ESTRUTURA DE DADOS – EDAS2

Tecnologia em Análise e Desenvolvimento de Sistemas

2º. Semestre – 2019

Aula – Lista Simplesmente Encadeada

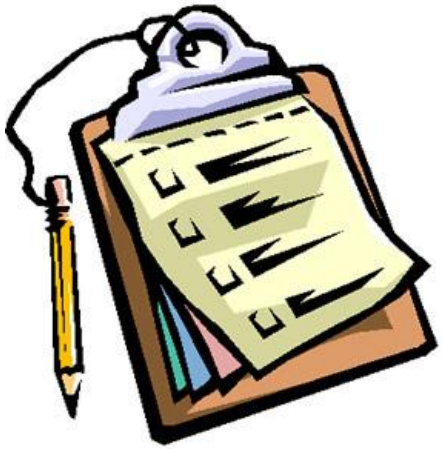
04/09/2019 e 06/09/2019

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Araraquara

Profa. Dra. Janaina Cintra Abib

Objetivo da Aula

- Compreender, estruturar e manipular um conjunto de dados organizados linearmente de forma dinâmica na memória primária, usando encadeamento simples.



Vamos pensar na Situação Problema:

Temos que organizar os dados de 5 alunos.

- Criamos uma estrutura ALUNO
- Criamos um vetor do tipo ALUNO
- Implementamos as operações

- **E SE... (busca + ordenação + quantidade)**

CONSIDERAÇÕES DA SOLUÇÃO

Conjuntos Dinâmicos

- Em computação é fundamental o trabalho com conjuntos de dados, que podem representar diferentes tipos de coleções como:
  - números,
  - dados de um ou vários funcionários,
  - dados de um ou vários produtos,
  - dados de um ou vários alunos, etc.
- Esses **conjuntos** são ditos **dinâmicos** porque os algoritmos que os manipulam fazem com que eles cresçam, diminuam ou sofram alterações ao longo do tempo.

- As principais operações sobre conjuntos dinâmicos são: inclusão, exclusão, alteração, buscas (várias formas), cálculos.

Lista Encadeada

- Uma estrutura de dados do tipo **lista** representa uma coleção de dados organizados em ordem linear.

Lista Encadeada

LISTA ESTÁTICA

- A lista é representada por um vetor, assim temos o uso de endereços contíguos de memória e a ordem linear é determinada por índices do vetor, o que exige um maior custo computacional.

LISTA DINÂMICA

- A lista é representada por elementos que possuem além do dado um ponteiro para o próximo elemento, ou seja, os elementos estão encadeados.
- Toda lista dinâmica possui pelo menos um ponteiro para o início.

Tipos de Listas

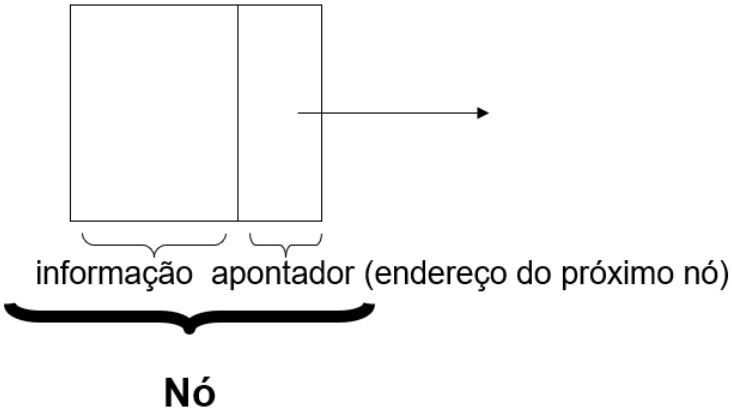
- **Simplesmente Encadeada**
- **Duplamente Encadeada**
- **Circular Simplesmente Encadeada**
- **Circular Duplamente Encadeada**
- **Com Descritor**

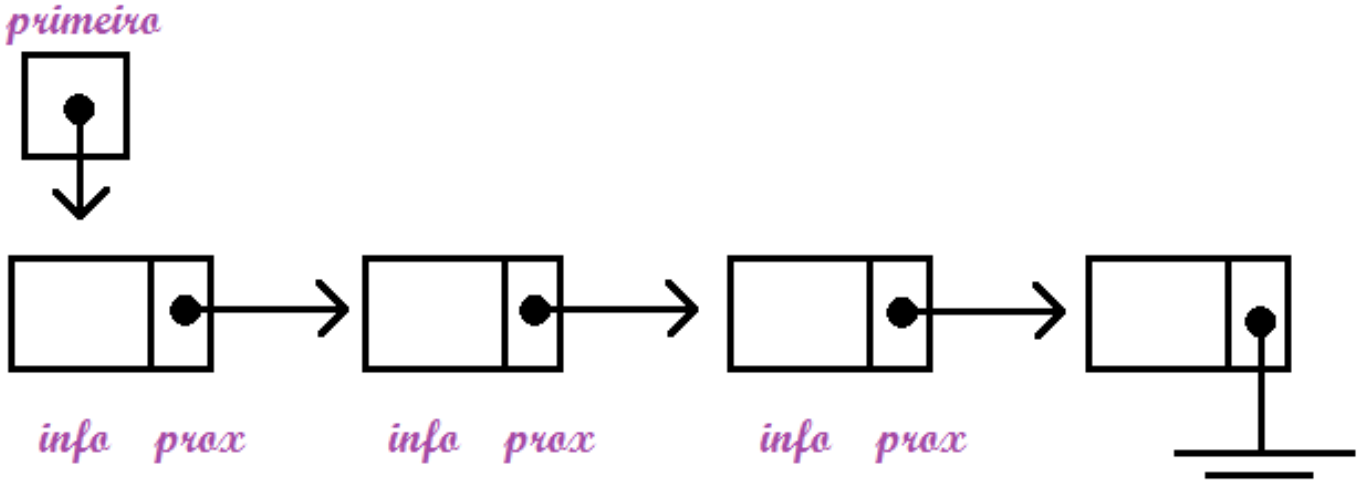
ORDENADA

NÃO ORDENADA

1. Lista Simplesmente Encadeada

```
struct no
{
        int info;
        struct no *prox;
};
```

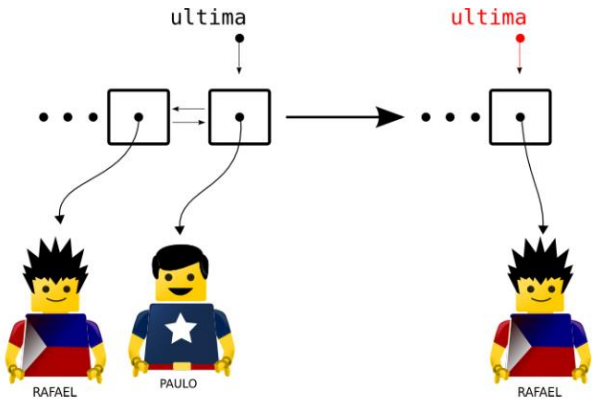




Lista Simplesmente Encadeada

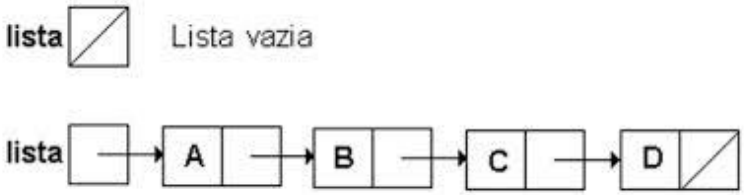
Não Ordenada

- Possui um ponteiro para o primeiro elemento (se este não existir temos uma lista vazia) e os elementos são inseridos no final ou no início da lista.



Ordenada

- É nomeada uma chave primária dentre os dados e os novos elementos são inseridos de forma a garantir a ordem pré-estabelecida da lista.



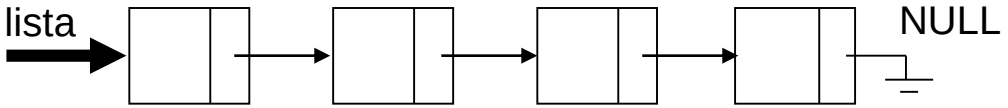
Implementações de Lista

- LSE não ordenada:
  - inserção no início da lista;
  - inserção no final da lista;
  - exclusão;
  - consulta;

- LSE ordenada:
  - inserção;
  - exclusão;
  - consulta;

Para simplificar nosso entendimento, neste primeiro momento vamos definir que nossa informação (INFO) será um valor inteiro, mas isso mudará em breve.

Representação da Lista Simplesmente Encadeada



- **Lista:**
- Possui os nós interligado, sendo que o ponteiro (apontador) do último nó da lista não aponta para nenhum outro nó e existe um **ponteiro Externo** a lista que aponta para seu primeiro nó.

- Todo nó da Lista só é acessível através do ponteiro externo.

- Quando um nó não puder ser acessado pelo ponteiro externo ele não mais pertencerá a Lista em questão.

- **Para facilitar vamos chamar os elementos do nó:**
- Campo informação: info
- Campo apontador (ponteiro): prox
- Ponteiro Externo: lista
- O prox do último nó aponta para NULL.

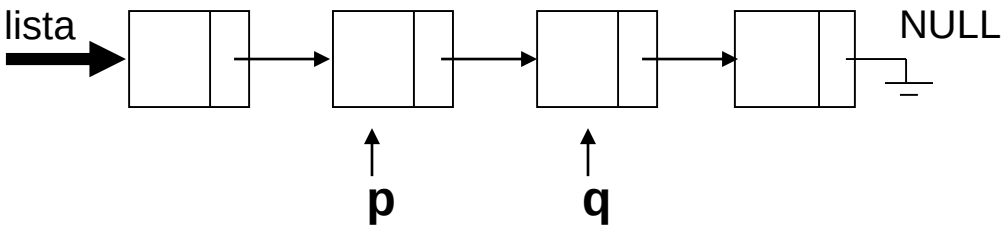
Lista Vazia
É a lista sem nós. Seu ponteiro externo (lista) aponta para NULL.

Podemos inicializar uma lista vazia pela operação:

```
lista = null
```

Notação a ser utilizada:
Se p é um ponteiro para um nó:
p: refere-se ao nó
p.info: refere-se à parte da informação desse nó
p.prox: refere-se à parte do endereço seguinte (também é um ponteiro)

Assim, dada a seguinte lista, onde:



Temos que:
p.prox é igual a q
(p.prox).info é igual a q.info

Criando a Lista Vazia

```c
struct no
{
    int info;
    struct no *prox;
};

typedef struct no LISTA;
```

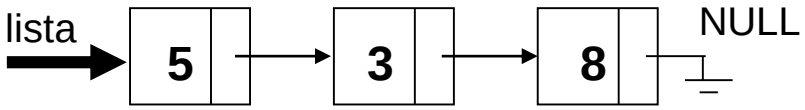

```c
void criarLista (LISTA **l)
{
    *l = NULL;
}

LISTA* criarLista2()
{
    LISTA *temp;

    temp = (LISTA*) malloc(sizeof(LISTA));

    if (temp != NULL)
    {
        temp = NULL;
    }
    return(temp);
}
```

Operações na Lista – **NÃO ORDENADA**

Inserindo nó no início da lista:
Relembrando: a lista agora é uma estrutura de dados dinâmica.

Assim sendo, o número de nós da lista pode variar consideravelmente à medida que elementos são inseridos e removidos.

Supondo que temos a seguinte lista de inteiros:



E queremos incluir o elemento 6 na **primeira** posição dessa lista (no início).

1º passo:
Obter um nó novo p/ armazenar o valor inteiro adicional.

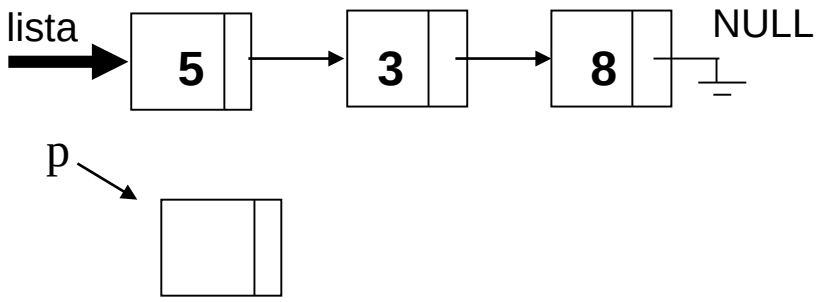
Para tanto é necessário um mecanismo para obter nós vazios novos a ser incluídos na lista. Admite-se a existência de um mecanismo para obter novos nós (criar) vazios.

A operação :
p = CriaNo

Obterá um novo nó vazio e definirá o conteúdo de uma variável chamada p com o endereço desse nó.

Dessa forma o valor de p é um ponteiro para esse nó recém-alocado.

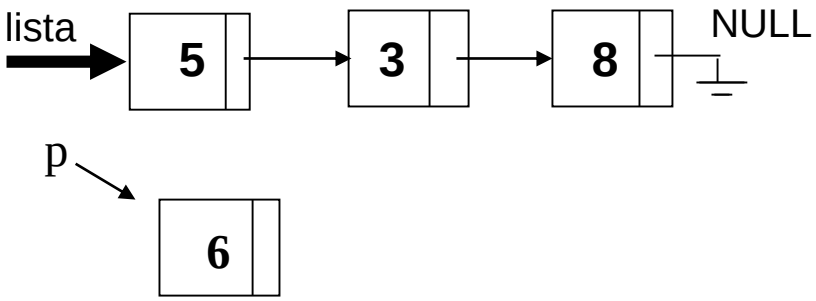
Depois de executar o primeiro passo, a nova configuração será:



2º passo:
Inserir o valor em questão na parte info do nó p.

Isso será feito através da operação:
p.info = 6

Depois de executar o segundo passo, a nova configuração será:

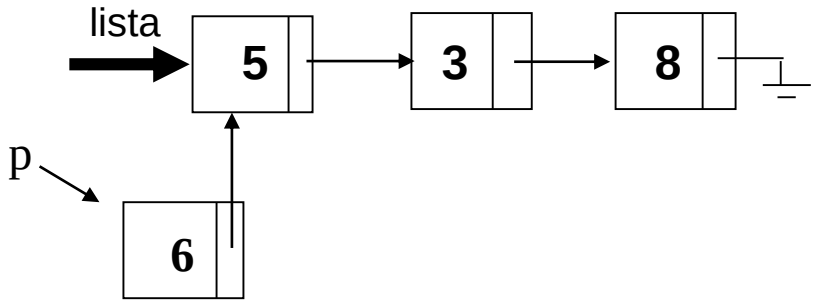


3º passo:
Fazer a parte prox do nó p apontar para o próximo nó da lista.

Como o nó p será inserido no início da lista a operação seguinte resolverá o problema:

p.prox = lista

A próxima configuração será:



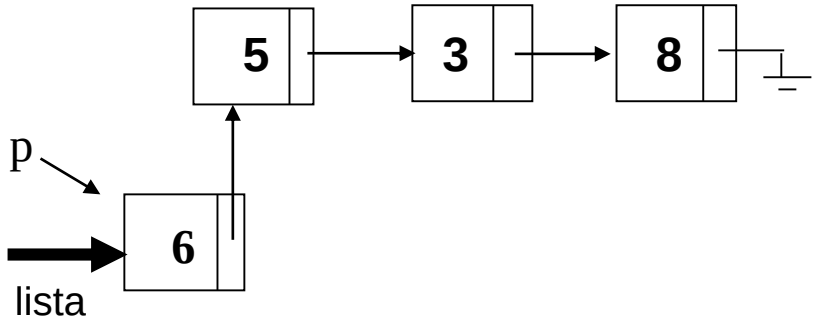
Neste momento o nó p ainda não pertence a lista pois, para pertencer a lista, um nó deve ser acessível pelo seu ponteiro externo, o que não se consegue fazer ainda com p.

4º passo:
Fazer o ponteiro do nó anterior apontar para o nó p.

Neste caso não existe nó anterior, mas existe um ponteiro anterior que é o próprio ponteiro externo a lista. Então a seguinte operação faz com que nó p seja incluído na lista:

lista = p;

Dessa forma a nova configuração será:



Resumindo os 4 passos, temos:

```
p =  CriaNo
p.info = 6;
p.prox = lista
lista =p
```

Generalizando o algoritmo abaixo para p.info receber qualquer valor, por exemplo, x:

```
p =  CriaNo
p.info = x;
p.prox = lista
lista =p
```

Logicamente, este algoritmo serve apenas para inserirmos um novo nó sempre na primeira posição.

Mas, com base nos 4 passos descrito s anteriormente podemos inserir um nó em qualquer posição de uma lista, mesmo se ela estiver vazia.

Relembrando os 4 passos:

- **Obter um nó novo p/ armazenar o valor inteiro adicional.**
- **Inserir o valor em questão na parte info do nó p.**
- **Fazer a parte prox do nó p apontar para o próximo nó da lista.**
- **Fazer o ponteiro do nó anterior apontar para o nó p.**

Inserindo no Início... Código

```
int inserirInicioLista1(LISTA **l, int valor)
{
    LISTA *temp;
    int resultado = TRUE;

    temp = malloc(sizeof(LISTA));

    if (temp == NULL)
       resultado = FALSE;
    else
    {

        temp->info = valor;

        temp->prox = *l;

        *l = temp;
    }

    return (resultado);
}
```

1º passo

2º passo

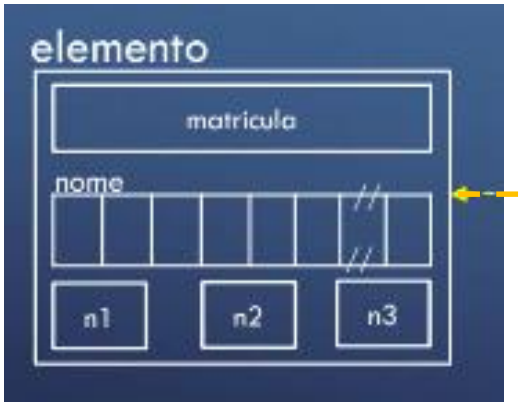
3º passo

4º passo

Mostrar Elementos da Lista

```
void escreverLista(LISTA *l)
{
      LISTA* temp;
      int cont = 0;
      temp = l;
      if (temp == NULL)
         printf("\n\nA lista esta vazia !");
      else
      {
          while(temp != NULL)
          {          cont++;
                     printf("Elemento %d = %d ", cont, temp->info);
                     temp = temp->prox;
          }
      }
}
```

Vamos Aplicar – LSE



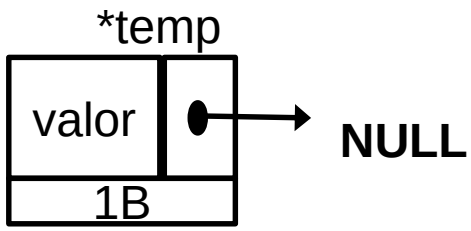

```c
struct elemento
{
    int matricula;
    char nome[30];
    float n1, n2, n3;
};

typedef struct elemento ALUNO;


struct no
{
    ALUNO info;
    struct no *prox;
};

typedef struct no LISTA;
```

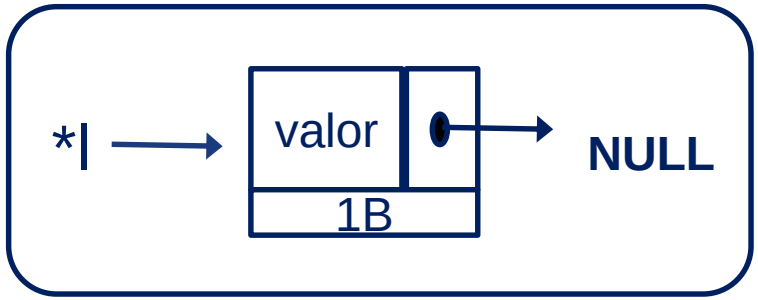
Agora é a sua vez!

- Considerando uma lista simplesmente encadeada e não ordenada, faça funções para
  - Inserir uma informação no final da lista;
  - Mostrar os valores da lista;
  - Consultar um valor da lista;
  - Remover um valor da lista;
  - Contar quantos nós tem a lista.
  - INSERÇÃO EM LISTA ORDENADA

INSERIR NO FIM 1/3

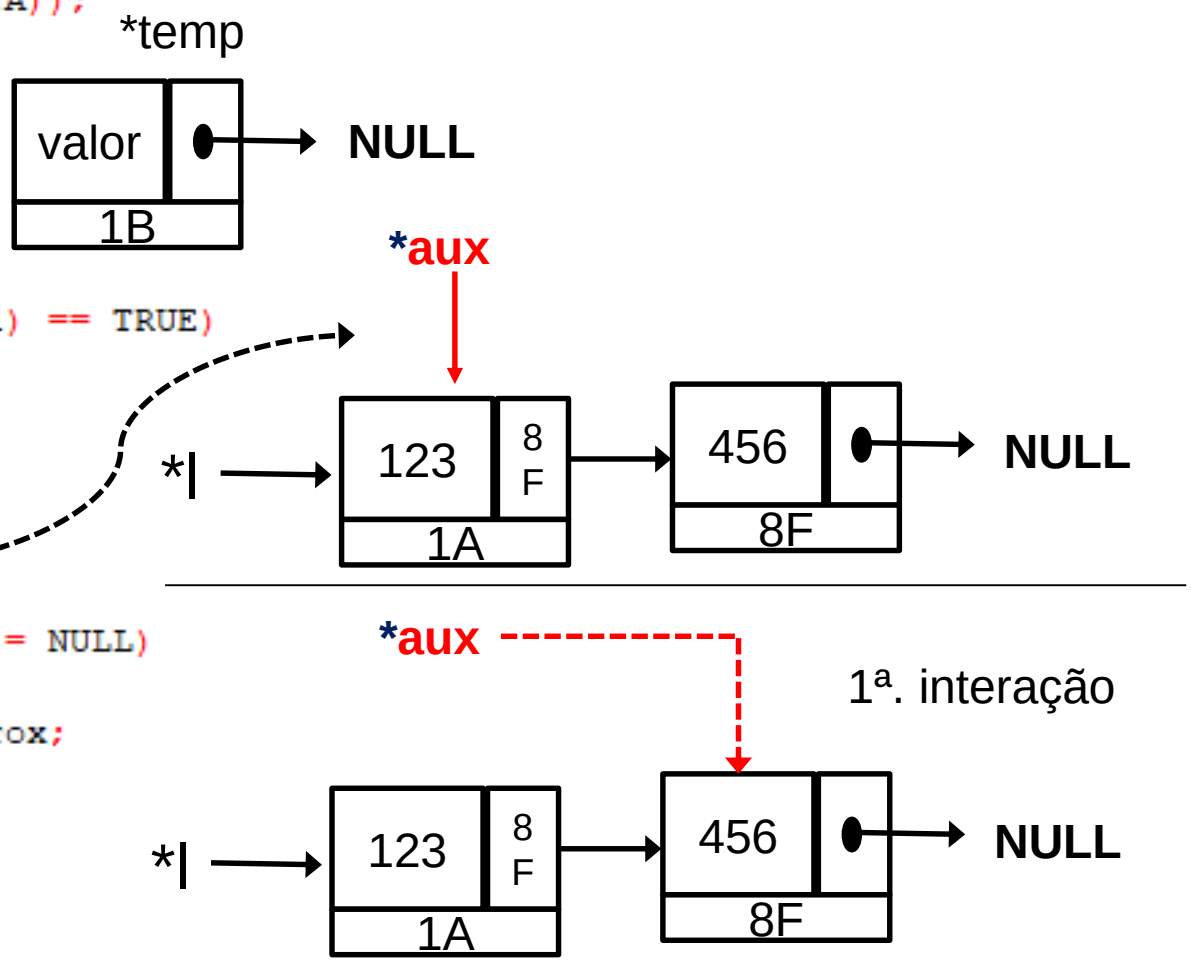
```
int inserirFimLista(LISTA **l, int valor)
{
    LISTA *temp;
    LISTA *aux;
    int resultado = TRUE;
    temp = malloc(sizeof(LISTA));
    if (temp == NULL)
       resultado = FALSE;
    else
    {
        temp->info = valor;
        temp->prox = NULL;
        if (verificarVazia(*l) == TRUE)
        {
            *l = temp;
        }
        else
        {
            aux = *l;
            while(aux->prox != NULL)
            {
                aux = aux->prox;
            }
            aux->prox = temp;
        }
    }
    return (resultado);
}
```





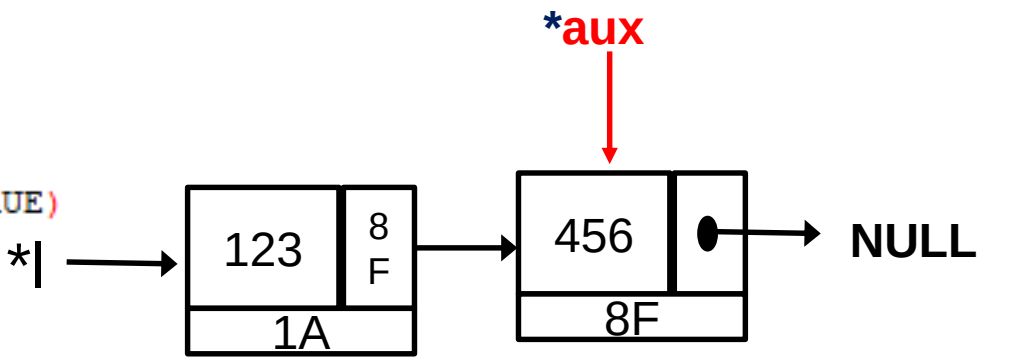
INSERIR NO FIM 2/3

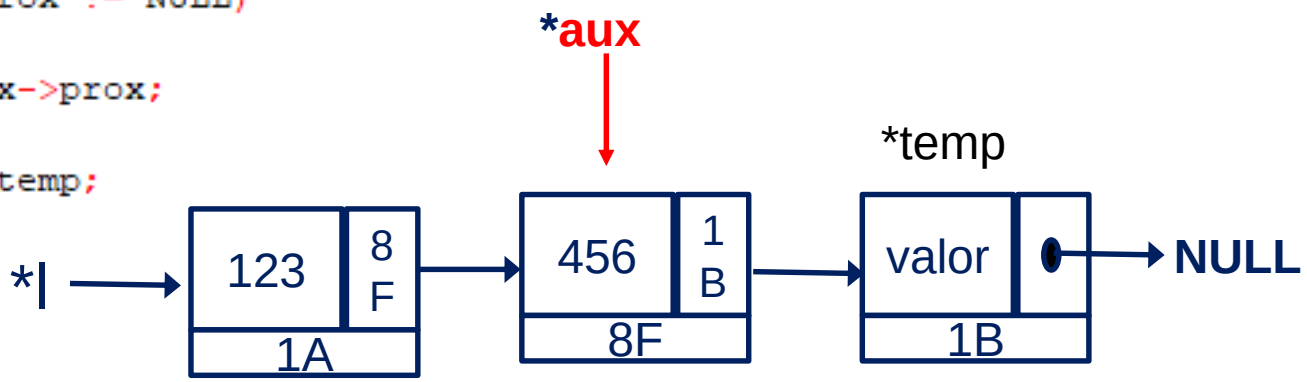
```
int inserirFimLista(LISTA **l, int valor)
{
    LISTA *temp;
    LISTA *aux;
    int resultado = TRUE;
    temp = malloc(sizeof(LISTA));
    if (temp == NULL)
       resultado = FALSE;
    else
    {
        temp->info = valor;
        temp->prox = NULL;
        if (verificarVazia(*l) == TRUE)
        {
            *l = temp;
        }
        else
        {
            aux = *l;
            while(aux->prox != NULL)
            {
                aux = aux->prox;
            }
            aux->prox = temp;
        }
    }
    return (resultado);
}
```



INSERIR NO FIM 3/3

```
int inserirFimLista(LISTA **l, int valor)
{
    LISTA *temp;
    LISTA *aux;
    int resultado = TRUE;
    temp = malloc(sizeof(LISTA));
    if (temp == NULL)
       resultado = FALSE;
    else
    {
        temp->info = valor;
        temp->prox = NULL;
        if (verificarVazia(*l) == TRUE)
        {
            *l = temp;
        }
        else
        {
            aux = *l;
            while(aux->prox != NULL)
            {
                aux = aux->prox;
            }
            aux->prox = temp;
        }
    }
    return (resultado);
}
```





VERIFICAR SE LISTA ESTÁ VAZIA
CONSULTAR ELEMENTO NA LISTA

```
int verificarVazia(LISTA *l)
{
    int resultado = FALSE;

    if (l == NULL)
    {
        resultado = TRUE;
    }
    return(resultado);
}
```

```
int consultarLista(LISTA *l, int valor)
{
    LISTA* aux;
    int resultado = FALSE;

    aux = l;

    if (aux != NULL)
    {
        while(aux != NULL)
        {
            if(aux->info == valor)
            {
                resultado = TRUE;
            }
            aux = aux->prox;
        }
    }
    return(resultado);
}
```

RECURSÃO
MOSTRAR ELEMENTO DA LISTA

```
void imprimirRec (LISTA *l)
{
    if (verificarVazia(l) == FALSE)
    {

        printf("Valor: %d \t",l->info);


        imprimirRec(l->prox);
    }

}
```

Que confusão!
Como fica isso na memória?

INSERIR ORDENADO

```
int inserirOrdenado(LISTA **l, int valor)
{
    LISTA *novo, *atual, *anterior;
    int resultado = TRUE;

    atual = *l;
    anterior = NULL;

    novo = (LISTA *) malloc(sizeof(LISTA));
    if(novo == NULL)
    {
        resultado = FALSE;
    }
    else
    {
        novo->info = valor;
        novo->prox = NULL;

        while((atual != NULL) && (valor > atual->info))
        {
            anterior = atual;
            atual = atual->prox;
        }

        if((*l == NULL) || (atual == *l))
        {
            novo->prox = *l;
            *l = novo;
        }
        else
        {
            novo->prox = atual;
            anterior->prox = novo;
        }
    }
    return(resultado);
}
```

// Aloca memória para novo NÓ

// Memória OK

// Guarda valor e acerta o ponteiro do novo NÓ

// Busca posição correta para inserir

// Inserção no **início** ou lista vazia

// Acerta ponteiros para colocar NÓ novo na lista

Exercício: Outras Funções LSE

- Implementar a função que apaga um determinado nó da lista em função de sua **posição** X.

  int apagarAtLista(Lista **, int x);

- Implementar uma função que compara o tamanho de duas listas.

  int compararLista(Lista *listaA, Lista *listaB);

  0 -> mesma quantidade de nós;
  1 -> listaA mais nós que listaB;
  2 -> listaB mais nós que listaA;

- Implementar uma função que copia (clone) uma lista.

  int copiarLista(Lista ** destino, Lista *origem);