UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Disciplina: Estrutura de Dados
Semestre: 2020.2
Professor: Leandro Carlos de Souza
Nome: Francisco Siqueira Carneiro da Cunha Neto
Matrícula: 20190029015

Exercícios de Fixação e Aprendizagem II

Questão 1.

Sobre Pilhas:

(a) Explique o funcionamento do TAD Pilha.

A pilha é uma estrutura de dados que armazena uma coleção de dados. O que a diferencia é a forma como esses dados são acessados: A pilha segue um modelo "last in first out", o que significa que o último dado a ser inserido é o primeiro a ser acessado e removido. Sempre que inserimos um dado na pilha, este é considerado como seu "topo", e só podemos interagir (isto é, acessar ou remover dados) com o topo da pilha. Quando removemos um dado (do topo) da pilha, o dado que foi inserido anteriormente a ele passa a ser o novo "topo". A pilha computacional funciona de maneira análoga a uma pilha física (como uma pilha de pratos).

(b) Crie um TAD de Pilha utilizando lista encadeada.

Código 1: linkedlist.h

```
1  #ifndef _LINKED_LIST_H_
2  #define _LINKED_LIST_H_
3
4  typedef struct linkedlist LinkedList;
5  typedef struct listnode ListNode;
6
7  LinkedList *ll_create_list();
8  ListNode *ll_create_node(float data);
9  int ll_is_empty(LinkedList *list);
10 int ll_size(LinkedList *list);
11 int ll_insert_first(LinkedList *list, float data);
12 int ll_insert_last(LinkedList *list, float data);
13 int ll_exists(LinkedList *list, float data);
14 void ll_print(LinkedList *list, char *message);
```

```
15 void ll_clear(LinkedList *list);
16 void ll_free(LinkedList *list);
17 int ll_remove(LinkedList *list, float data);
18 int ll_insert_sorted(LinkedList *list, float data);
19 float ll_first(LinkedList *list);
20 int ll_remove_first(LinkedList *list);
21 float ll_get(LinkedList *list, int pos);
22
23 #endif
```

Código 2: linkedlist.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "../include/linkedlist.h"
5
6  #ifndef _LINKED_LIST_C_
7  #define _LINKED_LIST_C_
8
9  // Struct da lista duplamente encadeada.
10 // Armazena um ponteiro para o primeiro e ultimo no da lista.
11 struct linkedlist
12 {
13     ListNode *head;
14 };
15
16 // Struct de no da lista.
17 // Armazena o dado informado de tipo real, e um ponteiro para o proximo
       no da lista.
18 struct listnode
19 {
20     float data;
21     ListNode *next;
22 };
23
24 // Cria uma nova lista encadeada, retornando seu endereco de memoria.
25 LinkedList *ll_create_list()
26 {
27     // Alocar memoria necessaria para a lista.
28     LinkedList *list = (LinkedList *)malloc(sizeof(LinkedList));
29
30     // Se a memoria foi alocada sem erros, definir seu primeiro no como
           nulo.
31     if (list)
32         list->head = NULL;
33
34     return list;
35 }
36
37 // Cria um no da lista, retornando seu endereco de memoria.
```

```c
ListNode *ll_create_node(float data)
{
    // Alocar memoria necessaria para um no.
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));

    // Se a memoria foi alocada sem erros, definir o dado armazenado, e o
        proximo no como nulo.
    if (node)
    {
        node->data = data;
        node->next = NULL;
    }

    return node;
}

// Checa se a lista esta vazia.
int ll_is_empty(LinkedList *list)
{
    // Quando ela esta vazia, seu primeiro elemento e nulo.
    return list->head == NULL;
}

// Retorna o tamanho da lista.
int ll_size(LinkedList *list)
{
    int counter = 0;
    // Percorre a lista inteira.
    for (ListNode *node = list->head; node != NULL; node = node->next)
    {
        // Incrementa o contador para cada no da lista.
        counter++;
    }
    return counter;
}

// Insere um elemento no comeco da lista.
int ll_insert_first(LinkedList *list, float data)
{
    ListNode *newnode = ll_create_node(data);
    // Se a memoria nao foi alocada, sai da funcao retornando erro.
    if (newnode == NULL)
        return 0;

    // Define o atual primeiro no como proximo do novo.
    newnode->next = list->head;
    // Define o primeiro no como o novo.
    list->head = newnode;
    return 1;
}

```

```c
// Insere um elemento no fim da lista.
int ll_insert_last(LinkedList *list, float data)
{
    // Percorre a lista inteira.
    ListNode *node = list->head;
    while (node->next != NULL)
    {
        node = node->next;
    }

    // Cria um novo no e o define proximo no do ultimo.
    node->next = ll_create_node(data);

    // Retorna erro se no nao pode ser criado.
    return node->next != NULL;
}

// Checa se existe um elemento na lista com esse dado.
int ll_exists(LinkedList *list, float data)
{
    // Percorre a lista inteira.
    for (ListNode *node = list->head; node != NULL; node = node->next)
    {
        // Se algum elemento tiver aquele dado, retorna verdadeiro.
        if (node->data == data)
            return 1;
    }

    // Se percorrer a lista inteira e nao tiver retornado nada, e porque
       o elemento nao existe.
    return 0;
}

// Imprime a lista no terminal.
void ll_print(LinkedList *list, char *message)
{
    printf("%s", message);
    // Imprime cada elemento.
    for (ListNode *node = list->head; node != NULL; node = node->next)
    {
        printf("%f ", node->data);
    }
    printf("\n");
}

// Libera a memoria alocada para todos os elementos da lista, tornando-a
   vazia.
void ll_clear(LinkedList *list)
{
    // Comecando do primeiro elemento da lista.
    ListNode *next = list->head;
```

```
137
138     while (next != NULL)
139     {
140         // Endereco do no que sera liberado nessa iteracao.
141         ListNode *current = next;
142         // Endereco do no que sera liberado na proxima iteracao e
                guardado em outra variavel para que nao se perca quando o
                atual for liberado.
143         next = current->next;
144
145         // Libera no atual.
146         free(current);
147     }
148
149     // Volta a lista para o seu estado inicial.
150     list->head = NULL;
151 }
152
153 // Libera a memoria alocada para a lista.
154 void ll_free(LinkedList *list)
155 {
156     // Libera a memoria alocada para cada elemento.
157     ll_clear(list);
158     // Libera a memoria alocada para a estrutura da lista.
159     free(list);
160 }
161
162 // Remove o no com o dado informado da lista.
163 int ll_remove(LinkedList *list, float data)
164 {
165     // Se o elemento a ser removido e o primeiro da lista
166     if (data == list->head->data)
167     {
168         ListNode *removal = list->head;
169         // Define o segundo no da lista como novo primeiro no.
170         list->head = removal->next;
171         // Libera a memoria alocada para o no removido.
172         free(removal);
173         return 1;
174     }
175
176     // Percorre a lista inteira.
177     for (ListNode *node = list->head; node != NULL; node = node->next)
178     {
179         ListNode *removal = node->next;
180
181         // Se nao chegou ao fim da lista, e o dado foi encontrado.
182         if (removal != NULL && removal->data == data)
183         {
184             // Conecta o no anterior ao removido com o proximo ao
                    removido.
```

```
185             node->next = removal->next;
186             // Libera a memoria alocada para o no removido.
187             free(removal);
188 
189             return 1;
190         }
191     }
192     return 0;
193 }
194 
195 // Insere um elemento na lista em ordem crescente.
196 int ll_insert_sorted(LinkedList *list, float data)
197 {
198     // Se a lista esta vazia, ou se o elemento for menor que o primeiro
        da lista, insere elemento no comeco.
199     if (list->head == NULL || list->head->data > data)
200     {
201         return ll_insert_first(list, data);
202     }
203 
204     // Cria o novo no.
205     ListNode *newnode = ll_create_node(data);
206     if (newnode == NULL)
207         return 0;
208 
209     ListNode *last = list->head;
210     ListNode *node = last->next;
211 
212     // Percorre a lista inteira.
213     while (node != NULL)
214     {
215         // Se o elemento atual e maior do que o novo
216         if (node->data > data)
217         {
218             // Insere o novo atras do atual
219             newnode->next = node;
220             last->next = newnode;
221 
222             // E retorna sucesso.
223             return 1;
224         }
225 
226         last = node;
227         node = node->next;
228     }
229 
230     // Se a lista inteira foi percorrida, insere o novo elemento no final
        da lista e retorna sucesso.
231     last->next = newnode;
232     return 1;
233 }
```

```
234
235 // Retorna o valor do primeiro elemento da lista.
236 float ll_first(LinkedList *list)
237 {
238     return list->head->data;
239 }
240
241 // Remove o primeiro no da lista.
242 int ll_remove_first(LinkedList *list)
243 {
244     ListNode *node = list->head;
245
246     // Define o primeiro no da lista como o atual segundo.
247     list->head = node->next;
248
249     // Libera a memoria alocada para o no removido.
250     free(node);
251     return 1;
252 }
253
254 // Retorna o valor armazenado no no de dada posicao.
255 float ll_get(LinkedList *list, int pos)
256 {
257     // Se a posicao nao existe, ou e a primeira, retorna o primeiro valor
        .
258     if (pos >= ll_size(list) || pos <= 0)
259         return list->head->data;
260
261     // Percorre a lista ate o no da posicao desejada.
262     ListNode *node = list->head;
263     for (int i = 0; i < pos; i++)
264     {
265         node = node->next;
266     }
267
268     // Retorna o conteudo daquele no.
269     return node->data;
270 }
271
272
273 #endif
```

Código 3: stack.h

```
1 #ifndef _STACK_H_
2 #define _STACK_H_
3
4 typedef struct stack Stack;
5
6 Stack* stack_create();
7 int stack_push(Stack* stack, float value);
```

```c
int stack_pop(Stack* stack, float *data);
int stack_peek(Stack* stack, float *data);
int stack_is_empty(Stack* stack);
void stack_free(Stack* stack);

#endif
```

Código 4: stack.c

```c
#include <stdlib.h>

#include "../include/linkedlist.h"
#include "../include/stack.h"

#ifndef _STACK_C_
#define _STACK_C_

// Struct da pilha
// Armazena a lista sobre a qual a pilha vai ser construida
struct stack
{
    LinkedList *list;
};

// Cria uma nova pilha, retornando seu endereco de memoria
// O(1)
Stack* stack_create()
{
    Stack *stack = (Stack*)malloc(sizeof(Stack));

    // Se a memoria foi alocada com sucesso
    if (stack)
    {
        // Cria a lista da pilha
        stack->list = ll_create_list();

        // Se a lista nao pode ser criado, retorna nulo
        if (!stack->list)
            return NULL;
    }

    return stack;
}

// Insere um elemento na pilha
// O(1)
int stack_push(Stack* stack, float data)
{
    // Insere um elemento ao inicio da lista da pilha
    return ll_insert_first(stack->list, data);
}
```

```c

// Remove um elemento da pilha, retornando seu valor
// O(1)
int stack_pop(Stack* stack, float *data)
{
    // Pega o elemento no topo da lista, retornando falso se isso falhar
    if (!stack_peek(stack, data))
        return 0;

    // Remove o elemento no topo da lista
    ll_remove_first(stack->list);
    return 1;
}

// Retorna o valor do elemento no topo da pilha
// O(1)
int stack_peek(Stack* stack, float *data)
{
    // Nao ha elementos para retornar caso a pilha esteja vazia
    if (stack_is_empty(stack))
        return 0;

    *data = ll_first(stack->list);
    return 1;
}

// Checa se a pilha esta vazia
// O(1)
int stack_is_empty(Stack* stack)
{
    return ll_is_empty(stack->list);
}

// Libera a memoria alocada para a pilha
// O(n) -- Precisa liberar a memoria de cada no da lista
void stack_free(Stack* stack)
{
    ll_free(stack->list);
    free(stack);
}

#endif
```

Código 5: main.c

```c
#include <stdio.h>
#include "../include/stack.h"

int main(void)
{
    Stack *stack = stack_create();
```

```
7
8       while (1)
9       {
10          float val;
11          if (stack_peek(stack, &val))
12          {
13              printf("Topo da pilha: %.2f\n", val);
14          }
15          else
16          {
17              if (stack_is_empty(stack))
18              {
19                  printf("Pilha esta vazia.\n");
20              }
21              else
22              {
23                  printf("Ocorreu um erro.\n");
24              }
25          }
26
27          printf("\n");
28
29          float newval;
30          printf("\nEmpilhar valor (99 para sair ou 98 para desempilhar): "
                );
31          scanf("%f", &newval);
32
33          if (newval == 99)
34          {
35              break;
36          }
37
38          if (newval == 98)
39          {
40              float popped;
41              if (stack_pop(stack, &popped))
42              {
43                  printf("Desempilhado: %.2f\n", popped);
44              }
45              else
46              {
47                  printf("Nao pode desempilhar (pilha vazia).\n");
48              }
49
50              continue;
51          }
52
53          if (!stack_push(stack, newval))
54          {
55              printf("Nao pode empilhar valor.\n");
56          }
```

```
    }

    stack_free(stack);

    return 0;
}
```

(c) Usando seu TAD pilha, crie uma função para verificar se uma expressão composta apenas por chaves, colchetes e parênteses, representada por uma cadeia, está ou não balanceada. Por exemplo, as expressões "[()()]"e "[([])]"estão balanceadas, mas as expressões "[(])"e "])()(["não estão.

Código 6: expr_check.h

```
#ifndef _EXPR_CHECK_H_
#define _EXPR_CHECK_H_

int valid_expr(char* expr);

#endif
```

Código 7: expr_check.c

```
#include "../include/expr_check.h"
#include "../include/stack.h"

#ifndef _EXPR_CHECK_C_
#define _EXPR_CHECK_C_

// Retorna verdadeiro se o caractere e de fechamento
int is_closing_char(char c)
{
    if (c == ']' || c == '}' || c == ')' || c == '>')
    {
        return 1;
    }
    return 0;
}

// Retorna o caractere de abertura daquele caractere de fechamento
char get_opening_char(char c)
{
    switch (c)
    {
    case ']':
        return '[';

    case '}':
        return '{';
```

```
27
28      case ')':
29          return '(';
30
31      case '>':
32          return '<';
33
34      default:
35          return 'e';
36      }
37      return 'e';
38  }
39
40  // Retorna verdadeiro se a expressao for valida, -1 em caso de erro
41  int valid_expr(char* expr)
42  {
43      // Cria pilha utilizada para avaliar a expressao
44      Stack *st = stack_create();
45      if (!st)
46          return -1;
47
48      // Percorre a string, armazenando os caracteres individuais em c
49      int i = 0;
50      char c = expr[i];
51      while (c != '\0')
52      {
53          // Se o caractere e de fechamento
54          if (is_closing_char(c))
55          {
56              // Desempilha um caractere, que deve ser a abertura de c
57              char popped;
58              if (!stack_pop(st, &popped))
59              {
60                  stack_free(st);
61                  return -1;
62              }
63
64              char target = get_opening_char(c);
65              if (target == 'e')
66              {
67                  stack_free(st);
68                  return -1;
69              }
70
71              // Se o caractere desempilhado nao e a abertura de c, a
                    expressao nao e valida
72              if (popped != target)
73              {
74                  stack_free(st);
75                  return 0;
76              }
```

```
77          }
78          else
79          {
80              // Se ele e de abertura, empilha o caractere
81              if(!stack_push(st, c))
82              {
83                  stack_free(st);
84                  return -1;
85              }
86          }
87
88          c = expr[++i];
89      }
90
91      // Se a pilha nao esta vazia ao final, nao houve o fechamento de
        algum caractere, e a expressao e invalida
92      if(!stack_is_empty(st))
93      {
94          stack_free(st);
95          return 0;
96      }
97
98      // Se nao saiu da funcao ate aqui, a expressao e balanceada
99      stack_free(st);
100     return 1;
101 }
102
103 #endif
```

Código 8: main.c

```
1  #include <stdio.h>
2  #include "../include/expr_check.h"
3
4  int main(void)
5  {
6      char expr[64];
7
8      while (1)
9      {
10         printf("Digite sua expressao (q para sair): ");
11         scanf("%s", expr);
12
13         if (expr[0] == 'q')
14             break;
15
16         int is_valid = valid_expr(expr);
17
18         if (is_valid == -1)
19         {
20             printf("Ocorreu um erro, tente novamente.\n");
```

```
            continue;
        }

        printf("A expressao %s esta balanceada? %d\n\n", expr, is_valid);
    }

    return 0;
}
```

Questão 2.

Sobre Filas:

(a) Explique o funcionamento do TAD Fila.

A fila também é uma estrutura de dados que armazena uma coleção de dados. Porém, na fila, esses dados são acessados seguindo o modelo "first in first out", o que significa que o primeiro dado a ser inserido é o primeiro a ser acessado e removido. A fila mantém uma referência para dois elementos, seu "inicio"e seu "fim". Sempre que um dado novo é inserido na fila, ele passa a ser o seu novo "fim", enquanto o dado que foi inserido a mais tempo, e ainda permanece na fila, é o seu "inicio". Só podemos interagir (isto é, acessar ou remover dados) com o inicio da fila, e sempre que um dado é removido do inicio, o dado que foi inserido logo após este passa a ser o novo inicio. A fila computacional funciona de maneira análoga a uma fila de pessoas.

(b) Crie um TAD de Fila utilizando array estático.

Código 9: queue.h

```h
#ifndef _QUEUE_H_
#define _QUEUE_H_

typedef struct queue Queue;

Queue *q_create(int size);
int q_enqueue(Queue *q, float data);
int q_dequeue(Queue *q, float *data);
int q_peek(Queue *q, float *data);
int q_is_empty(Queue *q);
void q_free(Queue *q);

#endif
```

Código 10: queue.c

```c
#include <stdlib.h>
#include <string.h>

#include "../include/queue.h"

#ifndef _QUEUE_C_
#define _QUEUE_C_

// Struct da fila
struct queue
{
    // Array que armazena a fila
```

```
13      float *ar;
14      // Indice do comeco da fila
15      int front;
16      // Indice do final da fila
17      int back;
18      // Tamanho do array que armazena a fila
19      int dim;
20      // Quantidade de elementos na fila
21      int n;
22  };
23
24  // Cria uma nova fila, retornando seu endereco de memoria
25  Queue *q_create(int size)
26  {
27      Queue *q = (Queue *)malloc(sizeof(Queue));
28
29      // Se a memoria nao foi alocada, retorna nulo
30      if (!q)
31          return NULL;
32
33      q->dim = size;
34      q->front = 0;
35      q->back = 0;
36      q->n = 0;
37
38      // Cria um novo array. Caso falhe, retorna nulo
39      q->ar = (float *)malloc(sizeof(float) * size);
40      if (!q->ar)
41          return NULL;
42
43      return q;
44  }
45
46  // Insere um elemento no fim da fila, retornando 0 em caso de erro
47  int q_enqueue(Queue *q, float data)
48  {
49      // Se a fila esta cheia, retorna 0
50      if (q->n >= q->dim)
51          return 0;
52
53      // Insere o novo elemento no final da fila
54      q->ar[q->back] = data;
55
56      // Incrementa a quantidade de elementos da fila
57      q->n++;
58
59      // Define novo final da fila
60      q->back = (q->back + 1) % q->dim;
61
62      return 1;
63  }
```

```
64
65  // Remove um elemento do comeco da fila, e a retorna pelo parametro de
        saida
66  int q_dequeue(Queue *q, float *data)
67  {
68      if (!q_peek(q, data))
69          return 0;
70
71      // Define novo inicio da fila
72      q->front = (q->front + 1) % q->dim;
73
74      // Decrementa a quantidade de elementos da fila
75      q->n--;
76
77      return 1;
78  }
79
80  // Retorna o elemento no comeco da fila, sem remove-lo
81  int q_peek(Queue *q, float *data)
82  {
83      // Falha caso a fila esteja vazia
84      if (q_is_empty(q))
85          return 0;
86
87      *data = q->ar[q->front];
88
89      return 1;
90  }
91
92  // Checa se a fila esta vazia
93  int q_is_empty(Queue *q)
94  {
95      return q->n == 0;
96  }
97
98  // Libera a memoria alocada para a fila
99  void q_free(Queue *q)
100 {
101     free(q->ar);
102     free(q);
103 }
104
105 #endif
```

Código 11: main.c

```
1  #include <stdio.h>
2  #include "../include/queue.h"
3
4  int main(void)
5  {
```

```
6       int size = 0;
7
8       while (size <= 0)
9       {
10          printf("Digite o tamanho da fila (>=0): ");
11          scanf("%d", &size);
12      }
13
14      Queue *q = q_create(size);
15
16      while (1)
17      {
18          float val;
19          if (q_peek(q, &val))
20          {
21              printf("Comeco da fila: %.2f\n", val);
22          }
23          else
24          {
25              if (q_is_empty(q))
26              {
27                  printf("Fila esta vazia.\n");
28              }
29              else
30              {
31                  printf("Ocorreu um erro.\n");
32              }
33          }
34
35          printf("\n");
36
37          float newval;
38          printf("\nEnfileirar valor (99 para sair ou 98 para desenfileirar
                ): ");
39          scanf("%f", &newval);
40
41          if (newval == 99)
42          {
43              break;
44          }
45
46          if (newval == 98)
47          {
48              float dequeued;
49              if (q_dequeue(q, &dequeued))
50              {
51                  printf("Desenfileirado: %.2f\n", dequeued);
52              }
53              else
54              {
55                  printf("Nao pode desenfileirar: fila vazia.\n");
```

```
            }

            continue;
        }

        if (!q_enqueue(q, newval))
        {
            printf("Nao pode enfileirar valor: fila cheia.\n");
        }
    }

    q_free(q);

    return 0;
}
```

(c) Usando seus TADs fila e pilha, implemente um programa em que o usuário digita cadeias e ele diz, para cada cadeia, se ela é palíndroma ou não.

Código 12: palindrome_checker.h

```
#ifndef _PALINDROME_CHECKER_H_
#define _PALINDROME_CHECKER_H_

int is_palindrome(char *str);

#endif
```

Código 13: palindrome_checker.c

```
#include "../include/queue.h"
#include "../include/stack.h"
#include <stdio.h>

#ifndef _PALINDROME_CHECKER_C_
#define _PALINDROME_CHECKER_C_

// Checa se uma frase e um palindromo
int is_palindrome(char *str)
{
    // Fila para checar os caracteres de frente pra tras
    Queue *q = q_create();
    // Pilha para checar os caracteres de tras pra frente
    Stack *s = stack_create();

    // Enfileira e empilha cada caractere da frase
    int i = 0;
    char c = str[i];
    while (c != '\0')
```

```
20      {
21          c = str[i++];
22          // Nao enfileira o caractere espaco ou nulo
23          if (c == ' ' || c == '\0')
24              continue;
25
26          if (!q_enqueue(q, c))
27          {
28              printf("Falha ao enfileirar caractere %c\n", c);
29              return -1;
30          }
31          if (!stack_push(s, c))
32          {
33              printf("Falha ao empilhar caractere %c\n", c);
34              return -1;
35          }
36      }
37
38      // Desenfileira e desemplilha ao mesmo tempo
39      while (!q_is_empty(q))
40      {
41          char queue_c;
42          if (!q_dequeue(q, &queue_c))
43          {
44              printf("Falha ao desenfileirar caractere %c\n", queue_c);
45              return -1;
46          }
47
48          char stack_c;
49          if (!stack_pop(s, &stack_c))
50          {
51              printf("Falha ao desempilhar caractere %c\n", stack_c);
52              return -1;
53          }
54
55          // Se forem diferentes, nao e palindromo
56          if (stack_c != queue_c)
57              return 0;
58      }
59
60      // Se passou do loop anterior, e um palindromo
61      return 1;
62  }
63
64  #endif
```

Código 14: main.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "../include/palindrome_checker.h"
```

```c

int main(void)
{
    char str[64];

    while (1)
    {
        printf("Digite sua frase (q para sair): ");
        fgets(str, 64, stdin);
        strtok(str, "\n");

        if (str[0] == 'q')
            break;

        int palindrome = is_palindrome(str);

        if (palindrome == -1)
        {
            printf("Ocorreu um erro, tente novamente.\n");
            continue;
        }

        printf("A frase \"%s\" eh um palindromo? %d\n\n", str, palindrome
            );
    }

    return 0;
}
```

Questão 3.

(a) Construa uma função que recebe um vetor e retorne o desvio-padrão dos seus valores.

Código 15: main.c

```c
#include <stdio.h>
#include <math.h>

float standart_deviation(float *ar, int size)
{
    float mean = 0;
    for (int i = 0; i < size; i++)
    {
        mean += ar[i];
    }
    mean /= size;

    float sum = 0;
```

```c
    for (int i = 0; i < size; i++)
    {
        sum += (ar[i] - mean) * (ar[i] - mean);
    }

    return sqrt(sum / size);
}

int main(void)
{
    int ar_size = 0;
    printf("Tamanho do vetor: ");
    scanf("%d", &ar_size);

    float ar[ar_size];

    for (int i = 0; i < ar_size; i++)
    {
        printf("Elemento %d: ", i);
        scanf("%f", &ar[i]);
    }

    printf("\nVetor: ");
    for (int i = 0; i < ar_size; i++)
    {
        printf("%.2f ", ar[i]);
    }

    float sd = standart_deviation(ar, ar_size);
    printf("\nDesvio padrao: %f\n", sd);

    return 0;
}
```

(b) Deduza a equação do número de passos (em função do tamanho do vetor).

Descrevendo as etapas da função, algoritmicamente, temos:

1 Atribuir 0 à variável "mean"

2 Somar cada valor do vetor à variável "mean"

3 Dividir a variável "mean"pelo tamanho do vetor

4 Atribuir 0 à variável "sum"

5 Para cada valor do vetor, somar à variável "sum"a diferença entre o valor e a variável "mean"ao quadrado

6 Dividir a variável "sum"pelo tamanho do vetor

7 Retornar a raiz quadrada do resultado da divisão

Considerando n como o tamanho do vetor, e y como o número de passos, podemos escrever o número de passos matematicamente como:

$$y(n) = 1 + n + 1 + 1 + n + 1 + 1$$

$$y(n) = 2n + 5$$

(c) Determine as complexidades da função construída. Explique.

Observe que:

$$n \leq 2n + 5, \text{ para qualquer } n \geq -5$$

$$3n \geq 2n + 5, \text{ para qualquer } n \geq 5$$

Desta forma, provamos que existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 n \leq y(n) \leq c_2 n$ para todo $n \geq n_0$. Nominalmente, $c_1 = 1,\ c_2 = 3,\ n_0 = 5$.

Assim, a complexidade da função, dá-se por $\Theta(n)$.

Questão 4.

Construa uma função recursiva para calcular o máximo divisor comum de dois números inteiros, usando o algoritmo de Euclides, satisfazendo as seguintes regras:
I. mdc(a,b) = a, se b = 0
II. mdc(a,b) = mdc(b,a mod b), se b > 0
III. mdc(a,b), se b < 0

Código 16: main.c

```c
#include <stdio.h>

int mdc(unsigned int m, unsigned int n)
{
    if (n == 0)
        return m;

    if (n > 0)
        return mdc(n, (m % n));

    if (n > m)
        return mdc(n, m);
}

int main(void)
{
    int m = 0, n = 0;

    printf("Digite a: ");
```

```
20      scanf("%d", &m);
21
22      printf("Digite b: ");
23      scanf("%d", &n);
24
25      int r = mdc(m, n);
26
27      printf("MDC de %d e %d = %d\n", m, n, r);
28
29      return 0;
30  }
```