UEM/CTC – Departamento de Informática
Curso: Ciência da Computação
Professor: Flávio Rogério Uber

Estrutura de Dados
(6884/3 e 4)

Ponteiros e Listas

Material Original: Prof. Dr. Yandre Maldonado e Gomes da Costa

Ponteiros

- Ponteiro é uma variável que possui o endereço de outra variável;
- É um poderoso recurso na programação;
- Existem computações que só podem ser expressas através do uso de ponteiros;
- Deve ser utilizado com critério e disciplina. Seu uso indevido pode levar a erros de execução de programas (acesso de endereço inválido);

Ponteiros

- Para definir que uma variável guardará o endereço de outra (ponteiro) utiliza-se na sua declaração o operador unário “*”;

Exemplo:

```
int *pont;      // define um ponteiro para um inteiro
int x, y;   // variáveis do tipo inteiro
```

Neste exemplo *pont* é uma variável do tipo ponteiro para inteiro, ou seja, ela irá receber um endereço de uma variável inteira.

Ponteiros

- Para se obter o endereço de uma variável, o operador unário “&” pode ser utilizado.
  - Considerando o exemplo anterior:

    pont = &x // *pont* recebe o endereço de *x*
  - Após esta instrução, diz-se que *pont* aponta para *x*;
  - É possível acessar o conteúdo da variável *x* através do ponteiro *pont* utilizando o operador unário “*”:

    y = *pont // *y* recebe o conteúdo da var. apontada por *pont*

Ponteiros

- Também seria possível definir o conteúdo de *x* através do ponteiro, como em:

  ***pont = y;**

  → *Neste caso o conteúdo da variável apontada por pont recebe o conteúdo de y;*

Exercícios

1) Encontre o erro:

a)

```
int main() {
        int x, *p;
        x = 100;
        p = x;
        printf(“Valor de p: %d.\n”, *p);
}
```

b)

```
void troca (int *i, int *j) {
        int *temp;
        *temp = *i;
        *i = *j;
        *j = *temp;
}
```

Listas Lineares

Listas Lineares

- Forma de interligar um conjunto de elementos;
- Nesta estrutura os elementos são distribuídos em sequência;
- Nesta estrutura, as operações inserir, retirar e localizar são definidas;
  - Itens podem ser acessados, inseridos ou retirados;
- Podem crescer ou diminuir ao longo da sua vida útil (de acordo com a demanda);
- Listas Encadeadas Dinamicamente são adequadas para aplicações onde não é possível prever a demanda por memória;
  - Permite a manipulação de quantidades imprevisíveis de dados;

Listas Lineares

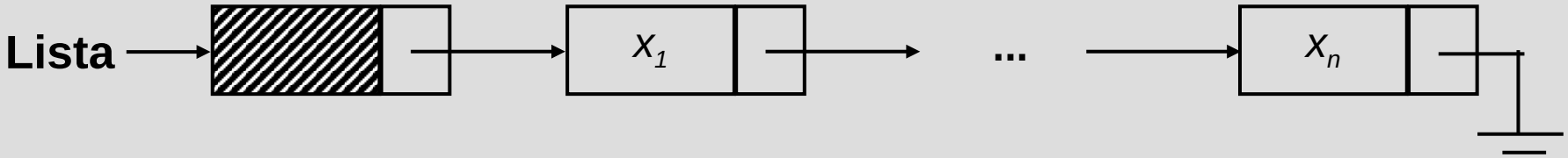
- Uma **Lista Linear** é uma seqüência de zero ou mais itens $x_1, x_2, \ldots, x_n$;
  - Onde x_i é de um determinado tipo;
  - n representa o tamanho da lista linear;
  - Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão;
  - Assumindo $n \geq 1$, x_1 é o primeiro item da lista e x_n é o último item da lista;
    - Em geral, x_i precede x_{i+1} para $i=1, 2,\ldots, n-1$;
    - E x_i sucede x_{i-1} para $i=2, 3,\ldots, n$;
    - O elemento x_i é dito estar na i-ésima posição da lista;

Listas Lineares

- Sobre uma lista, é definido um conjunto de operações. Algumas das mais comuns, necessárias à maioria das aplicações são:
  - Criar uma lista vazia (inicializar a lista);
  - Inserir um novo item imediatamente após o i-ésimo item;
  - Retirar o i-ésimo item;
  - Localizar um item;
  - Exibir os dados referentes a todos os itens armazenados na lista;

Listas Lineares

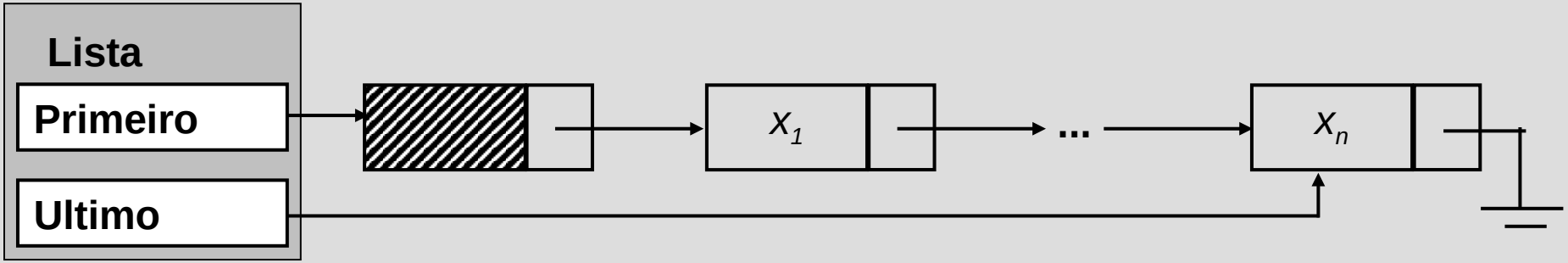
- Implementação de Lista através de apontadores:
  - Cada item da lista é encadeado com o seguinte através de uma variável do tipo apontador.
  - Isto permite o uso de posições não contíguas de memória;
  - É possível inserir e retirar elementos sem haver a necessidade de deslocar os itens seguintes da lista;



  - Observe que existe uma célula cabeça que aponta para x1, ela não contém informação mas é interessante que tenha estrutura igual as demais para simplificar a implementação;

Listas Lineares

- A lista é constituída por células, onde cada célula contém um item da lista e um apontador para a célula seguinte;
- O registro TipoLista, que será utilizado no código fonte a ser mostrado, contém um apontador para a célula cabeça e um apontador para a última célula da lista;



- A partir do ponteiro que aponta para a cabeça pode-se acessar todo e qualquer elemento da lista;

Listas Lineares

- A lista encadeada dinâmica permite que se insira ou remova elementos da lista a um custo constante*;
- Entretanto este tipo de implementação consome memória extra para o armazenamento dos ponteiros;
- Esta implementação é uma boa opção para os casos em que não há previsão sobre o crescimento da lista;

* Se desprezarmos o custo de localização da posição onde a operação será efetuada.

Listas Lineares

- Elementos da linguagem C a serem utilizados:
  - typedef;
  - -> (“seta”)
  - malloc();
  - free();

Listas Lineares

- **typedef**
  - Permite dar novos nomes aos tipos de dados;
  - Exemplos:

```
typedef float media;
...
media turma1;
//cria uma variável turma1 do tipo float
```

```
typedef struct {
  //membros
 } tipo_struct;
...
tipo_struct teste;
//cria uma variável teste do tipo tipo_struct
```

Listas Lineares

- **-> (“seta”)**
  - Permite referenciar um elemento individual de uma estrutura quando esta é acessada através de um ponteiro;
  - Exemplo:

```
struct empregado
 {
  char nome[80];
  int idade;
  float salario;
 };
...
struct empregado *p;
...
p->salario = 200;
```

Listas Lineares

- **malloc()**
  - Função que permite alocar memória livre;
  - Se a memória estiver cheia a função retorna um ponteiro NULL (nulo);
  - Exemplo:

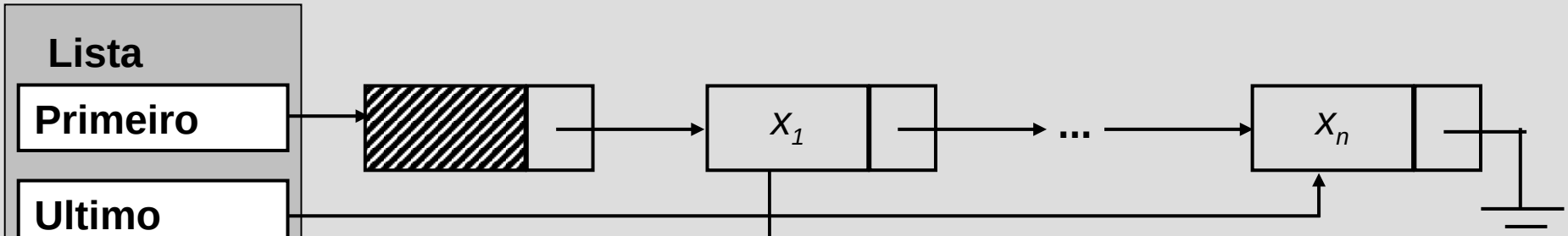
```
int *p;
p=(int *)malloc(sizeof(int));
```

Listas Lineares

- **free()**
  - Aceita como argumento um ponteiro e libera a área de memória apontada pelo mesmo;

```
free (p);
//libera o espaço de memória apontado por p;
```

Listas Lineares




```
typedef struct
 {
  int chave;
  char nome [30];
  int idade;
  //outros campos
 }TipoItem;
```

```
typedef struct
 {
  Celula* Primeiro;
  Celula* Ultimo;
 }TipoLista;
```

```
struct Celula
 {
  TipoItem Item;
  struct Celula* Prox;
 };
```

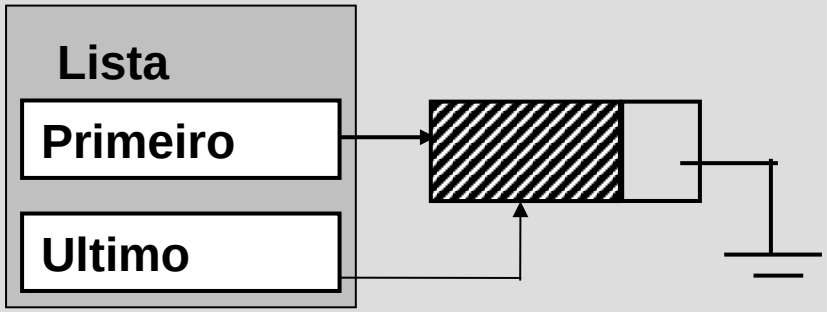
Listas Lineares

Operações:

```
void FLVazia (TipoLista * Lista);

int Vazia (TipoLista Lista);

void Insere (TipoItem x, TipoLista *Lista);

struct Celula* Localiza (int cod, TipoLista *Lista);

void Retira (struct Celula* p, TipoLista *Lista);

void Imprime (TipoLista Lista);
```

Listas Lineares

void FLVazia (TipoLista * Lista);




```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

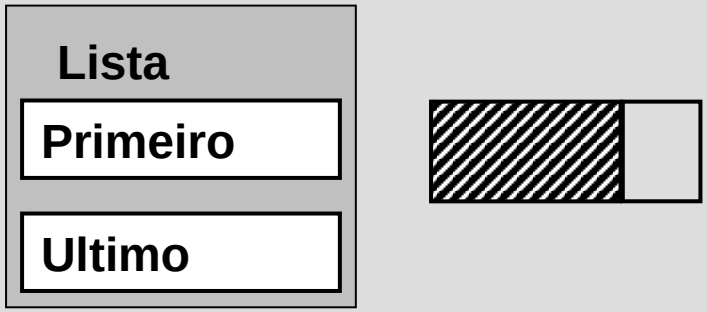
Listas Lineares

void FLVazia (TipoLista * Lista);


```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

Listas Lineares

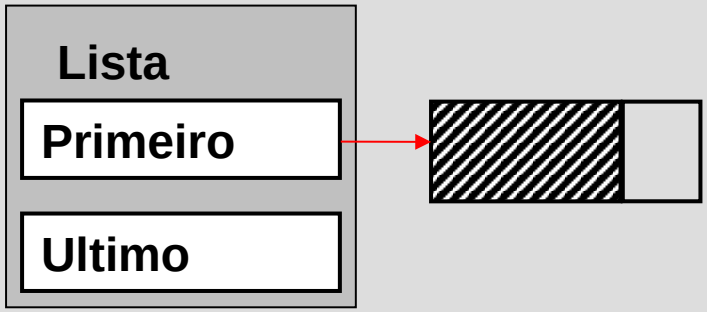
void FLVazia (TipoLista * Lista);




```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

Listas Lineares

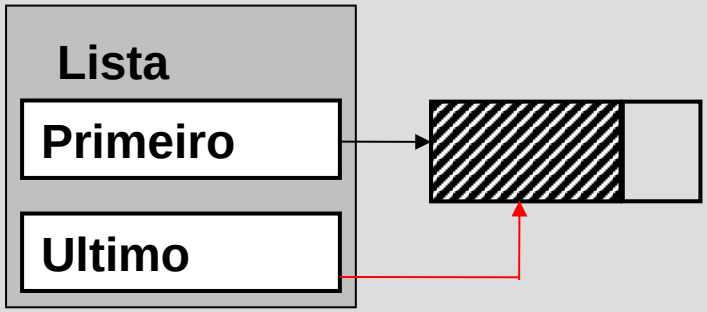
void FLVazia (TipoLista * Lista);




```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

Listas Lineares

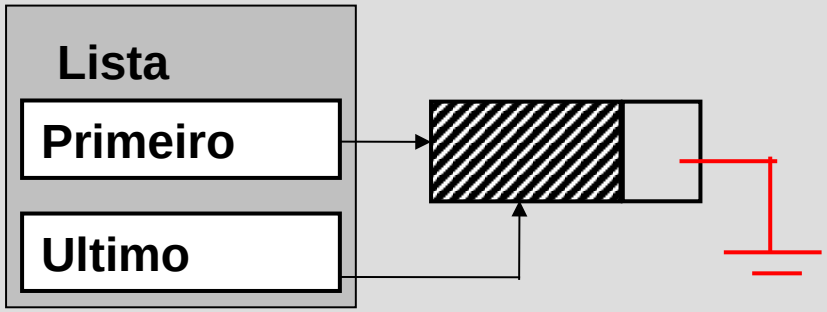
void FLVazia (TipoLista * Lista);




```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

Listas Lineares

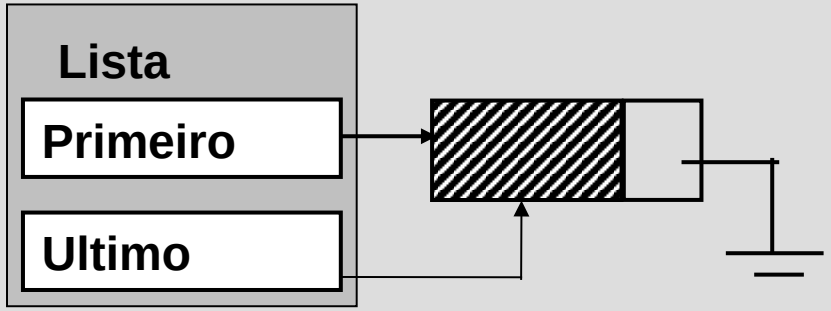
void FLVazia (TipoLista * Lista);




```
void FLVazia (TipoLista *Lista)
 {
  Lista -> Primeiro = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Primeiro;
  Lista -> Primeiro -> Prox = NULL;
 }
```

Listas Lineares

int Vazia (TipoLista Lista);

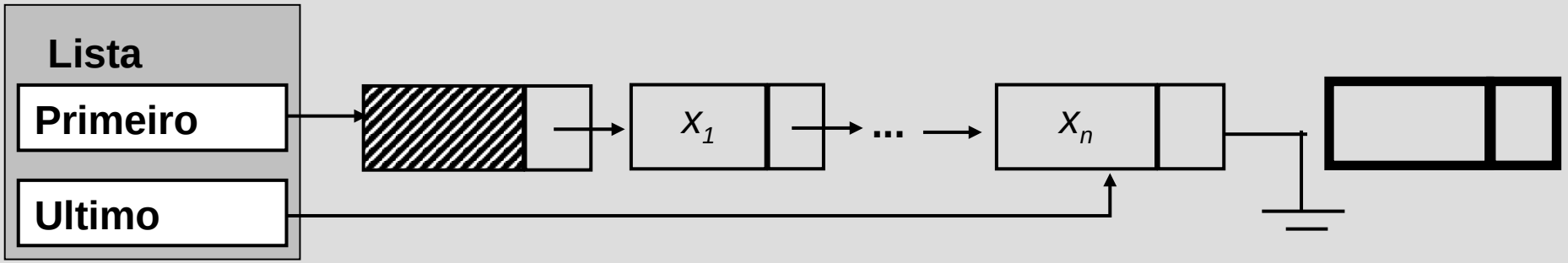


?

```
int Vazia (TipoLista Lista)
 {
  return (Lista.Primeiro == Lista.Ultimo);
 }
```

Listas Lineares

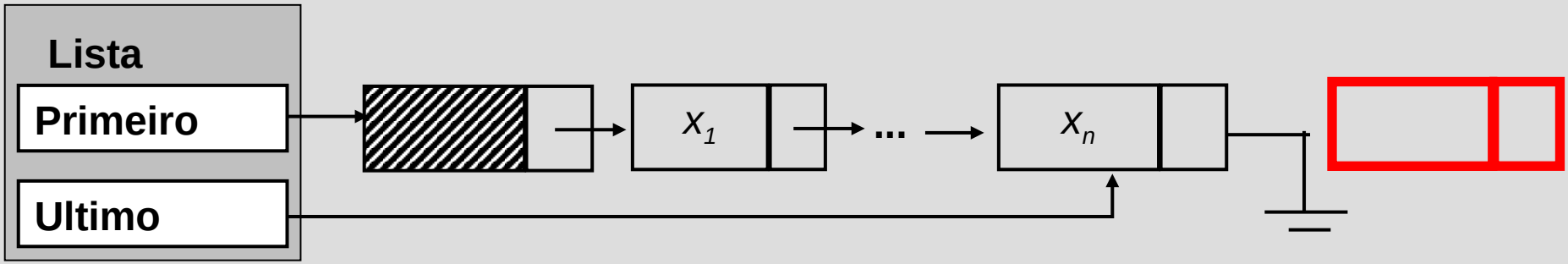
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

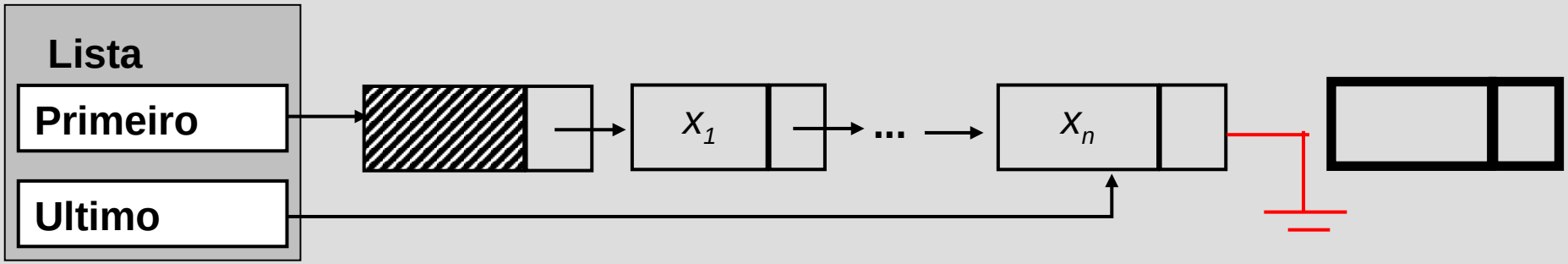
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

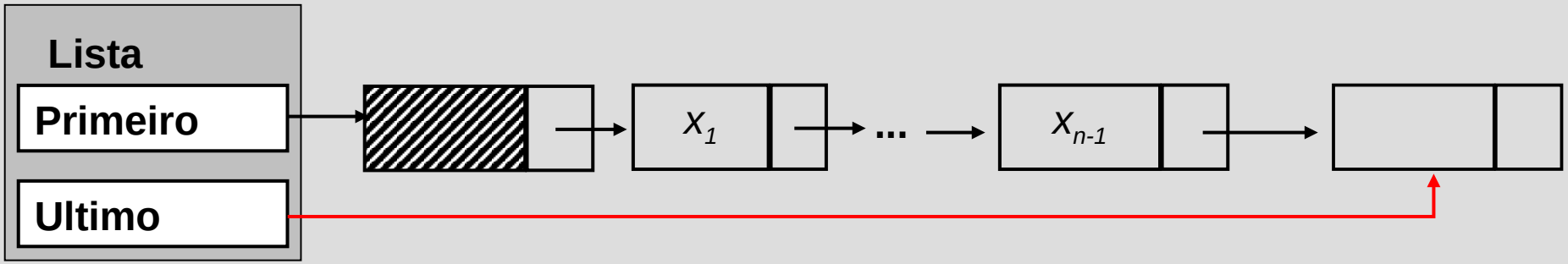
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

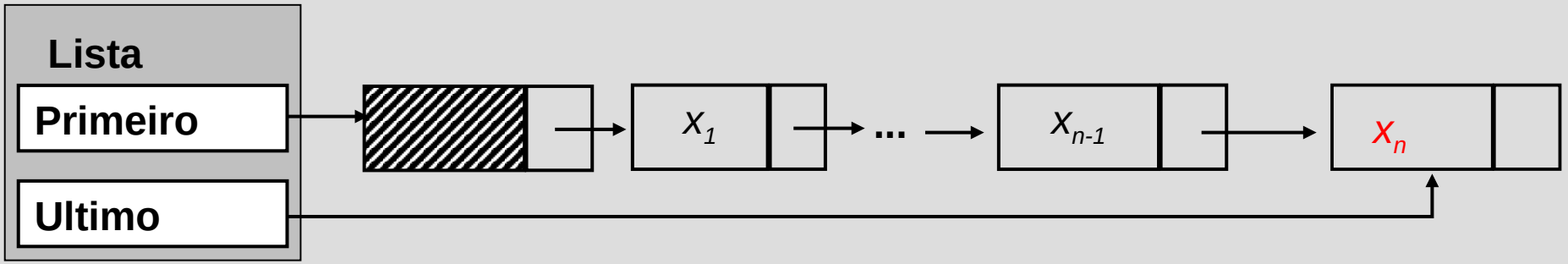
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

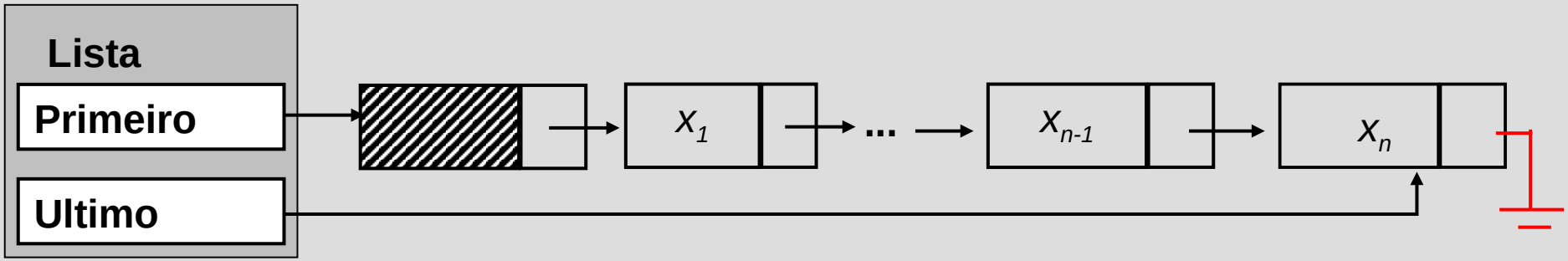
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

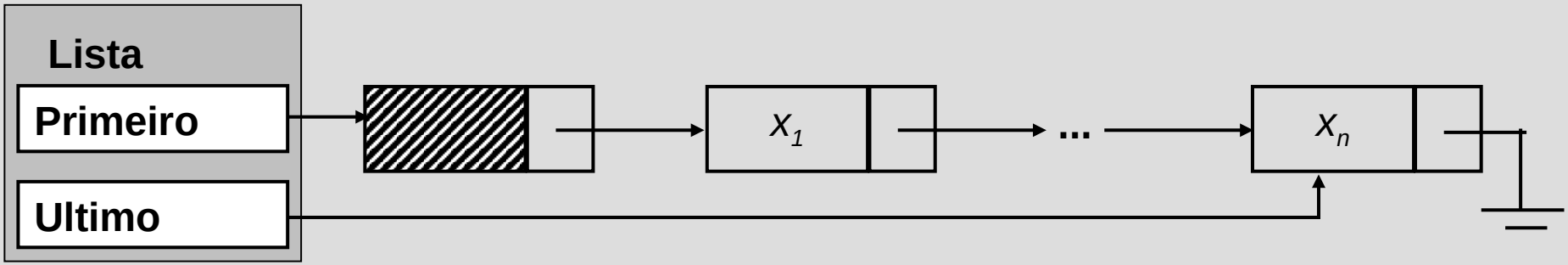
void Insere (TipoItem x, TipoLista *Lista);




```
void Insere (TipoItem x, TipoLista *Lista)
 {
  Lista -> Ultimo -> Prox = (struct Celula*) malloc(sizeof(struct Celula));
  Lista -> Ultimo = Lista -> Ultimo -> Prox;
  Lista -> Ultimo -> Item = x;
  Lista -> Ultimo -> Prox = NULL;
 }
```

Listas Lineares

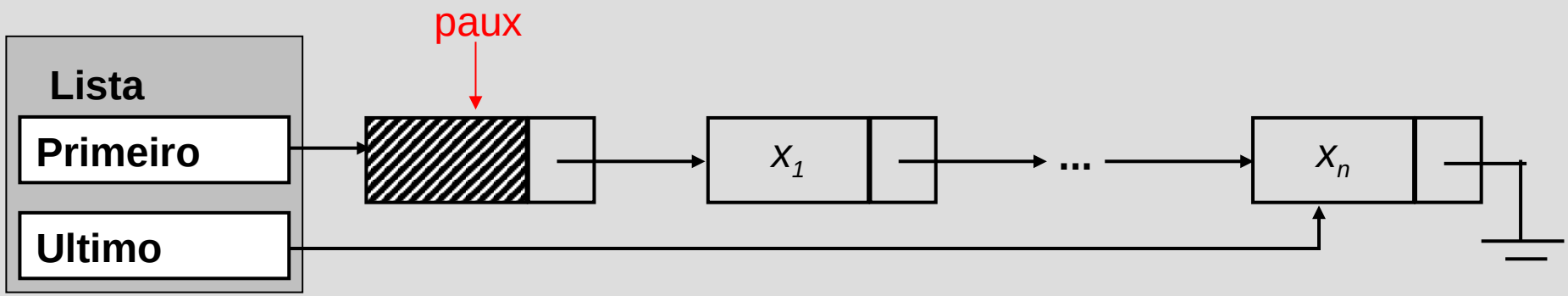
struct Celula* Localiza (int cod, TipoLista *Lista);




```
Celula* Localiza (int cod, TipoLista *Lista)
 {
  struct Celula* paux;
  paux = Lista->Primeiro;
  while ((paux->Prox != NULL) && (paux->Prox->Item.chave != cod))
   {
    paux = paux->Prox;
   }
  return (paux);
 }
```

Listas Lineares

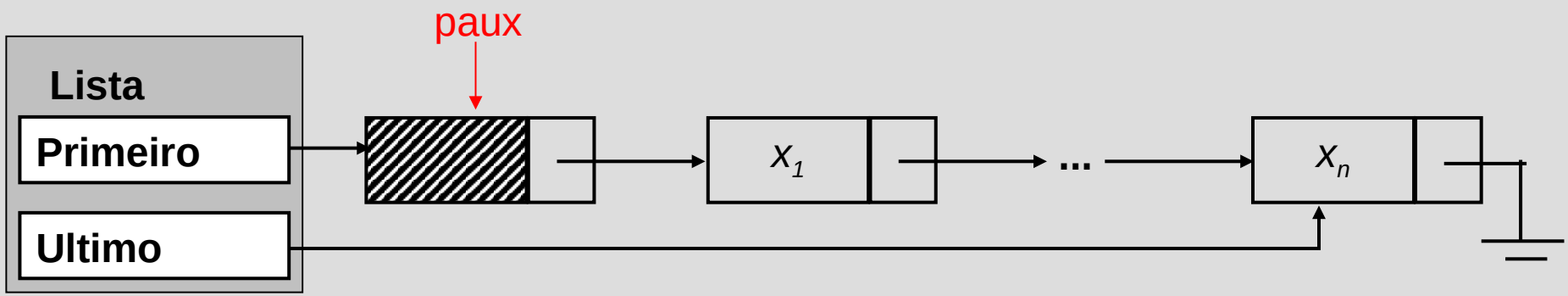
struct Celula* Localiza (int cod, TipoLista *Lista);




```
Celula* Localiza (int cod, TipoLista *Lista)
 {
  struct Celula* paux;
  paux = Lista->Primeiro;
  while ((paux->Prox != NULL) && (paux->Prox->Item.chave != cod))
   {
    paux = paux->Prox;
   }
  return (paux); //paux estará na posição anterior ao item desejado
 }
```

Listas Lineares

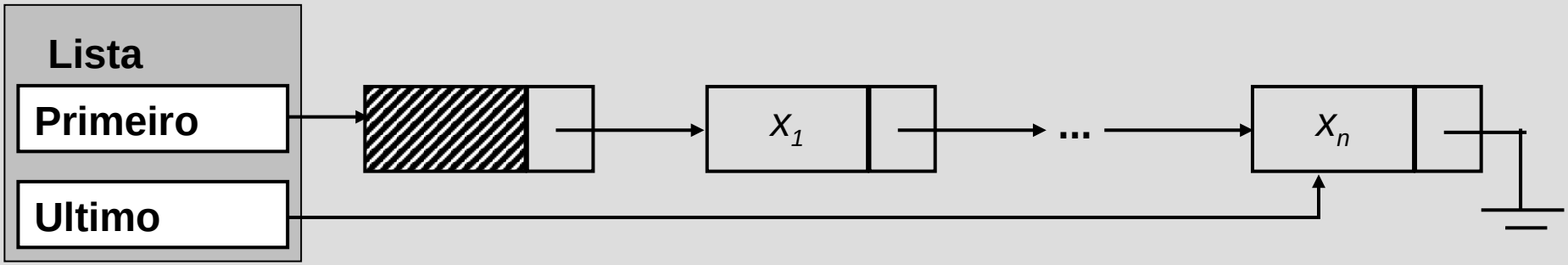
struct Celula* Localiza (int cod, TipoLista *Lista);




```
Celula* Localiza (int cod, TipoLista *Lista)
 {
  struct Celula* paux;
  paux = Lista->Primeiro;
  while ((paux->Prox != NULL) && (paux->Prox->Item.chave != cod))
   {
    paux = paux->Prox;
   }
  return (paux); //paux estará na posição anterior ao item desejado
 }
```

Listas Lineares

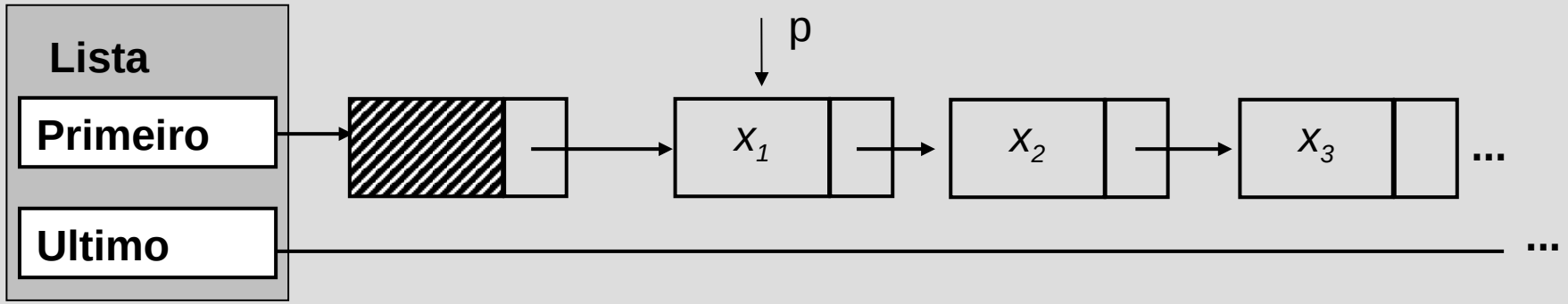
void Retira (struct Celula* p, TipoLista *Lista);



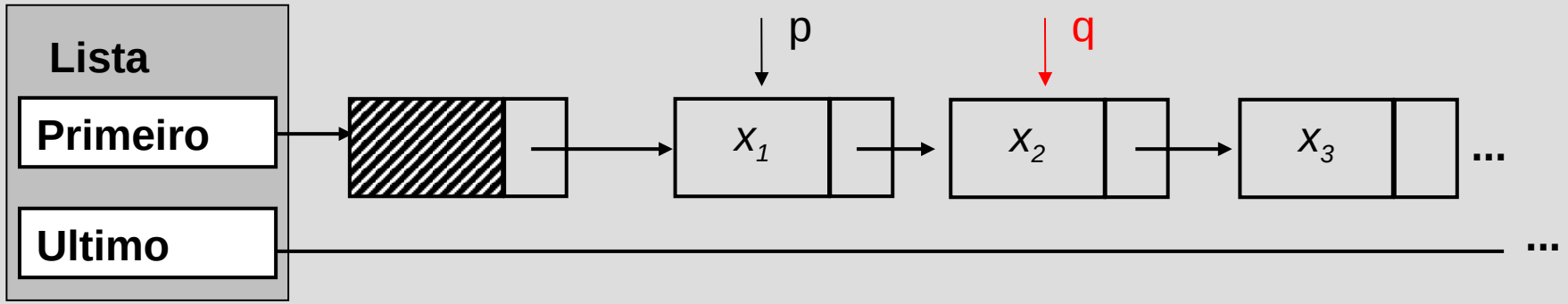
Vamos supor que foi feita uma chamada anterior para a função “Localiza” e portanto existe um ponteiro posicionado na célula anterior àquela que será removida.

Este ponteiro será passado como parâmetro para a função “Retira”

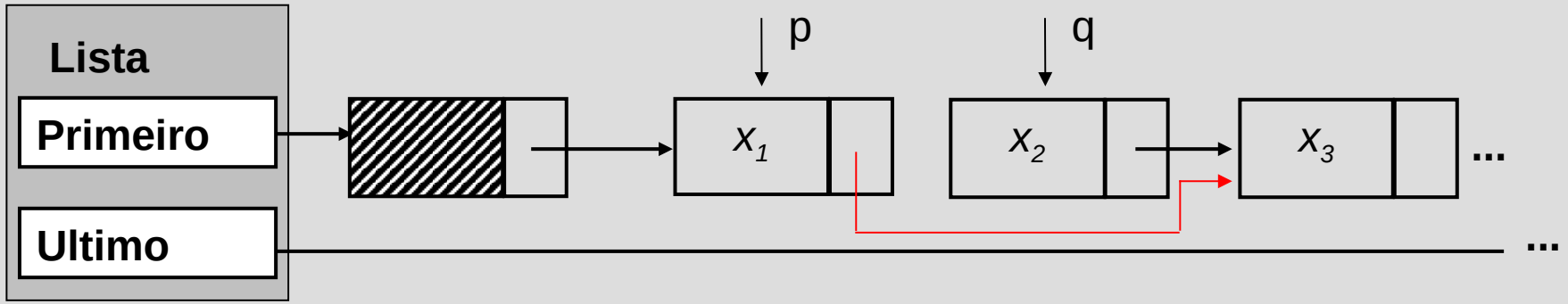
```
void Retira (Celula* p, TipoLista *Lista)
 {
  struct Celula* q;

  if (Vazia(*Lista)||p->Prox==NULL)
   {
    printf ("Erro: lista vazia ou posicao inexistente.\n");
    getch();
   }
  else
   {
    q = p->Prox;
    p->Prox = q->Prox;
    if (p->Prox == NULL)
          Lista->Ultimo = p;
    free(q);
   }
 }
```



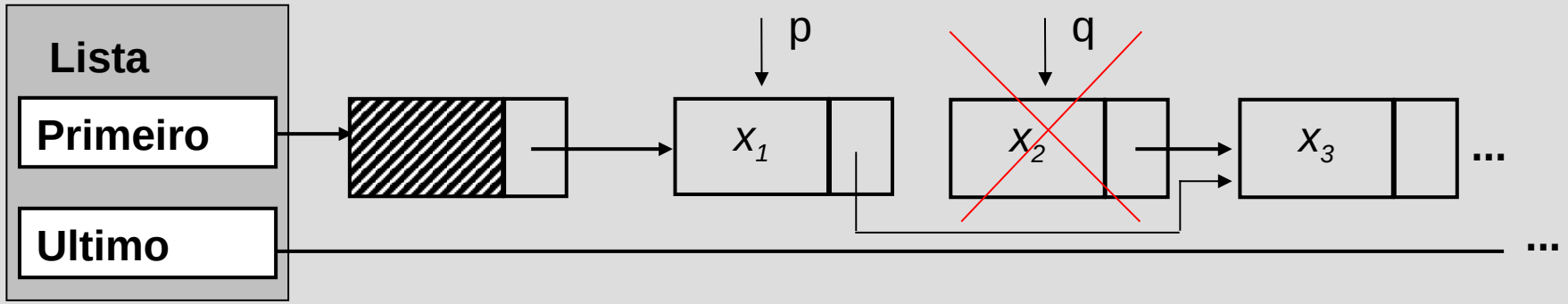
```
void Retira (Celula* p, TipoLista *Lista)
 {
  struct Celula* q;

  if (Vazia(*Lista)||p->Prox==NULL)
   {
    printf ("Erro: lista vazia ou posicao inexistente.\n");
    getch();
   }
  else
   {
    q = p->Prox;
    p->Prox = q->Prox;
    if (p->Prox == NULL)
          Lista->Ultimo = p;
    free(q);
   }
 }
```



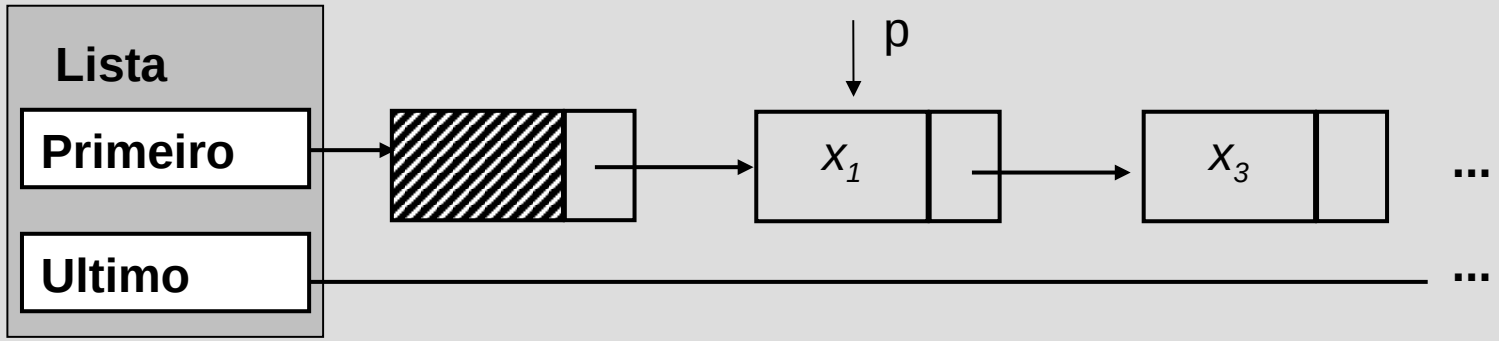
```
void Retira (Celula* p, TipoLista *Lista)
 {
  struct Celula* q;

  if (Vazia(*Lista)||p->Prox==NULL)
   {
    printf ("Erro: lista vazia ou posicao inexistente.\n");
    getch();
   }
  else
   {
    q = p->Prox;
    p->Prox = q->Prox;
    if (p->Prox == NULL)
          Lista->Ultimo = p;
    free(q);
   }
 }
```



```
void Retira (Celula* p, TipoLista *Lista)
 {
  struct Celula* q;

  if (Vazia(*Lista)||p->Prox==NULL)
   {
    printf ("Erro: lista vazia ou posicao inexistente.\n");
    getch();
   }
  else
   {
    q = p->Prox;
    p->Prox = q->Prox;
    if (p->Prox == NULL)
          Lista->Ultimo = p;
    free(q);
   }
 }
```

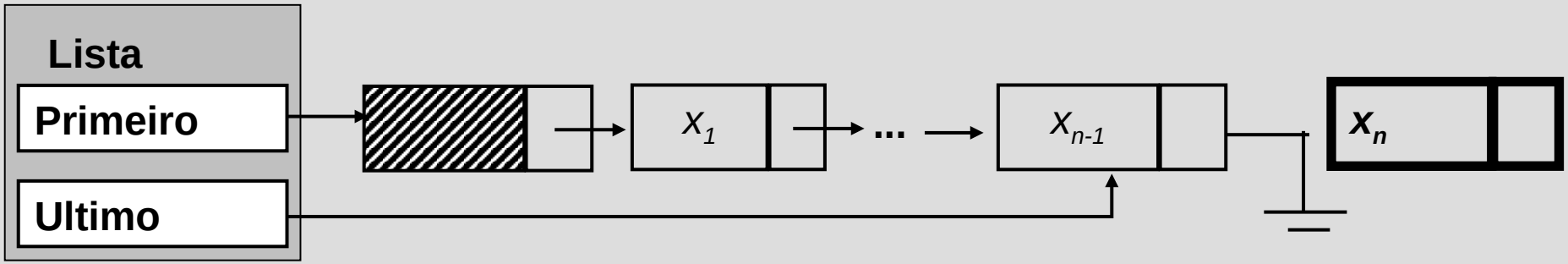


```
void Retira (Celula* p, TipoLista *Lista)
 {
  struct Celula* q;

  if (Vazia(*Lista)||p->Prox==NULL)
   {
    printf ("Erro: lista vazia ou posicao inexistente.\n");
    getch();
   }
  else
   {
    q = p->Prox;
    p->Prox = q->Prox;
    if (p->Prox == NULL)
          Lista->Ultimo = p;
    free(q);
   }
 }
```



Listas Lineares

void Imprime (TipoLista Lista);



Listas Lineares

```
void Imprime (TipoLista Lista)
 {
  struct Celula* Aux;
  int i=1;
  Aux = Lista.Primeiro -> Prox;
  while (Aux != NULL)
   {
    printf ("\n\nCodigo do elemento %d: %d", i, Aux->Item.chave);
    printf ("\nNome do elemento %d: %s", i, Aux->Item.nome);
    printf ("\nIdade do elemento %d: %d", i, Aux->Item.idade);
    getch();
    Aux=Aux->Prox;
    i++;
   }
 }
```