Agenda

1) Stacks Basics
2) Functions in Stack
3) Implementation of stack
   - using array
   - using LL
4) Balanced parenthesis *
5) Double char trouble
6) Evaluate postfix expression

} 3 questions

Stack Basics

Stacks follows : LIFO

↳ Last in first out

push (x) → add x onto the stack

pop () → remove topmost element of stack

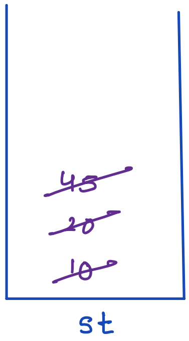
peek () → get topmost element of stack

size () → no. of elements in stack.

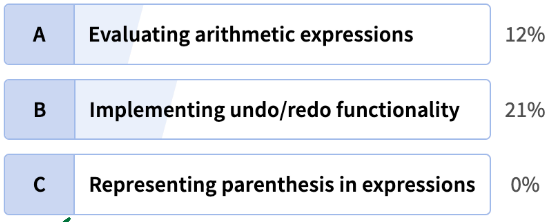
O(1)

↱ Java's stack

```
Stack < Integer > st = new Stack <> ( ) ;
st.push(10);
st.push (20);
st.push (45);
st.pop ( ) ;
SOPln(st.peek( )); ⟶ 20
st.pop( );
st.pop( );
SOPln(st.peek( )); ⟶ Empty stack exception
```



note : try not to do st.peek ( ) or st.pop( ) on empty stack.

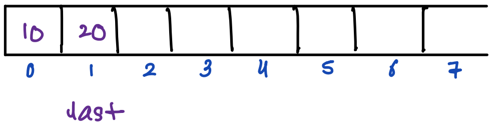


A	Evaluating arithmetic expressions	12%
B	Implementing undo/redo functionality	21%
C	Representing parenthesis in expressions	0%

[✓ D. All the above]

Implementation of Stack using Arrays

cap = 8



last = ~~-1~~
~~0~~
~~1~~
2

```
class Stack {
    int[] A;
    int last;

    Stack (int cap) {
        A = new int[cap];
        last = -1;
    }

    void push(int x) {
        last++;
        A[last] = x;
    }

    int pop() {
        int temp = A[last];
        A[last] = 0;
        last--;
        return temp;
    }

    int peek() {
        return A[last];
    }

    int size() {
        return last+1;
    }
}
```

```
Stack st = new Stack(8);
st.push(10); ✓
st.push(20); ✓
st.push(45); ✓
st.pop(); ✓
```

Implementation of Stack using LL

i) If tail is the working end

→ efficiency won't be achieved in pop()

push(x) → add Last (x) in LL → O(1) in SLL

pop() → remove last ( ) in LL → O(n) in SLL

ii) If head is the working end

→ efficiency can be achieved

push(x) → add First (x) in LL → O(1) in SLL

pop() → remove First ( ) in LL → O(1) in SLL

St.push(10)

St.push(20)

St.push(30)

Q.1 Given an expression containing 3 types of brackets: (,), {,}, [,]
Check if the given exp. is balanced or not.

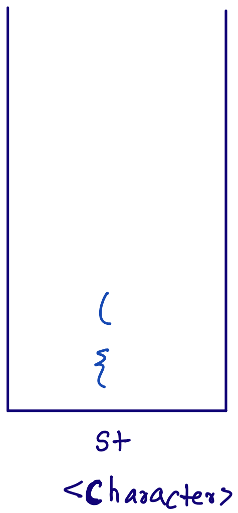
} (( ]) → false

(([ ])) → true

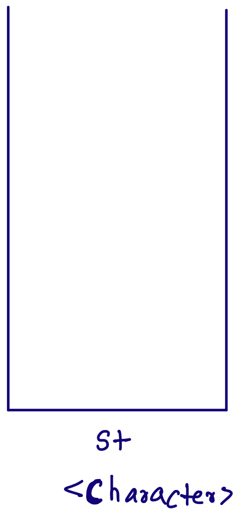
{ ([ ] }) → false

{ ( ) ( ) } [ ] → true

{ (~~[ ]~~ })
↑



~~{()()}[]~~
↑

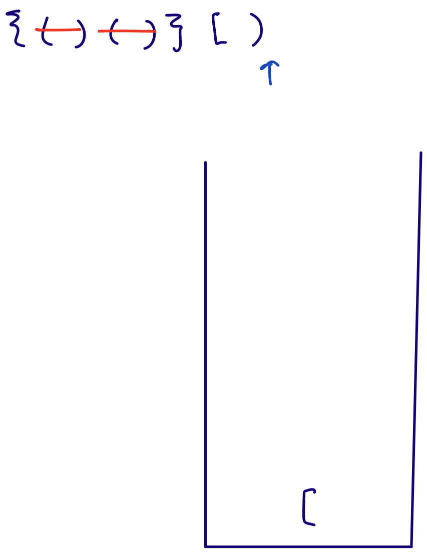


```
boolean  balancedParenthesis ( string str) {
    stack < Character> st= new stack<>();

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

        if (ch== '{' || ch== '(' || ch== '[') {
            st.push(ch);

        }
        else if (ch== '}' || ch== ')' || ch== ']') {
            if (isCompatible (ch, st.peek()) == false) {
                return false;
        }       }
        st.pop();
    }

    return st.size() == 0;
}
```

```
boolean isCompatible (char cl, char op) {
        if (cl == '}') {
              return op=='{';
        }
        else if (cl == ')') {
              return op == '(';
        }
        else {
              return op == '[';
        }
}
```

Dry run

```
boolean balancedParenthesis (string str) {
    Stack <Character> st = new Stack<>();

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

        if (ch == '{' || ch == '(' || ch == '[') {
            st.push(ch);

        }
        else if (ch == '}' || ch == ')' || ch == ']') {
            if (isCompatible (ch, st.peek()) == false) {
                return false;
            }
            st.pop();
        }
    }
    return st.size() == 0;
}
boolean isCompatible (char cl, char op) {
    if (cl == '}') {
        return op == '{';
    }
    else if (cl == ')') {
        return op == '(';
    }
    else {
        return op == '[';
    }
}
```



Q.2 Given a string S, remove equal pair of adjacent characters. Return the string without adjacent duplicates.

a~~bb~~d → ad

a~~bccb~~de → ade

a~~bb~~be → abe

ababab → ababab

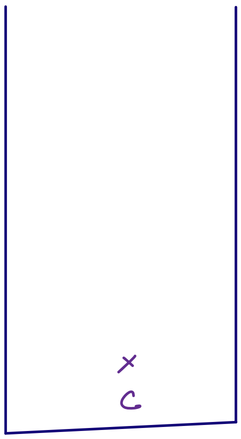
a~~debbecaacd~~ed → aed

~~dbbcbbcd~~cx → cx

~~dbbcbbcd~~cx

↑

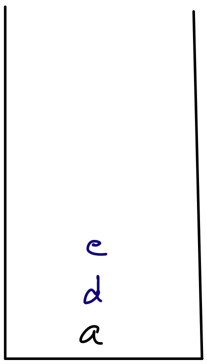
ans → cx



```
String  removeAdjEqual (string  str) {
    stack < Character> st = new  stack<>();

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

        if (st.size() == 0) {
            st.push(ch);
        }
        else  {
            if ( ch == st.peek()) {
                st.pop();
            }
            else {
                st.push(ch);
            }
        }
    }

    // build  ans  string  from stack  and  return  it

}
```

a~~bccb~~de

↑



ans= ade

TC: O(n)

SC: O(n)

Q.3 Given a postfix expression, return the evaluated answer.

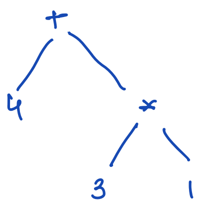
→ what is postfix expression : operator comes after operands

infix	postfix
2 + 5	2 5 +
7 * 3	7 3 *
4+3*1	4 3 1 * +

Infix

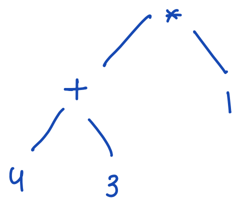
postfix

→ 4 + 3 * 1



4 3 1 * +

→ (4 + 3) * 1



4 3 + 1 *

Why postfix are superior

i) brackets are not needed

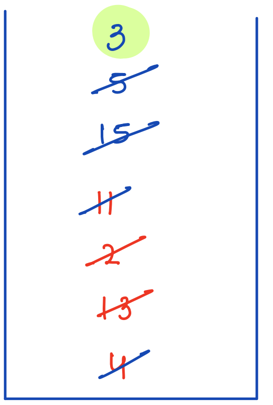
ii) Order in which operators are coming is same as execution order of operators.

["2", "1", "+", "3", "*"] → ans = 9

["4", "13", "5", "/", "+"] → ans = 6

A: ["4", "13", "2", "-", "+", "5", "/"] → ans=3

↑



Stack < Integer>

v2 = 5

v1 = 15

if A[i] is a number than push it to stack by converting into int.

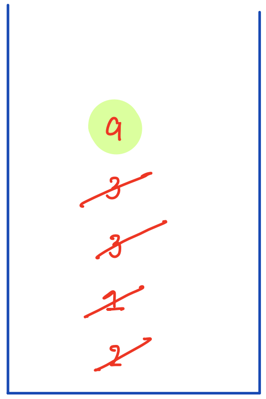
else if A[i] is −, +, /, * then pop last two operands, do calc. and push result on stack.

```
return st.peek();
```

* String to int

```
int val= Integer.parseInt(s);
```

["2", "1", "+", "3", "*"]



Stack < Integer>

v2 = 3

v1 = 3

if A [i] is a number than push it to stack by converting into int.

else if A[i] is −, +, /, * then pop last two operands, do calc. and push result on stack.

return st. peek ( );

code : todo