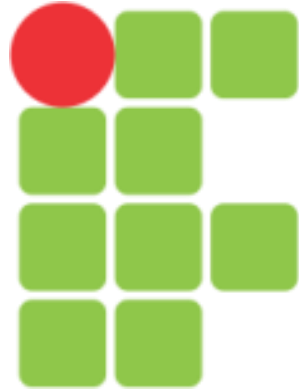




INSTITUTO FEDERAL  
CEARÁ



INSTITUTO FEDERAL  
CEARÁ



**Ernani Andrade Leite**

[ernani@ifce.edu.br](mailto:ernani@ifce.edu.br)



# Aula 04 – Estruturas de Dados

---

❑ Assunto: Ponteiros.

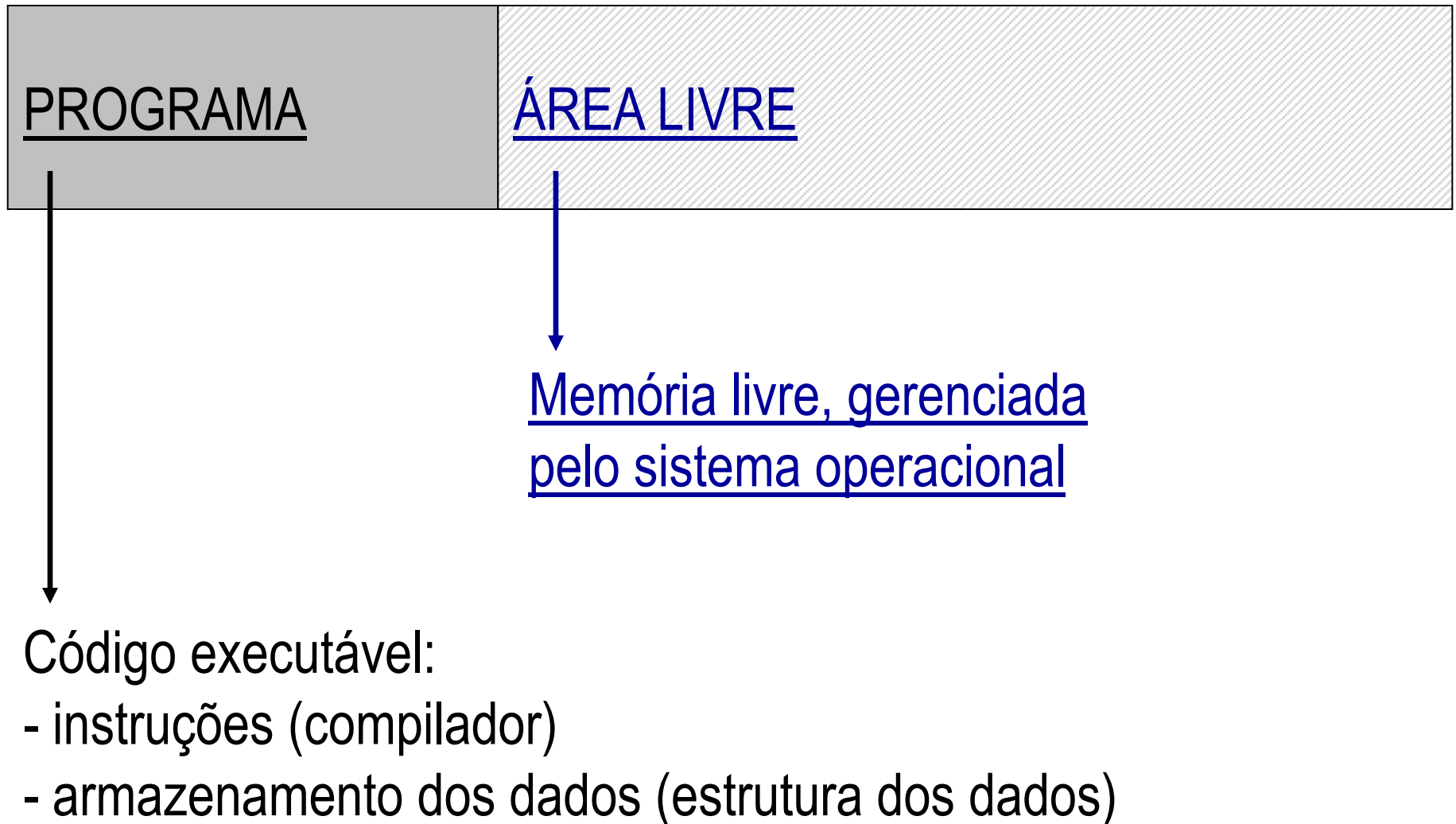
❑ Objetivos:

- Apresentar os conceitos elementares de ponteiros e sua aplicação no cotidiano;
- Aplicação de algoritmos em situações do dia-a-dia;
- Elaborar programas usando ponteiros.

❑ Roteiro:

- |                                 |  |
|---------------------------------|--|
| 1. Introdução.                  | 2. Definições e Conceitos.               |
| 3. Alocação Estática X Dinâmica | 4. Operando Ponteiros.                   |
| 5. Variáveis Dinâmicas.         | 6. Atribuindo um endereço a um Ponteiro. |

# Memória Disponível no Computador





# Alocação de Memória (1/2)

---

## ☐ Estática

- ☐ Quantidade total de memória utilizada pelos dados é previamente conhecida e definida de modo imutável, no próprio código-fonte do programa.
- ☐ Durante toda a execução, a quantidade de memória utilizada pelo programa não varia.
- ☐ Lista de variáveis declaradas.



# Alocação de Memória (2/2)

---

## ☐ Dinâmica

- ☐ Quanto o programa é capaz de criar novas variáveis enquanto está sendo executado.
- ☐ Alocação de memória para componentes individuais no instante em que eles começam a existir durante a execução do programa.
- ☐ malloc: *alocar* memória.
- ☐ free: *liberar* áreas da memória ocupadas.



# Alocação Estática

# X

# Alocação Dinâmica



# Alocação Estática (1/3)

---

- ❑ implementação simples: vetores (array)

- ❑ vantagem:

- ❑ acesso indexado ( $v_i$ )- todos os elementos da estrutura são igualmente acessíveis

- ❑ desvantagens:

- ❑ tamanho fixo: #define maxTam 1000

- ❑ tempo de compilação

- ❑ alocados em memória de forma estática



# Alocação Estática (2/3)

```
#define maxTam 1000
```

```
struct rgCliente {
```

```
    char nome[40];
```

```
    char sexo;
```

```
    int idade;
```

```
};
```

```
struct rgCliente Cliente[maxTam];
```

Neste exemplo são reservadas, durante toda a execução do programa, 1.000 posições para o vetor

**Cliente.**

Será que um sistema de cadastro de clientes, que usa um vetor com 1.000 posições é o suficiente ?

Será que nunca irá acontecer a necessidade de se cadastrar o cliente de número 1.001 ?

E o que acontece quando os clientes cadastrados nunca passarem de 100 ? As 900 posições de memória restantes, não poderão ser utilizadas por outras variáveis, pois já estão reservadas.





# Alocação Estática (3/3)

---

- ❑ Ao se determinar o máximo de elementos que o vetor irá conter, pode-se ocorrer um dos seguintes casos:
- ❑ **subdimensionamento**: haverá mais elementos a serem armazenados do que o vetor é capaz de conter;
- ❑ **superdimensionamento**: na maior parte do tempo, somente uma pequena porção do vetor será realmente utilizada.

# Alocação Dinâmica

---

- ☐ implementação eficiente: **ponteiros** ou **apontadores**
- ☐ vantagens:
  - ☐ tamanho variável
  - ☐ tempo de execução
  - ☐ alocados em memória de forma dinâmica
- ☐ desvantagem, ou restrição:
  - ☐ capacidade da memória, acesso sequencial

# Variável = endereço de memória

---

Área de memória onde dados são armazenados:

- ☐ variável estática (lista de variáveis)
  - ☐ existência prevista no código do programa (tempo de compilação)
  - ☐ quantidade de memória utilizada pelo programa não varia
  
- ☐ variável dinâmica (malloc, free)
  - ☐ passam a existir durante a execução do programa



# Variáveis Dinâmicas (ponteiros)

---

- ❑ ponteiros, ou apontadores, ou indicadores, ou referências
- ❑ variáveis especiais que armazenam endereços de memória, isto é, endereços de outras variáveis na memória
- ❑ armazenam um endereço e não um valor
- ❑ em C, o ponteiro é mais um tipo permitido como qualquer outro

# Variáveis Ponteiros

---

Para declarar um ponteiro de um certo tipo, usa-se a seguinte sintaxe:

```
TipoBase *nome;
```

Onde:

**TipoBase** é o tipo da informação que será apontada pelo ponteiro (informa o total de bytes ocupados pela informação)

o símbolo **\*** serve para indicar que se trata da definição de um ponteiro

**nome** corresponde ao nome da variável ponteiro



# Definição de Variáveis do Tipo Ponteiro:

## 1. Ponteiros p/tipos pré-definidos (básicos):

```
int *ptInt;  
float *ptFloat;  
char *ptChar; // definição de string
```

## 2. Ponteiros p/tipos definidos pelo programador:

```
struct rgFunc {  
    char nome[40];  
    char sexo;  
    float salario;  
};  
  
struct rgFunc *ptFunc;
```



# Os Operadores de Ponteiros: \* e & (1/2)

O operador & devolve o endereço na memória do seu operando. Por exemplo:

```
int x = 10;  
int *p = &x;
```

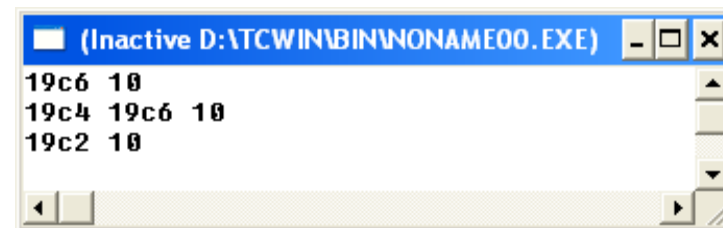
coloca em *p* o endereço da memória que contém a variável *x*. Esse endereço é a posição interna ao computador da variável. O endereço não tem relação alguma com o valor de *x*. O operador & pode ser imaginado como retornando “**o endereço de**”, ou seja, “*p* recebe o endereço de *x*”.

# Os Operadores de Ponteiros: \* e & (2/2)

O operador \* é o complemento de &, ele devolve o valor da variável localizada no endereço que o segue. Por exemplo, se *p* contém o endereço da variável *x*,

```
int x = 10;
int *p = &x;
int y = *p;
```

```
printf("%x %d\n", &x, x);
printf("%x %x %d\n", &p, p, *p);
printf("%x %d", &y, y);
```



coloca o valor de *x* em *y*. O operador \* pode ser imaginado como “no endereço”, ou seja, “*y* recebe o valor que está no endereço armazenado em *p*”.





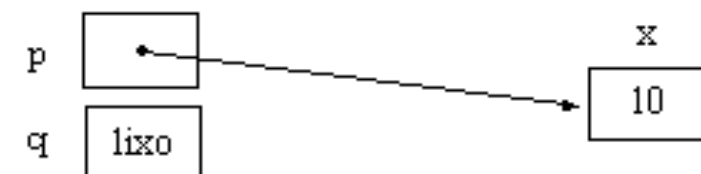
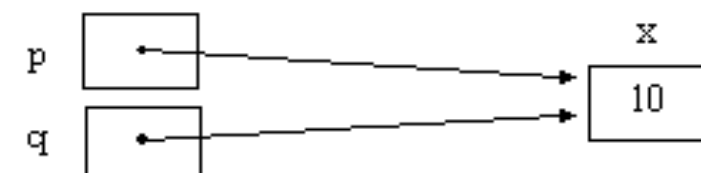
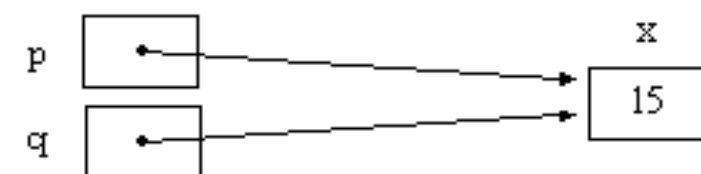
# Usando os operadores de ponteiros (\* e &)

```
void main() {  
    int x, *p;  
    x = 10;  
    p = &x;  
    printf("%d", *p);  
}
```

o operador & é usado para obter o endereço de uma variável estática

o operador \* é usado para obter o valor de um endereço de memória

**Obs.** quando um ponteiro contém o endereço de uma variável, diz-se que o ponteiro está "apontando" para essa variável. Se uma variável ponteiro *p* armazena um endereço de memória qualquer, então *\*p* apresenta o valor armazenado naquele endereço. Neste caso, *\*p* corresponde ao número inteiro 10.

código do programa	configuração da memória
<pre>#include "stdio.h";  void main() {     int x, *p, *q;</pre>	
<pre>x = 10;</pre>	
<pre>p = &amp;x;</pre>	
<pre>*q = 30;</pre>	<p><b>Aviso:</b> o endereço de memória armazenado em q não deve receber nenhum valor porque q ainda não aponta para ninguém (seu conteúdo é indefinido ou lixo).</p>
<pre>q = p;</pre>	
<pre>*p = *q + 5;</pre>	
<pre>printf("%d", *p); }</pre>	<p>Imprime o valor 15</p>



# Variáveis do Tipo Ponteiro

---

## Representação Interna:

As variáveis dinâmicas não são explicitamente declaradas como as estáticas, diretamente por apelidos ou identificadores.

São referenciadas por seus **endereços de memória**, armazenados em variáveis (estáticas) especiais chamadas de ponteiros ou apontadores.



# A Memória apontada pelo ponteiro:

---

Para trabalhar a memória que o ponteiro **p** está apontando, utiliza-se o operador **\***, com a sintaxe: **\*p**.

Naturalmente **p** deverá ter um endereço válido, isto é, **p** deve estar apontando para uma variável ou para uma memória de tipo compatível. Para tanto, dois outros tipos de atribuições podem ser feitos com variáveis do tipo ponteiro:

- a)** um ponteiro pode receber o conteúdo de outro ponteiro compatível (apontar para um endereço que contém um valor do mesmo tipo);
- b)** um ponteiro pode receber um endereço especial, chamado **NULL** (nulo), que serve para dizer que é um endereço nulo e que não haverá nenhuma variável neste endereço de memória.

# Alocação Dinâmica de Memória em C:

A área de memória que não é utilizada por um programa é organizada e gerenciada pelo sistema operacional, podendo ser alocada através de comandos padronizados oferecidos pela própria linguagem C.

Para **alocar** uma porção da área livre, basta utilizar o comando **malloc**, sendo **p** uma variável ponteiro: `p = (int *) malloc(sizeof(int));` aloca uma área de memória suficiente para armazenar um número inteiro e guarda o endereço ocupado em **p**, já o comando: `free(p);` serve para liberar a área de memória cujo endereço está em **p**. Uma área de memória, que foi alocada pelo comando **malloc**, somente volta a ficar disponível se for explicitamente liberada pelo comando **free**, caso contrário, a liberação da área só ocorrerá quando o programa que fez a alocação terminar de executar.

Para completar, `p = NULL;` marca **p** como um endereço nulo. Este endereço é útil para dizer que o ponteiro ainda não tem nenhum endereço válido (não aponta para ninguém).



# Representação Interna Variável Ponteiro:

```
void main() {
```

```
int *p;
```

I

aloca a variável estaticamente

aloca a variável dinamicamente

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

II

```
*p = 10;
```

III

```
...
```

```
free(p);
```

IV

```
}
```

memória usada pelo programa

p:

Lixo

memória livre no sistema



# Entendendo o código:

---

```
int *p;
```

a variável  $p$  é um ponteiro para um número inteiro, ou seja, “aponta” para um endereço de memória que será “alocado” para armazenar um número inteiro (4 bytes)

---

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

solicita ao sistema operacional 4 bytes da memória livre e o “endereço do espaço alocado” é colocado na variável ponteiro  $p$

---

```
*p = 10;
```

no endereço apontado por  $p$  armazena o número inteiro 10

---

```
free(p);
```

libera o espaço de memória ocupado cujo endereço está em  $p$  (devolve o recurso ao sistema operacional)

---



# Operadores ponto (.) e seta (->):

Os operadores ponto e seta são usados para especificar elementos individuais de estruturas (struct) e uniões. O operador ponto é usado para especificar diretamente o elemento e o operador seta para especificar o elemento através de um ponteiro.

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

```
struct rgPessoa {
    char nome[35];
    char apelido[20];
};
```

```
int main() {
    struct rgPessoa pessoa;
    struct rgPessoa *pontPessoa = &pessoa;

    strcpy(pessoa.nome, "Omero Francisco Bertol");
    strcpy(pessoa.apelido, "Chico");

    printf("Resultado:\n");
    printf("Nome...: %s\n", pontPessoa->nome);
    printf("Apelido: %s\n", pontPessoa->apelido);
}
```

```
D:\temp\ED325\Ponteiros\din3.exe
Resultado:
Nome...: Omero Francisco Bertol
Apelido: Chico

Process returned 15 (0xF)   execution time : 1.800 s
Press any key to continue.
```





# Listas Encadeadas com Vetor (1/2)

---

A forma mais simples de armazenar uma lista dentro do computador consiste em colocar os seus elementos em células de memória consecutivas (ou contíguas), um após o outro- utilizando o tipo estruturado homogêneo **vetor**.

A maior vantagem no uso de uma área sequencial de memória para armazenar uma lista linear é que, todos os elementos da estrutura são igualmente acessíveis, isto é, o tempo e o tipo de procedimentos para acessar qualquer um dos elementos do **vetor** são iguais.

O ponto fraco desta forma de armazenamento aparece quando é necessário inserir ou retirar elementos do meio da lista, quando então um certo esforço será necessário para movimentar os elementos, de modo a abrir espaço para inserção, ou de modo a ocupar o espaço liberado por um elemento que foi removido.

```
struct TipoItem {           // cada item da lista corresponde a um registro
    char nome[30];           // (TipoItem) composto apenas do campo nome
};

void estatica(void) {
    fflush(stdin);
#define maxTam 100           // maxTam = tamanho máximo da lista
    int n = 0, i;             // n = quantidade efetiva de itens na lista
    TipoItem lista[maxTam], x;
    while (1) {
        clrscr();
        printf("Lista de nomes (Alocação Estática)\n");
        for (i=0; i<n; i++)
            printf("%d- %s\n", i, lista[i].nome);
        printf("\nInforme um nome, (FIM) para encerrar:\n");
        gets(x.nome);
        if (strcmp(x.nome, "FIM") == 0)
            break;
        if (n == maxTam) {
            printf("\nErro: lista cheia !");
            printf("Pressione [algo] para prosseguir.\n"); getch();
        }
        else {
            lista[n] = x;      // coloca o item "x" na n-ésima posição da lista
            n = n + 1;         // o próximo item será colocado na posição n + 1
        }
    }
}
```

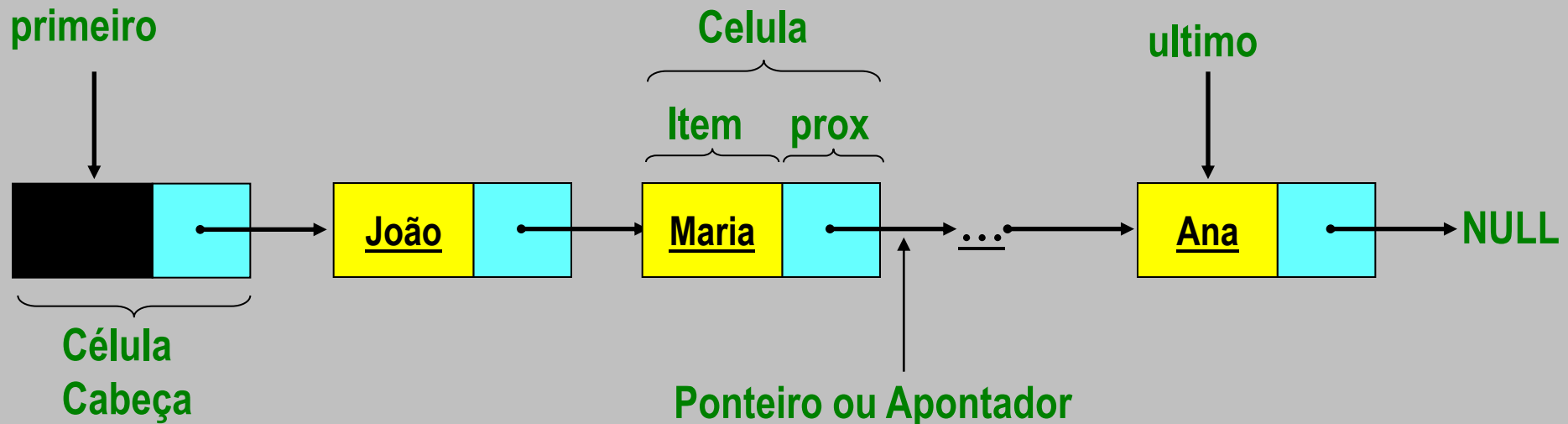
# Listas Encadeadas com Ponteiros (1/4)

Os tipos **ponteiros** ou **apontadores** são úteis para criar estruturas de dados **encadeadas**, do tipo listas (pilha e fila), árvores e grafos. Um apontador é uma variável que referencia uma outra variável **alocada dinamicamente**. Em geral a variável referenciada é definida como um registro que inclui também um apontador para outro elemento do mesmo tipo. Por Exemplo:

```
struct Celula {  
    Tipoltem Item;  
    Celula *prox;  
};  
Celula *primeiro, *ultimo, *p;
```

Sendo a variável *primeiro* um endereço para um registro *Celula* que contém o *Item* armazenado e o endereço (*prox*) da próxima célula da lista- é possível criar uma lista encadeada através de ponteiros.

# Listas Encadeadas com Ponteiros (2/4)



Na alocação encadeada, os elementos são armazenados em blocos de memória denominados **células**, ou **nodos**, sendo que cada célula é composta por dois campos: um para armazenar os dados (*Item*) e outro para armazenar o endereço do próximo elemento da lista (*prox*)- para manter a relação de ordem linear. São endereços especiais da lista encadeada com ponteiros: **a) primeiro**: endereço do primeiro elemento da lista (célula cabeça), e, **b) ultimo**: endereço do último elemento da lista (o endereço do elemento nulo (NULL) segue o último elemento da lista).

# Listas Encadeadas (3/4)

---

**Vantagem** de Listas Encadeadas com ponteiros ou apontadores:

- **Facilidade de inserir ou remover elementos do meio da lista.**

Como os elementos não precisam estar armazenados em posições consecutivas de memória, nenhum dado precisa ser movimentado, bastando atualizar o campo de ligação (**prox**) do elemento que precede aquele inserido ou removido.

**Desvantagem** de Listas Encadeadas com ponteiros ou apontadores:

- **Acessar uma posição específica dentro da lista.**

Como apenas o primeiro elemento é acessível diretamente através do endereço **primeiro**, deve-se partir do primeiro e ir seguindo os campos de ligação (**prox**), um a um, até atingir a posição desejada. Obviamente, para listas extensas, esta operação pode ter um alto custo em relação a tempo.



```
void dinamica(void) {
    fflush(stdin);
    struct Celula {
        TipoItem Item;
        Celula *prox;
    };
    TipoItem x;
    Celula *primeiro, *ultimo, *p;

    // cria a célula cabeça
    primeiro = (Celula *) malloc(sizeof(Celula));
    primeiro->prox = NULL;
    ultimo = primeiro;

    while (1) {
        clrscr(); printf("Lista de nomes (Alocação Dinâmica)\n");
        p = primeiro->prox;
        while (p != NULL) {
            printf("%s\n", p->Item.nome);
            p = p->prox;
        }
        printf("\nInforme um nome, (FIM) para encerrar:\n"); gets(x.nome);
        if (strcmp(x.nome, "FIM") == 0)
            break;

        // coloca o novo item no final da lista
        ultimo->prox = (Celula *) malloc(sizeof(Celula));
        ultimo = ultimo->prox;
        ultimo->Item = x;
        ultimo->prox = NULL;
    }

    p = primeiro; // elimina (libera) todos os espaços de memória alocados
    while (p != NULL) {
        primeiro = primeiro->prox;
        free(p);
        p = primeiro;
    }
}
```

# Criando a Célula Cabeça da Lista

Célula especial, que “não” armazena itens, mas que é usada para indicar o primeiro elemento da lista. **Obs.** quando a lista está vazia a única célula existente é a célula cabeça, portanto, antes de qualquer operação com a lista deve-se criar a célula cabeça.

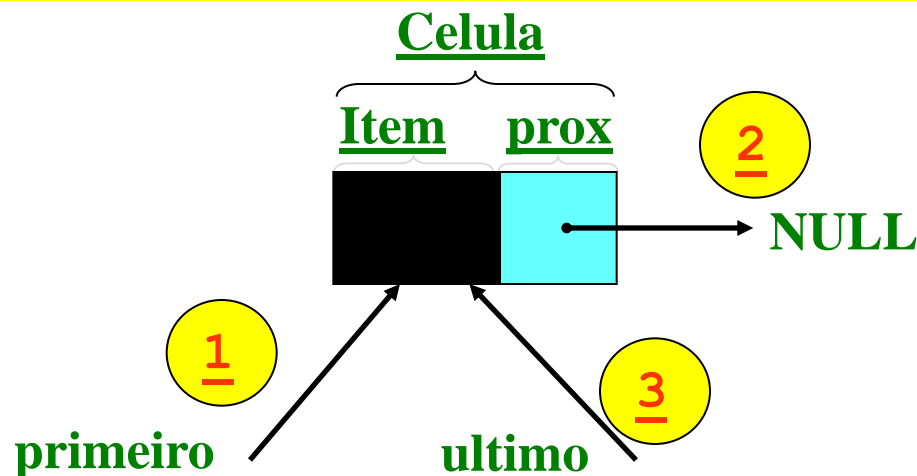
```
Celula *primeiro, *ultimo;
```

```
...
```

```
1. primeiro = (Celula *) malloc(sizeof(Celula));
```

```
2. primeiro->prox = NULL;
```

```
3. ultimo = primeiro;
```



# Entendendo o Código (1/2):

primeiro = (Celula \*)

malloc (sizeof(Celula));

chama a função **malloc** para alocar a memória dinamicamente, serão alocados sizeof(Celula) bytes

indica o valor retornado pela função **malloc**, ou seja, o endereço do espaço alocado pelo sistema operacional para armazenar uma célula da lista

primeiro->prox = NULL;

o operador -> (seta) é usado para referenciar elementos individuais (ou campos) de estruturas (ou registros) apontadas por variáveis ponteiros





# Entendendo o Código (2/2):

---

**Celula** \*primeiro, \*ultimo;

as variáveis *primeiro* e *ultimo* são ponteiros para um registro “Celula”, ou seja, “apontam” para endereços de memória que serão usados para armazenar os itens (ou células) da lista

---

primeiro = (**Celula** \*) malloc(sizeof(**Celula**));

solicita ao sistema operacional sizeof(Celula) bytes da memória livre e o “endereço do espaço alocado” é colocado na variável ponteiro *primeiro*

---

primeiro->prox = NULL;

no campo “prox” da célula apontada pelo ponteiro *primeiro* armazena o endereço NULL, ou seja, o endereço de ninguém

---

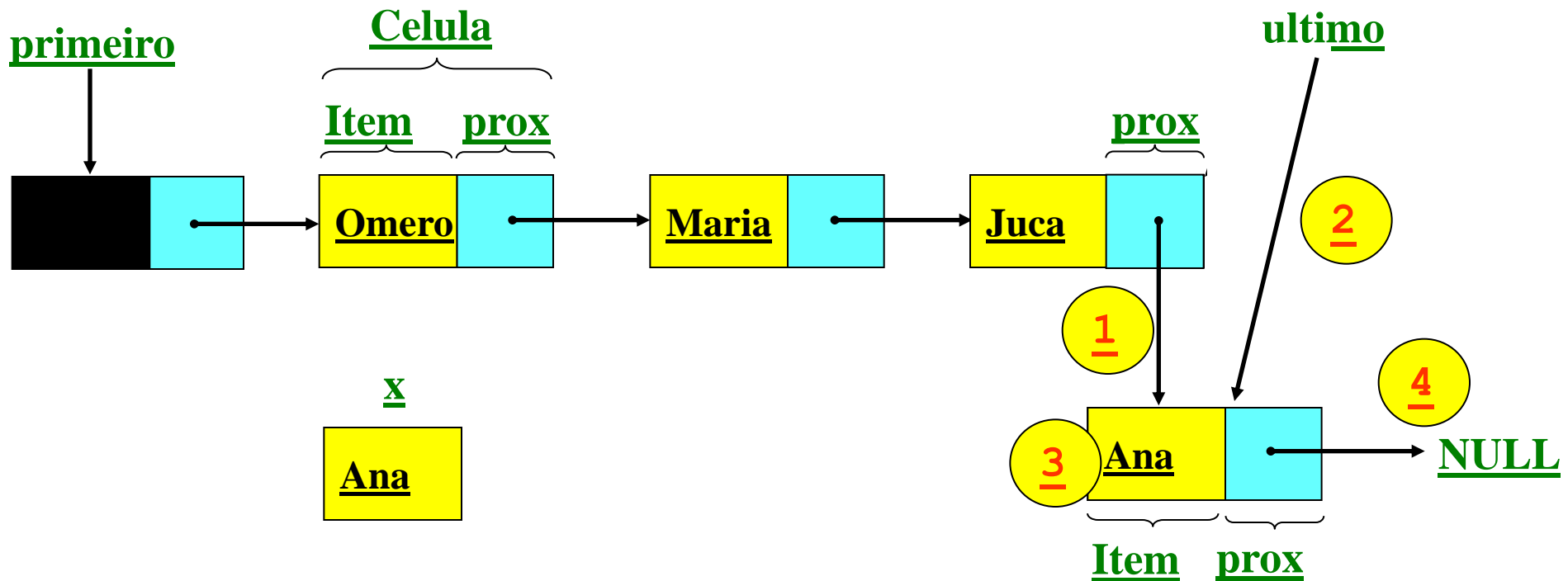
ultimo = primeiro;

o ponteiro *ultimo* aponta para o mesmo endereço armazenado no ponteiro *primeiro*

---

# Colocando o Novo Item no Final da Lista

1. `ultimo->prox = (Celula *) malloc(sizeof(Celula));`
2. `ultimo = ultimo->prox;`
3. `ultimo->Item = x;`
4. `ultimo->prox = NULL;`



# Percorrendo uma Lista Encadeada

O algoritmo para percorrer uma lista do primeiro elemento até o último, se resume, na utilização de um apontador auxiliar **p** que deve inicialmente receber o endereço da primeira célula da Lista (**p = primeiro->prox;**). A seguir, um processo repetitivo que faz **p** pular, através do campo de ligação (**prox**), do nodo atual para o nodo sucessor (**p = p->prox;**) até que **p** passe pelo último nodo da lista (**NULL**, o último nodo da lista aponta para o vazio).

Resumindo em termos de programa, o processo para percorrer uma lista encadeada pode ser descrito como segue:

```
Celula *p;  
...  
// endereço da primeira célula da lista  
p = primeiro->prox;  
while (p != NULL) {  
    printf("%s\n", p->Item.nome);  
// endereço do nodo sucessor (próximo)  
    p = p->prox;  
}
```



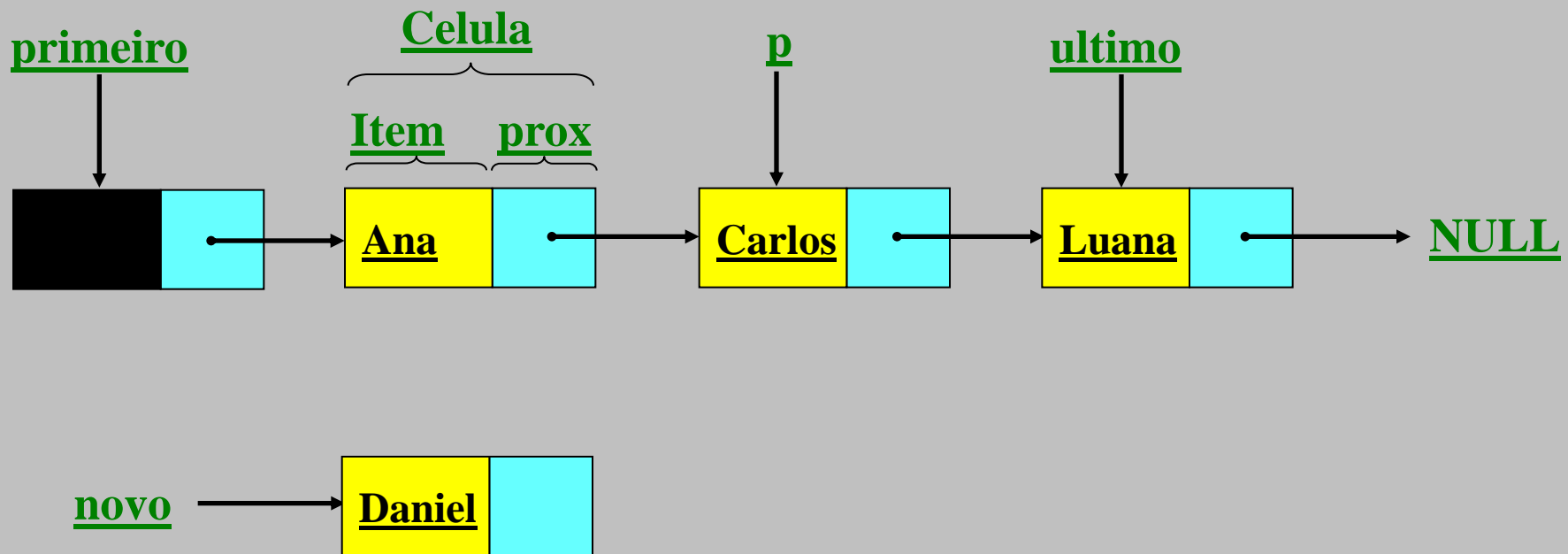
# Liberando os Espaços Alocados na Criação da Lista

O algoritmo para liberar os endereços de memória alocados na criação da lista deve percorrer, a partir da célula cabeça, todas as células da lista liberando o espaço de memória ocupado pela célula através da função **free**. Por exemplo:

```
Celula *primeiro, *p;  
...  
// endereço da célula cabeça  
p = primeiro;  
  
while (p != NULL) {  
// endereço do nodo sucessor (próximo)  
    primeiro = primeiro->prox;  
// libera a área de memória cujo endereço está em p  
    free(p);  
// próxima célula a ser liberada  
    p = primeiro;  
}
```

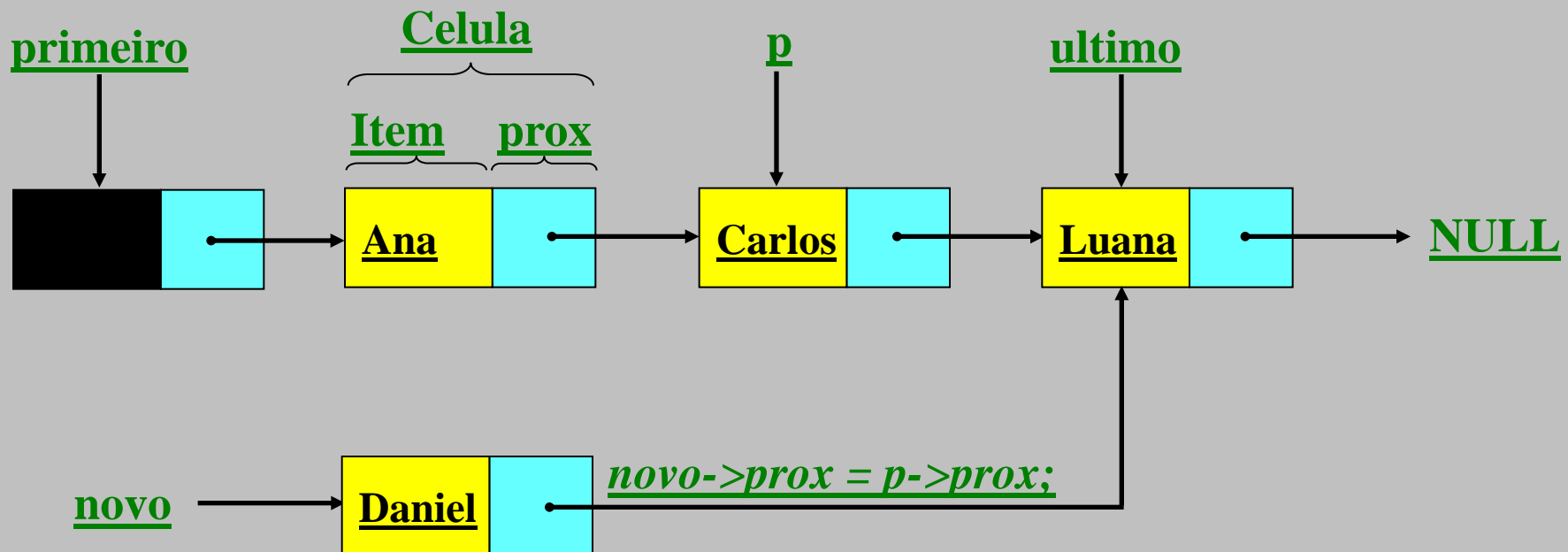
# Inserindo uma Nova Célula na Lista (1/4)

Dada uma cadeia de células (ou nodos), é necessário expandir esta cadeia com novos blocos. Para entender como isto pode ser feito, basta considerar a situação esquematizada, na Figura abaixo, onde um nodo de endereço **novo** deve ser inserido em uma Lista, após a posição indicada pelo ponteiro **p**:



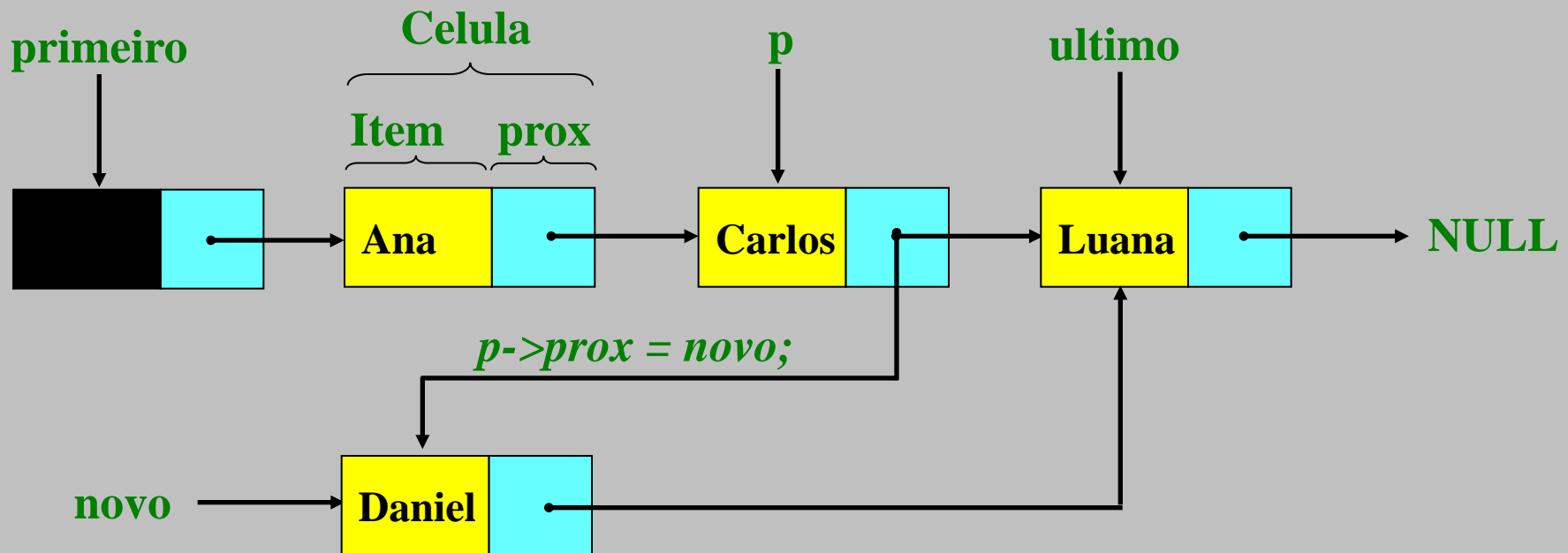
# Inserindo uma Nova Célula na Lista (2/4)

Como o novo nodo será inserido logo após aquele que armazena o elemento "*Carlos*", seu sucessor será aquele nodo que armazena o elemento "*Luana*". Para isto, basta atualizar o campo de ligação (***prox***) do novo nodo com o endereço daquele que guarda o elemento "*Luana*".



# Inserindo uma Nova Célula na Lista (3/4)

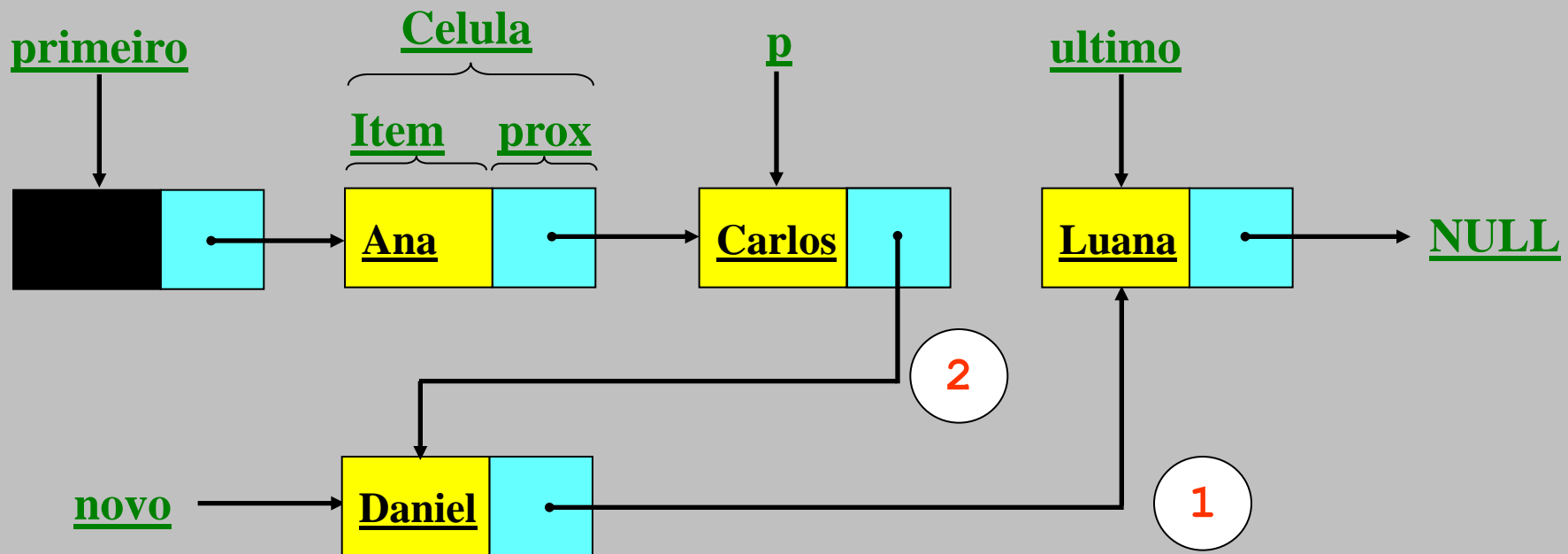
Na próxima etapa, faz-se o sucessor do nodo apontado por **p** ("Carlos") ser aquele apontado por **novo** ("Daniel").



# Inserindo uma Nova Célula na Lista (4/4)

Resumindo em termos de programa, o processo de inserção pode ser descrito como segue:

1. `novo->prox = p->prox;`
2. `p->prox = novo;`

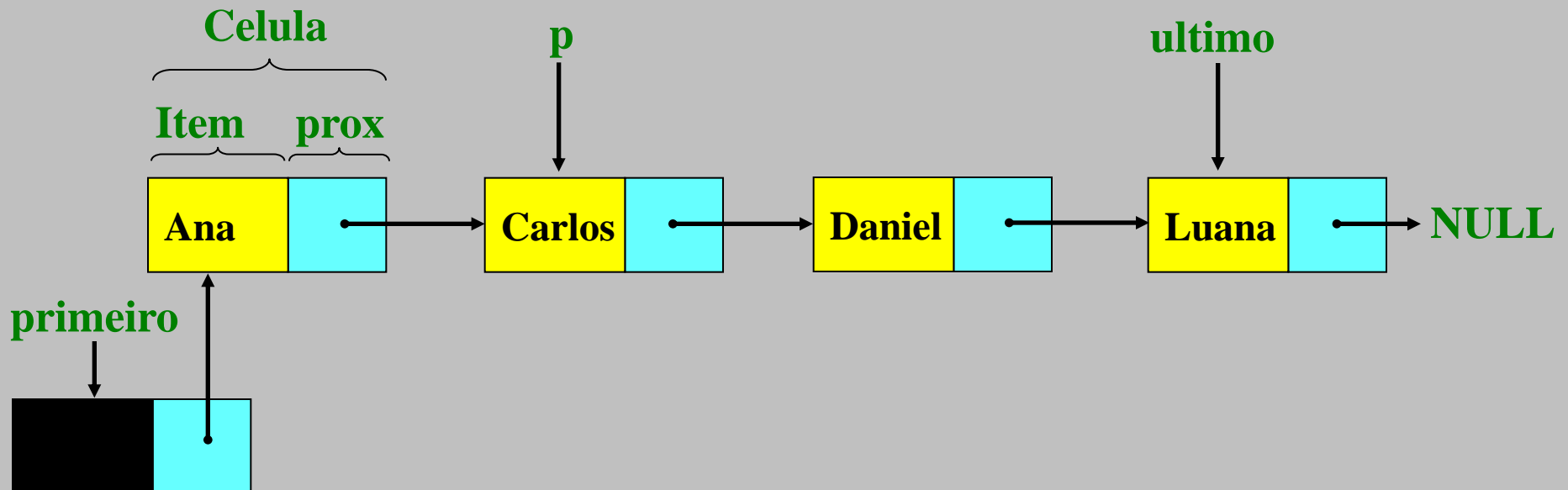




# Removendo uma Célula da Lista (1/5)

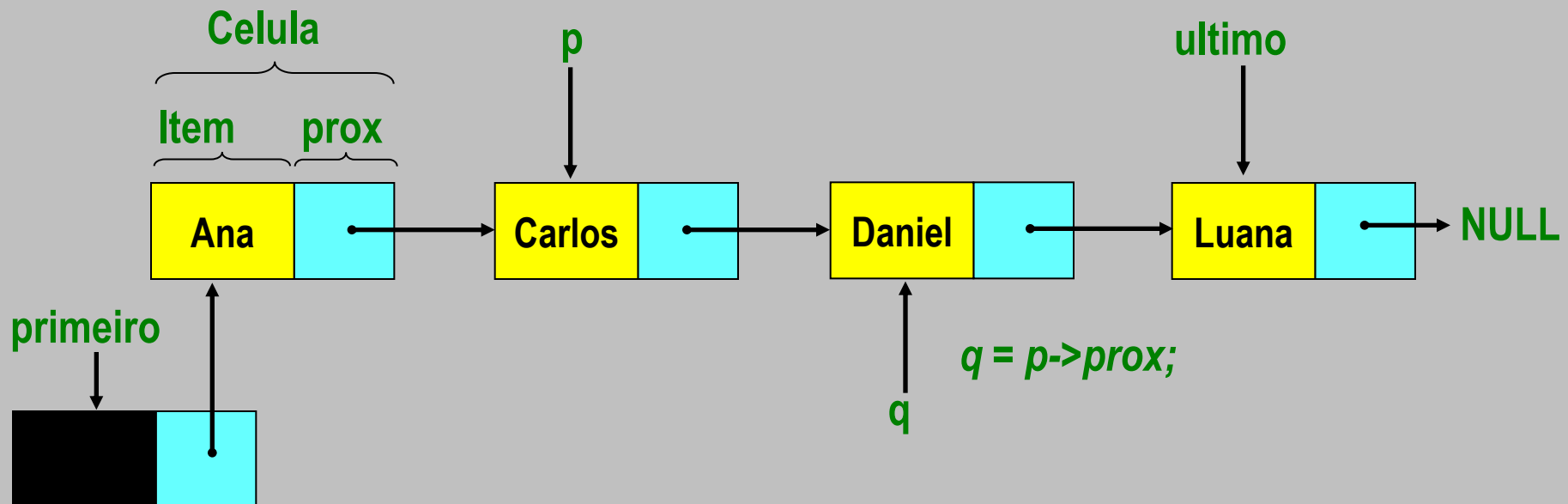
Considerando agora a situação esquematizada, na Figura abaixo, onde o nodo ("*Daniel*") sucessor do nodo indicado pelo ponteiro **p** ("*Carlos*") deve ser removido, ou excluído.

Situação antes da remoção.



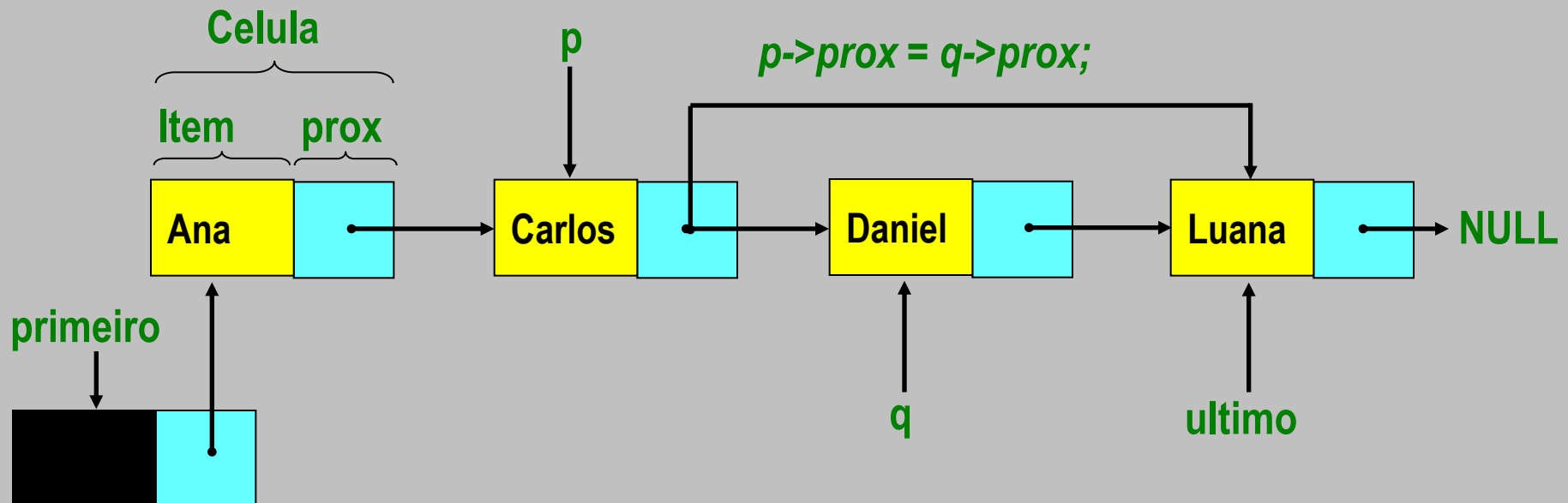
# Removendo uma Célula da Lista (2/5)

Antes de isolar o nodo "*Daniel*" a ser removido, o que tornaria o nodo inacessível, deve-se copiar o seu endereço para um ponteiro auxiliar **q**. Deste modo será possível acessar o nodo **q** para posteriormente devolvê-lo ao sistema gerenciador de memória, através da função **free.5**



# Removendo uma Célula da Lista (3/5)

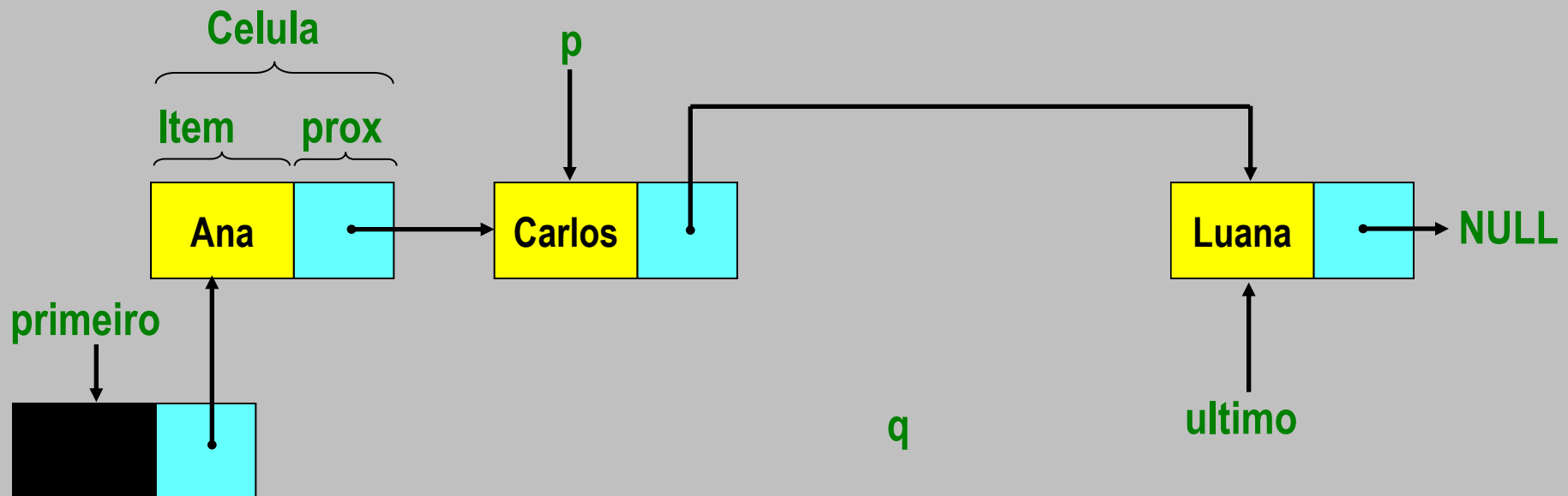
Como o nodo que armazena o elemento "*Daniel*" deve ser removido, o seu sucessor ("*Luana*") passa a ser o sucessor do nodo que o precede ("*Carlos*"); em outras palavras, o nodo que armazena o elemento "*Luana*" passa a ser o sucessor do nodo apontado por **p**.



# Removendo uma Célula da Lista (4/5)

Finalmente, o nodo apontado por **q** pode ser liberado e a remoção é concluída.

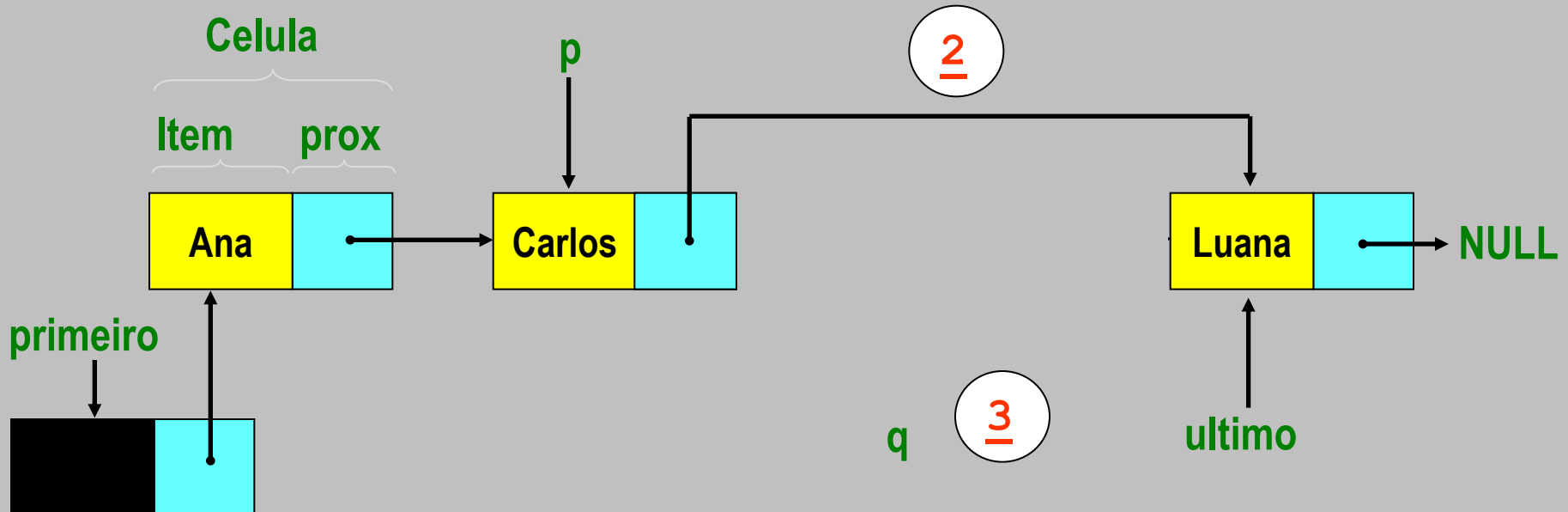
*free(q);*



# Removendo uma Célula da Lista (5/5)

Resumindo em termos de programa, o processo de remoção pode ser descrito como segue:

1.  $q = p \rightarrow \text{prox};$
2.  $p \rightarrow \text{prox} = q \rightarrow \text{prox};$
3. **free(q);**



# Referências

---

- ❑ Estrutura de Dados Fundamentais: conceitos e aplicações.
  - ❑ Silvio do Lago Pereira.
  - ❑ 2ª ed. - São Paulo: Érica, 1996.
- ❑ Instituto de Computação da UNICAMP
  - ❑ Flávio K. Miyazawa & Tomasz Kowaltowski
- ❑ Material adaptado do Prof.: Omero Francisco Bertol - UTFPR.