



UNIVERSIDADE FEDERAL DO CARIRI

Marcos Pereira da Silva

Osvaldo Soares Landim Júnior

Orientador: Roberto Hugo Wanderley Pinheiro

INTRODUÇÃO A ALGORITMOS E ESTRUTURAS DE DADOS

Índice

1. INTRODUÇÃO	4
2. PONTEIRO	5
2.1. Organização na memória	5
2.2. Conceituação geral de ponteiros	6
2.3. Aritmética de ponteiros	9
Exercícios	11
3. ALOCAÇÃO DE MEMÓRIA	16
3.1. Visão geral	16
3.2. Tipos de alocação	16
3.3. Alocação Estática vs Dinâmica	16
3.4. Alocando memória	18
3.5. Validando alocação	20
3.6. Liberando memória	21
Exercícios	22
4. REGISTROS	27
4.1. Anatomia de uma struct	27
4.2. Acessando campos dos registros	28
4.3. Alias de variáveis do tipo struct (usando o typedef)	33
Exercícios	35
5. INTRODUÇÃO (AED)	41
5.1. Tad	41
5.2. Fila - Explicação e Algoritmo	43
5.2.1. Operações com fila:	43
5.2.2. Inserção em uma fila:	46
5.2.3. Remover em uma fila:	49
5.3. Pilha - Explicação e Algoritmo	51
5.3.1. Operações com pilha	51
5.3.2. Inserção	54
5.3.3. Remoção	56
5.4. Lista - Explicação e Algoritmo	58
5.4.1. Criando a lista	58
5.4.2. Liberar lista	60
5.4.3. Inserção	60
5.4.4. Remoção	62
6. Noções de complexidade	65
6.1. Constante vs Linear	65
6.2. Complexidade da fila	68

6.2.1. Dinâmica	68
6.2.2. Estática	68
6.3. Complexidade da pilha	68
6.3.1. Dinâmica	68
6.3.2. Estática	68
6.4. Complexidade da lista	69
7. MODULARIZAÇÃO E BOAS PRÁTICAS DE PROGRAMAÇÃO	70
7.1. Visão geral	70
7.2. Demonstração	70

1. INTRODUÇÃO

Essa apostila é destinada aos estudantes do curso de Ciência da Computação na Universidade Federal do Cariri (UFCA), com o intuito de nivelar e melhorar a base de programação dos alunos que estão saindo da cadeira de Introdução à Programação e entrando na de Algoritmos e Estruturas de Dados 1 (AED1). Outro ponto é a diminuição do índice de reprovação da disciplina e facilitar a absorção do conteúdo de AED1 pelos alunos.

É de suma importância que o leitor se motive a realizar os exercícios propostos nesta apostila, porque, a parte prática de programação é o pilar não só da cadeira de Estruturas de Dados, mas de todo o processo de aprendizado no eixo relacionado a programação dentro do curso de Ciência da Computação. A apostila seguirá, assim como as cadeiras que estão ao redor dela, com a linguagem de programação **C**, e o fluxo do conteúdo se estende a partir do conceito e base sobre ponteiros, registros, passando pelos tópicos de lista, fila e pilha, até o tópico de boas práticas no desenvolvimento dos nossos algoritmos, para uma melhor compreensão, reuso e manutenção de código. É digno de nota que será assumido um domínio básico da linguagem C, de modo que alguns tópicos serão passados com mais velocidade. É importante seguir o fluxo da estrutura da apostila, porque há uma relação de complemento entre os tópicos, tornando o anterior um requisito para o próximo.

Um outro ponto importante é que a coleção de perguntas e respostas dos exercícios estará disponível no [github](https://github.com/marcosChalet/codigos-apostila)¹ para agilizar o acompanhamento da apostila e sanar dúvidas sobre a resolução de questões.

¹ <https://github.com/marcosChalet/codigos-apostila>

2. PONTEIRO

2.1. Organização na memória

Um dos recursos mais importantes e usados da linguagem C são os ponteiros, dado que ele pode ser utilizado com alocação dinâmica, tópico que será visto mais adiante, o seu uso possibilita uma grande arma para seus algoritmos. Por outro lado, é muito comum ouvir, de iniciantes em programação, reclamações sobre o quão complexo é o uso de ponteiros. Por conta disto, antes de entrar de fato no tópico, devemos ter em mente sempre a memória principal quando estamos usando ponteiros em nossos programas.

Seguindo, é necessário relembrar que quando executamos um programa, ele vai para a memória RAM de modo que suas variáveis e declarações ficam, também, por fazerem parte do programa, ou seja, todos armazenados na memória principal. Para facilitar o entendimento, analise a representação da memória principal na imagem-1. Em alguma posição da memória teremos o programa “**meu_programa.c**” com suas variáveis alocadas, no entanto, apesar da representação, sua forma de endereçamento não é necessariamente em um bloco contíguo de células.



imagem-1

Elaborando melhor o exemplo anterior para que fique mais claro o comportamento da memória, observe, na imagem imagem-2, que acima de cada endereço/célula temos o valor da variável, e em especial temos o ponteiro “**b**” apontando para o endereço do primeiro elemento do vetor “**dados**”, possibilitando que obtenha acesso aos elementos do vetor a partir do ponteiro “**b**”.



imagem-2

2.2. Conceituação geral de ponteiros

Agora entrando de fato em ponteiros, um ponteiro é uma variável como as que já temos costume de usar (int, char, float...), porém o que torna um ponteiro diferente delas é que ao invés de ter um valor diretamente atribuído, o que a ele é de fato adicionado são endereços de memória. Ou seja, para atribuir uma variável “normal” a uma do tipo ponteiro, utilizamos, por exemplo, o símbolo “&” antes da variável para que seja atribuído o seu endereço. Recapitulando, quando fazemos o trecho de código “**int *b = &a;**”, estamos especificando para a variável “**b**”, que se torna um ponteiro para uma variável do tipo **int** por ter sido usado o modificador (*) em sua declaração, que ela deve referenciar o endereço de memória da variável “**a**”, de tal modo que, já que o “conteúdo” do ponteiro está na variável “**a**”, as alterações em “**a**” irão impactar o ponteiro. Por outro lado, quando fazemos “**int b = a**” o que de fato está acontecendo é a atribuição como uma cópia do valor de “**a**” à “**b**”, de tal modo que quando alterado o valor de “**a**” o valor de “**b**” não sofrerá mudança. Veja o exemplo.

```
#include <stdio.h>

int main () {

    int a = 10;
    int b = a;

    /* Nesse momento o valor de b é igual ao valor de a */

    a = 20;

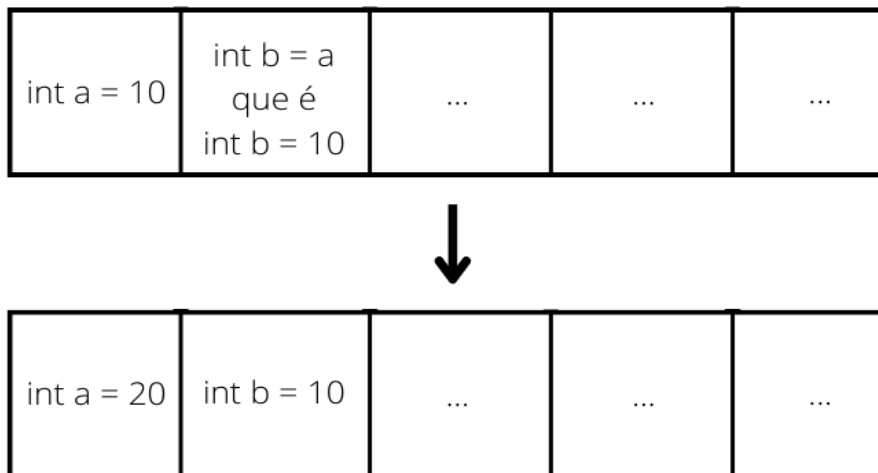
    /* Como b não é um ponteiro, a fica com o valor 20
     * mas b continua com 10
     * */

    printf("a = %d, b = %d\n", a, b); // a = 20 e b = 10

    return 0;
}
```

No primeiro caso temos um exemplo de código em que há a alteração de 10 para 20 no valor da variável “**a**”, porém como a atribuição de “**a**” para “**b**” não é utilizado ponteiro, não haverá erros na operação e o valor da variável “**b**” continuará sendo o mesmo na execução do **printf**.

Veja a representação.



Já no caso abaixo, o valor de “a” é 10 e o valor da variável “b” é o endereço de “a”, sendo assim, ao modificar o valor de “a” chegaremos no print final com “a” contendo o valor 20 e, ao fazer “*b” no print, iremos obter também o valor 20, já que o seu conteúdo ainda é o endereço de memória da variável “a”. Observe que o conteúdo de “b” é a variável “a” e para acessar o valor do conteúdo de “b”, usamos * antes da variável;

```
#include <stdio.h>

int main () {

    int a = 10;
    int *b = &a;

    /* Neste momento o valor de 'b' é o endereço de 'a' */

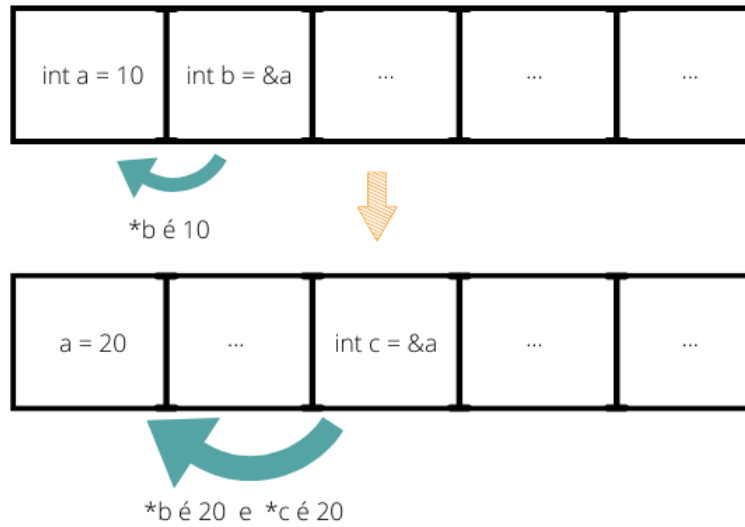
    a = 20;

    /* Agora como 'b' é um ponteiro, 'a' fica com o valor 20
     * e b continua com o endereço de memória onde 'a' está.
     * Note que para acessar o conteúdo de 'a' podemos usar
     * o ponteiro 'b'. Veja o printf.
     */

    printf("Valor de A: %d\n", *b);

    return 0;
}
```

Veja a abstração.



Faça testes com ponteiros do tipo `char`, `float`, e até mesmo ponteiro para outros ponteiros, para fixar melhor a forma de os manipular.

2.3. Aritmética de ponteiros

Outro ponto importante que deve ser lembrado é a aritmética de ponteiros, que em alguns momentos pode nos ser conveniente dentro de nossos algoritmos, seja para diminuir a verbosidade do código ou para otimizar um acesso. Ocorre de nos depararmos com trechos de código com o operador de auto-incremento (**++**), auto-decremento (**--**), de atribuição com soma (simplificado por "**+= 1**") ou o de subtração com soma (simplificado por "**-= 1**") dentro de algoritmos, seja em laços de repetição, chamadas de funções ou no corpo de uma função, com o intuito de adicionar uma unidade na variável que está sofrendo a operação. Estas operações também se aplicam aos ponteiros, porém de uma forma diferente. veja:

```
#include <stdio.h>

int main () {

    int arr[] = {1,3,5};
    int *ptArr;

    ptArr = arr; // ou ptArr = &(arr[0])

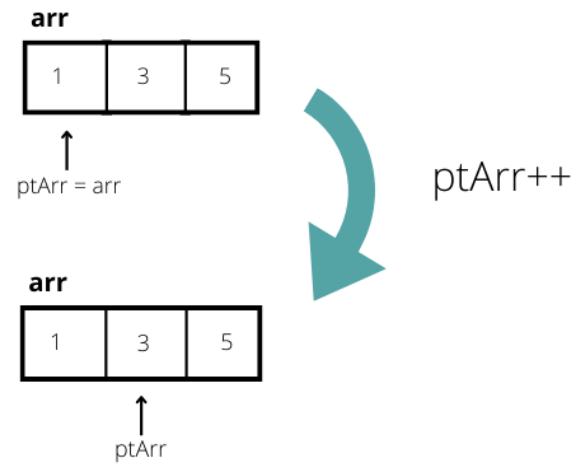
    ptArr++;

    printf("%d\n", *ptArr);

    return 0;
}
```

Analisando o código acima vemos que existe um vetor com três elementos e um ponteiro apontando para esse vetor, após isso o ponteiro para o vetor é incrementado com o operador de auto-incremento, e por fim mostra o conteúdo de quem o ponteiro está apontando. Neste ponto você pode tomar o equívoco de achar que o incremento se dá no valor do primeiro elemento do vetor, o número 1, e o print mostrar na tela o valor 2 como resultado, por outro lado, sendo um pouco mais atento, como um ponteiro aponta para endereço de memória você também pode achar que o incremento se dá em uma unidade no endereço de "**arr**". Destas duas linhas de pensamento, a segunda é a mais assertiva, porém, o incremento é feito com a quantidade de bytes do tipo da variável apontada. Ou seja, como nesse caso temos um vetor de inteiros e o ponteiro "**ptArr**" está apontando para uma variável do tipo **int** (o primeiro elemento do vetor), o deslocamento, na linguagem C, é de 4 bytes, fazendo com que "**ptArr**" assuma o endereço do segundo elemento do vetor e mostrando na tela o valor 3. Note que esse processo tem seu incremento ou decremento de forma equivalente para os outros tipos de variáveis, assumindo a quantidade de bytes do seu tipo.

Para melhor exemplificar o código anterior, observe a abstração do procedimento de incremento no ponteiro.



Exercícios

1. Utilizando um ponteiro o operador de auto-incremento e um laço de repetição: faça o valor da variável 'a' ser incrementado +1 por 5 vezes. Lembre-se da precedência dos operadores.

```
#include <stdio.h>

int main () {

    int a = 1;

    /*
     * Procedimento.
     * */

    printf("%d\n", a);

    return 0;
}
```

2. Utilizando um ponteiro: multiplique por 5 cada elemento do vetor, de tal forma que o laço no final obtenha a saída "5, 10, 15, 20, 25".

```
#include <stdio.h>

int main () {

    int vet[] = {1, 2, 3, 4, 5};
    int tamVet = sizeof(vet) / sizeof(vet[0]);

    /*
     * Procedimento
     * */

    for (int i = 0; i < tamVet; i++) {
        printf("%d, ", vet[i]);
    }

    return 0;
}
```

3. Resolva novamente o exemplo anterior, mas usando uma função para fazer a operação.

4. Utilizando ponteiros e dois laços de repetição: remova todos os espaços em branco da string e a ajuste de tal forma que haja uma alternância entre letras maiúsculas e minúsculas. Ex: "Uma String Qualquer" -> "UmAsTrInGqUaLqUeR"

```
#include <stdio.h>

int main () {

    char palavra[] = {"Uma String Qualquer"};

    /* Procedimento 1
     * */

    /* Procedimento 2
     * */

    printf("%s\n", palavra);

    return 0;
}
```

5. Continue a função swap de tal forma que ao passar duas posições válidas da lista de nomes “**lst**” haja a troca do conteúdo das posições no print final.

```
#include <stdio.h>

void swap( /* parâmetros */ ) {

    /*
     * Procedimento
     * */

}

int main () {

    /*
     * lst é uma lista de ponteiros, e cada ponteiro
     * aponta para um nome
     * */

    char *lst[] = {"Ana", "Carlos", "João",
                   "Vitória", "Amanda", NULL};

    /* swap( argumentos ) */

    for (int i = 0; lst[i] != NULL; i++) {
        printf("%s\n", lst[i]);
    }

    return 0;
}
```

6. Continue o exemplo abaixo criando as função de soma e subtração e mostre o resultado utilizando o ponteiro **ptOperacao** e a função **calcular**.

```
#include <stdio.h>

void calcular ( int a, int b, int (*funcCalculo)() ) {
    printf("Resultado: %d\n", funcCalculo(a, b));
}

int soma (const int a, const int b) {
    return a + b;
}
```

```

int main () {

    /* Ponteiro que aponta para uma função */
    int (*ptOperacao)();

    /* Atribuindo uma função ao ponteiro */
    ptOperacao = soma;

    /*
     * Passando os valores que sofrerão uma
     * operação matemática, e a função que
     * fará a operação
     * */
    calcular(3, 2, ptOperacao);

    /*
     * Procedimento
     */
    return 0;
}

```

7. Modifique a questão anterior para que não use o ponteiro **ptOperacao**.
8. Usando dois ponteiros, um chamado **nome** que irá apontar para cada palavra dentro de **nomesTeste** e outro chamado **letra** que apontará para **nome**. Seu objetivo é mostrar todas as strings letra a letra usando o ponteiro **letra**. Você não pode usar índices nem criar outras variáveis além dos dois ponteiros, apenas aritmética de ponteiros.

```

#include <stdio.h>

int main () {

    /* Vetor de ponteiros para strings */
    char *nomesTeste[] = {"Casa", "Carro", "Rua",
                          "Aniversário", "Mão", NULL};

    /*
     * Procedimento
     * */

    return 0;
}

```

9. Continue o exemplo usando ponteiros.

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

void mostrar (/* ... */, /* ... */, /* ... */) {
    printf("Idade: %d\n", /* ... */);
    printf("Nome: %s\n", /* ... */);

    puts("\n===== Meses =====");
    for (char ** mes = /* ... */; *mes != NULL; mes++) {
        printf("# %-17s #\n", /* ... */);
    }
    puts("=====");
}

int main () {

    int idade = 32;
    char nome[30];
    strcpy(nome, "Changemilson");
    char *meses[] = {"Janeiro", "Junho", "Maio",
                    "Julho", "Agosto", NULL};

    mostrar(&idade, &(nome[0]), meses);

    return 0;
}
```

10. Usando a solução da questão anterior: Crie um condicional no trecho que mostra os meses para que seja mostrado apenas os que começam com a letra “J”, de tal forma que não seja utilizado os colchetes para a seleção e comparação do caractere.

3. ALOCAÇÃO DE MEMÓRIA

3.1. Visão geral

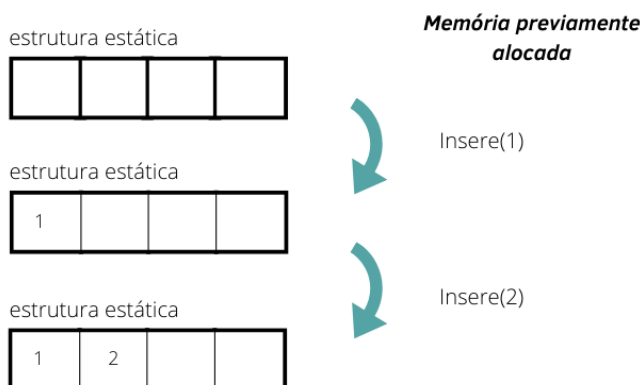
A alocação dinâmica, junto com os ponteiros, é bastante utilizada em diversos momentos quando estamos programando nossas estruturas de dados. Este tipo de alocação, na linguagem C, serve para reservar uma quantidade de memória e se dá através de algumas funções da biblioteca “**stdlib.h**” como malloc, calloc e realloc. Para esta apostila iremos concentrar esforços apenas no malloc, mas fica de incentivo para o leitor ler a documentação das outras funções para um entendimento mais aprofundado do tema.

3.2. Tipos de alocação

A alocação de memória de nossas estruturas de dados se dará de duas formas: através da alocação estática e da alocação dinâmica. Apesar da alocação estática ser bastante usada, será a alocação dinâmica o principal foco ao desenvolver os algoritmos da cadeira de Estruturas de Dados, já que ela requer mais cuidados e atenção do programador. No entanto, é muito importante ter domínio das duas formas para que faça a análise sobre em qual situação uma ou a outra deve ser utilizada.

3.3. Alocação Estática vs Dinâmica

Por um lado, ao alocar memória de forma estática, é definida uma quantidade fixa de memória que poderá ser criada em tempo de compilação, estruturada como um bloco sequencial de bytes. Esta memória fica disponível para uso enquanto a função que contém a variável alocada estiver em execução, mas quando a função sair da pilha de chamada, a memória também será perdida², levando consigo o seu conteúdo.



² A memória que foi alocada na stack não é necessariamente desalocada, mas sim abandonada como lixo de memória.

Por outro lado, a alocação dinâmica ocorre em tempo de execução e podendo ser alocada em toda área lógica da memória. É mais flexível que a alocação estática e dá liberdade ao programador para definir o seu “tempo de vida”, porém requer mais cuidado com a memória, já que ela não é desalocada quando a função é desempilhada, deixando a cargo do programador a desalocação de memória.

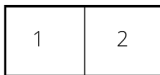
estrutura dinâmica



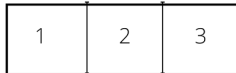
estrutura dinâmica



estrutura dinâmica



estrutura dinâmica



Insere(1)



Insere(2)




Insere(3)

***Memória alocada por
requisição***

3.4. Alocando memória

O processo de alocar memória é simples e requer o uso da biblioteca “**stdlib.h**” em nosso código. Veja a representação da sintaxe abaixo.



The diagram illustrates the syntax of the `malloc` function. It shows the function signature `void* malloc (size_t size);`. Three teal arrows point from text labels to parts of the signature: one from 'retorno da função' to `void*`, one from 'total de bytes a ser alocado' to `size`, and one from 'palavra reservada para alocação de memória' to `malloc`.

retorno da função

total de bytes a ser alocado

`void* malloc (size_t size);`

palavra reservada para alocação de memória

Observe que o retorno é um ponteiro do tipo **void**, para o bloco alocado, que será convertido de forma implícita ou explícita, através do cast, para o tipo associado a variável que irá receber esse ponteiro. Veja o exemplo:

```
#include <stdlib.h>

int main () {

    /*
     * Alocando 10 inteiros e atribuindo com
     * conversão IMPLÍCITA para vet1
     * */

    int *vet1 = malloc(sizeof(int) * 10);

    /*
     * Alocando 10 inteiros e atribuindo com
     * conversão EXPLÍCITA para vet1
     * */

    int *vet2 = (int*)malloc(sizeof(int) * 10);

    return 0;
}
```

É digno de nota que tomaremos como padrão a forma descrita com cast para desenvolver nossos algoritmos. Neste momento temos dois ponteiros que apontam para

um bloco sequencial de 10 inteiros na memória, no entanto poderia ser 10 float, char ou double caso trocássemos o tipo “**int**” por algum deles.

Agora vamos seguir preenchendo o vetor.

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

#define MAX_VET 10

int main () {

    int *vet2 = (int*)malloc(sizeof(int) * MAX_VET);

    for (int i = 0; i < MAX_VET; i++) {
        vet2[i] = i;
    }

    for (int i = 0; i < MAX_VET; i++) {
        printf("%d, ", vet2[i]);
    }

    return 0;
}
```

Veja que adicionei uma definição para determinar a quantidade de elementos (**MAX_VET**), adicionei a biblioteca “**stdio.h**” para podermos printar a saída e, como foi alocado um bloco sequencial, nós podemos usar os colchetes para acessar cada posição do vetor.

3.5. Validando alocação

Um procedimento que não deve ser esquecido é a verificação do ponteiro que está recebendo a memória alocada. Durante o processo de alocação de memória existe a possibilidade de dar algum erro nesse processo impossibilitando o uso da memória requisitada. Quando isso ocorre, a função “**malloc**” retorna “**NULL**” para que nos possibilite verificar que a alocação não deu certo, veja o exemplo abaixo.

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

#define MAX_VET 10

int main () {

    int *vet2 = (int*)malloc(sizeof(int) * MAX_VET);

    /*
     * Verificando o resultado da alocação
     * */

    if ( vet2 == NULL ) {
        printf("Erro ao tentar alocar memória!\n");
        exit(1);
    }

    for (int i = 0; i < MAX_VET; i++) {
        vet2[i] = i;
    }

    for (int i = 0; i < MAX_VET; i++) {
        printf("%d, ", vet2[i]);
    }

    return 0;
}
```

Observe que coloquei o status de saída como 1, em “**exit(1)**”, para informar ao sistema operacional que o programa não foi finalizado com sucesso.

3.6. Liberando memória

Como visto anteriormente, a memória alocada dinamicamente não é desalocada após o término da função, e deixando essa responsabilidade para o programador. Para nos auxiliar nesse processo, existe a função “**free**” que recebe um ponteiro e desaloca o bloco de memória que ele está apontando. Vamos continuar o exemplo anterior para exemplificar o uso do “**free**”.

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

#define MAX_VET 10

int main () {

    int *vet2 = (int*)malloc(sizeof(int) * MAX_VET);
    if (vet2 == NULL) {
        printf("Erro ao tentar alocar memória!\n");
        exit(1);
    }

    for (int i = 0; i < MAX_VET; i++) {
        vet2[i] = i;
    }

    for (int i = 0; i < MAX_VET; i++) {
        printf("%d, ", vet2[i]);
    }

    free(vet2);

    return 0;
}
```

Agora o bloco de 10 inteiros que foi alocado é liberado para que seja utilizado novamente.

Exercícios

1. Crie e preencha uma matriz 4x4 a partir do vetor estático de ponteiros dado.

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

#define LINHAS 4
#define COLUNAS 4

int main () {

    int *matriz[LINHAS]; // Vetor de ponteiros

    /*
     * Procedimento
     * */

    // Mostrando a matriz
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            printf("%-2d ", matriz[i][j]);
        }
        puts("");
    }

    return 0;
}
```

2. Utilizando a solução da questão anterior: Desalocar a memória da matriz criada dinamicamente.
3. Crie e preencha uma matriz com suas linhas e colunas alocadas dinamicamente.
4. Utilizando a solução da questão anterior: Desalocar completamente a matriz.
5. Crie um vetor e armazene 5 nomes (strings) alocado dinamicamente. Após isso mostre todas as strings na tela percorrendo o vetor.

6. Um ponto que programadores alertam muito na linguagem C é sobre algumas funções que não deveriam ser usadas para em determinados contextos. Uma delas era a função “**strcpy**” da biblioteca “**string.h**”. Esta função é responsável por copiar o conteúdo de uma string para outra, no entanto quando a string que será copiada consome mais espaço do que o array que receberá os novos dados é capaz de armazenar, já que não era especificado a quantidade de bytes dele, tínhamos um estouro no buffer. Dado este contexto, implemente uma função para copiar strings considerando o tamanho do array resultante. Considere que sua função recebe um ponteiro e uma string, e use a função “**realloc**” para que não haja perda de dados nem o estouro de buffer.

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

void mystrcpy (/* Parâmetros */) {

    /*
     * Procedimento
     * */
}

int main () {

    char nomePequeno[] = {"chalet"};
    char *nome = (char*)malloc(strlen(nomePequeno) + 1);
    strcpy(nome, nomePequeno);

    mystrcpy(nome, "novoNomeGrandeeeeeee");

    printf("%s\n", nome);

    return 0;
}
```

7. Utilize a função **free** para liberar a memória do vetor de nomes.

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

#define QTDNOMES 3
```

```

int main () {

    char **nomes = (char**)malloc(sizeof(char*)*QTDNOMES);
    if (nomes == NULL) exit(1);

    for (int i = 0; i < QTDNOMES; i++) {
        nomes[i] = (char*)malloc(strlen("chalet"));
        if (nomes[i] == NULL) exit(0);

        strcpy(nomes[i], "chalet");
    }

    /*
     * Procedimento
     * */

    return 0;
}

```

8. Aloque o ponteiro **vetores** para que ele seja um vetor de 10 ponteiros. Após isso, aloque e insira números em cada posição de **vetores**, com laços de repetição, de tal forma que a primeira posição vai ter os valores de 0 a 9, a segunda posição de 0 - 8, a terceira 0 - 7... Por fim, desaloque cada subvetor.

```

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

#define MAX 10

int main () {

    int ** vetores;

    /*
     * Procedimento
     * */

    /* Mostrando cada posição */
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX - i; j++) {
            printf("%-2d ", vetores[i][j]);
        }
        puts("");
    }
}

```



```

    }

    /*

    [Saída esperada]

    0  1  2  3  4  5  6  7  8  9
    0  1  2  3  4  5  6  7  8
    0  1  2  3  4  5  6  7
    0  1  2  3  4  5  6
    0  1  2  3  4  5
    0  1  2  3  4
    0  1  2  3
    0  1  2
    0  1
    0

    * */

    /*
    * Procedimento
    * */

    return 0;
}

```

9. A função **alocador** serve para alocar memória para variáveis do tipo **int***. Termine a função e aloeque memória para **vetInt** com ela para que continue o programa de tal forma que seja impresso os 10 inteiros que estão sendo preenchidos.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define TAM 10

int alocador (/* parâmetro 1 */, size_t size) {
    /*
    * Procedimento
    * */
}

```

```
int main () {  
  
    int *vetInt;  
  
    /*  
     * Procedimento  
     * */  
  
    for (int i = 0; i < TAM; i++)  
        vetInt[i] = i;  
  
    for (int i = 0; i < TAM; i++)  
        printf("%d, ", vetInt[i]);  
  
    return 0;  
}
```

10. Crie um vetor dinâmico, insira 5 strings e mostre de forma ordenada tomando como método de comparação a primeira letra. Você pode usar a função **qsort** da biblioteca **stdlib.h**. Por fim, desaloque o vetor com seus ponteiros para cada string.

4. REGISTROS

Dando sequência e agregando tudo que foi visto até aqui, daremos mais amplitude aos nossos algoritmos a partir do uso de registros, representado pela palavra reservada **struct**, em nossos códigos. Podemos conceituar “**struct**” como um tipo agregador de variáveis delimitadas pelo usuário programador, de modo que nos possibilita a criação de tipos específicos para nossa aplicação a deixando mais robusta e produtiva.

4.1. Anatomia de uma struct

Com um nome e uma sequência de declarações de variáveis dentro de um bloco iniciado com ‘{’ e finalizado com ‘};’, um struct permite criar variáveis do seu tipo, permitindo mais flexibilidade do que os tipos básicos de variáveis. Veja sua sintaxe em código:

```
struct nomeQualquer {  
    /*  
     * Sequência de declarações  
     */  
};  
  
/* Declarando uma variável do tipo nomeQualquer */  
struct nomeQualquer minhaVariavel;
```

É de extrema importância o uso do ‘;’ para delimitar o fim da definição. Esquecer isso pode dar muita dor de cabeça. Agora vamos exercitar um pouco criando um registro aluno com os campos de nome, matrícula e curso.

```
#include <stdio.h>  
  
struct aluno {  
    char *nome;  
    char *curso;  
    int matricula;  
};  
  
int main () {  
  
    /* Criando a variável para acessar seus campos */  
    struct aluno aluno;
```

```
    return 0;
}
```

Agora podemos acessar os campos nome, matricula e curso através da variável “**aluno**” criada pelo tipo “**struct aluno**”. Veja que além de podermos criar variáveis do tipo **int**, **float**, **double**... agora também podemos criar variáveis do tipo **struct aluno** especificado pela declaração do struct acima da função **main**.

4.2. Acessando campos dos registros

Para especificar qual campo queremos acessar dos nossos registros, nós utilizamos a **notação ponto**, ou operador de seleção direta, e em seguida qual campo que queremos manipular. Como exemplo iremos preencher os dados do aluno criado no tópico anterior. Observe.

```
#include <stdio.h>

struct aluno {
    char *nome;
    char *curso;
    int matricula;
};

int main () {

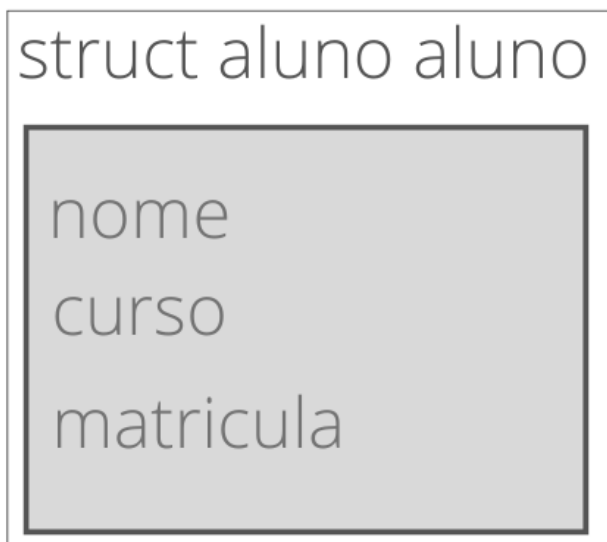
    /* Criando a variável para acessar seus campos */
    struct aluno aluno;

    char nome[] = {"chalet"};
    char curso[] = {"Ciência da Computação"};

    /* Atribuindo valores aos campos da variável aluno */
    aluno.nome = nome;
    aluno.curso = curso;
    aluno.matricula = 12345;

    return 0;
}
```

Nesse momento, pela simplicidade de manipular registros não alocados de forma dinâmica, pela clareza da notação ponto, já deve ter ficado evidente a forma de obter os dados da variável `aluno`. Veja a representação visual.



Antes de prosseguir, observe o trecho:

```
char nome[] = {"chalet"};
char curso[] = {"Ciência da Computação"};
```

Não foi adicionado um tamanho para a variável **nome** e **curso**. Isso é possível porque, como as duas variáveis estão sendo declaradas de forma estática (observe os colchetes) e o conteúdo a ser atribuído é conhecido, o seu tamanho é calculado em tempo de compilação. Trechos de código como esse poderão aparecer mais vezes durante a apostila, então fique esperto quando a isto.

Continuando, para finalizar o nosso exemplo, vamos printar os dados do aluno para ver se está tudo correto.

```
#include <stdio.h>

struct aluno {
    char *nome;
    char *curso;
    int matricula;
};

int main () {
```

```

/* Criando a variável para acessar seus campos */
struct aluno aluno;

char nome[] = {"chalet"};
char curso[] = {"Ciência da Computação"};

/* Atribuindo valores ao aluno */
aluno.nome = nome;
aluno.curso = curso;
aluno.matricula = 12345;

/* Mostrando os dados do aluno */
printf("Nome: %s\n", aluno.nome);
printf("Curso: %s\n", aluno.curso);
printf("Matrícula: %d\n", aluno.matricula);

return 0;
}

```

Melhorando o exemplo acima, vamos alocar a variável aluno dinamicamente. Para isso nós devemos criar a variável aluno como um ponteiro e utilizar o **malloc** para alocar memória, por consequência teremos que usar o (*) para especificar que queremos acessar o valor da variável aluno ao invés do endereço onde o bloco de bytes que foi alocado está.

```

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

struct aluno {
    char *nome;
    char *curso;
    int matricula;
};

int main () {

    /*
     * Alocando struct aluno bytes para a variável aluno
     * */

    struct aluno *aluno;
    aluno = (struct aluno*)malloc(sizeof(struct aluno));
    if (aluno == NULL) {
        perror("Problema na alocação da memória\n");
        exit(0);
    }
}

```

```

char nome[] = {"chalet"};
char curso[] = {"Ciência da Computação"};

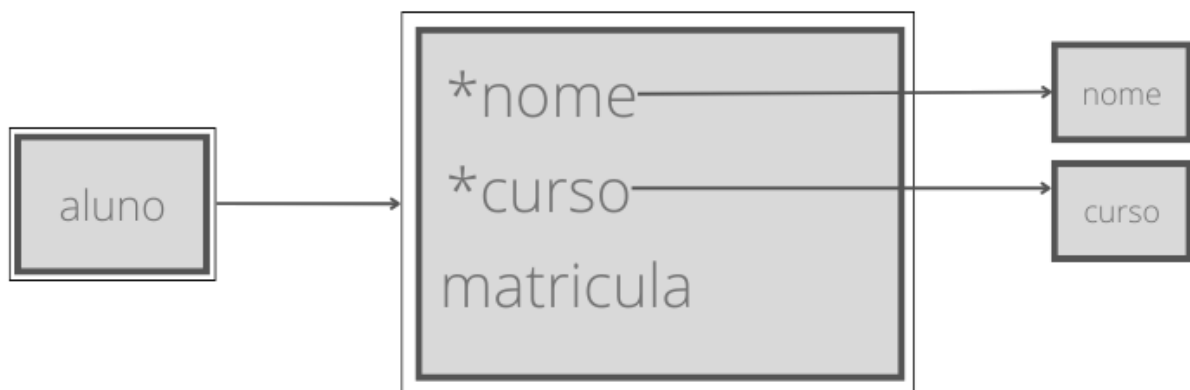
/* Modificando o conteúdo */
(*aluno).nome = nome;
(*aluno).curso = curso;
(*aluno).matricula = 12345;

printf("Nome: %s\n", (*aluno).nome);
printf("Curso: %s\n", (*aluno).curso);
printf("Matrícula: %d\n", (*aluno).matricula);

return 0;
}

```

Observe a representação da estrutura final do exemplo acima:



Como dito anteriormente, precisamos especificar que queremos acessar o valor do ponteiro aluno, com **(*aluno)**, para então especificar qual o campo queremos acessar com o operador de seleção direta **"."**. Esse processo se torna bastante chato ao passo que podemos ter que fazer diversos acessos, então, por este motivo, foi criada uma outra forma de acesso para simplificar a manipulação dos elementos de uma variável struct alocada dinamicamente. Veja a diferença:

```

// Método 1
(*aluno).nome

```

```
// Método 2  
aluno->nome
```

O uso da seta é apenas um “*açúcar semântico*”, ou seja, é apenas uma abstração da notação ponto ao acessar registros referenciados por ponteiro.

4.3. Alias de variáveis do tipo struct (usando o typedef)

Para finalizar o tópico de registros vamos melhorar o exemplo anterior dando um “apelido” à nossa estrutura, de tal forma que poderemos simplificar o tipo “**struct aluno**” para apenas “**Aluno**”, facilitando a escrita do nosso algoritmo. Quando estiver escrevendo programas maiores ficará mais evidente o ganho com esse processo de simplificação do código com a atribuição de novos nomes às nossas variáveis do tipo **struct**.

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

struct aluno {
    char *nome;
    char *curso;
    int matricula;
};

/* Apelidando 'struct aluno' de Aluno */
typedef struct aluno Aluno;

/* Apelidando 'struct aluno *' de ptAluno */
typedef struct aluno * ptAluno;

int main () {

    /*
     * Substituindo com os apelidos referentes
     * a estrutura aluno (Aluno) e ao ponteiro para a
     * estrutura aluno (ptAluno).
     */

    ptAluno aluno;
    aluno = (ptAluno)malloc(sizeof(Aluno));
    if (aluno == NULL) {
        perror("Problema na alocação da memória\n");
        exit(1);
    }

    char nome[] = {"chalet"};
    char curso[] = {"Ciência da Computação"};

    aluno->nome = nome;
    aluno->curso = curso;
```

```
aluno-&gtmatricula = 12345;

printf("Nome: %s\n", aluno-&gtnome);
printf("Curso: %s\n", aluno-&gtcurso);
printf("Matrícula: %d\n", aluno-&gtmatricula);

return 0;
}
```

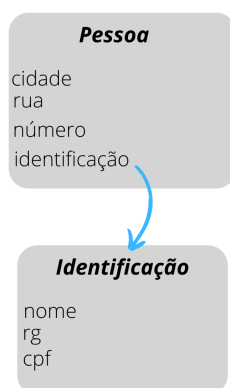
Observe que além de apelidar a variável **struct aluno** como sendo **Aluno** também apelidamos o ponteiro **struct aluno *** como sendo **ptAluno**, abstraindo ainda mais nosso código.

Lembrando também que o typedef pode ser em torno da criação da struct. Observe.

```
typedef struct aluno {
    char *nome;
    char *curso;
    int matricula;
}Aluno;
```

Exercícios

1. Crie um registro chamado **identificacao** contendo três variáveis, uma representando o **nome**, outra o **rg** e a última representando o **cpf**. Após isso crie um registro **pessoa** que tenha os campos **identificacao**, **cidade**, **rua** e **número**. Por fim, crie uma variável do tipo **pessoa**, preencha todos os dados e mostre na tela os dados inseridos a partir da variável do tipo **pessoa** criada.



2. Utilizando o exercício anterior. Crie uma vetor de pessoas (de forma estática), insira 5 elementos no vetor e mostre cada elemento do vetor.
3. Crie um registro chamado **Compra** contendo um vetor de 3 posições chamado **produtos** (cada posição do produto será um “**char ***”, que irá apontar para a string produto que será criada) e uma variável **subtotal**. Aloque um registro do tipo **Compra**. Preencha todos os campos da compra e mostre o resultado das inserções.
4. Aloque dinamicamente uma variável **pessoa** do tipo **Pessoa** e de forma estática uma **curso** do tipo **Curso**. Insira a variável **curso** no campo **formacao** da variável **pessoa**. Feito isso você deve preencher e mostrar todos os dados a partir da variável **pessoa**.

```

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

typedef struct curso {
    char nomeCurso[50];
    char nomeUniversidade[50];
}Curso;

typedef struct pessoa {

```

```

    int id;
    Curso* formacao;
}Pessoa;

int main () {

    /*
     * Procedimento
     * */

    return 0;
}

```

5. Usando a questão anterior. Aloque dinamicamente uma variável do tipo **Pessoa** e a variável **formacao** que está contida na variável do tipo **Pessoa** criada anteriormente. Após isso insira os dados do curso e da pessoa. Por fim, mostre todos os dados a partir da variável **pessoa**.
6. Com o código abaixo: Faça o swap de **p1** e **p2** usando **p3**.

```

#include <stdio.h>

typedef struct pessoa {
    int idade;
    char *nome;
}Pessoa;

int main () {

    Pessoa pessoa1 = {18, "José"};
    Pessoa pessoa2 = {20, "Maria"};

    Pessoa *p1, *p2, *p3;
    p1 = &pessoa1;
    p2 = &pessoa2;

    /* Antes */
    printf("pessoa1: {%d, %s}\n", p1->idade, p1->nome);
    printf("pessoa2: {%d, %s}\n", p2->idade, p2->nome);

    /*
     * Procedimento

```

```
    * */

    /* Depois */
    puts("");
    printf("pessoa1: {%d, %s}\n", p1->idade, p1->nome);
    printf("pessoa2: {%d, %s}\n", p2->idade, p2->nome);

    return 0;
}
```

7. Dado um vetor de 5 posições, desordenado, e um array com strings mapeando cada posição do vetor contendo o seu número por extenso, ou seja, se vetor[0] tiver o número 5, array[0] conterá a string “Cinco”, se vetor[1] tiver o valor 3, array[1] terá a string “Três”... Ordene os dados com a restrição de que seja feita apenas uma chamada para a função de ordenar. Você pode implementar a sua função de ordenação (insertion sort) ou usar o **qsort** da biblioteca stdlib.

```
#include <stdio.h>

int main () {

    char *arr[] = {"Três", "Quatro", "Dois", "Um", "Cinco",
    NULL};
    int vet[] = {3,4,2,1,5};

    // Quantidade de elementos em vet.
    int tamVet = sizeof(vet) / sizeof(vet[0]); //Elemento em vet

    /*
     * Procedimento
     * */

    /*
     * Mostre os dados
     * */

    return 0;
}

/*
 * Saída esperada
 * 1, Um
 * 2, Dois
 * 3, Três
 * 4, Quatro
 * 5, Cinco
 * */
```

8. Insira os valores da variável **segundo**, utilize o ponteiro **prox** da estrutura para ligar a variável **primeiro** a **segundo** e mostre os dados das duas a partir da variável **primeiro**.

```
#include<stdio.h>

// os naipes serão letras [A-Z]

typedef struct carta {
    char naipe;
    int numero;
}Carta;

typedef struct elemento {
    int id;
    struct carta carta;
    struct elemento *proxElemento;
}Elemento;

int main () {

    // Elemento xxxx = {id, {naipe, numero}, proxElemento}
    Elemento primeiro = {1, {'A', 4}, NULL};
    Elemento segundo;

    /*
     * Procedimento
     * */

    return 0;
}
```

9. Utilizando uma variável auxiliar para iterar a lista de 6 elementos, crie um laço de repetição que irá mostrar os dados de todos os elementos do código abaixo. Utilize o ponteiro **aux** e o ponteiro **proxElemento** para iterar a lista que começa com a variável **primeiro** e vai até NULL.

```
#include<stdio.h>

typedef struct carta {
    char naipe;
    int numero;
}Carta;
```

```

typedef struct elemento {
    int id;
    struct carta carta;
    struct elemento *proxElemento;
}Elemento;

int main () {

    // Elemento xxxx = {id, {naipe, numero}, proxElemento};
    Elemento sexto = {1, {'A', 1}, NULL };
    Elemento quinto = {2, {'B', 2}, &sexto };
    Elemento quarto = {3, {'C', 3}, &quinto };
    Elemento terceiro = {4, {'D', 4}, &quarto };
    Elemento segundo = {5, {'E', 5}, &terceiro };
    Elemento primeiro = {6, {'F', 6}, &segundo };
    Elemento *aux;

    /*
     * Temos elementos interligados através do ponteiro
     * proxElemento.
     *
     * [primeiro]-> [segundo]-> [terceiro]-> [quarto]->
     * [quinto]-> [sexto]-> NULL
     *
     * */

    /* atribuição de aux */
    while ( /* condicional de aux */ ) {
        /* print dos dados */
        /* andar com o ponteiro aux */
    }

    return 0;
}

```

10. Utilizando a resposta da questão anterior: Crie uma estrutura chamada **Descritor** que terá um ponteiro para referenciar o primeiro elemento da lista, outro para referenciar o último, e também uma variável para contabilizar a quantidade de elementos da lista. Use um nó/elemento **Descritor** na atribuição do **aux** e no condicional do laço de repetição.

5. INTRODUÇÃO (AED)

5.1. Tad

Em uma linguagem de programação, um tipo de dado define que tipo de informação pode ser armazenada em uma variável. Já um tipo estruturado de dados, define um conjunto de dados que são agrupados sob um mesmo nome, como um vetor ou uma struct.

Já um TAD define, além do tipo estruturado de dados, as funções de acesso aos dados, quais os parâmetros para estas funções e quais retornos estas funções geram quando se acessam os dados.

Utilizando a ideia de TAD, consegue-se implementar três conceitos fundamentais de Orientação a Objetos, que são:

- **Abstração:** refere-se à decomposição de um sistema complexo em partes fundamentais que são descritas com uma linguagem simples e precisa (para “abstrair” os detalhes desnecessários);
- **Encapsulamento:** define que cada elemento do sistema deve esconder dos demais a forma como ele está implementado, fornecendo apenas uma *‘interface de acesso’*. O encapsulamento define como se utilizam os dados, sem uma preocupação sobre como estes são implementados;
- **Modularidade:** define que um sistema pode ser construído a partir de módulos mais simples, que podem inclusive ser usados em outros sistemas. Estes módulos devem interagir entre si para executarem a função definida para o sistema;

Só que tad em C é um conceito mais pífio que o da orientação a objeto.

A Modularização é basicamente a convenção em C de como preparar dois arquivos para implementar o “tad”.

Arquivo “.h” : protótipo das funções , tipos de ponteiros e dados globalmente acessíveis.

Arquivo “.c”: declaração do tipo de dados e implementações das funções

Exemplo:

Arquivo.h

```
typedef struct {
    double x, y;
} PONTO;

void inicializaPonto(PONTO *p, int x, int y);
```

```
void imprimePonto(PONTO p);
```

teremos só a declaração das funções e os seus parâmetros.

Arquivo.c

```
#include "Arquivo.h"

void inicializaPonto(PONTO *p, int x, int y){
    /*a implementação da função */
}

void imprimePonto(PONTO p){
    /*a implementação da função */
}
```

nesse arquivo teremos as funções e o que elas executam. E por fim:

Main.c

```
#include<stdio.h>
#include "Arquivo.h"
int main(){

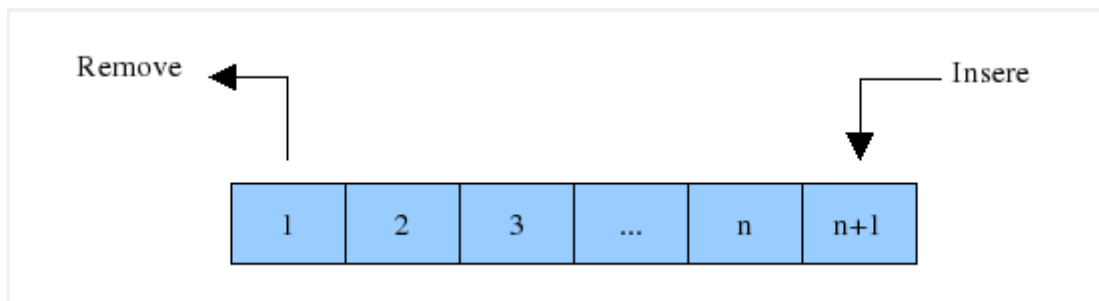
    struct Ponto p;
    inicializaPonto(PONTO *p, int x, int y);

    return 0;
}
```

Esse arquivo vai ser onde você vai chamar as funções para serem rodadas.

5.2. Fila - Explicação e Algoritmo

São estruturas de dados do tipo FIFO (first-in first-out), onde o primeiro elemento a ser inserido, será o primeiro a ser retirado, ou seja, adiciona-se itens no fim e remove-se do início.



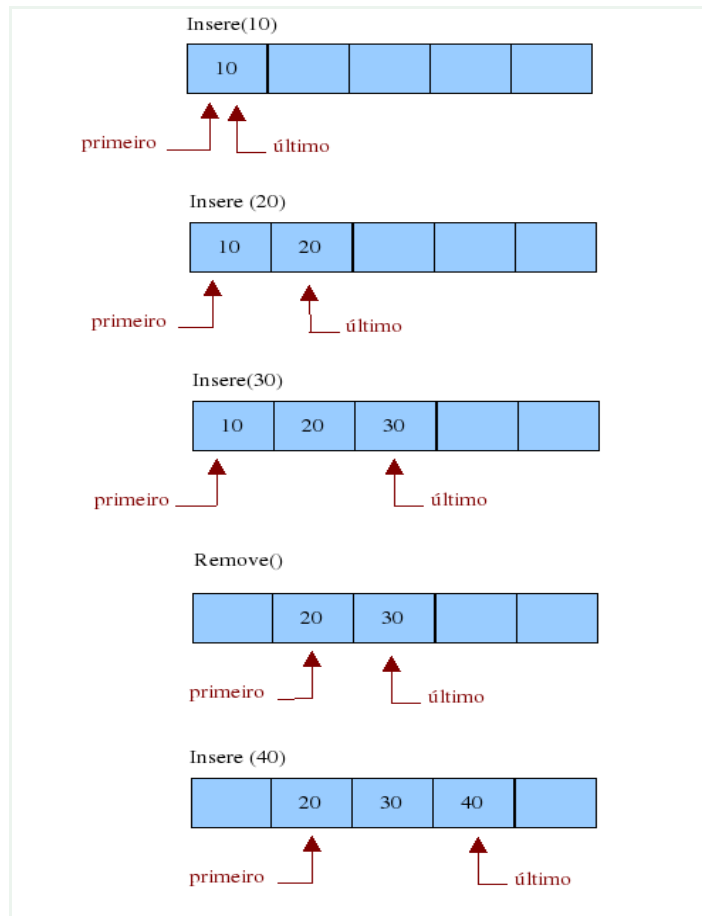
A implementação de filas pode ser realizada através de vetor (alocação do espaço de memória para os elementos é contígua) ou através de listas encadeadas.

5.2.1. Operações com fila:

Todas as operações em uma fila podem ser imaginadas como as que ocorrem numa fila de pessoas num banco, exceto que o elementos não se movem na fila, conforme o primeiro elemento é retirado. Isto seria muito custoso para o computador. O que se faz na realidade é indicar quem é o primeiro.

- criação da fila (informar a capacidade no caso de implementação sequencial - vetor);
- enfileirar (inserir) - o elemento é o parâmetro nesta operação;
- desenfileirar (remover);
- mostrar a fila (todos os elementos);
- verificar se a fila está vazia (isEmpty);

como mostrado na imagem a seguir supondo uma fila com capacidade para 5 elementos (5 nós).



Na realidade, a remoção de um elemento da fila é realizada apenas alterando-se a informação da posição do último.

Veja o algoritmo a seguir para Criar uma fila de números reais:

No arqv.c:

```
#include<stdio.h>
#include "arqv.h"
Fila *Criar(){
    Fila *fi = (Fila*) malloc(sizeof(Fila)); //alocado memória
    if(fi !=null){
        fi->inic = null;
        fi->fim = null;
    }
    return fi;
}
```

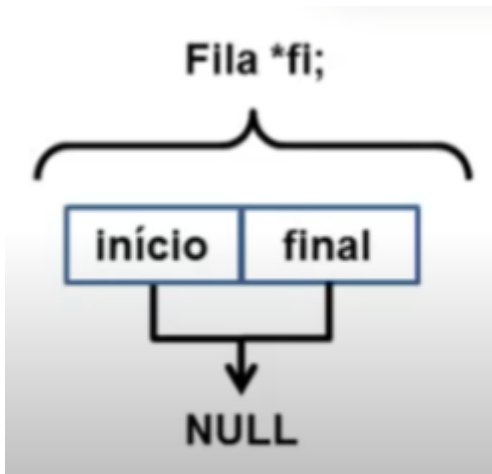


imagem-1

no arqv.h:

```
#include<stdio.h>
Struct dados {
    int numero;
}
struct elemento{
    struct elemento *prox;
    struct dados aluno;
}
typedef struct elemento elem;
struct No{
    struct elemento *inic;
    struct elemento *fim;
}
typedef struct No Fila;

Fila *Criar();
```

No main.c

```
#include<stdio.h>
#include "arqv.h"
int main(){

    Fila *a;
    a=NULL;
    a=Criar();
    struct dados b;

    return 0;
}
```

O que esse algoritmo está fazendo?

no main ele está criando uma fila e deixando como na imagem-1 deixando a fila apontando para o null.

5.2.2. Inserção em uma fila:



```

#include<stdio.h>
#include "arqv.h"

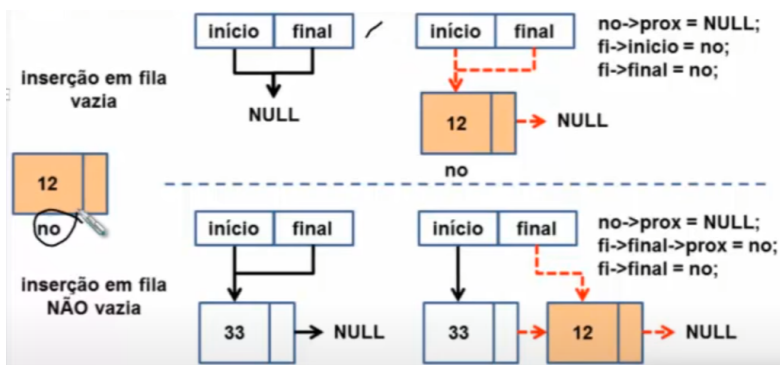
//passando qual fila vou inserir e quais são os dados
int inserir(Fila *fi,struct dado dados){

    //verificando se a fila existe
    if(fi==null) return 0;

    //alocando memória para o nó que você vai inserir
    Elem *novo = (Elem*)malloc(sizeof(Elem));

    if(novo == null) return 0;
    novo->aluno= dados;
    novo->prox = null;
    if(fi->inic){
        fi->fim->prox=novo;
    }else{
        fi->inic= novo;
    }
    fi->fim =novo;
    return 1;
}

```



arqv.h

```

#include<stdio.h>
Struct dados {
    int numero;
}
struct elemento{

```

```
    struct elemento *prox;
    struct dados aluno;
};

typedef struct elemento elem;

struct No{
    struct elemento *inic;
    struct elemento *fim;
};

typedef struct No Fila;

Fila *Criar();
int inserir(Fila *fi,struct dado dados);
```

main.c

```
#include<stdio.h>
#include "arqv.h"
int main(){

    Fila *a;
    a=NULL;
    a=Criar();
    struct dados b;
    b.numero = 6;
    inserir(a,b);
    return 0;
}
```


5.2.3. Remover em uma fila:

A remoção da fila é feita no início.

arqv.c

```
#include<stdio.h>
#include "arqv.h"
//passando os parâmetros qual fila vou inserir e quais são os dados
int remover(Fila *fi){
    if(fi==null)return 0; //verificando se a fila existe
    Elem* aux;
    aux=fi->inic;
    fi->inic = fi->inic->prox;
    if(fi->inic ==null) fi->fim =null;

    free(aux);
    return 1;
}
```

arqv.h

```
#include<stdio.h>
Struct dados {
    int numero;
};
struct elemento{
    struct elemento *prox;
    struct dados aluno;
};
typedef struct elemento elem;

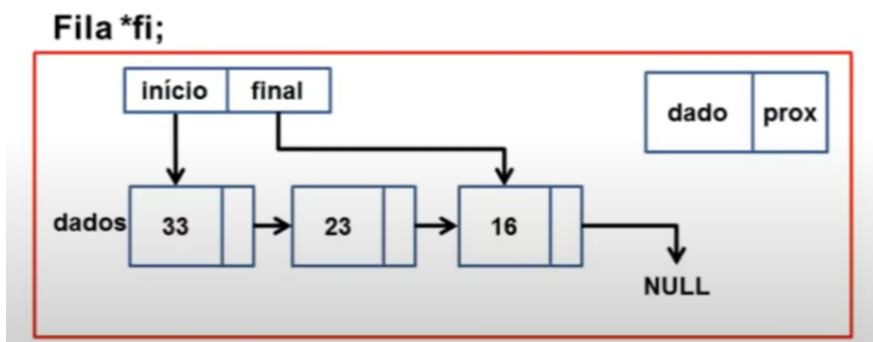
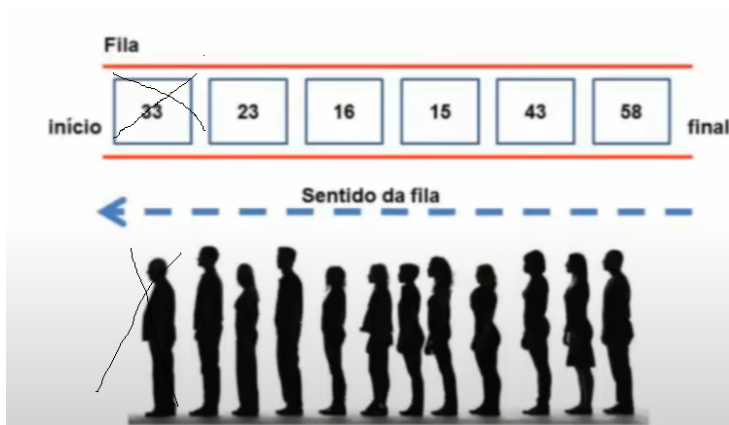
struct No{
    struct elemento *inic;
    struct elemento *fim;
};

typedef struct No Fila;
```

```

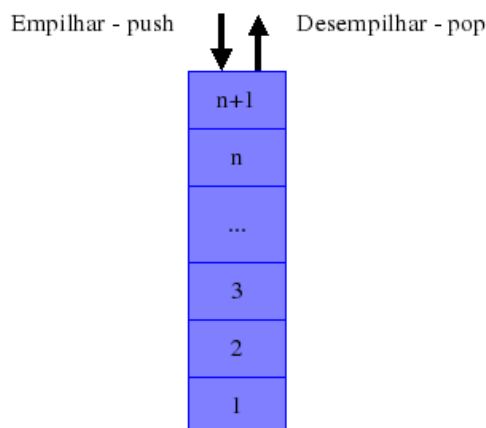
Fila *Criar();
int inserir(Fila *fi, struct dado dados);
int remover(Fila *fi)

```



5.3. Pilha - Explicação e Algoritmo

São estruturas de dados do tipo LIFO (last-in first-out), onde o último elemento a ser inserido, será o primeiro a ser retirado. Assim, uma pilha permite acesso a apenas um item de dados - o último inserido. Para processar o penúltimo item inserido, deve-se remover o último.



São exemplos de uso de pilha em um sistema:

- Funções recursivas em compiladores;
- Mecanismo de desfazer/refazer dos editores de texto;
- Navegação entre páginas Web;
- etc.

A implementação de pilhas pode ser realizada através de vetor (alocação do espaço de memória para os elementos é contígua) ou através de listas encadeadas.

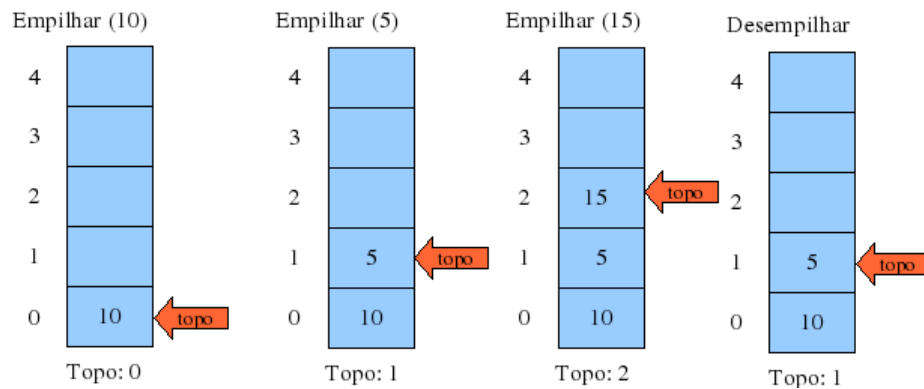
Numa pilha, a manipulação dos elementos é realizada em apenas uma das extremidades, chamada de topo, em oposição a outra extremidade, chamada de base.

5.3.1. Operações com pilha

Todas as operações em uma pilha podem ser imaginadas como as que ocorre numa pilha de pratos em um restaurante ou como num jogo com as cartas de um baralho:

- criação da pilha (informar a capacidade no caso de implementação sequencial - vetor);
- empilhar (push) - o elemento é o parâmetro nesta operação;
- desempilhar (pop);
- mostrar o topo;
- verificar se a pilha está vazia

Supondo uma pilha com capacidade para 5 elementos (5 nós).



Agora vamos ver no algoritmo

Criar pilha

arq.c

```
#include<stdio.h>
#include "arq.h"
Pilha *Criar(){
    Pilha *pi = (Pilha*) malloc(sizeof(Pilha)); //alocado memória
    if(pi !=null){
        pi->inic = null;
        pi->fim = null;
    }
    return pi;
}
```

arq.h

```
#include<stdio.h>
Struct dados {
    int numero;
}
struct elemento{
    struct elemento *prox;
    struct dados aluno;
};

typedef struct elemento elem;
```

```

struct No{
    struct elemento *inic;
    struct elemento *fim;
};

typedef struct No Pilha;

Pilha *Criar();

```

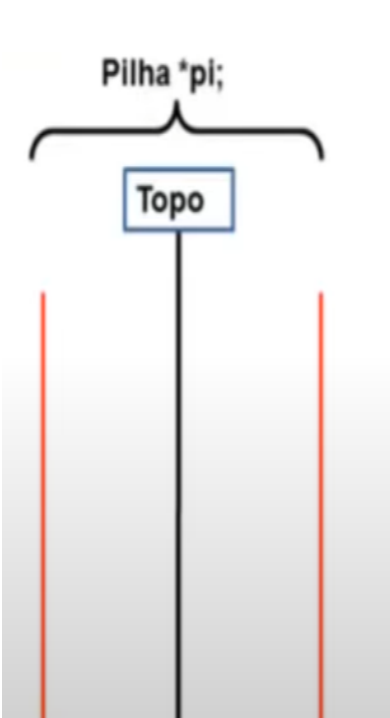
```

#include<stdio.h>
#include "arq.h"
int main(){

    Pilha *a;
    a=NULL;
    a=Criar();
    struct dados b;

    return 0;
}

```



O que esse algoritmo está fazendo é criando o top da pilha, e assim a inicializando com *NULL*.

5.3.2. Inserção

Arqv.c

```
#include<stdio.h>
#include "arq.h"
//passando os parâmetros qual fila vou inserir e quais são os dados
int inserir(Pilha *pi,struct dado dados){
    if(fi==null)return 0; //verificando se a fila existe
    Elem *novo = (Elem*)malloc(sizeof(Elem)); //alocando memoria o no q
                                                //vc vai inserir

    if(novo == null) return 0;
    novo->aluno= dados;
    novo->prox = (*pi);
    *pi=novo->prox;

    return 1;

}
```

Arqv.h

```
Pilha *Criar();
int inserir(Pilha *pi,struct dado dados);
```

E no main.c

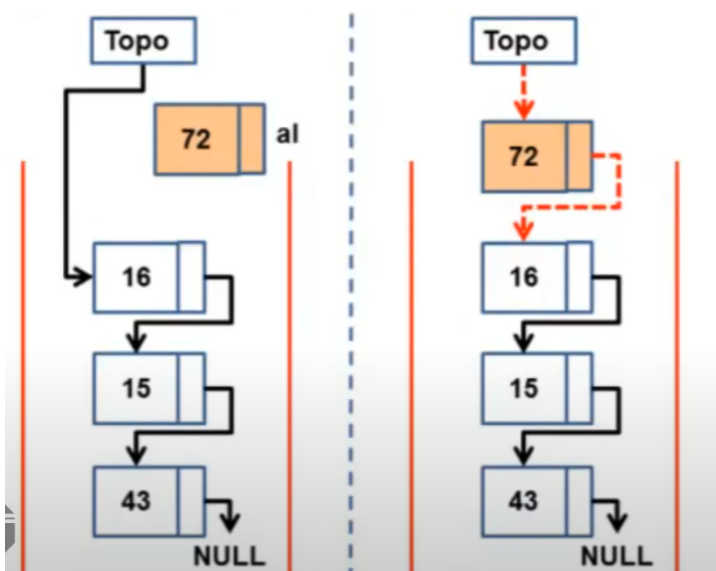
```
#include<stdio.h>
#include "arq.h"
int main(){

    Pilha *a;
    a=NULL;
    a=Criar();
    struct dados b;
    b.numero = 43;
    inserir(a,b);
    b.numero = 15;
    inserir(a,b);
    b.numero = 16;
    inserir(a,b);
    b.numero = 72;
    inserir(a,b);
    return 0;
}
```

Veja como irá funcionar na prática:

```
int x = insere_Pilha(pi, dados_aluno);
```

no->dados = al;
no->prox = (*pi);
***pi = no;**



O 72 que é o novo dado quer entrar na pilha, vai apontar para onde o topo estava apontando e o topo vai começar apontar para o 72 para eles não perderem o apontamento dos outros que já estavam na pilha.

5.3.3. Remoção

Arqv.c

```
#include<stdio.h>
#include "arq.h"
//passando os parâmetros qual fila vou inserir e quais são os dados
int remover(Pilha *pi){
    if(pi==null)return 0; //verificando se a fila existe
    Elem* aux;
    aux=*pi
    *pi=aux->prox;
    free(aux);
    return 1;
}
```

Arqv.h

```
Pilha *Criar();
int inserir(Pilha *pi,struct dado dados);
int remover(Pilha *pi);
```

E no main.c

```
#include<stdio.h>
#include "arq.h"
int main(){

    Pilha *a;
    a=NULL;
    a=Criar();
    struct dados b;
    b.numero = 43;
    inserir(a,b);
    b.numero = 15;
    inserir(a,b);
    b.numero = 16;
```



```

    inserir(a,b);
    b.numero = 72;
    inserir(a,b);
    remover(a);
    return 0;
}

```

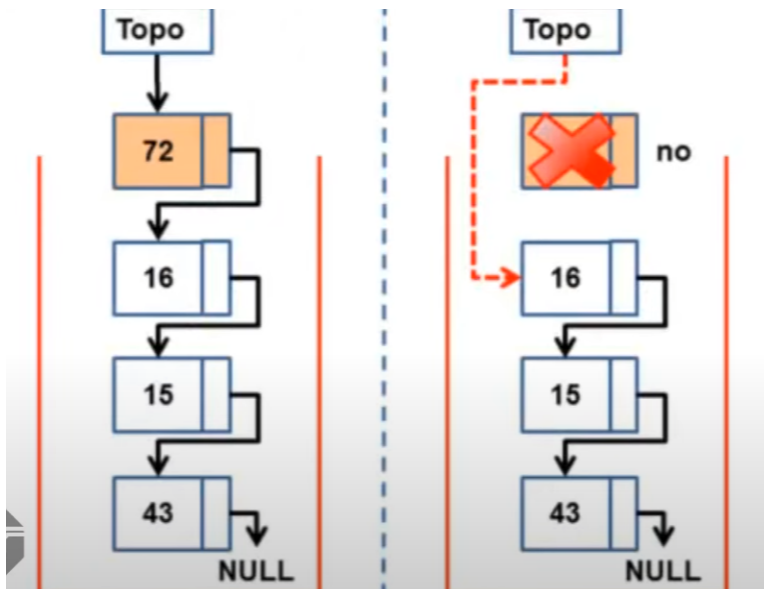
O que esse algoritmo está fazendo.

```

int x = remove_Pilha(pi);

```

Elem *no = *pi;
*pi = no->prox;
free(no);

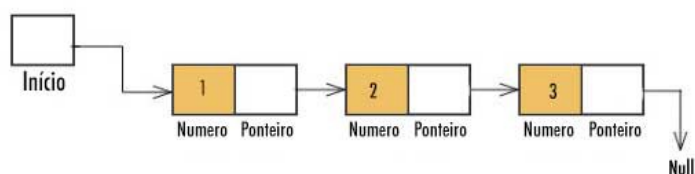


O ponteiro que referencia o topo aponta para o próximo do próximo elemento. Salva o nó intermediário com um ponteiro chamado *no*. Com isso ele poderá remover sem perder referência para o restante da pilha.

5.4. Lista - Explicação e Algoritmo

Linked list ou lista encadeada é um tipo de estrutura de dados que contém um grupo de nós interligados através de ponteiros, onde o ponteiro dentro da estrutura aponta para o próximo nó até que o ponteiro seja NULL indicando assim o fim da lista.

Nesse exemplo vou mostrar uma lista simples com um único valor inteiro e um ponteiro referenciando o próximo elemento, para exemplificar a construção de uma lista encadeada simples.



Vamos para o código:

5.4.1. Criando a lista

No arq.h

```
#include<stdio.h>
Struct dados {
    int numero;
}
struct elemento{
    struct elemento *prox;
    struct dados aluno;
}
typedef struct elemento elem;
typedef struct elemento *Lista;

Lista *Criar();
```

arq.c

```
#include<stdio.h>
#include "arq.h"
Lista *Criar(){
    Lista *li = (Lista*) malloc(sizeof(Lista)); //alocado memória
    if(li !=null){
```

```
    *li= null;

}
return li;
}
```

E no main.c

```
#include<stdio.h>
#include "arq.h"
int main(){

    Lista *a;
    a=NULL;
    a=Criar();
    struct dados b;

    return 0;
}
```

Analise a imagem abaixo, estou apenas criando a lista e apontando a referência do início da lista para NULL.



5.4.2. Liberar lista

arq.c

```
#include<stdio.h>
#include "arq.h"
void Liberar(Lista *li){
    if(li!=null){
        Elem * no;
        while((*li)!=null){
            no =*li;
            *li =(*li)->prox;
            free(no);
        }
        free(li);
    }
}
```

5.4.3. Inserção

Na lista teremos 3 tipos de inserção no início, no meio e no fim.

Inserir no início

```
int InserirInicio(Lista* li, struct dados novos){
    if(li==NULL)return 0;
    Elem *novo = (Elem*)malloc(sizeof(Elem));

    if(novo == NULL)return 0;
    novo->dados= novos;
    novo->prox=*li;
    *li=novo;
    return 1;
}
```

O inserir no início da lista é o mesmo inserir da pilha.

Inserir no fim

```
int InserirFim(Lista* li, struct dados novos){
    if(li==NULL)return 0;
    Elem *novo = (Elem*)malloc(sizeof(Elem));
    if(novo == NULL)return 0;
    novo->dados=novos;
    novo->prox= NULL;
    if(*li ==NULL){
        *li=novo;
    }else{
        Elem* aux;
        while (aux->prox !=NULL)
        {
            aux=aux->prox;
        }
        aux->prox=novo;
    }
    return 1;
}
```

o inserir no fim você vai percorrer a lista inteira para inserir no último elemento caso a lista tenha elementos caso não tenha ele vai inserir no início mesmo

Inserir no meio

```
InserirMeio(Lista* li, struct dados novos){
    if(li==NULL)return 0;
    Elem *novo = (Elem*)malloc(sizeof(Elem));
    if(novo == NULL)return 0;
    novo->dados=novos;

    if(*li ==NULL && (*li)->dados.matricula > novo->dados.matricula){

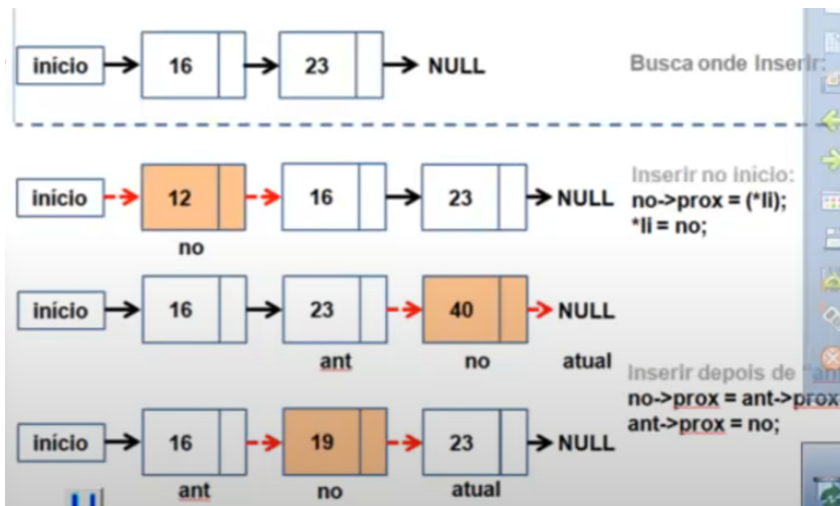
        novo->prox= *li;
        *li=novo;
    }else{
        Elem *ant = *li;
        Elem* aux= ant->prox;
        while (aux->prox !=NULL &&
            aux->dados.matricula > novo->dados.matricula)
        {
            ant=aux;
            aux=aux->prox;
        }
    }
}
```

```

    ant->prox=novo;
    novo->prox=aux;
}
}

```

O que o código está fazendo?



5.4.4. Remoção

Veremos a remoção no início, meio e fim. Porém, para facilitar o entendimento, veremos primeiro como remover no início e no fim, para depois vermos como será a remoção no meio.

Remover no início

```

int removerInic(Lista *li){
    if(li==NULL)return 0;
    Elem *aux;
    aux=*li;
    *li=(*li)->prox;
    free(aux);
    return 1;
}

```

O remover no início é básico, ele remove o primeiro elemento e coloca para lista apontar para o próximo.

Remover no fim

```
int removerFim(Lista *li){
    if(li==NULL)return 0;

    if ((*li)->prox==NULL)
    {
        Elem *aux;
        aux=*li;
        *li=(*li)->prox;
        free(aux);
        return 1;
    }else{
        Elem *aux=*li;
        Elem *ant;
        while (aux->prox !=NULL)
        {
            ant=aux;
            aux= aux->prox;
        }

        ant->prox=aux->prox;

        free(aux);
        return 1;
    }
}
```

Para remover no fim ele vai até os dois últimos da lista, remove o último e o antepenúltimo vai começar a apontar para **NULL** de modo que o seu ponteiro **prox** servirá de referência como o fim da lista.

Remover no meio

```
InserirMeio(Lista* li, struct dados novos){
    if(li==NULL)return 0;
    Elem *novo = (Elem*)malloc(sizeof(Elem));
    if(novo == NULL)return 0;
    novo->dados=novos;

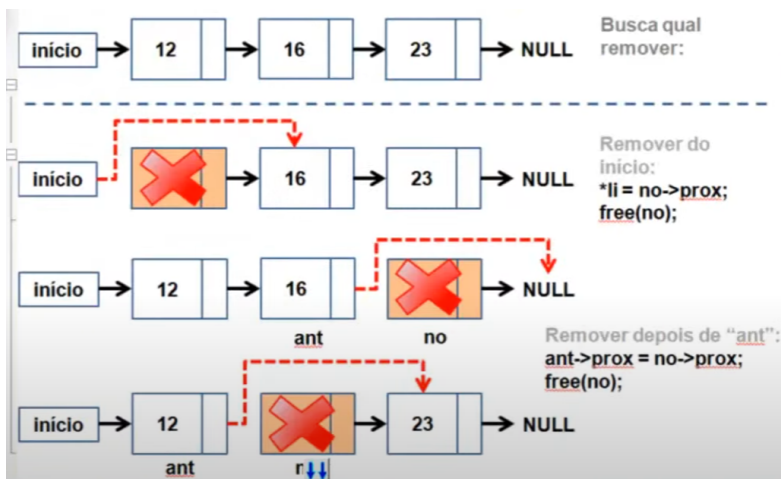
    if(*li ==NULL && (*li)->dados.matricula > novo->dados.matricula){

        novo->prox= *li;
        *li=novo;
    }else{
        Elem *ant = *li;
        Elem* aux= ant->prox;
        while (aux->prox !=NULL &&
```

```

        aux->dados.matricula > novo->dados.matricula)
    {
        ant=aux;
        aux=aux->prox;
    }
    ant->prox=novo;
    novo->prox=aux;
}
}

```



6. Noções de complexidade

6.1. Constante vs Linear

Seja no mercado de trabalho ou na universidade, um tema que é amplamente abordado e requisitado na Ciência da Computação é a complexidade de algoritmos. Por conta disso, vamos dar os primeiros passos nessa direção. A ideia aqui não será, nem de perto, nos aprofundar em complexidade, mas dar uma noção a ponto de não estranhar o tema quando visto de forma mais profunda nas cadeiras específicas.

A complexidade de algoritmos é dividida em dois pontos: a complexidade de tempo, que analisa o comportamento do algoritmo com o decorrer do tempo, e a complexidade de espaço, que analisa o uso de memória da máquina pelo algoritmo. Pelo avanço dos computadores em relação à quantidade de memória, a análise de espaço se torna menos importante para o escopo desta apostila, por conta disso iremos nos ater a complexidade de tempo.

Para uma primeira abordagem, vejo como suficiente entender a complexidade dos algoritmos vistos anteriormente em termos de complexidade constante e linear.

Por um lado, uma complexidade constante é aquela que podemos mensurar a quantidade de operações de forma discreta, ou seja, há uma quantidade finita e exata de operações a ser executada. Veja o trecho abaixo, podemos afirmar que a complexidade é constante porque temos a quantidade exata de operações, sem depender de alguma variável. Neste caso são 3 operações.

```
a = 2;  
b = a;  
b++;
```

Veja este outro exemplo com complexidade constante

```
for (int i = 0; i < 100; i++) { // 101 Opeações  
    printf("%d, ", i); // Uma operação por iteração * 100 iterações  
}
```

Observe que o print é executado 100 vezes, com o **i** indo de 0 a 99. Já dentro do laço **for** o **i** chega a assumir o valor 100 para que a condição **i < 100** não seja satisfeita e o programa possa seguir para fora do escopo do laço de repetição. Novamente, como podemos afirmar uma quantidade fixa de operações, dizemos que este trecho de código tem complexidade constante.

Por outro lado, uma complexidade linear se caracteriza quando não podemos afirmar um valor fixo de operações para o trecho que está sendo analisado. Veja:

```
for (int i = 0; i < n; i++) { // n iterações
    printf("%d, ", i); // Uma operação por iteração * n iterações
}
```

Como o laço depende da variável **n**, não podemos afirmar a quantidade de operações que há nesse trecho, então denominamos a complexidade de tempo como linear. É importante ter em mente que, neste caso, quando estamos falando de complexidade linear estamos olhando para o pior caso, veja mais um exemplo de complexidade linear:

```
scanf("%d", &num);
for (int i = 0; i < n; i++) { // No pior caso terá n iterações
    if (i == num) {
        printf("%d, ", i);
        break;
    }
}
```

Mesmo que **num** possa assumir 0, 1, 2, $n/2$, $n/10$... assumimos o pior caso para a complexidade, que é quando **num** é igual a **n**. Esse trecho, que é linear, em notação Big O é $O(n)$, mas nos aprofundar nisso fugiria do escopo desta apostila.

Observe que além de linear a complexidade também pode assumir uma complexidade quadrática... veja o exemplo:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        printf("%d, ", j); // n * n = n²
    }
}
```

Vemos que **i** depende de **n** de tal forma que cada passo depende do laço mais interno, já o laço mais interno tem **j** dependendo de **n** também. Ou seja, como para cada execução de **i** indo de **[0 - n]**, **j** também será executado **[0 - n]**.

Neste momento você já deve ter notado a semelhança com o algoritmo de preencher uma matriz. Analise a complexidade do código abaixo.

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

// Criando uma matriz n x n

int main () {

    int dim;
    scanf("%d", &dim);

    int **matriz = (int**) malloc(sizeof(int*)*dim);

    // (*) Há laços aninhados
    for (int i = 0; i < dim; i++) {
        matriz[i] = malloc(sizeof(int) * dim);

        for (int j = 0; j < dim; j++) {
            matriz[i][j] = i * dim + j + 1;
        }
    }

    // (#) Há laços aninhados
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            printf("%-3d ", matriz[i][j]);
        }
        printf("\n");
    }

    /* Observe que (*) e (#) não estão aninhados entre si */

    return 0;
}
```

Agora podemos ter uma melhor noção sobre o que estamos fazendo e em quais situações devemos usar os nossos algoritmos.

6.2. Complexidade da fila

6.2.1. Dinâmica

Tome o algoritmo de fila dinâmica visto antes como exemplo, fica fácil de ver que a complexidade de inserção é constante, porque, nesta implementação, temos um ponteiro referenciando o final da fila (local que será adicionado um novo nó). Do mesmo modo, a remoção também é constante, pois ela é feita com o ponteiro que está no início da fila.

É importante ter em mente que a implementação pode ser feita sem o ponteiro referenciando o final da fila, de modo que para inserir um novo nó será necessário percorrer toda a fila, assim tornando a complexidade linear.

6.2.2. Estática

Assim como na fila dinâmica a inserção na fila estática segue constante, porém sempre que formos “remover” um elemento, tirando ele do início, teremos que fazer um “*shift*” em todos os outros, tornando a complexidade de remoção como linear.

6.3. Complexidade da pilha

6.3.1. Dinâmica

Assim como a fila, tanto a remoção quanto a inserção é constante, já que ambas serão feitas sempre no “*topo*”, local onde teremos um ponteiro para serem feitas as manipulações.

Neste caso também vale ressaltar que a implementação pode ser feita de um modo bem inocente, acarretando em ter que percorrer toda a pilha para fazer uma remoção. Fica a deixa para implementar essa versão de pilha.

6.3.2. Estática

A pilha estática tem complexidade de inserção e “remoção” constante. Neste momento você já deve ter ficado esperto quanto a essa classificação simples de complexidade.

6.4. Complexidade da lista

Por fim, a lista dinâmica é dividida em 3 partes, inserir e remover no início, meio, e fim. O processo de inserir e remover no início é constante, já que temos um ponteiro para o primeiro elemento da lista. Inserir e remover no meio e no fim é linear, já que precisamos percorrer a lista para alcançar a posição requisitada. Neste último caso a complexidade de inserção e remoção pode ser constante caso haja um ponteiro referenciando o último elemento da lista, caso contrário a complexidade será linear. Faça o teste de mão para absorver melhor o conceito.

7. MODULARIZAÇÃO E BOAS PRÁTICAS DE PROGRAMAÇÃO

7.1. Visão geral

Por mais que na maioria do tempo na faculdade não esteja programando softwares de grande porte, sempre é importante ter boas práticas de programação, seja para permitir que tenhamos uma maior facilidade de modificação, que o código seja de fácil entendimento para si e para outros programadores, para um maior reuso ou para uma melhor organização do projeto. A separação em módulos é um grande aliado para manter nosso código limpo, porque nos permite codificar e compilar separadamente cada módulo.

Diferente da abordagem anterior em que tínhamos um arquivo `.c` com uma função `main` e um conjunto de funções declaradas dentro deste mesmo arquivo, agora iremos dividir a responsabilidade das partes que compunham um só `.c` para outros arquivos através da modularização, dividindo os arquivos de forma semântica.

7.2. Demonstração

Para exercitar vamos tomar como exemplo o projeto de uma calculadora que faz operações entre dois números. Teremos as funcionalidades de somar, subtrair, multiplicar e dividir. Para executar as funcionalidades devemos pegar o input do usuário, processar esse input realizando o cálculo e por fim mostrar o resultado na tela.

```
//app.c
#include <stdio.h>

double somar(const double n1, const double n2) {
    return n1 + n2;
}

double subtrair(const double n1, const double n2) {
    return n1 - n2;
}

double multiplicar(const double n1, const double n2) {
    return n1 * n2;
}

double dividir(const double n1, const double n2) {
    return n1 / n2;
}

int main () {
```

```

double num1, num2, resultado;
int op;

printf("Primeiro valor\n-> ");
scanf("%lf", &num1);
printf("Segundo valor\n-> ");
scanf("%lf", &num2);
printf("Operação:\n[1] somar\n[2] subtrair\n[3] multiplicar\n[4]
dividir\n-> ");
scanf("%d", &op);

if (op == 1)
    resultado = somar(num1, num2);
else if (op == 2)
    resultado = subtrair(num1, num2);
else if (op == 3)
    resultado = multiplicar(num1, num2);
else if (op == 4)
    resultado = dividir(num1, num2);

printf("O resultado é: %lf\n", resultado);

return 0;
}

```

Observe que não foi feito tratamento do input, de divisão por zero... O foco será apenas a organização e estrutura do nosso código.

Neste ponto temos algo minimamente funcional, porém estamos atribuindo ao programa a funcionalidade de fazer as operações, mostrar opções e resultados e receber os dados do usuário.

Uma melhor abordagem seria delegar a responsabilidade de fazer as operações, mostrar na tela e receber os dados para “módulos”. Para isso devemos criar um arquivo `.c` para cada comportamento com suas funções, e um arquivo `.h`, com a assinatura das funções que serão públicas, para cada um dos arquivos `.c`, ambos com o mesmo nome. É um programa bem simples, mas os passos para seu desenvolvimento te ajudarão muito ao desenvolver aplicações maiores. Veja como ficou:

Importamos as bibliotecas que iremos usar (serão criadas a baixo), e passamos os parâmetros necessários para cada uma delas. Esse será o coração do nosso programa.

```

// app.c
#include <stdio.h>
#include "telas.h"

```

```
#include "processaResultado.h"

int main () {

    double num1, num2, resultado;
    int op;

    recebeDados(&num1, &num2, &op);
    resultado = calculaResultado(num1, num2, op);
    mostraResultado(resultado);

    return 0;
}
```

Criamos as funções que serão utilizadas para realizar as operações no nosso programa.

```
// operacoes.c
#include "operacoes.h"

double somar(const double n1, const double n2) {
    return n1 + n2;
}

double subtrair(const double n1, const double n2) {
    return n1 - n2;
}

double multiplicar(const double n1, const double n2) {
    return n1 * n2;
}

double dividir(const double n1, const double n2) {
    return n1 / n2;
}
```

Temos que expor as funções no escopo público para permitir que sejam utilizadas por qualquer outro programa que a importe.

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

double somar(const double, const double);
double subtrair(const double, const double);
double multiplicar(const double, const double);
double dividir(const double, const double);
```


Aqui criamos uma “interface” que define qual operação foi selecionada.

```
// processaResultado.c
#include "operacoes.h"
#include "processaResultado.h"

double calculaResultado(const double n1, const double n2, const int op) {
    if (op == 1)
        return somar(n1, n2);
    else if (op == 2)
        return subtrair(n1, n2);
    else if (op == 3)
        return multiplicar(n1, n2);
    else if (op == 4)
        return dividir(n1, n2);
    return 0;
}
```

Expomos a função para o escopo público. Será usada pela função main do projeto.

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

/* Fará a operação entre os dois inteiros */
double calculaResultado(const double, const double, const int);
```

Criamos as telas exibidas para auxiliar o uso do programa e mostrar os resultados.

```
// telas.c
#include "telas.h"

void recebeDados(double *n1, double *n2, int *op) {
    printf("Primeiro valor\n-> ");
    scanf("%lf", n1);
    printf("Segundo valor\n-> ");
    scanf("%lf", n2);
    printf("Operação:\n[1] somar\n[2] subtrair\n[3] multiplicar\n[4] dividir\n-> ");
    scanf("%d", op);
}

void mostraResultado(const double resultado) {
    printf("O resultado é: %lf\n", resultado);
}
```

Aqui também expomos para o escopo público.

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

/* Pega os números que sofrerão a operação e a
 * operação que será feita entre os números.
 */
void recebeDados(double*, double*, int*);

/* Imprime o resultado da operação na tela. */
void mostraResultado(const double);
```

Para compilar tudo e executar pelo terminal basta digitar: ***gcc main.c operacoes.c processaResultado.c telas.c -o app***. Após isso execute o binário com ***./app***.

Esse exemplo tomou mais corpo após as alterações, mas o projeto está bem mais intuitivo e documentado, e é mais fácil de adicionar funcionalidades. Isso se evidencia quando essa sequência de passos é observada em um programa grande, por conta de seu conteúdo estar mais atomizado ao separar suas funcionalidades por tópicos.

É digno de nota que este programa pode não aparentar estar melhor estruturado quando comparado com o outro feito por completo em um arquivo só, porém com o crescimento do código, vai surgindo a necessidade de compilar apenas parte do projeto, já que o tempo de compilação é demorado, e isto só é possível se modularizar o código. Sinta-se encorajado a se aprofundar na linguagem C e aprender a usar ferramentas como *cmake* e *make*, estes conhecimentos, em algum momento, serão importantes na vida acadêmica (com trabalhos mais bem estruturados) ou como programador no mercado de trabalho, seja de forma direta ou indiretamente.

Referências

CORMEN, T. H. et al. **Introduction to Algorithms, third edition**. [s.l.] MIT Press, 2009.

SCHILDT, H. **C Completo e Total**. 3. ed. [s.l: s.n.] Press, 1997.

C++ Standard Library headers - cppreference.com. Disponível em: <<https://en.cppreference.com/w/cpp/header>>.

BACKES, A. **Linguagem C - Completa e Descomplicada**. 2. ed. [s.l: s.n.].