# ESTRUTURA DE DADOS II

**22/08/2023**

## Alocação Dinâmica de Memória

Durante a declaração das variáveis é feita a reserva de memória para cada uma das variáveis declaradas, sendo em quantidade de bytes, de acordo com o tipo de cada variável.

LINGUAGEM C:

- int = 2 bytes p/ número inteiro
- float = 4 bytes p/ número real
- char = 1 byte p/ cada palavra alocada

Essa alocação de memória é **ESTÁTICA**, ou seja, não se altera em tempo de execução.

Algumas aplicações precisam alocar memória em tempo de execução. A Linguagem C oferece a função "*Malloc*" para que se faça alocação de memória durante a execução do programa, ou seja, **ALOCAÇÃO DINÂMICA DE MEMÓRIA.**

Da mesma forma que é necessário alocar memória para armazenar informação é necessário **LIBERÁ-LA** em tempo de execução, quando alocada dinamicamente.

A função *"Free"* libera a memória apontada pelo ponteiro. O tamanho de memória liberado é o mesmo que foi alocado pela função *"Malloc".*

As funções Malloc() e Free() pertencem a biblioteca stdlib.h.

**Exemplo utilizando as duas funções:**

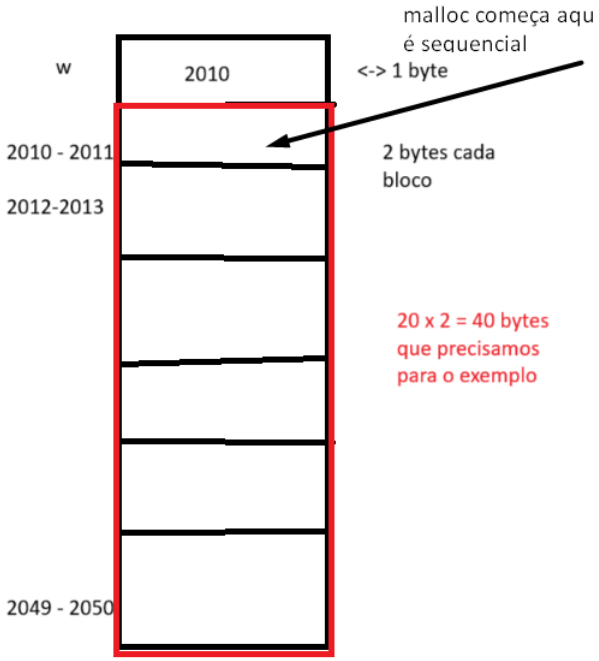
```
int *p, *q;

p = (int *) malloc (sizeof(int));

q = (int *) malloc (sizeof(int));
```

*p = 5;

*q = 10;

Free(p);

P=q;




*"função malloc explicada com desenho de memória"*

*Malloc () = significa que a função irá alocar "size" bytes de memória e retornará um ponteiro para o início da memória alocada.*

**EXEMPLO:**

*Int * w;*

*W = (int *) malloc (20 * sizeof(int));*

*"Durante a execução do programa você vai alocar 20x o valor de um inteiro"*

*Ponteiro = é uma variável tipada que armazena o endereço de memória de outra variável. O "*" indica a declaração do ponteiro.*

*Free() = função que libera a memória apontada pelo ponteiro deixando o programa mais leve.*

**EXEMPLO:**

*int  *p;*

*float *q;*

*p = (int *) malloc(25*sizeof(int));*

*q = (float *) malloc(sizeof(float));*

*free(p);  => Liberou os 50 bytes que 'p' aponta.*

*free(q);  => Liberou os 4 bytes que 'q' aponta.*

**29/08/2023**

## Listas Lineares

Fundamentos:

Uma lista linear é uma correlação L: [L1, L2, L3, ... Ln], n >= 0, cuja propriedade estrutural baseia-se na posição relativa dos elementos, que são dispostos linearmente. Portanto, uma lista linear agrupa informações referentes a um conjunto de elementos. Esses elementos são relacionados tais como:

Se n>0 l1 é o primeiro elemento

Ln é o último elemento

Para 0 < K < n o elemento Lk é precedido por Lk-1 e sucedido por Lk+1 cada elemento de uma lista é chamado de NÓ (NODO)

Operação com Listas:

É possível fazer:

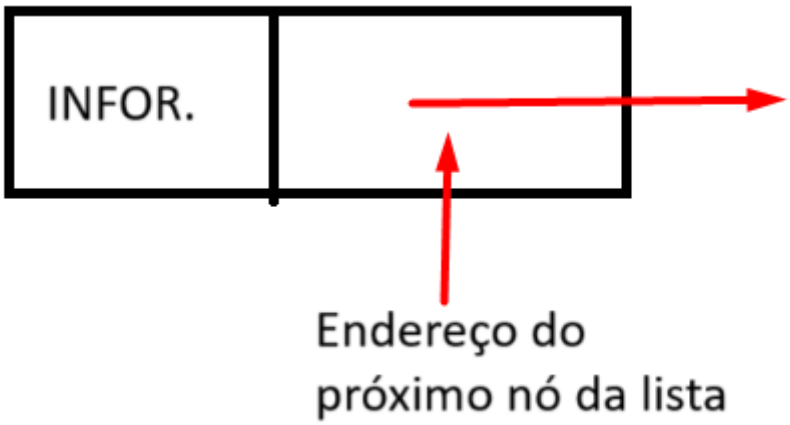
- Buscar um elemento
- Inserir/remover um elemento
- Alterar elementos
- Localizar o primeiro / último elemento
- Ordenar a lista
- Combinar duas ou mais listas

## Listas Encadeadas Dinâmicas

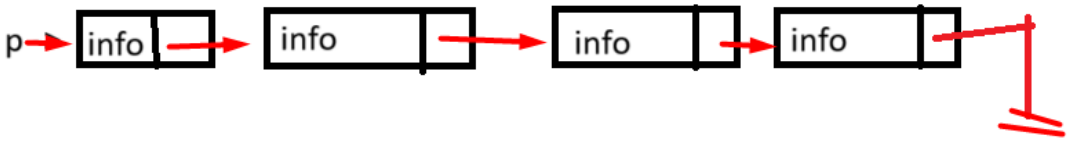
Uma lista encadeada dinâmica tem que cada elemento é chamado de nó. Cada nó é composto por, pelo menos dois campos:

- Um campo para informação
- Um campo com o endereço do próximo nó da lista

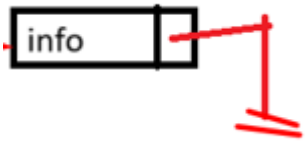
Representação do Nó:




Os nós estão ligados entre si pelo campo endereço e mostram o encadeamento




O campo endereço é chamado ponteiro para o próximo nó. A lista é acessada a partir do ponteiro P que aponta para o primeiro elemento da lista. O campo endereço do último nó da lista armazena **NULL** que indica o fim da lista.




**<- NULL**

Uma lista vazia é uma lista sem elementos e seu ponteiro **P** aponta para **NULL**.

## Estrutura do Nó da Lista

Cada nó da lista vai ser do tipo de uma estrutura (struct) criada pelo programador, para atender a necessidade da aplicação em desenvolvimento.

## Construindo a Estrutura do Nó

```
struct No{
        <tipo> info;
        struct No * prox;
}
```

<tipo> pode ser int, float, char, vetor, etc..

Struct No *p; => define o ponteiro para o início da lista

ou

typedef struct No lista;

Lista *p; => declaração de um nó da lista

## Alocação de Memória

P = (Lista *) malloc (sizeof(Lista));


P é um ponteiro para o nó da lista, portanto

p-> info permite acessa a info

p -> prox permite acessar o próximo elemento da lista

Lembre-se que o endereço e uma lista é o endereço do primeiro da lista.

## Lista Vazia

Lista *p;

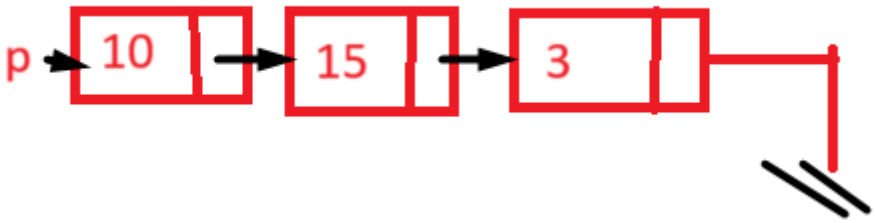
P = null;

Representação:

## Lista sem Cabeça

É uma lista cujo o primeiro nó é tão importante quanto os demais nós.

Exemplo:




## Lista com Cabeça

É uma lista, cujo conteúdo do primeiro nó é irrelevante, pois só serve para marcar o início da lista.


## Lista com Cabeça Vazia

Lista *p;

P = (Lista*) malloc (sizeof(Lista));

p->prox = null;

Representação:

**12/09/2023**

## Operações em Listas Encadeada

Pode-se realizar as seguintes operações em uma **LISTA ENCADEADA**

- Localizar um elemento
- Inserir/remover um elemento
- Criar uma lista vazia
- Alterar o conteúdo um nó
- Ordenar uma lista
- Inserir no Início/Meio/Fim da lista
- Remover do Início/Meio da lista

***Operação 2:*** Inserir no final da lista, para inserir um elemento no final da lista deve-se **VERIFICAR** se a lista está vazia. Neste caso o novo elemento será o primeiro nó da lista. Caso contrário deve-se localizar o final da lista usando um **PONTEIRO AUXILIAR** para depois inserir o **NOVO** elemento.

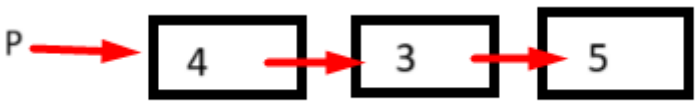
***Operação 3:*** Localizar um elemento da lista, essa operação consiste em percorrer a lista até encontrar a informação X, sendo x o valor que se deseja localizar na lista. Se o elemento pertencer a lista retornar o **ENDEREÇO DO NÓ** que o armazena. Caso contrário retornar **NULL.**

**EXEMPLOS:**

**1) Lista Vazia:**


**2) Elemento Pertence a Lista:**

**X = 3**



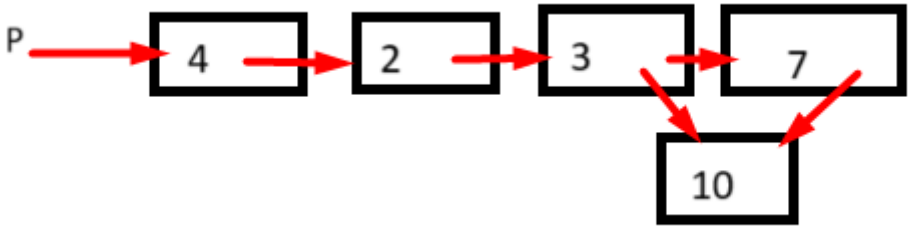

**3) Elemento não pertence a lista**

**X = 3**


***Operação 4:*** *Inserir um novo elemento na lista* ***APÓS*** *o* ***NÓ*** *que armazena a informação x. Essa operação consiste em localização o nó que armazena a informação x é inserir o novo nó após a posição de x. Se a lista estiver* ***VAZIA*** *ou x não pertencer a lista,* ***NÃO INSERIR.***

**EXEMPLO:** X pertence a lista

X = 3, Y = 10;




**26/09/2023**

## Busca e Inserção:

**Operações:** Inserir um novo nó na lista, em uma determinada posição por exemplo, ANTES do nó com conteúdo X. Se o no não existir, inserir no final da lista.

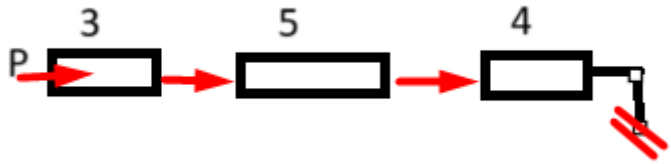
**EXEMPLO:**

**Caso 1:** Lista Vazia

X = 2.

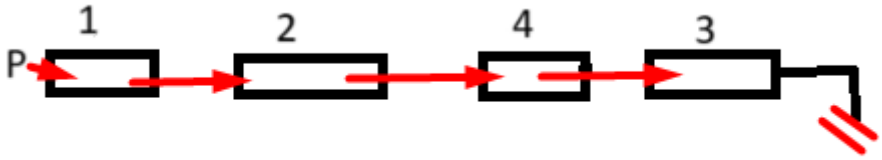

**Caso 2:** Elemento não pertence a lista.

X = 2.




**Caso 3:** Elemento pertence a lista.

X = 2




## Remoção de Elementos:

A operação de remoção consiste em remover um nó da lista considerando:

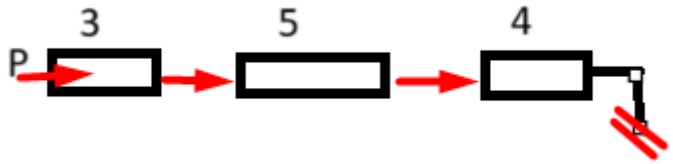
- Lista Vazia
- Elemento no início da lista
- Elemento no meio da lista
- Elemento no final da list
- Elemento não pertence a lista

**Operação 1:** Remover o primeiro elemento da lista. Nesta remoção, deve-se verificar se a lista está vazia, pois neste caso não se faz nada.

**EXEMPLO:**

**Caso 1**: Lista Vazia


**Caso 2:** Lista com elemento

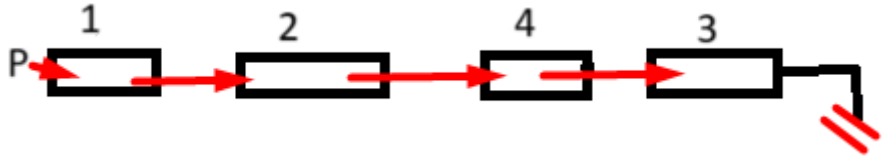



**Operação 2:** Remover o nó do fim da lista.

**Caso 1:** Lista Vazia


**Caso 2:** Lista com apenas um elemento


**Caso 3:** Lista com vários elementos




**Operação 3:** Remover o nó que armazena a informação X.

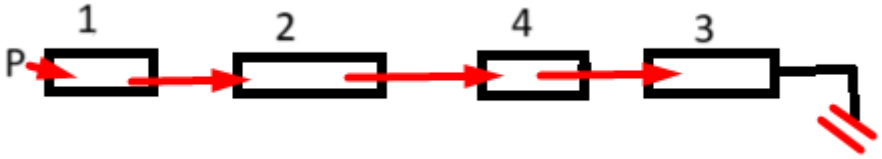
**Dificuldade:** Como determinar o nó que será removido?

**Problema:** Localizar o nó anterior a X, pois só percorremos pelo próximo. Não tem como “voltar” na lista.

**Solução:** Implementar um algoritmo de busca que retorna o nó anterior ao nó com a informação X.

**Caso 1:** Lista Vazia


**Caso 2:** Lista com Elementos onde E X não pertence a lista




**Caso 3:** Lista com Elementos onde E X pertence a lista