Basic Calculator

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 .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

Note: Do not use the `eval` built-in library function.

Solution 1

Simple iterative solution by identifying characters one by one. One important thing is that the input is valid, which means the parentheses are always paired and in order. Only 5 possible input we need to pay attention:

1. digit: it should be one digit from the current number
2. '+': number is over, we can add the previous number and start a new number
3. '-': same as above
4. '(': push the previous result and the sign into the stack, set result to 0, just calculate the new result within the parenthesis.
5. ')': pop out the top two numbers from stack, first one is the sign before this pair of parenthesis, second is the temporary result before this pair of parenthesis. We add them together.

Finally if there is only one number, from the above solution, we haven't add the number to the result, so we do a check see if the number is zero.

```java
public int calculate(String s) {
    Stack<Integer> stack = new Stack<Integer>();
    int result = 0;
    int number = 0;
    int sign = 1;
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if(Character.isDigit(c)){
            number = 10 * number + (int)(c - '0');
        }else if(c == '+'){
            result += sign * number;
            number = 0;
            sign = 1;
        }else if(c == '-'){
            result += sign * number;
            number = 0;
            sign = -1;
        }else if(c == '('){
            //we push the result first, then sign;
            stack.push(result);
            stack.push(sign);
            //reset the sign and result for the value in the parenthesis
            sign = 1;
            result = 0;
        }else if(c == ')'){
            result += sign * number;
            number = 0;
            result *= stack.pop();    //stack.pop() is the sign before the parent
hesis
            result += stack.pop();   //stack.pop() now is the result calculated b
efore the parenthesis

        }
    }
    if(number != 0) result += sign * number;
    return result;
}
```

written by southpenguin original link here

Solution 2

```
class Solution {
public:
    int calculate(string s) {
        stack <int> nums, ops;
        int num = 0;
        int rst = 0;
        int sign = 1;
        for (char c : s) {
            if (isdigit(c)) {
                num = num * 10 + c - '0';
            }
            else {
                rst += sign * num;
                num = 0;
                if (c == '+') sign = 1;
                if (c == '-') sign = -1;
                if (c == '(') {
                    nums.push(rst);
                    ops.push(sign);
                    rst = 0;
                    sign = 1;
                }
                if (c == ')' && ops.size()) {
                    rst = ops.top() * rst + nums.top();
                    ops.pop(); nums.pop();
                }
            }
        }
        rst += sign * num;
        return rst;
    }
};
```

written by d40a original link here

Solution 3

My approach is based on the fact that the final arithmetic operation on each number is not only depend on the sign directly operating on it, but all signs associated with each unmatched '(' before that number.

e.g. 5 - (6 + (4 - 7)), if we remove all parentheses, the expression becomes 5 - 6 - 4 + 7.

sign:

6: (-1)(1) = -1

4: (-1)(1)(1) = -1

7: (-1)(1)(-1) = 1

The effect of associated signs are cumulative, stack is builded based on this. Any improvement is welcome.

```
public int calculate(String s) {
    Deque<Integer> stack = new LinkedList<>();
    int rs = 0;
    int sign = 1;
    stack.push(1);
    for (int i = 0; i < s.length(); i++){
        if (s.charAt(i) == ' ') continue;
        else if (s.charAt(i) == '('){
            stack.push(stack.peekFirst() * sign);
            sign = 1;
        }
        else if (s.charAt(i) == ')') stack.pop();
        else if (s.charAt(i) == '+') sign = 1;
        else if (s.charAt(i) == '-') sign = -1;
        else{
            int temp = s.charAt(i) - '0';
            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1)))
                temp = temp * 10 + s.charAt(++i) - '0';
            rs += sign * stack.peekFirst() * temp;
        }
    }
    return rs;
}
```

written by bidp original link here