Stack

LIFO → Last In First Out

Mar-12

Duplicate brackets

TC : O(n)
SC : O(n)

(a+b + (c+d)) → false

(a+b)+((c+d)) → True

If in between ch count > 0, then ignore keep moving
↓
between '(' & ')'

If in btwn ch == 0 b/w '(' & ')' then true

Balanced brackets

TC O(n)
SC O(n)

Search counterpart char immediately, if present pop, else push the char itself.

* If stack is empty at last, → true (Balanced)
else → false (Not balanced)

Next greater element on right

* Brute force : TC $O(n^2)$ (two loops)
SC $O(1)$

* TC O(n) : Use stack → Travel Left → Right
SC O(n)

* If array taken then ordered
* If array not taken, then unordered

why O(n) TC, because every element is visited (push & pop) atmost twice

∴ TC ⇒ O(2n)
⇒ O(n)

* Ordered

→ Travel Right → Left

curEle >= stack top
pop;

curEle < stack top ⇒ this is the next greatest ele for curEle

Stock span

* Travel Left to right
* Consider -1 as initial value (in graph as a wall at left most side)
* while curr ele >= stack top => (pop all smaller elements)

~~Else Else~~ → find diff. (span), [curr day − ~~stack top~~ prev greater stock price day]

TC : O(n)
SC : O(n)

(If you look closely, this question is like finding next greater element on the left.)

Mar-13

Largest Area Histogram

TC O(n)

* Make use of two arrays, Left min arr & Right min arr
* Next smaller on right
* Next smaller on left

width = nextsmallerR[i] − nextsmallerL[i] − 1

⇒ width = nSR[i] − nSL[i] − 1

ht ⇒ arr[i]

area = width * ht

In code, travelling is done left to right.

Recommended: Try doing travel Right to left.

Travelling can be done from any side, focus on logic. Key is to pop larger elements

You can also save arr space, check tushar ray solution

Sliding window maximum 'k' size

Approaches: i) Using Stack TC $O(n)$ SC $O(n)$
ii) Using Array Deque TC $O(n)$ SC $O(n)$

(V) $\boxed{\text{Total no. of windows} \Rightarrow n-k+1}$

App1: Using Stack

* FIN Next greater element in arr with index as value.
* Use two pointers i, j

$\boxed{\begin{array}{l} i \rightarrow \text{starting point of windows} \\ j \rightarrow \text{for any} \end{array}}$

~~while j rem~~

* Travel each window
  - ↳ ~~cotin~~ At each window
    - ↳ while 'j' next greater element (NGR) lies within window range, update 'j' to that NGR.

[NGR → Next Greater on Right]

* If out of range then the 'j' value is the greatest (max) of the ~~cotind~~ current window.

```
//Code
for(i=0; i<=n-k; i++){
    if(j<i){
        j=i;
    }
    while(j remains in window range){
        j = ngr[i]; //update
    }
    ans[i] = arr[j];
}
```

(V) * Edge condition: If 'j' is left behind 'i' (i.e. j<i) then make j to i (j=i), to consider next window, moving towards

App2 : Queue (ArrayDeque)

* Add curr idx elements in AD.
* Pop all smaller elements ⇒ [curr ele > = AD.getLast()] // from last
* Ans. lies at peek of AD.

* If peek element doesn't lies in range of window then remove it. (This step must be done first in code.)

$\boxed{i \rightarrow \text{represents ending point of curr. window}}$

$\boxed{AD.peek() > i-k}$ // Only one element may be out of range at max.

(Then continue) (Else remove peek)

In stack method, $i \rightarrow$ ~~ending~~ starting pt. of window

In Deque method, $i \rightarrow$ ~~starting~~ ending pt. of window

Infix Evaluation $\boxed{TC\ O(n) \quad SC\ O(n)}$

Precedence, $(*, /) > (+, -)$

* Take two stacks of operands & operators
* If st.peek() (operator) precedence >= curr. operator, perform operation first & then move forward.

pop operators, have stack top() precedence > = curr operator precedence

Precedence order: '(' → 0

+, − → 1

*, / → 2

Infix conversions

Infix	Post-fix	Prefix
a+b	ab+	+ab
op1 + opr + op2	op1 + op2 + opr	opr + op1 + op2

TC O(n)
SC O(n)

Take 3 stack
- → 1 pre
- → 1 post
- → 1 operator

Smallest number following pattern (with stack)

Stack based:
TC O(n)
SC O(n)

* When ch == 'd' just push
* when ch == 'i' push, then pop all & add to ans string.

Refer codes for TC O(n) SC O(1) solution

Can also be done without stack, just use counter & reloop values SC O(1) TC O(n)
Refer self code for SC O(1). Refer video.

Celebrity problem

Approach 1:

arr[i][j] => 1 => i knows j

arr[i][j] => 0 => i doesn't know j

Two cases:

i know j => i can't be our celebrity

i doesn't know j => j can't be our celebrity

You get a potential candidate, now you need to check in row & col for conformation.

```
if (i != par_cand){
all ele of row of par_cand must be 0.
all ele of col of par_cand must be 1.
}
```

Approach 2: (Do check some cases)

i, j	j, i	
0	0	(Pop both) (both can't)
1	1	(pop both) (both can't)
0	1	i can be
1	0	j can be

0	0	0	0	0	1
1	1	1	0	1	1
2	1	1	0	1	1
3	1	0	0	0	0
4	1	1	1	1	0

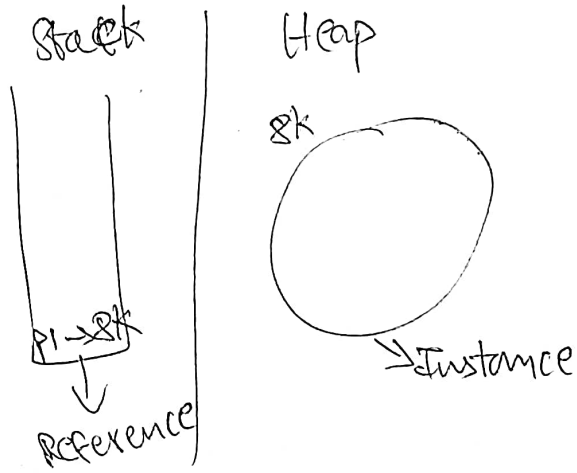
edge case

Here at end either you get 1 sure celebrity or if stack is empty at any point there there is no celebrity.

Even here you ~~get~~ may get only a doubt celebrity, edgecase. So you have to check for doubt celebrity in row & col.

Class → Encapsulation of data members with related functions.

Eg: class Person{
string name ; } → Data members
int age ;
public void sayHi( ){ }
}
}main( ){
Person p1 = new Person( );
}



Pass by value, pass by reference
↓ primitives ↓ non-primitive

1) Space allocation → Allocates space with respect to the data members.

2) Class parsing → overriding values of data members

3) Constructor firing

Constructor → 1) Function with name exactly similar to Class name

2) No-return type

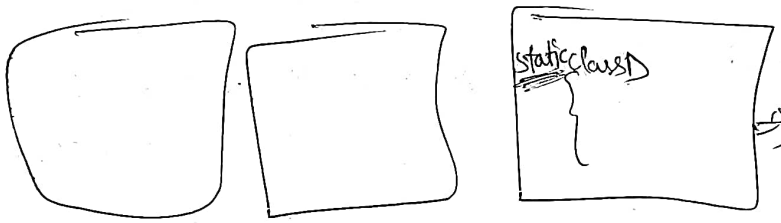
3) By default, It will return this.

4) Default constructors public ClassName() {

}

→ If you don't provide any constructor, java by default provides default constructor.

A.java B.java C.java



→ If you want to use class D with creating instance of C, then keep class D as static

---

Class name & file name must be same.

Normal stack : Use array with given capacity

Dynamic stack : Use array and grow array by creating new instance (double the capacity of previous array).

Normal Queue : ~~Use~~ Travel in circular way on array to use array space efficiently satisfying queue functionality

Dynamic Queue :

Use array and grow array creating new instance and copying previous array elements into new. (New array size to be doubled than previous, whenever array capacity is full)

Stack Using Queue - Push efficient

App 1 : Take two queue

- For push, add in q_1
- For pop, remove from q_1 until size != 1 and add in q_2
  - After this, the first element of q_1 is your top of stack.
  - Then swap q_1 & q_2

App 2 : One queue

- For push, just add to q
- For pop, do q.size() - 1 removals and add ~~push~~ to ~~q~~ ~~en~~ end of q.
  - Then 1st element of q is your stack top.

Stack Using Queue - Pop Efficient

Approach 1 : Two Queue →

1. Add element in q_2
2. Remove all elements from q_1 and add in q_2
3. Swap q_1 & q_2

Approach 2 : One Queue →

1. Add to last of q.
2. Remove while q size != 1 and add to last of q.
3. Peek of q ~~is~~ will be stack top.

Queue Using Stack - Add efficient

Approach : Two stacks

1. Push in stack 1 (val. of v to be added)
2. For pop, remove while st1.size() != 1 & push in st2.
3. Top of s1 is q.peek().
4. Then pop all from st2 & push in st1 (to maintain state)

You can't swap because direction of stack push, pop (LIFO) is different from Queue (FIFO)

Queue using stack - Remove Efficient

Approach : Two stacks

1. Remove all from st1 & push in st2.
2. Add element in st1 (val of q. to be added)
3. Removed (pop) all from st2 & push again over st1.
4. Top of st1 is q. peek()

(You can't swap here as well LIFO, FIFO ordering)

Also can't do with one stack (same LIFO, FIFO ordering)

Minimum Stack - 1

Approach 1: Two stack:

↳ ① Push st1 (val), if val >= st2.top

↳ push st2.top again in st2

else

↳ push val in st2

Minimum stack - Constant Space

Self-Approach 2: One stack ⇒ Use Pair class (val, min)

↳ Same logic as above, just use pair class in st. type

- See st.top min & compare with cur val

if (cur val < st.top.min)

↳ push Pair(val, val)

else → push Pair(val, st.top.min)

~~6, -3, 11, 7, 3, 8,~~

10, 8, 3, 7, 11, -3, 6

6, -3
-3, -3
11, 3
7, 3
3, 3
8, 8
10, 10

(Not exactly constant space, bcoz at last using two variables in Pair class so same complexity. Just different way)

~~Class Approach:~~

~~If cur.val <~~

Class Approach: MinStack (Constant Space)

* Take a variable, min.

While push, formula ⇒ $\boxed{n.v = org + org - pmin}$ — ①

$\boxed{min = org}$ — ②

$n.v = min + min - pmin$ Subs. ② in ①

$n.v = 2 * min - pmin$

While pop, formula ⇒ $\boxed{pmin = 2 * min - n.v}$ ⇒ $\boxed{min = 2 * min - tos}$

while push, ~~if val > min~~

↳ if $val^{org} < min \rightarrow nv = val^{org} + \underbrace{val^{org} - min}_{-ve\ value}$

↳ Ensures to give value less than min (fake value)

$\Rightarrow nv < min$ (will be)

↳ push nv (fake value)

↳ $\boxed{min = val^{org}}$

while pop,

if $tos < min \Rightarrow$ fake value

↳ ~~min =~~ ~~2 * min~~

$pmin \Rightarrow \boxed{min = 2 * min - tos}$

We alter min to smaller value, ~~because~~ to differentiate purpose & even other original value will be greater, thereby greater value of min will have no effect.

Two Stacks in an array:

↳ ① Take two variables, tos1, tos2 to maintain stacks ordering st1, st2 in array.

→ ② Fill st1 using tos1 from left

→ ③ Fill st2 using tos2 from right

Both stacks maintain tos in an array

Smallest Number following pattern

TC	O(n)
SC	O(n)

App 1 : with stack

- if ch == 'd', push (increment)
- if ch == 'I', push (increment) then pop all & add to ans.

At end pop all from stack & ~~push~~ add to ans (Edge case).

App 2 : without stack

Refer self code of O(1) SC, approach (Easier than GFG code)

Pseudocode :

TC	O(n)
SC	O(1)

```
String ans = "";
int i = 0;
int c = 1;
int d = 0; // To keep track of next 'd' characters
while (i < str.length){
    char ch = str.charAt(i);
    if(i==0 & ch=='I') //Edge case
        ans += c;
    if (ch=='d') d++; // Edge case
    int j = i+1;
    while (j < str.len & str.charAt(j) == 'd'){
        d++;
        j++; // count next consecutive 'd'
    }
    int k = d;
    while (d>=0) {
        ans += (c+d); // Forming answer
        d--;
    }
    d = 0;
    c += (k+1); // Tracks next curr value
    i = j;
}
```