Agenda

What are stacks?

Operations on stack

Implementation

Balanced Paranthesis

Double Char Trouble

Postfix Expression

} Ques

Recordings (Archive)

1. Sorting Follow Up (Inversion Count)

Java / Python Refreshes

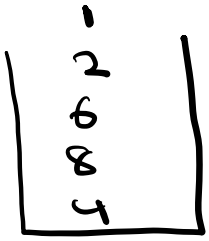
2. Binary Search (Nth Magical No.)

Avg Intermediate

3. LL (Clone LL)

All batches

Stack



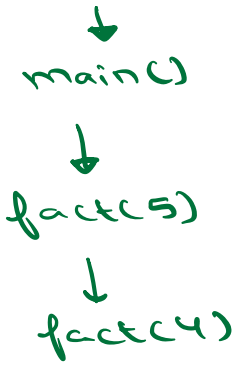
- Stack is a linear data structure that stores information in a sequence, from bottom to top.
- The data items can be accessed from top and new elements can only be added at top i.e. it follows LIFO (Last In First out) principle

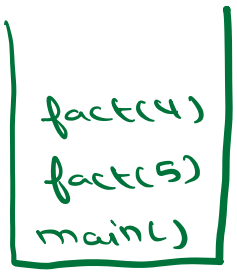
Examples

1. Pile of plates

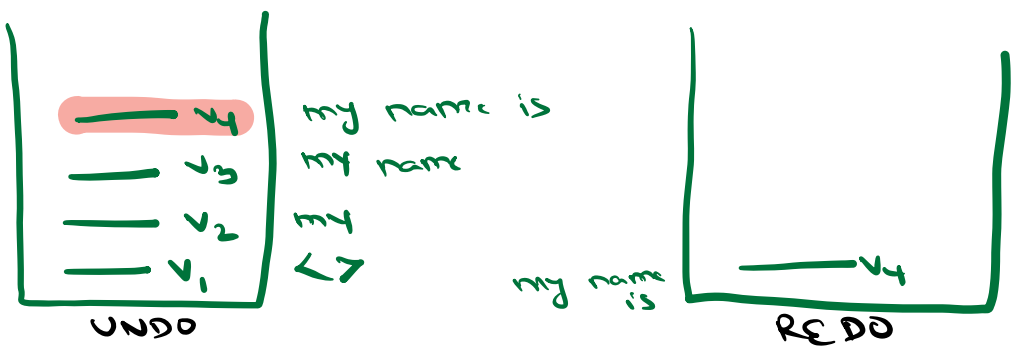
2. Stack of chairs

1. Recursion





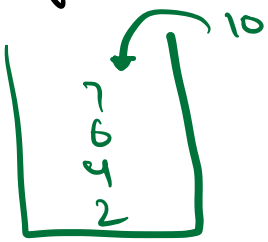
2. Undo Redo



my name is —undo→ my name —redo→ my name is

Operations on stack (TC is O(1))

1. push (data) → insert a new element added at top of stack



2. pop() → remove element from top of stack

int / void

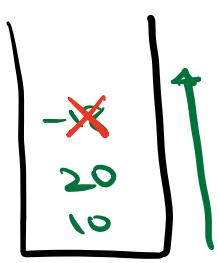
3. peek() / top() → return topmost element of stack, does not delete it

data = peek()

4. isEmpty() → checks where stack is empty or not

True → Empty
False → Non empty

```
Stack st = new stack()
st.push(10)
st.push(20)
st.push(-18)
int x = st.peek()
print(x)        // -18
```

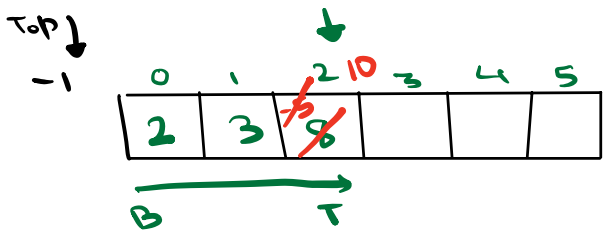


st.pop() -18 deleted

int y = st.peek()

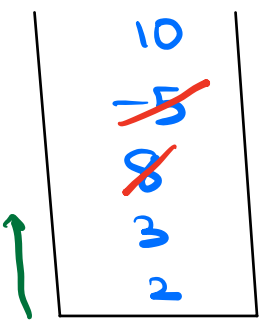
print(y) // 20

Implement Stack using Array



Arr → 6

	Opr		Top = -1
push(2)	top++	a[top] = 2	0
push(3)	top++	a[top] = 3	1
push(8)	top++	a[top] = 8	2
pop()	del 8	top--	1
push(-5)	top++	a[top] = -5	2
peek() / top()	-5 = a[top]		
pop()	del -5	top--	1
push(10)	top++	a[top] = 10	2



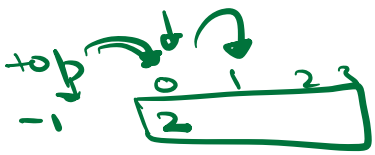
Top initially -1 → invalid idx → stack empty

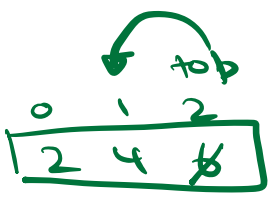
Bottom ⟶ Top

0 ⟶ top idx

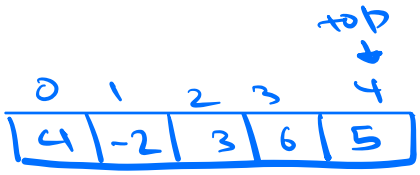
Stack of integers

```
class Stack {
    int arr[10]
    int top

    Stack() {
        top = -1
    }

    push(int x) {
        top++
        arr[top] = x
    }

    pop() {
        top--
    }

    int peek() {
        return arr[top]
    }

    bool isEmpty() {
        if (top == -1)
            return true
        else
            return false
    }
}
```





```
Stack st = new stack()
st.push(_)
```

① 

Overflow: When we try to store new data even when there is no empty space in array

```
push(int x) {
    if (top == arr.size() - 1) return
    top++
    arr[top] = x
}
```

②

Underflow: When we try to perform pop operation, or try to access stack element but stack is empty.

```
void pop() {
    if (isEmpty()) return
    top--
}
```

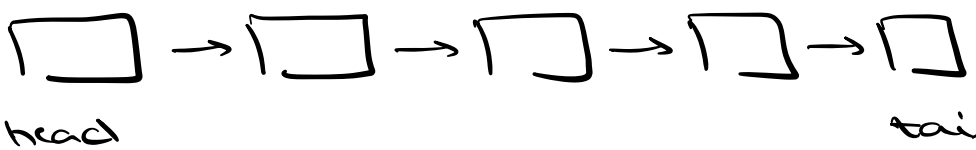
```
int peek() {
    if (isEmpty()) return -1
    return arr[top]
}
```

Array has fixed size

↓

Use dynamic array

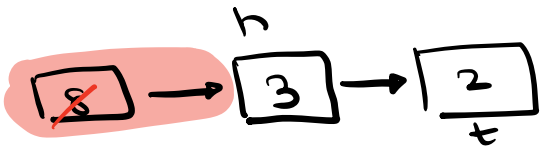
Implement stack using singly LL



- ✓ addHead() O(1)
- ✓ deleteHead() O(1)

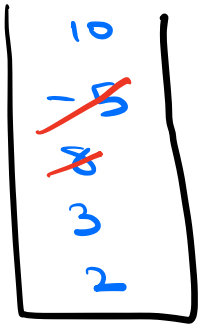
O(1) addTail()

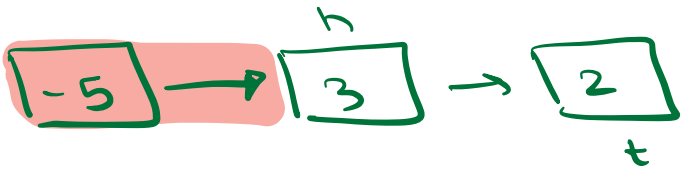
O(N) deleteTail()



h = NULL t = NULL

push(2)	add 2 at head
push (3)	add 3 at head
push (8)	add 8 at head
pop()	del 8 / del head
push (-5)	add -5 at head
peek() / top()	-5 (head.data)
pop()	del -5
push (10)	add 10 at head






```
Node  h = NULL, t = NULL
  int   size = 0
void  push (int  x) {
    Node  nn  = new  Node (x)
      nn. next = head
      head = nn
      size ++
```

```
void pop() {
    if (isEmpty()) return
    head = head.next
    size --
}

int peek() {
    if (isEmpty()) return -1
    return head.data
}

bool isEmpty() {
    if (head == NULL) or size == 0
                return true
    else
                return false
}
```

LL → only underflow happens

Don't need tail

10:38

1. Check given sequence of paranthesis is valid.

I/P → <>, (), []

eg. ( ) [< > ( )] True

( ) [< (>)] False

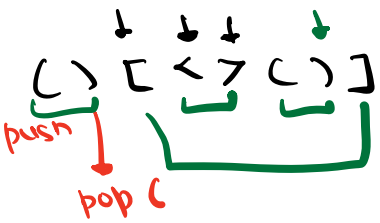
(< > [ ] False

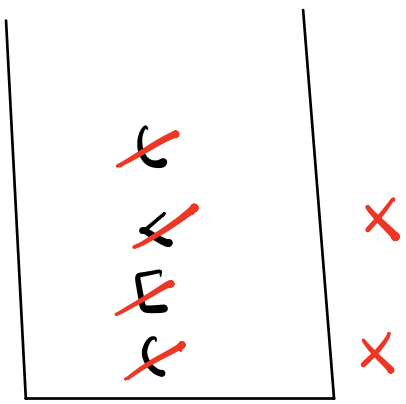
) ([ ] False

(< [ ] >) True

① Every opening bracket will have a closing bracket of same kind and vice versa

② A closing bracket should've a recent opening bracket

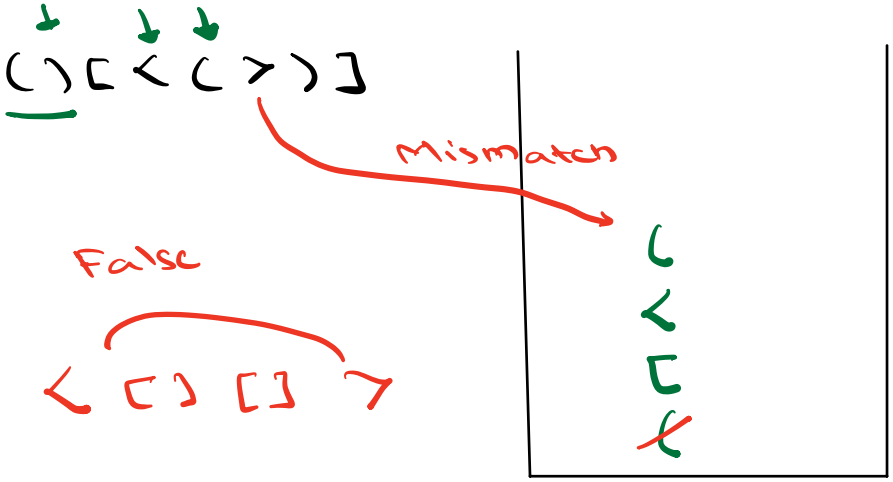




True

Opening bracket → pushed

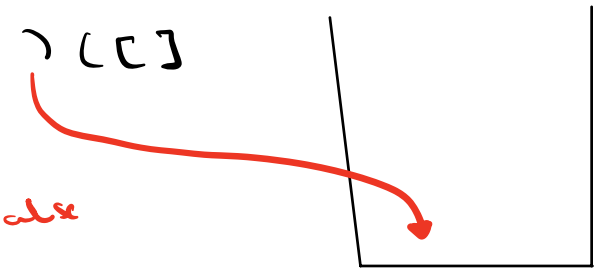
Closing bracket → delete its partner



False

< [ ] [ ] >

Opening bracket on stack should be of same type

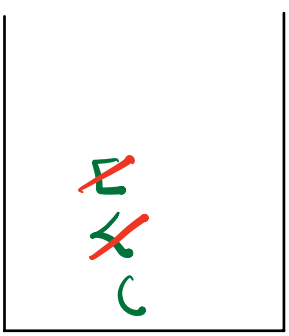


False

Closing bracket → stack is empty() → False

(< > [ ]

False



At end if stack is non empty → (did not find its partner → not valid

Travel the complete string <

if current char is open bracket
push in stack

else <), >,]

t → check char at top of stack

if (t is same as current char)
pop top char

else
return false

if (stack is empty)
return true

else
false

cur	top
)	(
>	<
]	[

TC : O(N)

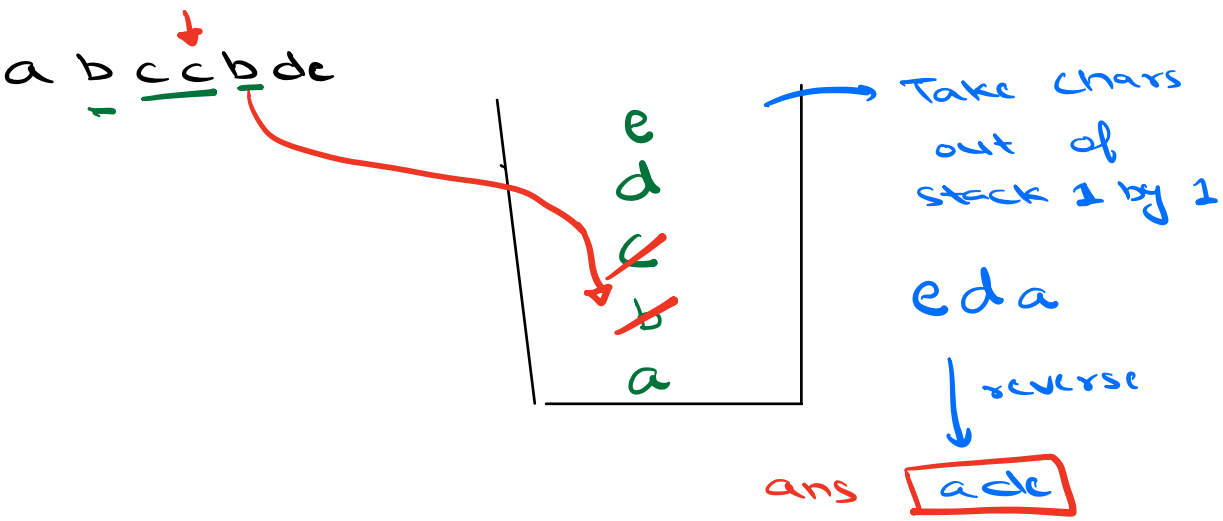
SC : O(N)

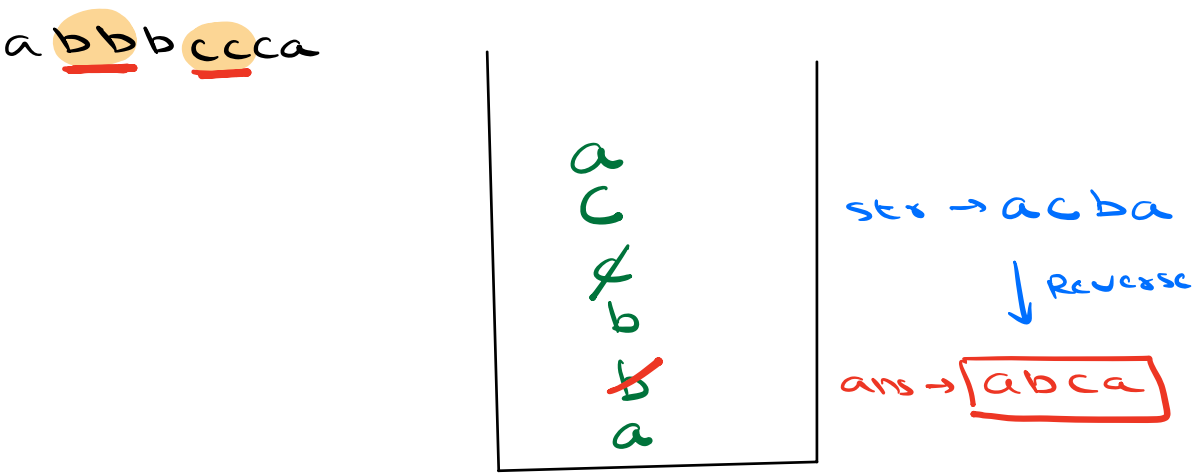
2. Given a string, remove equal pair of consecutive elements till possible.

abbc → ac

abccbde → abbde → ade

abbbccca → abca





abbcbbcacx → accacx → aacx

↓

cx

For each char {

if char matches stack top

pop

else

push (char)

}

Take everything out of stack (append in str)

↓

Reverse it → final string

TC: O(N)

SC: O(N)

```
stack <char> st
for (i=0 ; i < n ; i++) {
    cur = str[i]
    if (!st.isEmpty() &&
        cur == st.peek())
        st.pop()
    else
        st.push(cur)
}
string ans = ""
while (!st.isEmpty()) {
    ans += st.peek()
    st.pop()
}
```

reverse (ans)

---

3. Postfix Expression

Infix Expression

2 + 3

operand operator operand

↓

+
-
/
*

Postfix

2 3 +

op1 op2 operator

+
-
/
*

a + b

(a * b) - c

a b +

a b * c -

eg. a + b - c * (d + e)

↓

a + b - c * de+

a + b - cde+ *

↓

ab+ - cde+ *

ab+ cde+* -

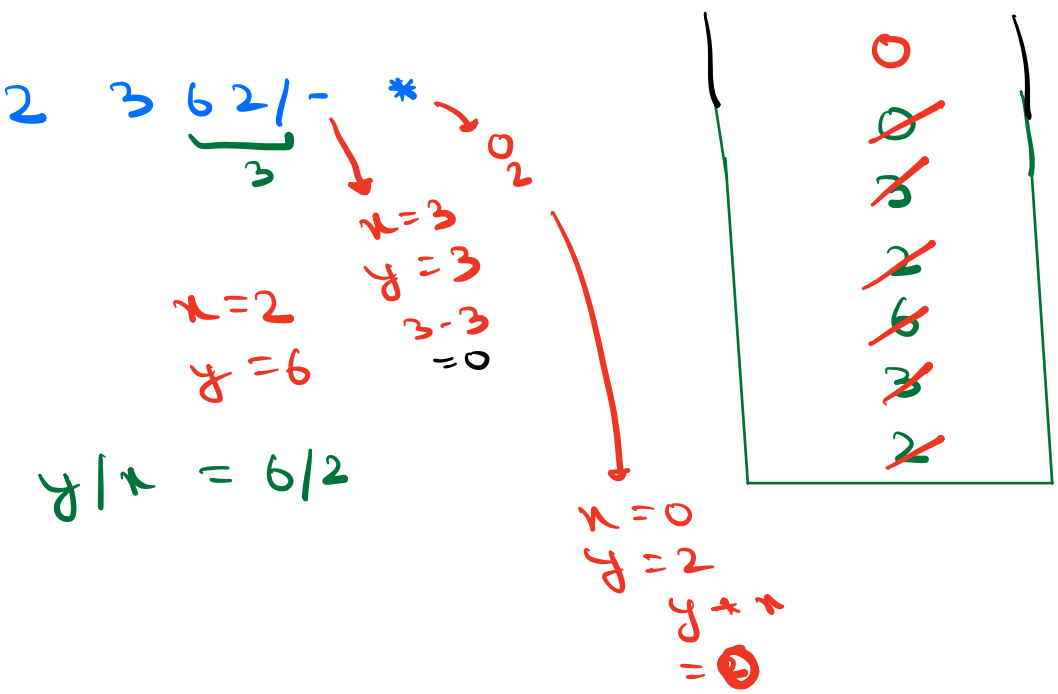
eg. $2 * (3 - (6/2))$ → $2 * (3 - 3)$ → $2 * 0$ → 0

$2 * (3 - 6\ 2/)$

$2 * 3\ 6\ 2/ -$

$2\ 3\ 6\ 2/ -\ *$

Evaluate postfix expression



top of stack = 0

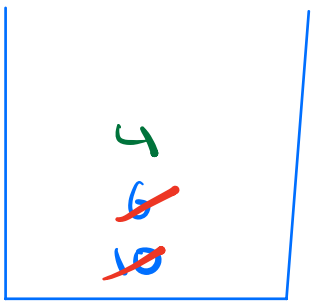
Infix

10 − 6

Postfix

10 6 -

ans is top of stack → 4

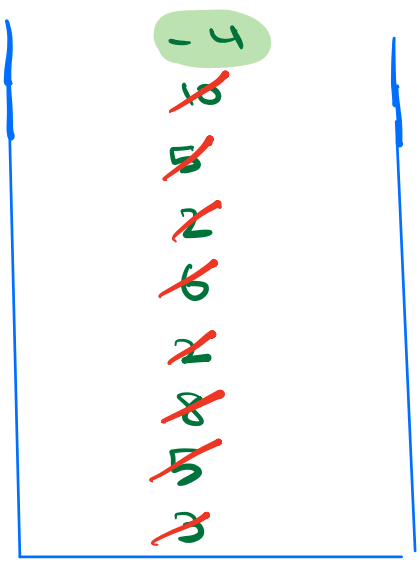


x = 6
y = 10
y − x
= 10 − 6 = 4

eg. 3 5 + 2 − 2 5 * −

① x = 5
y = 3
y + x
5 + 3 = 8

② x = 2
y = 8
y − x = 6



③ x = 5
y = 2
y + x = 10

④ x = 10
y = 6
y − x
6 − 10 = −4

top of str → −4

Traverse expression characters one by one {

if (char is operand)

push (operand)

else {

op2 = pop()

op1 = pop()

push (op1 operator op2)

(y operator x)

}

}

Final ans → stack top
