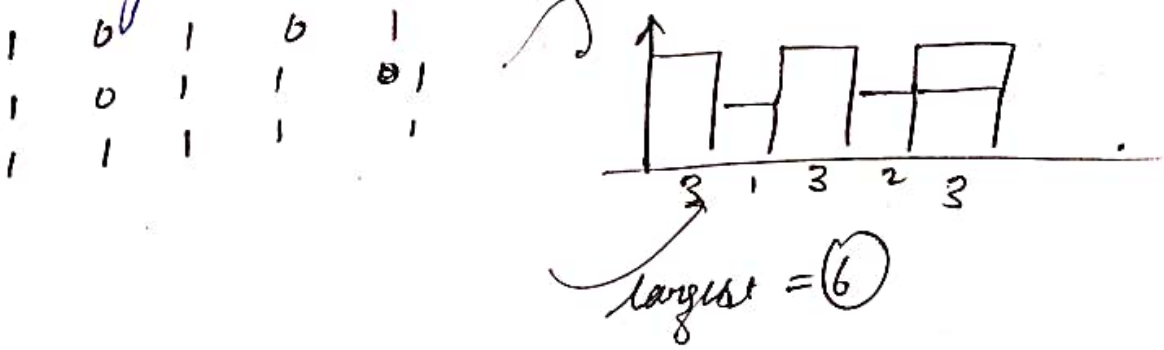
give area of max rectangle filled with 1s

using histogram approach

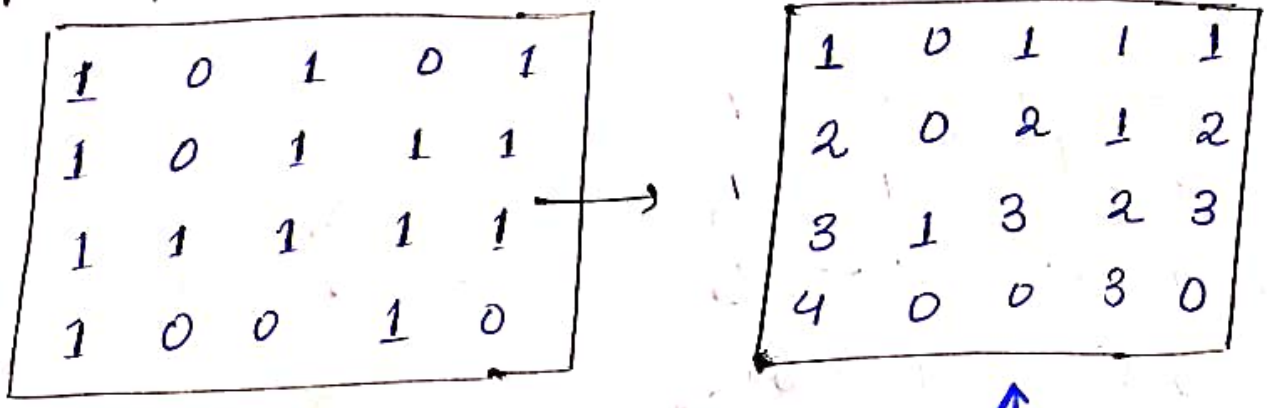
1 0 1 0 1 → 1st row



similarly do with other rows



precompute height



$O(n^2)$

```
fun(int(){){
    n, m,  psum[][]
    sum = 0
    fun(j = 0 ⟶ m-1){
```

histogram × O(m+n)

time - O(m×n) + O(n×2m)

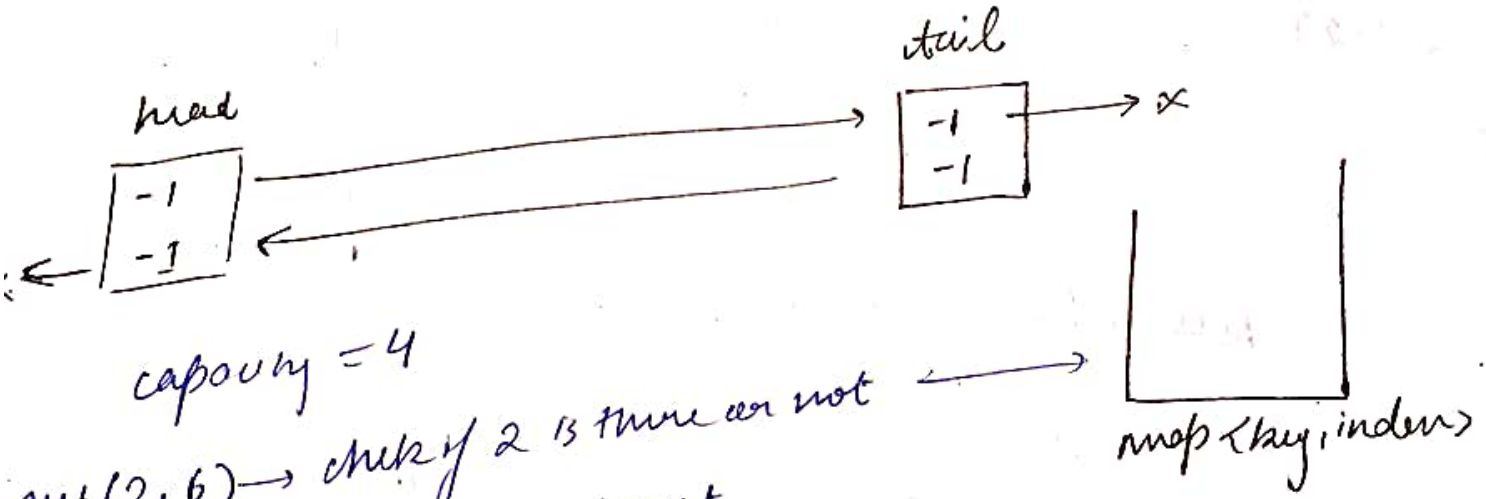
space - O(n²)

LRU cache

↑ last recently used

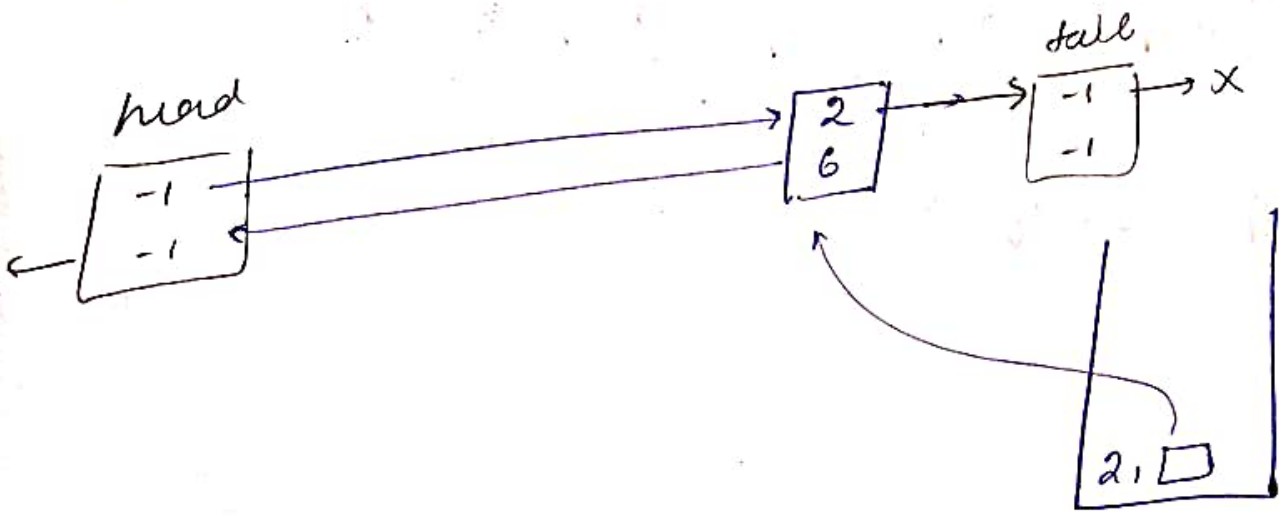
which takes capacity
when exceeds it takes out last recently used one
and get(int) which gives value in constant time by its key.

DLL and map data structures...

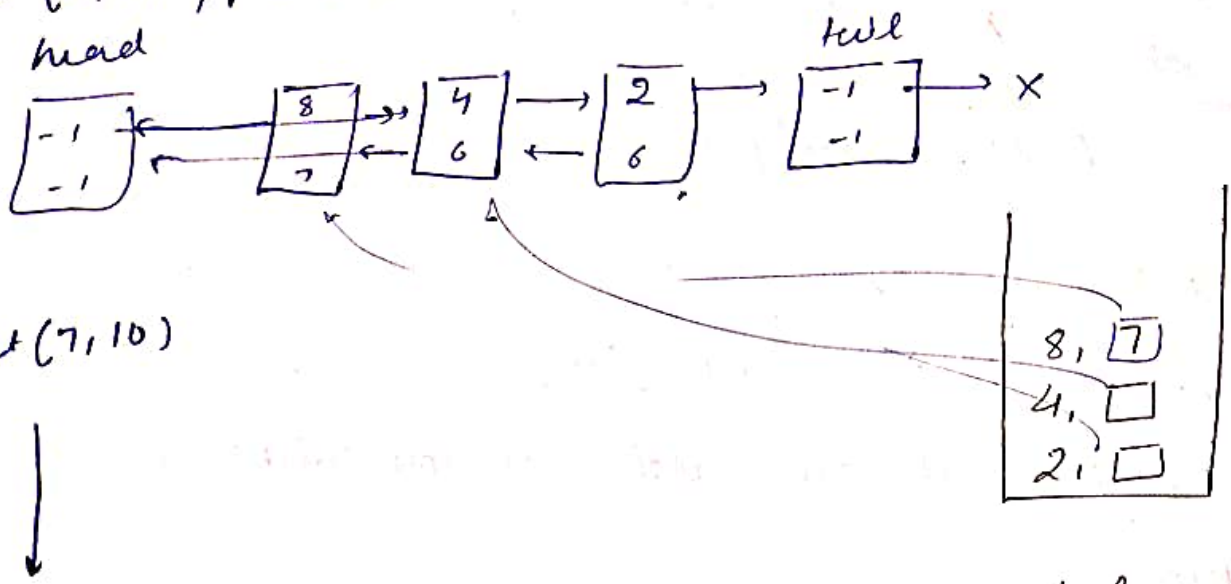


capacity = 4

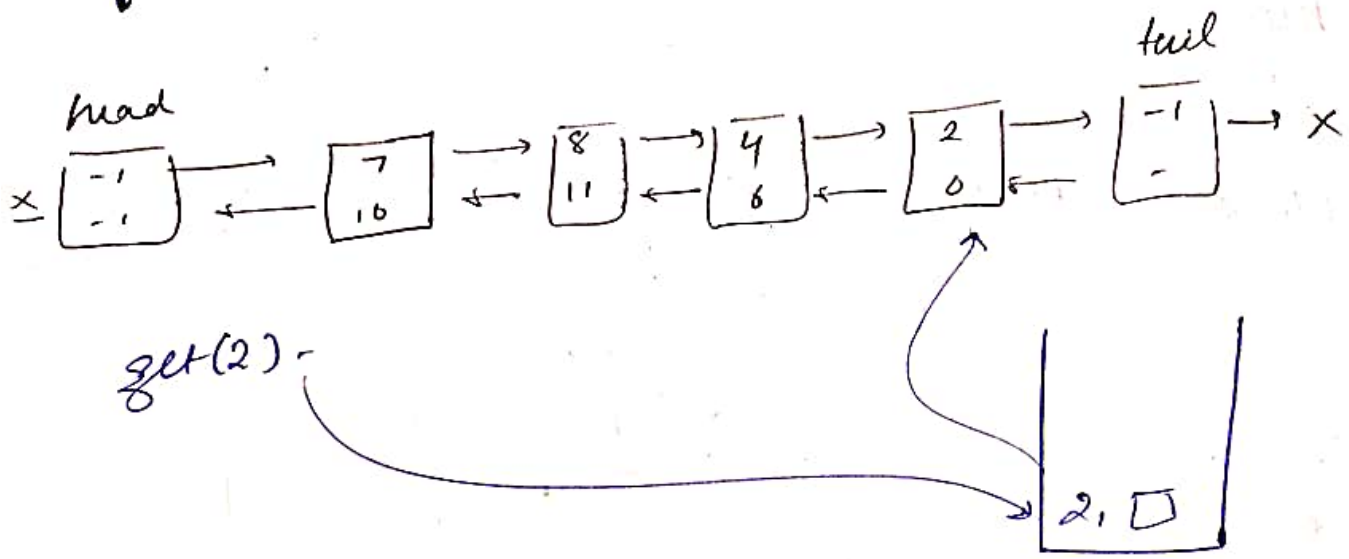
put(2, 6) → check 2 is there or not
key value ∴ not ∴ insert



put (4,6) , put (8,7)

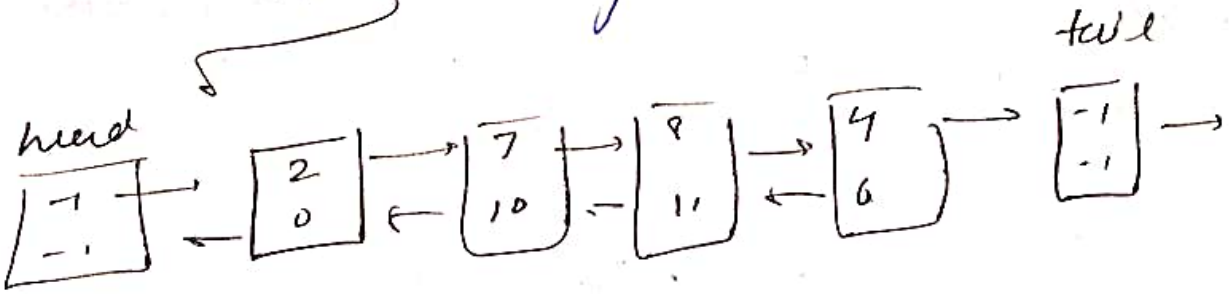


put (7,10)



get(2) -

now 2 has been used
∴, move to front



this is how it will works

```
insertAfterHead(node) {
    curHead = head->next
    head->next = node
    node->next = curr
    node->prev = head;
}
```

```
deleteNode(node) {
    prev = node->prev
    next = node->next
    prev->next = next
    next->prev = prev
}
```

```
class LRU cache {
    map<int, Node> mpp, capacity, head; tail

    (capacity) {
        this.capacity = capacity
        mpp.clear
        head → next = tail
        tail → prev = head
    }

int get(key) {
    if (!mpp.has(key)) return -1
    else { mpp[key] value; node
    }   deleteNode(node)
        insertNode(node)
        return node.value
    }

int put(key, value) {
    if (map.has(key)) {
            Node = mapp(key);
            Node.value = value;
            deleteNode(node)
            insertNode(node)
    } e
```

```
else {
    if(mpp.size == capacity) {
        deleteNode(
        Node = tail.pour
        mpp.delete(node.key);
        deleteNode(node)
    }
    Node newNode = new Node(key, val)
    mapp(key) = node
    insertNode(node)
}
```