EDA1

Aula prática **#2**

Stacks (continuação):

1. Resolva todos os exercícios da aula prática anterior, isto é, deve ter uma implementação de stacks funcional e deve usar essa implementação para resolver o exercício do equilíbrio dos parêntesis.

2. Foi dado nas aulas teóricas um algoritmo que, fazendo uso duma stack, lhe permite fazer a avaliação de expressões em postfix. Use o algoritmo (no papel), para fazer a avaliação das expressões que se seguem, apresentando a stack em todos os momentos da avaliação.

   a) **23 56 - 3 * 1 3 4 / + /**
   b) **2 4 7 32 / + 7 5 - 3 * - -**
   c) **2 6 + 4 7 * 5 - / 6 9 / * 4 9 * 5 3 / + -**

3. Considere o algoritmo de conversão infix-postfix dado nas aulas teóricas. Considere que ***input*** corresponde à string lida e a conversão da expressão lida é armazenada na variável ***output***. Assuma também que o ***input*** é analisado token a token, sendo um token a substring obtida de input, entre espaços. Sendo ***input*** = 2 * 8 + (10 + 5 * (20 / 4)) / (3 + -8)
   a) Qual o 5º token lido?
   b) Qual o ***output*** da conversão para postfix da expressão da variável ***input***, quando o token lido for a segundo ")"?
   c) Após a leitura do último ")", qual o conteúdo da stack? Desenhe a stack, indicando o seu topo.
   d) Qual a conversão para postfix da expressão lida?
   e) Usando a expressão obtida na alínea anterior, desenhe a stack de avaliação da expressão em postfix, quando for lido o token "-8".

4. Determine a complexidade das operações da Stack, justificando.

Aula prática **#2**

→ 5. Considere definida em C a função **q1**:

```
int q1(long n){
    long c=n;
    Stack S=CreateStack(20);
    while (n>0){
        Push(n%10,S);
        n=n/10;
    }
    while(!IsEmpty(S)){
        if(Pop(S)!=c%10)
            return 0;
        c=c/10;
    }
    return 1;
}
```

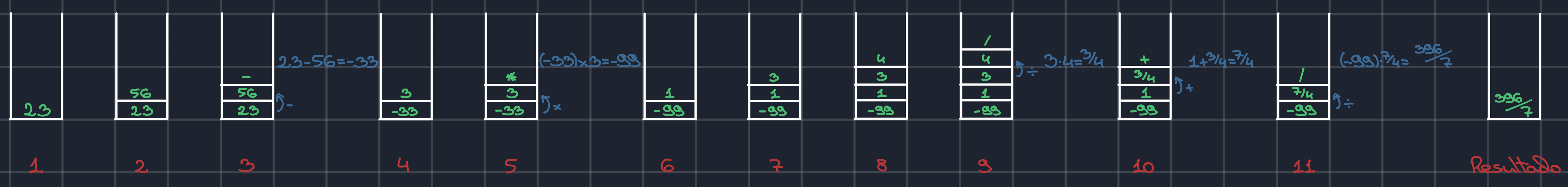
a) Qual o resultado da execução de **q1**(541545145)?

b) Como se apresenta a stack S na penúltima iteração do 1º ciclo durante a execução da alínea anterior? Desenhe a stack, não esquecendo de identificar o seu topo.

c) Qual a complexidade da função **q1**?

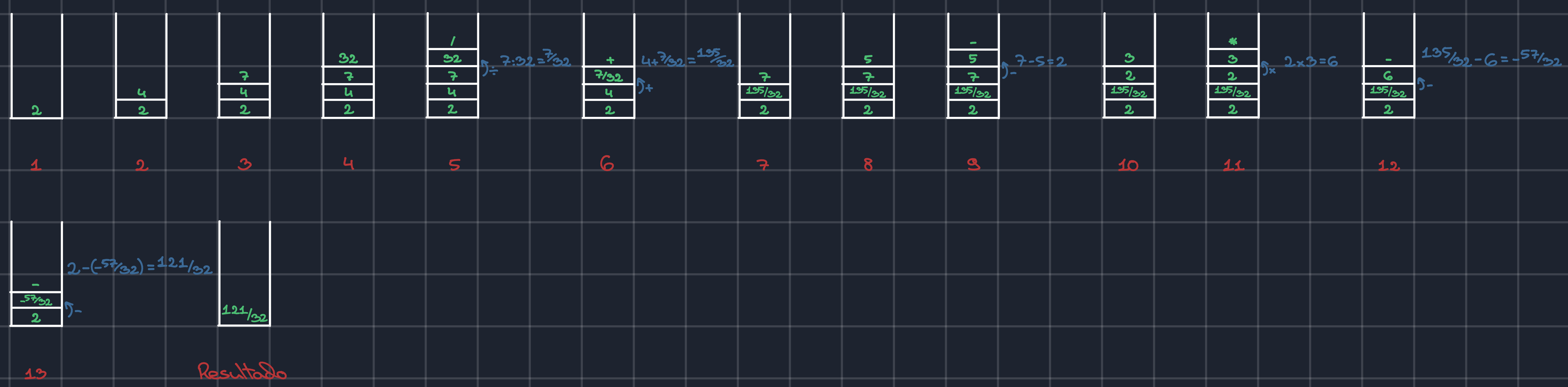
2. a) 23 56 - 3 * 1 3 4 / + /

1 2 3 4 5 6 7 8 9 10 11



b) 2 4 7 32 / + 7 5 - 3 * - -

1 2 3 4 5 6 7 8 9 10 11 12 13



c) 2 6 + 4 7 * 5 - / 6 9 / * 4 9 * 5 3 / + -

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

3. a) 2 * 8 + (10 + 5 * (20 / 4)) / (3 + -8)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

'(' é o 5: token lido

? b) 2 * 8 + (10 + 5 * (20 / 4)) ...

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1 Token: 2
Stack: Empty
Output: 2

2 Token: *
Stack: *
Output: 2

3 Token: 8
Stack: *
Output: 2 8 *

4 Token: +
Stack: +
Output: 2 8 *

5 Token: (
Stack: + (
Output: 2 8 *

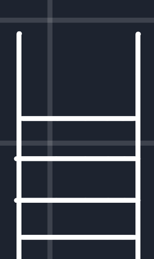
6 Token: 10
Stack: + (
Output: 2 8 * 10

7 Token: +
Stack: + (
Output: 2 8 * 10

8 Token: 5
Stack: + (
Output: 2 8 * 10 5

9 Token: *
Stack: * +
Output: 2 8 * 10 5 +

10 Token: (
Stack: * + (
Output: 2 8 * 10 5 +



11 Token: 20
Stack: * + (
Output: 2 8 * 10 5 + 20

12 Token: /
Stack: * + (
Output: 2 8 * 10 5 + 20 /

13 Token: 4
Stack: * + (
Output: 2 8 * 10 5 + 20 / 4

14 Token:)
Stack: * +
Output: 2 8 * 10 5 + 20 / 4 *

Desempilhar até encontrar '('

15 Token:)
Stack: +
Output: 2 8 * 10 5 + 20 / 4 * +

→ c) ... / (3 + -8)

16 17 18 19 20 21

16 Token: /
Stack:
Output:

17 Token: (
Stack:
Output:

18 Token: 3
Stack:
Output:

19 Token: +
Stack:
Output:

20 Token: -8
Stack:
Output:

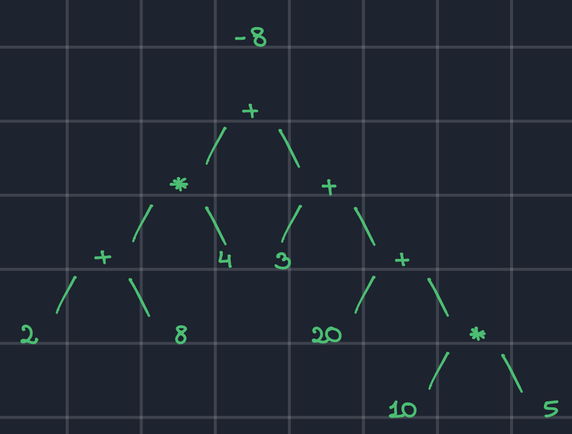
21 Token:)
Stack:
Output:

?

0

→ d) 2 8 * 10 5 + 20 \ 4 * 3 + -8 + ??

→ e)



??

4. 1 - Push
2 - Pop
3 - Top
4 - Empty

1. **Push (Inserir):**
   - A inserção de um elemento no topo da pilha geralmente tem complexidade de tempo constante O(1). Isso ocorre porque estamos simplesmente adicionando um elemento ao topo da estrutura de dados, e não precisamos percorrer elementos existentes.
2. **Pop (Remover):**
   - A remoção de um elemento do topo da pilha também tem complexidade de tempo constante O(1). Assim como no push, estamos apenas removendo o elemento do topo da pilha, e não precisamos percorrer outros elementos.
3. **Top/Peek (Acessar o topo):**
   - A operação de acesso ao elemento no topo da pilha sem removê-lo também tem complexidade de tempo constante O(1). Isso ocorre porque podemos acessar diretamente o elemento no topo da pilha sem precisar percorrer a estrutura.
4. **Empty (Verificar se está vazia):**
   - Verificar se a pilha está vazia também é uma operação de tempo constante O(1). Isso é possível porque normalmente mantemos um contador de elementos na pilha ou um marcador especial que indica se a pilha está vazia ou não.

Portanto, as operações comuns em uma pilha têm complexidade de tempo constante O(1). Isso é verdadeiro em situações normais em que não há necessidade de redimensionamento dinâmico da estrutura de dados subjacente.

5. a)
   b)
   c)