Questão 1
Letra A

1.1. PILHAS

Sendo a estrutura de dados mais utilizada em programação, sua principal característica é o fato de que os seus elementos são manuseados a partir do topo, seguindo o conceito de *Last In First Out*,logo os elementos são adicionados (*push*) e removidos (*pop*) do topo.

As pilhas normalmente são implementadas usando um vetor ou uma lista encadeada. A função de criação de uma pilha deve alocar dinamicamente a estrutura desta, inicializar seus campos e retornar um ponteiro que aponta para o topo da pilha.

1.1.1. IMPLEMENTAÇÃO COM VETOR

Para tal, deve-se ter um vetor para armazenar os elementos da pilha, de tal forma que os elementos inseridos ocupam as primeiras posições vetor e o topo da pilha é o número elementos - 1. Vale ressaltar, que esse tipo de implementação pode apresentar um valor máximo de elementos (vetor estático) ou indefinido (vetor dinâmico), obviamente que dependendo da disponibilidade de memória.

1.1.2. IMPLEMENTAÇÃO COM LISTA

Para tal, os elementos inseridos na pilha serão armazenados na lista, o que acaba se tornando simples se considerar o topo como primeiro elemento da lista, já que facilita os processos de *pop* e *push*. A implementação com lista se torna simples, pois a pilha seria basicamente a lista.

1.1.3. APLICAÇÕES

Um exemplo de aplicação são as calculadoras RNP: Elas operam com expressões pós-fixadas, o que faz com que cada operando seja empilhando numa pilha de valores e quando se encontra um operador, desempilha-se o número

apropriado de operandos(2 ou 1) e realiza a operação, em seguida empilhando seu resultado. Em operações simultâneas são feitas sequencialmente gerando resultados parciais no processo

1.2. FILAS

A base da ideia é o *First In First Out* e é análoga a um sistema de filar que conhecemos, o primeiro a entrar e a ser atendido.

1.2.1 IMPLEMENTAÇÃO COM VETOR

Passível, assim como as pilhas, de ser implementado com vetor dinâmico ou estático. Uma estratégia que facilita a implementação seria a inserção de novos elementos no final do vetor e a retirada no início, sempre deslocando os elementos conforme a operação e respeitando a ideia do *First In First Out*, o único fator que dificulta esse processo é o alto custo computacional por causa do deslocamento.

Para resolver isso, o ideal é fazer com que a variável que define o início ande ao longo do vetor, o que resolveria o problema de alto custo de processamento. Após o preenchimento finito dos dados, deve-se adotar uma estratégia circular fazendo com que o fim da lista seja considerado o primeiro elemento do vetor, caso lá hajam posições livres.

Para implementar uma fila utilizando vetor dinâmico, sua estrutura necessita, de um vetor dinâmico, a dimensão do vetor, o número de elementos armazenados na fila e o índice de inicio da fila

1.2.2. IMPLEMENTAÇÃO COM LISTA

Para implementar uma fila por meio de uma lista encadeada, como usualmente é feito, cada nó guarda um ponteiro para o próximo nó da lista. Como será necessário seguir a mesma lógica de vetor, com relação à estratégia contra o necessidade de alto custo em processamento, haverá também dois ponteiros, um apontando para o início e outro para o fim da fila.

1.3. FILAS DUPLAS

Trata-se de uma fila na qual é passível de inserir novos elementos em ambas as extremidades da mesma.

1.3.1. IMPLEMENTAÇÃO COM VETOR

A implementação usando vetor, para inserir no local que está sendo apontado o início deve-se adotar um decremento circular e acesso randômico aos elementos.

1.3.2. IMPLEMENTAÇÃO COM LISTA ENCADEADA

Com um empecilho a retirada de elemento do final da lista, diante do fato que o ponteiro para o último elemento da lista, não permite o acesso ao elemento anterior, que passaria a ser o último elemento. Para contornar isso se faz necessário o uso da estrutura de lista duplamente encadeada.

Letra B

pilha.h

```
#ifndef PILHA_H
#define PILHA_H

typedef struct pilha Pilha ;

Pilha* pilha_cria (void );
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_imprime(Pilha* p);
void pilha_libera (Pilha* p);

#endif
```

pilha.c

```
#include <stdlib.h>
#include <stdio.h>
#include "pilha.h"

struct pilha{
```

```
   int n; //numero de elementos armazenados
   int dim; //dimensão do vetor
   float* vet; //vetor de elementos
};

Pilha* pilha_cria (void){
   Pilha* p = (Pilha*) malloc(sizeof (Pilha));
   p->dim = 2;
   p->vet = (float *) malloc(p-> dim*sizeof (float ));
   p->n = 0;
   return p;
}

void pilha_push (Pilha* p, float v){

   if (p->n == p->dim) { // aumentando a capacidade da pilha, caso
esteja cheia
       p-> dim *= 2;
       p-> vet = (float *) realloc(p->vet, p-> dim*sizeof (float ));
   }
   p->vet[p->n++] = v; // insere elemento na próxima posição livre no
topo
}

float pilha_pop (Pilha* p){
   float v = p->vet[--p->n]; // retirando elemento do topo
   return v;
}

int pilha_vazia (Pilha* p){
   return (p->n == 0);
}

void pilha_libera (Pilha* p){
   free(p->vet);
   free(p);
}
void pilha_imprime (Pilha* p){
   for (int i=p->n-1; i>=0; i--)
       printf("% f\n",p-> vet[i]);
}
```

main.c

```c
#include <stdio.h>
#include "pilha.h"


int main(){
   Pilha* p = pilha_cria ();
   printf("\n%s", pilha_vazia (p) == 0 ? "Pilha Vazia\n":"Pilha Não
Vazia\n");
   pilha_push (p, 1.5);
   pilha_push (p, 1.3);
   pilha_push (p, 4.5);
   pilha_imprime(p);
   printf("Removido: %f\n",pilha_pop (p));
   printf("%s", pilha_vazia(p) == 0 ? "Pilha Vazia\n":"Pilha Não
Vazia\n");
   pilha_imprime(p);
   pilha_libera(p);
}
```

Letra C

pilha.h

```c
#ifndef PILHA_H
#define PILHA_H

typedef struct  pilha Pilha;
typedef struct listano ListaNo;

Pilha* pilha_cria ();
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);
void pilha_imprime (Pilha* p);

#endif
```

pilha.c

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

```c
#include <stdio.h>
#include "pilha.h"

struct pilha{
   ListaNo* topo;
};
struct listano {
   float info;
   ListaNo* prox;
};

Pilha* pilha_cria(){
   Pilha* p = (Pilha*) malloc(sizeof(Pilha));
   p->topo = NULL;
   return p;
}

void pilha_push (Pilha* p, float v){
   ListaNo* n = (ListaNo*) malloc(sizeof(ListaNo));
   n->info = v;
   n->prox = p->topo;
   p->topo = n;
}

float pilha_pop (Pilha* p){
   ListaNo* t = p->topo;
   float v = t->info;
   p->topo = t->prox;
   free(t);
   return v;
}

int pilha_vazia (Pilha* p){
   return (p->topo == NULL );
}

void pilha_libera (Pilha* p){
   ListaNo* q = p->topo;
   while (q!= NULL) {
       ListaNo* t = q->prox;
       free(q);
       q = t;
   }
```

```
   free(p);
}
void pilha_imprime (Pilha* p){
   for ( ListaNo* q=p->topo; q!= NULL; q=q->prox)
       printf("% f\n",q->info);
}
```

main.c

```
#include <stdio.h>
#include "pilha.h"


int main(){
   Pilha* p = pilha_cria ();
   printf("\n%s", pilha_vazia (p) == 0 ? "Pilha Vazia\n":"Pilha Não
Vazia\n");
   pilha_push (p, 1.5);
   pilha_push (p, 1.3);
   pilha_push (p, 4.5);
   pilha_imprime(p);
   printf("Removido: %f\n",pilha_pop (p));
   printf("%s", pilha_vazia(p) == 0 ? "Pilha Vazia\n":"Pilha Não
Vazia\n");
   pilha_imprime(p);
   pilha_libera(p);
}
```

Letra D

fila.h

```
#ifndef FILA_H
#define FILA_H

typedef struct fila Fila;

Fila* fila_cria (void );
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);
```

```
void fila_imprime (Fila* f);

#endif
```

fila.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "fila.h"

struct fila {
   int n; //número de elementos
   int ini; //índice da fila
   int dim; //dimensão do vetor
   float* vet; //vetor dos elementos
};

Fila* fila_cria (void ){
   Fila* f = (Fila*) malloc(sizeof(Fila));
   f->dim = 6;
   f->vet = (float*) malloc(f-> dim*sizeof (float ));
   f->n = 0;
   f->ini = 0;
   return f;
}
void fila_insere (Fila* f, float v){

   int fim;
   if (f->n == f-> dim) { //Se já estiver totalmente preenchido
       f->dim *= 2; //Dobra o tamanho
       f->vet = (float *) realloc(f->vet, f-> dim*sizeof (float ));
//aumenta o tamanho do vetor
       if (f->ini != 0){
           memmove(&f-> vet[f->dim - f->ini],//endereço de destino
               &f-> vet[f->ini], //endereço de origem
               (f->n - f->ini) * sizeof (float) //número de bytes
           );
       }
   }
   fim = (f->ini + f->n) % f->dim; //procurando um elemento com posição
livre
```

```
    f->vet[fim] = v; //inserindo elemento
    f->n++; //aumentando o número de elementos em um
}
float fila_retira (Fila* f){
    float v;
    v = f->vet[f->ini]; //Acessa o primeiro elemento da fila
    f->ini = (f->ini + 1) % f->dim; // Coloca o ínicio para o próximo
elemento
    f->n--; //Decrementa o número que representa a quantidade de
elementos do vetor
    return v; //Retorna o elemento excluído
}

int fila_vazia (Fila* f){
    return (f->n == 0); // Verifica se a lista esa vazia
}
void fila_libera (Fila* f){
    free(f->vet); //libera a memória alocada do vetor
    free(f); //Libera a memória alocada da fila
}

void fila_imprime (Fila* f){
    for(int i = 0;i < f->n;i++)
        printf("Elemento %i : %f\n",i,f->vet[f->ini+i]);
}
```

main.c

```
# include <stdio.h>
# include "fila.h"

int main (){
    Fila* f = fila_cria();
    fila_insere(f,20.0);
    fila_insere(f,20.8);
    fila_insere(f,21.2);
    fila_insere(f,24.3);
    fila_imprime(f);
    printf(" Primeiro elemento: % f\n", fila_retira(f));
    printf(" Segundo elemento: % f\n", fila_retira(f));
    printf(" Configuracao da fila:\n");
```

```
    fila_imprime(f);
    fila_libera(f);
    return 0;
}
```

Letra E

fila.h

```
#ifndef FILA_H
#define FILA_H

typedef struct fila Fila ;
typedef struct elemento Elemento;

Fila* fila_cria();
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);
void fila_imprime (Fila* f);

#endif
```

fila.c

```
#include <stdlib.h>
#include <stdio.h>
#include "fila.h"

struct elemento{
    float info;
    Elemento* prox;
};
struct fila{
    Elemento* ini;
    Elemento* fim;
};

Fila* fila_cria(){
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = NULL;
```

```
   return f;
}

void fila_insere(Fila* f,float v){
   Elemento* a = (Elemento*) malloc (sizeof(Elemento)); //Criando
elemento
   a->info = v; //Armazenando o elemento
   a->prox = NULL; //Próximo igual a NULL
   if(f->fim !=NULL)
       f->fim->prox = a;
   else
       f->ini = a;

   f->fim = a;
}

float fila_retira (Fila* f){
   Elemento* t = f->ini;
   float v = t->info;
   f->ini = t->prox;
   if(f->ini == NULL)
       f->fim = NULL;
   free(t);
   return v;
}

int fila_vazia(Fila* f){
   return (f->ini == NULL);
}

void fila_libera (Fila* f){
   Elemento* q = f->ini;
   while(q!=NULL){
       Elemento* t = q->prox;
       free(q);
       q = t;
   }
   free(f);
}

void fila_imprime (Fila* f){
  for ( Elemento* q=f->ini; q!= NULL; q=q->prox)
       printf("% f\n",q-> info);
```

```
}
```

main.c

```c
# include <stdio.h>
# include "fila.h"

int main (){
   Fila* f = fila_cria();
   fila_insere(f,20.0);
   fila_insere(f,20.8);
   fila_insere(f,21.2);
   fila_insere(f,24.3);
   fila_imprime(f);
   printf(" Primeiro elemento: % f\n", fila_retira(f));
   printf(" Segundo elemento: % f\n", fila_retira(f));
   printf(" Configuracao da fila:\n");
   fila_imprime(f);
   fila_libera(f);
   return 0;
}
```

Questão 2

Letra A

2. CAPÍTULO 19: TABELAS DE DISPERSÃO

As tabelas de dispersão surge como uma alternativa para minimizar o esforço computacional, porém utilizando mais memória, sendo esta proporcional ao número de elementos armazenados

2.1. IDEIA CENTRAL

A ideia central é a identificação das partes significativas da chave de busca, isso conforme a dimensão do conjunto de objetos com o qual eu estou identificando, para acelerar as operações.

Dependendo do subgrupo em que se está realizando uma operação, esta pode ser feita de forma mais rápida, apenas identificando as partes significativas, as diferenças entre cada objeto daquele subgrupo. Isto traz como benefício uma diminuição do uso excedente de memória, porém pode trazer também um problema que seria a colisão, dígitos significativos de diferentes elementos iguais, existem métodos para tratamento desse fato, porém nada totalmente eficaz.

2.2. FUNÇÃO DE DISPERSÃO

Esta mapeia uma chave de busca em um índice da tabela, para tal ela deve, sempre que possível, apresentar as seguintes propriedades:

- Ser eficientemente avaliada, sendo isto necessário para a realização do acesso rápido, que é justamente o uso da tabela de dispersão;
- Espalhar bem as chaves de busca, o que minimiza a ocorrência de colisões, o que deve ser feito combinado com uma folga de 25% da tabela. O ideal é uma taxa de ocupação de 25% a 75%, sendo 50% o valor ótimo.

2.3.TRATAMENTO DE COLISÃO

Há várias estratégias, mas em todas a tabela de dispersão, esta é representada por um vetor de ponteiros, o qual representa a informação.

2.3.1. USO DA POSIÇÃO CONSECUTIVA LIVRE

Nesta e na próxima estratégia, será discutido o problema de que os elementos que colidem são armazenados em outros índices, ainda não ocupados da própria tabela.

Nesta, se a função de dispersão mapeia a chave de busca para um índice já ocupado, procura-se o próximo, usando o incremento circular, índice livre para o armazenamento do novo elemento.

Deve-se frisar que a existência de algum elemento mapeado para o mesmo índice, não garante que o elemento que será buscado esteja presente. A partir do índice mapeado, deve-se buscar o elemento utilizando como chave de comparação a real chave de busca.

Vale ressaltar que essa estratégia visa concentrar os lugares ocupados na tabela, enquanto o ideal, seria ocupar dispersar.

2.3.2. USO DE UMA SEGUNDA FUNÇÃO DE DISPERSÃO

A fim de evitar a concentração de posições ocupadas na tabela, esta estratégia realiza uma variação na forma de procurar uma posição livre, a fim de armazenar o elemento que colidiu. Para uma com números inteiros, a função de busca seria: h(x)=(N-2)-x%(N-2), sendo x a chave e N a dimensão da tabela. Caso haja colisão, deve-se procurar uma posição livre na tabela com incrementos, ainda circulares.

Duas observações devem ser feitas a partir dessa segunda função de dispersão: ela nunca poderá retornar zero, pois isto representaria que o índice não foi incrementado, e que ela preferencialmente não deve retornar um número divisor da dimensão da tabela, o que limitaria a procura de uma posição livre em um subconjunto restrito que no caso seria os índices da tabela.

2.3.3. USO DE LISTAS ENCADEADAS

Outra forma de tratar uma colisão é por meio de um encadeamento externo, que se dá por meio da representação de cada elemento da tabela de dispersão por um ponteiro para a montagem de uma lista encadeada.

2.4. TABELA DE DISPERSÃO DINÂMICA

Um grande problema, que pode ocasionar no gasto de memória de forma desnecessária. é a necessidade de definir a dimensão da tabela. A maneira mais simples de definir uma tabela de dispersão dinâmica é a partir do redimensionamento da tabela ao longo do seu uso, a partir da definição de limites de aplicações.

2.5. APLICAÇÕES

Existem diferentes aplicações para o uso de tabelas de dispersão entre elas há por exemplo, a exibição da frequência de palavras em um texto.

Letra B

hashA.h

```
#ifndef HASH
#define HASH

typedef struct pessoa Pessoa;
typedef struct hash Hash;

static int hash (long int cpf, Hash* tabela);
static void redimensiona (Hash* tabela);
Hash* create_hash();
void print_Pessoa(Pessoa* a);
int remove_hash(Hash* tabela, long int cpf);
Pessoa* insert_hash(Hash* tabela, Pessoa* pessoa);
Pessoa* search_hash(Hash* tabela, long int cpf);
Pessoa* init_Pessoa(long int cpf,char* nome,char sexo,int idade);
#endif
```

hashA.c

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

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


struct pessoa
{
   long int cpf;
   char nome[40];
   char sexo;//M = Masculino/F = Feminino
   int idade;
};

struct hash {
   int n; // numero de elementos inseridos
   int dim; // dimensao da tabela
   Pessoa** p;
};

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

static void redimensiona (Hash* tabela){
   int dim = tabela->dim;
   Pessoa** ant = tabela->p;
   tabela->n = 0;
   tabela->dim *= 1.765;
   tabela->p  = (Pessoa**) malloc(tabela->dim*sizeof(Pessoa));

   for (int i=0; i<tabela->dim; ++i)
       tabela->p[i] = NULL;

   for (int i=0;i<tabela->dim;i++){
       if(ant[i]){
           insert_hash(tabela,ant[i]);
       }
   }
   free(ant);

}

int remove_hash(Hash* tabela, long int cpf){
```

```
    int x = hash(cpf, tabela);

    while (tabela->p[x] != NULL) {

        if (tabela->p[x]->cpf == cpf)
        {
            free(tabela->p[x]);
            return 1;
        }

        x = (++x) % tabela->dim;
    }

    return 0;
}

Hash* create_hash(){
    Hash* tabela = (Hash*) malloc(sizeof(Hash));
    if(tabela){
        tabela->n = 0;
        tabela->dim = 10;
        tabela->p = (Pessoa**) malloc(tabela->dim*sizeof(Pessoa));

        for (int i=0;i<tabela->dim;i++)
            tabela->p[i] = NULL;

    }
    return tabela;
}
Pessoa* insert_hash(Hash* tabela, Pessoa* pessoa){
    if(tabela->n > 0.75*tabela->dim)
        redimensiona(tabela);

    int x = hash(pessoa->cpf,tabela);

    while(tabela->p[x]!=NULL)
        x = (++x)%tabela->dim;

    tabela->p[x] = pessoa;
    tabela->n++;
    return pessoa;
}
```

```
Pessoa* search_hash(Hash* tabela, int long cpf){
   int h1 = hash(cpf,tabela);
   int h2 = hash(cpf,tabela);

   while(tabela->p[h1]){
       if(tabela->p[h1]->cpf == cpf)
           return tabela->p[h1];

       h1 = (h1+h2)%tabela->dim; //incremento circular
   }
   return NULL;
}
Pessoa* init_Pessoa(long int cpf,char* nome,char sexo,int idade){
   Pessoa* p = (Pessoa*) malloc(sizeof(Pessoa));
   if(p){
       p->cpf = cpf;
       strcpy(p->nome, nome);
       p->sexo = sexo;
       p->idade = idade;

   }
   return p;
}

void print_Pessoa(Pessoa* a){
   printf("\n%s\n", a->nome);
   printf("%c\n", a->sexo);
   printf("%ld\n", a->cpf);
   printf("%d\n", a->idade);
}
```

main.c

```
#include <stdio.h>
#include "hashA.h"

int main(){

   Hash* tab = create_hash();
```

```
    Pessoa *p1 = init_Pessoa(12919902407, "Yasmin", 'F',19);
    Pessoa *p2 = init_Pessoa(12345678910, "Lucas",'M',14);
    Pessoa *p3 = init_Pessoa(12457689098, "Adriano",'M',14);
    Pessoa *p4 = init_Pessoa(56239813801, "Lucia", 'F',10);

    //INSERINDO
    insert_hash(tab, p1);
    insert_hash(tab, p2);
    insert_hash(tab, p3);
    insert_hash(tab, p4);

    //PESQUISANDO
    Pessoa* a;
    a = search_hash(tab, 12919902407);
    if (a != NULL)
        print_Pessoa(a);
    else
        printf("CPF não encontrado!\n");

    a = search_hash(tab, 98712312341);

    if (a != NULL)
        print_Pessoa(a);
    else
        printf("CPF não encontrado!\n");


    //REMOVENDO

    printf("%s", remove_hash(tab, 12919902407) ? "Removido\n" : "CPF não
encontrado!\n");

    a = search_hash(tab, 12883424462);
    if (a != NULL)
        print_Pessoa(a);
    else
        printf("CPF não encontrado!\n");


    return 0;
}
```

Letra C

hashB.h

```
#ifndef HASH
#define HASH

typedef struct pessoa Pessoa;
typedef struct hash Hash;

static int hash (long int cpf, Hash* tabela);
static void redimensiona (Hash* tabela);
Hash* create_hash();
void print_Pessoa(Pessoa* a);
int remove_hash(Hash* tabela, long int cpf);
Pessoa* insert_hash(Hash* tabela, Pessoa* pessoa);
Pessoa* search_hash(Hash* tabela, long int cpf);
Pessoa* init_Pessoa(long int cpf,char* nome,char sexo,int idade);
#endif
```

hashB.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hashB.h"


struct pessoa
{
   long int cpf;
   char nome[40];
   char sexo;//M = Masculino/F = Feminino
   int idade;
};

struct hash {
   int n; // numero de elementos inseridos
   int dim; // dimensao da tabela
   Pessoa** p;
};

static int hash (long int cpf, Hash* tabela){
   return ((tabela->dim - 2) - cpf % (tabela->dim - 2)); // criação do
hash
}
```

```c
static void redimensiona (Hash* tabela){
   int dim = tabela->dim;
   Pessoa** ant = tabela->p;
   tabela->n = 0;
   tabela->dim *= 1.765;
   tabela->p  = (Pessoa**) malloc(tabela->dim*sizeof(Pessoa));

   for (int i=0; i<tabela->dim; ++i)
       tabela->p[i] = NULL;

   for (int i=0;i<tabela->dim;i++){
       if(ant[i]){
           insert_hash(tabela,ant[i]);
       }
   }
   free(ant);

}

int remove_hash(Hash* tabela, long int cpf){

   int x = hash(cpf, tabela);

   while (tabela->p[x] != NULL) {

       if (tabela->p[x]->cpf == cpf)
       {
           free(tabela->p[x]);
           return 1;
       }

       x = (++x) % tabela->dim;
   }

   return 0;
}

Hash* create_hash(){
   Hash* tabela = (Hash*) malloc(sizeof(Hash));
   if(tabela){
       tabela->n = 0;
       tabela->dim = 10;
```

```
        tabela->p = (Pessoa**) malloc(tabela->dim*sizeof(Pessoa));

        for (int i=0;i<tabela->dim;i++)
            tabela->p[i] = NULL;

    }
    return tabela;
}
Pessoa* insert_hash(Hash* tabela, Pessoa* pessoa){
    if(tabela->n > 0.75*tabela->dim)
        redimensiona(tabela);

    int x = hash(pessoa->cpf,tabela);

    while(tabela->p[x]!=NULL)
        x = (++x)%tabela->dim;

    tabela->p[x] = pessoa;
    tabela->n++;
    return pessoa;
}
Pessoa* search_hash(Hash* tabela, int long cpf){
    int h1 = hash(cpf,tabela);
    int h2 = hash(cpf,tabela);

    while(tabela->p[h1]){
        if(tabela->p[h1]->cpf == cpf)
            return tabela->p[h1];

        h1 = (h1+h2)%tabela->dim; //incremento circular
    }
    return NULL;
}
Pessoa* init_Pessoa(long int cpf,char* nome,char sexo,int idade){
    Pessoa* p = (Pessoa*) malloc(sizeof(Pessoa));
    if(p){
        p->cpf = cpf;
        strcpy(p->nome, nome);
        p->sexo = sexo;
        p->idade = idade;

    }
    return p;
```

```
}

void print_Pessoa(Pessoa* a){
   printf("\n%s\n", a->nome);
   printf("%c\n", a->sexo);
   printf("%ld\n", a->cpf);
   printf("%d\n", a->idade);
}
```

main.c

```
#include <stdio.h>
#include "hashB.h"

int main(){

   Hash* tab = create_hash();

   Pessoa *p1 = init_Pessoa(12919902407, "Yasmin", 'F',19);
   Pessoa *p2 = init_Pessoa(12345678910, "Lucas",'M',14);
   Pessoa *p3 = init_Pessoa(12457689098, "Adriano",'M',14);
   Pessoa *p4 = init_Pessoa(56239813801, "Lucia", 'F',10);

   //INSERINDO
   insert_hash(tab, p1);
   insert_hash(tab, p2);
   insert_hash(tab, p3);
   insert_hash(tab, p4);

   //PESQUISANDO
   Pessoa* a;
   a = search_hash(tab, 12919902407);
   if (a != NULL)
       print_Pessoa(a);
   else
       printf("CPF não encontrado!\n");

   a = search_hash(tab, 98712312341);

   if (a != NULL)
       print_Pessoa(a);
```

```
    else
        printf("CPF não encontrado!\n");


    //REMOVENDO

    printf("%s", remove_hash(tab, 12919902407) ? "Removido\n" : "CPF não
encontrado!\n");

    a = search_hash(tab, 12883424462);
    if (a != NULL)
        print_Pessoa(a);
    else
        printf("CPF não encontrado!\n");


    return 0;
}
```

Questão 3

Letra A - Pergunta 1

3. TÓPICO 2.3 : ORDENS ASSINTÓTICAS

Sendo esta definida como o crescimento da complexidade para entradas suficientemente grandes. Um algoritmo para ser considerado assintoticamente mais eficiente este deve ser estruturado para todas as entradas, exceto para aquelas relativamente pequenas.

O crescimento assintótico se refere a constante que representa a variação da função ao longo de seu gráfico.Usualmente o cálculo da complexidade se fundamente em determinar a ordem de magnitude do número de operações fundamentais para a execução do algoritmo, para tal há por exemplo, a cota assintótica superior, que trata-se de uma função que cresce mais rapidamente que outra, permanecendo acima a partir de certo ponto. Existem várias comparações de complexidade assintótica, as mais usadas é a O, a Ω, e a Θ.

Segundo a notação O, se uma função cresce mais do que a outra, podendo ser a partir de um certo ponto, dizemos que a linear da que cresce menos é o O (função que cresce mais). De forma geral, analisa-se pensando que existe uma constante real e positiva que multiplicada pela função que menos cresce, resulta na que mais cresce. Dizer que um algoritmo é O(1) significa que o número de operações fundamentais executadas é limitado por uma constante.

Já a notação Ω define uma cota assintótica inferior a menos de constantes, nesse caso se uma função que cresce mais que possui como Ω (a função que menos cresce). Isso significa que existe uma constante real positiva que a partir de certo ponto na função, o valor da que menos cresce é sempre menor ou igual ao produto dessa constante real com a função que mais cresce.

Já a notação Θ define um limite assintótico exato, a menos de constantes. Neste caso, é se as funções crescerem com a mesma rapidez, a partir de um certo ponto, logo um é Θ da outra. Isso significa que uma função é menor ou igual a uma constante real positiva multiplicado por outra função, sendo a recíproca também verdadeira. Vale ressaltar que a ordem Θ é um caso particular das ordens Ω e O, já que se caso uma função é
a Θ de outra, logo ela também será a Ω e a O.

Letra A - Pergunta 2

Diante do fato de que $n \leq O(n^2)$ ou $n \in O(n^2)$ e que n é menor ou igual à taxa de n^2, logo pode-se afirmar que $f(n) = n$ é $O(n^2)$.

Letra B

```
Pilha* pilha_cria (void );  A partir de uma pilha autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void pilha_push (Pilha* p, float v);-> A partir de uma pilha
autoajustável, qualquer sequência de P operações têm um custo
proporcional a P,o que consequentemente indica que cada operação tem um
custo constante de O(1).
float pilha_pop (Pilha* p);-> A partir de uma pilha autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
int pilha_vazia (Pilha* p); -> A partir de uma pilha autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void pilha_imprime(Pilha* p);-> A partir de uma pilha autoajustável,
para realizar essa operação dado um valor n de elementos dentro da
pilha entre realizará f(n) operações a partir da função f(n)=3n+1, logo
como o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo
a complexidade O(n).
void pilha_libera (Pilha* p); -> A partir de uma pilha autoajustável,
para realizar essa operação dado um valor n de elementos dentro da
pilha entre realizará f(n) operações a partir da função f(n)=4n+2, logo
como o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo
a complexidade O(n).
```

Letra C

```
Pilha* pilha_cria ();-> A partir de uma pilha autoajustável, qualquer
sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void pilha_push (Pilha* p, float v);-> A partir de uma pilha
autoajustável, qualquer sequência de P operações têm um custo
proporcional a P,o que consequentemente indica que cada operação tem um
custo constante de O(1).
float pilha_pop (Pilha* p);-> A partir de uma pilha autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
```

```
consequentemente indica que cada operação tem um custo constante de
O(1).
int pilha_vazia (Pilha* p);-> A partir de uma pilha autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void pilha_libera (Pilha* p);-> A partir de uma pilha autoajustável,
para realizar essa operação dado um valor n de elementos dentro da
pilha entre realizará f(n) operações a partir da função f(n)=4n+2, logo
como o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo
a complexidade O(n).
void pilha_imprime (Pilha* p);-> A partir de uma pilha autoajustável,
para realizar essa operação dado um valor n de elementos dentro da
pilha entre realizará f(n) operações a partir da função f(n)=3n+1, logo
como o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo
a complexidade O(n).
```

Letra D

```
Fila* fila_cria (void );-> A partir de uma fila autoajustável, qualquer
sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void fila_insere (Fila* f, float v);-> A partir de uma fila
autoajustável, qualquer sequência de P operações têm um custo
proporcional a P,o que consequentemente indica que cada operação tem um
custo constante de O(1).
float fila_retira (Fila* f);-> A partir de uma fila autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
int fila_vazia (Fila* f);-> A partir de uma fila autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void fila_libera (Fila* f);-> A partir de uma fila autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void fila_imprime (Fila* f);-> A partir de uma fila autoajustável, para
realizar essa operação dado um valor n de elementos dentro da pilha
entre realizará f(n) operações a partir da função f(n)=3n+1, logo como
```

```
o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo a
complexidade O(n).
```

Letra E

```
Fila* fila_cria();-> A partir de uma fila autoajustável, qualquer
sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void fila_insere (Fila* f, float v);-> A partir de uma fila
autoajustável, qualquer sequência de P operações têm um custo
proporcional a P,o que consequentemente indica que cada operação tem um
custo constante de O(1).
float fila_retira (Fila* f);-> A partir de uma fila autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
int fila_vazia (Fila* f);-> A partir de uma fila autoajustável,
qualquer sequência de P operações têm um custo proporcional a P,o que
consequentemente indica que cada operação tem um custo constante de
O(1).
void fila_libera (Fila* f);-> A partir de uma pilha autoajustável, para
realizar essa operação dado um valor n de elementos dentro da pilha
entre realizará f(n) operações a partir da função f(n)=4n+2, logo como
o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo a
complexidade O(n).

void fila_imprime (Fila* f);-> A partir de uma fila autoajustável, para
realizar essa operação dado um valor n de elementos dentro da pilha
entre realizará f(n) operações a partir da função f(n)=3n+1, logo como
o termo que mais cresce é 3n, troca-se o coeficiente por 1, sendo a
complexidade O(n).
```