MAUÁ ESCOLA DE ENGENHARIA

12

Alocação Dinâmica e Listas Ligadas

Conteúdo 3º Bimestre

- **Tipos Abstratos de Dados (TAD) – Implementações genéricas de comportamentos que podem ser aplicadas para qualquer tipo de dados, variando em função do problema modelado.**

- **Tipos de TAD estudados no bimestre:**
  - Lista;
    - Pilha;
    - Fila;
  - Grafo:
    - Árvore.

A memória R.A.M.

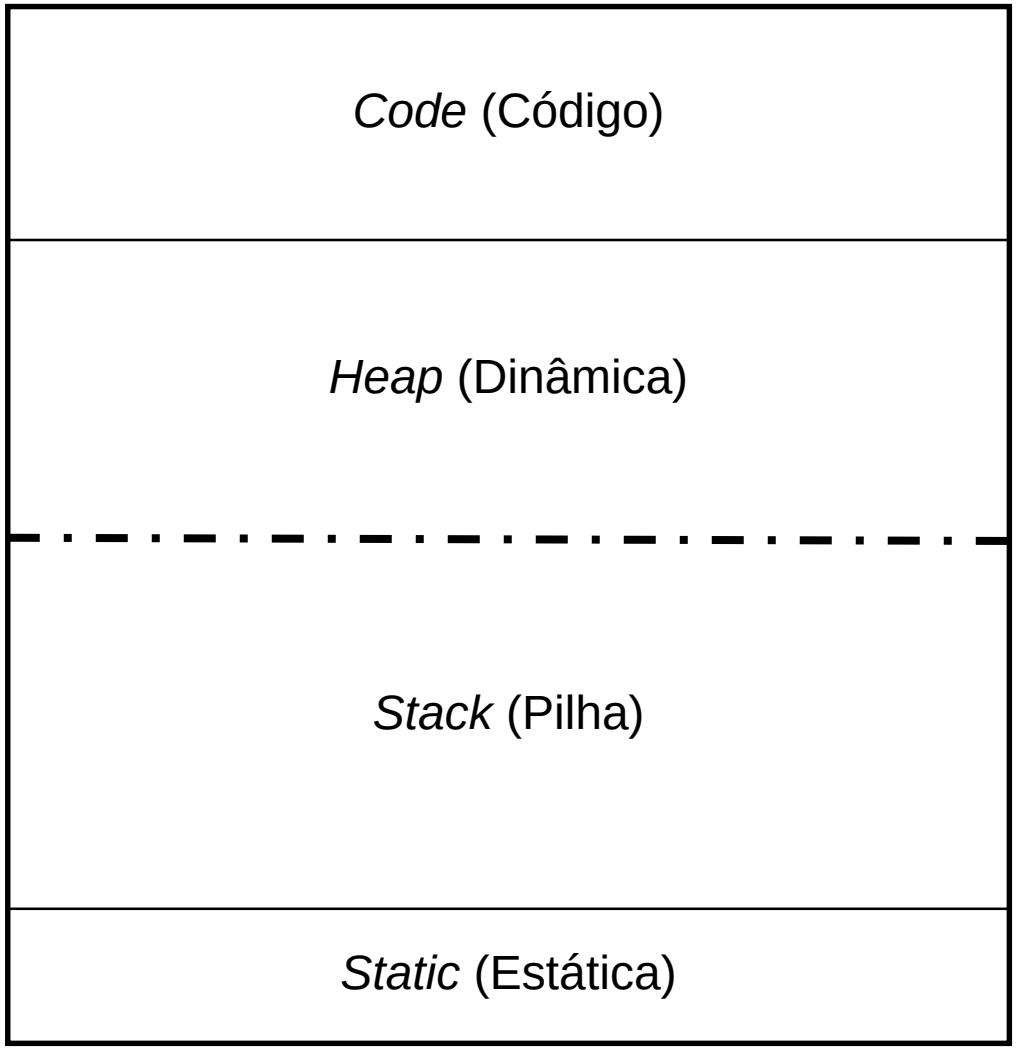
- A memória R.A.M. (*Random Access Memory*) é a memória controlada diretamente pelo microprocessador;

- nela os programas, com suas instruções em linguagem de máquina e dados, são carregados ao serem executados;

- para gerenciar de forma organizada a memória R.A.M. o microprocessador a divide em 4 áreas:

A divisão da Memória R.A.M.

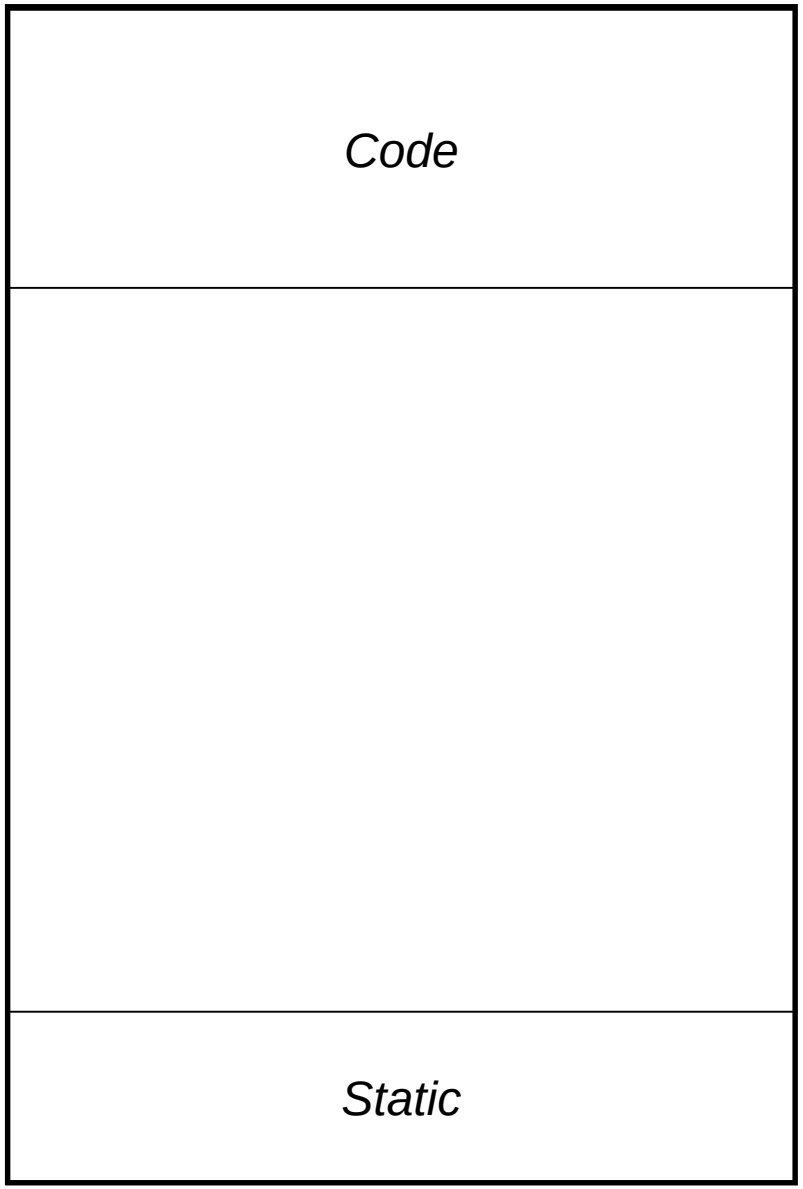
Executando um programa



A divisão da Memória R.A.M.

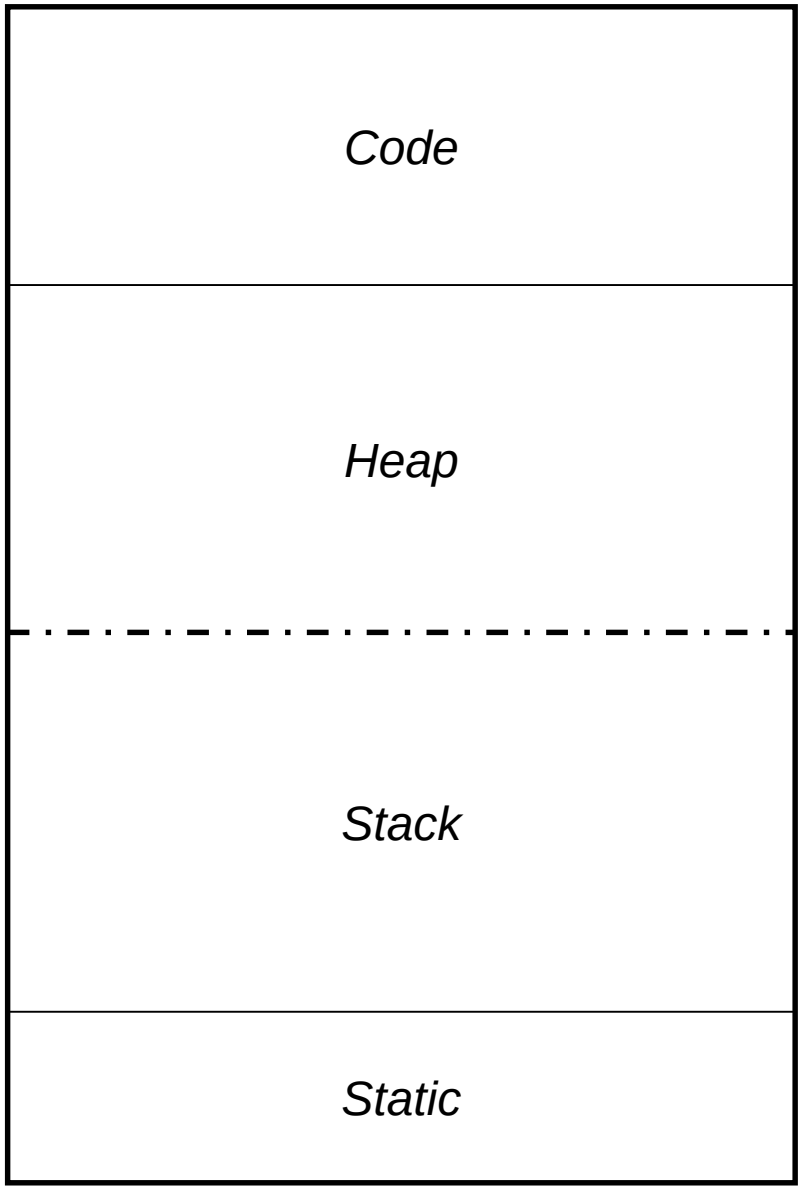
- ***Code***: região da memória destinada ao armazenamento das instruções do **programa**, assim que o programa é executado.

- ***Static***: após o programa ser carregado, as **variáveis globais** e as **constantes** são alocadas na memória estática.



A divisão da Memória R.A.M.

- ***Stack***: utilizada para armazenar informações sobre as **sub-rotinas**, seus pontos de retorno e variáveis locais.
- ***Heap***: aqui é feita a alocação dinâmica (em tempo de execução).
- Não existe um limite físico entre a *Stack* e a *Heap*.



Durante a execução

O processador começa a

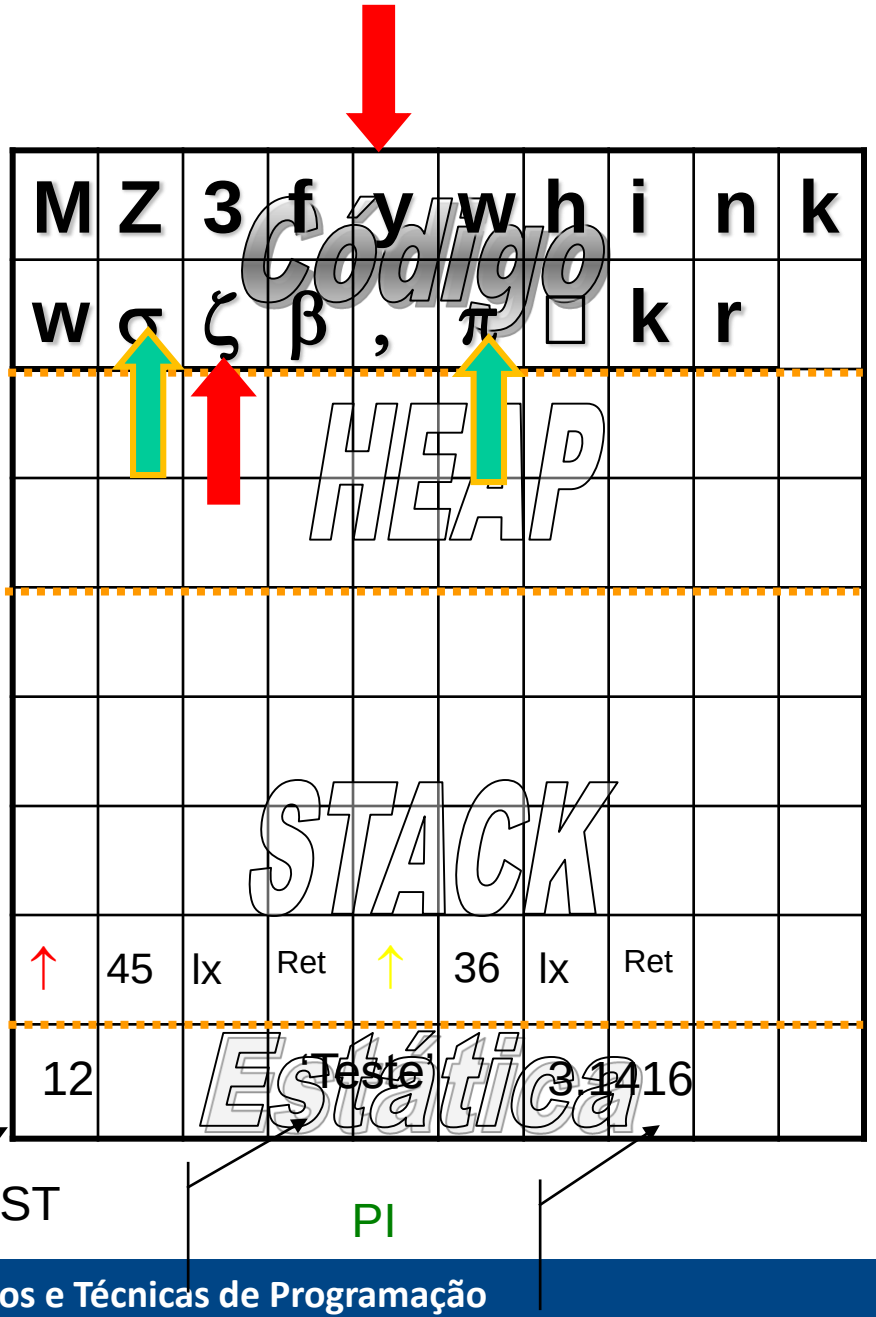
executar o Programa Principal

É feita a chamada de uma sub-rotina

Reserva-se no STACK lugar para informações necessárias: variáveis locais, valores de parâmetros, endereço de retorno...

A sub-rotina chama outra...

Reserva-se no STACK lugar para informações necessárias: variáveis locais, valores de parâmetros, endereço de retorno...



Alocação Dinâmica de Memória

- **Qual a solução quando a quantidade de posições de memória para implementar uma solução é desconhecida? A declaração de vetores com tamanhos grande pode representar um desperdício de memória ou ainda sua falta em algumas situações.**
- **A *alocação dinâmica* possibilita alocar memória em tempo de execução do código, criando um ponteiro para um vetor com o tamanho desejado na memória *heap*.**

Alocação Dinâmica de Memória

A alocação dinâmica na linguagem C é realizada utilizando a função *malloc*(). Ela retorna um ponteiro para a primeira posição de memória do bloco alocado.

Protótipo:

void ***malloc** (**unsigned int** *num*);

Observação: como a função retorna um ponteiro do tipo ***void***, seu valor deve ser convertido para um ponteiro do tipo utilizado no código.

Alocação Dinâmica de Memória

Exemplo:

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

int main()
{
  int n, i, *dados;
  printf("Informe o tamanho do vetor: ");
  scanf("%i", &n);
  dados = (int*) malloc(n * sizeof(int));
  for (i = 0; i < n; i++)
     scanf("%i", (dados+i) );
  for (i = 0; i < n; i++)
     printf("Elemento%i: %i\n",i, *(dados+i) );
  free(dados);
  return 0;
}
```

Alocação Dinâmica de Memória

- **A função *sizeof*() retorna o tamanho em bytes de um tipo fornecido como parâmetro. É utilizada para determinar a quantidade de memória necessária para alocação.**

- **Enquanto que a função *free*() devolve para o sistema a memória tomada durante a alocação dinâmica. Quando um elemento alocado de forma dinâmica não for mais utilizado, a memória deve ser liberada utilizando a função *free*().**

Alocação Dinâmica de Memória

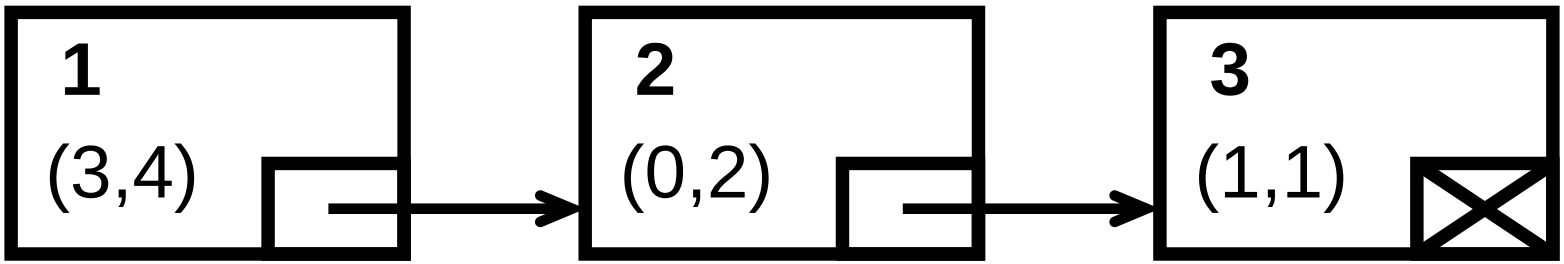
Exemplo:

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

int main(){
    int n, i;
    float s = 0;
    float *x;
    printf("Digite o numero de elementos: ");
    scanf("%i",&n);
    x = (float *) malloc(n*sizeof(float));
    /* sempre é possível alocar essa quantidade de memória? */
    for (i=0; i<n; i++){
        printf("Digite o elemento x[%i]: ", i);
        scanf("%f",&x[i]);
        s = s + x[i];
    }
    printf("A media vale: %4.2f\n", s/n);
    free(x);
    return 0;
}
```

Listas - Definição

- **Listas são estruturas *dinâmicas* de dados (utilizam alocação dinâmica de memória) e permitem manipular seus elementos (inserir, remover, buscar, ordenar etc.).**

- **A título de comparação, as *variáveis indexadas* (vetores, matrizes) são estruturas de dados estáticas porque o número de elementos reservados efetivamente não pode ser alterado durante a execução do programa.**



Resumindo . . .

Variáveis indexadas estáticas

Sempre ocupam a mesma quantidade de memória;

Variáveis indexadas dinâmicas

Ocupam a quantidade de memória necessária;

Podemos inserir ou remover elementos que estão nas últimas posições;

Listas Ligadas

Ocupam a quantidade de memória necessária;

Podemos inserir ou remover elementos em qualquer posição.

A criação dinâmica da estrutura dinâmica

- **Estabelecer ligações entre as informações armazenadas na memória;**
- **uma informação precisa saber o endereço da outra;**
- **as informações são alocadas dinamicamente na *Heap*;**
- **primeiro elemento (cabeça):**
  - não possui nenhum dado referente ao conteúdo da lista;
  - facilita as operações com a lista, embora não seja obrigatória sua criação;
- **são facilmente criadas quando utilizamos structs.**

A criação dinâmica da estrutura dinâmica

Definição do tipo (**TLista**) para os nós da lista:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
typedef struct {
     float valor;
     char texto[MAX];
} Dados;

typedef struct SLista {
     Dados dados;
     struct SLista *prox;
} TLista;
```

*TLista = **struct** SLista*

*Ponteiro para outro **nó** da lista*

*Tipo de dados para os **nós** da lista*

Código para criação da lista

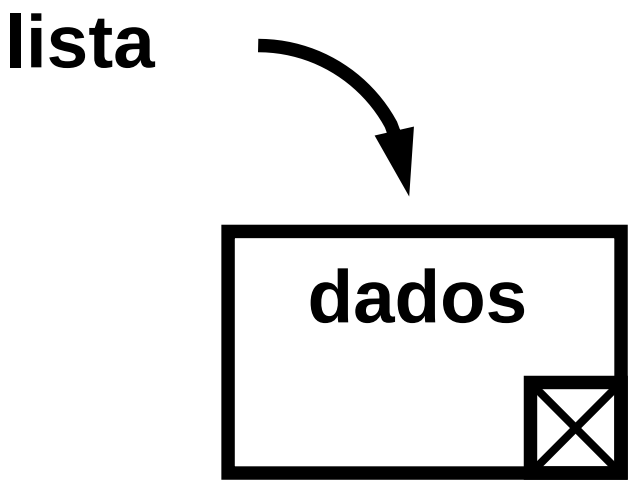
```
int main() {
    char op;
    Dados dados;
    TLista *lista;
    lista = CriarLista();
    do {
        printf("valor: ");
        scanf("%f",&dados.valor);
        printf("texto: ");
        LerString(dados.texto);
        InserirNoFim(lista, dados);
        printf("Mais itens (S/N)? ");
        scanf("%c", &op);
    } while (toupper(op) != 'N');
    ExibirLista(lista);
    lista = DestruirLista(lista);
    return 0;
}
```

Lembrando . . .

```
void LerString(char s[])
{
   setbuf(stdin, 0);
   // ou fflush(stdin);
   fgets(s, MAX, stdin);
   if (s[strlen(s)-1] == '\n')
       s[strlen(s)-1] = '\0';
}
```

Elemento Cabeça

Para facilitar a manipulação da lista será usada uma **célula-cabeça**, apontada por um ponteiro:



O cabeça não serve para armazenar informações, mas sim, para indicar a "existência" da lista, a partir de um certo endereço de memória...

- No programa principal, o ponteiro ***lista*** sempre apontará para o cabeça.

Criação da Lista

Chamada da função ***CriarLista***:

```
lista = CriarLista();
```

Ações:

1) Alocar memória para o cabeça;

2) Alocação OK? indicar lista vazia (ponteiros);

3) Retornar o endereço do elemento cabeça para o programa principal.

Protótipo: `TLista* CriarLista(void);`

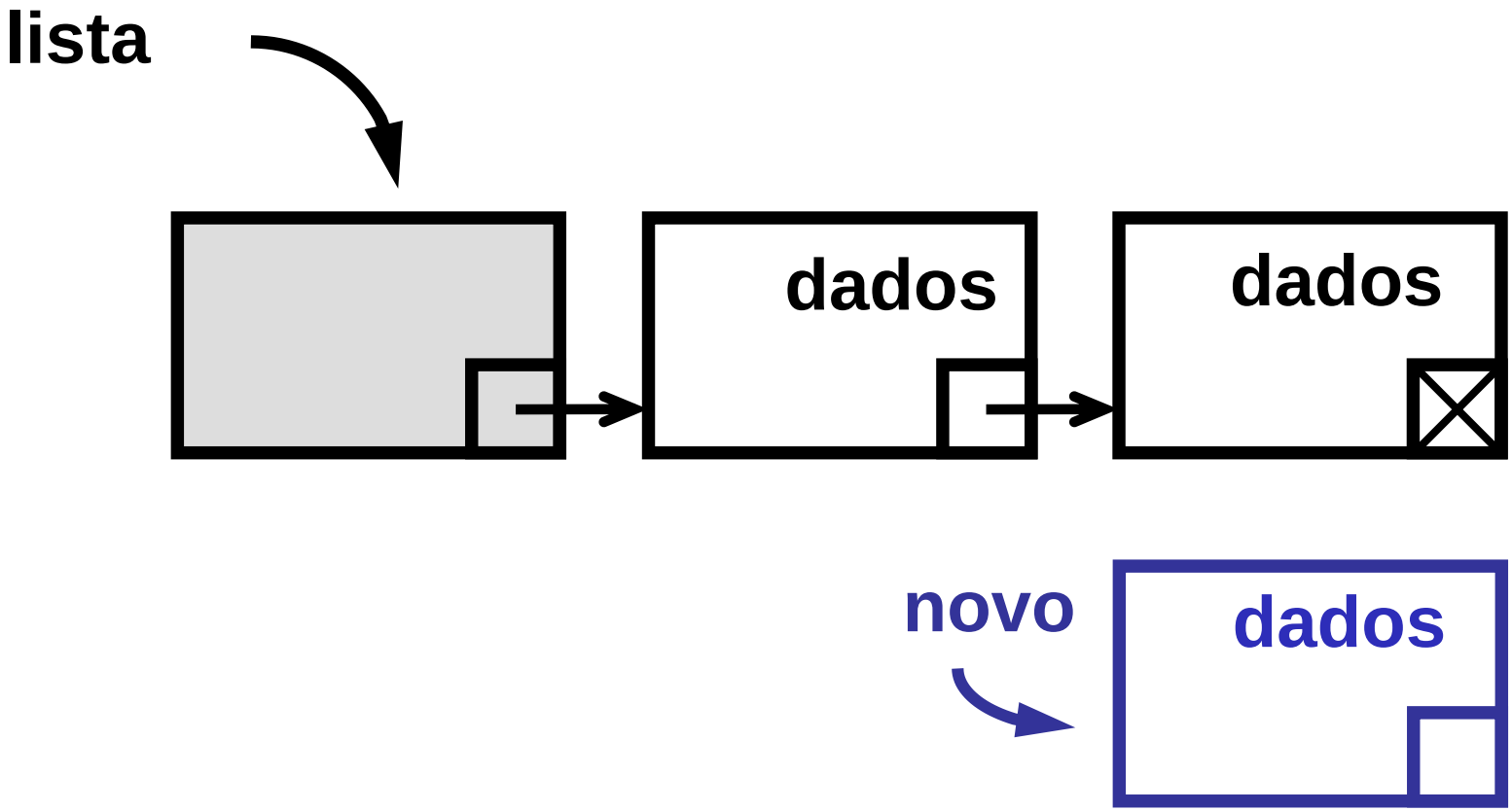
Retorno: endereço da célula-cabeçalho

Criação da Lista

```
TLista* CriarLista(void) {
  TLista *p;    /* Ponteiro para a lista */
  p = (TLista*) malloc(sizeof(TLista));
  if (p == NULL){
    printf("Não pode criar a lista");
    exit(EXIT_FAILURE);
  }

  p->prox = NULL;    /* Atribui ponteiro nulo */
  return p;    /* Retorna endereço da lista */
}
```

Inserção na Lista

- A inserção será feita sempre no final da lista (**FILA**).
- A função de inserção deve receber o endereço do elemento cabeça e do novo nó a ser inserido (“criado” previamente).



Inserção na Lista

No programa principal...

```
InserirNoFim(lista, dados);
```

Ações:

1) Encontrar o final da lista (*loop*);

2) Do último nó, apontar para o novo nó a ser inserido;

3) Encerrar a lista no nó inserido apontando-o para NULL.

Protótipo: `void InserirNoFim(TLista*, Dados);`

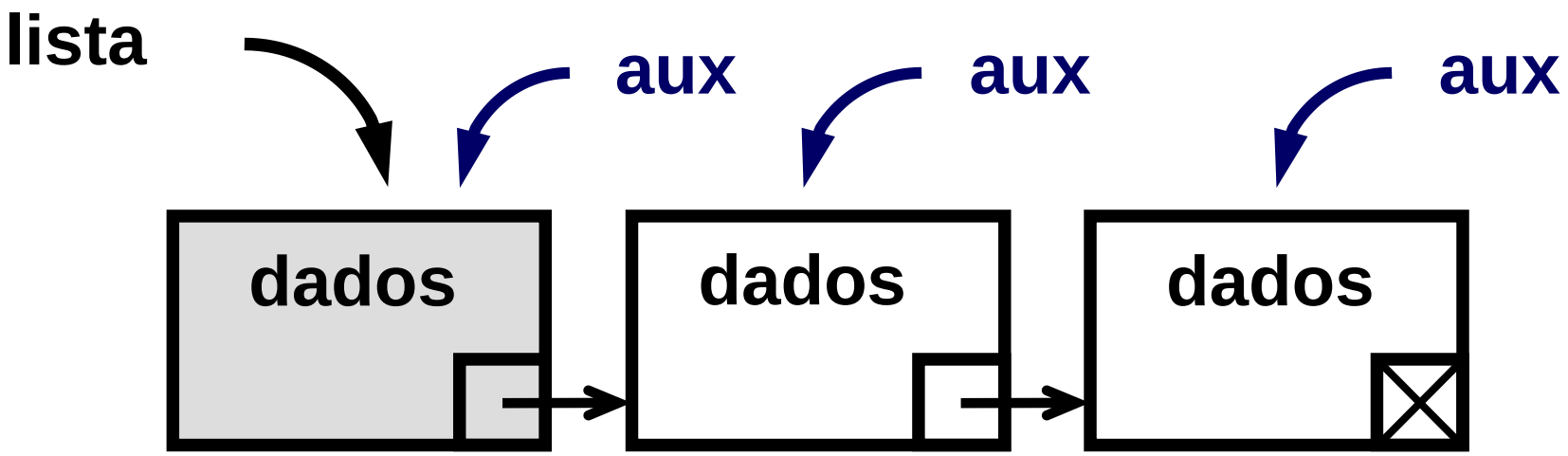
Endereço da célula-cabeçalho

Endereço do novo nó

Inserção na Lista

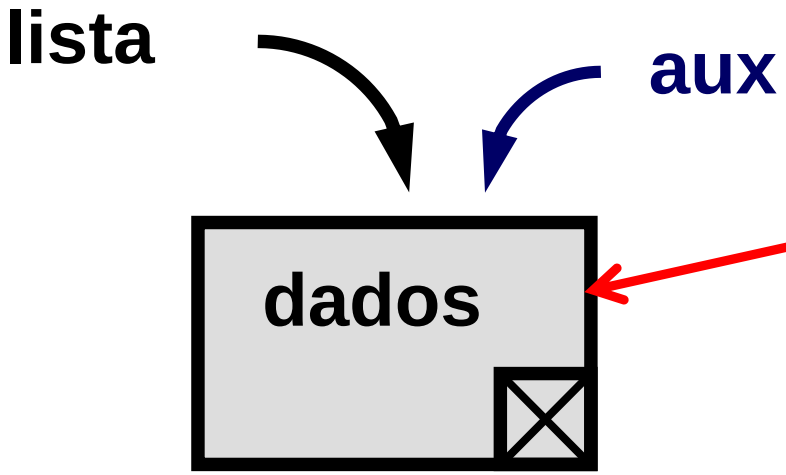
```
void InserirNoFim(TLista *p, Dados dados) {
    TLista* novo;
    novo = (TLista*) malloc(sizeof(TLista));
    if (novo == NULL) {
        printf("Nao foi possivel alocar memoria!");
        exit(EXIT_FAILURE);
    }

    novo->dados = dados;

    TLista *aux;
    aux = p;
    while(aux->prox != NULL)
        aux = aux->prox;

    novo->prox = NULL;
    aux->prox = novo;
}
```

Exibição da Lista



Para avançar...

```
aux = aux->Proximo;
```



Lista vazia:
aux->Proximo == NULL

Exibição da Lista

Chamada da função ***ExibirLista***, no programa principal:

```
ExibirLista(lista);
```

Ações (se a lista não estiver vazia):

1) Apontar para o 1º nó com informação;

2) Exibir informações;

3) Avançar para o próximo nó.

} Repetir, até exibir as informações do último nó...

Protótipo: `void ExibirLista(TLista*);`

Endereço da célula-cabeçalho

Exibição da Lista

```
void ExibirLista(TLista *p) {
    TLista *aux;
    aux = p;
    aux = aux->prox;
    while(aux != NULL) {
        printf("%4.2f\t%s\n",
                    aux->dados.valor,
                    aux->dados.texto);
        aux = aux->prox;
    }
}
```

Destruição da Lista

```
TLista* DestruirLista(TLista *p) {
    Tlista *aux;
    aux = p;
    while(aux->prox != NULL) {
        aux = aux->prox;
        free(p);
        p = aux;
    }
    free(p);
    return NULL;
}
```