Evaluate Reverse Polish Notation

Medium

Created	@2025年4月29日 上午11:59
Question Type	Stack
Difficulty	Medium
LeetCode Question Link	https://leetcode.com/problems/evaluate-reverse-polish-notation/

1. Question Self-understanding:

1.1 Description:

Based on my understanding, we want to implement an algorithm to evaluate arithmetic expressions written in Reverse Polish Notation.

1.2 Input:

The input is a list of strings, each representing either a number or an operator.

1.3 Input Assumption

We can assume that we will never be given a sequence that results in division by zero.

1.4 Output:

The output should be an integer.

1.5 Example:

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

1.6 Other Q&A:

- **How should we handle division since the output must be an integer?**

  We will perform integer division by rounding down the result (i.e., truncate toward zero).

- **Can we assume the list length is always greater than 0?**

  Yes.

2. Attempt 1:

2.1 Thought:

- To solve this question, we first need to understand Reverse Polish Notation (RPN). For example, if we want to express x+y, in RPN we would write it as x y +.This suggests that when we encounter an operator, such as "+", "-", "*", or "/", we need to **look back** at the previous two numbers. It makes me think that we should focus on the idea of "reversing" — that is, handling the previous two numbers once we see an operator.

  To manage this, we want to store the numbers somewhere, and this leads me to believe that using a **stack** is the best approach. Every time we encounter a number, we push it onto the stack. When we see an operator, we **pop** the top two numbers from the stack, perform the operation, and then **push** the result back onto the stack. We continue this process until we reach the end of the input.

  At the end, the only number remaining on the stack will be our final result.

2.2 Pseudo-Code: (Ignore this part. It’s a draft for brainstorming.)

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        # storage for numbers
        stack = []

        # for each item in tokens:
        #   if it's a number, push it onto the stack
        #   if it's an operator, pop the top two values, evaluate them, and push the re
        # return the top element in the stack
```

2.3 Implementation through python:

```
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        # Initialize an empty stack to store the numbers
        stack = []

        # go through each token in the input list
        for token in tokens:
            if token not in "+-*/":

                stack.append(int(token))
                continue

            top = stack.pop()  # pop the top element from the stack
            second_top = stack.pop()

            if token == "+":

                stack.append(second_top + top)

            elif token == "-":

                stack.append(second_top - top)

            elif token == "*":

                stack.append(second_top * top)

            else:

                # Perform integer division and truncate towards zero
                stack.append(int(second_top / top))
```

```
    return stack[0]  # The result will be the only element left in the stack
```

2.4 Time Complexity and Space Complexity

2.4.1 Time Complexity:

- Each is processed at most twice—once when pushed, once when popped—resulting in $O(2 * n) \in O(n)$.

2.4.2 Space Complexity:

- We store up to n values on the stack in the worst case, so the space complexity is $O(n)$ space.