Stacks

Defn :-

- 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 and new elements can only be added to the top, i.e it follows **LIFO (Last In First Out)** principle.

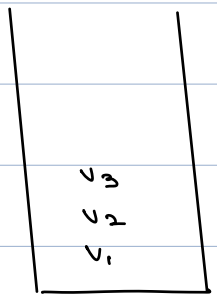
1) Pile of Plates.

2) Stack of chairs.

Algorithmic Examples :-

1) Recursion,

2) Undo - Redo.



$v_1 \rightarrow v_2 \rightarrow v_3$

ArrayList
↳ add
↳ remove.
↳ get,

Operations on Stack

1) push (data) ⟶ inserts a new element into top of the stack.

2) pop() ⟶ Removes an element out of the stack.

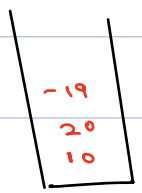
3) peek() → gives access to top element of stack, without removing it.

↓
top()

```
— st —
int x = st.peek();
```

4) isEmpty() → checks stack is empty or not,

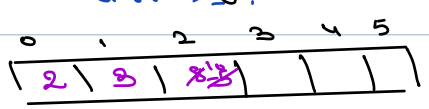
```
Stack<Integer> st = new Stack();
st.push(10);
st.push(20);
st.push(-19);
```



→ Implementation of stacks using Arrays.

arr -> 6.

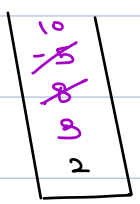
top = ~~-1~~ ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 2



push(2) ✓

push(3) ✓

push(8) ✓



- ✓ pop();, ⟶
- ✓ push(-5)
- ✓ peek() ⟶
- ✓ pop() ⟶
- ✓ push(10)

```
class Stack {
    private int[] arr;
    private int top;

    Stack() {
        arr = new int[6];
        top = -1
    }

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

    int pop() {
        int x = arr[top];
        top--;
        return x;
    }
}
```

int peek () {

return arr[top];

}

boolean isEmpty () {

if (top == -1) {

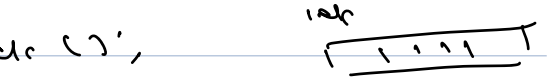
return true;

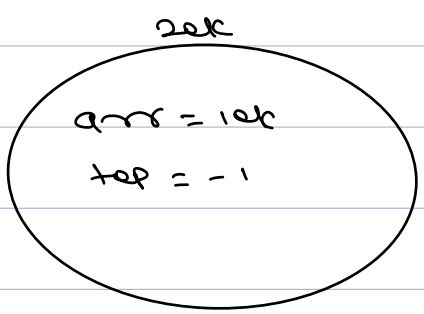
}

return false;

}

Stack st = new stack ();





Overflow

```
void push(x){
    // Whenever our counter reaches to the size of the array
    // It means stack is already full
    if(t >= A.size())
        return;
    t++;
    A[t] = x;
}
```

Underflow :-

- Underflow means when we try to perform pop operation or try to access the element of stack but there are none. Again we have to introduce conditions during pop and top operation.

```
void pop(){
    if(isEmpty()) return;
    t--;
}
```

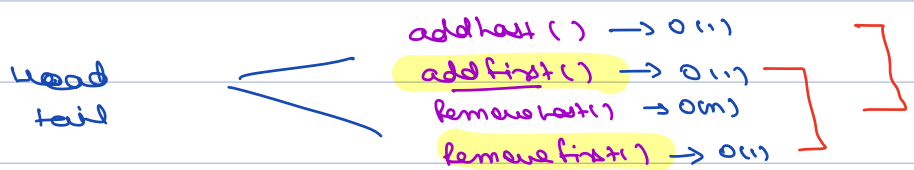
peek

```
int top(){
    if(isEmpty()) return -1;
    return A[t];
}
```

Problem :-

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

-: Implementing Stacks using LL :-



Push(2) ✓

Push(3) ✓

Push(8) ✓

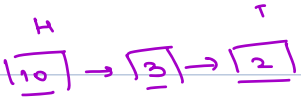
Pop(); ✓ → ✓

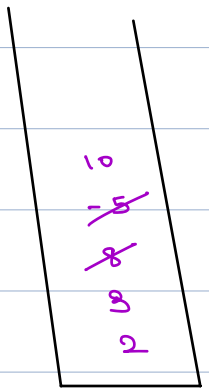
Push(-5) ✓

Peek() → -5

Pop() → ✓

Push(10) ✓






```
void push(data){
    new_node = Create a new Node with 'data'
    new_node.next = head
    head = new_node
    // Increment size
    t++
}
```

→ internal variable for size.

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

→ Peek()

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

Ques Check given, sequence of parenthesis is valid ? { , [, (

e.g -> ( ) [{ } ( )] → True

( ) [{ (})] → Invalid

(({ } [ ] → invalid

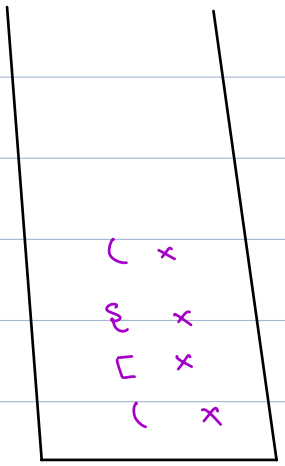
) ([ ] → invalid

({ [ ] }) → valid

→ closing Bracket → opening brkt should be of same type.

→ All opening Brts, should be closed.

( ) [{ } ( )]



([{ ( ) }])
(x
[
{
)
return false .

([ ])
return false .

({ [ ] })
(x
{ x
[x
return false .

```
//String
Stack<char> st = new Stack<>();
```

T.C → O(n), S.C → O(n).

Traverse complete string (for →)

ch →

openBracket → push(x)

close →

st is empty → return false

check top of stack();

match → pop();

no match :-

↳ return false;

3

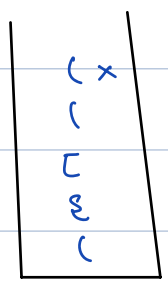
Break

10:11 pm – 10:21 pm

stack → empty → return True

else ↳ return false;

[{ [( )] }]]



Ques :-

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

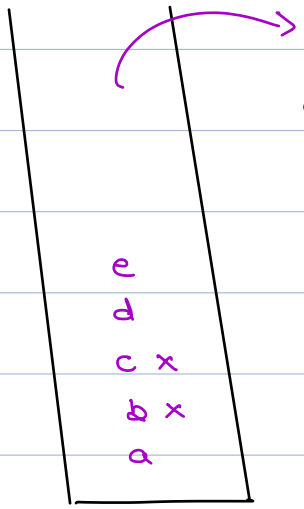
e.g → abbc → ac

abccbde → abbde → ade

abbbccca → abcccca → abca

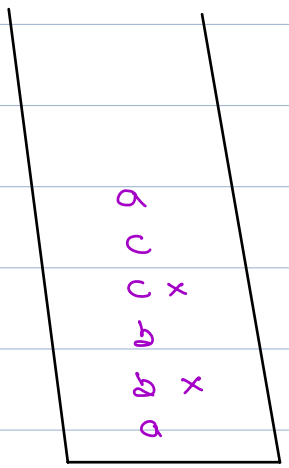
abbcbbcacx → accacx → aacx → cx.

a b c c b d e



e d a
rev
a d e

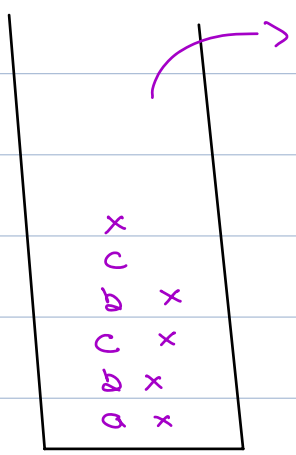
a b b b c c c a



→ a c b a
rev
a b c a

e.g,

a b b c b b c a c x



x c
rev
c x

∀ characters

if top matches → pop()

else push(x)

↳ reverse what you got out of stack.

T.C → O(n)

S.C → O(n)

Post fix Notation

Infix Notation

2 + 3

operand | operand

operator

Post fix Notation

2 3 +

operand1 operand2 operator

e.g.,

a + b − c * (d + e)

↓

a + b − c * de+

↓

a + b − cde+*

↓

ab+ − cde+*

↓

ab+cde+*−

e.g , 2 * (3 - (6 / 2))

→ 2 * (3 - 6 2 /)

→ 2 * 3 6 2 / -

→ 2 3 6 2 / - *

2 * (3 - 3)

2 * 0

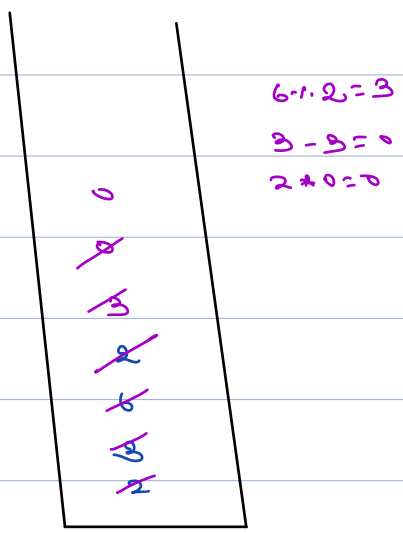
0.

Benefits of Postfix

1) Easier to Evaluate → O(n)
2) Doesn't have any brackets

Evaluate Postfix Expression

2 3 6 2 / - *

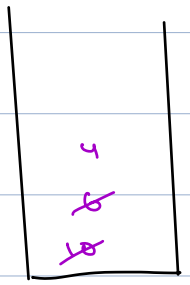


6 / 2 = 3

3 - 3 = 0

2 * 0 = 0

eg) 10 − 6 → 10 6 −



10 − 6

e.g 2 3 5 + 2 − 2 5 * −

3 + 5
8 − 2
2 * 5
6 − 10

T.C → O(n)
S.C → O(n)

Travel the Expression

```
if (operand) {
    push (opr);
}
else {
    x = pop();
    y = pop();
    push (y opr x);
}
```

whatever is left in stack is ans.

Summary

- 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.

Corrected code

```
public class Solution {


    public int[] searchRange(final int[] A, int B) {
        int n = A.length;
        int lo=0;
        int hi=n-1;
        int ans_left=-1;

        //find left_min
        while(lo<=hi){
            int mid=lo+(hi-lo)/2;
```

```
        if(A[mid]<B){
            lo=mid+1;
        } else if(A[mid]>B){
            // hi=mid+1;
            hi = mid - 1;
        } else if(A[mid]==B){
            hi=mid+1;
            ans_left=mid;
        }
    }


    int ans_right=-1;

    //find right_min
    lo=0;
    hi=n-1;
    while(lo<=hi){
        int mid=lo+(hi-lo)/2;

        if(A[mid]<B){
```

```
            lo=mid+1;
        } else if(A[mid]>B){
            hi=mid-1;
        } else if(A[mid]==B){
            lo=mid+1;
            ans_right=mid;
        }
    }

    return new int[]{ans_left, ans_right};
  }
}
```