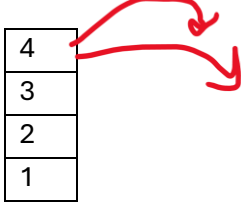


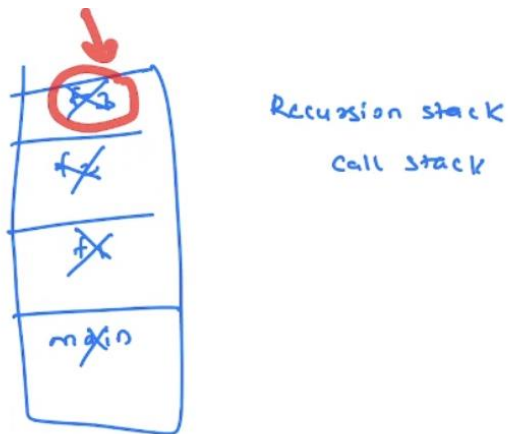
STACKS1: Implementation & Basic Problems

Stack:

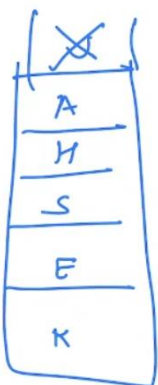
A linear data structure that stores information in a sequence LIFO (Last in First Out)



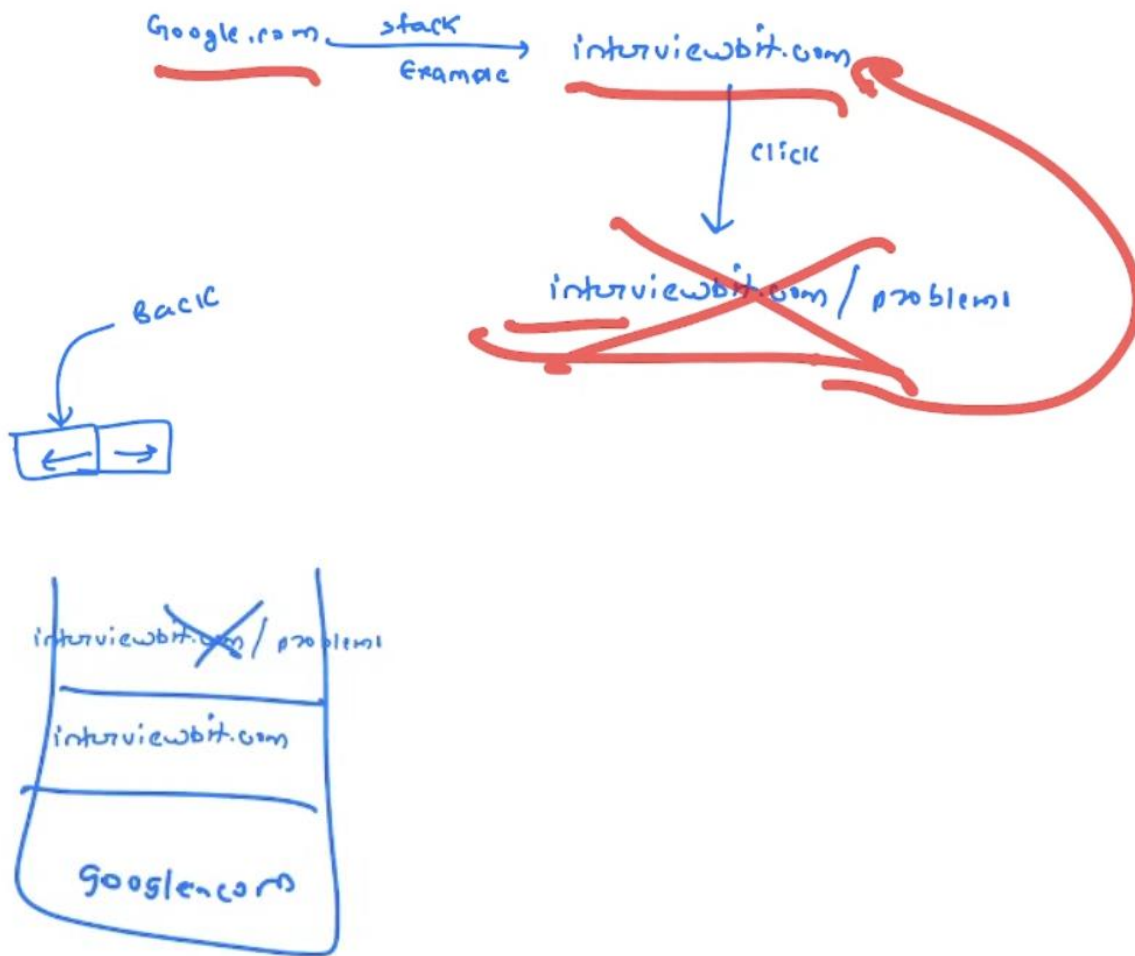
Example -1: Recursion Stack



Example-2: Undo/Redo Feature – Whenever you want to add element, it gets added and when you want to remove something, you can remove from the top.



Example-3: When you browse something in Google. Go to one site, and with the help of back button we can go back and these added in stack first, will remove first.



Operations on Stack

1. **Push(x):** Insert the given element x on top of the stack
2. **Pop:** Remove an element from the top
3. **Peek:** Returning the top element from the stack without modifying anything
4. **Isempty:** Gives True/False, if stack is empty
5. **Size():** Return the size of the stack. TC for Size is also $O(1)$ you will have to maintain the size in a variable whenever you add or remove an element, you don't have to iterate anything extra for calculating the size.

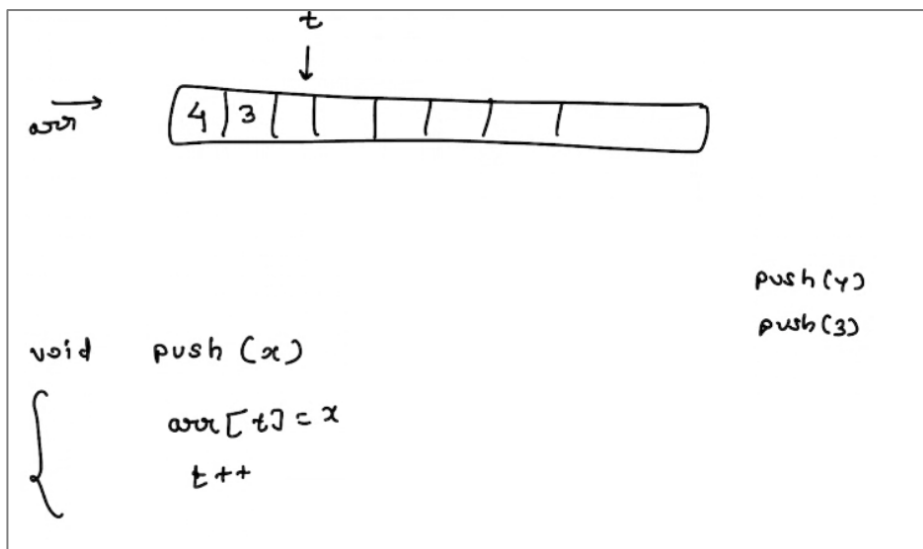
TC for above operations $O(1)$

Implement a stack using Arrays

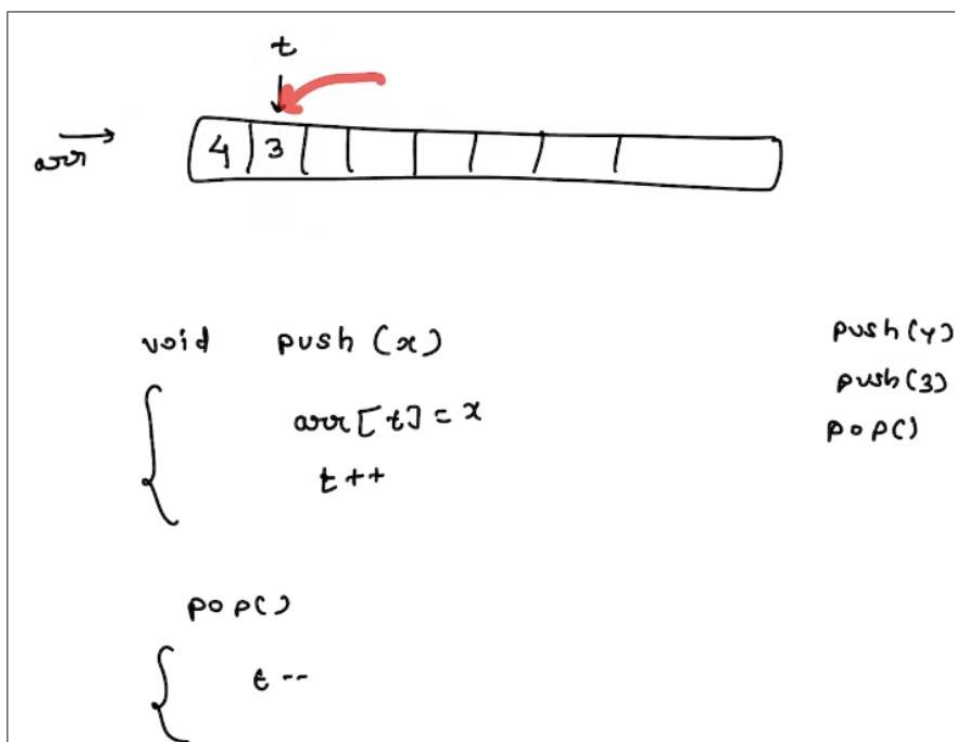
stack <int> st =

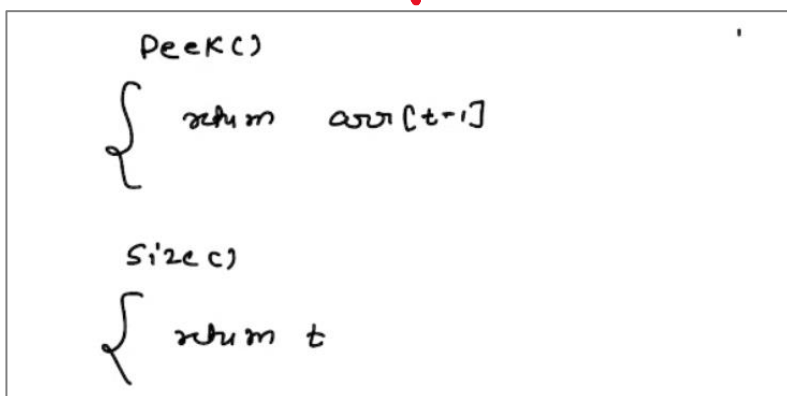
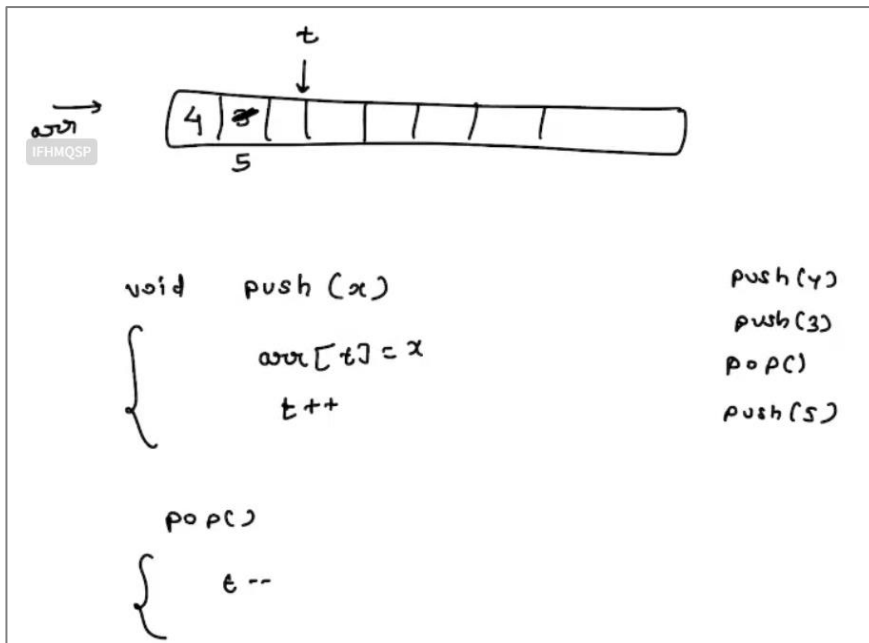
This t will denote where an element is added at the last

When ever you want to add an element you push



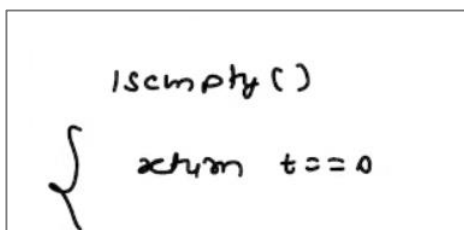
Now popping the element so 3 will be deleted, and t goes at 2nd position





For the size of stack, if t is pointing at 1st position, then we have 0 elements. Here t is representing where the next element should be added.

If we want to get the top element is, we will check $t-1$ position.



Edge Cases

```

pop()
{
    if (t == 0) return
    t--
}

```

If stack is empty and we keep using pop function → it will cause underflow

```

peek()
{
    if (t == 0) return null
    return arr[t-1]
}

```

If $t == 0$, and we use peek function, we might get NullPointerException error.

```

void push(x)
{
    if (t == N) return
    arr[t] = x
    t++
}

```

where N = capacity of an array

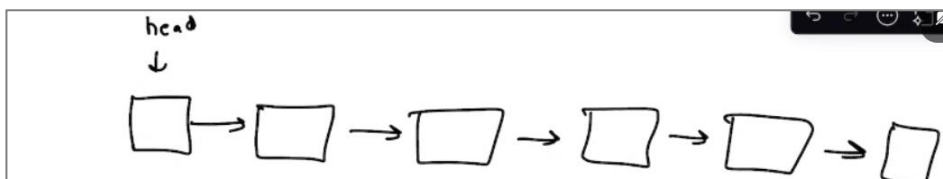
int $t = 0$, t is index where we can push an element into stack/array. So where t is pointing to, we will add any element.

Implementation Using Linked List

Linked List is a linear ds, it has two ends – head and tail.

You can push or pop in head or tail

TC for insertion at head



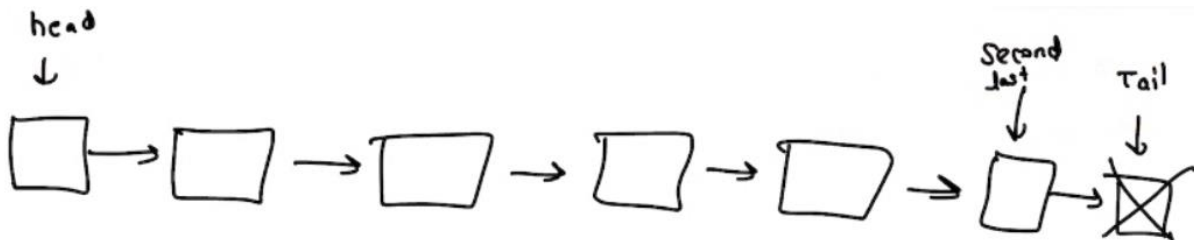
Addition element is backwards (add new node before head and assigning head to the new node x) and removing its front.

	Tail	Head
push	$O(N)$ $O(1)$	$O(1)$
pop	$O(N)$	$O(1)$

In stack, Insertion should be done at tail and deletion at head

Addition/Insertion at tail can be optimized $O(N)$ by $O(1)$ → If you have the reference for tail.

But for pop function, you cannot optimize because you need a reference for second last element also.



Whenever we asked to initialize a stack, you can initialize a head reference. Whenever you add an element, you can create a new node and push before head, and you can delete the head and point new node to head.next.

```

push(x)
    ↑
    tmp
{
    Node tmp = new Node(x)
    tmp.next = head
    head = tmp
    cnt++
}

pop()
{
    head = head.next
    cnt--
}

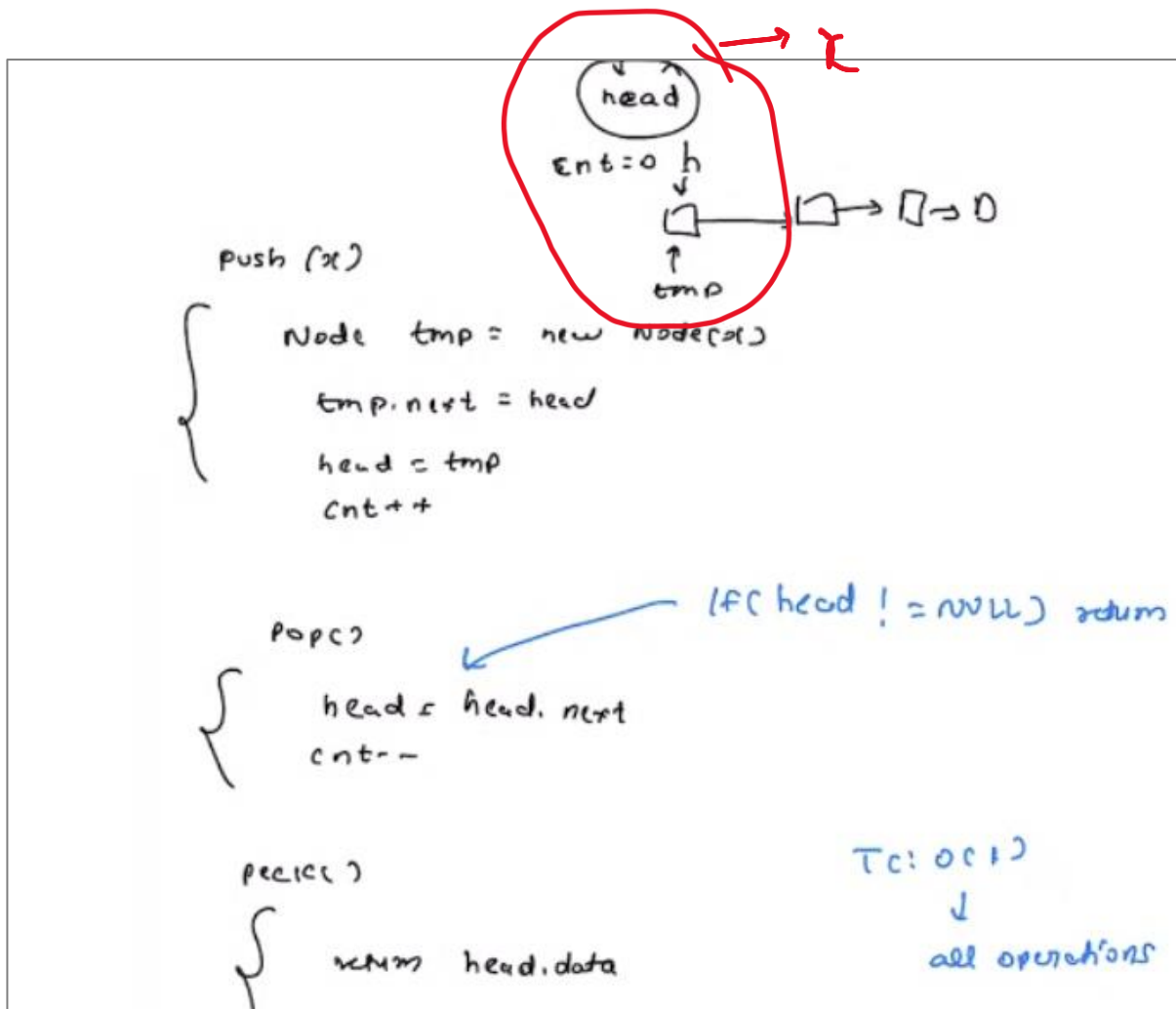
peek()
{
    return head.data
}

size()
{
    return cnt
}

isEmpty()
{
    return cnt == 0
}

```

With head, you can use a variable cnt, for getting the size.



1. Problem Statement – Parenthesis sequence

Given an expression string **A**, examine whether the pairs and the orders of “{”, “}”, “(”, “)”, “[”, “]” are correct in **A**.

} [] () a + - /

Example:

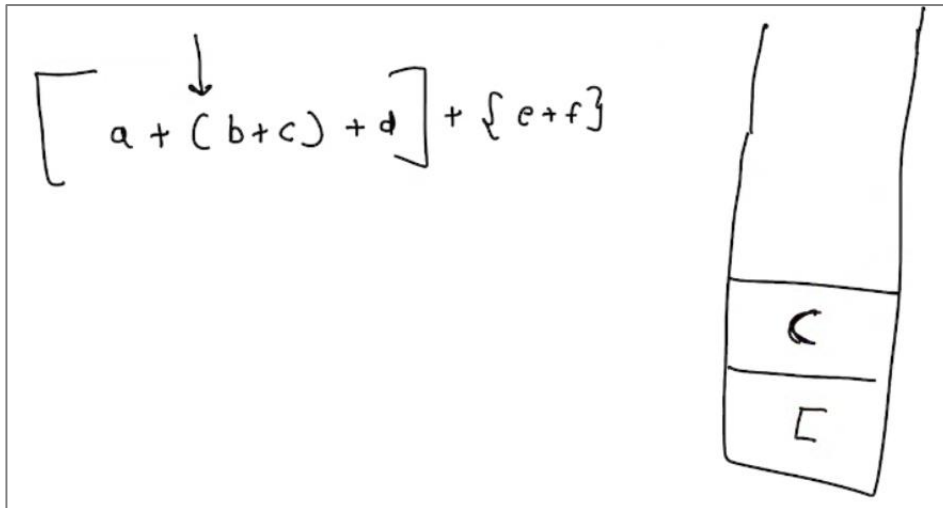
1. (a+b) o/p: valid parenthesis sequence
2. (a + b} o/p: not valid parenthesis
3. (a+b : not valid parenthesis
4. ((a +b: not valid parenthesis
5. [(a+b)]: valid parenthesis
6. [(a+b)]: not valid parenthesis

[a + (b + c) + d] + { e + f }

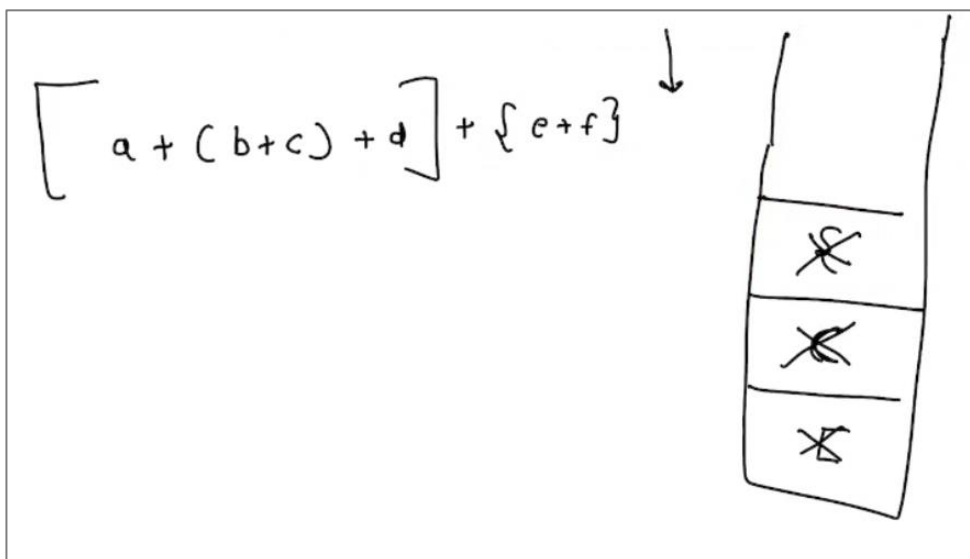
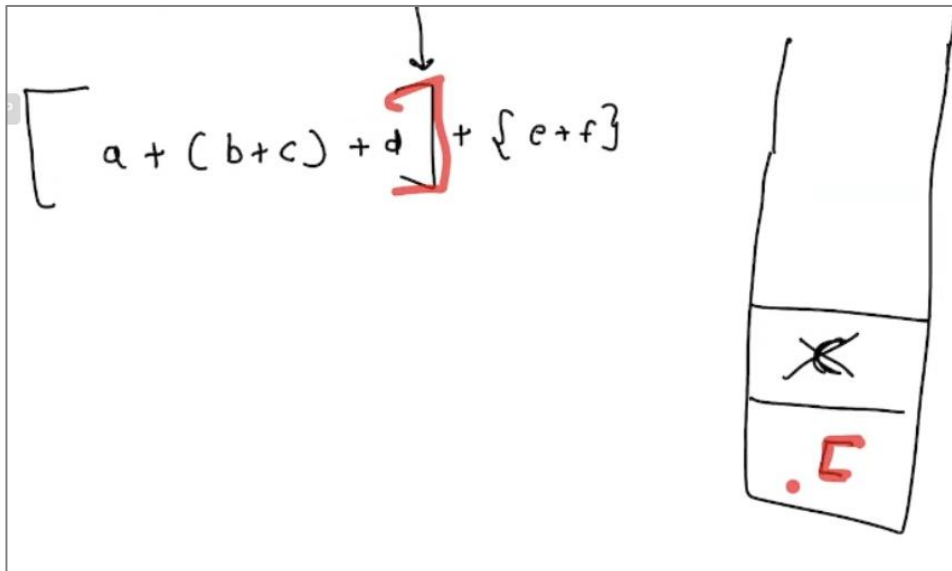
Problem Constraints: $1 \leq |A| \leq 100$

Input Format: The first and the only argument of input contains the string **A** having the parenthesis sequence.

Output Format: Return 0 if the parenthesis sequence is not balanced.
Return 1 if the parenthesis sequence is balanced.

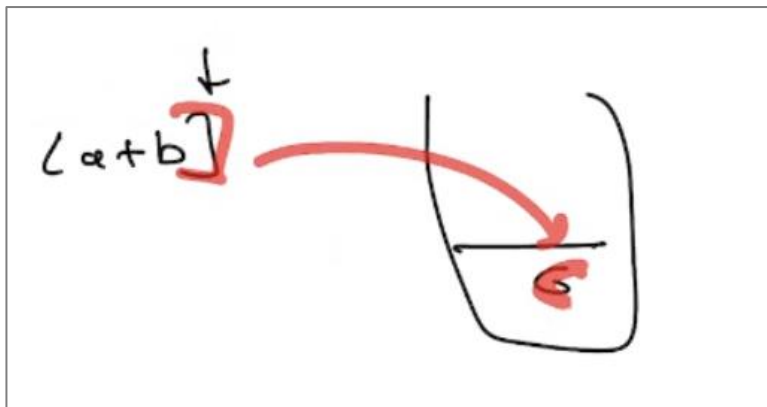


This open round bracket should match with the closed round bracket, if they are of same kind, delete it,



At last, if nothing is left in the stack, then it is valid parenthesis.

Example 2:



This sequence is not valid coz we have round bracket in the stack but square bracket is found next.

Pseudocode

```
Stack < char > st;  
for ( i = 0; i < N; i++)  
{  
    if ( s[i] == '(' || s[i] == '[' || s[i] == '{' )  
    {  
        st.push(s[i])  
    }  
    else if ( s[i] == ')' || s[i] == ']' || s[i] == '}' )  
    {  
        if ( s[i] == '}' )  
        {  
            if ( st.size == 0 || st.peek() != '{' )  
                return false;  
        }  
        if ( s[i] == ']' )  
        {  
            if ( st.size == 0 || st.peek() != '[' )  
                return false;  
        }  
        if ( s[i] == ')' )  
        {  
            if ( st.size == 0 || st.peek() != '(' )  
                return false;  
        }  
    }  
}  
return true
```

```
return st.size == 0
```

return true when stack is empty as no invalid parenthesis was found

TC = O(N) → N is due to a for loop

SC = O(N) → since you're utilizing space and in worst case it would be N

Actual Code

```
public class Solution {
    public int solve(String A) {
        Stack <Character> st = new Stack<>();
        for(int i=0; i<A.length(); i++){
            char ch = A.charAt(i);
            if(ch == '(' || ch == '{' || ch == '['){
                st.push(ch);
            }else if(ch == ')' || ch == '}' || ch == '']){
                if(st.isEmpty() ||
                    (ch == ')' && st.peek() != '(') || (st.peek() != '(' && ch == ')') ----> to make complete (), if stack doesnot have closing ) -> return 0
                    (ch == '}' && st.peek() != '{') ||
                    (ch == ']' && st.peek() != '[')){
                    return 0;
                }
                st.pop();
            }
        }
        return st.isEmpty()?1:0;
    }
}
```

2.Problem Statement - Double Character Trouble

Given a string, remove a pair of consecutive equal characters. Keep doing it until you can. Return final ans string.

You have a string, denoted as **A**.

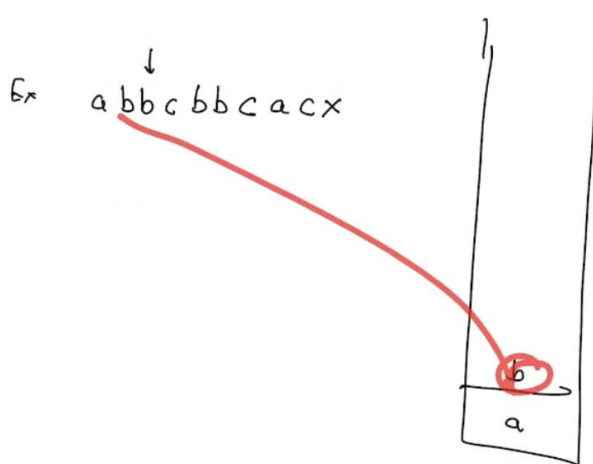
To transform the string, you should perform the following operation repeatedly:

1. Identify the **first occurrence** of **consecutive identical pairs** of characters within the string.
2. **Remove this pair** of identical characters from the string.
3. Repeat steps 1 and 2 until there are **no more** consecutive identical pairs of characters.

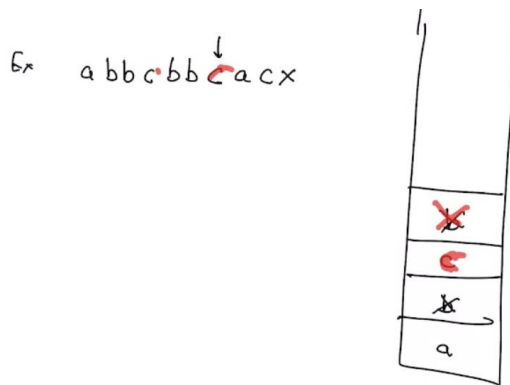
The **final result** will be the transformed string.

Example: abcdcd → delete dd → abcc → remove cc → ab

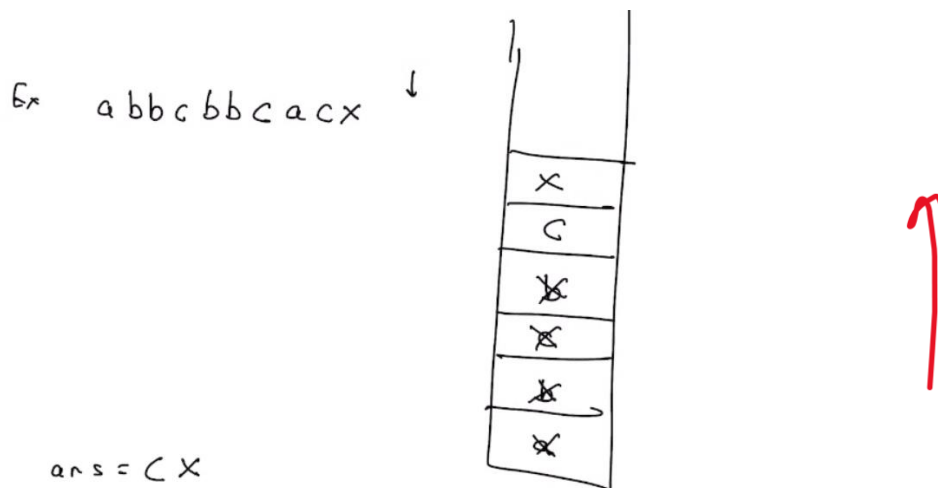
ans = ab



Keep adding character to the stack. Check if it is matching with the new character, if yes, the remove that character



Next character is c, in the string and in the stack → remove c



Your ans will be form bottom to top.

Actual Code

```

public class Solution {
    public String solve(String A) {
        Stack <Character> st = new Stack <>();
        for(int i = 0; i < A.length(); i++){                // takes O(N)
            char ch = A.charAt(i);
            if(!st.isEmpty() && st.peek() == ch){
                st.pop();
            }else {
                st.push(ch);
            }
        }

        StringBuilder res = new StringBuilder();            //takes O(N)
        while (!st.isEmpty()) {
            res.append(st.pop());
        }
        return res.reverse().toString();
    }
}

```

TC = $O(N + N) = O(N)$

SC = $O(N)$

3. Problem Statement – Evaluate Postfix

An arithmetic expression is given by a string array **A** of size **N**. Evaluate the value of an arithmetic expression in **Reverse Polish Notation**.

Valid operators are +, -, *, /. Each string may be an **integer** or an **operator**.

Note: Reverse Polish Notation is equivalent to **Postfix Expression**, where operators are written **after** their operands

Problem Constraints: $1 \leq N \leq 10^5$

Input Format: The only argument given is string array A.

Output Format: Return the value of arithmetic expression formed using reverse Polish Notation.

Example Input 1: A = ["2", "1", "+", "3", "*"]

Example Output 1: 9

Example Explanation 1:

starting from backside:

* : () * ()

3 : () * (3)

+ : (() + ()) * (3)

1 : (() + (1)) * (3)

2 : ((2) + (1)) * (3)

((2) + (1)) * (3) = 9

Example Input 2: A = ["4", "13", "5", "/", "+"]

Example Output 2: 6

Explanation 2:

starting from backside:

$+: () + ()$

$/: () + (() / ())$

$5: () + (() / (5))$

$13: () + ((13) / (5))$

$4: (4) + ((13) / (5))$

$(4) + ((13) / (5)) = 6$

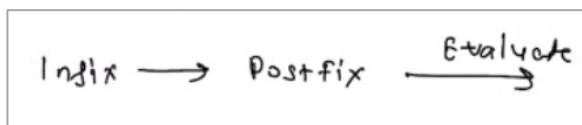
$2 + 3$ is called Infix \rightarrow operator between the numbers

$+23$ is prefix \rightarrow '+' operator in the first

$23 +$ is postfix \rightarrow operator is at the last

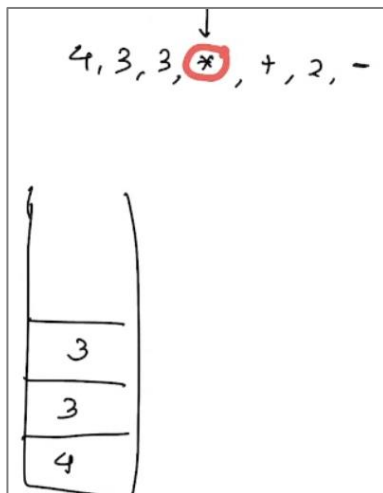
$(2 + 3) - 5$	$- + 23 5$	$23 + 5 -$
Infix	Prefix	Postfix

The compiler doesnot know bodmas, so it converts infix to postfix and then evaluate.



Example:

$4, 3, 3, *, +, 2, -$




When you see operands, just store them, then $a = 3, b = 3 \rightarrow$ pop them out

Now $3 * 3 = 9$

Add 9 in the stack and so on

4, 3, 3, *, +, 2, - ↓

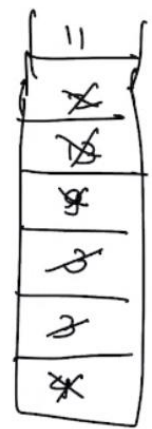


int b = 3
int a = 3
 $3 \times 3 = 9$

b = 9
a = 4
 $a + b = 9 + 4 = 13$

pop a = 4, pop b = 9

4, 3, 3, *, +, 2, - ↓



b = 2
a = 13
 $a - b = 13 - 2 = 11$

int b = 3
int a = 3
 $3 \times 3 = 9$

b = 9
a = 4
 $a + b = 9 + 4 = 13$

Whatever value is remaining in the stack will be the ans

```

for (i = 0 to n-1) {
    if (A[i] is operand/integer) {
        st.push(A[i])
    }
    else {
        operator = getOperator(A[i])
        x = st.pop()
        y = st.pop()
        z = y operator x
        st.push(z)
    }
}
return st.peek()

```

Actual Code

```

public class Solution {
    public int evalRPN(ArrayList<String> A) {
        Stack<Integer> st = new Stack<>();
        for(int i=0; i<A.size(); i++){
            String str = A.get(i);
            if(isOperator(str) ){
                int b = st.pop();
                int a = st.pop();
                int res = 0;
                if(str.equals("+")){
                    res = a+b;
                }else if(str.equals("-")){
                    res = a-b;
                }else if(str.equals("*")){
                    res = a*b;
                }else if(str.equals("/")){
                    if(b !=0) res = a/b;
                }
            }
        }
    }
}

```



```

        st.push(res);
    }else{
        st.push(Integer.parseInt(str)); //convert a string representing a number (operand) into an integer
        //and push it onto the stack
    }
}
return st.pop();
}
private boolean isOperator(String str){
    return str.equals("+") || str.equals("-") || str.equals("*") || str.equals("/");
}
}

```

Stack <String> st

for (i = 0 ; i < n ; i++)

{

 if (s[i] is operand)

 {

 st.push (s[i])

 }

 else if (s[i] is operator)

 {

 b = st.pop()

 a = st.pop()

 c = perform a s[i] b

 st.push (c)

 }

 }

return st.top()

 st.pop()

4.Problem Statement - Min StackSolved

Design a stack that supports **push**, **pop**, **top**, and **retrieve the minimum element** in constant time.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

getMin() -- Retrieve the minimum element in the stack.

NOTE:

All the operations have to be constant time operations.

getMin() should return **-1** if the stack is empty.

pop() should return **nothing** if the stack is empty.

top() should return **-1** if the stack is empty.

Problem Constraints: $1 \leq \text{Number of Function calls} \leq 10^7$

Input Format: Functions will be called by the checker code automatically.

Output Format: Each function should return the values as defined by the problem statement.

Example Input

Input 1:

push(1)

push(2)

push(-2)

getMin()

pop()

getMin()

top()

Input 2:

getMin()

pop()

top()

Example Output

Output 1:

-2 1 2

Output 2:

-1 -1

Example Explanation

Explanation 1:

Let the initial stack be : []

1) push(1) : [1]

2) push(2) : [1, 2]

3) push(-2) : [1, 2, -2]

4) getMin() : Returns -2 as the minimum element in the stack is -2.

5) pop() : Return -2 as -2 is the topmost element in the stack.

6) getMin() : Returns 1 as the minimum element in stack is 1.

7) top() : Return 2 as 2 is the topmost element in the stack.

Actual Code

```
class Solution {
    Stack<Integer> st = new Stack<>();
    Stack<Integer> minst = new Stack<>();
    public void push(int x) {
        st.push(x);
        if(minst.isEmpty()){
            minst.push(x);
        }else if (x <= minst.peek()){
            minst.push(x);
        }
    }
    public void pop() {
        if(st.isEmpty()) return;
        int num = st.pop();
        if(num == minst.peek()) minst.pop();
    }
    public int top() {
        if(st.isEmpty()) return -1;
        return st.peek();
    }
    public int getMin() {
        if(minst.isEmpty()) return -1;
        return minst.peek();
    }
}
```

5.Problem Statement - Check two bracket expressions

Given two strings A and B. Each string represents an expression consisting of lowercase English alphabets, '+', '-', '(', and ')'.

The task is to compare them and check if they are similar. If they are identical, return 1 else, return 0.

NOTE: It may be assumed that there are at most 26 operands from 'a' to 'z', and every operand appears only once.

Problem Constraints: 1 <= length of the each String <= 100

Input Format: The given arguments are string A and string B.

Output Format: Return 1 if they represent the same expression else return 0.

Example Input1:

A = "-(a+b+c)"

B = "-a-b-c"

Example Output1:1

Input 2:

A = "a-b-(c-d)"

B = "a-b-c-d"

Output 2:

0

Example Explanation

Explanation 1:

The expression "-(a+b+c)" can be written as "-a-b-c" which is equal as B.

Explanation 2:

Both the expression are different.

Actual Code

```
public class Solution {
    public int solve(String A, String B) {
        int [] resultA = evaluateExpression(A);
        int [] resultB = evaluateExpression(B);

        for(int i =0; i<26; i++){
            if(resultA[i] != resultB[i]) return 0;
        }
        return 1;
    }
    private int[] evaluateExpression(String expr){
        int[] result = new int[26];
        Stack<Integer> signStack = new Stack<>();
        signStack.push(1);

        int currentSign = 1;
        for(int i =0; i < expr.length(); i++){
            char ch = expr.charAt(i);
            if(ch == '+' || ch == '-'){
                currentSign = (ch == '+')? 1: -1;
            } else if(ch == '('){
                signStack.push(signStack.peek() * currentSign);
                currentSign =1;
            } else if(ch == ')'){
                signStack.pop();
            } else if(Character.isLetter(ch)){
                result[ch - 'a'] += signStack.peek() * currentSign;
                currentSign =1;
            }
        }
        return result;
    }
}
```

6. Problem Statement - Redundant Braces

Problem Description

Given a string **A** denoting an expression. It contains the following operators '+', '-', '*', '/'.

Check whether A has redundant braces or not.

NOTE: A will be always a valid expression and will not contain any white spaces.

Problem Constraints

$1 \leq |A| \leq 10^5$

Input Format

The only argument given is string A.

Output Format

Return 1 if A has redundant braces else, return 0.

Example Input

Input 1:

A = "((a+b))"

Input 2:

A = "(a+(a+b))"

Example Output

Output 1:

1

Output 2:

0

Example Explanation

Explanation 1:

((a+b)) has redundant braces so answer will be 1.

Explanation 2:

(a+(a+b)) doesn't have have any redundant braces so answer will be 0.

Actual Code

```
public class Solution {
    public int braces(String A) {
        Stack<Character> st = new Stack<>();

        for (int i = 0; i < A.length(); i++) {
            char ch = A.charAt(i);

            if (ch == '(' || ch == '+' || ch == '-' || ch == '*' || ch == '/') {
                st.push(ch);
            } else if (ch == ')') {
                boolean hasOperator = false;

                while (!st.isEmpty() && st.peek() != '(') {
                    char top = st.pop();
                    if (top == '+' || top == '-' || top == '*' || top == '/') {
                        hasOperator = true;
                    }
                }
                if (!hasOperator) return 1;
            }
        }
        return 0;
    }
}
```

```

    }
}

if (!st.isEmpty() && st.peek() == '(') {
    st.pop(); // Pop the matching '('
}

if (!hasOperator) {
    return 1; // Redundant braces
}
}
}

return 0; // No redundant braces
}
}

```

7.Problem Statement - Infix to Postfix

Problem Description

Given string **A** denoting an infix expression. Convert the infix expression into a postfix expression.

String A consists of ^, /, *, +, -, (,) and **lowercase English alphabets** where lowercase English alphabets are operands and ^, /, *, +, - are operators.

Find and return the postfix expression of A.

NOTE:

- ^ has the highest precedence.
- / and * have equal precedence but greater than + and -.
- + and - have equal precedence and lowest precedence among given operators.

Problem Constraints

1 <= length of the string <= 500000

Input Format

The only argument given is string A.

Output Format

Return a string denoting the postfix conversion of A.

Example Input

Input 1:

A = "x^y/(a*z)+b"

Input 2:

A = "a+b*(c^d-e)^(f+g*h)-i"

Example Output

Output 1:

"xy^az*/b+"

Output 2:

"abcd^e-fgh*+^*+j-"

Example Explanation

Explanation 1:

Output denotes the postfix expression of the given input.

Actual Code

```
public class Solution {
    public String solve(String A) {
        StringBuilder result = new StringBuilder();
        Stack<Character> st = new Stack<>();

        for (int i = 0; i < A.length(); i++) {
            char ch = A.charAt(i);

            if (Character.isLetter(ch)) {
                // Append operands directly to the result
                result.append(ch);
            } else if (ch == '(') {
                st.push(ch);
            } else if (ch == ')') {
                while (!st.isEmpty() && st.peek() != '(') { //Pop operators from the stack until an opening
                    // parenthesis ( is encountered.
                    result.append(st.pop());
                }
                st.pop(); // Pop the '('
            } else {
                // Operator
                while (!st.isEmpty() && precedence(st.peek()) >= precedence(ch)) {
                    result.append(st.pop());
                }
                st.push(ch);
            }
        }

        // Pop remaining operators in the stack
        while (!st.isEmpty()) {
            result.append(st.pop());
        }

        return result.toString();
    }

    private int precedence(char ch) {
        if (ch == '^') return 3;
    }
}
```

```
    if (ch == '*' || ch == '/') return 2;  
    if (ch == '+' || ch == '-') return 1;  
    return 0;  
}  
}
```