UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

Laura de Faria Maranhão Ayres - 20180019070

EFA2_ESTRUTURA_DE_DADOS

João Pessoa – PB

2020

QUESTÃO 1

1.1. RESUMO

As pilhas seguem uma lógica simples e específica de armazenamento e retirada dos elementos contidos. O conceito principal se aplica na regra LIFO (Last in first out), na qual apenas o último elemento que foi armazenado se encontra no topo e pode ser retirado da pilha.

Essa estrutura possui algumas operações para manipular e acessar as informações dela. Nisso, as duas operações básicas necessárias são: a operação para empilhar um novo elemento, inserindo-o no topo (Push), e a operação para desempilhar um elemento, removendo-o do topo (Pop). Além disso, funções como criar uma pilha vazia, liberar estrutura ou verificar se a pilha possui elementos, são outras possibilidades de implementação.

Em aplicações computacionais que precisam de uma estrutura de pilha, é comum saber de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nestes casos, a implementação da pilha pode ser feita usando um vetor estático, mas é necessário verificar a capacidade da pilha para inserção.

As filas seguem uma lógica contrária as Pilhas em questão de armazenamentos, usando a regra da FIFO (First in First out), a qual os primeiros elementos que são armazenados serão os primeiros elementos a também serem retirados.

Com isso, assim como nas pilhas, as filas possui alguns comandos básicos para manipulação, sendo eles: Criar uma fila; inserir um novo elemento na fila; remover um elemento da Fila; verificar se a fila está vazia; Liberar a estrutura da fila. Semelhante a pilha, independentemente da estratégia de implementação, podem-se trabalhar com vetor estático, dinâmico, lista encadeada, dupla, dupla com vetor e dupla com lista.

1.2. CÓDIGOS | (LETRAS B-E)

PilhaVet.h

```h
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>

struct st{
    int* elementos;
    int tamanho;
    int total;
};

typedef struct st Stack;

Stack* create_stack();
int stack_size(Stack* stk);
bool stack_empty(Stack* stk);
int stack_push(Stack** stk, int info);
int stack_pop(Stack** stk, int *info);
void clear_stack(Stack** stk);
void print_stack(Stack* stk);
```

PilhaVet.c

```c
Stack* create_stack()               //Cria a pilha
{
    Stack* pilha = (Stack*) malloc(sizeof(Stack));    //Aloca espaço para a pilha

    if(!pilha){
        exit(1);
    }

    pilha->elementos = (int*) malloc(2*sizeof(int));  //Aloca memória para o vetor de elementos

    if(!pilha->elementos){
        exit(1);
    }

    pilha->tamanho = 0;         //Tamanho da pilha
    pilha->total = 2;           //Espaço de armazenamento total

    return pilha;
}
```

```c
int stack_size(Stack* stk)        //Determina o tamanho da pilha
{
    return stk->tamanho;
}

bool stack_empty(Stack* stk)      //Confere se a pilha está vazia
{
    if(!stk){
        return true;
    }
    return false;
}
```

```c
void clear_stack(Stack** stk)     //Esvazia totalmente a pilha
{
        if(*stk == NULL){         //Caso já esteja vazia, apenas retorna
           return;
        }

        free((*stk)->elementos);  //Libera o espaço de memória anteriormente armazenado
        free(*stk);
        *stk = NULL;              //Vetor vazio
}
```

```c
int stack_push(Stack** stk, int info)     //Insere elemento no fim da pilha
{
    if(*stk == NULL){               //Caso não possua pilha ele cria
        *stk = create_stack();
    }

    Stack* pilha = *stk;            //Ponteiro para pilha

    /*Caso tenha atingido seu tamanho máximo*/
    if(pilha->tamanho == pilha->total){  //Compara o tamanho da pilha com seu espaço de armazenamento
        pilha->elementos = (int*) realloc(pilha->elementos, (2* pilha->total) *sizeof(int));

        if(!pilha->elementos){    //Erro ao alocar
            return 1;
        }

        pilha->total = pilha->total*2;   //Dobra tamanho total
    }

    pilha->elementos[pilha->tamanho] = info;    //insere as informações do elemento no final
    pilha->tamanho++;                           //Pilha aumentou de tamanho
    return 0;
}
```

```c
int stack_pop(Stack** stk, int *info)
{
    if(*stk == NULL){
        return 1;
    }

    Stack* pilha = *stk;

    if(pilha->tamanho == 0){
        return 0;
    }

    *info = pilha->elementos[pilha->tamanho-1];
    pilha->tamanho--;

    if(pilha->tamanho == 0){    //quando a pilha está com seu tamanho zerado
        clear_stack(stk);       //apaga os elementos que estavam nela
    }

    return 0;
}
```

```c
void print_stack(Stack* stk)    //Imprime todos os elementos da pilha
{

    int i;
    printf("Pilha = [");

    for(i=0; i < stack_size(stk); i++){        //Percorre a lista e imprime cada termo
        if(stk->elementos[i]){
            printf("%d",stk->elementos[i]);

            if(i != (stack_size(stk)-1)){          //Coloca a vírgura para separar enquanto não chegar no fim
                printf(", ");
            }
        }
    }
    printf("]\n");
}
```

PilhaList.h

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

struct no                                   //Estrutura de cada elemento da lista
{
    int info;                               //Informação da estrutura
    struct no* proximo;                     //Ponteiro para o próximo nó da lista
};

typedef struct no L_no;

L_no * create_no(int info);
int l_size(L_no* lhead);
int empty(L_no* lhead);
void clear_list(L_no** lhead);
int stack_push(L_no** lhead, int info);
int stack_pop(L_no** lhead);
void print_stack(L_no* lhead);
```

PilhaList.c

```c
L_no * create_no(int info)                    //Cria um nó na lista
{
    L_no* nozinho = (L_no*) malloc(sizeof(L_no));   //Aloca o espaço de memória do nó

    if(nozinho){
        nozinho->info = info;                 //Armazena as informações e ponteiro aponta para NULL
        nozinho->proximo = NULL;
    }

    return nozinho;
}
```

```c
int l_size(L_no* lhead)                       //Verifica o tamanho da lista
{
    int cont = 0;

    while(lhead != NULL){  //Percorre a lista, incrementando o contador a cada elemento passado
        cont++;
        lhead = lhead->proximo;
    }

    return cont;                              //Retorna o contador
}
```

```c
int empty(L_no* lhead)                        //Verifica se a lista está vazia
{
    if(lhead == NULL){
        return 1;
    }
    return 0;
}
```

```c
void clear_list(L_no** lhead)                 //Esvazia a lista
{
    L_no *lista = *lhead;
    L_no *lista_proximo;                      //Ponteiro para o próximo elemento da lista

    while(lista != NULL){                     //Libera os elementos da lista
        lista_proximo = lista->proximo;
        free(lista);
        lista = lista_proximo;
    }
    *lhead = NULL;                            //Cabeça da lista aponta para NULL, indicando que está vazia
}
```

```c
int stack_push(L_no** lhead, int info)        //Insere um elemento no fim da pilha
{
    L_no* stk = create_no(info);

    if(stk){
        if(empty(*lhead)){                    //Caso a lista esteja vazia, insere na primeira posição

            stk->proximo = *lhead;
            *lhead = stk;
            return 0;

        }

        L_no* lista = *lhead;
        while(lista->proximo != NULL){        //Avança até a última posição da pilha e insere
            lista = lista->proximo;
        }

        lista->proximo = stk;
        return 0;
    }

    return 1;
}
```

```c
int stack_pop(L_no** lhead)
{
    /* Verifica se a lista está vazia */
    if(empty(*lhead)){
        return 1;
    }

    L_no* lista = *lhead;                    //Ponteiro para percorrer a lista

    /* Procura o último elemento na lista*/
    while (lista->proximo != NULL) {
        lista = lista->proximo;
    }

    /* Retira esse elemento (LIFO) */
    *lhead = lista;
    free(lista);

    return 0;
}
```

```c
void print_stack(L_no* lhead)              //Imprime todos os elementos da pilha
{

    L_no* lista = lhead;
    int i;
    printf("Pilha = [");

    for(i=0; i < l_size(lhead); i++){      //Percorre a lista e imprime cada termo
        printf("%d",lista->info);
        lista = lista->proximo;

        if(i != (l_size(lhead)-1)){        //Coloca a virgura para separar enquanto não chegar no fim
            printf(", ");
        }
    }
    printf("]\n");
}
```

FilaVet.h

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

struct qu{
    int* elementos;
    int tamanho;
    int inicio;
    int final;
    int total;
};

typedef struct qu Queue;

Queue* create_queue();
int queue_size(Queue* q);
bool queue_empty(Queue* q);
int queue_insert(Queue** q, int info);
int queue_remove(Queue** q, int *info);
void clear_queue(Queue** q);
void print_queue(Queue* q);
```

FilaVet.c

Comentário extra: A variável “Final” é um parâmetro para o tamanho verdadeiro do vetor, uma vez que com a utilização do realloc durante a remoção de um termos o tamanho do vetor é reduzido, contudo o espaço antes armazenado para o termo removido permanece. Assim, junto com a variável “início” torna-se possível manipular o tamanho da fila após a realocação de memória. Diferente disso, a variável “tamanho” indica a quantidade de elementos no vetor, ao invés do espaço alocado.

```
3   Queue* create_queue()
4   {
5       Queue* fila = (Queue*) malloc(sizeof(Queue));      //Aloca espaço para a fila
6
7       if(!fila){
8           exit(1);
9       }
10
11      fila->elementos = (int*) malloc(2*sizeof(int));    //Aloca memória para o vetor de elementos
12
13      if(!fila->elementos){
14          exit(1);
15      }
16
17      fila->tamanho = 0;              //Tamanho da fila
18      fila->final = 0;                //Utilizado para localizar o final da fila
19      fila->inicio = 0;               //Utilizado para determinar o início da fila
20      fila->total = 2;                //Espaço de armazenamento total
21
22      return fila;
23  }
```

```
25  int queue_size(Queue* q)                    //Determina o tamanho da fila
26  {
27      return q->tamanho;
28  }
29
30  bool queue_empty(Queue* q)                  //Confere se a fila está vazia
31  {
32      if(!q){
33          return true;
34      }
35      return false;
36  }
```

```c
int queue_insert(Queue** q, int info)       //Insere elemento no fim da fila
{
    if(*q == NULL){               //Caso não possua fila ele cria
        *q = create_queue();
    }

    Queue* fila = *q;            //Ponteiro para fila

    /*Caso tenha atingido seu tamanho máximo*/
    if(fila->final == fila->total){  //Compara o tamanho da fila com seu espaço de armazenamento
        fila->elementos = (int*) realloc(fila->elementos, (2* fila->total) *sizeof(int));

        if(!fila->elementos){   //Erro ao alocar
            return 1;
        }

        fila->total = fila->total*2;      //Dobra tamanho total
    }

    fila->elementos[fila->final] = info;  //insere as informações do elemento no final
    fila->final++;                        //Informa que a fila aumentou de tamanho (+1)
    fila->tamanho++;                      //Fila aumentou de tamanho
    return 0;
}
```

```c
int queue_remove(Queue** q, int *info)      //Retira elemento do início da fila
{
    if(*q == NULL){
        return 1;
    }

    Queue* fila = *q;

    if(fila->tamanho == 0){                 //Fila vazia
        return 0;
    }

    *info = fila->elementos[fila->inicio];
    fila->inicio++;            //Aponta para segundo elemento do vetor
    fila->tamanho--;           //Diminui o tamanho da fila

    if(fila->tamanho == 0){    //quando a fila está com seu tamanho zerado
        clear_queue(q);       //libera o espaço de memória dos elementos retirados
    }

    return 0;
}
```

```c
void clear_queue(Queue** q)     //Esvazia totalmente a fila
{
        if(*q == NULL){          //Caso já esteja vazia, apenas retorna
            return;
        }

        free((*q)->elementos);  //Libera o espaço de memória anteriormente armazenado
        free(*q);
        *q = NULL;              //Vetor vazio
}
```

```c
void print_queue(Queue* q)    //Imprime todos os elementos da fila
{

    int i;
    printf("Fila = [");

    for(i=0; i < queue_size(q); i++){        //Percorre a lista e imprime cada termo
        if(q->elementos[i]){
            printf("%d",q->elementos[i]);

            if(i != (queue_size(q)-1)){          //Coloca a virgura para separar enquanto não chegar no fim
                printf(", ");
            }
        }
    }
    printf("]\n");
}
```

FilaList.h

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

struct no                                       //Estrutura de cada elemento da lista
{
    int info;                                   //Informação da estrutura
    struct no* proximo;                         //Ponteiro para o próximo nó da lista
};

typedef struct no L_no;

L_no * create_no(int info);
int l_size(L_no* lhead);
int empty(L_no* lhead);
void clear_list(L_no** lhead);
int queue_insert(L_no** lhead, int info);
int queue_remove(L_no** lhead);
void print_queue(L_no* lhead);
```

FilaList.c

```c
L_no * create_no(int info)                      //Cria um nó na lista
{
    L_no* nozinho = (L_no*) malloc(sizeof(L_no));    //Aloca o espaço de memória do nó

    if(nozinho){                                //Armazena infomações e aponta para NULL
        nozinho->info = info;
        nozinho->proximo = NULL;
    }

    return nozinho;
}
```

```c
int l_size(L_no* lhead)                         //Verifica o tamnho da lista
{
    int cont = 0;

    while(lhead != NULL){  //Percorre a lista, incrementando o contador a cada elemento passado
        cont++;
        lhead = lhead->proximo;
    }

    return cont;
}
```

```c
int empty(L_no* lhead)                          //Verifica se a lista está vazia
{
    if(lhead == NULL){
        return 1;
    }
    return 0;
}
```

```c
void clear_list(L_no** lhead)                   //Esvazia a lista
{
    L_no *lista = *lhead;
    L_no *lista_proximo;                        //Ponteiro para o próximo elemento da lista

    while(lista != NULL){                       //Libera os elementos da lista
        lista_proximo = lista->proximo;
        free(lista);
        lista = lista_proximo;
    }
    *lhead = NULL;                              //Cabeça da lista aponta para NULL, indicando que está vazia
}
```

```c
int queue_insert(L_no** lhead, int info)        //Insere um elemento no fim da fila
{
    L_no* qu = create_no(info);

    if(qu){

        if(empty(*lhead)){                          //Caso a lista esteja vazia, insere na primeira posição

            qu->proximo = *lhead;
            *lhead = qu;
            return 0;

        }

        L_no* lista = *lhead;
        while(lista->proximo != NULL){              //Avança até a última posição da fila e insere
            lista = lista->proximo;
        }

        lista->proximo = qu;
        return 0;

    }

    return 1;
}
```

```c
int queue_remove(L_no** lhead)                  //Remove elemento do início da fila
{
    /* Verifica se a lista está vazia */
    if(empty(*lhead)){
        return 1;
    }

    L_no* lista = *lhead;                       //Ponteiro para percorrer a lista

    /* Retira o primeiro elemento da fila (FIFO) */
    *lhead = lista->proximo;
    free(lista);

    return 0;
}
```

```c
void print_queue(L_no* lhead)               //Imprime todos os elementos da fila
{

    L_no* lista = lhead;
    int i;
    printf("Fila = [");

    for(i=0; i < l_size(lhead); i++){       //Percorre a lista e imprime cada termo
        printf("%d",lista->info);
        lista = lista->proximo;

        if(i != (l_size(lhead)-1)){         //Coloca a vírgura para separar enquanto não chegar no fim
            printf(", ");
        }
    }
    printf("]\n");
}
```

QUESTÃO 2

2.1. RESUMO

A tabela hash é uma estrutura de dado elaborada para otimizar os acessos aos elementos inseridos em sua estrutura interna (podendo acessá-los em tempo constante O(1)), porém, aumentando o custo de memória exigido de forma proporcional a quantidades de elementos que se deseja inserir. A estrutura de dado consiste em gerar um código (através de uma função hash) para o aluno que se deseja inserir, e, por meio deste identificador, armazenar em um vetor interno. Caso já haja um valor armazenado na estrutura interna para um dado código, há uma colisão.

Em caso de colisões, a tabela hash pode ser implementada de duas formas, enquanto uma busca um novo lugar para inserir o elemento, o outro adiciona este junto ao que já está armazenado. A primeira pode ser implementada de duas formas distintas, enquanto uma busca o próximo espaço livre do vetor (custando mais interações para inserção e busca), o outro gera um novo código hash, inserindo-o em uma outra casa vazia do vetor, que não necessariamente é a próxima livre. Desta forma, pode-se acessar o elemento apenas recalculando através dessa “dispersão dupla”.

Uma terceira forma de lidar com as colisões está em um vetor que armazena listas encadeadas. Assim, ao lidar com uma colisão, basta inserir no fim da lista localizada para onde o elemento foi mapeado. Quando for necessário buscar o elemento, basta calcular um único código hash e manipular o elemento através da própria lista.

2.2. CÓDIGOS | LETRAS (B-C)

Hash_PosicaoLivre.h

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

typedef struct pes Pessoa;
typedef struct hash Hash;
int hash(Hash* tabela, int cpf);
Hash* hash_create();
Pessoa* pes_create(char* nome, int cpf);
void hash_free(Hash** tabela);
void redimenciona (Hash** tabela);
Pessoa* hash_search(Hash* tabela, int cpf);
Pessoa* hash_insert(Hash** tabela, Pessoa* indv);
void print hash(Hash *tabela);
```

Hash_PosicaoLivre.c

```c
#include "Hash_PosicaoLivre.h"

struct pes{
    char nome[100];
    int cpf;          // 5 ultimos números do cpf da pessoa
};

struct hash{          //struct da tabela hash
    int tamanho;
    int elementos;
    Pessoa** vetor;
};

int hash(Hash* tabela, int cpf)
{
    return cpf%tabela->tamanho;
}

Hash* hash_create()
{
    int i;
    Hash* h = (Hash*) malloc(sizeof (Hash));   //Aloca memória tabela
    h->vetor = (Pessoa**) malloc(13*sizeof(Pessoa*)); //Aloca memória vetor

    for(i=0; i<13 ; i++){
        h-> vetor[i] = NULL;
    }

    h->tamanho = 13;                //tamanho total
    h->elementos = 0;

    return h;
}
```

```c
Pessoa* pes_create(char* nome, int cpf)
{
    Pessoa* pes = (Pessoa*) malloc(sizeof(Pessoa));

    strcpy(pes->nome, nome);
    pes->cpf = cpf;

    return pes;
}

void hash_free(Hash** tabela)
{
    int i;
    Hash* tab = *tabela;

    for(i=0; i< tab->tamanho; i++){
        if(tab->vetor[i])
            free(tab->vetor[i]);
    }

    free(tab->vetor);
    free(*tabela);
}
```

```c
void redimenciona (Hash** tabela)
{

    Hash* hsh = *tabela;
    int i;
    int anterior = hsh->tamanho;
    Pessoa** pes = hsh->vetor;

    hsh->elementos = 0;
    hsh->tamanho *= 1.947;
    hsh->vetor = (Pessoa**) malloc(hsh->tamanho*sizeof(Pessoa*));

    for(i=0; i< hsh->tamanho; ++i) {
        hsh->vetor[i] = NULL;
    }
    for(i=0; i<anterior; ++i){
        if(pes[i]){
            hash_insert(tabela, pes[i]);
        }
    }

    free(pes);
}
```

```c
Pessoa* hash_search(Hash* tabela, int cpf)
{
    int h = hash(tabela, cpf);

    while(tabela->vetor[h] != NULL){
        if(tabela->vetor[h]->cpf == cpf){
            return tabela->vetor[h];
        }
        h = (h+1) % tabela->tamanho;
    }

    return NULL;
}

Pessoa* hash_insert(Hash** tabela, Pessoa* indv)
{
    int h = hash(*tabela, indv->cpf);
    Hash* tab = *tabela;

    if(tab->elementos>(0.75*tab->tamanho)){
        redimenciona(tabela);
    }

    while(tab->vetor[h] != NULL){
        h = (h+1) % tab->tamanho;
    }

    tab->vetor[h] = indv;
    tab-> elementos++;

    return indv;
}
```

```c
void print_hash(Hash *tabela)
{
    int i;
    for (i = 0; i < tabela->tamanho; i++){

        if (tabela->vetor[i] != NULL){

            printf("Nome: %s, CPF: %d\n", tabela->vetor[i]->nome, tabela->vetor[i]->cpf);
        }
    }
}
```

Hash_DispersaoDupla.h

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

typedef struct pes Pessoa;
typedef struct hash Hash;
int hash_2(Hash* tabela, int cpf)
int hash(Hash* tabela, int cpf);
Hash* hash_create();
Pessoa* pes_create(char* nome, int cpf);
void hash_free(Hash** tabela);
void redimenciona (Hash** tabela);
Pessoa* hash_search(Hash* tabela, int cpf);
Pessoa* hash_insert(Hash** tabela, Pessoa* indv);
void print_hash(Hash *tabela);
```

Hash_DispersaoDupla.c

```c
#include "Hash_DispersaoDupla.h"

struct pes{
    char nome[100];
    int cpf;          // 5 ultimos números do cpf da pessoa
};

struct hash{          //struct da tabela hash
    int tamanho;
    int elementos;
    Pessoa** vetor;
};

int hash_2(Hash* tabela, int cpf){
    return  tabela->tamanho - 2 - cpf%(tabela->tamanho - 2);
}

int hash(Hash* tabela, int cpf)
{
    return cpf%tabela->tamanho;
}
```

```c
Hash* hash_create()
{
    int i;
    Hash* h = (Hash*) malloc(sizeof (Hash));   //Aloca memória tabela
    h->vetor = (Pessoa**) malloc(13*sizeof(Pessoa*)); //Aloca memória vetor

    for(i=0; i<13 ; i++){
        h-> vetor[i] = NULL;
    }

    h->tamanho = 13;                  //tamanho total
    h->elementos = 0;

    return h;
}

Pessoa* pes_create(char* nome, int cpf)
{
    Pessoa* pes = (Pessoa*) malloc(sizeof(Pessoa));

    strcpy(pes->nome, nome);
    pes->cpf = cpf;

    return pes;
}
```

```c
void hash_free(Hash** tabela)
{
    int i;
    Hash* tab = *tabela;

    for(i=0; i< tab->tamanho; i++){
        if(tab->vetor[i])
            free(tab->vetor[i]);
    }

    free(tab->vetor);
    free(*tabela);
}
```

```c
void redimenciona (Hash** tabela)
{

    Hash* hsh = *tabela;
    int i;
    int anterior = hsh->tamanho;
    Pessoa** pes = hsh->vetor;

    hsh->elementos = 0;
    hsh->tamanho *= 1.947;
    hsh->vetor = (Pessoa**) malloc(hsh->tamanho*sizeof(Pessoa*));

    for(i=0; i< hsh->tamanho; ++i) {
        hsh->vetor[i] = NULL;
    }
    for(i=0; i<anterior; ++i){
        if(pes[i]){
            hash_insert(tabela, pes[i]);
        }
    }

    free(pes);
}
```

```c
Pessoa* hash_search(Hash* tabela, int cpf)
{
    int h = hash(tabela, cpf);
    int h2 = hash2(tabela, cpf);

    while(tabela->vetor[h] != NULL){
        if(tabela->vetor[h]->cpf == cpf){
            return tabela->vetor[h];
        }
        h = (h+h2) % tabela->tamanho;
    }

    return NULL;
}

Pessoa* hash_insert(Hash** tabela, Pessoa* indv)
{
    int h = hash(*tabela, indv->cpf);
    int h2 = hash2(*tabela, indv->cpf);
    Hash* tab = *tabela;

    if(tab->elementos>(0.75*tab->tamanho)){
        redimenciona(tabela);
    }

    while(tab->vetor[h] != NULL){
        h = (h+h2) % tab->tamanho;
    }

    tab->vetor[h] = indv;
    tab-> elementos++;

    return indv;
}
```

```c
void print_hash(Hash *tabela)
{
    int i;
    for (i = 0; i < tabela->tamanho; i++){

        if (tabela->vetor[i] != NULL){

            printf("Nome: %s, CPF: %d\n", tabela->vetor[i]->nome, tabela->vetor[i]->cpf);
        }
    }
}
```

QUESTÃO 3 | LETRAS B-E

Legendas:
O(1) - Não realiza instruções que dependem de n elementos da pilha/fila, ou seja, cada linha é atribuída a uma única operação, então a complexidade é constante (constante * 1).
O(n) - Possui dentro da função um for/while, realizando constante * n instruções dentro deles.
O(n²) - Ocorre quando possui dois laços de repetição, sendo um dentro do outro (ex: for dentro de outro for), necessitando de constante * n² instruções.

```
"PilhaVet.h"

Stack* create_stack(); -> O(1)

int stack_size(Stack* stk); -> O(1)

bool stack_empty(Stack* stk); -> O(1)

int stack_push(Stack** stk, int info); -> O(1)

int stack_pop(Stack** stk, int *info); -> O(1)

void clear_stack(Stack** stk); -> O(1)

void print_stack(Stack* stk); -> Melhor caso: O(1); Médio e pior caso: O(n)
```

```
"PilhaList.h"

L_no * create_no(int info); -> O(1)

int l_size(L_no* lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)

int empty(L_no* lhead); -> O(1)

void clear_list(L_no** lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)

int stack_push(L_no** lhead, int info); -> Melhor caso: O(1); Médio e pior caso: O(n)

int stack_pop(L_no** lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)

void print_stack(L_no* lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)
```

```
"FilaVet.h"

Queue* create_queue(); -> O(1)

int queue_size(Queue* q); -> O(1)

bool queue_empty(Queue* q); -> O(1)

int queue_insert(Queue** q, int info); -> O(1)

int queue_remove(Queue** q, int *info); -> O(1)

void clear_queue(Queue** q); -> O(1)

void print_queue(Queue* q); ->  Melhor caso: O(1); Médio e pior caso: O(n)
```

```
"FilaList.h"

L_no * create_no(int info); -> O(1)

int l_size(L_no* lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)

int empty(L_no* lhead); -> O(1)

void clear_list(L_no** lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)

int queue_insert(L_no** lhead, int info); ->  Melhor caso: O(1); Médio e pior caso: O(n)

int queue_remove(L_no** lhead); -> O(1)

void print_queue(L_no* lhead); ->  Melhor caso: O(1); Médio e pior caso: O(n)
```