Evaluate Reverse polish notation.

This one is an expression based question

little bit intro to postfix and prefix expression

1) Postfix expression : Postfix expression also known as reverse polish notation is notation used by computer to evaluate expressions.

   In this the operands are followed by operator. Example : 2 3 + 4 *

2) Prefix expression : Prefix expression also known as polish notation is another type of notation used by computers to evaluate expressions.

Identification of data structure to solve the problem

- trying to solve the problem using brute force approach would be $O(n^2)$ as we need to perform '3' things

  1) To iterate the array (O(N))
  2) To perform operation on the array & O(N)
  3) To delete the elements on the indices where operation was performed. (O(N))

~~The Brute force Pseudo code is as follows but it with it is just abstract though.~~

But still as the re-iteration takes place at most 'n' times which might be less than n we can say loop will be executed n^2 times at max.

arr [strings] //array of strings ;

result ;

check = true ;
int n = arr.size() ;
while (check)

check = false ;
for i to n

if (arr[i] == "+" ||
arr[i] == "-" ||
arr[i] == "*" ||
arr[i] == "/")

{

switch (tokens[i][0]) {

case '+' :
arr[i] = to_string (stoi (arr [i-2])
+ stoi (arr [i-1]) ;

break ;

case '-' :
arr[i] = to-string (stoi (arr [i-2])
- stoi (arr [i-1]) ;

break ;

case '*' :
arr[i] = to_string (stoi (arr [i-2]) *
stoi (arr [i-1]) ;

break ;

case '/' :
arr[i] = to-string (stoi (arr [i-2])
/ stoi (
arr [i-1]) ;

break ;

2 3 + 4 *
0 1 2 3 4
→

2 + 3 = 5
"+"

5 4 *
5 * 4 = 9 ← result.

T: O(n²)

S: O(1)

```
            check = true;
            arr.erase(arr.begin()+i-2, token.begin()+i);
            n = tokens.size(); // new size
            break;
        }
    }
    return stoi(arr[arr.size()-1]);
}
```

removing the range (O(n))

As you might have seen this approach is similar to brute force used in valid parenthesis problem.

This is because it is expression / pattern matching with a LIFO style format where the last expression needs to be solved first before first one gets solved)

Optimisations

⇒

First way to optimize the code is to use a stack to track the result.

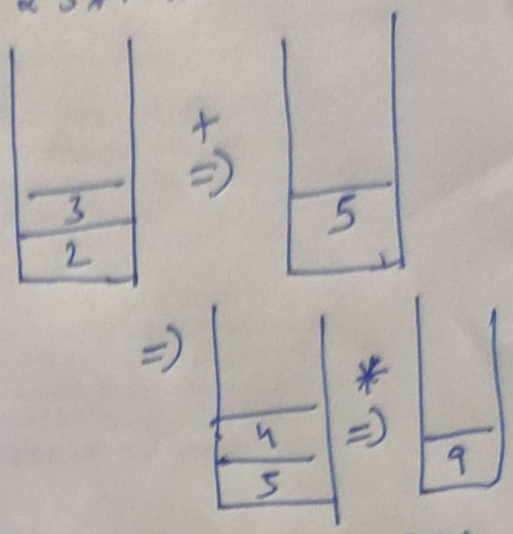
Changes to pseudocode:

~~public:~~

~~int evalRPN (vector<string~~

```
stack<int> result;
for (auto & it : tokens) {
    if (it != "+" && it != "-" && it != "/" && it != "*")
        result.push(stoi(it));
    else{
        int opr2 = result.top();
        result.pop();
        int opr1 = result.top();
        result.pop();
        if (it == "+") {
            result.push(opr1 + opr2);
        }
        else if (it == "-") {
            result.push(opr1 - opr2);
        }
        else if (it == "*") {
            result.push(opr1 * opr2);
        }
        else if (it == "/") {
            result.push(opr1 / opr2);
        }
    }
}
```

T: O(N)
S: O(N)



return result.top();

But here we can see we are resting too much more over we could have moved the check for operations to separate function for separation of concerns.

Thus new optimised code becomes:

```
class Solution {
  private:
      stack<int> result;
  public:
      int operation(int &opr1, int &opr2, string &opr)
      {   if (opr == "+")
              return opr1 + opr2;
          else if (opr == "-")
              return opr1 - opr2;
          else if (opr == "*")
              return opr1 * opr2;
          else if (opr == "/")
              return opr1 / opr2;
          else
              return NULL;
      }
```

```
int evalRPN(vector<string>& tokens) {
    for (auto& it : tokens) {
        if (it == "+" || it == "-" || it == "/" || it == "*")
        {   int opr2 = result.top();
            result.pop();
            int opr2 = result.ptop();
            result.pop();
            result.push(operation(opr1, opr2, it));
        }
        else {
            result.push(stoi(it));
        }
    }
    return result.top();
}
};
```