Listas Lineares

Livro “Projeto de Algoritmos” – Nívio Ziviani
Capítulo 3 – Seção 3.1
http://www2.dcc.ufmg.br/livros/algoritmos/
(adaptado)

Agenda

- Listas lineares
- TAD para listas lineares
- Alocação sequencial
- Alocação encadeada

Listas Lineares

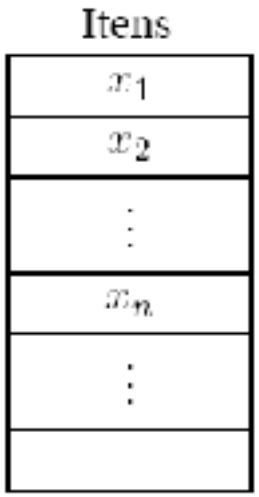
- Maneira de representar elementos de um conjunto.
- Itens podem ser acessados, inseridos ou retirados de uma lista.
- Podem crescer ou diminuir de tamanho durante a execução.
- Adequadas quando não é possível prever a demanda por memória

Definição de Listas Lineares

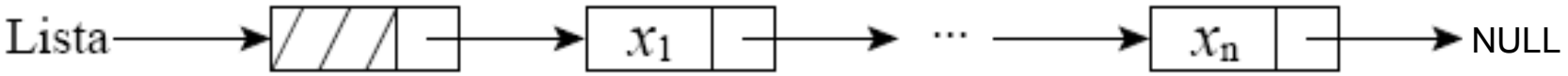
- **Sequência de zero ou mais itens**
  - $x_1, x_2, \cdots, x_n$, na qual x_i é de um determinado tipo e n representa o tamanho da lista linear.

- **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.
  - x_i precede x_{i+1} para $i = 1,2,\cdots,n-1$
  - x_i sucede x_{i-1} para $i = 2,3,\cdots,n$
  - o elemento x_i é dito estar na i-ésima posição da lista.

Listas Lineares

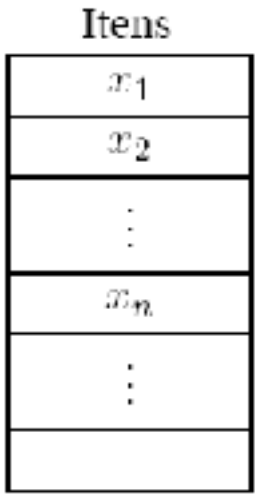



```
struct Aluno {
    string nome;
    int matricula;
    char conceito;
};
```

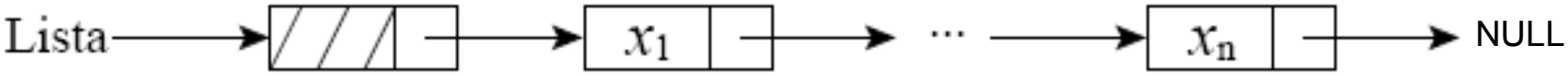


TAD Listas Lineares

- TAD: Agrupa a ***estrutura de dados*** juntamente com as ***operações*** que podem ser feitas sobre esses dados.
- Operações para listas lineares.




```
struct Aluno {
    string nome;
    int matricula;
    char conceito;
};
```



TAD Listas Lineares

- **O conjunto de operações a ser definido depende de cada aplicação.**
- **Um conjunto de operações necessário a uma maioria de aplicações é:**
  1) Criar uma lista linear vazia.
  2) Inserir um novo item imediatamente após o i-ésimo item.
  3) Retirar o i-ésimo item.
  4) Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
  5) Combinar duas ou mais listas lineares em uma lista única.
  6) Partir uma lista linear em duas ou mais listas.
  7) Fazer uma cópia da lista linear.
  8) Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
  9) Pesquisar a ocorrência de um item com um valor particular em algum componente.

Implementações de Listas Lineares

- **As duas representações mais utilizadas são as implementações**
  - **Alocação sequencial (arranjo) e encadeada.**

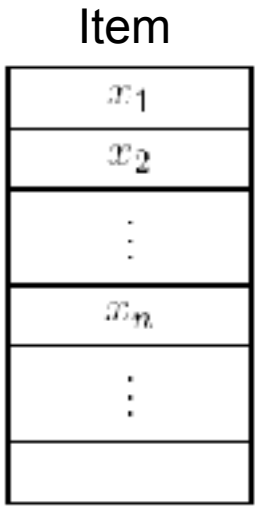
- **Exemplo de Conjunto de Operações:**
  1) FLVazia(Lista). Faz a lista ficar vazia.
  2) Insere(x, Lista). Insere x após o último item da lista.
  3) Retira(p, Lista, x). Retorna o item x que está na posição p da lista, retirando-o da lista e deslocando os itens a partir da posição p+1 para as posições anteriores.
  4) Vazia(Lista). Esta função retorna true se lista vazia; senão retorna false.
  5) Imprime(Lista). Imprime os itens da lista na ordem de ocorrência.

Alocação Sequencial

- Localização na memória:
  - Posições contíguas.
- Visita:
  - Pode ser percorrida em qualquer direção.
  - Permite acesso aleatório.
- Inserção:
  - Realizada após o último item com custo constante.
  - Um novo item no meio da lista custo não constante.
- Remoção:
  - Final da lista: custo constante
  - Meio ou início: requer deslocamento de itens

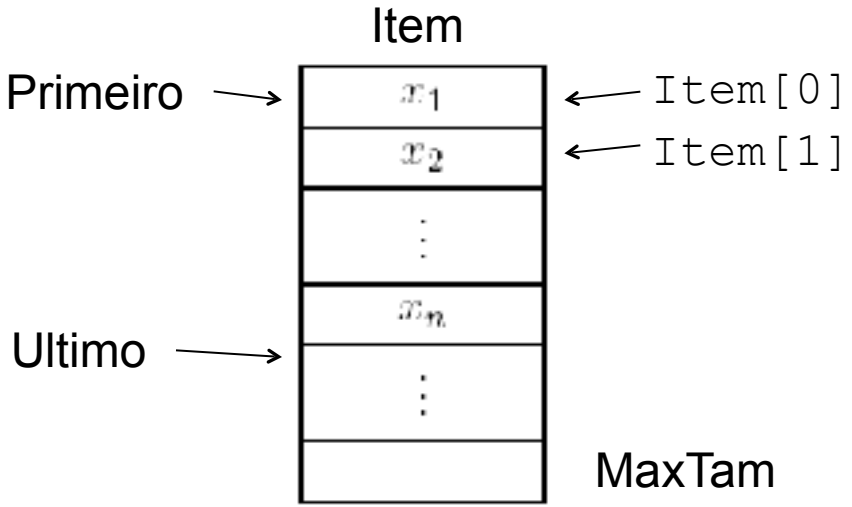
Alocação Sequencial (estrutura)

- Os itens armazenados em um array.
- MaxTam define o tamanho máximo permitido para a lista.
- O campo Último aponta para a posição seguinte a do último elemento da lista. (primeira posição vazia)
- O *i-ésimo item da lista está armazenado na i-ésima-1 posição do array, 0 ≤ i < Último.* `(Item[i])`



Alocação Sequencial (estrutura)

- Os itens armazenados em um array.
- MaxTam define o tamanho máximo permitido para a lista.
- O campo Último aponta para a posição seguinte a do último elemento da lista. (primeira posição vazia)
- O *i-ésimo item da lista está armazenado na i-ésima-1 posição do array, 0 ≤ i < Último.* `(Item[i])`



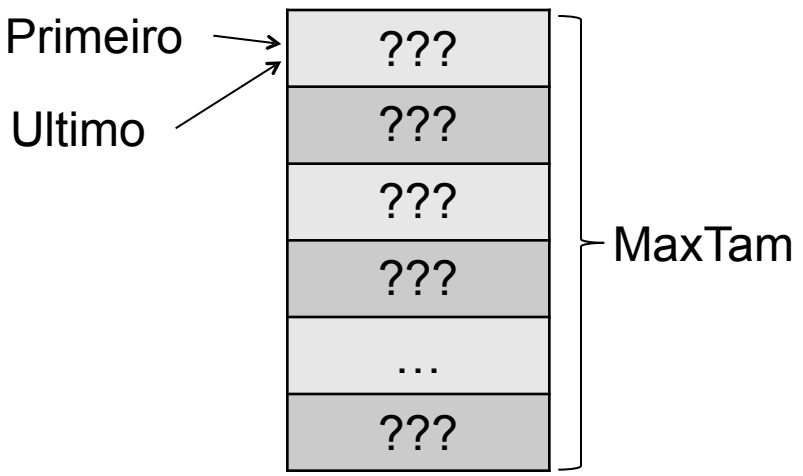
Código!!

Alocação Sequencial (operações)

```
/* faz lista ficar vazia */
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = InicioArranjo;
  Lista->Ultimo = Lista->Primeiro;
}  /* FLVazia */
```

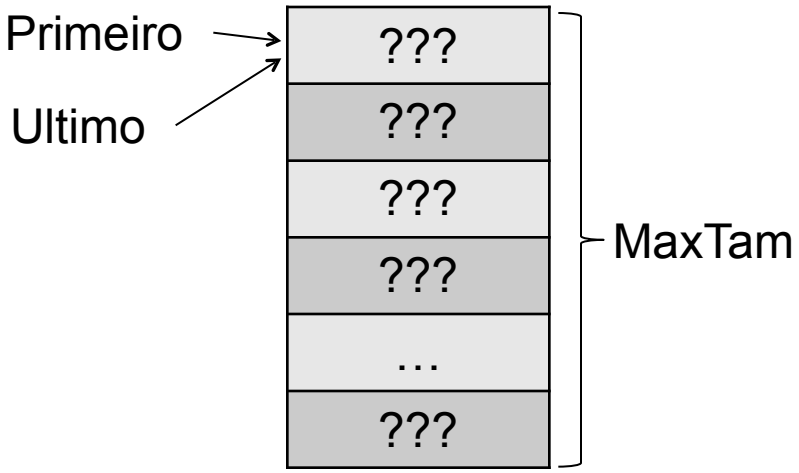
Alocação Sequencial (operações)

```
/* faz lista ficar vazia */
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = InicioArranjo;
  Lista->Ultimo = Lista->Primeiro;
}  /* FLVazia */
```



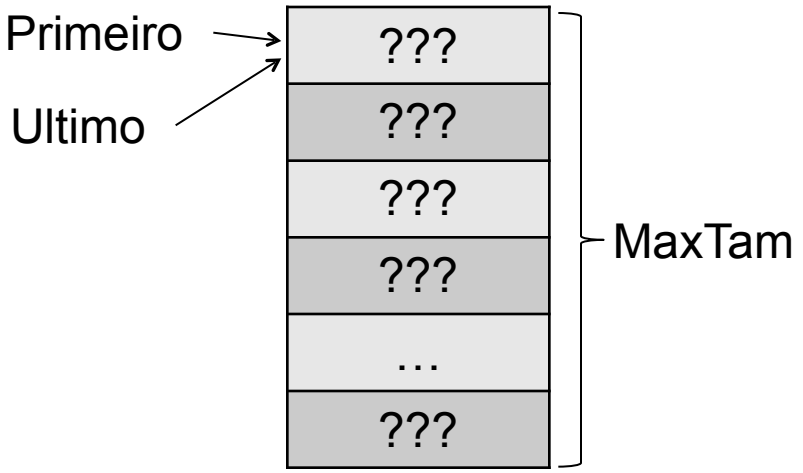
Alocação Sequencial (operações)

```
/* faz lista ficar vazia */
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = InicioArranjo;
  Lista->Ultimo = Lista->Primeiro;
}  /* FLVazia */


/* testa se a lista está vazia */
int Vazia(TipoLista *Lista)
{
  return (Lista->Primeiro == Lista->Ultimo);
}  /* Vazia */
```



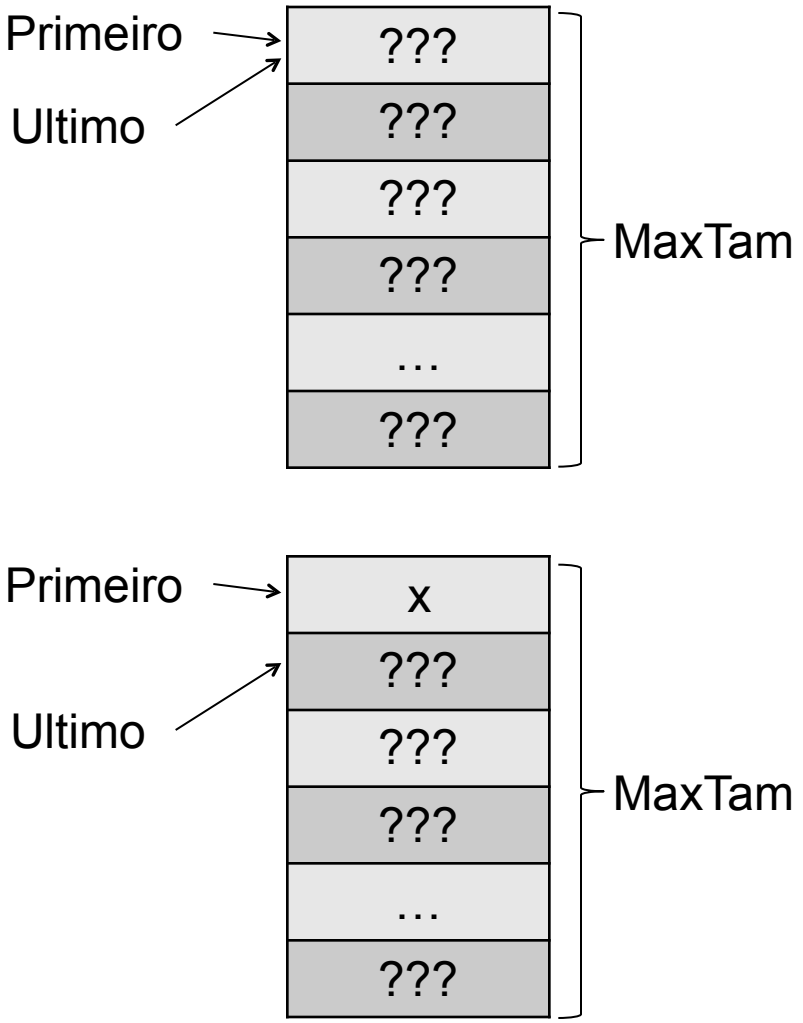
Alocação Sequencial (operações)

```
void Insere(TipoItem x, TipoLista *Lista)
{
  if (Lista->Ultimo >= MaxTam)
     printf("Lista esta cheia\n");
  else
  {
     Lista->Item[Lista->Ultimo] = x;
     Lista->Ultimo++;
  }
}  /* Insere */
```



Alocação Sequencial (operações)

```
void Insere(TipoItem x, TipoLista *Lista)
{
  if (Lista->Ultimo >= MaxTam)
     printf("Lista esta cheia\n");
  else
  {
     Lista->Item[Lista->Ultimo] = x;
     Lista->Ultimo++;
  }
}  /* Insere */
```

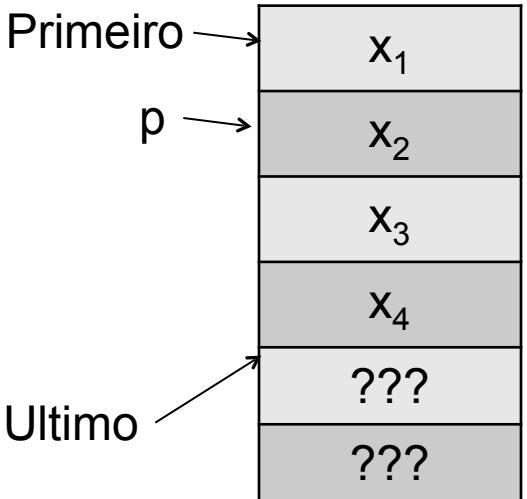


Alocação Sequencial (operações)

```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item) {
  int Aux;

  if (Vazia(Lista) || p >= Lista->Ultimo) {
    printf("Erro: Posicao nao existe\n");
    return;
  }
  *Item = Lista->Item[p];
  Lista->Ultimo--;
  for (Aux = p+1; Aux <= Lista->Ultimo; Aux++)
    Lista->Item[Aux - 1] = Lista->Item[Aux];
}
```

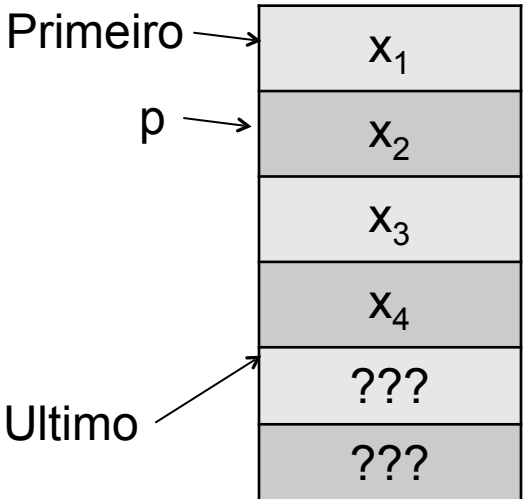
Alocação Sequencial (operações)

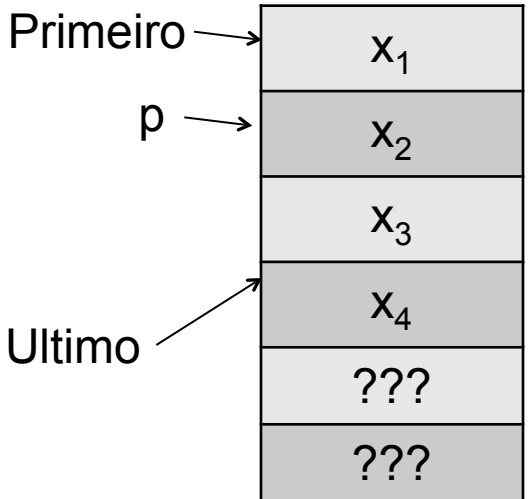
```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item) {
  int Aux;

  if (Vazia(Lista) || p >= Lista->Ultimo) {
    printf("Erro: Posicao nao existe\n");
    return;
  }
  *Item = Lista->Item[p];
  Lista->Ultimo--;
  for (Aux = p+1; Aux <= Lista->Ultimo; Aux++)
    Lista->Item[Aux - 1] = Lista->Item[Aux];
}
```



Alocação Sequencial (operações)

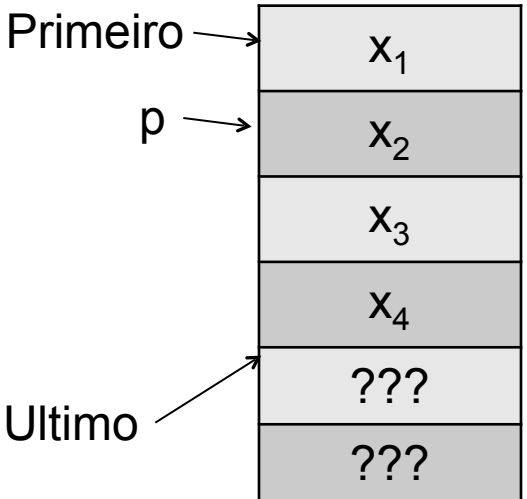
```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item) {
  int Aux;

  if (Vazia(Lista) || p >= Lista->Ultimo) {
    printf("Erro: Posicao nao existe\n");
    return;
  }
  *Item = Lista->Item[p];
  Lista->Ultimo--;
  for (Aux = p+1; Aux <= Lista->Ultimo; Aux++)
    Lista->Item[Aux - 1] = Lista->Item[Aux];
}
```

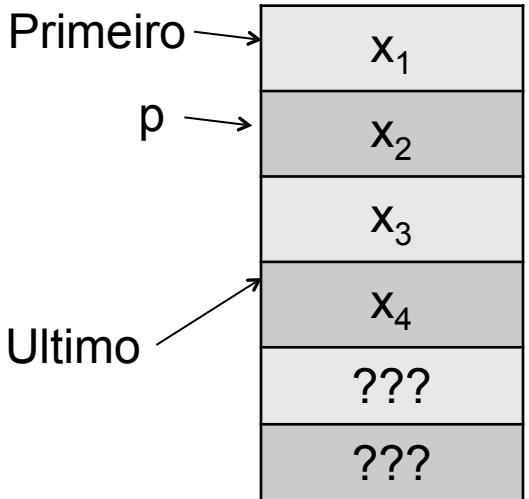


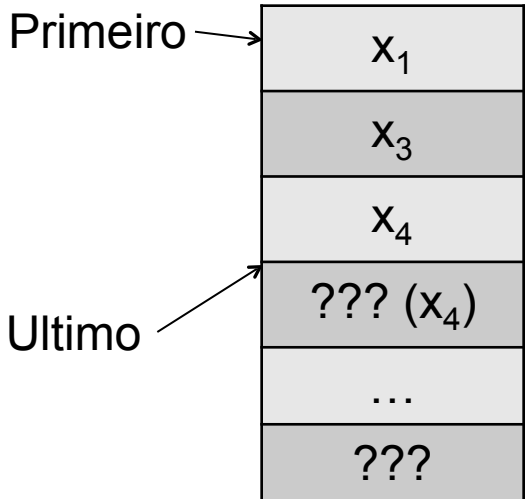


Alocação Sequencial (operações)

```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item) {
  int Aux;

  if (Vazia(Lista) || p >= Lista->Ultimo) {
    printf("Erro: Posicao nao existe\n");
    return;
  }
  *Item = Lista->Item[p];
  Lista->Ultimo--;
  for (Aux = p+1; Aux <= Lista->Ultimo; Aux++)
    Lista->Item[Aux - 1] = Lista->Item[Aux];
}
```

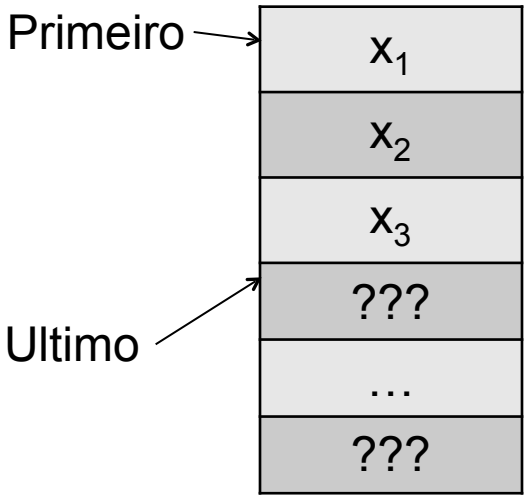






Alocação Sequencial (operações)

```
void Imprime(TipoLista *Lista)
{
  Apontador i;

  for (i = Lista->Primeiro; i < Lista->Ultimo; i++)

     printf("%d\n", Lista->Item[i].Chave);
}
```



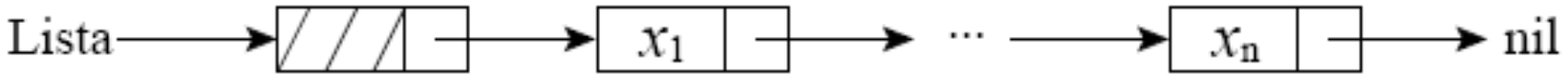
Alocação Sequencial (vantagens e desvantagens)

- Vantagens:
  - economia de memória (os apontadores são implícitos nesta estrutura).
  - Acesso a um item qualquer é O(1).

- Desvantagens:
  - custo para inserir ou retirar itens da lista, que **pode causar um deslocamento de todos os itens**, no pior caso; O(n)
  - em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o Pascal pode ser problemática porque neste caso o **tamanho máximo** da lista tem de ser definido em **tempo de compilação**.

Alocação Encadeada

- Cada item é encadeado com o seguinte mediante uma variável do tipo Apontador.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma ***célula cabeça*** para simplificar as operações sobre a lista.



Alocação Encadeada

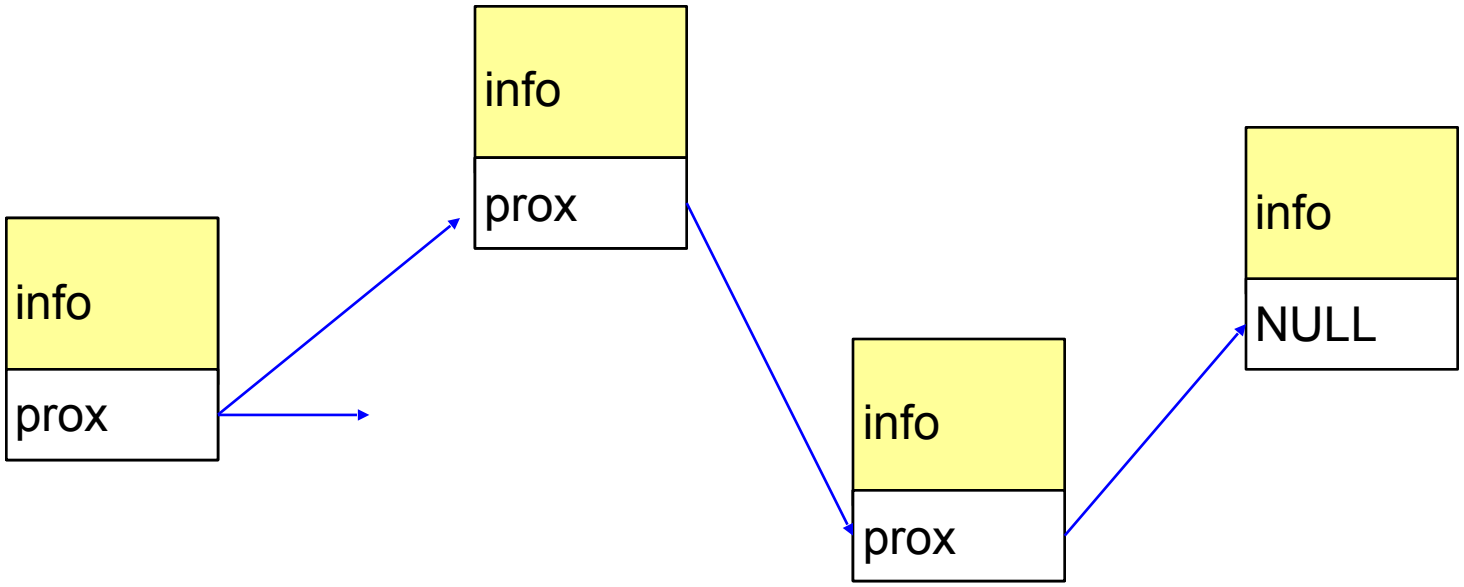
- A lista é constituída de células.
- Cada célula contém um item da lista e um apontador para a célula seguinte.
- O registro TipoLista contém um apontador para a célula cabeça e um apontador para a última célula da lista.

Alocação Encadeada

- Localização na memória:
  - Posições não sequenciais.
- Visita:
  - Apenas na direção x_i para x_{i+1}.
  - Permite apenas acesso sequencial.
- Inserção:
  - Realizada em qualquer posição com custo constante.
- Remoção:
  - Custo constante em qualquer posição.

Alocação Encadeada

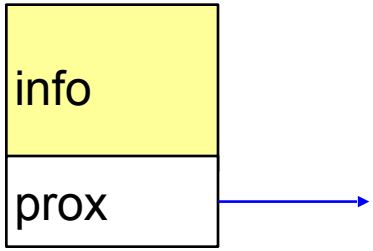
- Características:
  - Tamanho da lista não é pré-definido
  - Cada elemento guarda quem é o próximo
  - Elementos não estão contíguos na memória



Sobre os Elementos da Lista

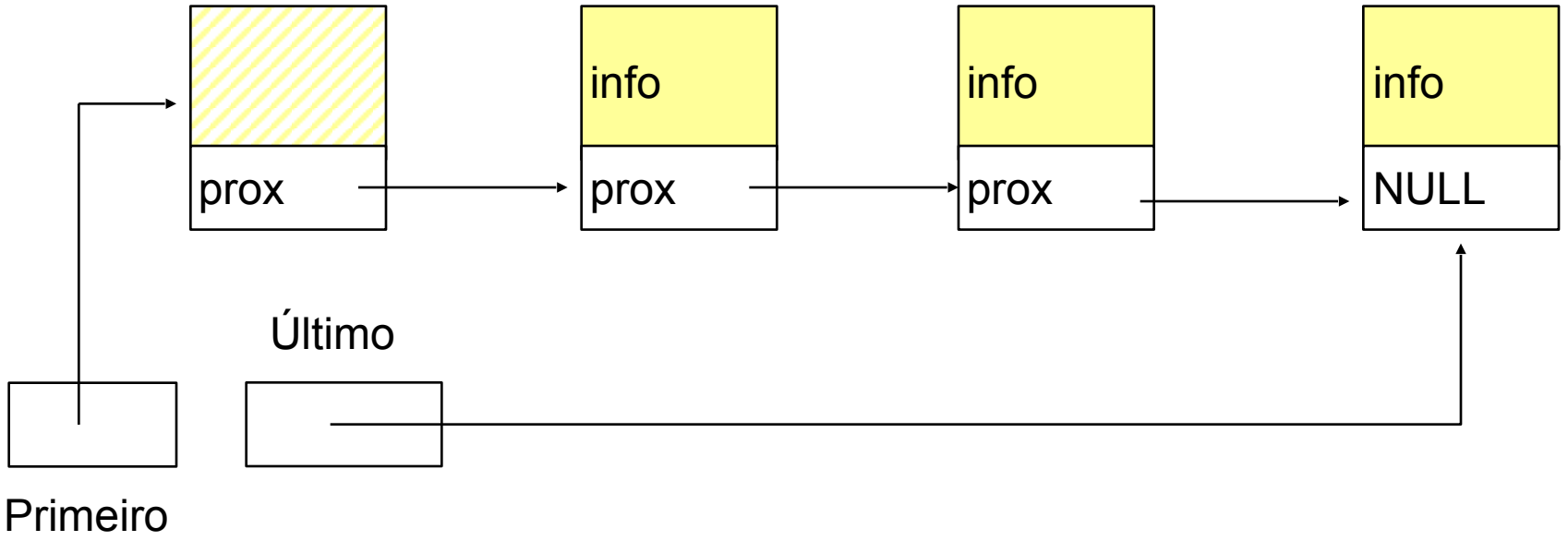
- Elemento: guarda as informações sobre cada elemento.

- Para isso define-se cada elemento como uma estrutura que possui:
  - campos de informações
  - ponteiro para o próximo elemento



Sobre a Lista

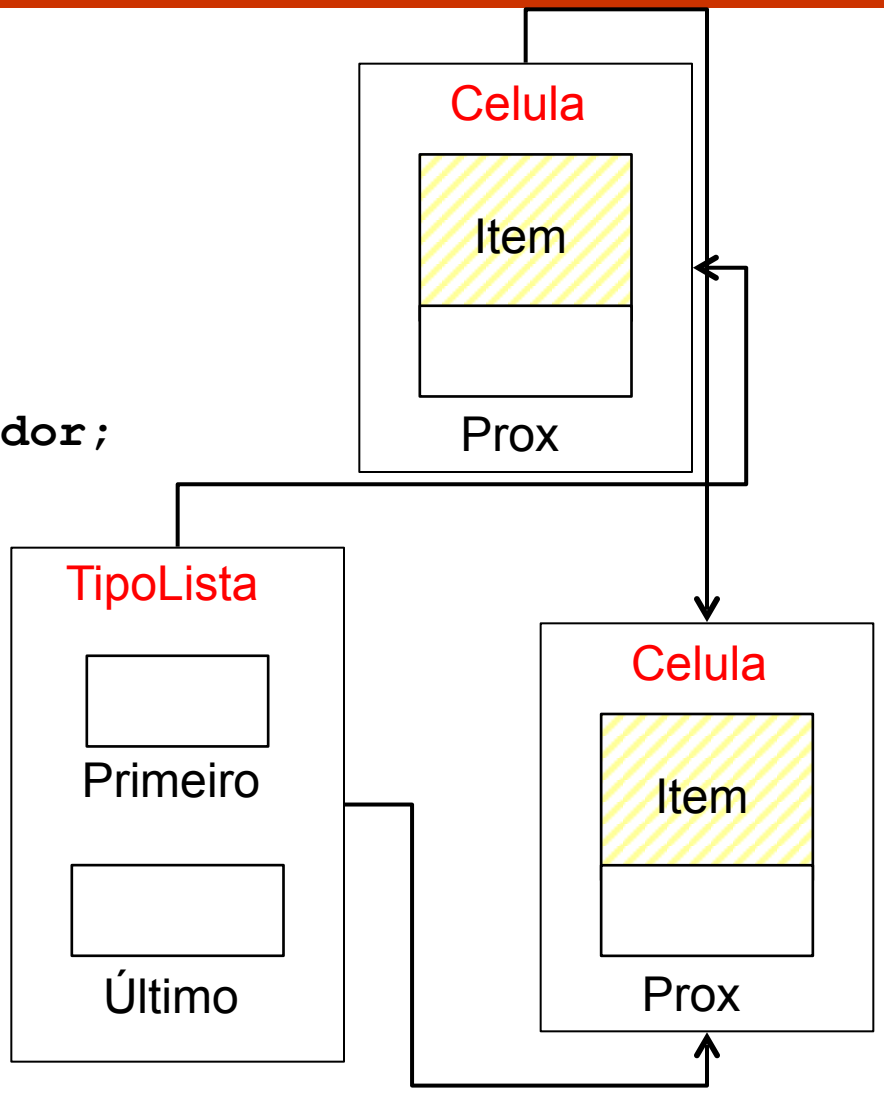
- Uma lista é definida como um apontador para a primeira célula.
- Uma lista pode ter uma célula cabeça.



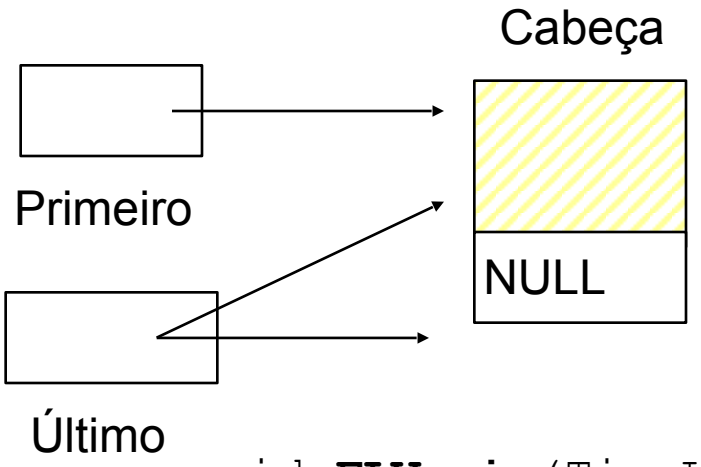
- Uma lista pode ter um apontador para o último elemento

Implementação em C

```
typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct Celula_str *Apontador;

typedef struct Celula_str {
  TipoItem Item;
  Apontador Prox;
} Celula;

typedef struct {
  Apontador Primeiro, Ultimo;
} TipoLista;
```

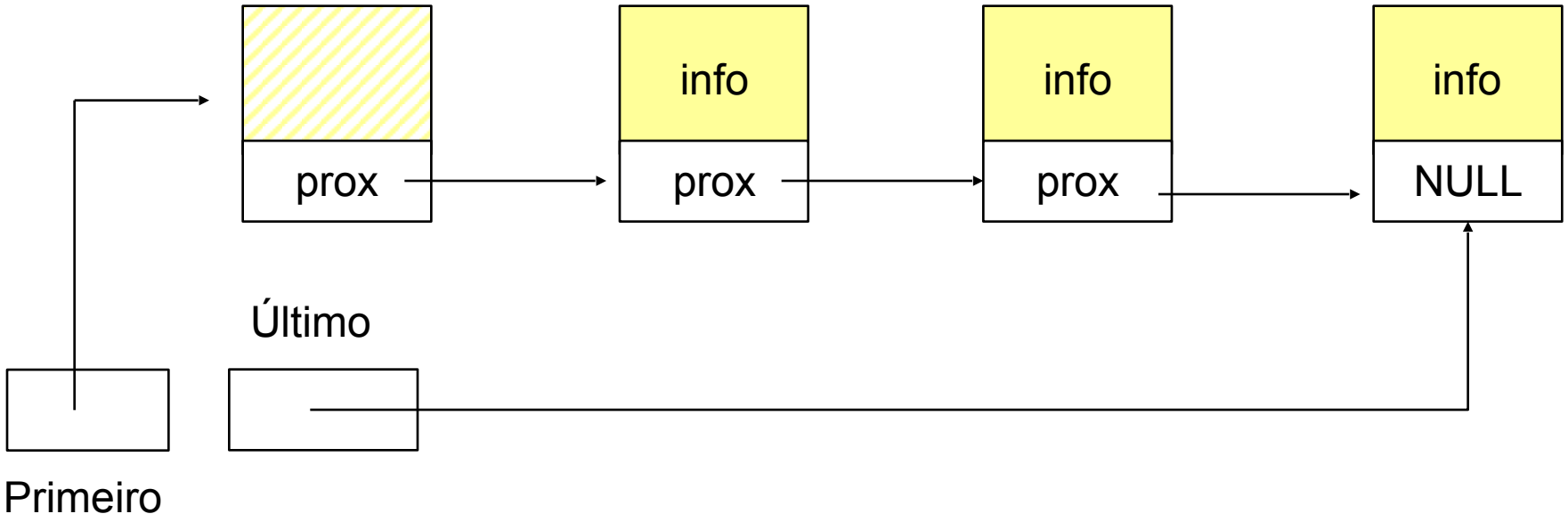


Cria Lista Vazia



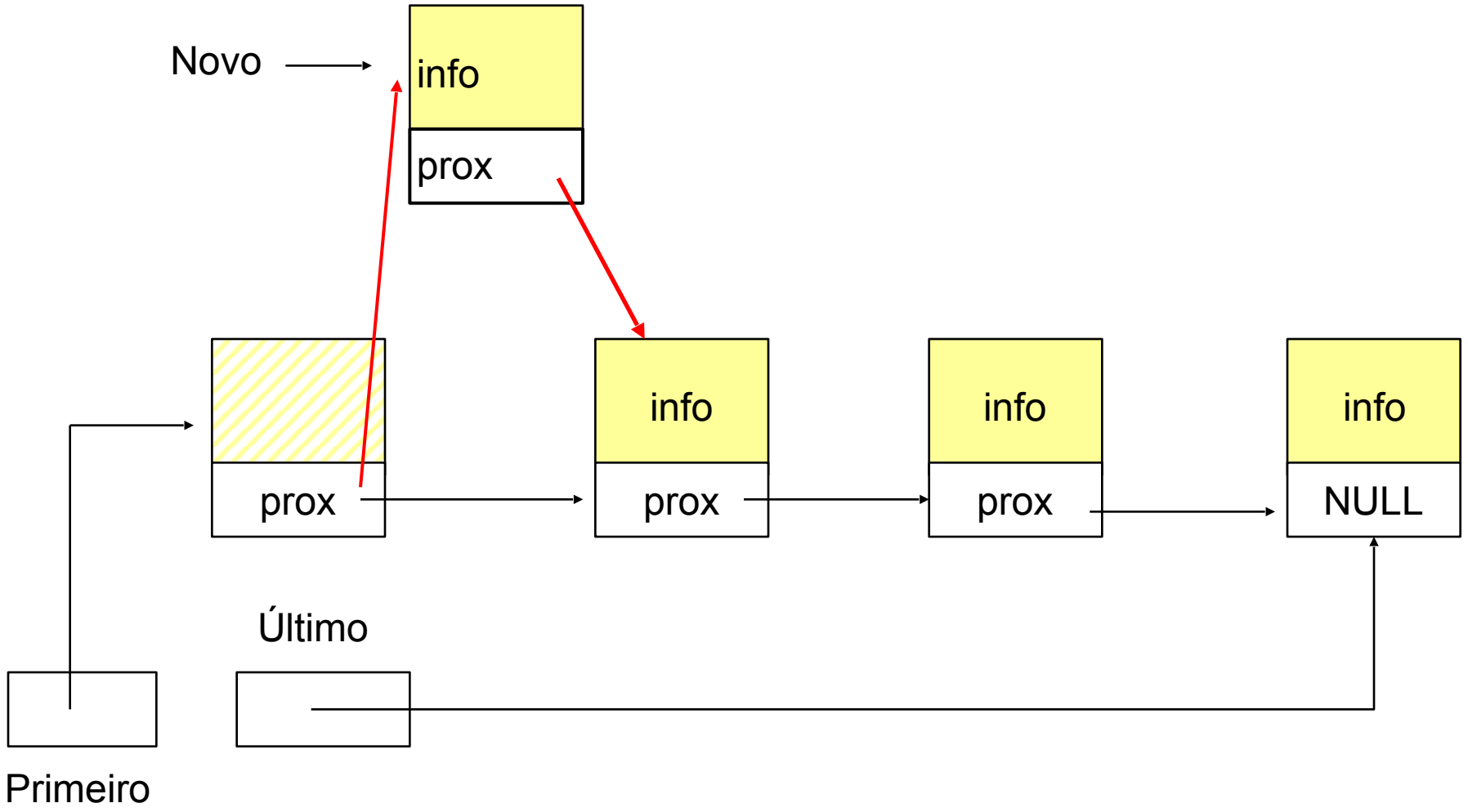

```
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = (Apontador) malloc(sizeof(Celula));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}

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

Inserção de Elementos na Lista

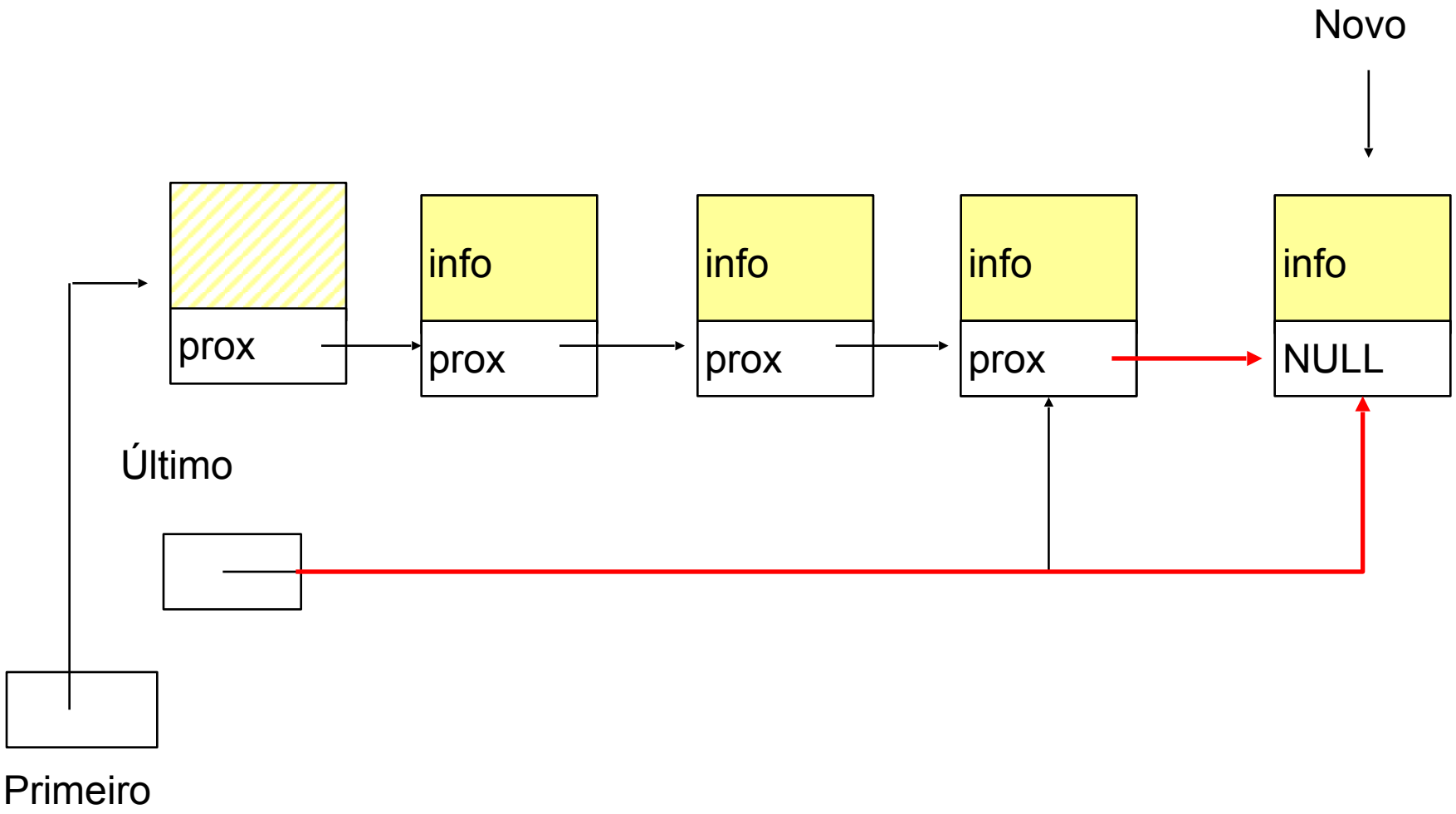


- 3 opções de posição onde pode inserir:
  - 1ª. posição
  - última posição
  - Após um elemento qualquer E

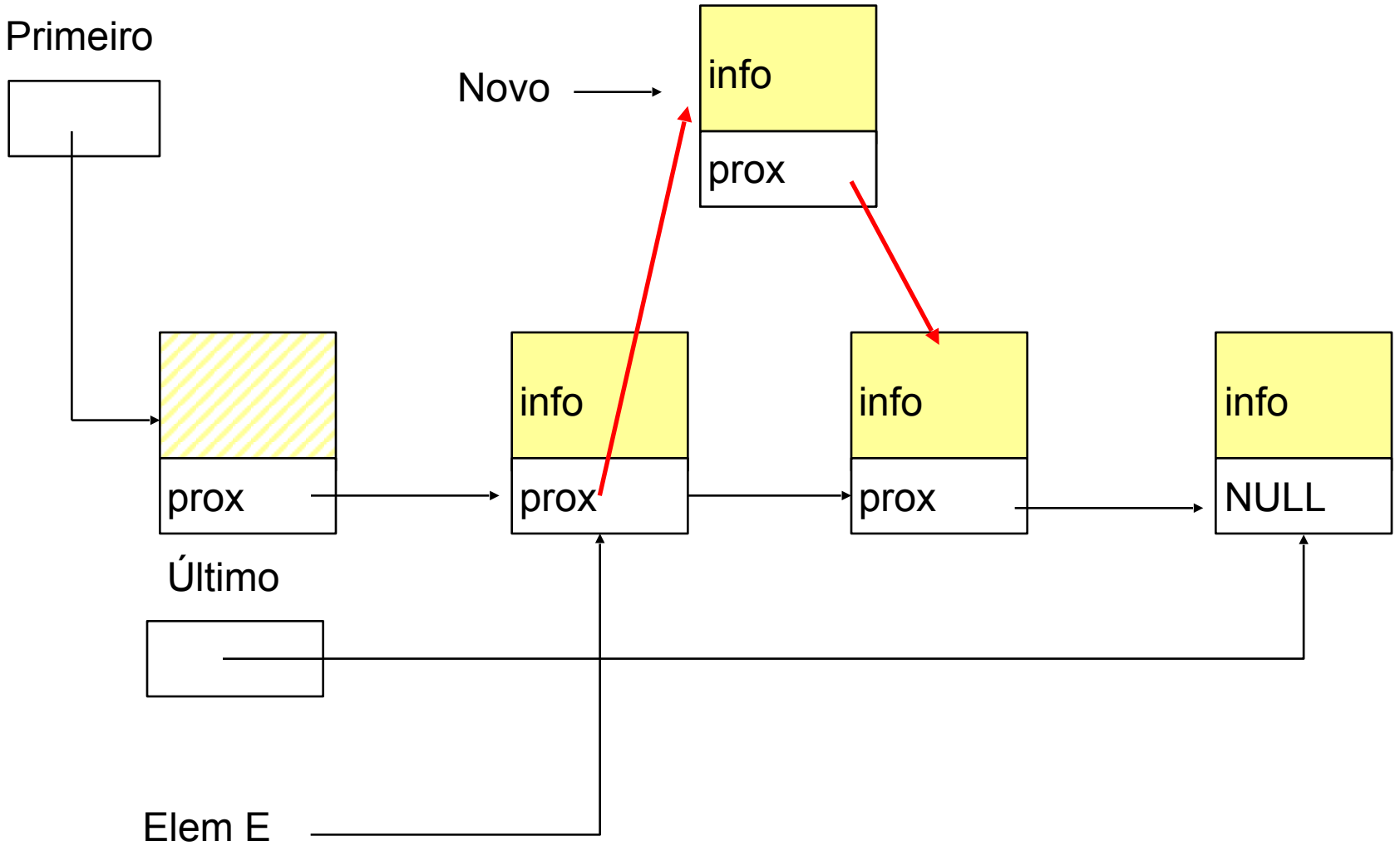
Inserção na Primeira Posição



Inserção na Última Posição



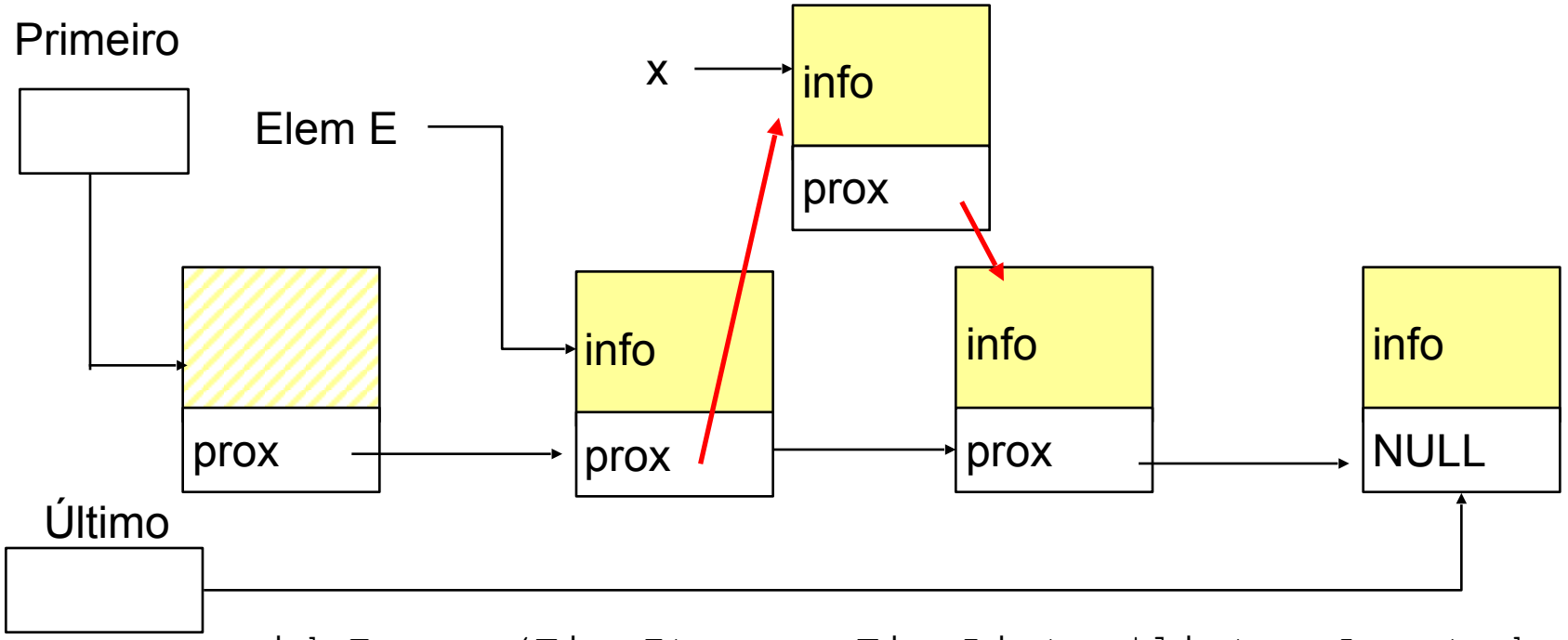
Inserção na Após o Elemento E



Inserção de Elementos na Lista

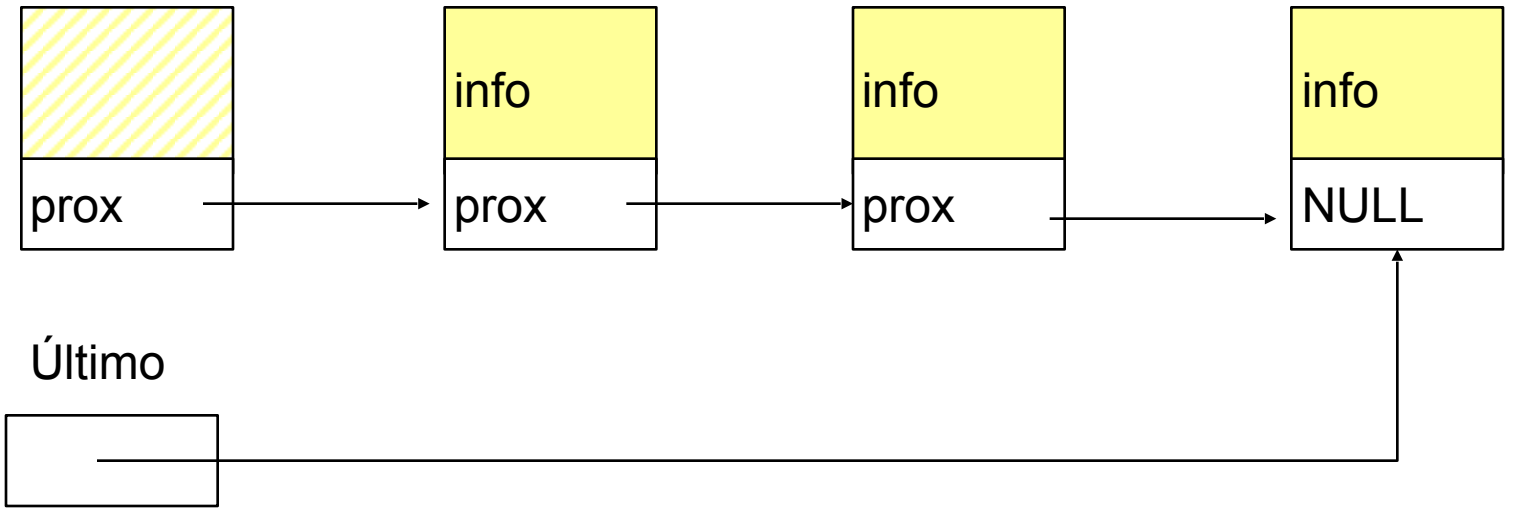
- Na verdade, as 3 opções de inserção são equivalentes a inserir após uma célula apontada por **p**
  - 1ª. posição (p é a célula cabeça)
  - Última posição (p é o último)
  - Após um elemento qualquer E (p aponta para E)

Inserção na Após o Elemento E



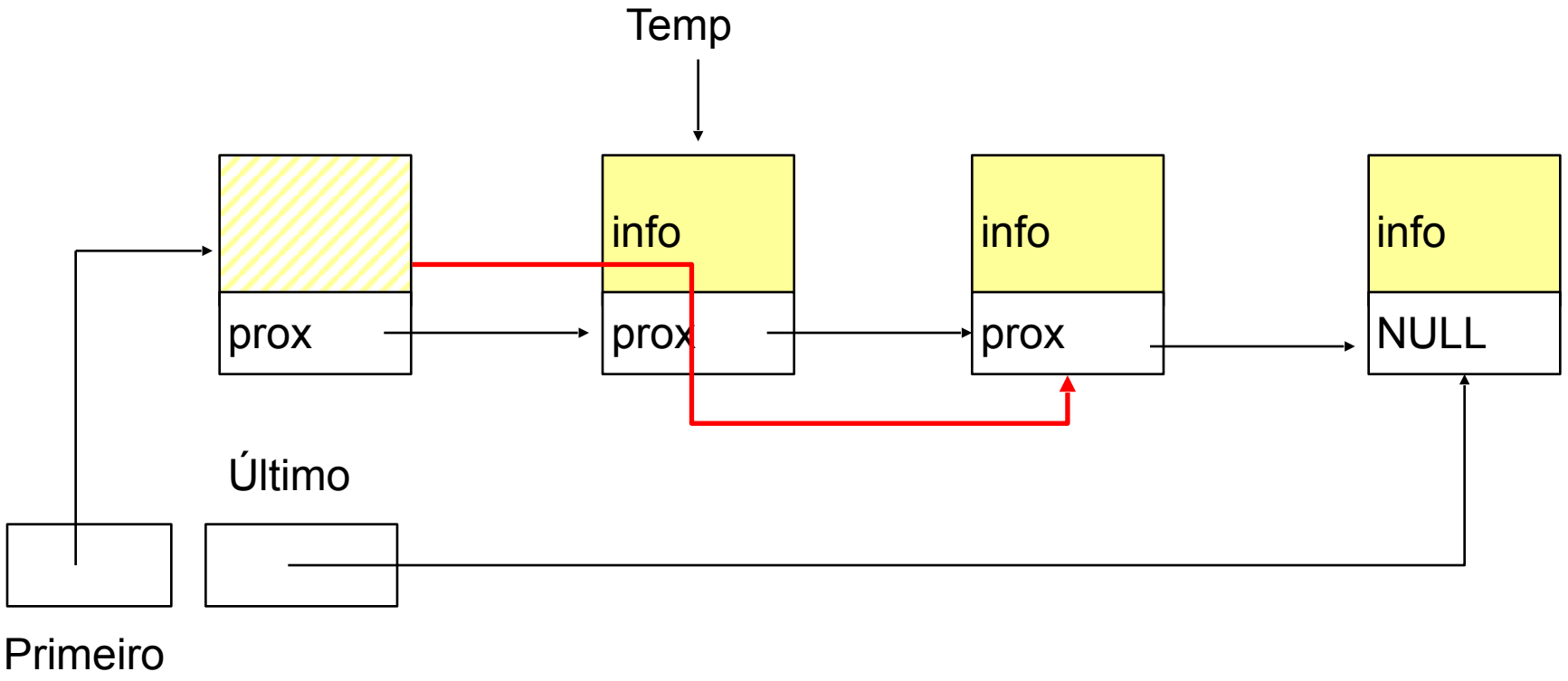

```
void Insere(TipoItem x, TipoLista *lista, Apontador E){
Apontador novo;

   novo = (Apontador) malloc(sizeof(Celula));
          novo->Item = x;
   novo->prox = E->prox;
   E->prox = novo;
}
```

Retirada de Elementos na Lista

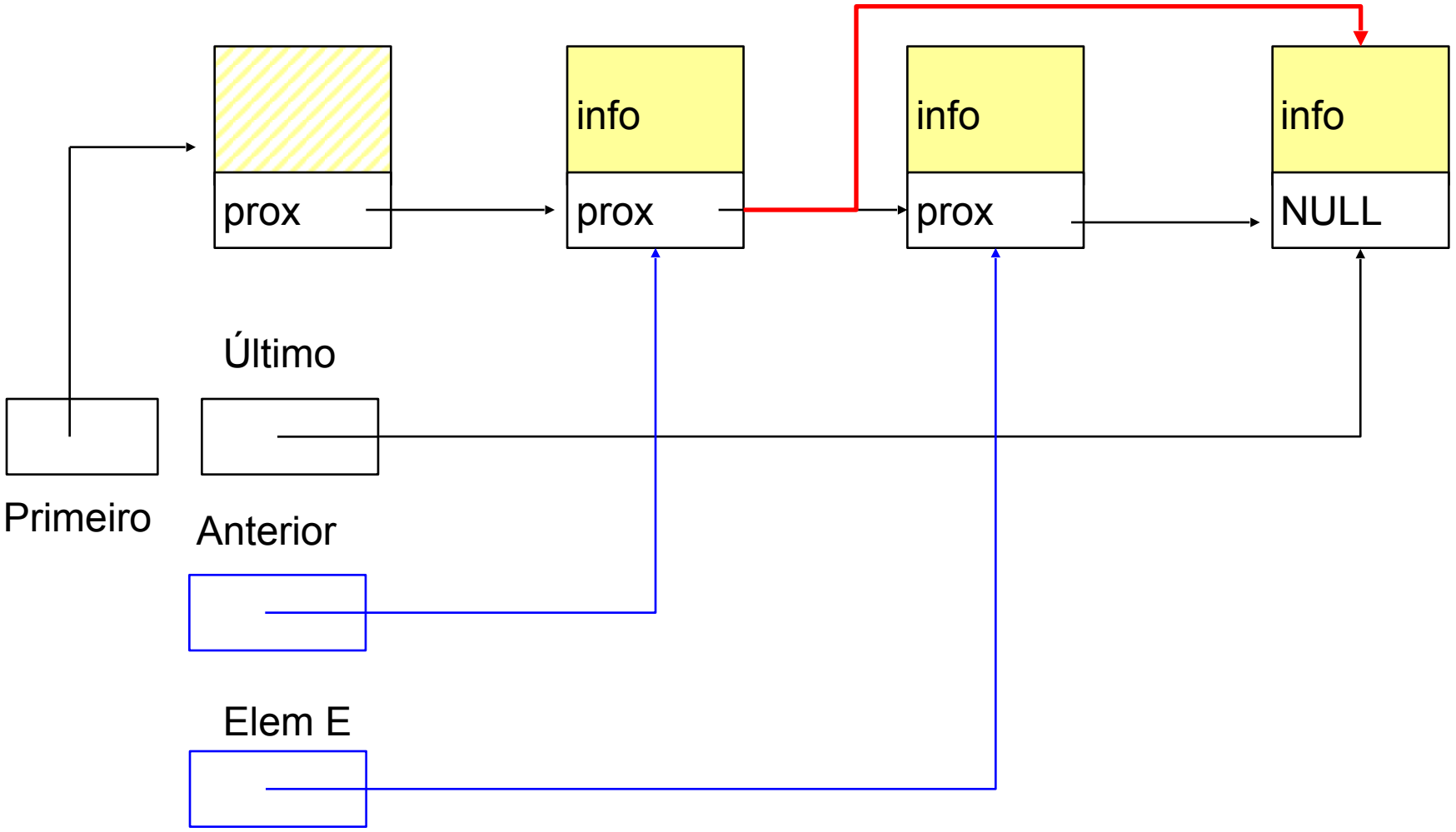


- 3 opções de posição de onde pode retirar:
  - 1ª. posição
  - última posição
  - Um elemento qualquer E

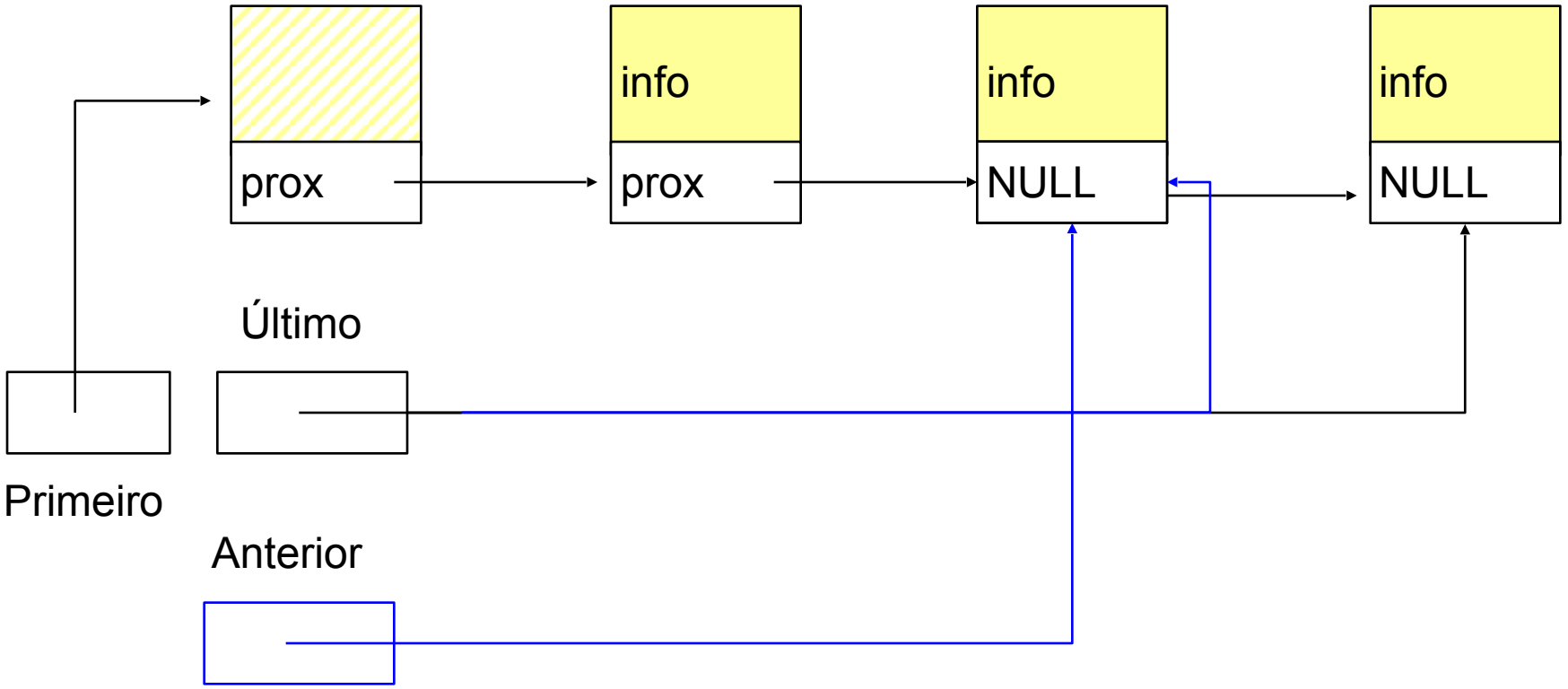
Retirada do Elemento na Primeira Posição da Lista



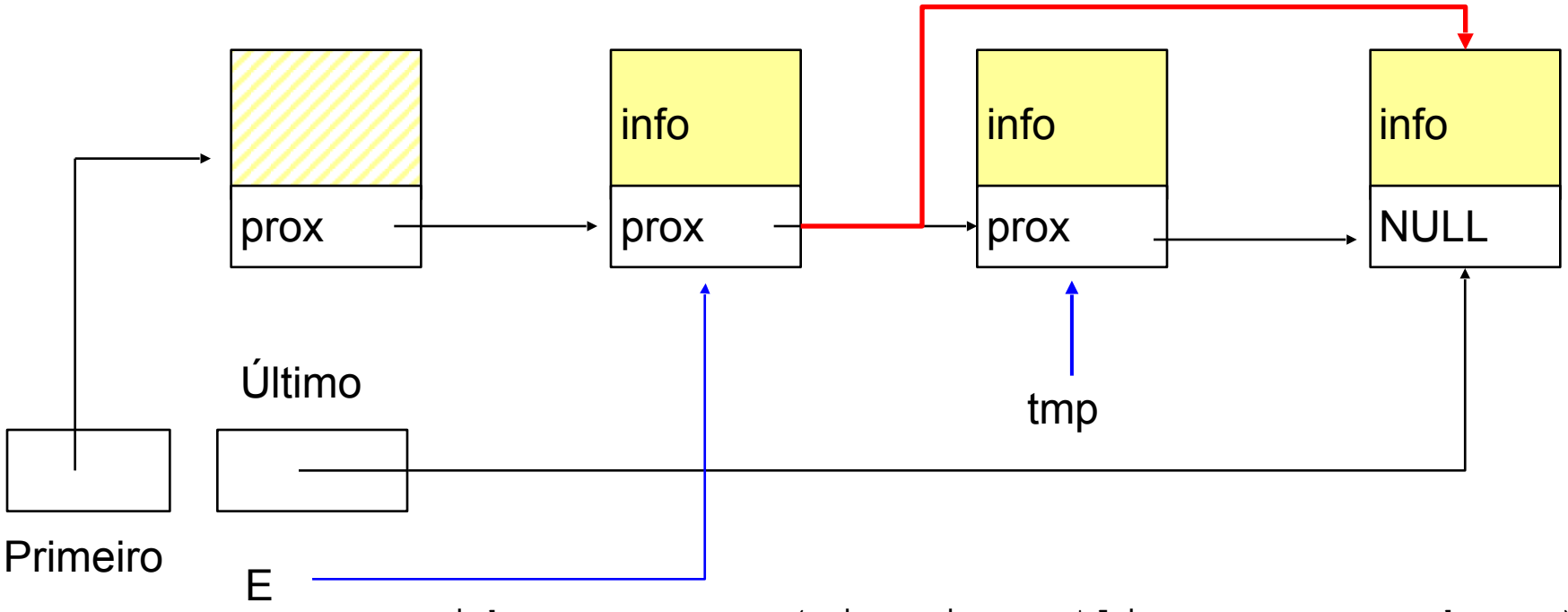
Retirada do Elemento E da Lista



Retirada do Último Elemento da Lista



Retirada do elemento após E da Lista




```
void RemoveProx(TipoLista *lista, Apontador E){
Apontador tmp;

    tmp = E->prox;
    if (tmp != NULL) {
              E->prox = tmp->prox;
       free(tmp);
    } else { E->prox = NULL; }
}
```

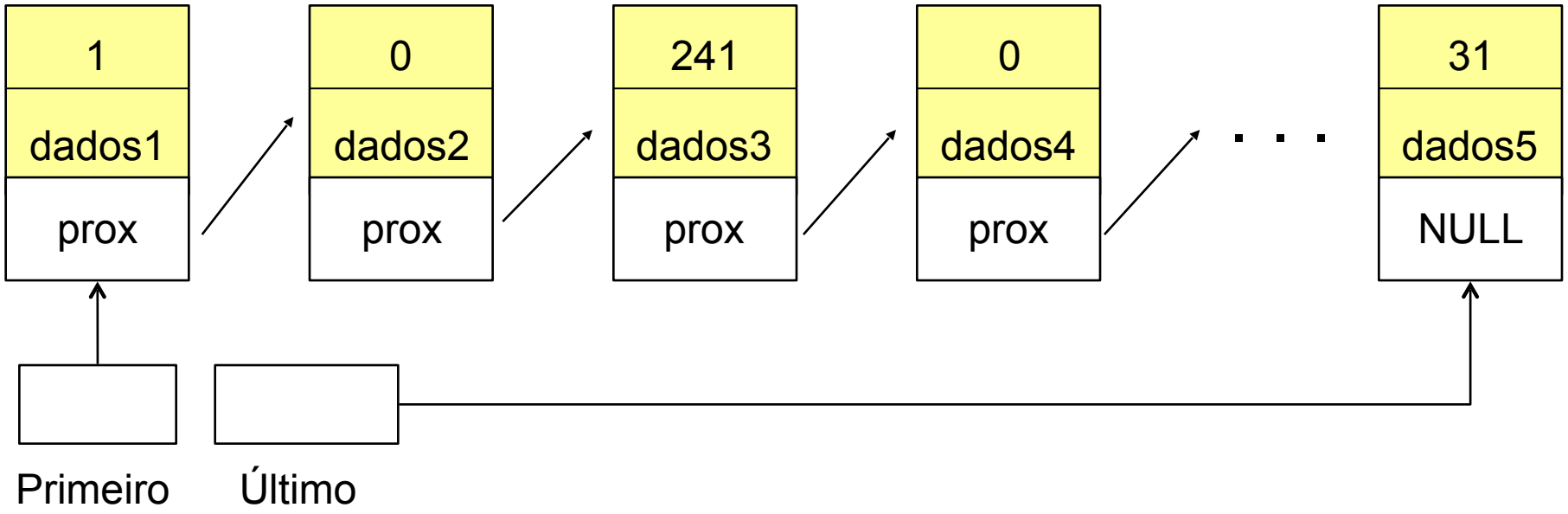
Alocação Encadeada (vantagens e desvantagens)

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido a priori).

- Desvantagem:
  - Utilização de memória extra para armazenar os apontadores.
  - O(n) para acessar um item no pior caso

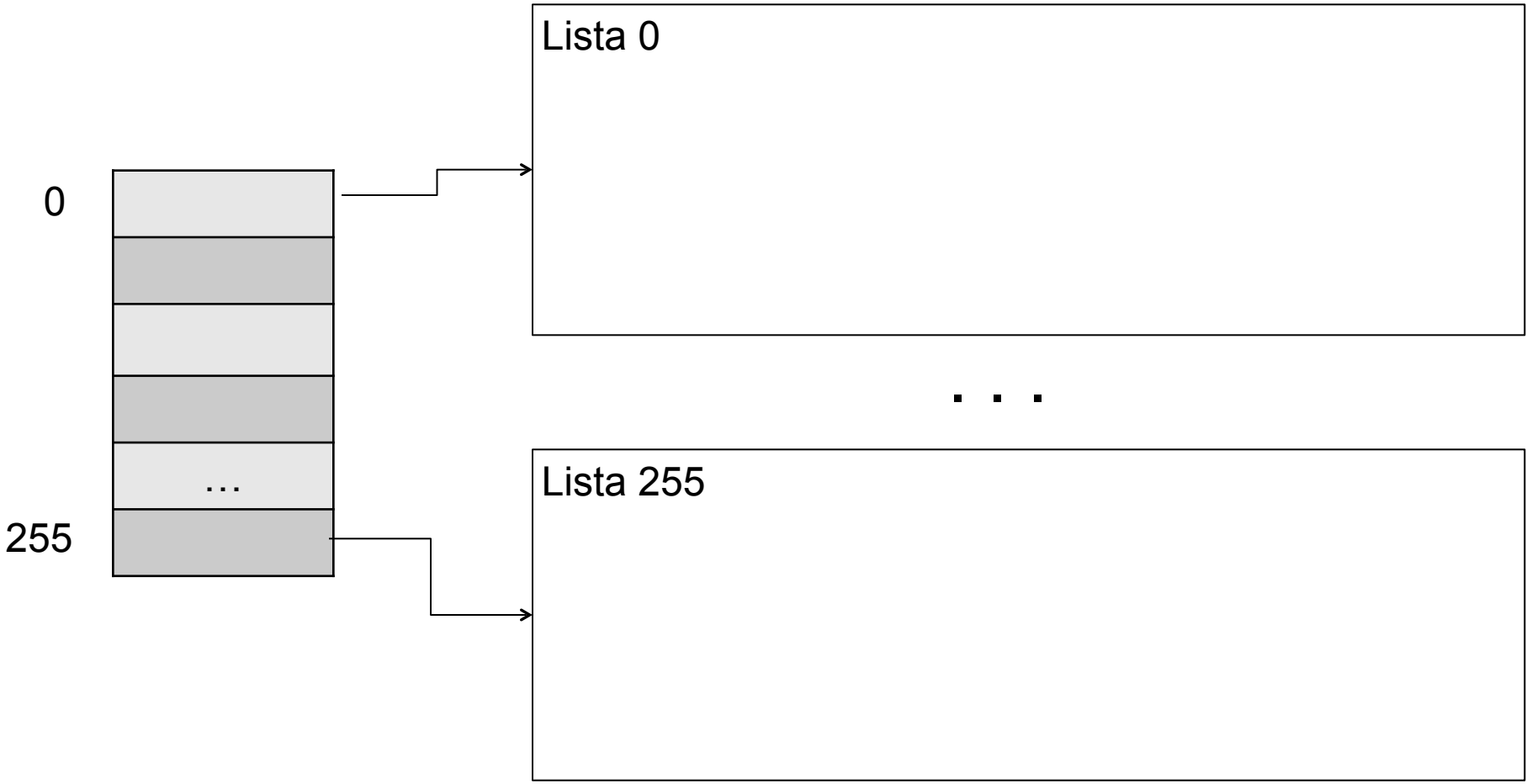
Exemplo: Ordenação

- **Problema:** Ordenar uma lista com alocação encadeada em *tempo linear.* Esta lista apresenta chaves inteiras com valores entre 0 e 255.



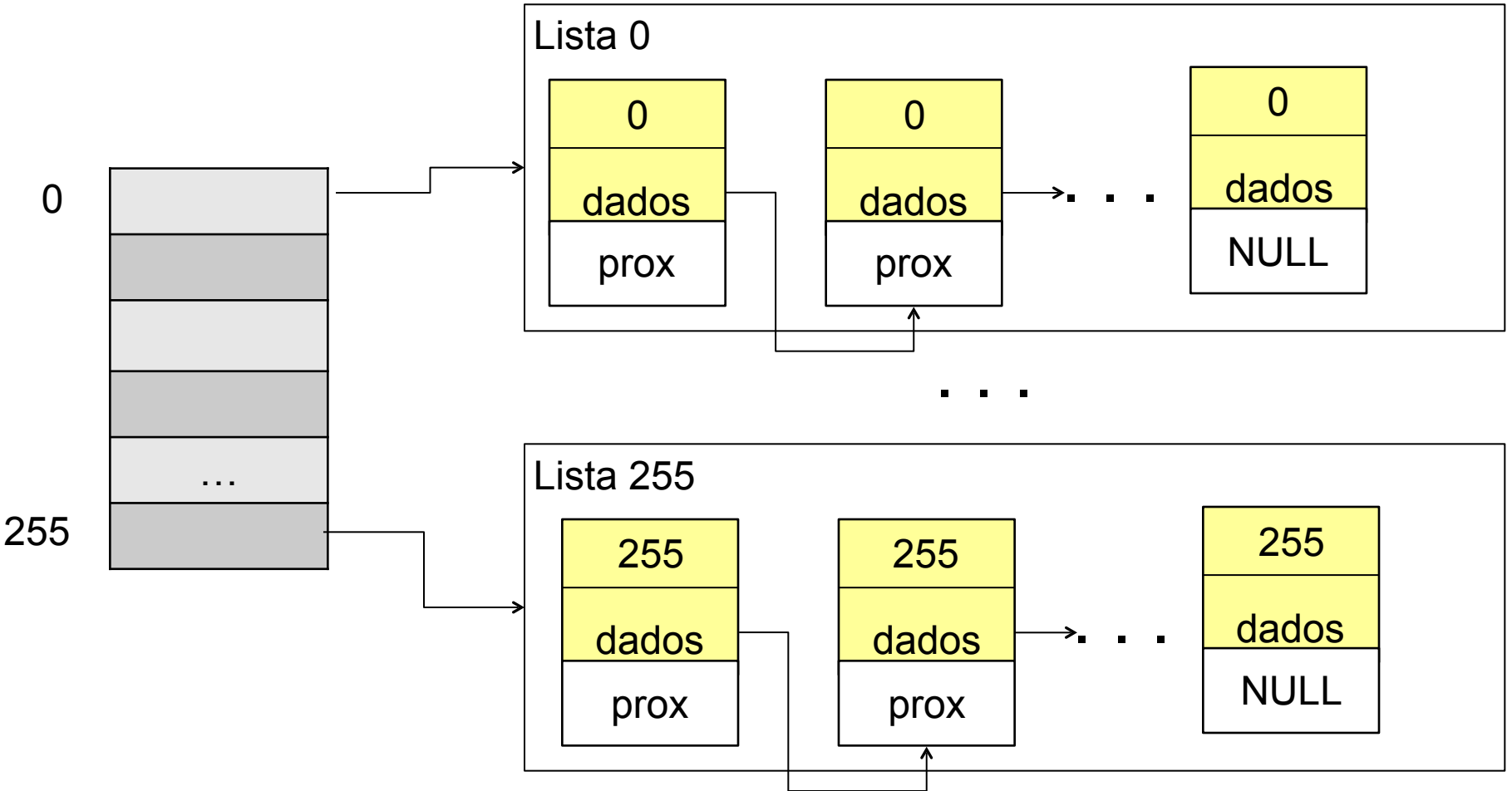
Exemplo: Ordenação

- **Solução:** Criar um vetor com 256 posições contendo ponteiros para listas com alocação dinâmica.



Exemplo: Ordenação

- **Solução:** Criar um vetor com 256 posições contendo ponteiros para listas com alocação dinâmica.

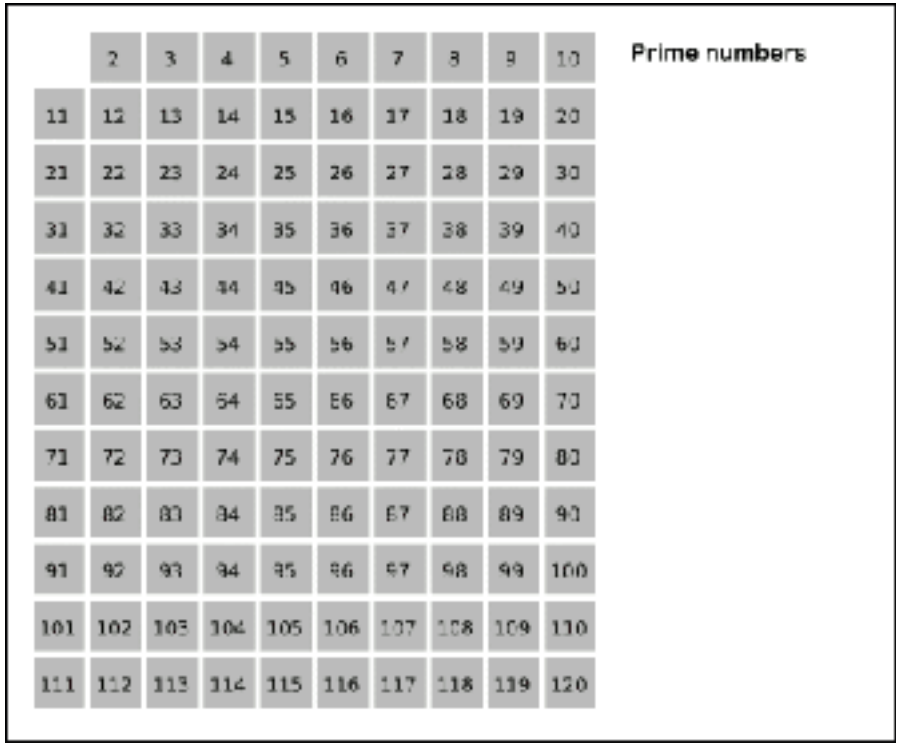


Exemplo: Ordenação

- **Ordenação:**
  - Percorrer lista original
    - Utilizar a chave de cada elemento para indexar o vetor
    - Insere elemento como último elemento da lista correspondente
  - Cria uma nova lista com alocação dinâmica
  - Percorrer cada elemento do vetor em ordem sequencial
    - Percorre cada item da lista correspondente
      - Insere item na nova lista

Exemplo: Crivo de Eratóstenes

- Crie uma lista com números de 2 até n.
- Encontre o primeiro número da lista.
- Remova da lista todos os múltiplos do número primo encontrado.
- O próximo número da lista é primo.
- Repita o procedimento.
- Ao final, a lista contém somente números primos.



Exercícios

- Implemente uma função que, dada uma lista encadeada e uma determinada chave C, remove o elemento com essa chave

- Implemente uma função que remova todos os elementos de valor par de uma lista encadeada