



UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Aluno: **Rosivaldo Lucas da Silva**

Matrícula: **20190028170**

Disciplina: **Estruturas de Dados**

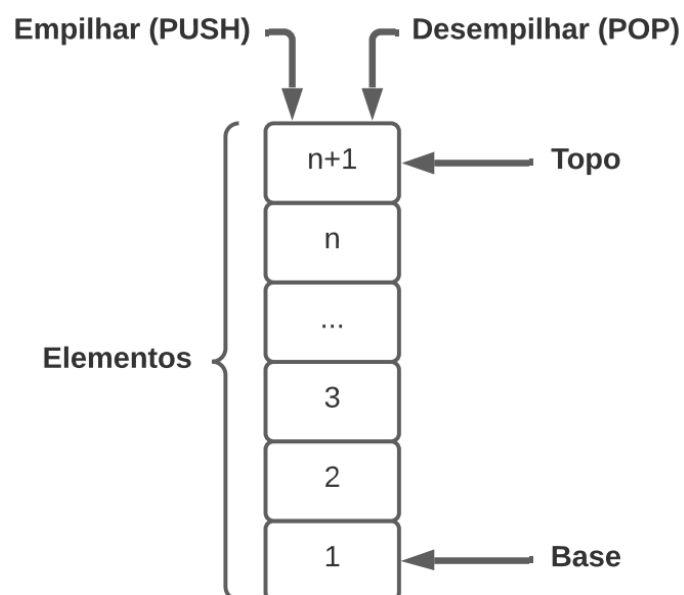
Semestre: **2021.1**

Exercício de Fixação e Aprendizagem II

QUESTÃO 1

A. TAD Pilha:

O TAD Pilha é uma estrutura de dados que tem como característica a inserção e remoção de elementos que seguem o modelo LIFO (Last In, First Out), onde o primeiro elemento a ser empilhado é o último a ser desempilhado e o último elemento a ser empilhado será o primeiro a ser desempilhado.



A estrutura Pilha pode ser implementada usando diferentes abordagens, desde a utilização de arrays estáticos ou dinâmicos e também utilizando o

conceito de listas encadeadas. A interface do TAD Pilha é composta por uma série de funções que independem da implementação escolhida e as principais funções implementadas para a manipulação de uma pilha são:

- Criar uma pilha vazia.
- Inserir um elemento no topo (push).
- Remover o elemento do topo (pop).
- Verificar se a pilha está vazia.
- Verificar o tamanho da pilha.
- Liberar estrutura pilha.

As operações de empilhar e desempilhar em uma estrutura pilha que utiliza a implementação de lista simplesmente encadeada são muito simples e fáceis de serem implementadas. Na operação de empilhar (push) todo novo elemento a ser inserido deverá ser adicionado no topo da pilha, o processo é simples de ser implementado pois na estrutura pilha é criado um ponteiro que contém sempre a referência para o topo, assim bastando apenas fazer com que o topo passe a apontar para o novo elemento criado e também devemos fazer com que o elemento anterior ao novo topo aponte para ele de modo a garantir a sequência de elementos. Na operação de desempilhar (pop) é necessário fazer com que o elemento anterior ao topo atual passe a apontar para o ponteiro NULL e fazer com que o topo passe a apontar para o elemento antecessor ao antigo topo. Essas operações sempre serão realizadas manipulando o topo da pilha, mantendo a sua característica LIFO.

B. Implementação do TAD Pilha utilizando lista encadeada:

- Interface com as funções implementadas para o tipo TAD Pilha

```
#ifndef _PILHA_LISTA_H
#define _PILHA_LISTA_H

typedef struct pilha pilha_t;

pilha_t *p_cria();
int p_libera(pilha_t *p);
int p_empilha(pilha_t *p, char elem);
int p_desempilha(pilha_t *p, char *elem);
int p_tamanho(pilha_t *p);
int p_vazia(pilha_t *p);

#endif
```

- Declaração dos pacotes utilizados e estruturas criadas

```
#include <stdlib.h>
#include <string.h>
#include "pilha_lista.h"

typedef struct no No;
struct no {
    char elem;
    No *prox;
};

struct pilha {
    No *topo;
    int tam;
};
```

- Função que aloca e cria uma estrutura do tipo pilha_t

```
pilha_t *p_cria() {
    pilha_t *p = (pilha_t *) malloc(sizeof(pilha_t));

    if (p == NULL) return NULL;

    p->topo = NULL;
    p->tam = 0;

    return p;
}
```

- Função que libera os elementos e a estrutura pilha

```
int p_libera(pilha_t *p) {
    if (p == NULL) return 0;

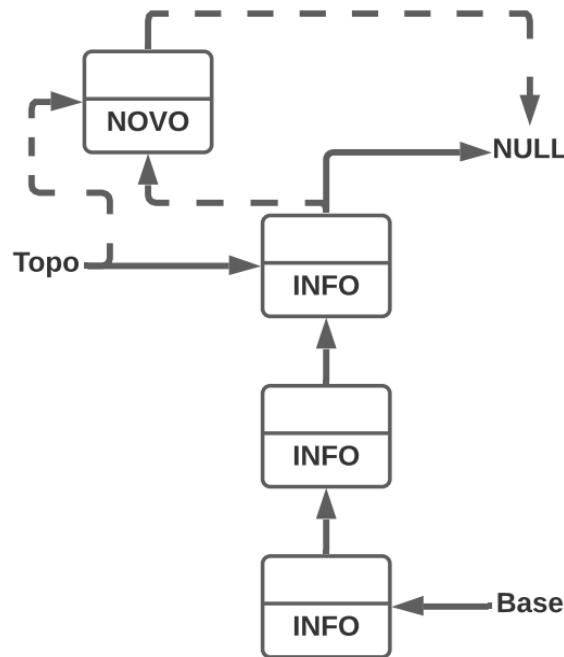
    No *aux = p->topo;
    while (aux != NULL) {
        No *aux2 = aux->prox;
        free(aux);
        aux = aux2;
    }

    free(p);

    return 1;
}
```

- **Função que recebe a pilha e o elemento a ser empilhado**

Operação realizada manipulando o topo da pilha, fazendo com que o antigo topo passe a apontar para o novo elemento a ser empilhado e fazendo com que o novo elemento aponte para o ponteiro NULL e por fim é realizado a atualização do ponteiro topo que passará a apontar para o novo elemento.



Operação de empilhar (PUSH) um novo elemento em uma Pilha que utiliza Lista Simplesmente Encadeada

```

int p_empilha(pilha_t *p, char elem) {
    if (p == NULL) return 0;

    No *novo = (No *) malloc(sizeof(No));
    if (novo == NULL) return 0;

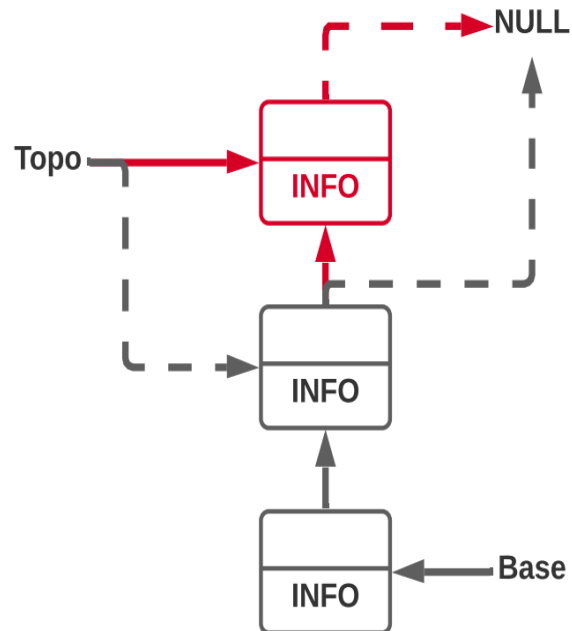
    novo->elem = elem;
    novo->prox = p->topo;

    p->topo = novo;
    p->tam += 1;
    return 1;
}

```

- **Função que recebe a pilha e o elemento a ser desempilhado**

Operação realizada manipulando o topo da pilha. O topo passa a apontar para o elemento antecessor ao antigo topo, o novo topo aponta para NULL e o antigo topo é desalocado da memória.



Operação de desempilhar (POP) um elemento em uma Pilha que utiliza Lista Simplesmente Encadeada

```
int p_desempilha(pilha_t *p, char *elem) {
    if (p == NULL || p_vazia(p)) return 0;

    No *t = p->topo;
    *elem = t->elem;

    p->topo = t->prox;
    p->tam -= 1;
    free(t);

    return 1;
}
```

- **Função que recebe a pilha e retorna o seu tamanho**

```
int p_tamanho(pilha_t *p) {
    if (p == NULL) return -1;

    return p->tam;
}
```

- Função que recebe a pilha e retorna se está vazia ou não

```
int p_vazia(pilha_t *p) {
    if (p == NULL) return 0;

    if (p->tam == 0) return 1;

    return 0;
}
```

C. Utilizando o TAD Pilha:

Para testar o funcionamento do TAD Pilha é implementado uma função que verifica o balanceamento de expressões do tipo: "[{ }(){}]", "[{([])}]", "[{ }()]". A função utiliza a propriedade de inserção e remoção do TAD Pilha para implementar a tarefa de verificação do balanceamento de expressões. A função recebe como parâmetro a expressão e itera por cada elemento realizando operações de empilhar quando é encontrado o símbolo de "{", "[" ou "(" e é realizada a operação de desempilhar quando é encontrado o oposto dos símbolos, posteriormente é realizada a comparação do símbolo desempilhado com o símbolo corrente da iteração e se os símbolos forem diferentes é adicionado o valor zero na flag **balan** que começa com o valor 1, considerando que a expressão já está balanceada.

```
int expressao_ta_balanceada(char *ex) {
    int i = 0;
    int balan = 1;

    char elem;

    pilha_t *p = p_cria();

    while (ex[i] != '\0' && balan != 0) {
        if (ex[i] == '[' || ex[i] == '(' || ex[i] == '{') {
            p_empilha(p, ex[i]);
        } else {
            if (p_vazia(p)) {
                balanceada = 0;
            } else {
                p_desempilha(p, &elem);

                if (ex[i] == ']' && elem != '[') balan = 0;
                if (ex[i] == ')' && elem != '(') balan = 0;
                if (ex[i] == '}' && elem != '{') balan = 0;
            }
        }
        i++;
    }
    return balan;
}
```

```

    }

    i++;
}

if (!p_vazia(p)) {
    balan = 0;
}

p_libera(p);

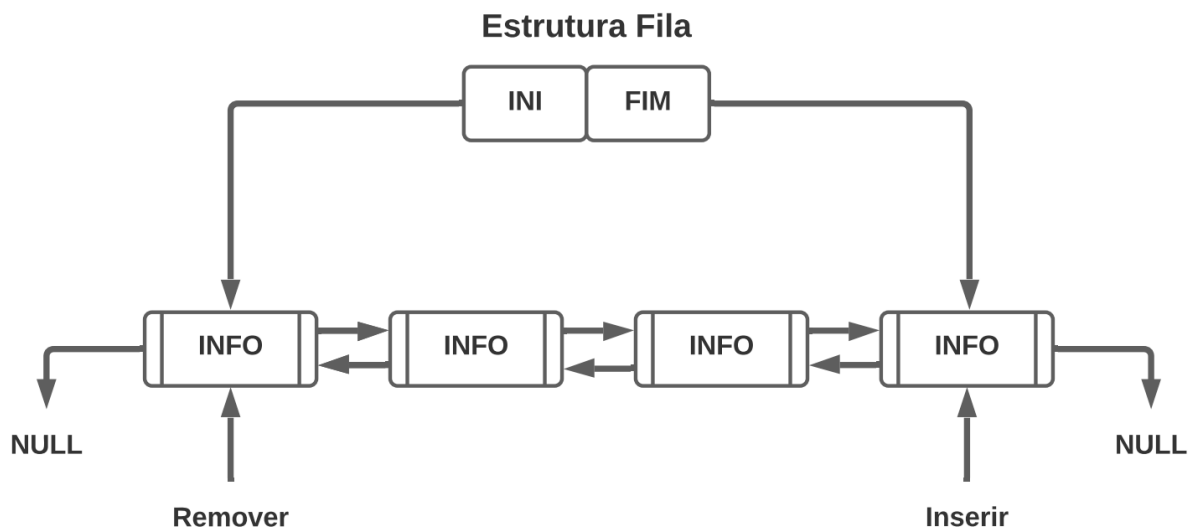
return balan;
}

```

QUESTÃO 2

A. TAD Fila:

O TAD Fila é uma estrutura de dados que tem como característica a inserção e remoção de elementos que seguem o modelo FIFO (First In, First Out), onde o primeiro elemento a ser inserido é o primeiro a ser removido e o último elemento a ser inserido será o último a ser removido.



Fila que utiliza uma Lista Duplamente Encadeada

Semelhante ao TAD Pilha, o TAD Fila pode ser implementado de diferentes formas como utilizando arrays estáticos ou dinâmicos e também utilizando o conceito de listas encadeadas. Sua interface também independe da abordagem de implementação escolhida, as principais funções que são implementadas para a manipulação do TAD Fila são:

- Criar uma fila vazia.
- Inserir um elemento no fim.
- Remover o elemento no início.
- Verificar se a fila está vazia.
- Verificar o tamanho da fila.
- Liberar estrutura fila

As operações de inserção e remoção em uma fila que utiliza listas encadeadas é mais simples, pois a inserção consiste em fazer com que o último elemento aponte para o novo elemento criado. A operação de remoção também é simples, pois consiste em remover o primeiro elemento e fazer com que o início da fila receba o sucesso do elemento retirado.

B. Implementação do TAD Fila utilizando lista duplamente encadeada:

- Interface com as funções implementadas para o tipo TAD Fila

```
#ifndef _FILA_LISTA_DUPLA_H
#define _FILA_LISTA_DUPLA_H

typedef struct fila fila_t;

fila_t *f_cria(void);
int f_libera(fila_t *f);
int f_insere(fila_t *f, char elem);
int f_remove(fila_t *f, char *elem);
int f_vazia(fila_t *f);
int f_tamanho(fila_t *f);

#endif
```

- Declaração dos pacotes utilizados e estruturas criadas

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

typedef struct no No;
struct no {
    char elem;
    No *ant, *prox;
};

struct fila {
    No *ini, *fim;
```



```
int tam;
};
```

- Função que aloca e cria uma estrutura do tipo fila_t

```
fila_t *f_cria(void) {
    fila_t *f = (fila_t *) malloc(sizeof(fila_t));

    if (f == NULL) return NULL;

    f->ini = NULL;
    f->fim = NULL;
    f->tam = 0;

    return f;
}
```

- Função que libera os elementos e a estrutura fila

```
int f_libera(fila_t *f) {
    if (f == NULL) return 0;

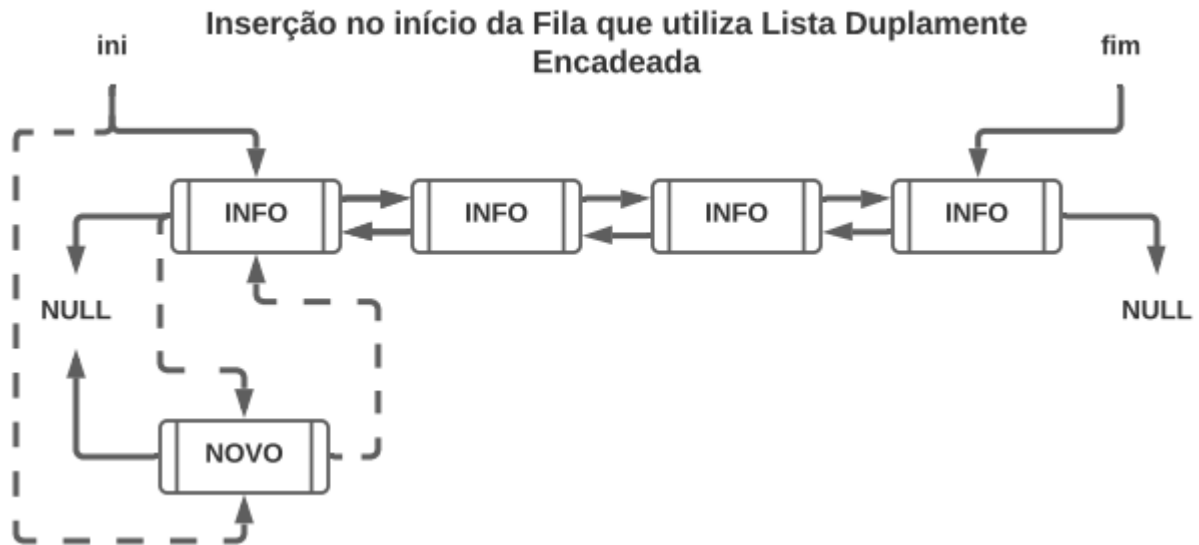
    No *p = f->ini;
    while (p != NULL) {
        f->ini = p->prox;
        free(p);
        p = f->ini;
    }

    free(f);

    return 1;
}
```

- Função que recebe a fila e o elemento a ser inserido

Para realizar a inserção de um novo elemento numa fila que utiliza uma lista duplamente encadeada é necessário fazer com que o ponteiro **ant** do primeiro elemento passe a apontar para o novo elemento e fazer com que o ponteiro **prox** do novo elemento passa a apontar para o elemento sucesso do antigo primeiro elemento e por fim fazer com que o ponteiro **ant** do novo elemento aponte para **NULL** e o ponteiro que apontava para o antigo primeiro elemento passe a apontar para o novo elemento criado.



```
int f_inserir(fila_t *f, char elem) {
    if (f == NULL) return 0;

    No *novo = (No *) malloc(sizeof(No));

    if (novo == NULL) return 0;

    novo->elem = elem;
    novo->ant = NULL;
    novo->prox = NULL;

    if (f_vazia(f) == 1) {
        f->ini = novo;
        f->fim = novo;
    } else {
        novo->ant = f->fim;
        f->fim->prox = novo;
        f->fim = novo;
    }

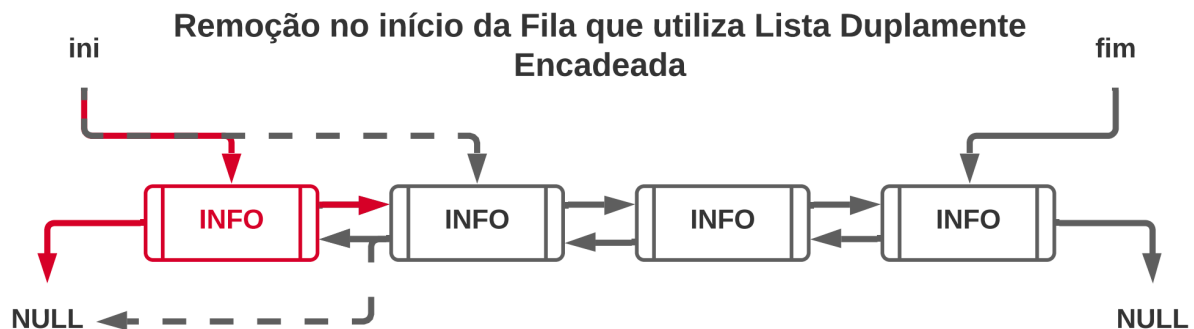
    f->tam += 1;

    return 1;
}
```

- **Função que recebe a fila e o elemento a ser removido**

Na remoção a fila tem como característica de que seja removido o primeiro elemento que foi inserido em sua estrutura, para realizar a implementação dessa funcionalidade em uma fila que é implementada utilizando lista

duplamente encadeada é necessário fazer com que o ponteiro ini aponte para o sucesso do antigo primeiro elemento e retornar a sua informação e posteriormente realizar a liberação do mesmo na memória e por fim atualizar o ponteiro ant do novo primeiro elemento para NULL.



```
int f_remove(fila_t *f, char *elem) {
    if (f == NULL || f_vazia(f)) return 0;

    No *aux = f->ini;
    *elem = aux->elem;

    if (f->ini == f->fim) {
        f->ini = NULL;
        f->fim = NULL;

        free(aux);
    } else {
        f->ini = aux->prox;
        f->ini->ant = NULL;

        free(aux);
    }

    f->tam -= 1;

    return 1;
}
```

- Função que recebe a fila e retorna se está vazia ou não

```
int f_vazia(fila_t *f) {
    if (f == NULL) return -1;

    if (f->tam != 0) {
        return 0;
    }
}
```

```
    return 1;
}
```

- Função que recebe a fila e retorna o seu tamanho

```
int f_tamanho(fila_t *f) {
    if (f == NULL) return -1;

    return f->tam;
}
```

C. Utilizando os TADs Pilha e Fila implementados:

Implementação de um programa que dado uma cadeia é realizado a verificação se a mesma é palíndroma ou não.

- Pacotes utilizados

```
#include <stdio.h>
#include <string.h>
#include "fila_lista_dupla.h"
#include "../Q1/pilha_lista.h"
```

- Função que realiza a lógica para determinar se a cadeia é palíndroma ou não

A função recebe uma palavra e realiza a iteração sobre cada letra encontrada e as adiciona em uma pilha e fila, depois do processo de invasão das letras da palavra na pilha e fila é realizado o processo de retirada dos elementos das estruturas, como a pilha tem seus elementos retirados na ordem inversa do que são adicionadas é verificado se os elementos retirados da pilha na ordem inversa são iguais aos elementos retirados da fila, se os elementos forem diferentes a palavra passada não é palíndromo e a flag **palíndromo** que começa com 1 é atualizada para 0, assim informando que a palavra não é palíndromo.

```
int eh_palindromo(char *palavra) {
    int i = 0, palindromo = 1;
    char pal_p[21], pal_f[21];

    pilha_t *p = p_cria();
    fila_t *f = f_cria();

    while (palavra[i] != '\0') {
        p_empilha(p, palavra[i]);
        f_insere(f, palavra[i]);
        i++;
    }
```

```

}

i = 0;
while (!p_vazia(p) && !f_vazia(f)) {
    p_desempilha(p, &pal_p[i]);
    f_remove(f, &pal_f[i]);

    if (pal_p[i] != pal_f[i]) {
        palindromo = 0;
    }

    i++;
}

if (strcmp(palavra, pal_p) != 0 || strcmp(palavra, pal_f) != 0) {
    palindromo = 0;
}

return palindromo;
}

```

- Função main que solicita uma palavra ao usuário e realiza a verificação se ela é palíndroma ou não

```

int main(void) {
    char p[21];

    printf("palavra: ");
    scanf("%s", p);

    if (eh_palindromo(p) == 1) {
        printf("eh palindromo\n");
    } else {
        printf("nao eh palindromo\n");
    }

    return 0;
}

```

QUESTÃO 3

A. Função que recebe um vetor de inteiros e o seu tamanho e retorna o maior valor encontrado no array

Aqui vamos considerar que atribuições e verificações tem custo 1

```
int retorna_maior_valor(int array[], int tam) {  
    int i;  
    int maior = array[0];           1  
  
    for (i = 0; i < tam; i++) {      2n  
        if (maior < array[i]) {      n  
            maior = array[i];        n  
        }  
    }  
  
    return maior;                    1  
}
```

$$f(n) = 1 + 2n + n + n + 1$$

$$f(n) = 2n + 2 + 2n$$

$$f(n) = 4n + 2$$

B. Equação do número de passos em função do tamanho do vetor

$$f(n) = 1 + 2n + n + n + 1$$

$$f(n) = 2n + 2 + 2n$$

$$f(n) = 4n + 2$$

C. Complexidade da função deduzida no passo anterior

Descartando as constantes 4 e 2 na função encontrada para o algoritmo do cálculo do maior valor do vetor de n elementos, podemos inferir que

$f(n)$ é $O(n)$ e também $f(n)$ é $\Omega(n)$, logo podemos dizer que $f(n)$ estar entre as duas funções e também $f(n)$ é $\Theta(n)$.

- Verificando se $f(n)$ é $O(n)$

Na função $f(n)$ o número 4 e 2 são constantes e podem ser desconsiderados ficando apenas o $f(n) = n$, assim confirmamos de que a função $f(n)$ é $O(n)$

- Verificando se $f(n)$ é $\Omega(n)$

Aqui vamos demonstrar que existe uma constante c e um valor positivo n_0 inicial de forma que para todos o valor n maiores ou iguais a n_0 ,

$$4n + 2 \geq cn.$$

$$4n + 2 \geq cn$$

$$4n - cn \geq -2$$

$$cn - 4n \geq 2$$

$$n(c - 4) \geq 2$$

$$n \geq \frac{2}{(c-4)}$$

Esta expressão é válida para $c = 5$ e $n_0 = 1$

Logo a função $f(n)$ é $\Omega(n)$

- Verificando se $f(n)$ é $\Theta(n)$

Para uma função ser $\Theta(n)$ ela deve ser $\Omega(n)$ e $O(n)$ e foi mostrado nos passos anteriores que a função $f(n)$ satisfaz essas duas condições, logo a função $f(n)$ é $\Theta(n)$

QUESTÃO 4

- O algoritmo de Euclides é um clássico caso onde se pode utilizar a recursão para se chegar no resultado do problema do cálculo do mínimo divisor comum (mdc). Para a implementação do algoritmo serão considerados dois números inteiros, mas o algoritmo pode ser adaptado para encontrar o mdc de dois ou mais números.

- **Funcionamento do algoritmo implementado**

A função que irá calcular o mdc recebe dois números inteiros a e b como parâmetros e realiza as verificações:

- I. Caso $b = 0$, o valor de a é retornado. Essa é a condição de parada da função. Todo algoritmo que for implementar recursão deve ter uma condição de para não entrar em loop infinito.
- II. Caso $b > 0$, a função é chamada novamente, agora passando o valor de b como primeiro parâmetro e o resto da divisão dos valores de $a \% b$, e esse processo será realizado até que o parâmetro b seja igual a zero.
- III. Caso $b < 0$, a função é chamada novamente, mas passando como parâmetros os valores de a e $-b$, assim quando a função for executada com estes valores o b passará a ser positivo e irá ser executado o caso II até que o valor de b seja zero.

- **Cálculo do mdc utilizando o algoritmo de Euclides**

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

```
int mdc_recursivo(int a, int b) {  
    if (b == 0) {  
        return a;  
    } else if (b > 0) {  
        int mod = a % b;  
        return mdc_recursivo(b, mod);  
    } else {  
        return mdc_recursivo(a, -b);  
    }  
}  
  
int main(void) {  
  
    int mdc = mdc_recursivo(726, -275);  
  
    printf("mdc = %d\n", mdc);  
  
    return 0;  
}
```