<https://leetcode.com/problems/evaluate-reverse-polish-notation/description/>

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

**Note** that:

The valid operators are '+', '-', '\*', and '/'.

Each operand may be an integer or another expression.

The division between two integers always truncates toward zero.

There will not be any division by zero.

The input represents a valid arithmetic expression in a reverse polish notation.

The answer and all the intermediate calculations can be represented in a 32-bit integer.

**Example 1:**

**Input:** tokens = ["2","1","+","3","\*"]

**Output:** 9

**Explanation:** ((2 + 1) \* 3) = 9

**Example 2:**

**Input:** tokens = ["4","13","5","/","+"]

**Output:** 6

**Explanation:** (4 + (13 / 5)) = 6

**Example 3:**

**Input:** tokens = ["10","6","9","3","+","-11","\*","/","\*","17","+","5","+"]

**Output:** 22

**Explanation:** ((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5= ((10 \* (6 / (12 \* -11))) + 17) + 5= ((10 \* (6 / -132)) + 17) + 5= ((10 \* 0) + 17) + 5= (0 + 17) + 5= 17 + 5= 22

**Constraints:**

1 <= tokens.length <= 10^4

tokens[i] is either an operator: "+", "-", "\*", or "/", or an integer in the range [-200, 200].

**Attempt 1: 2024-07-05**

**Solution 1: Stack (10 min)**

class Solution {

    public int evalRPN(String[] tokens) {

        Stack<Integer> stack = new Stack<>();

        for(String token : tokens) {

            if(token.equals("+")) {

                int a = stack.pop();

                int b = stack.pop();

                stack.push(a + b);

            } else if(token.equals("-")) {

                int a = stack.pop();

                int b = stack.pop();

                stack.push(b - a);

            } else if(token.equals("\*")) {

                int a = stack.pop();

                int b = stack.pop();

                stack.push(a \* b);

            } else if(token.equals("/")) {

                int a = stack.pop();

                int b = stack.pop();

                stack.push(b / a);

            } else {

                stack.push(Integer.parseInt(token));

            }

        }

        return stack.pop();

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**What is Reverse Polish Notation ?**

**Refer to**

<https://mathworld.wolfram.com/ReversePolishNotation.html>

Reverse Polish notation (RPN) is a method for representing expressions in which the operator symbol is placed after the arguments being operated on. Polish notation, in which the operator comes before the operands, was invented in the 1920s by the Polish mathematician Jan Lucasiewicz. In the late 1950s, Australian philosopher and computer scientist Charles L. Hamblin suggested placing the operator after the operands and hence created reverse polish notation.

For example, the following RPN expression will produce the sum of 2 and 3, namely 5: 2 3 +.

Reverse Polish notation, also known as postfix notation, contrasts with the "infix notation" of standard arithmetic expressions in which the operator symbol appears between the operands.

RPN has the property that brackets are not required to represent the order of evaluation or grouping of the terms. RPN expressions are simply evaluated from left to right and this greatly simplifies the computation of the expression within computer programs. As an example, the arithmetic expression (3+4)×5 can be expressed in RPN as 3 4 + 5 ×.

In practice RPN can be conveniently evaluated using a stack structure. Reading the expression from left to right, the following operations are performed:

If a value appears next in the expression, push this value on to the stack.

If an operator appears next, pop two items from the top of the stack and push the result of the operation on to the stack.

A standard infix arithmetic expression can be converted to an RPN expression using a parsing algorithm as a recursive descent parse.

RPN is used in Hewlett Packard and some Texas Instruments calculators and internally in some computer languages.

**Refer to**

<https://leetcode.com/problems/evaluate-reverse-polish-notation/solutions/47430/java-accepted-code-stack-implementation/>

The Reverse Polish Notation is a stack of operations, thus, I decided to use java.util.Stack to solve this problem. As you can see, I add every token as an integer in the stack, unless it's an operation. In that case, I pop two elements from the stack and then save the result back to it. After all operations are done through, the remaining element in the stack will be the result.

import java.util.Stack;

public class Solution {

public int evalRPN(String[] tokens) {

int a,b;

Stack<Integer> S = new Stack<Integer>();

for (String s : tokens) {

if(s.equals("+")) {

S.add(S.pop()+S.pop());

}

else if(s.equals("/")) {

b = S.pop();

a = S.pop();

S.add(a / b);

}

else if(s.equals("\*")) {

S.add(S.pop() \* S.pop());

}

else if(s.equals("-")) {

b = S.pop();

a = S.pop();

S.add(a - b);

}

else {

S.add(Integer.parseInt(s));

}

}

return S.pop();

}

}

**Refer to**

<https://leetcode.com/problems/evaluate-reverse-polish-notation/solutions/47430/java-accepted-code-stack-implementation/comments/180708>

**后缀表达式的计算方法：遍历整个表达式，如果为数字则入栈，如果为符号则将前面两个数字出栈，先出栈的在右边后出栈的在左边符号放中间算出来结果再扔到栈中即可。**（loop through the whole expression, if number then push, if operator then pop two numbers, calc the result with the operator, push res back to stack）

// 这不就是常考点之一的计算后缀表达式吗

class Solution {

public int evalRPN(String[] tokens) {

Deque<Integer> deque = new LinkedList<>();

for (String s: tokens) {

if (s.equals("\*") || s.equals("/") || s.equals("+") || s.equals("-")) {

int right = deque.removeFirst();

int left = deque.removeFirst();

if (s.equals("\*")) {

deque.addFirst(left \* right);

}

if (s.equals("/")) {

deque.addFirst(left / right);

}

if (s.equals("+")) {

deque.addFirst(left + right);

}

if (s.equals("-")) {

deque.addFirst(left - right);

}

} else {

deque.addFirst(Integer.valueOf(s));

}

}

return deque.peekFirst();

}

}

**Refer to**

<https://algo.monster/liteproblems/150>

**Problem Description**

You are asked to evaluate an arithmetic expression provided as an array of strings, tokens, which uses Reverse Polish Notation (RPN). This notation places the operator after the operands. For example, the expression "3 4 +" in RPN is equivalent to "3 + 4" in standard notation. Your task is to calculate the result of the expression and return the resulting integer value.

Several points to consider for this problem are:

The expression only contains the operators +, -, \*, and /.

Operands could be either integers or sub-expressions.

When performing division, the result is always truncated towards zero.

The expression does not contain any division by zero.

The expression given is always valid and can be evaluated without error.

The result of the evaluated expression and any intermediate operations will fit within a 32-bit integer.

**Intuition**

To solve this problem, we need to understand the nature of Reverse Polish Notation. In RPN, every time we encounter an operator, it applies to the last two operands that were seen. A stack is the perfect data structure for this evaluation because it allows us to keep track of the operands as they appear and then apply the operators in the correct order.

The intuition for the solution is as follows:

We iterate through each string (token) in the tokens array.

If the token is a number (single digit or multiple digits), we convert it to an integer and push it onto a stack.

If the token is an operator, we pop the last two numbers from the stack and apply this operator; these two numbers are the operands for the operator.

The result of the operation is then pushed back onto the stack.

After applying an operator, the stack should be in a state that is ready to process the next token.

When we've processed all the tokens, the stack will contain only one element, which is the final result of the expression.

Division in Python by default results in a floating-point number. Since the problem states that the division result should truncate toward zero, we explicitly convert the result to an int, which discards the decimal part.

The key here is to iterate through the tokens once and perform operations in order, ensuring the stack's top two elements are the operands for any operator we come across.

**Solution Approach**

The solution makes use of a very simple yet powerful algorithm that utilizes a stack data structure to process the given tokens one by one. Here are the steps it follows:

Initialize an empty list nums that will act as a stack to store the operands.

Iterate over each token in the tokens array.

**If the token is a numeric value (identified by either being a digit or having more than one character, which accounts for numbers like "-2"), we convert the token to an integer (Integer.parseInt will help to keep the negative operator) and push it onto the stack.**

If the token is an operator (+, -, \*, /), we perform the following:

Pop the top two elements from the stack. Since the last element added is at the top of the stack, we'll refer to these as the second operand (at nums[-1]) and the first operand (at nums[-2]) in that order.

Apply the operator to these two operands. For addition, subtraction, and multiplication, this is straightforward.

For division, we apply integer division which is the same as dividing and then applying the int function to truncate towards zero. This is important as it handles the truncation towards zero for negative numbers correctly. The simple floor division operator // in Python truncates towards negative infinity, which can give incorrect results for negative quotients.

The result of the operation is then placed back into the stack at the position of the first operand (nums[-2]).

The second operand (nums[-1]), which has already been used, is removed from the stack.

After processing all the tokens, there should be a single element left in the nums stack. This element is the result of evaluating the expression.

The algorithm used here is particularly efficient because it has a linear time complexity, processing each token exactly once. The space complexity of this approach is also linear, as it depends on the number of tokens that are pushed into the stack. The use of the stack ensures that the operands for any operator are always readily available at the top of the stack.

Here is a snippet of how the arithmetic operations are processed:

Addition: nums[-2] += nums[-1]

Subtraction: nums[-2] -= nums[-1]

Multiplication: nums[-2] \*= nums[-1]

Division: nums[-2] = int(nums[-2] / nums[-1])

Once finished, the program returns nums[0] as the result of the expression.

**Example Walkthrough**

Let's use the following RPN expression as our example: "2 1 + 3 \*" which, in standard notation, translates to (2 + 1) \* 3.

By following the solution approach:

We initialize an empty list nums to serve as our stack: nums = [].

We iterate through the tokens: ["2", "1", "+", "3", "\*"].

a. The first token is "2", which is a number. We push it onto the stack: nums = [2].

b. The second token is "1", which is also a number. We push it onto the stack: nums = [2, 1].

c. The third token is "+", which is an operator. We need to pop the top two numbers and apply the operator: - We pop the first operand: secondOperand = nums.pop(), which is 1. Now nums = [2]. - We pop the second operand: firstOperand = nums.pop(), which is 2. Now nums = []. - We add the two operands: stackResult = firstOperand + secondOperand, which equals 2 + 1 = 3. - We push the result onto the stack: nums = [3].

d. The fourth token is "3", a number. We push it onto the stack: nums = [3, 3].

e. The fifth token is "\*", an operator: - We pop the second operand, secondOperand = nums.pop(), which is 3, leaving nums = [3]. - We pop the first operand, firstOperand = nums.pop(), which is 3, leaving nums = []. - We multiply the two operands: stackResult = firstOperand \* secondOperand, which equals 3 \* 3 = 9. - We push the result onto the stack: nums = [9].

After processing all the tokens, we are left with a single element in our stack nums = [9], which is our result.

So, the given RPN expression "2 1 + 3 \*" evaluates to 9. Thus, the function would return 9 as the result of the expression.

**Solution Implementation**

class Solution {

public int evalRPN(String[] tokens) {

Deque<Integer> stack = new ArrayDeque<>(); // Create a stack to hold integer values

// Iterate over each token in the input array

for (String token : tokens) {

// Check if the token is a number (either single digit or multi-digit)

if (token.length() > 1 || Character.isDigit(token.charAt(0))) {

// Push the number onto the stack

stack.push(Integer.parseInt(token));

} else {

// Pop the top two elements for the operator

int secondOperand = stack.pop();

int firstOperand = stack.pop();

// Apply the operator on the two operands based on the token

switch (token) {

case "+":

stack.push(firstOperand + secondOperand); // Add and push the result

break;

case "-":

stack.push(firstOperand - secondOperand); // Subtract and push the result

break;

case "\*":

stack.push(firstOperand \* secondOperand); // Multiply and push the result

break;

case "/":

stack.push(firstOperand / secondOperand); // Divide and push the result

break;

}

}

}

// The final result is the only element in the stack, return it

return stack.pop();

}

}

**Time and Space Complexity**

The given Python function evalRPN evaluates Reverse Polish Notation (RPN) expressions. The time complexity and space complexity analysis are as follows:

**Time Complexity**

The time complexity of this function is O(n), where n is the number of tokens in the input list tokens. This is because the function iterates through each token exactly once. Each operation within the loop, including arithmetic operations and stack operations (append and pop), can be considered to have constant time complexity, O(1).

**Space Complexity**

The space complexity of the code is O(n) in the worst case, where n is the number of tokens in the list. This worst-case scenario occurs when all tokens are numbers and are thus pushed onto the stack. In the best-case scenario, where the input is balanced with numbers and operators, the space complexity could be better than O(n), but the upper bound remains O(n). The auxiliary space required is for the nums stack used to perform the calculations. There are no other data structures that use significant memory.

**Refer to**

[L772.Basic Calculator III (Ref.L224,L227)](note://WEB8b3c61c712e02a5dd91a90fb4ed6c3b8)