



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

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <conio.h>
5
6  struct st{
7      int* elementos;
8      int tamanho;
9      int total;
10 };
11
12 typedef struct st Stack;
13
14 Stack* create_stack();
15 int stack_size(Stack* stk);
16 bool stack_empty(Stack* stk);
17 int stack_push(Stack** stk, int info);
18 int stack_pop(Stack** stk, int *info);
19 void clear_stack(Stack** stk);
20 void print_stack(Stack* stk);
```

PilhaVet.c

```
3  Stack* create_stack()                //Cria a pilha
4  {
5      Stack* pilha = (Stack*) malloc(sizeof(Stack)); //Aloca espaço para a pilha
6
7      if(!pilha){
8          exit(1);
9      }
10
11     pilha->elementos = (int*) malloc(2*sizeof(int)); //Aloca memória para o vetor de elementos
12
13     if(!pilha->elementos){
14         exit(1);
15     }
16
17     pilha->tamanho = 0;                //Tamanho da pilha
18     pilha->total = 2;                  //Espaço de armazenamento total
19
20     return pilha;
21 }
```

```

23 int stack_size(Stack* stk)           //Determina o tamanho da pilha
24 {
25     return stk->tamanho;
26 }
27
28 bool stack_empty(Stack* stk)         //Confere se a pilha está vazia
29 {
30     if(!stk){
31         return true;
32     }
33     return false;
34 }

```

```

82 void clear_stack(Stack** stk)        //Esvazia totalmente a pilha
83 {
84     if(*stk == NULL){                //Caso já esteja vazia, apenas retorna
85         return;
86     }
87
88     free((*stk)->elementos); //Libera o espaço de memória anteriormente armazenado
89     free(*stk);
90     *stk = NULL;                //Vetor vazio
91 }

```

```

36 int stack_push(Stack** stk, int info) //Insere elemento no fim da pilha
37 {
38     if(*stk == NULL){                //Caso não possua pilha ele cria
39         *stk = create_stack();
40     }
41
42     Stack* pilha = *stk;             //Ponteiro para pilha
43
44     /*Caso tenha atingido seu tamanho máximo*/
45     if(pilha->tamanho == pilha->total){ //Compara o tamanho da pilha com seu espaço de armazenamento
46         pilha->elementos = (int*) realloc(pilha->elementos, (2* pilha->total) *sizeof(int));
47
48         if(!pilha->elementos){        //Erro ao alocar
49             return 1;
50         }
51
52         pilha->total = pilha->total*2; //Dobra tamanho total
53     }
54
55     pilha->elementos[pilha->tamanho] = info; //insere as informações do elemento no final
56     pilha->tamanho++;                  //Pilha aumentou de tamanho
57     return 0;
58 }

```

```

60 int stack_pop(Stack** stk, int *info)
61 {
62     if(*stk == NULL){
63         return 1;
64     }
65
66     Stack* pilha = *stk;
67
68     if(pilha->tamanho == 0){
69         return 0;
70     }
71
72     *info = pilha->elementos[pilha->tamanho-1];
73     pilha->tamanho--;
74
75     if(pilha->tamanho == 0){ //quando a pilha está com seu tamanho zerado
76         clear_stack(stk); //apaga os elementos que estavam nela
77     }
78
79     return 0;
80 }

93 void print_stack(Stack* stk) //Imprime todos os elementos da pilha
94 {
95
96     int i;
97     printf("Pilha = [");
98
99     for(i=0; i < stack_size(stk); i++){ //Percorre a lista e imprime cada termo
100         if(stk->elementos[i]){
101             printf("%d", stk->elementos[i]);
102
103             if(i != (stack_size(stk)-1)){ //Coloca a vírgula para separar enquanto não chegar no fim
104                 printf(", ");
105             }
106         }
107     }
108     printf("]\n");
109 }

```

PilhaList.h

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <conio.h>
5
6  struct no //Estrutura de cada elemento da lista
7  {
8      int info; //Informação da estrutura
9      struct no* proximo; //Ponteiro para o próximo nó da lista
10 };
11
12 typedef struct no L_no;
13
14 L_no * create_no(int info);
15 int l_size(L_no* lhead);
16 int empty(L_no* lhead);
17 void clear_list(L_no** lhead);
18 int stack_push(L_no** lhead, int info);
19 int stack_pop(L_no** lhead);
20 void print_stack(L_no* lhead);

```

PilhaList.c

```
3  L_no * create_no(int info)                //Cria um nó na lista
4  {
5      L_no* nozinho = (L_no*) malloc(sizeof(L_no)); //Aloca o espaço de memória do nó
6
7      if(nozinho){
8          nozinho->info = info;                //Armazena as informações e ponteiro aponta para NULL
9          nozinho->proximo = NULL;
10     }
11
12     return nozinho;
13 }

15 int l_size(L_no* lhead)                   //Verifica o tamanho da lista
16 {
17     int cont = 0;
18
19     while(lhead != NULL){ //Percorre a lista, incrementando o contador a cada elemento passado
20         cont++;
21         lhead = lhead->proximo;
22     }
23
24     return cont;                          //Retorna o contador
25 }

27 int empty(L_no* lhead)                    //Verifica se a lista está vazia
28 {
29     if(lhead == NULL){
30         return 1;
31     }
32     return 0;
33 }

35 void clear_list(L_no** lhead)             //Esvazia a lista
36 {
37     L_no *lista = *lhead;
38     L_no *lista_proximo;                  //Ponteiro para o próximo elemento da lista
39
40     while(lista != NULL){                  //Libera os elementos da lista
41         lista_proximo = lista->proximo;
42         free(lista);
43         lista = lista_proximo;
44     }
45     *lhead = NULL;                        //Cabeça da lista aponta para NULL, indicando que está vazia
46 }

48 int stack_push(L_no** lhead, int info)    //Insere um elemento no fim da pilha
49 {
50     L_no* stk = create_no(info);
51
52     if(stk){
53         if(empty(*lhead)){                //Caso a lista esteja vazia, insere na primeira posição
54             stk->proximo = *lhead;
55             *lhead = stk;
56             return 0;
57         }
58
59         L_no* lista = *lhead;
60         while(lista->proximo != NULL){    //Avança até a última posição da pilha e insere
61             lista = lista->proximo;
62         }
63
64         lista->proximo = stk;
65         return 0;
66     }
67
68     return 1;
69 }
70
71 }
```



```

73 int stack_pop(L_no** lhead)
74 {
75     /* Verifica se a lista está vazia */
76     if(empty(*lhead)){
77         return 1;
78     }
79
80     L_no* lista = *lhead;           //Ponteiro para percorrer a lista
81
82     /* Procura o último elemento na lista*/
83     while (lista->proximo != NULL) {
84         lista = lista->proximo;
85     }
86
87     /* Retira esse elemento (LIFO) */
88     *lhead = lista;
89     free(lista);
90
91     return 0;
92 }

94 void print_stack(L_no* lhead)       //Imprime todos os elementos da pilha
95 {
96     L_no* lista = lhead;
97     int i;
98     printf("Pilha = ");
99
100     for(i=0; i < l_size(lhead); i++){ //Percorre a lista e imprime cada termo
101         printf("%d", lista->info);
102         lista = lista->proximo;
103
104         if(i != (l_size(lhead)-1)){ //Coloca a vírgula para separar enquanto não chegar no fim
105             printf(", ");
106         }
107     }
108     printf("\n");
109 }
110

```

FilaVet.h

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <conio.h>
5
6  struct qu{
7      int* elementos;
8      int tamanho;
9      int inicio;
10     int final;
11     int total;
12 };
13
14 typedef struct qu Queue;
15
16 Queue* create_queue();
17 int queue_size(Queue* q);
18 bool queue_empty(Queue* q);
19 int queue_insert(Queue** q, int info);
20 int queue_remove(Queue** q, int *info);
21 void clear_queue(Queue** q);
22 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 inicio da fila
20     fila->total = 2;    //Espaço de armazenamento total
21
22     return fila;
23 }
24
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 }
```



```

38 int queue_insert(Queue** q, int info) //Insere elemento no fim da fila
39 {
40     if(*q == NULL){ //Caso não possua fila ele cria
41         *q = create_queue();
42     }
43
44     Queue* fila = *q; //Ponteiro para fila
45
46     /*Caso tenha atingido seu tamanho máximo*/
47     if(fila->final == fila->total){ //Compara o tamanho da fila com seu espaço de armazenamento
48         fila->elementos = (int*) realloc(fila->elementos, (2* fila->total) *sizeof(int));
49
50         if(!fila->elementos){ //Erro ao alocar
51             return 1;
52         }
53
54         fila->total = fila->total*2; //Dobra tamanho total
55     }
56
57     fila->elementos[fila->final] = info; //insere as informações do elemento no final
58     fila->final++; //Informa que a fila aumentou de tamanho (+1)
59     fila->tamanho++; //Fila aumentou de tamanho
60     return 0;
61 }

```

```

63 int queue_remove(Queue** q, int *info) //Retira elemento do início da fila
64 {
65     if(*q == NULL){
66         return 1;
67     }
68
69     Queue* fila = *q;
70
71     if(fila->tamanho == 0){ //Fila vazia
72         return 0;
73     }
74
75     *info = fila->elementos[fila->inicio];
76     fila->inicio++; //Aponta para segundo elemento do vetor
77     fila->tamanho--; //Diminui o tamanho da fila
78
79     if(fila->tamanho == 0){ //quando a fila está com seu tamanho zerado
80         clear_queue(q); //libera o espaço de memória dos elementos retirados
81     }
82
83     return 0;
84 }

```

```

86 void clear_queue(Queue** q) //Esvazia totalmente a fila
87 {
88     if(*q == NULL){ //Caso já esteja vazia, apenas retorna
89         return;
90     }
91
92     free((*q)->elementos); //Libera o espaço de memória anteriormente armazenado
93     free(*q);
94     *q = NULL; //Vetor vazio
95 }

```

```

97 void print_queue(Queue* q)    //Imprime todos os elementos da fila
98 {
99
100     int i;
101     printf("Fila = [");
102
103     for(i=0; i < queue_size(q); i++){    //Percorre a lista e imprime cada termo
104         if(q->elementos[i]){
105             printf("%d", q->elementos[i]);
106
107             if(i != (queue_size(q)-1)){    //Coloca a vírgula para separar enquanto não chegar no fim
108                 printf(", ");
109             }
110         }
111     }
112     printf("]\n");
113 }

```

FilaList.h

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <conio.h>
5
6  struct no    //Estrutura de cada elemento da lista
7  {
8      int info;    //Informação da estrutura
9      struct no* proximo;    //Ponteiro para o próximo nó da lista
10 }
11
12 typedef struct no L_no;
13
14 L_no * create_no(int info);
15 int l_size(L_no* lhead);
16 int empty(L_no* lhead);
17 void clear_list(L_no** lhead);
18 int queue_insert(L_no** lhead, int info);
19 int queue_remove(L_no** lhead);
20 void print_queue(L_no* lhead);

```

FilaList.c

```

3  L_no * create_no(int info)    //Cria um nó na lista
4  {
5      L_no* nozinho = (L_no*) malloc(sizeof(L_no));    //Aloca o espaço de memória do nó
6
7      if(nozinho){    //Armazena informações e aponta para NULL
8          nozinho->info = info;
9          nozinho->proximo = NULL;
10     }
11
12     return nozinho;
13 }
14
15 int l_size(L_no* lhead)    //Verifica o tamanho da lista
16 {
17     int cont = 0;
18
19     while(lhead != NULL){    //Percorre a lista, incrementando o contador a cada elemento passado
20         cont++;
21         lhead = lhead->proximo;
22     }
23
24     return cont;
25 }

```

```

27 int empty(L_no* lhead)                                //Verifica se a lista está vazia
28 {
29     if(lhead == NULL){
30         return 1;
31     }
32     return 0;
33 }

35 void clear_list(L_no** lhead)                          //Esvazia a lista
36 {
37     L_no *lista = *lhead;
38     L_no *lista_proximo;
39     //Ponteiro para o próximo elemento da lista
40     while(lista != NULL){
41         lista_proximo = lista->proximo;
42         free(lista);
43         lista = lista_proximo;
44     }
45     *lhead = NULL;
46     //Cabeça da lista aponta para NULL, indicando que está vazia

48 int queue_insert(L_no** lhead, int info)               //Insere um elemento no fim da fila
49 {
50     L_no* qu = create_no(info);
51
52     if(qu){
53
54         if(empty(*lhead)){
55             //Caso a lista esteja vazia, insere na primeira posição
56             qu->proximo = *lhead;
57             *lhead = qu;
58             return 0;
59         }
60
61         L_no* lista = *lhead;
62         while(lista->proximo != NULL){
63             lista = lista->proximo;
64         }
65         //Avança até a última posição da fila e insere
66
67         lista->proximo = qu;
68         return 0;
69     }
70
71     return 1;
72 }
73

75 int queue_remove(L_no** lhead)                        //Remove elemento do início da fila
76 {
77     /* Verifica se a lista está vazia */
78     if(empty(*lhead)){
79         return 1;
80     }
81
82     L_no* lista = *lhead;
83     //Ponteiro para percorrer a lista
84
85     /* Retira o primeiro elemento da fila (FIFO) */
86     *lhead = lista->proximo;
87     free(lista);
88
89     return 0;

```

```

91 void print_queue(L_no* lhead) //Imprime todos os elementos da fila
92 {
93
94     L_no* lista = lhead;
95     int i;
96     printf("Fila = [");
97
98     for(i=0; i < l_size(lhead); i++){ //Percorre a lista e imprime cada termo
99         printf("%d", lista->info);
100         lista = lista->proximo;
101
102         if(i != (l_size(lhead)-1)){ //Coloca a vírgula para separar enquanto não chegar no fim
103             printf(", ");
104         }
105     }
106     printf("]\n");
107 }

```

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

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4  #include<conio.h>
5
6  typedef struct pes Pessoa;
7  typedef struct hash Hash;
8  int hash(Hash* tabela, int cpf);
9  Hash* hash_create();
10 Pessoa* pes_create(char* nome, int cpf);
11 void hash_free(Hash** tabela);
12 void redimenciona (Hash** tabela);
13 Pessoa* hash_search(Hash* tabela, int cpf);
14 Pessoa* hash_insert(Hash** tabela, Pessoa* indiv);
15 void print hash(Hash *tabela);
```

Hash_PosicaoLivre.c

```
1  #include "Hash_PosicaoLivre.h"
2
3  struct pes{
4      char nome[100];
5      int cpf;          // 5 ultimos números do cpf da pessoa
6  };
7
8  struct hash{          //struct da tabela hash
9      int tamanho;
10     int elementos;
11     Pessoa** vetor;
12 };
13
14 int hash(Hash* tabela, int cpf)
15 {
16     return cpf%tabela->tamanho;
17 }
18
19 Hash* hash_create()
20 {
21     int i;
22     Hash* h = (Hash*) malloc(sizeof (Hash));    //Aloca memória tabela
23     h->vetor = (Pessoa**) malloc(13*sizeof(Pessoa*)); //Aloca memória vetor
24
25     for(i=0; i<13 ; i++){
26         h->vetor[i] = NULL;
27     }
28
29     h->tamanho = 13;          //tamanho total
30     h->elementos = 0;
31
32     return h;
33 }
```



```

35 Pessoa* pes_create(char* nome, int cpf)
36 {
37     Pessoa* pes = (Pessoa*) malloc(sizeof(Pessoa));
38
39     strcpy(pes->nome, nome);
40     pes->cpf = cpf;
41
42     return pes;
43 }
44
45 void hash_free(Hash** tabela)
46 {
47     int i;
48     Hash* tab = *tabela;
49
50     for(i=0; i< tab->tamanho; i++){
51         if(tab->vetor[i])
52             free(tab->vetor[i]);
53     }
54
55     free(tab->vetor);
56     free(*tabela);
57 }

```

```

59 void redimenciona (Hash** tabela)
60 {
61
62     Hash* hsh = *tabela;
63     int i;
64     int anterior = hsh->tamanho;
65     Pessoa** pes = hsh->vetor;
66
67     hsh->elementos = 0;
68     hsh->tamanho *= 1.947;
69     hsh->vetor = (Pessoa**) malloc(hsh->tamanho*sizeof(Pessoa));
70
71     for(i=0; i< hsh->tamanho; ++i) {
72         hsh->vetor[i] = NULL;
73     }
74     for(i=0; i<anterior; ++i){
75         if(pes[i]){
76             hash_insert(tabela, pes[i]);
77         }
78     }
79
80     free(pes);
81 }

```

```

83 Pessoa* hash_search(Hash* tabela, int cpf)
84 {
85     int h = hash(tabela, cpf);
86
87     while(tabela->vetor[h] != NULL){
88         if(tabela->vetor[h]->cpf == cpf){
89             return tabela->vetor[h];
90         }
91         h = (h+1) % tabela->tamanho;
92     }
93
94     return NULL;
95 }
96
97 Pessoa* hash_insert(Hash** tabela, Pessoa* indiv)
98 {
99     int h = hash(*tabela, indiv->cpf);
100     Hash* tab = *tabela;
101
102     if(tab->elementos > (0.75*tab->tamanho)){
103         redimenciona(tabela);
104     }
105
106     while(tab->vetor[h] != NULL){
107         h = (h+1) % tab->tamanho;
108     }
109
110     tab->vetor[h] = indiv;
111     tab->elementos++;
112
113     return indiv;
114 }

```

```

116 void print_hash(Hash *tabela)
117 {
118     int i;
119     for (i = 0; i < tabela->tamanho; i++){
120
121         if (tabela->vetor[i] != NULL){
122
123             printf("Nome: %s, CPF: %d\n", tabela->vetor[i]->nome, tabela->vetor[i]->cpf);
124         }
125     }
126 }

```

Hash_DispersaoDupla.h

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4  #include<conio.h>
5
6  typedef struct pes Pessoa;
7  typedef struct hash Hash;
8  int hash_2(Hash* tabela, int cpf)
9  int hash(Hash* tabela, int cpf);
10 Hash* hash_create();
11 Pessoa* pes_create(char* nome, int cpf);
12 void hash_free(Hash** tabela);
13 void redimenciona (Hash** tabela);
14 Pessoa* hash_search(Hash* tabela, int cpf);
15 Pessoa* hash_insert(Hash** tabela, Pessoa* indiv);
16 void print_hash(Hash *tabela);
```

Hash_DispersaoDupla.c

```
1  #include "Hash_DispersaoDupla.h"
2
3  struct pes{
4      char nome[100];
5      int cpf;          // 5 ultimos números do cpf da pessoa
6  };
7
8  struct hash{          //struct da tabela hash
9      int tamanho;
10     int elementos;
11     Pessoa** vetor;
12 };
13
14 int hash_2(Hash* tabela, int cpf){
15     return tabela->tamanho - 2 - cpf%(tabela->tamanho - 2);
16 }
17
18 int hash(Hash* tabela, int cpf)
19 {
20     return cpf%tabela->tamanho;
21 }
```

```

23 Hash* hash_create()
24 {
25     int i;
26     Hash* h = (Hash*) malloc(sizeof (Hash)); //Aloca memória tabela
27     h->vetor = (Pessoa**) malloc(13*sizeof(Pessoa*)); //Aloca memória vetor
28
29     for(i=0; i<13 ; i++){
30         h->vetor[i] = NULL;
31     }
32
33     h->tamanho = 13; //tamanho total
34     h->elementos = 0;
35
36     return h;
37 }
38
39 Pessoa* pes_create(char* nome, int cpf)
40 {
41     Pessoa* pes = (Pessoa*) malloc(sizeof(Pessoa));
42
43     strcpy(pes->nome, nome);
44     pes->cpf = cpf;
45
46     return pes;
47 }
48
49 void hash_free(Hash** tabela)
50 {
51     int i;
52     Hash* tab = *tabela;
53
54     for(i=0; i< tab->tamanho; i++){
55         if(tab->vetor[i])
56             free(tab->vetor[i]);
57     }
58
59     free(tab->vetor);
60     free(*tabela);
61 }

```

```

63 void redimenciona (Hash** tabela)
64 {
65
66     Hash* hsh = *tabela;
67     int i;
68     int anterior = hsh->tamanho;
69     Pessoa** pes = hsh->vetor;
70
71     hsh->elementos = 0;
72     hsh->tamanho *= 1.947;
73     hsh->vetor = (Pessoa**) malloc(hsh->tamanho*sizeof(Pessoa*));
74
75     for(i=0; i< hsh->tamanho; ++i) {
76         hsh->vetor[i] = NULL;
77     }
78     for(i=0; i<anterior; ++i){
79         if(pes[i]){
80             hash_insert(tabela, pes[i]);
81         }
82     }
83
84     free(pes);
85 }

```

```

87 Pessoa* hash_search(Hash* tabela, int cpf)
88 {
89     int h = hash(tabela, cpf);
90     int h2 = hash2(tabela, cpf);
91
92     while(tabela->vetor[h] != NULL){
93         if(tabela->vetor[h]->cpf == cpf){
94             return tabela->vetor[h];
95         }
96         h = (h+h2) % tabela->tamanho;
97     }
98
99     return NULL;
100 }
101
102 Pessoa* hash_insert(Hash** tabela, Pessoa* indiv)
103 {
104     int h = hash(*tabela, indiv->cpf);
105     int h2 = hash2(*tabela, indiv->cpf);
106     Hash* tab = *tabela;
107
108     if(tab->elementos>(0.75*tab->tamanho)){
109         redimenciona(tabela);
110     }
111
112     while(tab->vetor[h] != NULL){
113         h = (h+h2) % tab->tamanho;
114     }
115
116     tab->vetor[h] = indiv;
117     tab->elementos++;
118
119     return indiv;
120 }

```

```
122 void print_hash(Hash *tabela)
123 {
124     int i;
125     for (i = 0; i < tabela->tamanho; i++){
126
127         if (tabela->vetor[i] != NULL){
128
129             printf("Nome: %s, CPF: %d\n", tabela->vetor[i]->nome, tabela->vetor[i]->cpf);
130         }
131     }
132 }
```


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.

$O(n^2)$ - Ocorre quando possui dois laços de repetição, sendo um dentro do outro (ex: for dentro de outro for), necessitando de constante * n^2 instruções.

```
6  "PilhaVet.h"
7
8  Stack* create_stack(); -> O(1)
9
10 int stack_size(Stack* stk); -> O(1)
11
12 bool stack_empty(Stack* stk); -> O(1)
13
14 int stack_push(Stack** stk, int info); -> O(1)
15
16 int stack_pop(Stack** stk, int *info); -> O(1)
17
18 void clear_stack(Stack** stk); -> O(1)
19
20 void print_stack(Stack* stk); -> Melhor caso: O(1); Médio e pior caso: O(n)
```

```
23 "PilhaList.h"
24
25 L_no * create_no(int info); -> O(1)
26
27 int l_size(L_no* lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
28
29 int empty(L_no* lhead); -> O(1)
30
31 void clear_list(L_no** lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
32
33 int stack_push(L_no** lhead, int info); -> Melhor caso: O(1); Médio e pior caso: O(n)
34
35 int stack_pop(L_no** lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
36
37 void print_stack(L_no* lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
```

```
40 "FilaVet.h"
41
42 Queue* create_queue(); -> O(1)
43
44 int queue_size(Queue* q); -> O(1)
45
46 bool queue_empty(Queue* q); -> O(1)
47
48 int queue_insert(Queue** q, int info); -> O(1)
49
50 int queue_remove(Queue** q, int *info); -> O(1)
51
52 void clear_queue(Queue** q); -> O(1)
53
54 void print_queue(Queue* q); -> Melhor caso: O(1); Médio e pior caso: O(n)
```

```
56 "FilaList.h"
57
58 L_no * create_no(int info); -> O(1)
59
60 int l_size(L_no* lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
61
62 int empty(L_no* lhead); -> O(1)
63
64 void clear_list(L_no** lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
65
66 int queue_insert(L_no** lhead, int info); -> Melhor caso: O(1); Médio e pior caso: O(n)
67
68 int queue_remove(L_no** lhead); -> O(1)
69
70 void print_queue(L_no* lhead); -> Melhor caso: O(1); Médio e pior caso: O(n)
```