Previous Next

224. Basic Calculator

June 24, 2019 | 47.7K views

Average Rating: 4.55 (44 votes)

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, **non-negative** integers and empty spaces ` `.

Example 1:

```
Input: "1 + 1"
Output: 2
```

Example 2:

```
Input: " 2-1 + 2 "
Output: 3
```

Example 3:

```
Input: "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

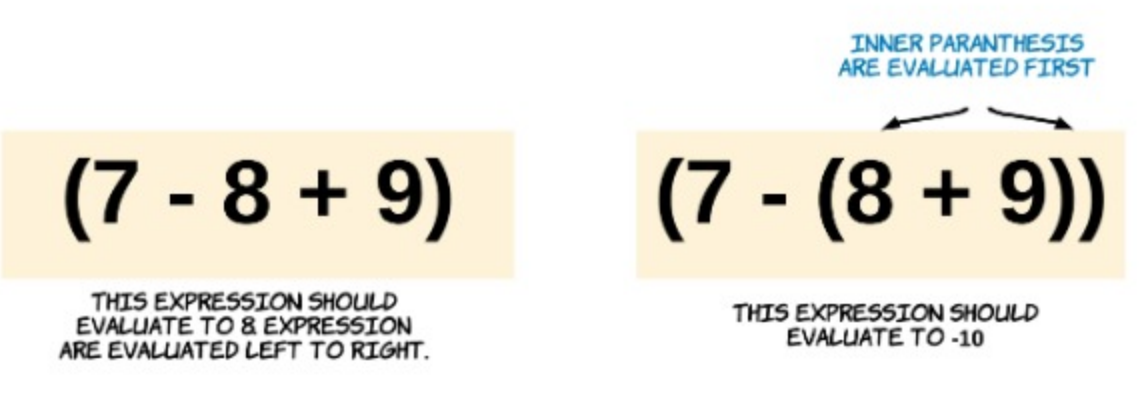
Note:

- You may assume that the given expression is always valid.
- **Do not** use the `eval` built-in library function.

Solution

This problem is all about understanding the following:

- Input always contains valid strings
- The rules of addition and subtraction
- Implication of precedence by parenthesis
- Spaces do not affect the evaluation of the input expression

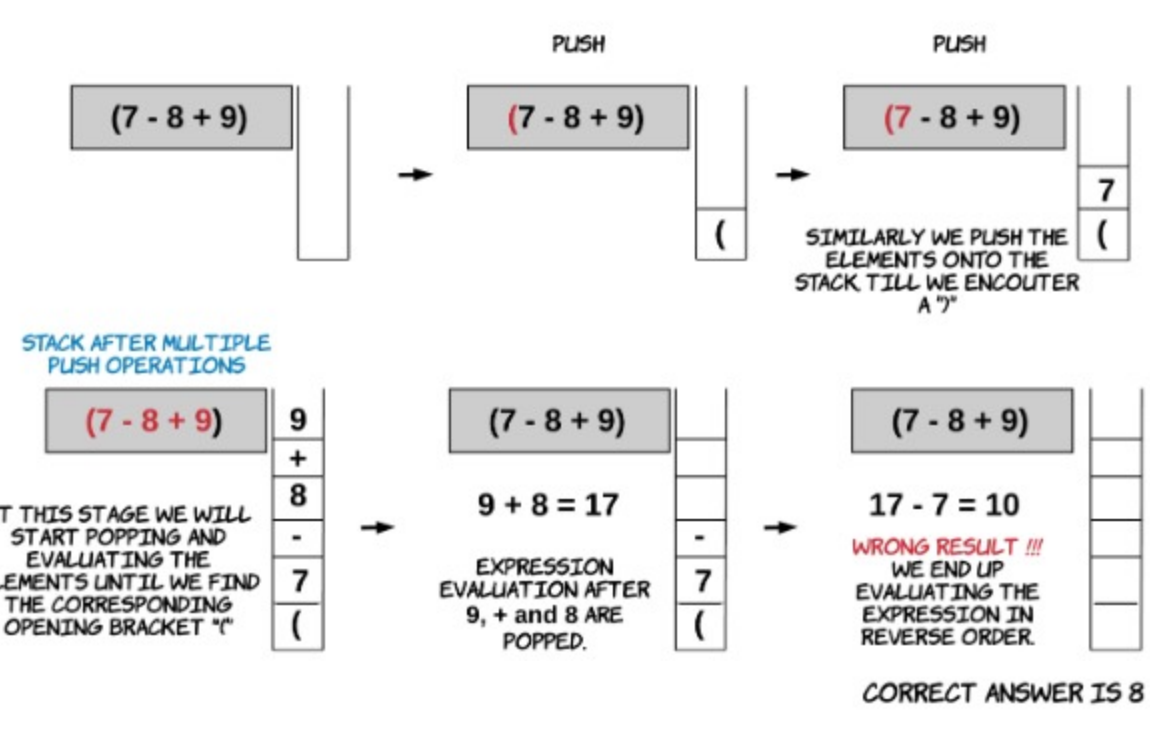


Approach 1: Stack and String Reversal

Intuition

This question qualifies really well for a stack question. Since the expression might have parenthesis, we can use a stack to find the value for each sub-expression within a parenthesis. Essentially, we need to *delay* processing the main expression until we are done evaluating the interim sub-expressions within parenthesis and to introduce this delay, we use a stack.

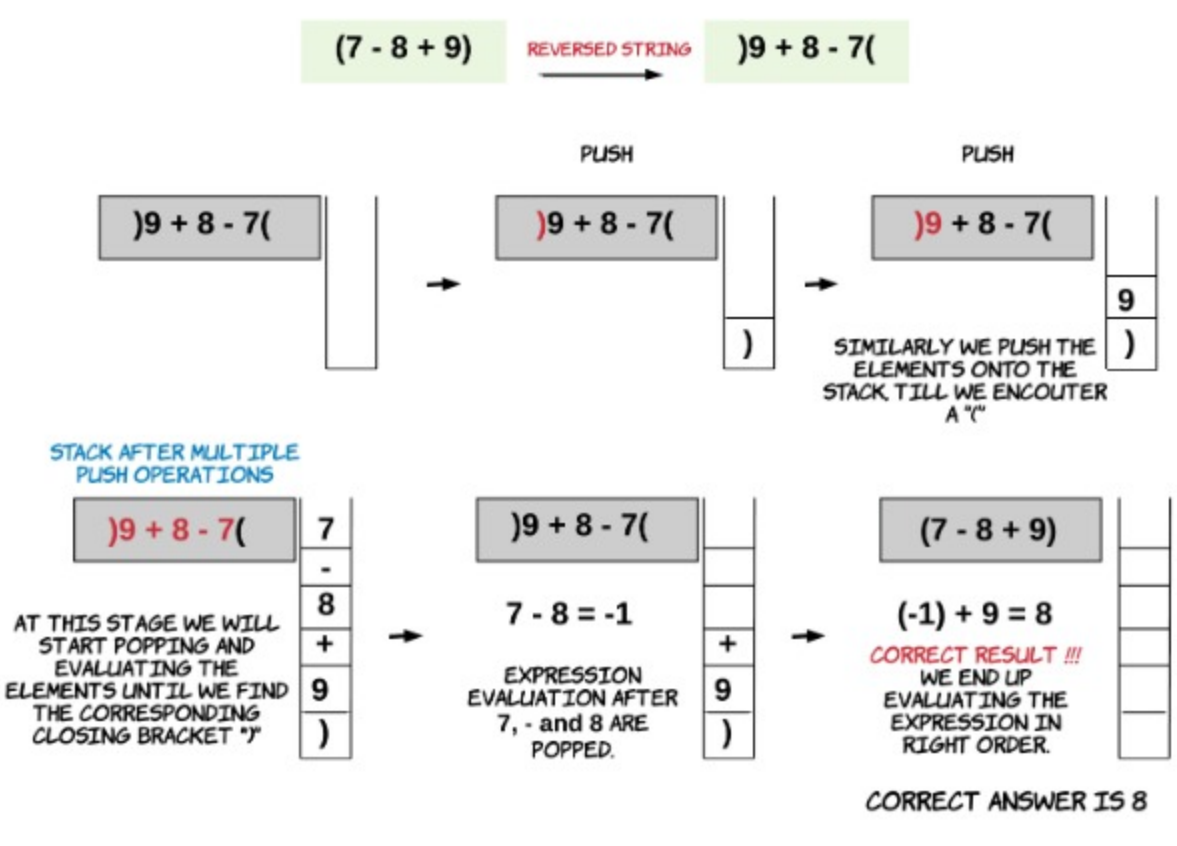
We push the elements of the expression one by one onto the stack until we get a closing bracket `)`. Then we pop the elements from the stack one by one and evaluate the expression on-the-go. This is done till we find the corresponding `(` opening bracket. This kind of evaluation is very common when using the stack data structure. However, if you notice the way we calculate the final answer, you will realize that we actually process the values from right to left whereas it should be the other way around.



From the above example we realize that following the simple stack push and pop methodology will not help us here. We need to understand how `+` and `-` work. `+` follows the associative property. For the expression $A + B + C$, we have $(A + B) + C = A + (B + C)$. However, `-` does not follow this rule which is the root cause of all the problems in this approach.

If we use a stack and read the elements of the expression from left to right, we end up evaluating the expression from right-to-left. This means we are expecting $(A - B) - C$ to be equal to $(C - B) - A$ which is not correct. Subtraction is neither associative nor commutative.

This problem could be solved very easily by reversing the string and then using basic drill using a stack. Reversing a string helps since we now put the elements of the expression into the stack from right to left and evaluation for the expression is done correctly from left to right.



Algorithm

1. Iterate the expression string in reverse order one character at a time. Since we are reading the expression character by character, we need to be careful when we are reading digits and non-digits.
2. The operands could be formed by multiple characters. A string "123" would mean a numeric 123, which could be formed as: `123` >> `120 + 3` >> `100 + 20 + 3`. Thus, if the character read is a digit we need to form the operand by multiplying a power of `10` to the current digit and adding it to the overall operand. We do this since we are processing the string in the reverse order.
3. The operands could be formed by multiple characters. We need to keep track of an on-going operand. This part is a bit tricky since in this case the string is reversed. Once we encounter a character which is not a digit, we push the operand onto the stack.
4. When we encounter an opening parenthesis `(`, this means an expression just ended. Recall we have reversed the expression. So an opening bracket would signify the end of the an expression. This calls for evaluation of the expression by popping operands and operators off the stack till we pop corresponding closing parenthesis. The final result of the expression is pushed back onto the stack.

   > Note: We are evaluating all the sub-expressions within the main expression. The sub-expressions on the right get evaluated first but the main expression itself is evaluated from left to right when all its components are resolved, which is very important for correct results.

   For eg. For expression $A - (B + C) + (D + E - F)$, $D + E - F$ is evaluated before $B + C$. While evaluating $D + E - F$ the order is from left to right. Similarly for the parent expression, all the child components are evaluated and stored on the stack so that final evaluation is left to right.

5. Push the other non-digits onto to the stack.

6. Do this until we get the final result. It's possible that we don't have any more characters left to process but the stack is still non-empty. This would happen when the main expression is not enclosed by parenthesis. So, once we are done evaluating the entire expression, we check if the stack is non-empty. If it is, we treat the elements in it as one final expression and evaluate it the same way we would if we had encountered an opening bracket.

   We can also cover the original expression with a set of parenthesis to avoid this extra call.

Java | Python | Copy

```python
class Solution:

    def evaluate_expr(self, stack):

        res = stack.pop() if stack else 0

        # Evaluate the expression till we get corresponding ')'
        while stack and stack[-1] != ')':
            sign = stack.pop()
            if sign == '+':
                res += stack.pop()
            else:
                res -= stack.pop()
        return res

    def calculate(self, s: str) -> int:

        stack = []
        n, operand = 0, 0

        for i in range(len(s) - 1, -1, -1):
            ch = s[i]

            if ch.isdigit():

                # Forming the operand - in reverse order.
                operand = (10**n * int(ch)) + operand
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of the string.
- Space Complexity: $O(N)$, where N is the length of the string.

Approach 2: Stack and No String Reversal

Intuition

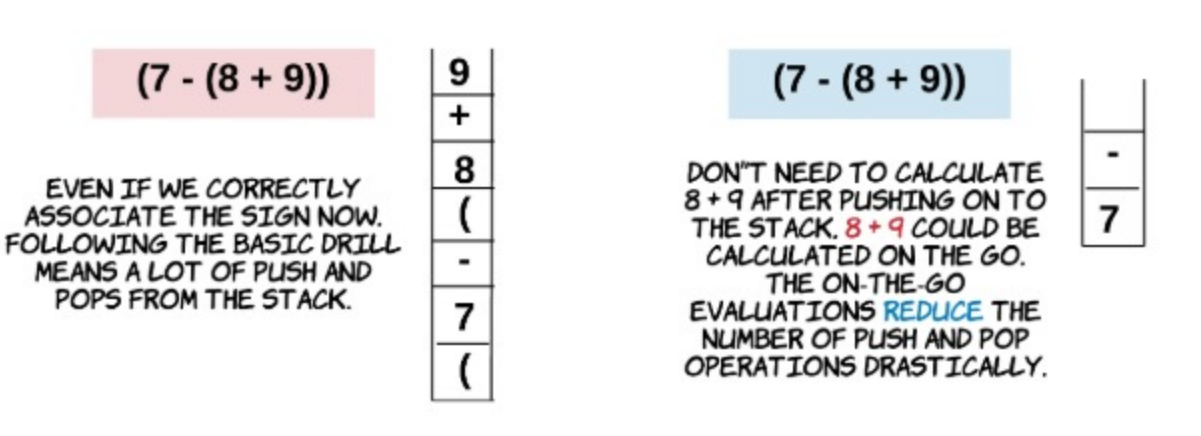
A very easy way to solve the problem of associativity for `-` we tackled in the previous approach, is to use `-` operator as the magnitude for the operand to the right of the operator. Once we start using `-` as a magnitude for the operands, we just have one operator left which is addition and `+` is associative.

for e.g. $A - B - C$ could be re-written as $A + (-B) + (-C)$.

> The re-written expression would follow associativity rule. Thus evaluating the expression from right or left, won't change the result.

What we need to keep in mind is that the expressions given would be complicated, i.e. there would be expressions nested within other expressions. Even if we have something like `(A - (B - C))` we need to associate the `negative sign` outside of `B-C` with the result of `B-C` instead of just with `B`.

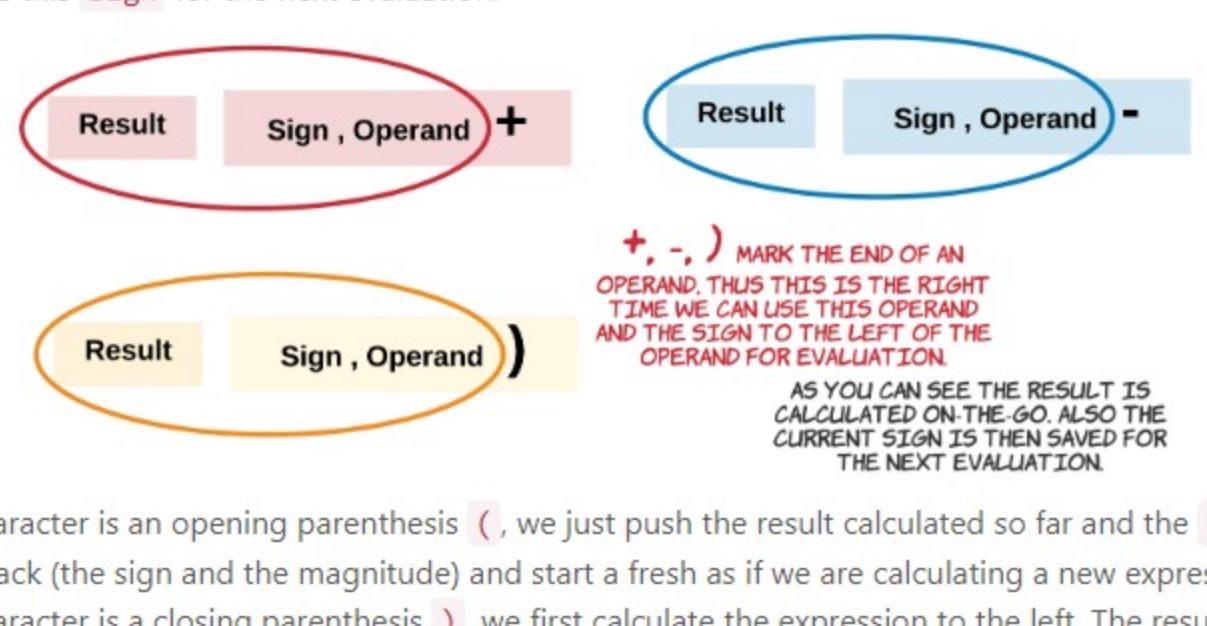
We can solve this problem by following the basic drill before and associating the sign with the expression to the right of it. However, the approach that we will instead take has a small twist to it in that we will be evaluating most of the expression on-the-go. This reduces the number of push and pop operations.



Follow the below steps closely. This algorithm is inspired from discussion post by southpenguin.

Algorithm

1. Iterate the expression string one character at a time. Since we are reading the expression character by character, we need to be careful when we are reading digits and non-digits.
2. The operands could be formed by multiple characters. A string "123" would mean a numeric 123, which could be formed as: `123` >> `120 + 3` >> `100 + 20 + 3`. Thus, if the character read is a digit we need to form the operand by multiplying `10` to the previously formed continuing operand and adding the digit to it.
3. Whenever we encounter an operator such as `+` or `-` we first evaluate the expression to the left and then save this `sign` for the next evaluation.



4. If the character is an opening parenthesis `(`, we just push the result calculated so far and the `sign` on to the stack (the sign and the magnitude) and start a fresh as if we are calculating a new expression.
5. If the character is a closing parenthesis `)`, we first calculate the expression to the left. The result from this would be the result of the expression within the set of parenthesis that just concluded. This result is then multiplied with the sign, if there is any on top of the stack. Remember we saved the `sign` on top of the stack when we had encountered an open parenthesis? This sign is associated with the parenthesis that started then, thus when the expression ends or concludes, we pop the `sign` and multiply it with result of the expression. It is then just added to the next element on top of the stack.

Java | Python | Copy

```python
class Solution:
    def calculate(self, s: str) -> int:

        stack = []
        operand = 0
        res = 0 # For the on-going result
        sign = 1 # 1 means positive, -1 means negative

        for ch in s:
            if ch.isdigit():

                # Forming operand, since it could be more than one digit
                operand = (operand * 10) + int(ch)

            elif ch == '+':

                # Evaluate the expression to the left,
                # with result, sign, operand
                res += sign * operand

                # Save the recently encountered '+' sign
                sign = 1

                # Reset operand
                operand = 0

            elif ch == '-':
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of the string. The difference in time complexity between this approach and the previous one is that every character in this approach will get processed exactly once. However, in the previous approach, each character can potentially get processed twice, once when it's pushed onto the stack and once when it's popped for processing of the final result (or a subexpression). That's why this approach is faster.
- Space Complexity: $O(N)$, where N is the length of the string.

Rate this article:

Previous Next

Comments: 18 Sort By

Type comment here... (Markdown is supported)

Preview Post

ouchxp ★ 60 January 30, 2020 3:50 PM

I was asked with two follow up questions, hope these will help.

1. Support a ternary operator like in Java.
2. A method to print the expression string that identicaly to the input, preserves all the formats like extra spaces. Simply store the original input is not allowed.

Read More

23 Share Reply

SHOW 1 REPLY

Frank_Chen_CMU ★ 21 July 5, 2019 11:15 PM

Great explanation. And the graph is also very fascinating.

6 Share Reply

kelvinsilva111 ★ 4 December 2, 2019 1:26 AM

https://en.wikipedia.org/wiki/Shunting-yard_algorithm

3 Share Reply

SHOW 1 REPLY

dhrubo ★ 9 June 30, 2019 9:09 AM

very nice explanation. Thank you

2 Share Reply

rahultimate ★ 2 October 14, 2019 10:14 AM

learn lr parsing lmao

2 Share Reply

SHOW 1 REPLY

tngo0508 ★ 1 November 17, 2019 4:43 AM

For approach 1, I tried test case : "- (3 + (4 + 5))" and it failed.

1 Share Reply

SHOW 2 REPLIES

eefiasfira ★ 595 November 15, 2019 7:45 AM

Here's another version of Approach 2 using recursion. This runs faster than the stack based approach (beats 100%) and IMO is cleaner. The problem with my approach is, it requires the current index in the string to be shared across recursive calls (which can be fixed by using a separate class to hold the index and string across the recursive stack).

Read More

1 Share Reply

user8176K ★ 1 October 25, 2019 11:27 AM

Why are we not using queue?

1 Share Reply

SHOW 1 REPLY

aparajit ★ 7 July 3, 2020 1:28 AM

you can also check this out:
https://www.youtube.com/watch?v=vq-nUF0G4fI

Read More

0 Share Reply

pegasusflyboy ★ 1 July 2, 2020 8:42 PM

In the first solution, why do we check if the stack is empty within evaluateExpr? Is there ever a way where it could be empty after calling evaluateExpr?

0 Share Reply

‹ 1 2 ›