TC => O(log n)

Stacks and Queues

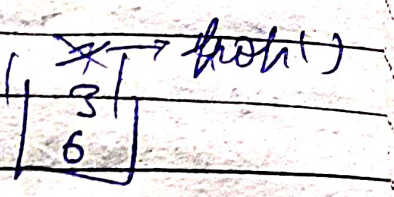
Stack → LIFO (Last in First out)

push(6) → push(3) → push(7)

top() = 7

pop(), top() = 3

pop() → removes as well as return top element.



Q74. Implement stack using array.

array → acting as stack. top → index of top most element

top = -1, no elements in stack.

~~stack~~

methods -

void push(i) ~~{~~ arr[++top] = i;

int pop() {
  if top == -1 return -1 // Stack underflow
  return arr[top--]; }

int top() {
  ~~return~~ if top == -1 return -1
  return arr[top]; }

stack overflow => adding elements when stack is already full

stack underflow => removing elements when stack is empty

Approach 2.
with 1 queue.

Date : Page :

→ push element to queue
→ remove n-1 elements from queue and push them again.

```
push(x) {
  q.push(x);
  for (i=0; i<q.size()-1; i++)
    q.push(q.top())
    q.pop();
}
```

```
pop()
{
return
return q.pop()
}
```

TC ⇒ O(N)
SC ⇒ O(1)

Q. 77. Implement Queue using stack
Approach 1.
take 2 stacks

push(x) TC → O(N)
① $S_1 \to S_2$ SC → O(2N)
② $x \to S_1$
③ $S_2 \to S_1$

Approach 2 (amortized)

push() → push elements in stack1
pop() → if stack2 is empty, transfer all elements in stack2 from stack1.
top() → same as pop.

SC ⇒ O(2N) TC ⇒

	stack1.empty()	stack1.empty()
push()	O(1)	O(1)
pop()	O(N)	O(1)
top()	O(N)	O(1)

Q78. Valid Parathesis (Balanced Brackets)
Brackets () {} []
→ each open bracket has close bracket

Approach -
→ when there is opening bracket, push it to stack.
→ when there is closing, check if stack is non empty, ~~if its em~~
→ if stack is empty, return false
→ if its not, pop the element from stack and check whether opening and closing are same
→ if they are not same, return false.
→ if they are, move to next element.
→ moment when all characters are checked and stack is empty, return true

TC ⇒ O(N) SC ⇒ O(N)

Q79. Next greater element.
→ find greater element ahead of the ~~rotated~~ circular array

Approach 1 Brute force approach
~~for i = (0 → n-1) {~~ TC ⇒ $O(N^2)$
~~for j = (i+1 → n-1)~~

Approach 2 Using stack. (LIFO)
→ if stack is empty, return -1
→ if stack.top() is less than checking no, pop()
→ else return stack.top() and push checking no. to stack
→ copy paste all the elements ahead of array
→ arr[i % n]

TC ⇒ O(2n + 2n)
SC → O(N)

Q80 Implement LRU cache (IMP)
→ Least Recently Used Cache
function
① get (key) = value ② put (key, value)
→ Remove LRU guy that inserted

size = 2
put (1, 1) {{1,1}}
put (2, 2) {{1,1}, {2,2}}
get (1) → 1.
put (3, 3) → {{1,1}, {3,3}}
get (2) → -1
put (4, 4) → {{4,4}, {3,3}}
get (1) → -1
get (3) → 3 Implement both in O(1)
get (4) → 4.

Data Structure ⇒ Hash Map and Doubly linked list
→ hashmap : key and address
→ DLL : key and value
→ recently used cache : just after head.
→ least recently used : just before tail

Q81. Implement LFU cache
(Least frequently used cache)
get(key) → get value of key, if doesn't exist : -1
put (key, value), in case size is full, remove
least frequently used cache and if
there are multiple possibility, remove least
recently used cache
2 maps ⇒ map<freq, list> freq list
map<key, node*> key node.

Q2. Largest Rectangle in Histogram.

→ given histogram

→ ~~find the area of largest rectangle in histogram~~

Date: Page:

Approach 1. Brute force (Naive solution)

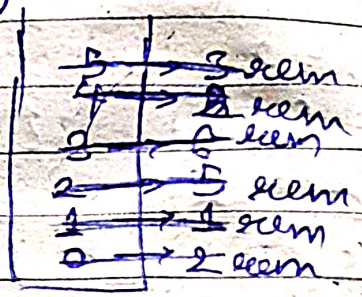
→ try every possible rectangle

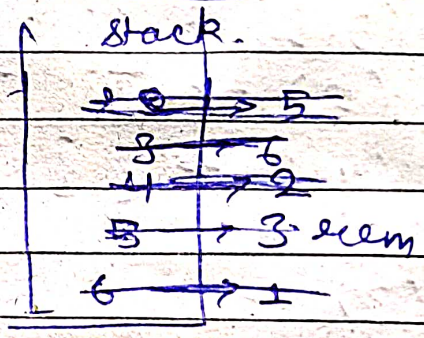
→ find smaller element to right as well as left.

TC → $O(N^2)$

Approach 2. (2 pass)

	0	1	2	3	4	5	6
left small	0	0	2	3	2	5	0
right small	0	6	3	3	5	5	6

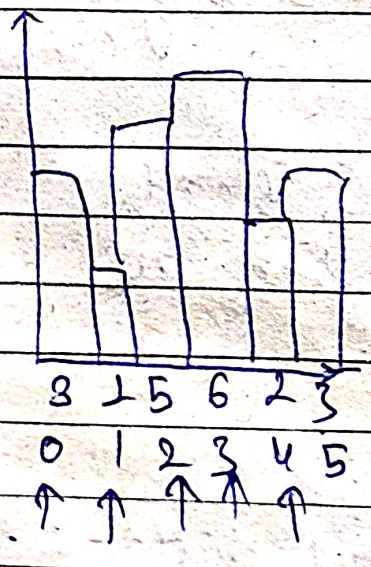




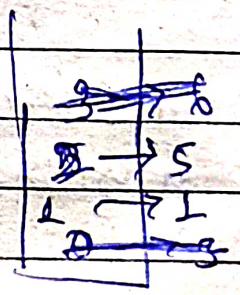
TC → $O(N) + O(N)$

$O(N) + O(N) \approx O(N)$

Approach 3. (1 pass solution)



max = ~~0~~ 3 6



→ maintain increasing st. fashion

→ find right smaller along with left smaller and also area.

TC → $O(N) + O(N)$

SC → $O(N)$

code

```
int largestRectangleArea (vector< int>& histo) {
    stack< int> st;
    int maxA = 0;
    int n = histo.size();
    for (int i=0 ; i<= n ; i++) {
        while (! st.empty() && (i==n || histo[st.top()]
                                >= histo[i]))
        {
        int height = histo[st.top()];
        st.pop();
        int width;
        if (st.empty()) width = i;
        else width = i - st.top() - 1;
        maxA = max(maxA, width * height);
        }
        st.push(i);
    }
    return maxA;
}
```

Q83. Sliding window maximum.

Approach 1 Naive solution

→ Brute force

→ O(N*K)

→ try each and every ~~sotu~~ subarray

Approach 2. ~~Next~~

use of doubly ended linked list

→ greater element will be at the front of doubly ended queue, and elements smaller than that will be placed at the back

Date: Page:

→ we first need to make sure if the element in front is within limits of our bounds, if it is not, then it is removed.
→ we will be removing all elements at the back of deque that are smaller than the element that we are placing and checking currently so as to maintain decreasing fashion.
→ the element will be pushed from back
→ the front element will only be pushed to result array iff $i > k-1$

Q84. Implement min stack.
approach → O(1)
use of 2 stack.
→ one stack is normal, another stack contains numbers in increasing order from top
→ push(x):
s1.push(x), if (x == getMin()) s2.push(x)

→ pop()
~~s1.pop()~~, if (~~get~~ s1.front() == getMin) s2.pop();
s1.pop()

→ getMin()
return s2.top()

→ top()
return s1.top.

Complexity of all op:
TC → O(1)
SC → O(N)

Date: Page:

Q85 Rotten orange.

→ given m × n oranges

0 = empty 1 = fresh orange 2 = rotten orange

→ if any rotten orange in adjacent to fresh orange, fresh becomes rotten

find minimum minutes needed to rotten all oranges

if impossible, return -1

Approach.. BFS (Breadth first search.)

non empty cell = 8.

→ queue data structure, save (i, j)

→ push all the rotten orange coordinate in queue

→ at each iteration, rot all oranges adjacent to orange in queue

→ perform steps untill queue is not empty.

→ also count total rotten oranges in queue.

TC → O(N*N) ×4

SC → O(N*N)

Q86 Stock Span Problem

→ given series of n daily quotes for a stock.

→ calculate span of stock's price for all n days.

→ use of stack containing 2 integers in each element

→ one represents prices and another one represents the stock span on that particular day

TC → amortized O(1)

SC → O(N)

→ remove all the elements from stack where prices are smaller and increment count of ans by i

→ push the results & price to stack.

Date : Page :

Q87 Find maximum of minimums for every window size in given array

→ given array a[n]

→ find maximum of minimums for every window size.

eg. arr[] = [10, 20, 30, 50, 70, 30]

window size = 1.

windows = [10] [20] [30] [50] [70] [30]

minimum = [10] [20] [30] [50] [70] [30]

max = 70.

window size = 2

windows = [10, 20] [20, 30] [30, 50] [50, 70] [70, 30]

minimum = [10] [20] [30] [50] [30]

max = 50.

window size = 3

windows = [10, 20, 30] [20, 30, 50] [30, 50, 70] [50, 70, 30]

minimum = 10 20 30 30

max = 30

and so on :

Approach 1. Naive solution (Brute force)

→ generate each and every possible window and find minimum of all of them

→ then find maximum in each case

TC = $O(N^3)$ SC → $O(1)$

Date Page

Approach 2.

→ Find previous small and next smaller element and save the results in each case.

~~for i~~

→ now we will find if any particular element tends to be solution or not.

how? find len = right[i] - left[i] - 1;

where i determines index of current element and then update ans[len] with max(ans[len], a[i]);

What about the left over elements?

ans[i] = max(ans[i], ans[i+1]);

TC => O(n) SC => O(n).