Agenda

i) Stack basics

ii) Double character trouble

iii) Expression evaluation

Stack: It follows LIFO

↳ last in first out

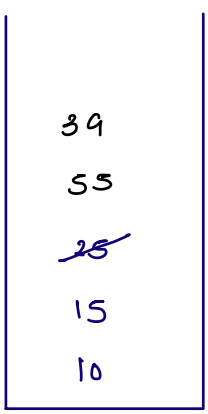
```
Stack<Integer> st = new Stack<>();
```

st → name of stack

|| `st.push(val)`, `st.pop()`, `st.peek()`

- `st.push(val)` → push val into the stack
- `st.pop()` → pop/remove the topmost element of stack
- `st.peek()` → gives the topmost value of stack

```
st.push(10);
st.push(15);
st.push(25);
st.pop(); → 25
st.push(55);
st.push(39);
sopln(st.peek()); → 39
```



eg2. `Stack<Integer> st = new Stack<>();`

`st.pop();`

In empty stack if we try to do st.peek(), st.pop()
we will get exception: **EmptyStackException**

Real Life use of stack:

i) ← (back) during browsing

ii) Undo / redo

iii) Expression evaluation

Q.1 Given a string, keep removing the same consecutive char until no more occurrence of same consecutive char remains. Return the final answer string.

Allowed TC: O(n)

str = a b ~~c c~~ b c

↓

a ~~b b~~ c

↓

ac

str = k b c ~~a a a~~ c b m b

↓

k b ~~c c~~ b m b

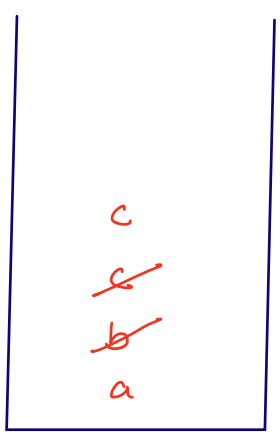
↓

k ~~b b~~ m b

↓

k m b

a ~~b~~ ~~c c~~ ~~b~~ c

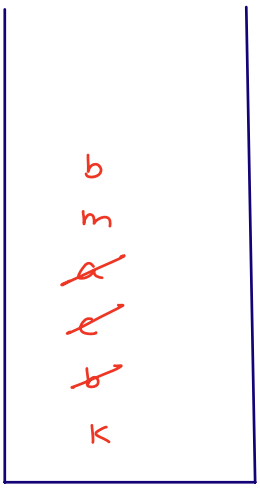


ca

ans = ac

dch = b

k b c a a a a a c b m b




```
for ( int i=0;  i < str.length() ;     ) {
    char ch= str.charAt(i);
    if (st.size() == 0 || st.peek() != ch) {
        st.push(ch);
        i++;
    }
    else {
        char dch = st.pop();
        while ( i < str.length() && str.charAt(i) == dch) {
            i++;
        }
    }
}
```

→ extra problem

// create final ans by travelling stack ⇒

Double character trouble

{ same consecutive chars can come only two times }

eg1. a b ~~c c~~ b a

a ~~b b~~ a

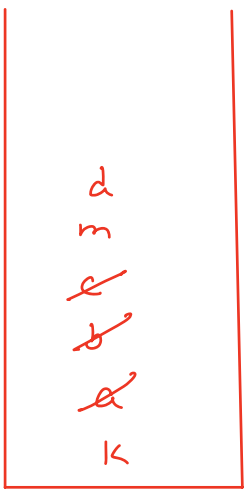
~~a a~~

.

eg2. k ~~a~~ ~~b~~ ~~c c~~ ~~b~~ ~~a~~ m d

d

```
for ( i -> 0 to str.length() ) {
    if ( st.size() == 0 || st.peek() != ch) {
        st.push();
    }
    else {
        st.pop();
    }
}
```



Expression Evaluation and conversion

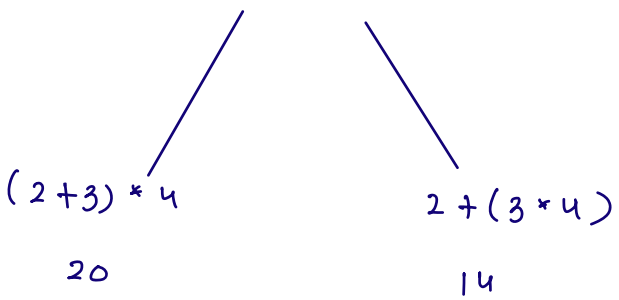
Infix expression — **Postfix expression** (or Reverse polish notation)

`2 + 3` — `2 3 +`

2, 3 → operands

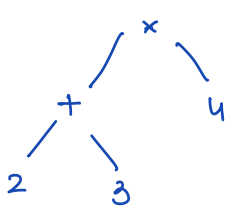
\+ → operator

Infix → 2 + 3 * 4 ⟹ * will work first

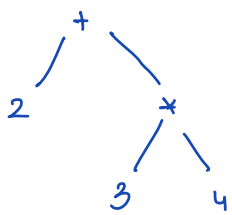


Postfix →

(2+3) * 4 → 2 3 + 4 *



2+(3*4) → 2 3 4 * +



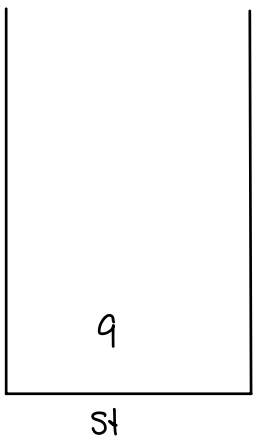
i) Postfix exp are easy to evaluate
- no brackets
- no operator priority

the order of operators and the their order of evaluation is same in postfix.

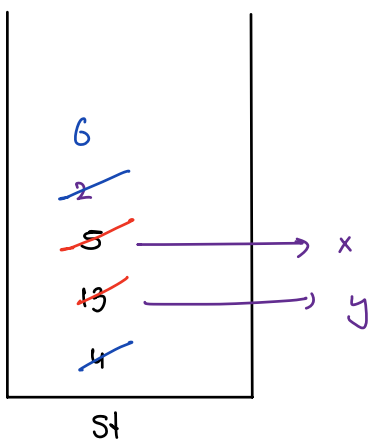
Given postfix expression, evaluate it and return final answer.

→ Postfix evaluation

eg1 ["2", "1", "+", "3", "*"]



eg2 ["4", "13", "5", "/", "+"]



y opr x

↓

operator

→ infix to postfix conversion

^ → power

+ , - , / , *

priority →

^

/ , *

+ , -

eg1 : "x^y/(a*z)+b"

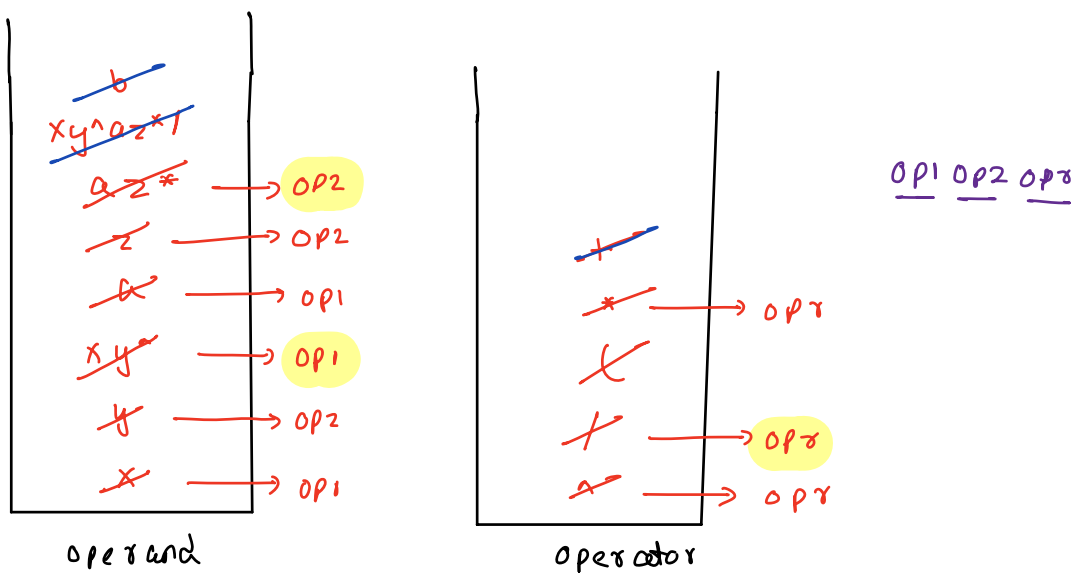
xy^ az*/ b+

eg2 : a+b*c

abc*+

eg1 : "x^y/(a*z)+b"

xy^az*/b+



op1 op2 opr

eg2 : "(a+b*c-d)+x/y*z"

Steps

if ch is operand then push it to operand stack

if ch is '(' then push it to operator stack

if ch is ')' then evaluate till '(' comes

if ch is operator then evalute till

operator.size() > 0

operator.peek() != '('

operator.peek() is having higher or equal priority than current opr.

"(a+b*c-d) + x/y*z"

Steps

if ch is operand then push it to operand stack

if ch is '(' then push it to operator stack

if ch is ')' then evaluate till '(' comes

if ch is operator then evaluate till

- operator.size() > 0
- operator.peek() != '('
- operator.peek() is having higher or equal priority than current opr.

then add curr opr

final evaluate