

UNIVERSIDADE DE BRASÍLIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA: ESTRUTURA DE DADOS

RELATÓRIO DO TRABALHO 1

ALUNO: JOÃO VÍTOR ARANTES CABRAL – 17/0126048

Avaliação de Expressões Aritméticas (forma posfixa) e Calculadora

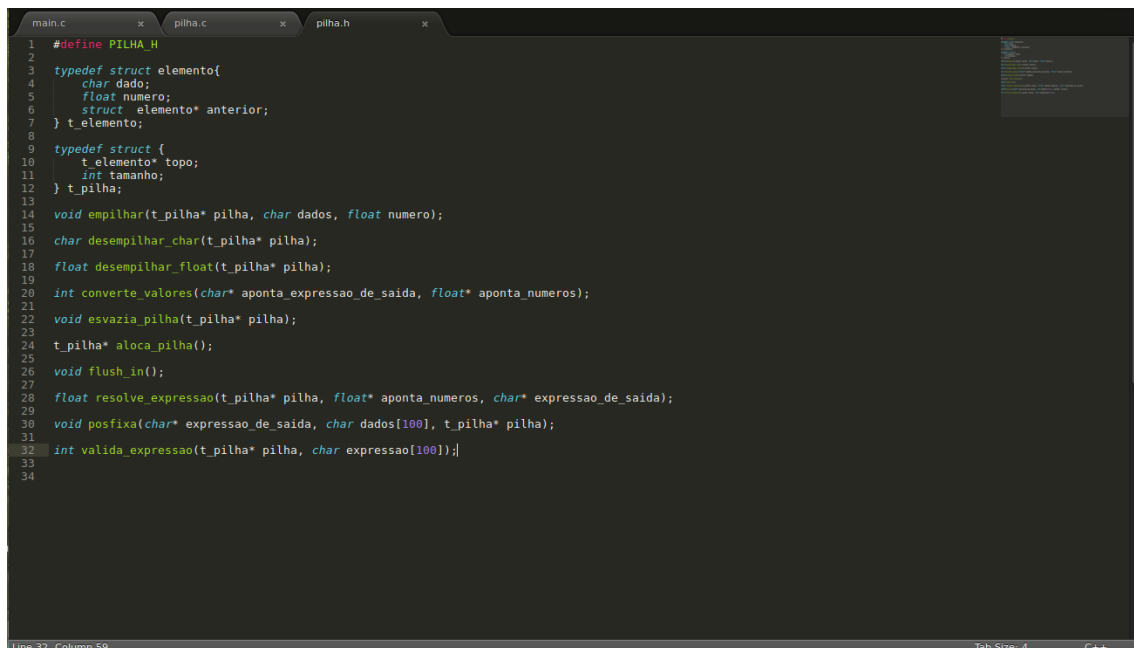
1.Introdução

A Notação Polonesa Reversa é um método de representação de expressões matemáticas que visa facilitar a resolução dessas através do uso da computação.

Nessa notação, os operadores são colocados na frente dos operandos de modo que a expressão matemática seja percorrida da esquerda para a direita até que um operador seja encontrado; após isso o caminho de volta é percorrido até que sejam encontrados dois números com os quais a operação será realizada.

2.Biblioteca e Estruturas

Para organizar os algoritmos, foi criada uma biblioteca ("pilha.h") com 2 tipos de estruturas (elemento e t_pilha) e 10 assinaturas de funções.



```
1 #define PILHA_H
2
3 typedef struct elemento{
4     char dado;
5     float numero;
6     struct elemento* anterior;
7 } t_elemento;
8
9 typedef struct {
10     t_elemento* topo;
11     int tamanho;
12 } t_pilha;
13
14 void empilhar(t_pilha* pilha, char dados, float numero);
15
16 char desempilhar_char(t_pilha* pilha);
17
18 float desempilhar_float(t_pilha* pilha);
19
20 int converte_valores(char* aponta expressao_de_saida, float* aponta_numeros);
21
22 void esvazia_pilha(t_pilha* pilha);
23
24 t_pilha* aloca_pilha();
25
26 void flush_in();
27
28 float resolve_expressao(t_pilha* pilha, float* aponta_numeros, char* expressao_de_saida);
29
30 void posfixa(char* expressao_de_saida, char dados[100], t_pilha* pilha);
31
32 int valida_expressao(t_pilha* pilha, char expressao[100]);
33
34
```

A pilha foi estruturada conforme uma lista através de alocação dinâmica de memória, sendo seus elos formados por estruturas do tipo `t_elemento`, que armazenam dois tipos de dado: `char` e `float`; sendo o primeiro utilizado nos dois algoritmos e o segundo apenas no segundo.

O tipo `t_pilha` possui apenas um numero para indicar seu tamanho e um ponteiro do tipo `t_elemento` que aponta para o topo da pilha.

3.Funções básicas

As funções que compõe a base dos 2 algoritmos são: `empilhar`, `desempilhar_float`, `desempilhar_char`, `converte_valores`, `esvazia_pilha` e `aloca_pilha`.

Abaixo, uma breve descrição de cada função:

`aloca_pilha` - Função que aloca memória para um ponteiro do tipo `t_pilha`, inicia suas variáveis e o retorna.

```

23 t_pilha* aloca_pilha() {
24     t_pilha* pilha = (t_pilha*) malloc(sizeof(t_pilha));
25     pilha->tamanho = 0;
26     pilha->topo = NULL;
27     return pilha;
28 }

```

empilhar - Aloca memória para um ponteiro do tipo `t_elemento` e guarda em suas variáveis os valores fornecidos pelo usuário; depois atualiza o valor do tamanho da pilha e o elemento que está no topo, sendo o topo antigo o elemento anterior ao novo elemento alocado.

```

14 void empilhar(t_pilha* pilha, char dados, float numero) {
15     t_elemento* elemento = (t_elemento*) malloc(sizeof(t_elemento));
16     elemento->dado = dados;
17     elemento->numero = numero;
18     elemento->anterior = pilha->topo;
19     pilha->topo = elemento;
20     pilha->tamanho++;
21 }

```

desempilhar_char - Recebe um ponteiro do tipo `t_pilha`, caso a pilha não esteja vazia, armazena o valor do tipo `char` do elemento que está no topo em uma variável, libera o espaço ocupado por este elemento, atualiza o tamanho da pilha e estabelece o elemento anterior como o novo topo. Após tudo isso, retorna o caractere armazenado. Caso a pilha esteja vazia, libera o espaço armazenado para a pilha.

```

30 char desempilhar_char(t_pilha* pilha) {
31     char inutil, dado;
32     if (pilha->tamanho != 0) {
33         dado = pilha->topo->dado;
34         t_elemento* auxiliar = pilha->topo->anterior;
35         free(pilha->topo);
36         pilha->topo = auxiliar;
37         pilha->tamanho--;
38     } else {
39         printf("Ocorreu um underflow na pilha (ou esta sendo esvaziada). Pressione enter para continuar.\n");
40         flush_in();
41         scanf("%c", &inutil);
42         flush_in();
43         free(pilha);
44         return 'w';
45     }
46     return dado;
47 }

```

desempilhar_float - Funcionalidade similar a `desempilhar_char`, a única diferença é que é retornado um valor do tipo `float` armazenado no topo da pilha.

```

49 float desempilhar_float(t_pilha* pilha) {
50     char inutil;
51     float dado;
52     if (pilha->tamanho != 0) {
53         dado = pilha->topo->numero;
54         t_elemento* auxiliar = pilha->topo->anterior;
55         free(pilha->topo);
56         pilha->topo = auxiliar;
57         pilha->tamanho--;
58     } else {
59         printf("Ocorreu um underflow na pilha (ou esta sendo esvaziada). Pressione enter para continuar.\n");
60         flush_in();
61         scanf("%c", &inutil);
62         flush_in();
63         free(pilha);
64         return 0.0;
65     }
66     return dado;
67 }

```

esvazia_pilha - Funcionalidade que recebe um ponteiro do tipo `t_pilha` e libera todo o espaço de memória alocado para a pilha através de chamadas consecutivas da função `desempilha_char`.

```

69 void esvazia_pilha(t_pilha* pilha) {
70     char esvazia = 'z';
71     while (esvazia != 'w') {
72         esvazia = desempilhar_char(pilha);
73     }
74 }

```

converte_valores - A função mais complexa dentre as básicas, recebe um ponteiro para um vetor de caracteres e um ponteiro para um vetor de números de ponto flutuante. O primeiro passo realizado é retirar todos os espaços entre os caracteres do vetor recebido. Após isso, a expressão é percorrida até que um caractere que representa um número seja encontrado; os algarismos são lidos e multiplicados com o seu índice (unidade, dezena, milhar), se uma vírgula for encontrada, o índice se torna uma potência de base 10 com expoente negativo incrementado a cada leitura. O número gerado é então armazenado em outra variável cujo valor será armazenado no vetor de números recebido pela função.

Os últimos 2 passos são repetidos até que toda a expressão seja percorrida. A função retorna um valor do tipo inteiro correspondente ao número de valores armazenados no vetor.

```

76 int converte_valores(char* aponta_expressao_de_saida, float* aponta_numeros) {
77     int quantidade = 0, indice = 0, i = 0, j = 0;
78     float soma = 0.0, unidade = 1.0;
79     char expressao[100];
80     while (*aponta_expressao_de_saida != '\0') {
81         if (*aponta_expressao_de_saida == ' ') {
82             expressao[i] = '_';
83         } else {
84             expressao[i] = *aponta_expressao_de_saida;
85         }
86         aponta_expressao_de_saida++;
87         i++;
88     }
89     expressao[i] = '\0';
90     while (expressao[j] != '\0') {
91         indice = 0;
92         unidade = 1.0;
93         soma = 0.0;
94         if (expressao[j] >= '0' && expressao[j] <= '9') {
95             while ((expressao[j] >= '0' && expressao[j] <= '9') || expressao[j] == ',') {
96                 if (expressao[j] == ',') {
97                     indice = 1;
98                 }
99                 if (indice == 0 && soma == 0.0 && expressao[j] != ',') {
100                     soma+= expressao[j] - '0';
101                 } else if (indice == 0 && expressao[j] != ',') {
102                     soma*= 10.0;
103                     soma+= expressao[j] - '0';
104                 } else if (indice == 1 && expressao[j] != ',') {
105                     soma+= (expressao[j] - '0') * (unidade * 0.1);
106                     unidade*= 0.1;
107                 }
108                 j++;
109             }
110             *aponta_numeros = soma;
111             aponta_numeros++;
112             quantidade++;
113         }
114         j++;
115     }
116     return quantidade;
117 }

```

4.Algoritmo 1

Boa parte do algoritmo 1 está localizado na função main, primeiro é solicitado que o usuário forneça ao programa uma expressão matemática composta de até 100 caracteres. Após isso, é verificado se a expressão é válida através da função valida_expressao:

```

179 int main(int argc, const char* argv[]) {
180     t_pilha* pilha;
181     int opcao = 1, sucesso = 0, quantidade_numeros = 0, i = 0, j = 0;
182     char dados[100], inutil, expressao_de_saida[100];
183     char* aponta_expressao_de_saida = &expressao_de_saida[0];
184     char* aponta_dados = &dados[0];
185     float numeros[100], resultado = 0.0;
186     float* aponta_numeros = &numeros[0];
187     while(opcao != 0) {
188         while (aponta_expressao_de_saida != &expressao_de_saida[0]) {
189             *aponta_expressao_de_saida = '\0';
190             aponta_expressao_de_saida--;
191         }
192         for (i = 0; i < 100; i++) {
193             dados[i] = '\0';
194         }
195         aponta_numeros = &numeros[0];
196         aponta_expressao_de_saida = &expressao_de_saida[0];
197         printf("1 - Resolucao de expressoes matematicas\n");
198         printf("2 - Calculadora\n\n");
199         printf("0 - Sair\n");
200         scanf("%d", &opcao);
201         if (opcao == 1){
202             pilha = aloca_pilha();
203             printf("Digite uma expressao matematica de ate 100 caracteres:\n");
204             flush in();
205             scanf("%100s", dados);
206             sucesso = valida_expressao(pilha, dados);

```

valida_expresao – Essa função recebe a expressão fornecida pelo usuário, assim como a pilha alocada. Para validar a expressão, são comparados os fechamentos de parênteses, colchetes e chaves com suas aberturas correspondentes; caso não haja estes elementos na pilha e/ou não exista correspondência entre a abertura e o fechamento destes, a função retorna 0 para indicar que a expressão fornecida é inválida, caso contrário, retorna 1. A função também faz uso de um contador para verificar quantos escopos foram abertos e se todos foram fechados.

```

main.c x pilha.c x pilha.h x
133 int valida_expressao(t_pilha* pilha, char* expressao){
134     t_elemento* auxiliar = pilha->topo;
135     char comparador;
136     int i = 0, contador = 0;
137     while (expressao[i] != '\0'){
138         if (expressao[i] == '{' || expressao[i] == '[' || expressao[i] == '(') {
139             empilhar(pilha, expressao[i], 0.0);
140             contador++;
141         } else if (expressao[i] == '}' || ']' || ')') {
142             if (pilha->tamanho == 0) {
143                 return 0;
144             }
145             comparador = desempilhar_char(pilha);
146             if (comparador != '{' || '[' || '(') {
147                 esvazia_pilha(pilha);
148                 return 0;
149             }
150             contador--;
151         } else if (expressao[i] == '|') {
152             if (pilha->tamanho == 0) {
153                 return 0;
154             }
155             comparador = desempilhar_char(pilha);
156             if (comparador != '|' || '[' || '(') {
157                 esvazia_pilha(pilha);
158                 return 0;
159             }
160             contador--;
161         } else if (expressao[i] == '|') {
162             if (pilha->tamanho == 0) {
163                 return 0;
164             }
165             comparador = desempilhar_char(pilha);
166             if (comparador != '|' || '[' || '(') {
167                 esvazia_pilha(pilha);
168                 return 0;
169             }
170             contador--;
171         }
172         i++;
173     }
174     if (contador != 0) {
175         return 0;
176     }
177     return 1;

```

Após verificar a validade da expressão, esta é convertida da forma infixa para a posfixa (Notação Polonesa Reversa) através da chamada da função posfixa:

```
201         if (opcao == 1){
202             pilha = aloca_pilha();
203             printf("Digite uma expressao matematica de ate 100 caracteres:\n");
204             flush_in();
205             scanf("%100s", dados);
206             sucesso = valida_expressao(pilha, dados);
207             if (sucesso == 1) {
208                 printf("A expressao fornecida e valida. Pressione enter para continuar.\n");
209                 flush_in();
210                 scanf("%c", &inutil);
211                 esvazia_pilha(pilha);
212                 pilha = aloca_pilha();
213                 posfixa(aponta_expressao_de_saida, dados, pilha);
214                 printf("A expressao na forma posfixa e:\n");
215                 printf("%s\n", expressao_de_saida);
216                 printf("Pressione enter para continuar\n");
217                 flush_in();
218                 scanf("%c", &inutil);
219                 aponta_expressao_de_saida = &expressao_de_saida[0];
220                 quantidade_numeros = converte_valores(aponta_expressao_de_saida, aponta_numeros);
221                 aponta_numeros = &numeros[0];
222                 aponta_expressao_de_saida = &expressao_de_saida[0];

```

posfixa – Provavelmente a função mais complexa de todo o programa, recebe um ponteiro para a expressão de saída, a expressão fornecida pelo usuário e a pilha. A string dados é totalmente percorrida seguindo os seguintes parâmetros:

- Se um número é encontrado, ele é copiado para a expressão de saída;
- Se um '+' ou um '-' é encontrado, a pilha é esvaziada totalmente (a não ser que um parêntese de abertura seja encontrado, nesse caso ele é empilhado de volta) copiando todos os operadores encontrados para a expressão de saída, após isso o operador é empilhado;
- Se um '*' ou um '/' é encontrado, todos os '/' e '*' da pilha são desempilhados e copiados para a expressão de saída, após isso o operador é empilhado;
- Se uma abertura de escopo é encontrada, ela é empilhada;
- Se um fechamento de escopo é encontrado, a pilha é esvaziada até que a abertura do escopo correspondente seja encontrada, sendo todos os operadores encontrados copiados para a expressão de saída.

Após isso a pilha é esvaziada e os operadores restantes são copiados para a expressão de saída.

```

56 void posfixa(char* expressao_de_saida, char dados[100], t_pilha* pilha) {
57     int i = 0, j = 0;
58     char caractere = 'w';
59     while (dados[i] != '\0') {
60         if ((dados[i] >= '0' && dados[i] <= '9') || dados[i] == ',') {
61             while ((dados[i] >= '0' && dados[i] <= '9') || dados[i] == ',') {
62                 *expressao_de_saida = dados[i];
63                 expressao_de_saida++;
64                 i++;
65             }
66             *expressao_de_saida = ' ';
67             expressao_de_saida++;
68         }
69         if (dados[i] == '+' || dados[i] == '-') {
70             if (pilha->tamanho == 0) {
71                 empilhar(pilha, dados[i], 0.0);
72             } else {
73                 while ((caractere != '{' && caractere != '[' && caractere != '(' && pilha->tamanho != 0) {
74                     caractere = desempilhar_char(pilha);
75                     if (caractere != '{' && caractere != '[' && caractere != '(') {
76                         *expressao_de_saida = caractere;
77                         expressao_de_saida++;
78                         *expressao_de_saida = ' ';
79                         expressao_de_saida++;
80                     } else {
81                         empilhar(pilha, caractere, 0.0);
82                     }
83                 }
84                 empilhar(pilha, dados[i], 0.0);
85             }
86         }
87         } else if (dados[i] == '*' || dados[i] == '/') {
88             caractere = '/';
89             if (pilha->tamanho != 0) {
90                 while ((caractere == '/' || caractere == '*') && pilha->tamanho != 0) {
91                     caractere = desempilhar_char(pilha);
92                     if (caractere == '/' || caractere == '*') {
93                         *expressao_de_saida = caractere;
94                         expressao_de_saida++;
95                         *expressao_de_saida = ' ';
96                         expressao_de_saida++;
97                     } else {
98                         empilhar(pilha, caractere, 0.0);
99                     }
100                 }
101             }
102             empilhar(pilha, dados[i], 0.0);
103         } else if (dados[i] == '(' || dados[i] == '[' || dados[i] == '{') {
104             empilhar(pilha, dados[i], 0.0);
105         } else if (dados[i] == ')' || dados[i] == ']' || dados[i] == '}') {
106             caractere = 'w';
107             while ((caractere != '{' && caractere != '[' && caractere != '(' && pilha->tamanho != 0) {
108                 caractere = desempilhar_char(pilha);
109                 if (caractere != '{' && caractere != '[' && caractere != '(') {
110                     *expressao_de_saida = caractere;
111                     expressao_de_saida++;
112                     *expressao_de_saida = ' ';
113                     expressao_de_saida++;
114                 }
115             }
116             i++;
117         }
118     }
119     while (pilha->tamanho != 0) {
120         caractere = desempilhar_char(pilha);
121         if (caractere != '{' && caractere != '[' && caractere != '(') {
122             *expressao_de_saida = caractere;
123             expressao_de_saida++;
124             *expressao_de_saida = ' ';
125             expressao_de_saida++;
126         }
127     }
128     free(pilha);
129     *expressao_de_saida = '\0';
130 }

```

Depois que a expressão se encontra na Notação Polonesa Reversa, todos os números encontrados na expressão de saída resultante são convertidos para variáveis do tipo float na função `converte_valores`.

Finalmente, a expressão fornecida é resolvida pela função `resolve_expressao` e o resultado é mostrado ao usuário:


```

220     quantidade_numeros = converte_valores(aponta_expressao_de_saida, aponta_numeros);
221     aponta_numeros = &numeros[0];
222     aponta_expressao_de_saida = &expressao_de_saida[0];
223     pilha = aloca_pilha();
224     resultado = resolve_expressao(pilha, aponta_numeros, aponta_expressao_de_saida);
225     printf("O resultado da expressao e:\n");
226     printf("%f\n", resultado);
227     printf("Pressione enter para continuar\n");
228     flush_in();
229     scanf("%c", &inutil);
230 } else {
231     printf("A expressao fornecida nao e valida. Pressione enter para continuar.\n");
232     flush_in();
233     scanf("%c", &inutil);
234 }

```

resolve_expressao – Essa função recebe um ponteiro para o vetor de float números, assim como um ponteiro para a expressão de saída e a pilha. A expressão de saída é armazenada em outra string, sendo que seus espaços são substituídos por ‘_’; após isso a nova expressão é percorrida seguindo os seguintes parâmetros:

- Se um número é encontrado, é empilhado o valor float armazenado no endereço do ponteiro de números. Após isso o ponteiro é incrementado e o resto do número é percorrido na expressão;
- Se um operador é encontrado e o tamanho da pilha é maior ou igual a dois, a pilha é desempilhada duas vezes e a operação é realizada, sendo o valor anterior ao topo a base para a operação (por exemplo, se o valor no topo da pilha é 2 e o valor abaixo é 3, a operação realizada é $3 / 2$);
- Após ter percorrido toda a expressão, o único valor restante na pilha é o resultado daquela, sendo este desempilhado e retornado para a função main, a pilha é liberada.

```

10 char expressao[100];
11 while (*expressao_de_saida != '\0') {
12     if (*expressao_de_saida == ' ') {
13         expressao[i] = ' ';
14     } else {
15         expressao[i] = *expressao_de_saida;
16     }
17     expressao_de_saida++;
18     i++;
19 }
20 expressao[i] = '\0';
21 while (expressao[j] != '\0') {
22     if (expressao[j] >= '0' && expressao[j] <= '9') {
23         empilhar(pilha, 'w', *aponta_numeros);
24         aponta_numeros++;
25         while ((expressao[j] >= '0' && expressao[j] <= '9') || expressao[j] == ',') {
26             j++;
27         }
28     } else if (expressao[j] == '+' && pilha->tamanho >= 2) {
29         operando_1 = desempilhar_float(pilha);
30         operando_2 = desempilhar_float(pilha);
31         resultado = operando_2 + operando_1;
32         empilhar(pilha, 'w', resultado);
33     } else if (expressao[j] == '-' && pilha->tamanho >= 2) {
34         operando_1 = desempilhar_float(pilha);
35         operando_2 = desempilhar_float(pilha);
36         resultado = operando_2 - operando_1;
37         empilhar(pilha, 'w', resultado);
38     } else if (expressao[j] == '/' && pilha->tamanho >= 2) {
39         operando_1 = desempilhar_float(pilha);
40         operando_2 = desempilhar_float(pilha);
41         resultado = operando_2 / operando_1;
42         empilhar(pilha, 'w', resultado);
43     } else if (expressao[j] == '*' && pilha->tamanho >= 2) {
44         operando_1 = desempilhar_float(pilha);
45         operando_2 = desempilhar_float(pilha);
46         resultado = operando_2 * operando_1;
47         empilhar(pilha, '*', resultado);
48     }
49     j++;
50 }
51 resultado = desempilhar_float(pilha);
52 free(pilha);
53 return resultado;
54 }

```

5.Algoritmo 2

O algoritmo 2 é praticamente todo implementado na função main, sendo utilizadas algumas funções citadas anteriormente.

A calculadora se inicia com uma pilha nova e com uma interface que é atualizada a cada operação solicitada pelo usuário. A string fornecida por este também é atualizada de forma similar.

A variável quantidade_numeros permite ao programa saber quantos números serão impressos na interface e em qual posição do vetor números o ponteiro aponta números está. Essa variável é atualizada a cada número fornecido pelo usuário. Como a função converte_valores retorna a quantidade de números convertidos de string para float, é relativamente fácil incrementar a

quantidade de números presentes na calculadora (os números convertidos são empilhados):

```
235     } else if (opcao == 2) {
236         float operando_1 = 0.0;
237         float operando_2 = 0.0;
238         pilha = aloca_pilha();
239         quantidade_numeros = 0;
240         while (dados[0] != 's') {
241             system("clear");
242             for (i = 0; i < 100; i++) {
243                 dados[i] = '\0';
244             }
245             i = 0;
246             aponta_dados = &dados[0];
247             printf("Calculadora. Digite s para sair\n");
248             while (i < quantidade_numeros) {
249                 printf("%d. %f\n", i, numeros[i]);
250                 i++;
251             }
252             printf("%d.", i);
253             flush_in();
254             aponta_numeros = &numeros[i];
255             scanf("%100s", dados);
256             if (dados[0] == 's' || dados[0] == 'S') {
257                 break;
258             } else if (dados[0] >= '0' && dados[0] <= '9') {
259                 quantidade_numeros += converte_valores(aponta_dados, aponta_numeros);
260                 empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
261             } else if ((dados[0] == '-' && (dados[1] >= '0' && dados[1] <= '9')) {
262                 aponta_dados = &dados[1];
263                 quantidade_numeros += converte_valores(aponta_dados, aponta_numeros);
264                 numeros[quantidade_numeros - 1] *= -1.0;
```

Ao solicitar uma operação, o algoritmo confere se esta é válida e se a pilha possui 2 ou mais operandos e realiza a operação tendo como base o número abaixo do topo da pilha (assim como no algoritmo 1). Além disso, há o operador especial “!”, que ao ler um operador, realiza a operação correspondente com todos os números armazenados na pilha:

```
264         numeros[quantidade_numeros - 1] *= -1.0;
265         empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
266     } else if (dados[0] == '!') {
267         if (dados[0] == '!') {
268             if (pilha->tamanho >= 2) {
269                 while (pilha->tamanho >= 2) {
270                     operando_1 = desempilhar_float(pilha);
271                     operando_2 = desempilhar_float(pilha);
272                     quantidade_numeros--;
273                     numeros[quantidade_numeros - 1] = operando_2 + operando_1;
274                     empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
275                 }
276             } else {
277                 printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para con");
278                 flush_in();
279                 scanf("%c", &inutil);
280             }
281         } else if (dados[0] == '-') {
282             if (pilha->tamanho >= 2) {
283                 while (pilha->tamanho >= 2) {
284                     operando_1 = desempilhar_float(pilha);
285                     operando_2 = desempilhar_float(pilha);
286                     quantidade_numeros--;
287                     numeros[quantidade_numeros - 1] = operando_2 - operando_1;
288                     empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
289                 }
290             } else {
291                 printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para con");
292                 flush_in();
293                 scanf("%c", &inutil);
294             }
295         } else if (dados[0] == '*') {
296             if (pilha->tamanho >= 2) {
297                 while (pilha->tamanho >= 2) {
298                     operando_1 = desempilhar_float(pilha);
299                     operando_2 = desempilhar_float(pilha);
300                     quantidade_numeros--;
301                     numeros[quantidade_numeros - 1] = operando_2 * operando_1;
302                     empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
303                 }
304             } else {
305                 printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para con");
306                 flush_in();
307             }
308         }
```

```

308         scanf("%c", &inutil);
309     }
310     } else if (dados[0] == '/') {
311         if (pilha->tamanho >= 2) {
312             while (pilha->tamanho >= 2) {
313                 operando_1 = desempilhar_float(pilha);
314                 operando_2 = desempilhar_float(pilha);
315                 quantidade_numeros--;
316                 numeros[quantidade_numeros - 1] = operando_2 / operando_1;
317                 empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
318             }
319         } else {
320             printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para con");
321             flush_in();
322             scanf("%c", &inutil);
323         }
324     }
325     } else if (dados[0] == '+') {
326         if (pilha->tamanho >= 2) {
327             operando_1 = desempilhar_float(pilha);
328             operando_2 = desempilhar_float(pilha);
329             quantidade_numeros--;
330             numeros[quantidade_numeros - 1] = operando_2 + operando_1;
331             empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
332         } else {
333             printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para continu");
334             flush_in();
335             scanf("%c", &inutil);
336         }
337     } else if (dados[0] == '-') {
338         if (pilha->tamanho >= 2) {
339             operando_1 = desempilhar_float(pilha);
340             operando_2 = desempilhar_float(pilha);
341             quantidade_numeros--;
342             numeros[quantidade_numeros - 1] = operando_2 - operando_1;
343             empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
344         } else {
345             printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para continu");
346             flush_in();
347             scanf("%c", &inutil);
348         }
349     } else if (dados[0] == '*') {
350         if (pilha->tamanho >= 2) {
351             operando_1 = desempilhar_float(pilha);

```

```

351             operando_1 = desempilhar_float(pilha);
352             operando_2 = desempilhar_float(pilha);
353             quantidade_numeros--;
354             numeros[quantidade_numeros - 1] = operando_2 * operando_1;
355             empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
356         } else {
357             printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para continu");
358             flush_in();
359             scanf("%c", &inutil);
360         }
361     } else if (dados[0] == '/') {
362         if (pilha->tamanho >= 2) {
363             operando_1 = desempilhar_float(pilha);
364             operando_2 = desempilhar_float(pilha);
365             quantidade_numeros--;
366             numeros[quantidade_numeros - 1] = operando_2 / operando_1;
367             empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
368         } else {
369             printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para continu");
370             flush_in();
371             scanf("%c", &inutil);
372         }

```

O operador de cópia “c” lê a quantidade de vezes que se deve copiar o número base no topo da pilha, e obtém o número a ser copiado no valor armazenado abaixo daquele:

```

373     } else if (dados[0] == 'c') {
374         if (pilha->tamanho >= 2) {
375             operando_1 = desempilhar_float(pilha);
376             operando_2 = desempilhar_float(pilha);
377             if (operando_1 > 0) {
378                 quantidade_numeros = 2;
379                 for (j = 0; j < operando_1; j++) {
380                     quantidade_numeros++;
381                     numeros[quantidade_numeros - 1] = operando_2;
382                     empilhar(pilha, 'w', numeros[quantidade_numeros - 1]);
383                 }
384             } else {
385                 empilhar(pilha, 'w', operando_2);
386                 empilhar(pilha, 'w', operando_1);
387                 printf("Voce digitou um valor negativo ou nulo para o numero de copias. Pressione enter duas vezes para con");
388                 flush_in();
389                 scanf("%c", &inutil);
390             }

```

```

390         }
391     } else {
392         printf("Quantidade de numeros insuficiente pra realizar essa operacao. Pressione enter duas vezes para continu");
393         flush_in();
394         scanf("%c", &inutil);
395     }
396 }
397
398 }
399
400 }
401
402 }
403
404 }
405
406 }
407
408 }
409
410 }

```

