Respostas da EAF 2 de Estrutura de Dados

Victoria Monteiro Pontes

Outubro 2020

Questão 1: Sobre pilhas e filas:

(a) Utilizando suas palavras, faça o resumo do ”Capítulo 15 - Pilhas e filas” do livro ”Celes, W. et al.; Introdução a Estrutura de Dados com Técnicas de Programação em C, 2ed., Elsevier, 2016”.

Resposta:

Pilhas

A pilha é uma estrutura de dados bastante simples que segue o princípio LIFO (Last In First Out), que dita que o último elemento inserido, ou seja, o elemento que está no *topo*, na estrutura é o primeiro a ser acessado.

Há dois modos mais utilizados de implementarmos uma pilha: utilizando um vetor ou uma lista encadeada. Quando utilizamos vetores estáticos, normalmente já sabemos qual será a quantidade máxima de elementos a ser inserida na estrura. Entranto, se não soubermos e tivermos que realocar espaço na memória, utilizamos vetores dinâmicos.

A interface de um tipo pilha geralmente possui essas 5 funções:

- Criar pilha vazia
- Checar se a pilha está vazia
- Adicionar elemento ao topo da pilha
- Retirar (ou retornar) um elemento do topo da pilha

- Remover a pilha por completo

Observação: A função para adicionar um elemento ao topo da pilha é, muitas vezes, referida como *"push"*, enquanto a função de remover (ou retornar) um elemento da pilha é referido como *"pop"*.

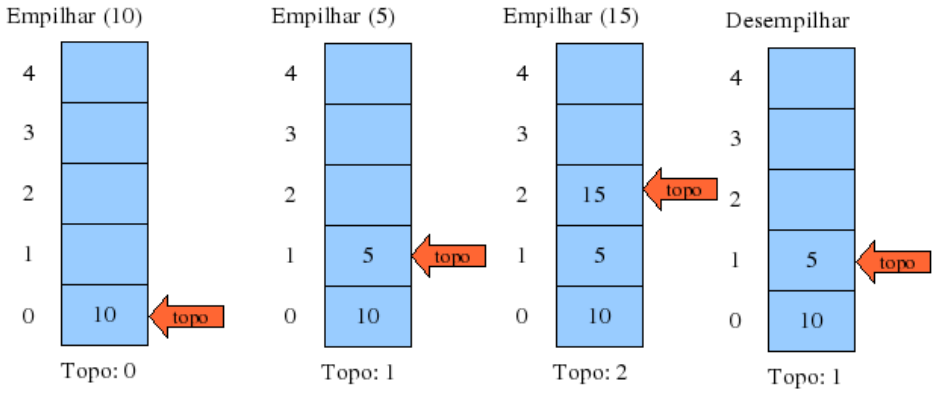


Figure 1: Ilustração do funcionamento de uma pilha

Filas

A fila é uma estrutura de dados que segue o princípio FIFO (First In First Out) que dita que o primeiro elemento inserido é o primeiro elemento a ser acessado ou retirado. Portanto, em uma estrutura de dados do tipo fila, um elemento deve ser inserido no final e outro deve ser acessado ou removido no início da fila.

Podemos implementar a fila de duas formas: utilizando um vetor ou uma lista. Ao utilizarmos um vetor, assim como nas pilhas, podemos optar por um vetor estático ou um vetor dinâmico, dependendo se sabemos ou não a quantidade de elementos máxima a serem inseridos na fila.

Ao implementarmos uma fila com uma lista simplesmente encadeada, além de termos que guardar um ponteiro para o próximo nó da lista, devemos também guardar um ponteiro para o início e final da lista, pois esses ponteiros são necessários para as operações de remoção e adição de elementos na fila.

Depois de decidirmos como iremos implementar a lista, criamos a interface da estrutura que geralmente possui essas funções:

- Criar um fila vazia
- Inserir um elemento no fim
- Retirar o elemento do início
- Verificar se a fila está vazia
- Liberar a fila

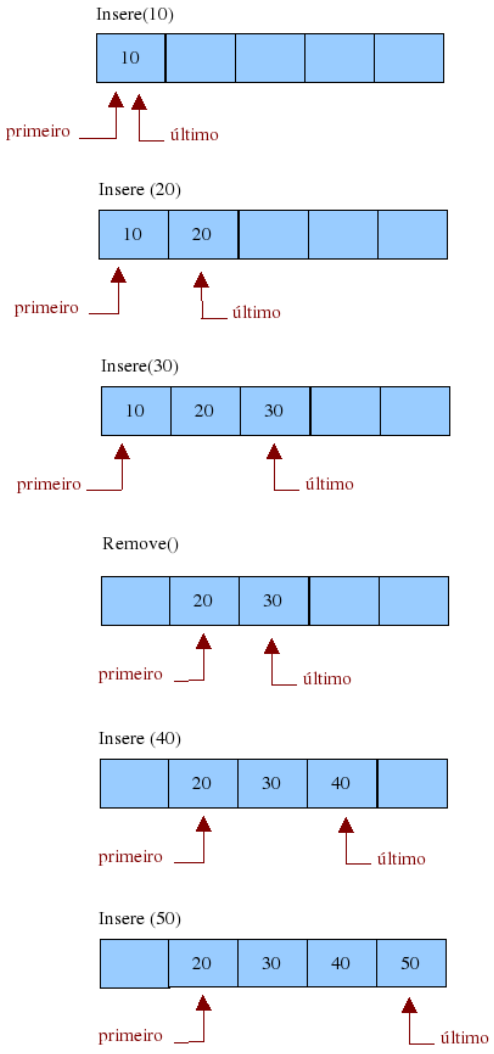


Figure 2: Ilustração do funcionamento de uma fila

Filas duplas:

O conceito de fila dupla refere-se a estrutura de dados com a possibilidade de adicionar ou retirar elementos no início ou no fim de uma fila. Dessa forma, teríamos ”duas” filas em uma só, sendo uma o inverso da outra.

A implementação de uma fila dupla também pode ser feita com um vetor ou com uma lista. Ao utilizarmos um vetor, estático ou dinâmico, temos uma facilidade muito maior de implementação do que com uma lista simplesmente encadeada, pois vetores permitem acessos randômicos ao seus elementos. Isso não ocorre com uma implementação em lista, e portanto, a função de retirar no fim da lista não é executada de forma eficiente, pois, no último nó, possuímos apenas o ponteiro para um próximo, não um ponteiro para o anterior que passaria a ser o último. Dessa forma, uma implementação de uma fila dupla com listas sempre terá a característica de ser duplamente encadeada.

(b) Implemente uma versão do TAD Pilha utilizando vetor dinâmico.
Resposta:

ItemB: Interface das funções no arquivo pilha_dinamica.h

```
#ifndef PILHA_DINAMICA
#define PILHA_DINAMICA


typedef struct pilha Pilha;


/*Cria pilha vazia*/
Pilha * cria_pilha();


/*Insere elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento);

/*Retorno de elemento que estah no topo*/
int pilha_pop(Pilha *pilha, int *elemento);
```

```c
/*Funcao que checa se a pilha eh vazia*/
int pilha_vazia(Pilha *pilha);

/*Exclui a pilha completamente*/
void libera_pilha(Pilha *pilha);

#endif
```

ItemB: Implementação das funções no arquivo pilha_dinamica.c

```c
#ifndef PILHA_DINAMICA
#include "pilha_dinamica.h"
#include <stdlib.h>
#include <stdio.h>


struct pilha
{
    int *vetor;
    int top;
    int vetor_size;
};


/*Cria pilha*/
Pilha * cria_pilha(){
    Pilha *pi;
    pi = (Pilha *) malloc (sizeof(Pilha));

    if (pi){
        pi->top = -1;
        pi->vetor = (int *) malloc (sizeof(int));
        pi->vetor_size = 1; /*Cabe um elemento no vetor*/
    }

    return pi;
}
```

```c
/*Insere elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento){

    if (pilha->top == -1){

        /*Se for o primeiro elemento
          a ser inserido*/

        pilha->vetor[0] = elemento;

        pilha->top += 1;
        printf("Testanto se deu certo: %d\n", pilha->vetor[0]);
        return 1;

    }else if (pilha->top == pilha->vetor_size - 1) {
        /*Se a pilha jah estiver cheia
          alocamos um novo espaco*/

        pilha->top += 1;

        int posicao = pilha->top;

        pilha->vetor = (int *) realloc (pilha->vetor,
            (pilha->vetor[0] + pilha->top)*sizeof(int));

        if (pilha->vetor){
            pilha->vetor[posicao] = elemento;
            printf("Testanto se deu certo: %d\n", pilha->vetor[posicao]);
            return 1;
        }



    }else{
```

```
        /*Se a pilha ainda nao
           estiver cheia apenas
           inserimos o elemento*/

        pilha->top += 1;
        int posicao = pilha->top;
        pilha->vetor[pilha->top] = elemento;
        printf("Testanto se deu certo: %d\n", pilha->vetor[posicao]);
        return 1;
    }

    return 0;
}



/*Retorno de elemento que estah no topo*/
int pilha_pop(Pilha *pilha, int *elemento){

    if(!pilha)
        return 0;

    if(pilha->top == -1)
        return 0;

    *elemento = pilha->vetor[pilha->top];
    pilha->top -= 1;
    return 1;

}



/*Funcao que checa se a pilha eh vazia*/
int pilha_vazia (Pilha *pilha){
    return (pilha->top == -1);
```

```
}



/*Exclui a pilha completamente*/
void libera_pilha(Pilha *pilha){
    free(pilha->vetor);
    free(pilha->top);
    free(pilha->vetor_size);
    free(pilha);
};



#endif
```

ItemB: Teste de funcionalidade da pilha em um programa main.c

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

int main (){

    Pilha *p;
    int i;

    /*Criando uma pilha vazia com vetor dinamico*/
    p = cria_pilha();

    printf("Endereco da alocacao de memoria: %x\n", p);



    /*Testando a funcao pilha_vazia*/
    if (pilha_vazia(p)){
        printf("Pilha vazia\n");
    }else{
```

```
    printf("Pilha nao vazia\n");
}



/*Adicionando o primeiro elemento na pilha*/
pilha_push(p, 1);

pilha_push(p, 2);

pilha_push(p, 3);



/*Testando a funcao pilha_vazia*/
if (pilha_vazia(p)){
    printf("Pilha vazia\n");
}else{
    printf("Pilha nao vazia\n");
}



/*Tirando elementos da pilha pelo topo*/
int elemento;
pilha_pop(p, &elemento);
printf("Elemento do topo: %d\n", elemento);

pilha_pop(p, &elemento);
printf("Elemento do topo: %d\n", elemento);

pilha_pop(p, &elemento);
printf("Elemento do topo: %d\n", elemento);



/*Excluindo a pilha por completo*/
libera_pilha(p);
```

```
    return 0;
}
```

(c) Implemente uma versão TAD Pilha utilizando lista encadeada Resposta:

ItemC: Interface das funções do arquivo pilha_encadeada.h

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

typedef struct elemento No;

typedef struct pilha Pilha;


/*Criar pilha vazia*/
Pilha * cria_pilha();


/*Inserir elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento);


/*Retornar ou retirar elemento da pilha*/
int pilha_pop(Pilha *pilha);



/*Checa se a pilha eh vazia*/
int pilha_vazia(Pilha *pilha);


/*Exclui a pilha completamente*/
void libera_pilha(Pilha **pilha);
```

```c
/*Retorna o numero de elementos na pilha*/
int pilha_size(Pilha *pilha);

#endif
```

ItemC: Implementação das funções do arquivo pilha_encadeada.c

```c
#ifndef _PILHA_ENCADEADA_C
#define _PILHA_ENCADEADA_C

#include <stdio.h>
#include <stdlib.h>
#include "pilha_encadeada.h"

struct elemento
{
    int elemento;
    struct No *prox;
};

struct pilha
{
    struct No* elemento_topo;
    int topo;
};



/*Criar pilha vazia*/
Pilha * cria_pilha(){

    Pilha* p = (Pilha *) malloc (sizeof(Pilha));
    p->elemento_topo = (No*) malloc (sizeof(No));
    p->topo = 0;
    return p;
```

```
}



/*Inserir elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento){
    No* node = (No*) malloc (sizeof(No));
    node->prox = NULL;
    node->elemento = elemento;

    if (pilha->topo == 0){
        pilha->elemento_topo = node;
        pilha->topo = 1;

        printf("Numero no topo da pilha: %d\n", node->elemento);
        printf("Numero de elementos inseridos: %d\n", pilha->topo);

    }else{

        No* aux;
        aux = pilha->elemento_topo;
        pilha->elemento_topo = node;
        node->prox = aux;

        pilha->topo += 1;

        printf("Numero no topo da pilha: %d\n", node->elemento);
        printf("Numero de elementos inseridos: %d\n", pilha->topo);

    }

};



/*Retornar ou retirar elemento da pilha*/
int pilha_pop(Pilha *pilha){
```

```
    No* aux;
    int elemento;
    aux = pilha->elemento_topo;
    elemento = aux->elemento;
    pilha->elemento_topo = aux->prox;

    pilha->topo -= 1;

    free(aux);

    return elemento;
};



/*Checa se a pilha eh vazia*/
int pilha_vazia(Pilha *pilha){

    return (pilha->topo == 0);
};



/*Exclui a pilha completamente*/
void libera_pilha(Pilha **pilha){
    No* aux; /*Noh que percorre os elementos da pilha*/
    Pilha* auxiliar = pilha;

    int size = auxiliar->topo;
    int i;

    for (i = 0; i < size; i++){
        /*Liberando memoria de noh em noh*/
        aux = auxiliar->elemento_topo;
        auxiliar->elemento_topo = aux->prox;
        free(aux);
    }
```

```
    free(auxiliar);
    free(*pilha);
    free(pilha);
    *pilha = NULL;
};

/*Retorna o numero de elementos na pilha*/
int pilha_size(Pilha *pilha){
    return pilha->topo;
};




#endif
```

ItemC: Teste das funções no arquivo main.c

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

int main (){

    /*Testando a criacao da pilha*/
    Pilha* pi;
    pi = cria_pilha();

    printf("Endereco de memoria alocado para a pilha: %x\n", pi);




    /*Testando insercao de elemento na pilha*/
    pilha_push(pi, 2);

    pilha_push(pi, 4);
```

```
    pilha_push(pi, 50);

    printf("Numero de elementos antes da retirada: %d\n\n", pilha_size(pi));


    /*Testanto retirada dos elementos do topo*/
    int valor_topo1 = pilha_pop(pi);
    printf("Elemento no topo da pilha: %d\n Numero de elementos atualizado: %d\n
            \n", valor_topo1, pilha_size(pi));

    int valor_topo2 = pilha_pop(pi);
    printf("Elemento no topo da pilha: %d\n Numero de elementos atualizado: %d\n
            \n", valor_topo2, pilha_size(pi));

    int valor_topo3 = pilha_pop(pi);
    printf("Elemento no topo da pilha: %d\n Numero de elementos atualizado: %d\n
            \n", valor_topo3, pilha_size(pi));


    libera_pilha(&pi);

    return 0;
}
```

(d) Implemente uma versão do TAD Fila utilizando vetor dinâmico
Resposta:

ItemD: Interface das funções implementada no arquivo fila_dinamica.h

```
#ifndef _FILA_DINAMICA_H
#define _FILA_DINAMICA_H

typedef struct fila Fila;


/*Funcao para criar fila vazia*/
Fila* cria_fila();
```

```c
/*Funcao para inserir elementos na fila*/
int fila_push(Fila *fila, int elemento);


/*Funcao para acessar ou retirar elemento da fila*/
int fila_pop(Fila*fila);


/*Funcao checa se a fila eh vazia*/
int fila_vazia(Fila *fila);


/*Funcao que exclui a fila completamente*/
void libera_fila(Fila *fila);

#endif
```

ItemD: Funções implementadas no arquivo fila_dinamica.c

```c
#ifndef _FILA_DINAMICA_C
#define _FILA_DINAMICA_C

#include <stdio.h>
#include <stdlib.h>
#include "fila_dinamica.h"

struct fila
{
    int *vetor; /*Vetor com elementos da fila*/
    int begin; /*posicao do inicio da fila*/
    int size; /*numero de elementos dentro da fila*/
    int dim; /*dimensão do vetor*/
};


/*Funcao para criar fila vazia*/
```

```
Fila* cria_fila(){
    Fila* f = (Fila*) malloc (sizeof(Fila));

    f->dim = 1;

    f->vetor = (int *) malloc (sizeof(int)*f->dim);

    f->begin = 0;

    f->size = 0; /*Fila vazia*/

    return f;
};



/*Funcao para inserir elementos na fila*/
/*No caso da fila, os elementos sao adicionados ao
  final do vetor*/
int fila_push(Fila *fila, int elemento){

    int aux;

    if (fila->dim == fila->size){
        /*Se o vetor jah estiver cheio realocamos a memoria*/

        fila->dim *= 2; /*dobramos a capacidade do vetor*/

        fila->vetor = (int *) realloc (fila->vetor, sizeof(int) * fila->dim);

        if (fila->begin != 0){
            memmove(&fila->vetor[fila->dim - fila->begin],
            &fila->vetor[fila->begin],(fila->size - fila->begin)*sizeof(int));
        }
    }


    aux = (fila->begin + fila->size) % fila->dim;
```

```
        fila->vetor[aux] = elemento;

        fila->size++;

    return 1;
};



/*Funcao para acessar ou retirar elemento da fila*/
/*No caso da fila, os elementos sao retirados no
   inicio do vetor*/
int fila_pop(Fila*fila){

    int popped = fila->vetor[fila->begin];

    fila->begin = (fila->begin + 1) % fila->dim;
    fila->size--;

    return popped;
};



/*Funcao checa se a fila eh vazia*/
int fila_vazia(Fila *fila){
    return (fila->size == 0);
};


/*Funcao que exclui a fila completamente*/
void libera_fila(Fila *fila){
    free(fila->vetor);
    free(fila);
};
#endif
```

ItemD: Funções testadas no arquivo main.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "fila_dinamica.h"


int main () {

    /*Testando funcao que cria pilha vazia*/
    Fila* fi = cria_fila();

    printf ("Endereco da memoria alocada para a fila: %x\n", fi);

    /*Testando a funcao que checa se a pilha eh vazia*/
    if (fila_vazia(fi)){
        printf("Pilha vazia\n");

    }else{
        printf("Pilha nao vazia\n");
    }


    /*Testando a funcao de insercao de elementos na fila*/
    fila_push(fi, 1);

    fila_push(fi, 30);

    fila_push(fi, 15);

    /*Testando se a fila esta vazia novamente*/
    if (fila_vazia(fi)){
        printf("Pilha vazia\n");

    }else{
        printf("Pilha nao vazia\n");
    }
```

```
    /*Testando a retirada ou acesso aos elementos da fila*/
    printf("Primeiro elemento da fila: %d\n", fila_pop(fi));

    printf("Primeiro elemento da fila: %d\n", fila_pop(fi));

    printf("Primeiro elemento da fila: %d\n", fila_pop(fi));

    return 0;
}
```

(e) Implemente uma versão do TAD Fila utilizando lista encadeada
Resposta:

ItemE: Interface das funções implementada no arquivo fila_encadeada.h

```
#ifndef FILA_ENCADEADA_H
#define FILA_ENCADEADA_H


typedef struct elemento No;


typedef struct fila Fila;


/*Cria fila vazia*/
Fila* cria_fila();


/*insere elemento no final da fila*/
int fila_push(Fila *fila, int elemento);


/*Retira e retorna elemento do inicio da fila*/
int fila_pop(Fila *fila);
```

```
/*Checa se a fila eh vazia*/
int fila_vazia(Fila *fila);


/*Exclui a fila completamente*/
void libera_fila(Fila *fila);


/*Retorna o tamanho da fila*/
int fila_size(Fila* fila);


void print_fila(Fila *fila);

#endif
```

ItemE: Funções implementadas no arquivo fila_encadeada.c

```
#ifndef FILA_ENCADEADA_C
#define FILA_ENCADEADA_C

#include<stdio.h>
#include<stdlib.h>
#include "fila_encadeada.h"

struct elemento
{
    int valor;
    No* prox;
};

struct fila
{
    No* inicio;
    No* fim;
    int size;
};
```

```
/*Cria fila vazia*/
Fila* cria_fila(){

    /*Aloca-se memoria mas nenhum valor eh inserido*/
    Fila* fila = (Fila*) malloc (sizeof(Fila));

    fila->fim = NULL;

    fila->inicio = NULL;

    fila->size = 0;

};


/*insere elemento no final da fila*/
int fila_push(Fila *fila, int elemento){
    /*Criamos um noh para percorrer a fila*/
    No* aux = (No*) malloc (sizeof(No));

    aux->prox = NULL;
    aux->valor = elemento;

    if(fila->fim == NULL){
        fila->inicio = aux;

    }else{
        fila->fim->prox = aux;
    }

    fila->fim = aux;

    fila->size++;
};
```

```
/*Retira e retorna elemento do inicio da fila*/
int fila_pop(Fila *fila){
    /*Noh auxiliar para percorrer lista*/
    No* aux = fila->inicio;
    int popped;

    popped = aux->valor;
    fila->inicio = aux->prox;

    free(aux);
    if (!fila->inicio){
        fila->fim = NULL;
    }

    fila->size--;
    return popped;
};


/*Checa se a fila eh vazia*/
int fila_vazia(Fila *fila){
    return (fila->inicio == NULL);
};



/*Exclui a fila completamente*/
void libera_fila(Fila *fila){
    /*Devemos percorrer a lista, apagando elemento
    por elemento*/
    int i;
    No* aux1 = fila->inicio;

    while (aux1)
    {
        /* code */
        No* aux2;
        aux2 = fila->inicio->prox;
```

```
        free(aux1);

        aux1 = aux2;
    }

    free(fila);
};



int fila_size(Fila* fila){
    return fila->size;
}




void print_fila(Fila* fila){
    No* aux = fila->inicio;
    while (aux != NULL){
        printf("Componente da fila: %d\n", aux->valor);

        aux = aux->prox;
    }
}

#endif
```

ItemE: Funções testadas no arquivo main.c

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


int main (){

    int i;
```

```c
int valor;

/*Criando fila vazia*/
Fila *fi;
fi = cria_fila();

printf("Endereco de memoria alocada para a TAD: %x\n", fi);



/*Testando se a fila eh vazia*/
if (fila_vazia(fi)){
    printf("Vazia\n");
}else
{
    printf("Nao vazia\n");
}



/*Inserindo elementos*/
for(i = 0; i < 7; i++){
    fila_push(fi, i);
}

printf("Tamanho da fila depois da insercao: %d\n", fila_size(fi));



/*Testando se a fila eh vazia novamente*/
if (fila_vazia(fi)){
    printf("Vazia\n");
}else
{
    printf("Nao vazia\n");
}
```

```
    print_fila(fi);

    /*Retirando e retornando elementos*/
    for (i = 0; i < 7; i++){

        printf("Elemento retirado da fila: %d  ", fila_pop(fi));
        printf("Tamanho da fila: %d\n\n", fila_size(fi));
    }


    libera_fila(fi);


    return 0;
}
```

Questão 2: Sobre tabelas de dispersão

(a) Utilizando suas palavras, faça um resumo do ”Capítulo 19 - Tabelas de dispersão” do livro ”Celes, W. et al.; Introdução a Estrutura de Dados com Técnicas de Programação em C, 2ed., Elsevier, 2016”.

Resposta:

1. Definição:

   Uma tabela de dispersão (ou tabela *hash*) é uma estrutura de dados que possui uma definição um tanto quanto vaga. O seu conceito principal, entretanto, está em utilizar um mecanismo de busca para resgatar um determinado valor da memória de forma eficiente. Para nos aprofundarmos, precisamos lembrar um pouco do conceito de *array*, uma estrutura de busca eficiente, pois funciona através da busca por índices, e por isso, permite acessos randômicos rápidos. Uma tabela *hash* terá um conceito semelhante, em que podemos utilizar um ”índice” ou chave de busca para acessar um determinado valor armazenado em ordem constante $O(1)$.

Para que haja este acesso de forma eficiente, dada uma chave de busca, devemos utilizar apenas seus dígitos mais significativos como índices da tabela. Dessa forma, economizamos memória.

2. Função de dispersão (ou função *hashing*)

   A função *hashing* tem um papel fundamental de mapear para uma chave de busca um índice da tabela. Isso significa que para uma chave de busca ela retornará um índice onde o valor desejado pode ser acessado.Ou seja, uma função *hashing* "dispersa" os dados pela tabela.

   A implementação de uma função de dispersão precisa ser bem pensada, pois a velocidade de acesso característico da tabela *hash* é diretamente associada à estrutura dessa função. Uma implementação que permite a ocorrência de muitas colisões (quando há um mapeamento de dois dados diferentes para o mesmo índice de uma tabela) enfraquece a eficiência da estrutura, uma vez que os tratamentos de colisões são necessários com mais frequência. Para evitar as colisões e minimizá-las, devemos ocupar a tabela com até 50% da sua capacidade total e escrever o código de forma a dispersar os dados o máximo o possível pela estrutura. Outro ação importante é escolher um número primo para ser a dimensão da tabela.

3. Tratamentos de colisão

   Os tratamentos de colisão são necessários quando duas chaves de busca são mapeadas para o mesmo índice da tabela, ou seja, um dos dados a ser inserido na tabela é direcionado para um índice já preenchido. Para tratar dessa ocorrência, vários métodos são criados, de forma que, ainda que a colisão aconteça, o segundo elemento será armazenado em algum lugar da tabela e, portanto, o dado não será perdido. Esses métodos de tratamento são necessários na implementação de funções que manipulam a tabela *hash*, como a função de inserir um elemento ou a função de buscá-lo, por exemplo.

4. A implementação da tabela *hash* e vetores

   A implementação da tabela *hash* é feita com vetores estáticos ou dinâmicos e, assim como em estruturas como a pilha e a fila, temos como critério de escolha de forma de implementação se sabemos

ou não a quantidade de elementos a serem inseridos na tabela. Se não soubermos, o mais apropriado é criar um vetor dinâmico cuja memória pode ser realocada se o tamanho máximo de vetores for atingido. Caso já soubermos qual o número de elementos que serão inseridos, podemos utilizar vetores estáticos.

(b) Implemente uma versão do TAD Hashmap utilizando vetor dinâmico. Utilize o método da divisão para dispersão e o método da primeira posição livre para tratamento de colisões

ItemB: Interface da funções no arquivo hash_dinamico.h

```
#ifndef _HASH_DINAMICO_H
#define _HASH_DINAMICO_H

/*Definimos as estruturas*/
typedef struct pessoa Pessoa;

typedef struct  hash Hash;


/*funcao cria_pessoa*/
Pessoa* cria_pessoa(char* nome_completo, unsigned long int cpf);


/*funcao que retorna cpf da pessoa*/
unsigned long int get_cpf(Pessoa* pessoa);


/*Criamos a tabela hash vazia*/
Hash* cria_tabela();


/*Criamos a funcao hashing para dispersao dos elementos*/
unsigned long int hashing(Hash* ha, unsigned long int cpf);
```

```c
/*Inserimos um elemento na tabela
(utilizando método da primeira posicao
livre para tratamento de colisoes)*/
int insere_hash (Hash* hash, Pessoa* person);



/*Retirando elementos do hash*/
int retira_hash(Hash* hash, unsigned long int cpf);



/*Procurando por elementos na tabela de dispersao pelo cpf*/
char* search_table(Hash* hash, unsigned long int cpf);

/*Printa os itens da tabela*/
void print_table(Hash* hash);


/*Exclui a tabela hashing completamente*/
void free_hash(Hash* hash);



#endif
```

ItemB: Funções implementadas no arquivo hash_dinamico.c

```c
#ifndef _HASH_DINAMICO_C
#define _HASH_DINAMICO_C

#define N 1001

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

```c
struct pessoa
{
    char* nome;/*Nome da pessoa*/
    unsigned long int cpf;// Apenas os cinco ultimos numeros
};

struct hash
{
    Pessoa** lista; /*vetor de ponteiros para pessoas*/
    int tamanho; /*Capacidade do vetor*/
    int n_pessoas;/*quantidade de pessoas dentro do vetor*/
};



/*funcao cria_pessoa*/
Pessoa* cria_pessoa(char* nome_completo, unsigned long int cpf){
    Pessoa* p = (Pessoa*) malloc (sizeof(Pessoa));

    p->nome = nome_completo;
    p->cpf = cpf;
    return p;
};



/*funcao que retorna cpf da pessoa*/
unsigned long int get_cpf(Pessoa* pessoa){
    return pessoa->cpf;
};







/*Criamos a tabela hash vazia*/
Hash* cria_tabela(){
```

```
    int i;

    /*Alocamos memoria inicial para a estrutura*/
    Hash* h = (Hash*) malloc (sizeof(Hash));

    /*Alocamos memoria inicial para o vetor de ponteiros*/
    h->lista = (Pessoa*) malloc (sizeof(Pessoa)*N);

    /*Deixamos todos os ponteiros do vetor nulos*/
    for (i = 0; i < N; i++){
        h->lista[i] = NULL;
    }

    h->n_pessoas = 0;
    h->tamanho = N;

    return h;

};



/*Criamos a funcao hashing para dispersao dos elementos*/
unsigned long int hashing(Hash* ha, unsigned long int cpf){
    return cpf % (ha->tamanho);
};




/*Inserimos um elemento na tabela
(utilizando método da primeira posicao
livre para tratamento de colisoes)*/
int insere_hash (Hash* hash, Pessoa* person){
    int index = hashing(hash, person->cpf);

    /*Se o numero de elementos for 75% da capacidade do vetor
    realocamos mais espaco */
```

```
    int aux = 3 * hash->tamanho/4;

    if (hash->n_pessoas > aux){

        hash->tamanho *=2;

        printf("Foi utilizada a realocacao de memoria\n");
        hash->lista = (Pessoa*) realloc (hash->lista, sizeof(Pessoa)*
                                                    hash->tamanho);

    }



    if (hash->lista[index] == NULL){
        hash->lista[index] = person;
        printf("Foi inserido na tebela hash\n");


    }else{
        printf("Teve que passar por tratamento de colisao na insercao");
        /*Tratando colisoes*/
        int i;
        int aux2;
        for(i = 0; i < hash->tamanho; i++){
            aux2 = (i + index) % hash->tamanho;

            if(hash->lista[aux2] == NULL){
                hash->lista[aux2] = person;
                break;
            }
        }
    }

    hash->n_pessoas++;

    return 1;
};
```

```
/*Retirando elementos do hash*/
int retira_hash(Hash* hash, unsigned long int cpf){
    unsigned long int index = hashing(hash, cpf);
    int aux;

    if(hash->lista[index]->cpf == cpf){
        free(hash->lista[index]);
        return 1;

    }else{
        for(int i = 0; i < hash->tamanho; i++){
            aux = (i + index) % hash->tamanho;

            if (hash->lista[aux]->cpf == cpf){
                free(hash->lista[aux]);
                return 1;
            }
        }
    }


    return 0;
};
```

```c
/*Procurando por elementos na tabela de dispersao pelo cpf*/
char* search_table(Hash* hash, unsigned long int cpf){
    /*variavel que armazena o indice do vetor que queremos*/
    unsigned long int index = hashing(hash, cpf);
    int i;
    int auxiliar;
    char* nome;
    int flag = 0; /*Checa se foi encontrado*/

    if(hash->lista[index]->cpf == cpf){
        nome = hash->lista[index]->nome;
        flag++;
    }else{
        printf("Tratando colisoes na procura pela pessoa\n");
        /*Tratando colisoes*/
        for (i = 0; i < hash->tamanho; i++){
            auxiliar = (i + index) % hash->tamanho;

            if(hash->lista[auxiliar]->cpf == cpf){
                nome = hash->lista[auxiliar]->nome;
                flag++;
                break;
            }
        }
    }

    if (flag == 0){
        return "0"; /*Item nao encontrado*/
    }

    return nome;
};



/*Printando os itens da tabela*/
void print_table(Hash* hash){
```

```
    int i;
    for (i = 0; i < hash->tamanho; i++){
        if(hash->lista[i] != NULL){
            printf("Nome: %s, CPF: %d\n", hash->lista[i]->nome,
                                                hash->lista[i]->cpf);
        }
    };
};



/*Exclui a tabela hashing completamente*/
void free_hash(Hash* hash){
    int i;
    for (i = 0; i < hash->tamanho; i++){
        free(hash->lista[i]);
    }

    free(hash);
};


#endif
```

ItemB: Teste das funções em um arquivo main.c

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


int main (){

    /*Criando tabela hash*/
    Hash* tabela = cria_tabela();
```

```
/*Criando uma struct de pessoa*/
Pessoa* person1 = cria_pessoa("John Person Paraiba", 56454);

Pessoa* person2 = cria_pessoa("Victoria Monteiro Pontes", 12345);

Pessoa* person3 = cria_pessoa("Eu amo chocolate", 54321);

/*Inserindo na tabela hash*/
insere_hash(tabela, person1);

insere_hash(tabela, person2);

insere_hash(tabela, person3);


print_table(tabela);


/*Busca na tabela*/

printf("%s\n", search_table(tabela, get_cpf(person1)));


printf("%s\n", search_table(tabela, get_cpf(person2)));


printf("%s\n", search_table(tabela, get_cpf(person3)));



/*Retirando da tabela*/
int teste;
teste = retira_hash(tabela, get_cpf(person1));

/*Confirmando que o item que foi retirado*/
if(teste) printf("Item retirado");
else printf("Item ainda estah na tabela");
```

```
    free_hash(tabela);


    return 0;
}
```

(c) Implemente uma versão do TAD Hashmap utilizando vetor dinâmico. Utilize o método da divisão para a dispersão e o método da dispersão dupla para tratamento de colisões:

Observação: Como a implementação do vetor dinâmico e do método de divisão para dispersão é o mesmo, a única modificação feita no arquivo hash_dinamico.h e hash_dinamico.c é na função de hashing e de inserção. Portanto, será posto aqui nesta questão apenas a modificação dessas funções, todo o resto do código que não era afetado por essa modificação de método foi reaproveitado do item B para o item C. Vale lembrar que as duas questões estão implementadas separadamente em suas devidas pastas, apenas os nomes dos arquivos são iguais.

ItemC: Funções na interface do arquivo hash_dinamico.h

```
#ifndef _HASH_DINAMICO_H
#define _HASH_DINAMICO_H


/*Definimos as estruturas*/
typedef struct pessoa Pessoa;

typedef struct  hash Hash;



/*funcao cria_pessoa*/
Pessoa* cria_pessoa(char* nome_completo, unsigned long int cpf);


/*funcao que retorna cpf da pessoa*/
unsigned long int get_cpf(Pessoa* pessoa);
```

```c
/*Criamos a tabela hash vazia*/
Hash* cria_tabela();


/*Criamos a funcao hashing para dispersao dos elementos*/
unsigned long int hashing(Hash* ha, unsigned long int cpf);


/*Criamos uma segunda funcao hashing para o tratamento de coisões*/
unsigned long int hashing2(Hash* ha, unsigned long int cpf);



/*Inserimos um elemento na tabela
(utilizando método da double hashing
para o tratamento de colisoes)*/
int insere_hash (Hash* hash, Pessoa* person);




/*Retirando elementos do hash*/
int retira_hash(Hash* hash, unsigned long int cpf);



/*Procurando por elementos na tabela de dispersao pelo cpf*/
char* search_table(Hash* hash, unsigned long int cpf);

/*Printa os itens da tabela*/
void print_table(Hash* hash);

/*Exclui a tabela hashing completamente*/
void free_hash(Hash* hash);
```

```
#endif
```

ItemC: Implementação dessas funções no arquivo hash_dinamico.c

```
#ifndef _HASH_DINAMICO_C
#define _HASH_DINAMICO_C

#define N 1001

#include <stdio.h>
#include <stdlib.h>
#include "hash_dinamico.h"

struct pessoa
{
    char* nome;/*Nome da pessoa*/
    unsigned long int cpf;// Apenas os cinco ultimos numeros
};

struct hash
{
    Pessoa** lista; /*vetor de ponteiros para pessoas*/
    int tamanho; /*Capacidade do vetor*/
    int n_pessoas;/*quantidade de pessoas dentro do vetor*/
};



/*funcao cria_pessoa*/
Pessoa* cria_pessoa(char* nome_completo, unsigned long int cpf){
    Pessoa* p = (Pessoa*) malloc (sizeof(Pessoa));

    p->nome = nome_completo;
    p->cpf = cpf;
    return p;
};
```

```c
/*funcao que retorna cpf da pessoa*/
unsigned long int get_cpf(Pessoa* pessoa){
    return pessoa->cpf;
};
```

```c
/*Criamos a tabela hash vazia*/
Hash* cria_tabela(){
    int i;

    /*Alocamos memoria inicial para a estrutura*/
    Hash* h = (Hash*) malloc (sizeof(Hash));

    /*Alocamos memoria inicial para o vetor de ponteiros*/
    h->lista = (Pessoa*) malloc (sizeof(Pessoa)*N);

    /*Deixamos todos os ponteiros do vetor nulos*/
    for (i = 0; i < N; i++){
        h->lista[i] = NULL;
    }

    h->n_pessoas = 0;
    h->tamanho = N;

    return h;

};
```

```c
/*Criamos a funcao hashing para dispersao dos elementos*/
unsigned long int hashing(Hash* ha, unsigned long int cpf){
    return cpf % (ha->tamanho);
```

```
};


/*Criamos uma segunda funcao hashing para o tratamento de coisões*/
unsigned long int hashing2(Hash* ha, unsigned long int cpf){
    return cpf % (ha->tamanho*2/5);
};




/*Inserimos um elemento na tabela
(utilizando método de double hashing ou funcao dupla dispersao)*/
int insere_hash (Hash* hash, Pessoa* person){
    int index = hashing(hash, person->cpf);

    /*Se o numero de elementos for 75% da capacidade do vetor
    realocamos mais espaco */
    int aux = 3 * hash->tamanho/4;

    if (hash->n_pessoas > aux){

        hash->tamanho *=2;
        printf("Foi utilizada a realocacao de memoria\n");
        hash->lista = (Pessoa*) realloc (hash->lista,
                                    sizeof(Pessoa)* hash->tamanho);

    }



    if (hash->lista[index] == NULL){
        hash->lista[index] = person;
        printf("Foi inserido na tebela hash\n");


    }else{
```

```
        printf("Teve que passar por tratamento de colisao na insercao");
        /*Tratando colisoes*/
        int index2 = hashing2(hash, person->cpf);
        int h = (index + index2) % hash->tamanho;

        if (hash->lista[h] == NULL){
            hash->lista[h] = person;
        }else{
            return 0;
        }

    }

    hash->n_pessoas++;

    return 1;
};
```

```
/*Retirando elementos do hash*/
int retira_hash(Hash* hash, unsigned long int cpf){
    unsigned long int index = hashing(hash, cpf);
    int aux;

    if(hash->lista[index]->cpf == cpf){
        free(hash->lista[index]);
        return 1;

    }else{
        for(int i = 0; i < hash->tamanho; i++){
            aux = (i + index) % hash->tamanho;

            if (hash->lista[aux]->cpf == cpf){
                free(hash->lista[aux]);
```

```
                return 1;
            }
        }
    }


    return 0;
};
```

```
/*Procurando por elementos na tabela de dispersao pelo cpf*/
char* search_table(Hash* hash, unsigned long int cpf){
    /*variavel que armazena o indice do vetor que queremos*/
    unsigned long int index = hashing(hash, cpf);
    int i;
    int auxiliar;
    char* nome;
    int flag = 0; /*Checa se foi encontrado*/

    if(hash->lista[index]->cpf == cpf){
        nome = hash->lista[index]->nome;
        flag++;
    }else{
        printf("Tratando colisoes na procura pelo meliante\n");
        /*Tratando colisoes*/
        for (i = 0; i < hash->tamanho; i++){
            auxiliar = (i + index) % hash->tamanho;

            if(hash->lista[auxiliar]->cpf == cpf){
                nome = hash->lista[auxiliar]->nome;
                flag++;
                break;
            }
```

```
        }
    }

    if (flag == 0){
        return "0"; /*Item nao encontrado*/
    }

    return nome;
};



/*Printando os itens da tabela*/
void print_table(Hash* hash){
    int i;
    for (i = 0; i < hash->tamanho; i++){
        if(hash->lista[i] != NULL){
            printf("Nome: %s, CPF: %d\n", hash->lista[i]->nome,
                                            hash->lista[i]->cpf);
        }
    };
};



/*Exclui a tabela hashing completamente*/
void free_hash(Hash* hash){
    int i;
    for (i = 0; i < hash->tamanho; i++){
        free(hash->lista[i]);
    }

    free(hash);
};


#endif
```

ItemC: Teste dessas funções no arquivo main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash_dinamico.h"


int main (){

    /*Criando tabela hash*/
    Hash* tabela = cria_tabela();

    /*Criando uma struct de pessoa*/
    Pessoa* person1 = cria_pessoa("Carolyn Wonderland", 56454);

    Pessoa* person2 = cria_pessoa("Shannon McNally", 12345);

    Pessoa* person3 = cria_pessoa("Cantoras e guitarristas", 54321);

    /*Inserindo na tabela hash*/
    insere_hash(tabela, person1);

    insere_hash(tabela, person2);

    insere_hash(tabela, person3);

    print_table(tabela);

    /*Busca na tabela*/

    printf("%s\n", search_table(tabela, get_cpf(person1)));


    printf("%s\n", search_table(tabela, get_cpf(person2)));


    printf("%s\n", search_table(tabela, get_cpf(person3)));
```

```
    /*Retirando da tabela*/
    int teste;
    teste = retira_hash(tabela, get_cpf(person1));

    /*Confirmando que o item que foi retirado*/
    if(teste) printf("Item retirado");
    else printf("Item ainda estah na tabela");

    free_hash(tabela);


    return 0;
}
```

Questão 3: Sobre a complexidade de algoritmos

(a) Utilizando suas palavras, faça o resumo da ”Seção 2.3 - Ordens Assintóticas” do livro ”Toscani, L. V. e Veloso, P. A. S.; Complexidade de Algoritmos, vol. 13, 3 ed., UFRGS, 2012”. Utilize o conhecimento adquirido e justifique se podemos afirmar que $f(n) = n$ é $O(n^2)$

Resposta:

1. **Ordens de crescimento assintótico:**

   Ao vermos uma expressão como $n + 10$ ou $n^2 + 1$, a maioria das pessoas pensa automaticamente em valores pequenos. Ao analisarmos um algoritmo, devemos fazer exatamente o contrário: ignorar os valores pequenos e focar nos valores enormes ou suficientemente grandes de n.

   Para valores enormes de n, as funções:

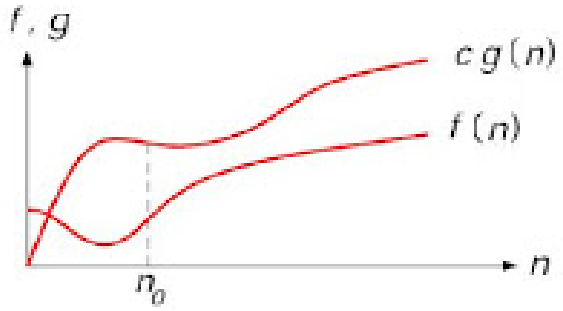
   $$n^2, (3/2)n^2, 9999n^2, n^2/1000, n^2 + 100$$

   crescem todas com a mesma velocidade em função de n e portanto são todas ”equivalentes”.

A matemática que foca nesse aspecto é chamada assintótica. Nesta área, as funções são classificadas em ”ordens assintóticas”. Todas as funções de uma mesma ordem são ”equivalentes”. As funções citadas, por exemplo, pertencem à ordem n^2.

2. **Cota assintótica superior**

   Uma cota assintótica superior é uma função que, comparada a outra, cresce mais rapidamente, ou seja, a partir de um número n_0 suficientemente grande, ela passará a ser sempre maior que a outra função a qual é comparada. Por exemplo:



$$f(n) = O(g(n))$$

Figure 3: Gráfico mostrando o crescimento de duas funções

Nesta figura, temos que a partir de um determinado número n_0 a função $c.g(n)$ domina sobre a função $f(n)$, logo, a função $c.g(n)$ é uma cota assintótica superior.

3. **Notação** O

   A notação O define, matematicamente, uma cota assintótica superior. De forma bem simplificada, tendo duas funções assintoticamente não-negativas f e g, se f está na ordem O de g, isso significa que a partir de um determinado número n_0, para todo $n > n_0$, a expressão $f(n) \leq c.g(n)$ sempre será verdadeira, sendo c uma constante positiva.

   A notação assintótica desse exemplo é dado por $f = O(g)$.

4. **Notação Ω**

   A ordem Ω tem um conceito contrário à ordem O, portanto, ela define, matematicamente, uma cota assintótica inferior. Ou seja, dadas as funções f e g definidas nas mesmas condições anteriores, f está na ordem Ω de g e portanto, $f = \Omega(g)$, se a partir de um determinado número n_0 suficientemente grande, $f(n)$ sempre será maior que $c.g(n)$. O que nos leva a expressão $f(n) \geq c.g(n)$ para todo $n > n_0$, sendo c uma constante positiva.

5. **Notação Θ**

   Além dos conceitos de cotas assintóticas superiores e inferiores, existe o conceito de cota assintótica exata, que define que duas funções que, comparadas uma com a outra, crescem com a mesma velocidade. Matematicamente, esse conceito é definido pela notação Θ. Ou seja, dadas as funções f e g definidas nas condições anteriores, ambas pertencem a mesma ordem ($f = \Theta(g)$) se $f = O(g)$ e $f = \Omega(g)$. Portanto, existem constantes positivas c e d tais que satisfazem, respectivamente, uma condição de existência de cota assintótica superior e inferior, e a expressão $d.g(n) \leq f(n) \leq c.g(n)$ se torna verdadeira para todo n suficientemente grande.

6. **Aplicação para a medição de desempenho de um algoritmo**

   A análise assintótica é necessária na área de algoritmos e programação para termos uma estimativa do desempenho computacional. Ao analisarmos a complexidade de um algoritmo, estamos fazendo uma estimativa de eficiência desse algoritmo. Através das definições matemáticas de cotas assintóticas superior, inferior e exatas, podemos estimar a quantidade iterações ou ”passos” no algoritmo em função do tamanho da sua entrada.

7. **Podemos afirmar que $f(n) = n$ é $O(n^2)$?**

   A resposta é **sim**. A justificativa é que a função, como podemos analisar, é linear, ou seja, o resultado da função (o valor de $f(n)$) é influenciado diretamente pelo tamanho da entrada (n), sem fatores exponenciais. Assim, ele cresce mais devagar que n^2, então temos que: $f(n) \leq c.g(n)$, sendo $g(n) = n^2$ e c uma constante positiva.

(b) Deduzir a complexidade, utilizando a notação O (o-grande) para os itens das funções implementadas na questão 1 item (b)

Resposta:

Observação: As respostas terão a implementação das funções e alguns comentários no código sobre como se obteve a resposta. No caso, $T(n)$ irá significar a quantidade de operações em função do tamanho da entrada(n). Portanto, em cada implementação, haverá a contagem de operações feitas. Laços *for* não-aninhados possuem n operações, enquanto dois laços *for* aninhados terão n^2 operações. Outra observação a ser feita é que em casos de função *malloc* e outras funções comuns de serem chamadas, também foram consideradas como 1 operação só por não sabermos como ela foi implementada e portanto, não sabemos como funcionam suas operações. Ainda assim, a estimativa é apenas para deduzir a ordem das funções em função da entrada n, portanto, a proposta prevalece.

1. Função cria_pilha()

```
/*Cria pilha*/
Pilha * cria_pilha(){
    Pilha *pi;
    pi = (Pilha *) malloc (sizeof(Pilha));/*1 atribuição*/

    if (pi){                                /*1 comparação*/
        pi->top = -1; /*1 atribuição*/
        pi->vetor = (int *) malloc (sizeof(int)); /*1 atribuição*/
        pi->vetor_size = 1; /*Cabe um elemento no vetor*//*1 atribui-
                                                          ção*/
    }

    return pi;  /*1 retorno*/
}
```

São 5 operações, independente do tamanho da entrada (não possui).

$T(n) = 1 + 1 + 1 + 1 + 1 -> T(n) = 5$.

Portanto sua complexidade é da ordem $O(1)$, por ser constante.

2. Função pilha_push(Pilha* pilha, int elemento)

```
    /*Insere elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento){

    if (pilha->top == -1){ /*1 comparação*/

        /*Se for o primeiro elemento
          a ser inserido*/

        pilha->vetor[0] = elemento; /*1 atribuição*/

        pilha->top += 1;  /*1 atribuição*/

        return 1; /*1 retorno*/

    }else if (pilha->top == pilha->vetor_size - 1) /*1 comparação*/
    {
        /*Se a pilha jah estiver cheia
          alocamos um novo espaco*/

        pilha->top += 1; /*1 atribuição*/

        int posicao = pilha->top; /*1 atribuição*/

        pilha->vetor = (int *) realloc (pilha->vetor, (pilha->vetor[0] +
            pilha->top)*sizeof(int)); /*1 atribuição*/

        if (pilha->vetor){ /*1 comparação*/
            pilha->vetor[posicao] = elemento; /*1 atribuição*/

            return 1; /*1 retorno*/
        }

    }else{ /*1 comparação*/
        /*Se a pilha ainda nao
           estiver cheia apenas
           inserimos o elemento*/

        pilha->top += 1; /*1 atribuição*/
```

```
        int posicao = pilha->top; /*1 atribuição*/
        pilha->vetor[pilha->top] = elemento; /*1 atribuição*/

        return 1; /*1 retorno*/
    }

    return 0; /*1 retorno*/
}
```

São 17 operações, independente do tamanho da entrada.

$T(n) = 17$

Portanto sua complexidade é da ordem $O(1)$, por ser constante.

3. Função pilha_pop(Pilha* pilha)

```
    /*Retorno de elemento que estah no topo*/
int pilha_pop(Pilha *pilha, int *elemento){

    if(!pilha) /*1 comparação*/
        return 0; /*1 retorno*/

    if(pilha->top == -1) /*1 comparação*/
        return 0;  /*1 retorno*/

    *elemento = pilha->vetor[pilha->top]; /*1 atribuição*/
    pilha->top -= 1;  /*1 atribuição*/
    return 1; /*1 retorno*/

}
```

São 7 operações, independentes da entrada.

$T(n) = 7$

Portanto a complexidade é da ordem $O(1)$, por ser constante.

4. Função pilha_vazia()

```
    /*Funcao que checa se a pilha eh vazia*/
int pilha_vazia (Pilha *pilha){
```

```
    return (pilha->top == -1); /*1 retorno*/
}
```

É feita somente uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$

5. Função libera_pilha(Pilha* pilha)

```
    /*Exclui a pilha completamente*/
void libera_pilha(Pilha *pilha){
    free(pilha->vetor); /*1 operação*/
    free(pilha->top);  /*1 operação*/
    free(pilha->vetor_size); /*1 operação*/
    free(pilha);  /*1 operação*/
};
```

São feitas 4 operações.

$T(n) = 4$

Portanto, complexidade de ordem $O(1)$

(c) Deduzir a complexidade, usando notação O (o-grande)para os itens das funções implementadas na questão 1 item (c)

Resposta:

1. Função cria_pilha

```
    /*Criar pilha vazia*/
Pilha * cria_pilha(){

    Pilha* p = (Pilha *) malloc (sizeof(Pilha)); /*1 operação*/
    p->elemento_topo = (No*) malloc (sizeof(No)); /*1 operação*/
    p->topo = 0; /*1 atribuição*/
    return p; /*1 retorno*/
}
```

São 4 operações, independente da entrada (não possui).

$T(n) = 4$

Portanto a complexidade é de ordem $O(1)$

2. Função pilha_push(Pilha* pilha, int elemento)

```
    /*Inserir elemento no topo da pilha*/
int pilha_push(Pilha *pilha, int elemento){
    No* node = (No*) malloc (sizeof(No)); /*1 operação*/
    node->prox = NULL; /*1 atribuição*/
    node->elemento = elemento; /*1 atribuição*/

    if (pilha->topo == 0){ /*1 comparação*/
        pilha->elemento_topo = node; /*1 atribuição*/
        pilha->topo = 1; /*1 atribuição*/

    }else{

        No* aux;
        aux = pilha->elemento_topo; /*1 atribuição*/
        pilha->elemento_topo = node; /*1 atribuição*/
        node->prox = aux; /*1 atribuição*/

        pilha->topo += 1; /*1 atribuição*/
    }

};
```

São 10 operações, independentes da entrada.

$T(n) = 10$

Portanto a complexidade é da ordem $O(1)$.

3. Função pilha_pop(Pilha* pilha)

```
    /*Retornar ou retirar elemento da pilha*/
int pilha_pop(Pilha *pilha){
    No* aux;
    int elemento;
    aux = pilha->elemento_topo; /*1 atribuição*/
    elemento = aux->elemento; /*1 atribuição*/
    pilha->elemento_topo = aux->prox;  /*1 atribuição*/
```

```
    pilha->topo -= 1; /*1 atribuição*/

    free(aux); /*1 operação*/

    return elemento; /*1 retorno*/
};
```

São 6 operações, independentes da entrada.

$T(n) = 6$

Portanto a complexidade é da ordem $O(1)$.

4. Função pilha_vazia(Pilha* pilha)

```
    /*Checa se a pilha eh vazia*/
int pilha_vazia(Pilha *pilha){

    return (pilha->topo == 0); /*1 retorno*/
};
```

É feita apenas uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$

5. Função libera_pilha(Pilha* pilha)

```
    /*Exclui a pilha completamente*/
void libera_pilha(Pilha **pilha){
    No* aux; /*Noh que percorre os elementos da pilha*/
    Pilha* auxiliar = pilha; /*1 atribuição*/

    int size = auxiliar->topo; /*1 atribuição*/
    int i;

    for (i = 0; i < size; i++){   /*n operações*/
        /*Liberando memoria de noh em noh*/
        aux = auxiliar->elemento_topo; /*n atribuições*/
        auxiliar->elemento_topo = aux->prox; /*n atribuições*/
        free(aux); /*n operações*/
```

```
    }

    free(auxiliar); /*1 operação*/
    free(*pilha); /*1 operação*/
    free(pilha); /*1 operação*/
    *pilha = NULL; /*1 atribuição*/
};
```

São $4n + 6$ operações, dependendo do tamanho da entrada n.

$T(n) = 4n + 6$

Portanto a complexidade é de ordem $O(n)$.

6. Função pilha_size(Pilha* pilha)

```
    /*Retorna o numero de elementos na pilha*/
int pilha_size(Pilha *pilha){
    return pilha->topo; /*1 retorno*/
};
```

É feita apenas uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$

(d) Deduzir a complexidade, usando notação O (o-grande)para os itens das funções implementadas na questão 1 item (d)

Resposta:

1. Função cria_fila()

```
    /*Funcao para criar fila vazia*/
Fila* cria_fila(){
    Fila* f = (Fila*) malloc (sizeof(Fila)); /*1 operação*/

    f->dim = 1; /*1 atribuição*/

    f->vetor = (int *) malloc (sizeof(int)*f->dim); /*1 operação*/

    f->begin = 0;   /*1 atribuição*/
```

```
    f->size = 0; /*Fila vazia*/ /*1 atribuição*/

    return f; /*1 retorno*/
};
```

São 6 operações, independentes da entrada (não possui).

$T(n) = 6$

Portanto a complexidade é da ordem $O(1)$.

2. Função fila_push(Fila* fila, int elemento)

```
    /*Funcao para inserir elementos na fila*/
/*No caso da fila, os elementos sao adicionados ao
  final do vetor*/
int fila_push(Fila *fila, int elemento){

    int aux;

    if (fila->dim == fila->size){ /*1 comparação*/
        /*Se o vetor jah estiver cheio realocamos a memoria*/

        fila->dim *= 2; /*dobramos a capacidade do vetor*/ /*1 atribuição*/

        fila->vetor = (int *) realloc (fila->vetor,
                                sizeof(int) * fila->dim); /*1 operação*/

        if (fila->begin != 0){ /*1 comparação*/
            memmove(&fila->vetor[fila->dim - fila->begin],
            &fila->vetor[fila->begin], (fila->size - fila->begin)*sizeof
            (int)); /*1 operação*/
        }
    }


    aux = (fila->begin + fila->size) % fila->dim; /*1 atribuição*/
        fila->vetor[aux] = elemento;/*1 atribuição*/
```

```
        fila->size++;/*1 atribuição*/

    return 1; /*1 retorno*/
};
```

São 7 operações, independentes da entrada.

$T(n) = 7$

Portanto a complexidade é da ordem $O(1)$, por ser constante.

3. Função fila_pop(Fila* fila)

```
/*Funcao para acessar ou retirar elemento da fila*/
/*No caso da fila, os elementos sao retirados no
   inicio do vetor*/
int fila_pop(Fila*fila){

    int popped = fila->vetor[fila->begin]; /*1 atribuição*/

    fila->begin = (fila->begin + 1) % fila->dim; /*1 atribuição*/
    fila->size--; /*1 atribuição*/

    return popped; /*1 retorno*/
};
```

São 4 operações, independente da entrada.

$T(n) = 4$

Portanto a complexidade é de ordem $O(1)$

4. Função fila_vazia(Fila* fila)

```
/*Funcao checa se a fila eh vazia*/
int fila_vazia(Fila *fila){
    return (fila->size == 0); /*1 retorno*/
};
```

É feita apenas uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$

5. Função libera_fila(Fila* fila)

```
/*Funcao que exclui a fila completamente*/
void libera_fila(Fila *fila){
    free(fila->vetor); /*1 operação*/
    free(fila); /*1 operação*/
};
```

São 2 operações, independente do tamanho da entrada.

$T(n) = 2$

Portanto, complexidade de ordem $O(1)$

(e) Deduzir a complexidade, usando notação O (o-grande)para os itens das funções implementadas na questão 1 item (e)

Resposta:

1. Função cria_fila()

```
/*Cria fila vazia*/
Fila* cria_fila(){

    /*Aloca-se memoria mas nenhum valor eh inserido*/
    Fila* fila = (Fila*) malloc (sizeof(Fila)); /*1 operação*/

    fila->fim = NULL; /*1 atribuição*/

    fila->inicio = NULL; /*1 atribuição*/

    fila->size = 0; /*1 atribuição*/

};
```

São 4 operações, independentes de tamanho de entrada (não possui).

$T(n) = 4$

Portanto, a complexidade é de ordem $O(1)$

2. Função fila_push(Fila* fila, int elemento)

```
/*insere elemento no final da fila*/
int fila_push(Fila *fila, int elemento){
    /*Criamos um noh para percorrer a fila*/
    No* aux = (No*) malloc (sizeof(No)); /*1 operação*/

    aux->prox = NULL; /*1 atribuição*/
    aux->valor = elemento; /*1 atribuição*/

    if(fila->fim == NULL){ /*1 comparação*/
        fila->inicio = aux; /*1 atribuição*/

    }else{ /*1 comparação*/
        fila->fim->prox = aux; /*1 atribuição*/
    }

    fila->fim = aux; /*1 atribuição*/

    fila->size++; /*1 atribuição*/
};
```

São 8 operações, independentes de tamanho de entrada.

$T(n) = 8$

Portanto, a complexidade é de ordem $O(1)$

3. Função fila_pop(Fila* fila)

```
/*Retira e retorna elemento do inicio da fila*/
int fila_pop(Fila *fila){
    /*Noh auxiliar para percorrer lista*/
    No* aux = fila->inicio; /*1 atribuição*/
    int popped;

    popped = aux->valor; /*1 atribuição*/
    fila->inicio = aux->prox; /*1 atribuição*/

    free(aux); /*1 operação*/
    if (!fila->inicio){ /*1 comparação*/
        fila->fim = NULL; /*1 atribuição*/
```

```
    }

    fila->size--; /*1 atribuição*/
    return popped; /*1 retorno*/
};
```

São 8 operações, independentes de tamanho da entrada.

$T(n) = 8$

Portanto, a complexidade é de ordem $O(1)$

4. Função fila_vazia(Fila* fila)

```
/*Checa se a fila eh vazia*/
int fila_vazia(Fila *fila){
    return (fila->inicio == NULL); /*1 retorno*/
};
```

É feita apenas uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$.

5. Função libera_fila(Fila* fila)

```
/*Exclui a fila completamente*/
void libera_fila(Fila *fila){
    /*Devemos percorrer a lista, apagando elemento
    por elemento*/
    int i;
    No* aux1 = fila->inicio; /*1 atribuição*/

    while (aux1) /*n iterações*/
    {
        /* code */
        No* aux2;
        aux2 = fila->inicio->prox; /*n atribuições*/
        free(aux1); /*n operações*/

        aux1 = aux2; /*n atribuições*/
```

```
    }

    free(fila); /*1 operação*/
};
```

São $4n + 2$ operações, ou seja, $T(n) = 4n + 2$. Portanto, a complexidade é de ordem $O(n)$

6. Função fila_size(Fila* fila)

```
int fila_size(Fila* fila){
    return fila->size; /*1 retorno*/
}
```

É feita apenas uma operação.

$T(n) = 1$

Portanto, complexidade de ordem $O(1)$.

7. Função print_fila(Fila* fila)

```
void print_fila(Fila* fila){
    No* aux = fila->inicio; /*1 atribuição*/
    while (aux != NULL){ /*n iterações*/
        printf("Componente da fila: %d\n", aux->valor);/*n operações*/

        aux = aux->prox; /*n atribuições*/
    }
}
```

São $3n + 1$ operações, ou seja, $T(n) = 3n + 1$. Portanto, a complexidade é da ordem de $O(n)$.