Stacks 1: Implementation & Basic Problems

↳ A Stack is a linear data structure that stores information in a sequence, from bottom to top.

↳ The data items can only be accessed from the top & new elements can only be added to the top, i.e. it follows LIFO (Last in First out) principle

Real Life Examples

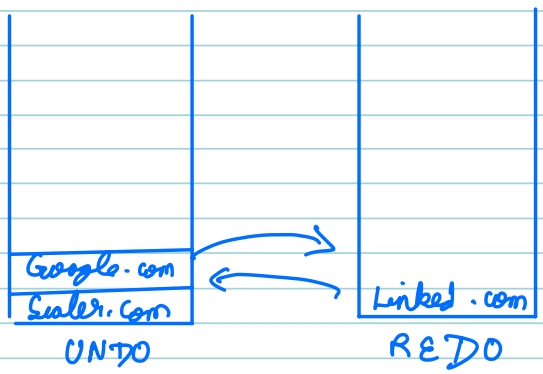
(1) Pile of Plates

(2) Pile of chairs

Quiz 1 :- what is the most common application of Stack ?

- → Evaluating arithmetic Expression
- → Implementing undo / redo functionality
- → Used in recursion internally.

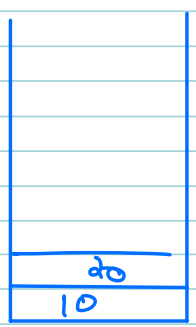
UNDO / REDO



STACK

OPERATIONS OF STACK

1.) PUSH (data) → Insert a new element into top of the stack

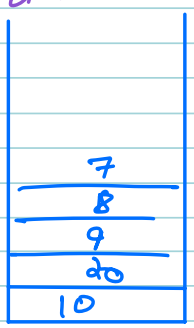


2.) Pop () → Remove an element out of the stack.

3.> Peek() ⟶ Gives access to top element of stack, without removing it.

↓

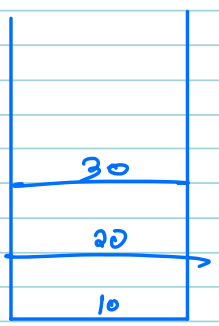
(top())



4.> isEmpty() ⟶ checks stack is empty or not.

SYNTAX

```
Stack<Integer> st = new Stack<>();
st.push(10)
st.push(20)
st.push(30)
```



Quiz 2 :- what is the time complexity of the push & pop operations in a stack.

↳ O(1) for push & pop.

Implementation Of Stack Using Arrays

Internally array but outer person should feel that we are using stack

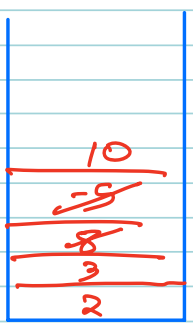
arr =

0	1	2	3	4	5
2	3	8 → 10			

TOP = ~~-1 0 1 2 3 2 1~~

↳ operations

→ Push (2) → top++ → place value
→ Push (3) → top++
→ Push (8) → top++
→ Pop ( ) → top--
→ Push (-5) → top++
→ Peek ( ) → return arr [top]
→ Pop ( ) → top--
→ Push (10) → top++
→ Pop ( ) → top--

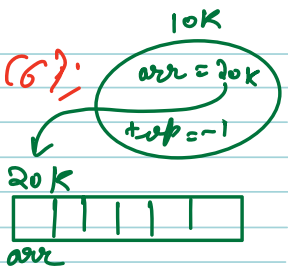


CODE

```
class Stack {
    Private int[] arr;
    Private int top;
    stack ( data) {
        arr = new int [data];
        top = -1
    }

    void push ( int data ){
        top++
        arr [top ] = data;
    }
    int pop( ) {
        int x = arr [top ];
        top--;
        return x;
    }

    int peek ( ) {
        return arr [top ];
    }

    boolean is Empty ( ) {
        if ( top == -1) {
            return True;
        }
        return False;
    }
```

}

Stack st = new Stack(6);



Q Here stack internally is using arrays only but can anyone with 'st' variable access array

↳ No

Quiz :- what happens when you try to pop an element from an empty stack?

↳ It causes an underflow

OVERFLOW

↳ when stack is full.

```
void push ( int data ){
    if ( top >= A.size()-1 ){
        return;
    }
    top++
    arr[top] = data;
}
```

Eg:

UNDERFLOW

↳ Try to pop / peek from empty stack.

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

Eg:

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

Problem in Implementation using Arrays

We have to predefine the stack size to create array. To overcome this we can create Dynamic array which can grow or shrink at runtime according to need.

IMPLEMENTATION OF STACK USING LINKED LIST (LL)

operations of LL

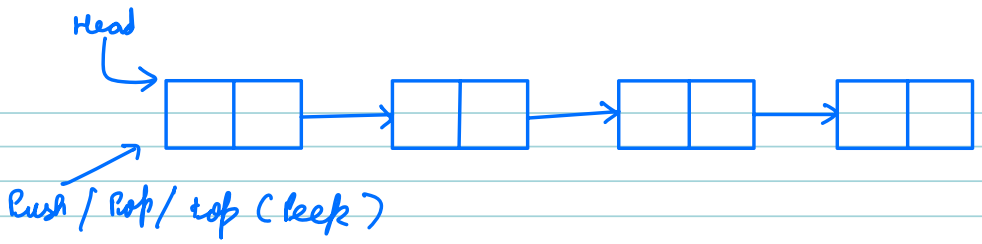
head & tail is maintained

- → addLast() → O(1)
- → addFirst() → O(1)
- → removeLast() → O(N)
- → removeFirst() → O(1)

Q which Combination of LL I should use?

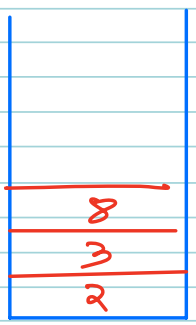
(i) addLast(), removeLast()

(ii) addFirst(), removeFirst()



DRY RUN

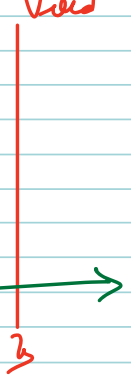
→ Push (2) → addFirst()
→ Push (3) → addFirst()
→ push (8) → addFirst()
→ Pop() → removeFirst()
→ Push (-5)
→ Peek()
→ Pop()
→ Push (10)



Code

```
Void push (data) {
    Node temp = new Node(data);
    temp.next = head;
    head = temp;
    t++;
```

For Size of Stack.



```
void pop() {
    if (head == null) {
        return;
    }
    head = head.next;
    t--;
}

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

TC → O(1)

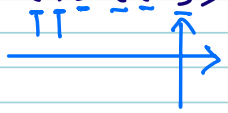
Question :- check whether the given sequence of parenthesis is Valid ?

{ , [, (

Ex :-

1. ( ) [{ } ( )] → Valid
2. ( ) [{ (})] → Invalid
3. ({ } [ ] → Invalid
4.) ([ ] → Invalid
5. ({ [ ] }) → Valid.

Q why ( ) [{ (})] is Invalid ?



Because of this bracket.

Q why ({ } [ ] is Invalid ?

↳ No Closing bracket available.

Quiz :- which of the following expression is balanced w.r.t. to parenthesis ?

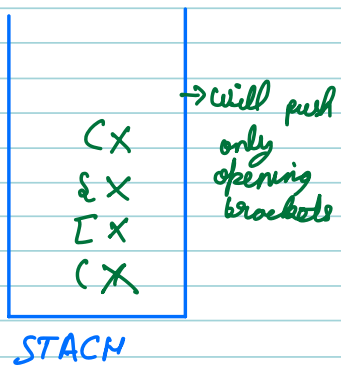
↳ (a * b) * c

DRY RUNS

Ex1

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
( ) [{ } ( )] ⇒ End

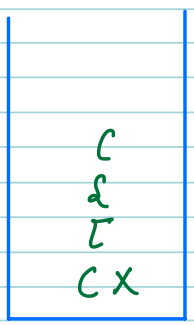
⇒ VALID



Ex2 :-

↓ ↓ ↓ ↓ ↓ ↓
( ) [{ (})]

⇒ If No match, its Invalid.



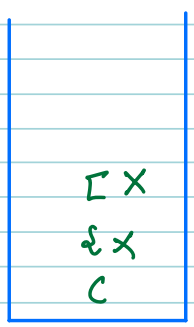
Ex3

↓
) ([ ]

⇒ If Empty, then Invalid.

Ex4

↓ ↓ ↓ ↓ ↓
({ } [ ] ⇒



⇒ Invalid, if stack is Not empty at last.

PSEUDO CODE

```
bool isValidParenthesis(sequence)$
    For all character :
        if char is -> '{' '[' '(' :
            Push char to stack
        else if char is closing :
            if stack is Empty :-
                return False;
            Pop element
            if popped bracket doesn't match :-
                return False
    If Stack is not Empty :
        return False
    else
        return True
```

TC → O(N)

SC → O(N)

Question :- Given a String, remove equal pair of consecutive elements till possible.

Ex :- ① a b b c → a c

② a b c c b d e → a b b d e → a d e

③ e b b b c c c a → a b c c c a → a b c a

Quiz 5 :- If we remove equal pair of consecutive character in this String. Then what will be Ans?

⇒ a b b c b b c a c x

↓

a c c a c x

↓

a a c x

↓

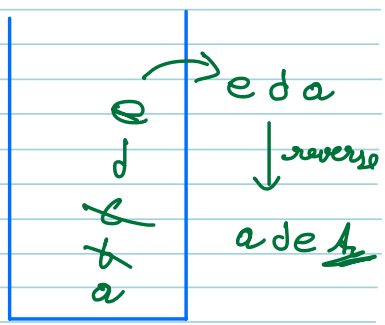
c x

DRY RUN

Ex1 :- a b c c b d e

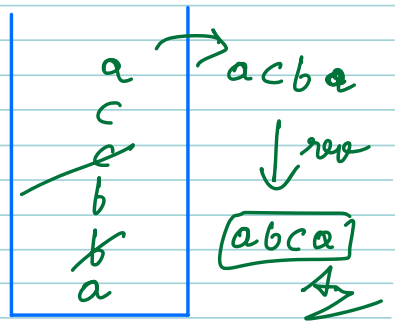
Step 1 :- check if element of top matches, if not insert, if yes remove

Step 2 :- At End pop the element from stack & reverse to get order.



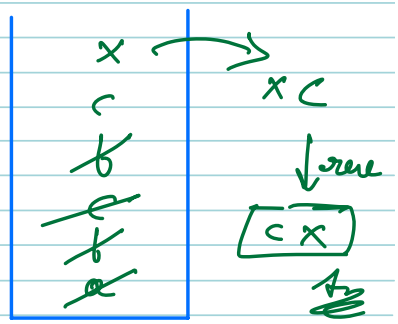
Ex 2:-

a bbb ccc a



Ex 3:-

a bb c bb c a c x



PSEUDO CODE

```
For all characters :-
    if top matches ( ) {
        pop ( )
    } else {
        Push ( character )
    }
}
build string by taking from stack
Reverse string.
```

Q Can we do something for avoiding reverse at last

↳ Iterate from last [i.e. Right to Left]

TC → O(N)

SC → O(N)

POSTFIX NOTATION

Infix Notation	Postfix Notation
2 + 3 (+ : operator; 2, 3 : operands)	2 3 + (+ : operator; 2, 3 : operands)

* The only difference is writing of expression

Eg:- a+b−c * (d+e) ← Infix Notation

↓

a+b−c * de+

↓

a+b − cde+ *

↓

ab+ − cde+ *

↓

ab+ cde+ * − ← Postfix

Eg 2 $2 * (3 - (6/2))$ ← Infix Notation

↓

$2 * (3 - 62/)$

↓

$2 * 362/-$

↓

$2362/-*$ ← Postfix Notation.

⇒ Mathematically

Infix Notation	Postfix Notation
⇒ $2 * (3-(6/2))$	⇒ $2362/-*$
⇒ $2 * (3-3)$	↳ Can be resolved in 1 iteration.
⇒ $2 * 0$	
⇒ 0	
took 3 iterations	

Benefits of Postfix Notation

→ Faster / Easier to Evaluate

→ Doesn't have any brackets.

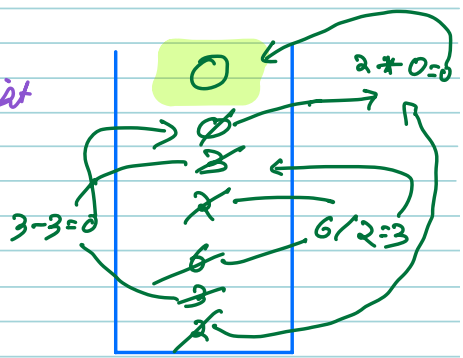
question:- Given a Postfix expression, evaluate it.

Ex

2 3 6 2 / - *

Step1:- Iterate from Left to Right

Step2:- Keep Inserting, if got operator then pull the top two numbers, apply operator & push it back.

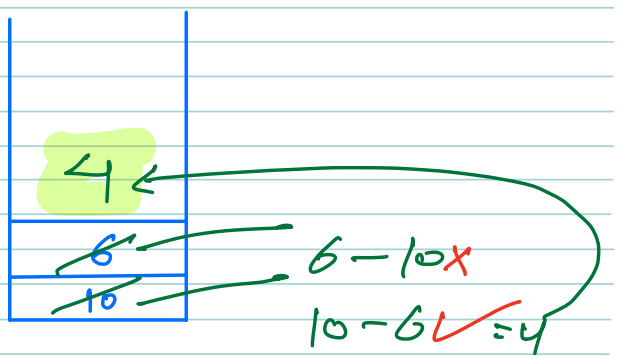


Step3:- After Inserting from Left to Right only one number will left & that's A

Ex2

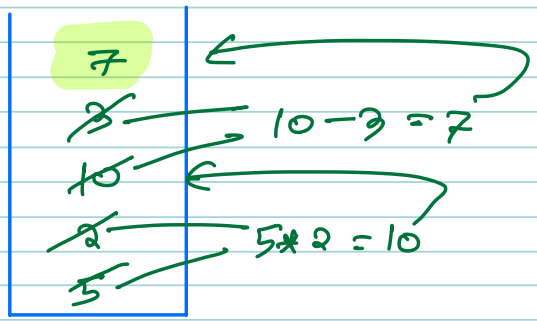
10 | - | 6 ⇒ 10 | 6 | -

Infix Postfix



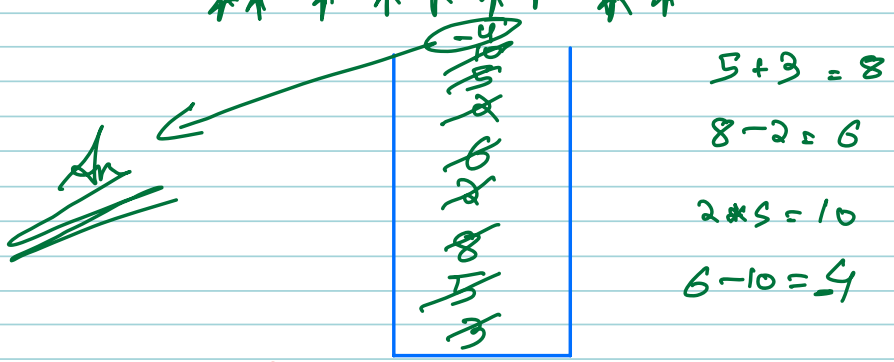
Quiz 6 :- Answer using stack for this Postfix

5 , 2 , * , 3 , -



Quiz 7 :- Evaluate the given postfix Expression.

3 5 + 2 - 2 5 * -

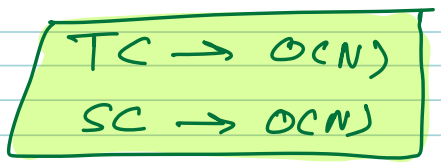


PSEUDO CODE

Travel the Expression :

```
if (operand) {
    Push (operand);
} else {
    x = pop();
    y = pop();
    Push (y operator x);
}
```

whatever left store Ans



SUMMARY OF LECTURE

Conclusion

- Stack is a linear data structure which stores data in LIFO manner.
- The element which is inserted last will be popped out first.
- In stack only top element is accessible.
- All operations of stack are of constant time `O(1)`.
- Stack is used at several places in real life problems, like postfix evaluation in calculators, valid paranthesis check in code editor.
- We can implement stack with arrays or dynamic linked list in a conveninent manner.