DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Listas Lineares

Professores: Luiz Chaimowicz e Raquel Prates

Agenda

- 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.

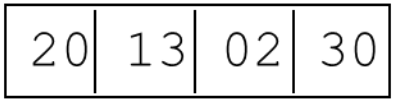
Representação de Lista na Memória

- As duas representações mais utilizadas são as implementações:
  - Alocação sequencial
  - Alocação encadeada

TAD Listas Lineares

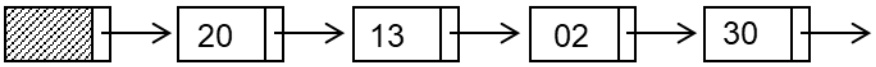
- TAD: Agrupa a ***estrutura de dados*** juntamente com as ***operações*** que podem ser feitas sobre esses dados.

Implementação por Vetores:




```
Void Insere(int x, Lista *L) {
   for(i=0;...) {...} // desloca
   L[0] = x;
}
```

Implementação por Listas Encadeadas




```
Void Insere(int x, Lista *L) {
    p = CriaNovaCelula(x);
    L->primeiro = p;
    ...
}
```

Programa usuário do TAD:

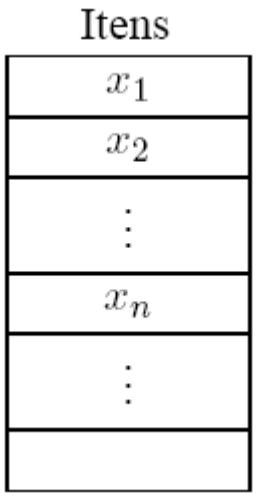
```
int main() {
   Lista L;
   int x;

   x = 20;
   FazListaVazia(&L);
   Insere(x,&L);
   ...
}
```

Operações sobre Listas

- **Exemplo das principais operações:**
  1. CriarListaVazia(Lista, Tam). Cria a Lista de tamanho Tam e retorna-a como uma ListaVazia
  2. FazListaVazia(Lista). Faz a lista ficar vazia.
  3. Inserir(Pos, Elemento, Lista). Insere o Elemento na Lista, na posição Pos.
  4. Retira(Pos, Lista, Elem). Retorna o Elemento que está na posição Pos da lista, retirando-o da lista e deslocando os itens a partir da posição p+1 para as posições anteriores.
  5. Vazia(Lista). Esta função retorna true se lista vazia; senão retorna false.
  6. Imprime(Lista). Imprime os itens da lista na ordem de ocorrência.

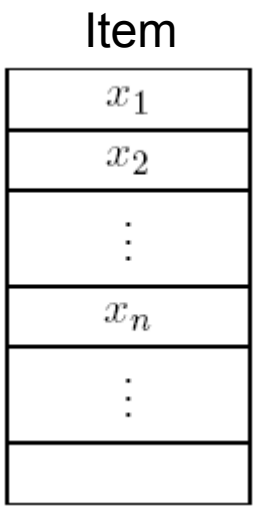
Alocação Sequencial

- 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



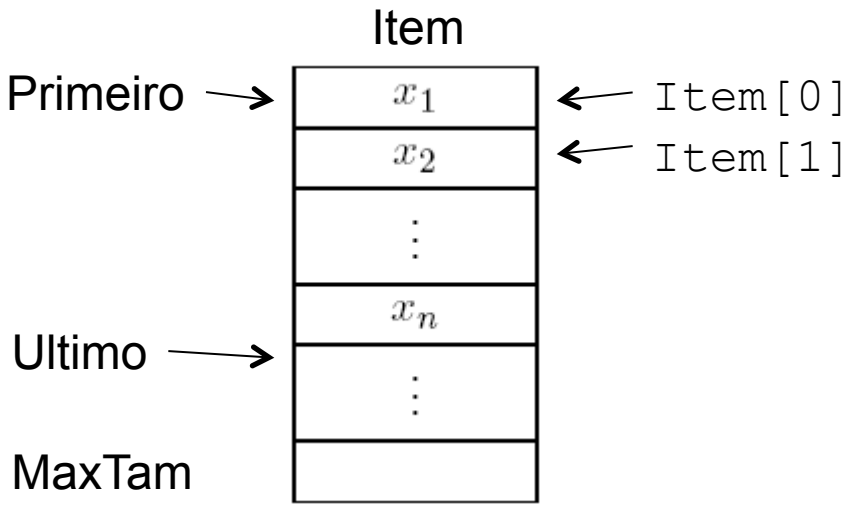
Alocação Sequencial (estrutura)

- Os itens armazenados em um vetor.
- Tem-se uma constante que define o tamanho máximo permitido para a lista.
- Pode ser interessante ter um campo que "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 vetor, 0 ≤ i < Último.* `(Item[i])`



Exemplo de Estrutura de Dados para Lista de Alocação Sequencial

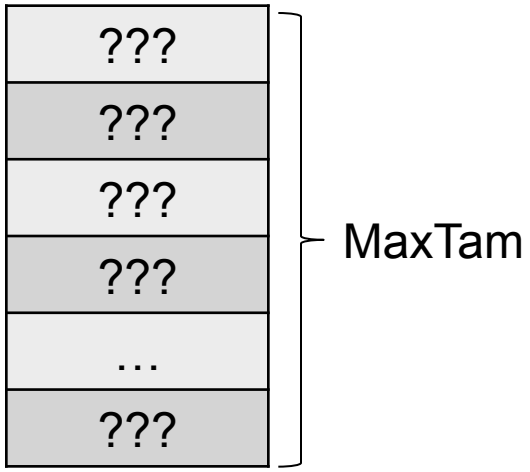
```
#define InicioArranjo   0
#define MaxTam          1000

typedef int TipoChave;

typedef int Apontador;

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

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Primeiro, Ultimo;
} TipoLista;
```



Alocação Sequencial (operações)

```
CriarListaVazia(Lista, Tamanho)
/* Cria a Lista de tamanho Tam e
   retorna-a como uma ListaVazia.*/
```

- Aloca espaço para a lista
- Define-a como sendo uma lista vazia

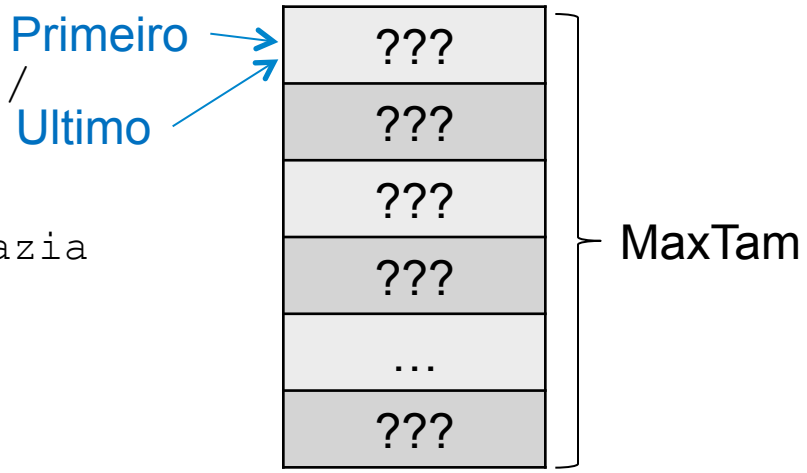



```
FazListaVazia(Lista)
/* Faz a lista ficar vazia*/
```

- Recebe uma lista existente
- Retorna-a como lista vazia (alguma indicação de que não tem elementos)

Alocação Sequencial (operações)

```
CriarListaVazia(Lista, Tamanho)
/* Cria a Lista de tamanho Tam e
   retorna-a como uma ListaVazia.*/
```



- Aloca espaço para a lista
- Define-a como sendo uma lista vazia

```
FazListaVazia(Lista)
/* Faz a lista ficar vazia*/
```

- Recebe uma lista existente
- Retorna-a como lista vazia (alguma indicação de que não tem elementos)

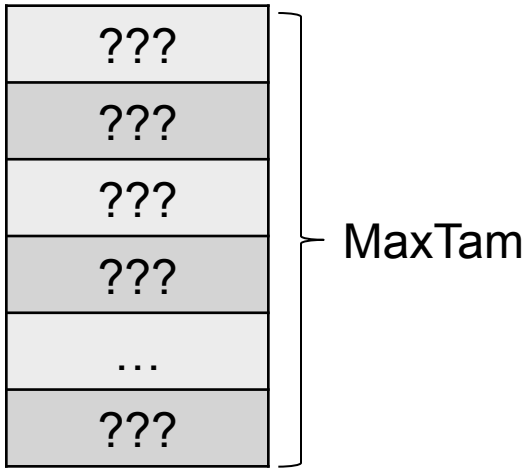
Exemplo:

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

Alocação Sequencial (operações)

```
Boolean Vazia(Lista)
/* Verifica se a Lista é vazia */
```

- De acordo com a definição do que consiste uma lista vazia na estrutura de dados, verifica se a condição é atendida
- Exemplos
  - Lista.tamanho == 0
  - Lista.Ultimo == Lista.Primeiro

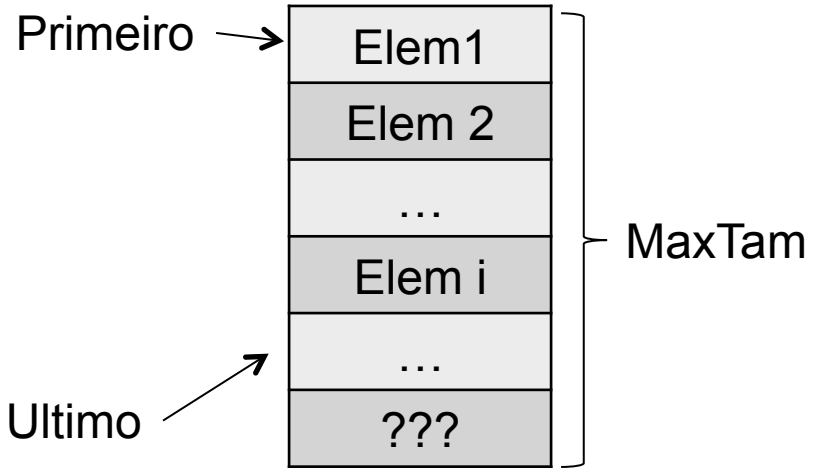


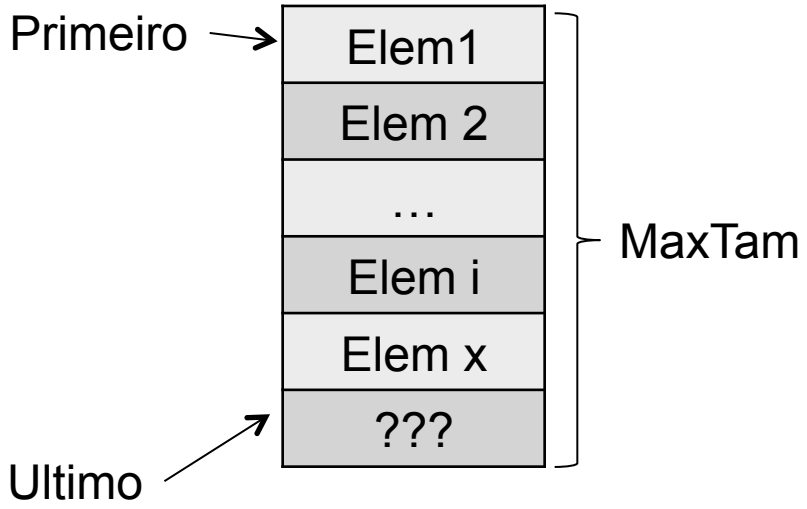
Exemplo em C (TAD do Livro)

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

Alocação Sequencial (operações)

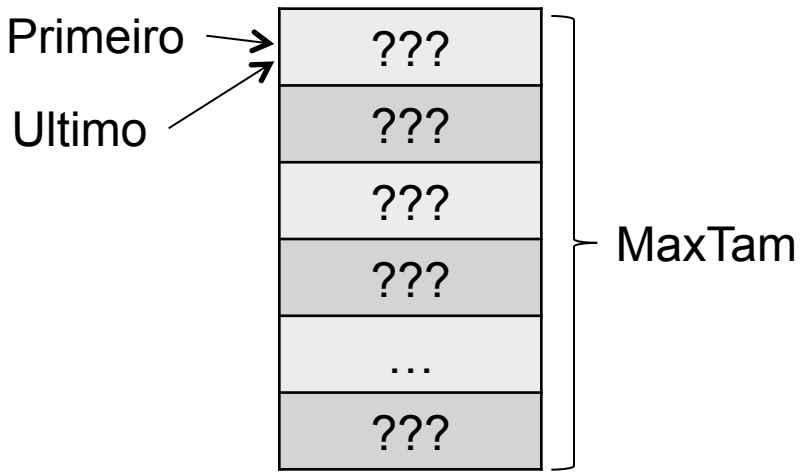
```
InsereFinal(TipoItem Item, TipoLista Lista)
{
      se(Lista.Ultimo >= MaxTam)
           msg("Lista esta cheia");
       senão {
           Lista.Item[Lista.Ultimo] = Item;
           Lista.Ultimo = Lista.Ultimo + 1;
       }
 }
```





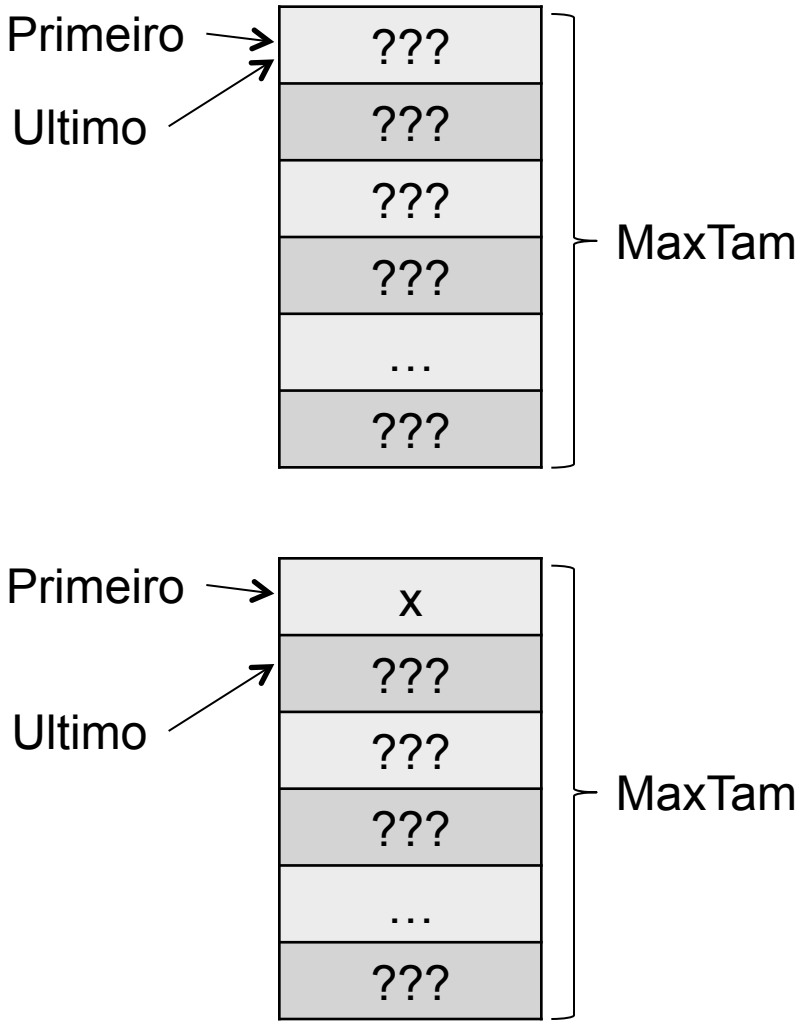
Alocação Sequencial (operações)

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



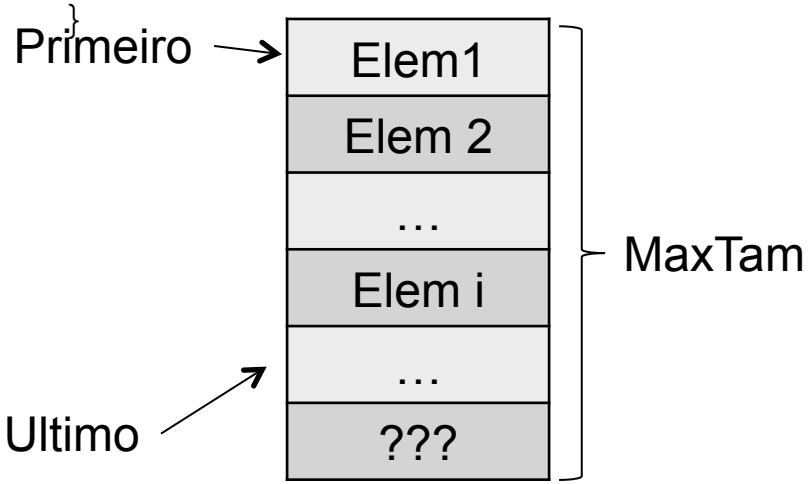
Alocação Sequencial (operações)

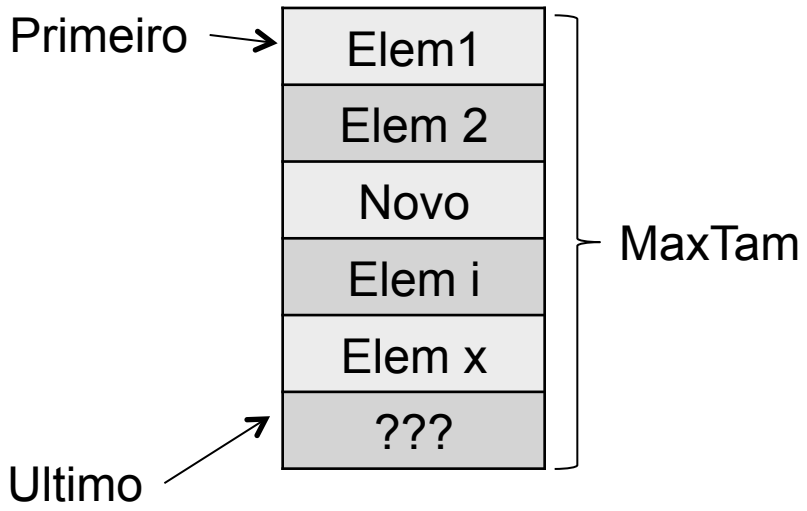
```
void Insere(TipoItem x, TipoLista *Lista)
{
  if (Lista->Ultimo >= MaxTam)
     printf("Lista esta cheia\n");
  else
  {
     Lista->Item[Lista->Ultimo] = x;
     Lista->Ultimo++;
  }
}  /* Insere */
```



Alocação Sequencial (operações)

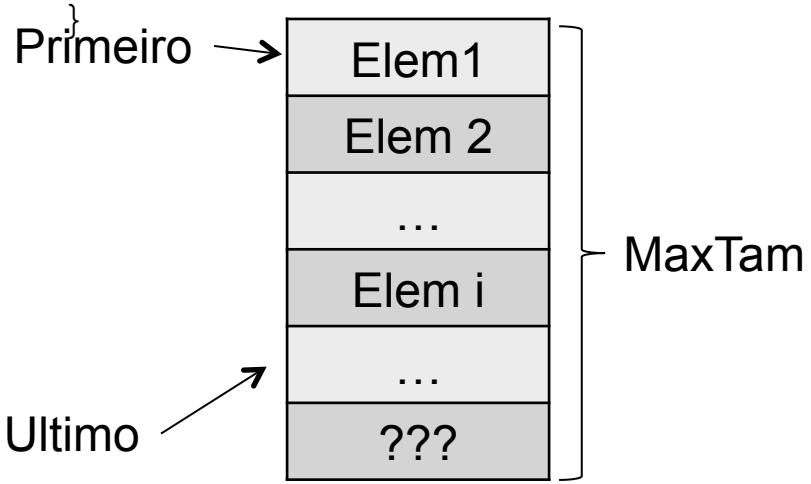
```
    InserePos(TipoItem Item, TipoLista Lista, inteiro Pos)
    {
        se (Lista.Ultimo >= MaxTam)
      msg("Lista esta cheia");
   senão {
      para i = Ultimo+1 até Pos
          Lista.Item[i+1] = Lista.Item[i];
    Lista.Item[Pos] = Item;
    Lista.Ultimo = Lista.Ultimo + 1;
   }
}
```

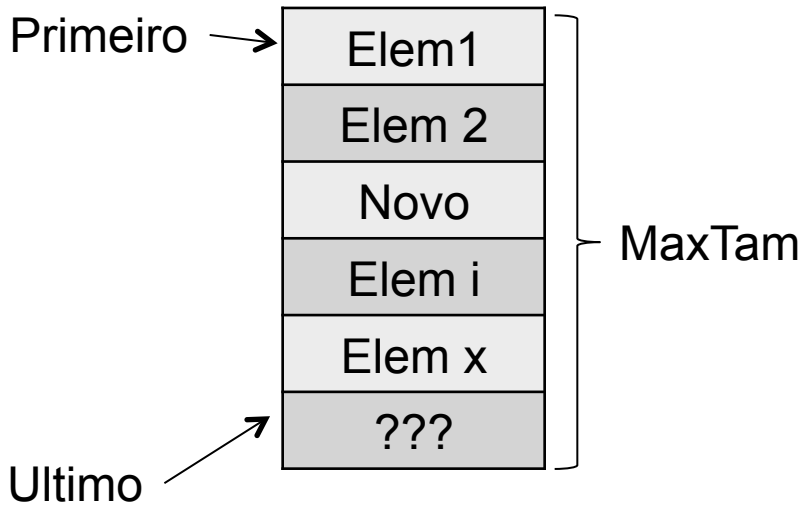




Alocação Sequencial (operações)

```
InserePos(inteiro Pos, TipoItem Item, TipoLista Lista)
 {
     se (Lista.Ultimo >= MaxTam)
  msg("Lista esta cheia");
senão {
   para i = Ultimo até (Pos+1)
       Lista.Item[i] = Lista.Item[i-1];
 Lista.Item[Pos] = Item;
 Lista.Ultimo = Lista.Ultimo + 1;
}
}
```





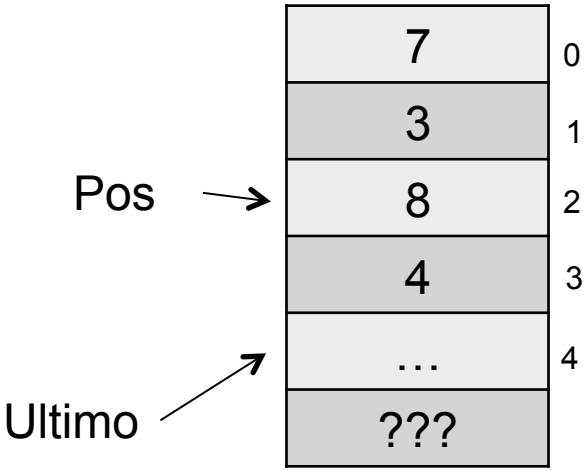
Alocação Sequencial (operações)

```
InserePos(inteiro Pos, TipoItem Item, TipoLista Lista)
{
    se(Lista.Ultimo >= MaxTam)
        msg("Lista esta cheia");
     senão {
        para i = Ultimo até (Pos+1)
        Lista.Item[i] = Lista.Item[i-1];
        Lista.Item[Pos] = Item;
        Lista.Ultimo = Lista.Ultimo + 1;
      }
  }
```

Alocação Sequencial (operações)

```
InserePos(inteiro Pos, TipoItem Item, TipoLista Lista)
{
    se(Lista.Ultimo >= MaxTam)
        msg("Lista esta cheia");
     senão {
        para i = Ultimo até (Pos+1)
        Lista.Item[i] = Lista.Item[i-1];
        Lista.Item[Pos] = Item;
        Lista.Ultimo = Lista.Ultimo + 1;
      }
}
```

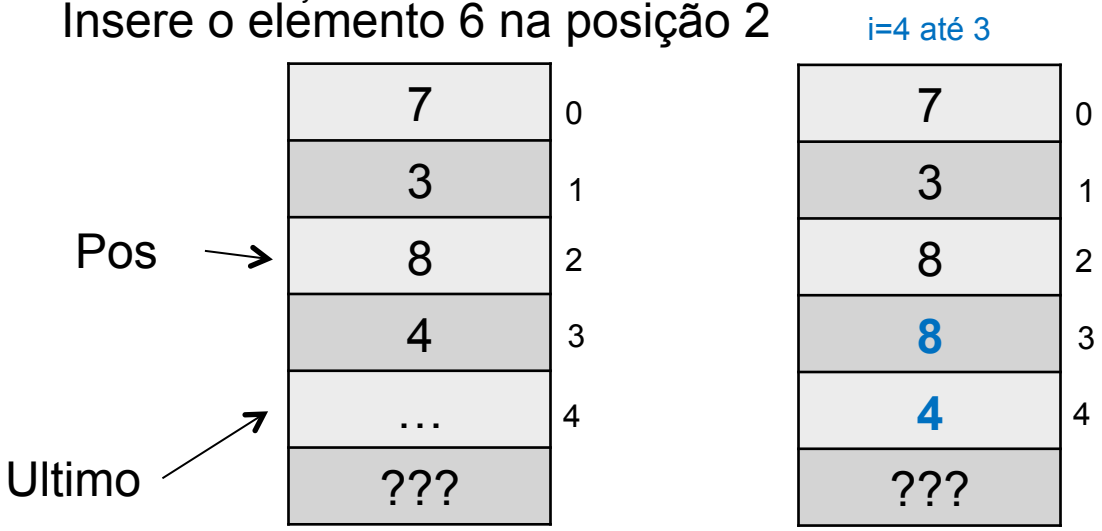
Insere o elemento 6 na posição 2



Alocação Sequencial (operações)

```
InserePos(inteiro Pos, TipoItem Item, TipoLista Lista)
{
    se (Lista.Ultimo >= MaxTam)
       msg("Lista esta cheia");
    senão {
       para i = Ultimo até (Pos+1)
         Lista.Item[i] = Lista.Item[i-1];
         Lista.Item[Pos] = Item;
         Lista.Ultimo = Lista.Ultimo + 1;
    }
}
```

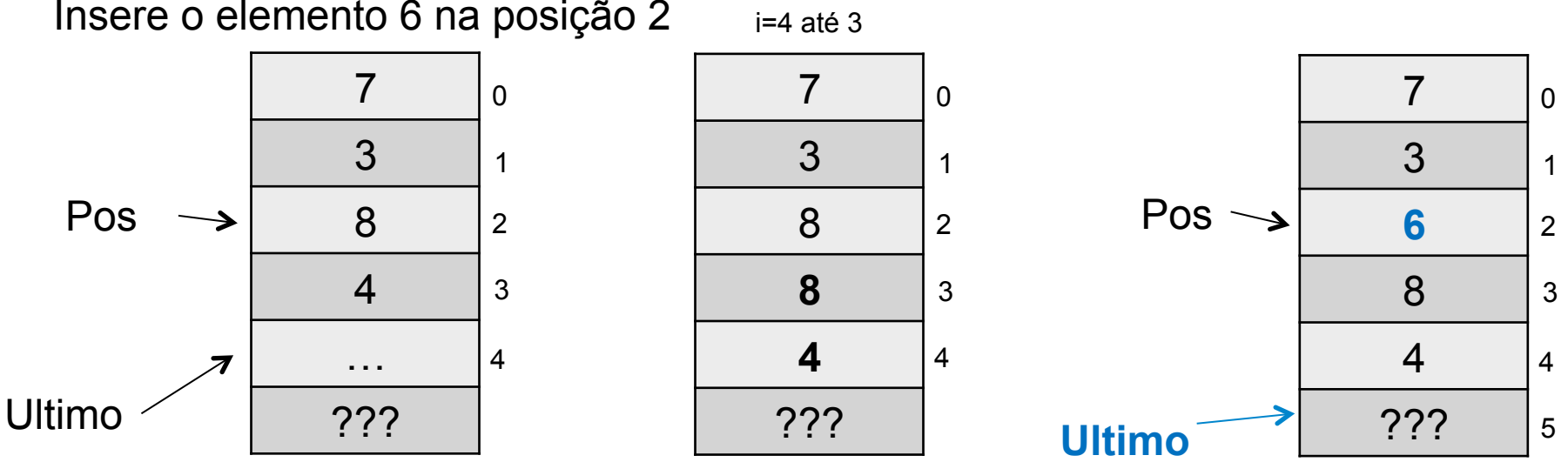
Insere o elemento 6 na posição 2



Alocação Sequencial (operações)

```
InserePos(inteiro Pos, TipoItem Item, TipoLista Lista)
{
    se (Lista.Ultimo >= MaxTam)
        msg("Lista esta cheia");
    senão {
        para i = Ultimo até (Pos+1)
            Lista.Item[i] = Lista.Item[i-1];
            Lista.Item[Pos] = Item;
            Lista.Ultimo = Lista.Ultimo + 1;
    }
}
```

Insere o elemento 6 na posição 2



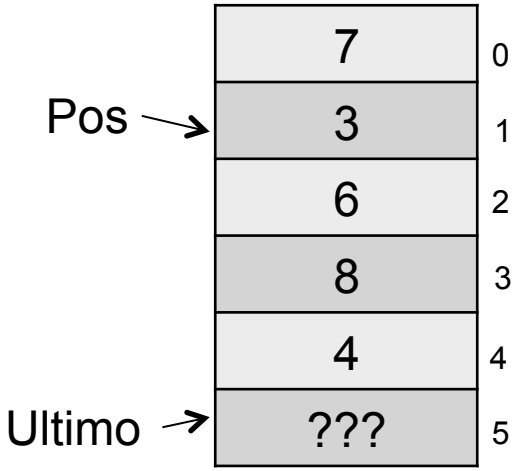
Alocação Sequencial (operações)

```
Retira(inteiro Pos, TipoItem Item, TipoLista Lista){
    se (Pos >= MaxTam) ou (Pos < 0) ou Vazia(Lista)
       msg("Erro: Posição não existe");
     senão {
       Item = Lista.Item[Pos];
       Lista.Ultimo = Lista.Ultimo - 1;

       para i = Pos até Ultimo
         Lista.Item[i] = Lista.Item[i+1];
     }
}
```

Alocação Sequencial (operações)

```
Retira(inteiro Pos, TipoItem Item, TipoLista Lista){
    se (Pos >= MaxTam) ou (Pos < 0) ou Vazia(Lista)
       msg("Erro: Posição não existe");
     senão {
       Item = Lista.Item[Pos];
       Lista.Ultimo = Lista.Ultimo - 1;

       para i = Pos até Ultimo
         Lista.Item[i] = Lista.Item[i+1];
     }
```

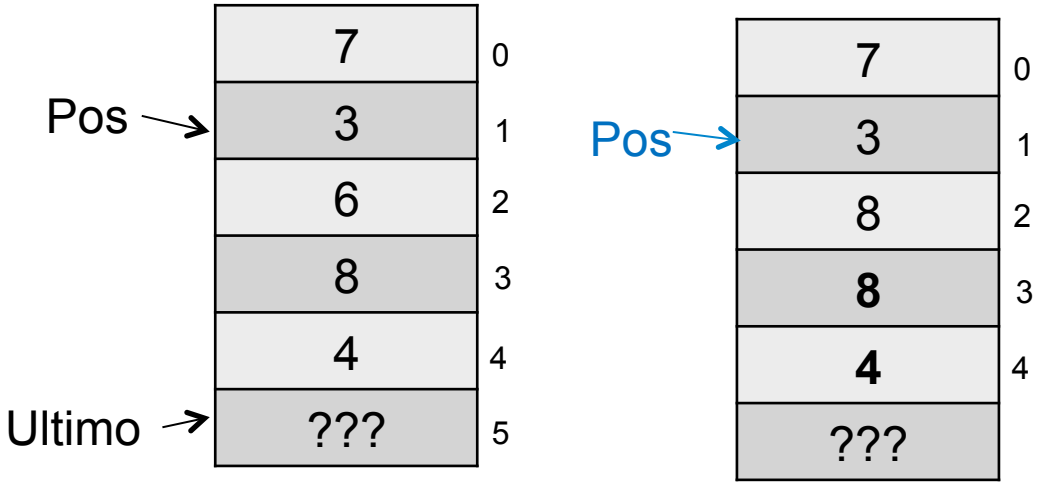
Retira o elemento da posição 1



Alocação Sequencial (operações)

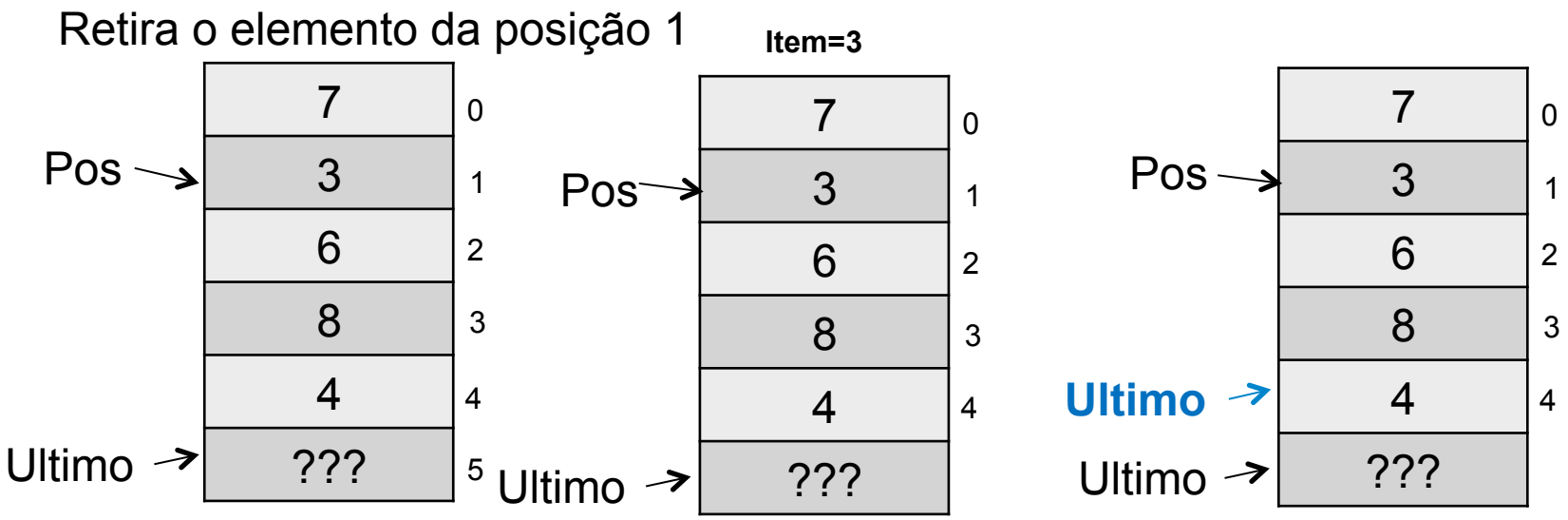
```
Retira(inteiro Pos, TipoItem Item, TipoLista Lista){
    se (Pos >= MaxTam) ou (Pos < 0) ou Vazia(Lista)
       msg("Erro: Posição não existe");
    senão {
       Item = Lista.Item[Pos];
       Lista.Ultimo = Lista.Ultimo - 1;

       para i = Pos até Ultimo
         Lista.Item[i] = Lista.Item[i+1];
     }
}
```

Retira o elemento da posição 1



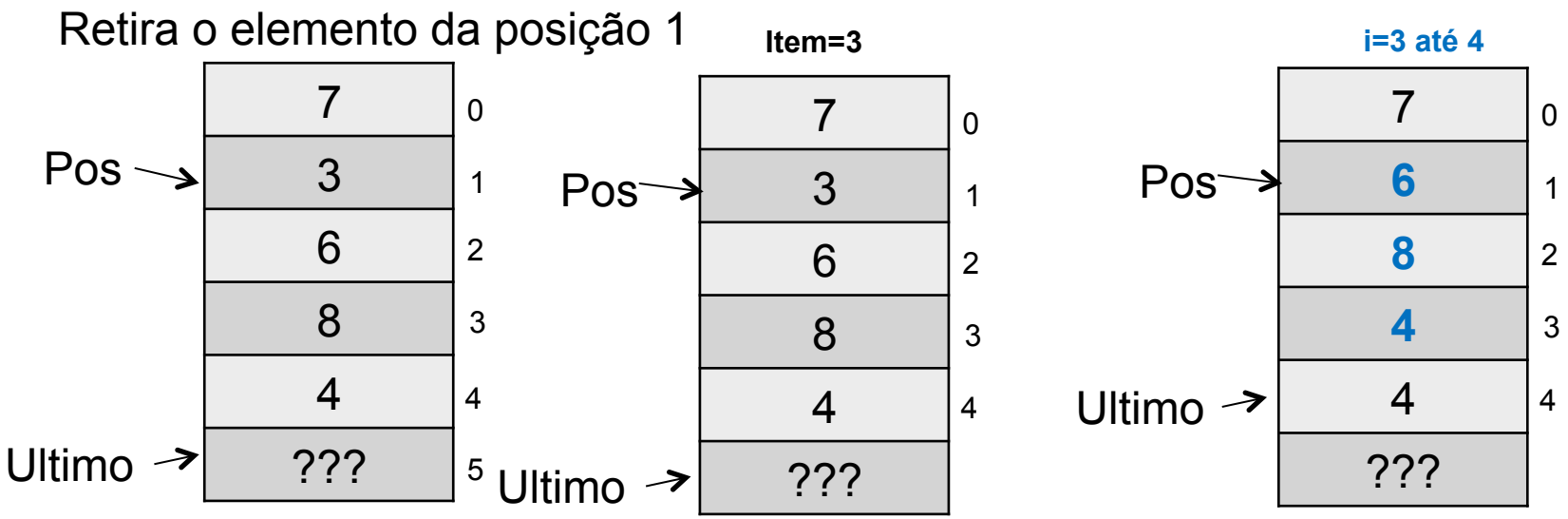
Alocação Sequencial (operações)

```
Retira(inteiro Pos, TipoItem Item, TipoLista Lista){
    se (Pos >= MaxTam) ou (Pos < 0) ou Vazia(Lista)
       msg("Erro: Posição não existe");
    senão {
       Item = Lista.Item[Pos];
       Lista.Ultimo = Lista.Ultimo - 1;

       para i = Pos até Ultimo
          Lista.Item[i] = Lista.Item[i+1];
    }
}
```



Alocação Sequencial (operações)

```
Retira(inteiro Pos, TipoItem Item, TipoLista Lista){
    se (Pos >= MaxTam) ou (Pos < 0) ou Vazia(Lista)
       msg("Erro: Posição não existe");
    senão {
       Item = Lista.Item[Pos];
       Lista.Ultimo = Lista.Ultimo - 1;

       para i = Pos até Ultimo
          Lista.Item[i] = Lista.Item[i+1];
    }
}
```



Alocação Sequencial

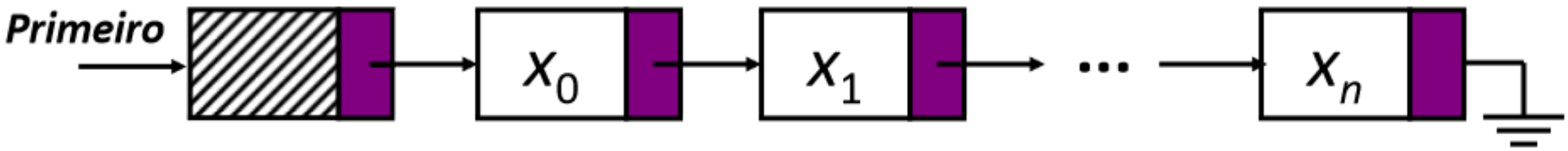
- Vantagens:
  - economia de memória, pois cada elemento da lista armazena apenas os dados.
  - A estrutura da lista é definida implicitamente
  - Acesso a um item qualquer é **CONSTANTE**

Alocação Sequencial

- Desvantagens:
  - custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens
    - no pior caso **É IGUAL AO TAMANHO DO VETOR: CUSTO LINEAR!**
  - O tamanho máximo da lista é **fixo** e definido em tempo de compilação!
    - Pouco útil para aplicações em que **não existe previsão sobre o crescimento** da lista.

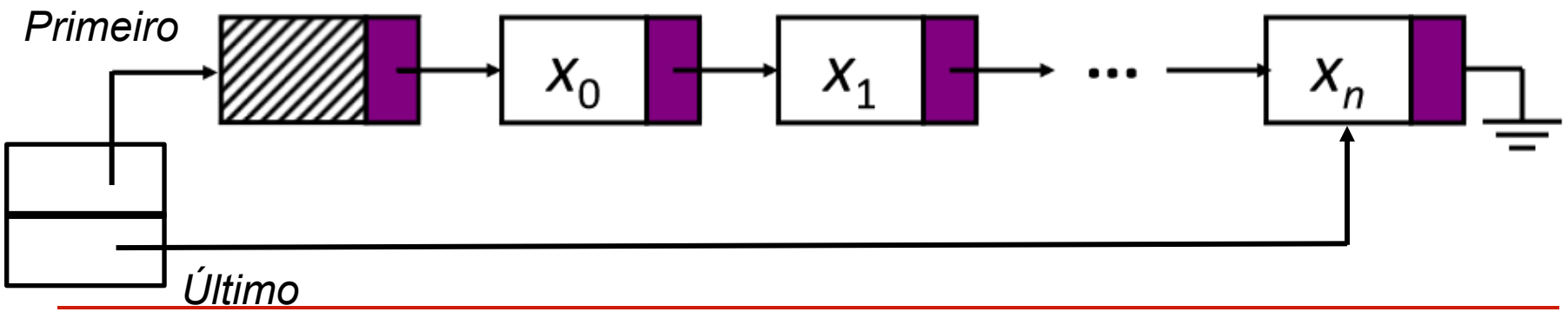
Alocação Encadeada

- Permite utilizar posições não contíguas de memória
- Permite o crescimento e redução da lista
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista



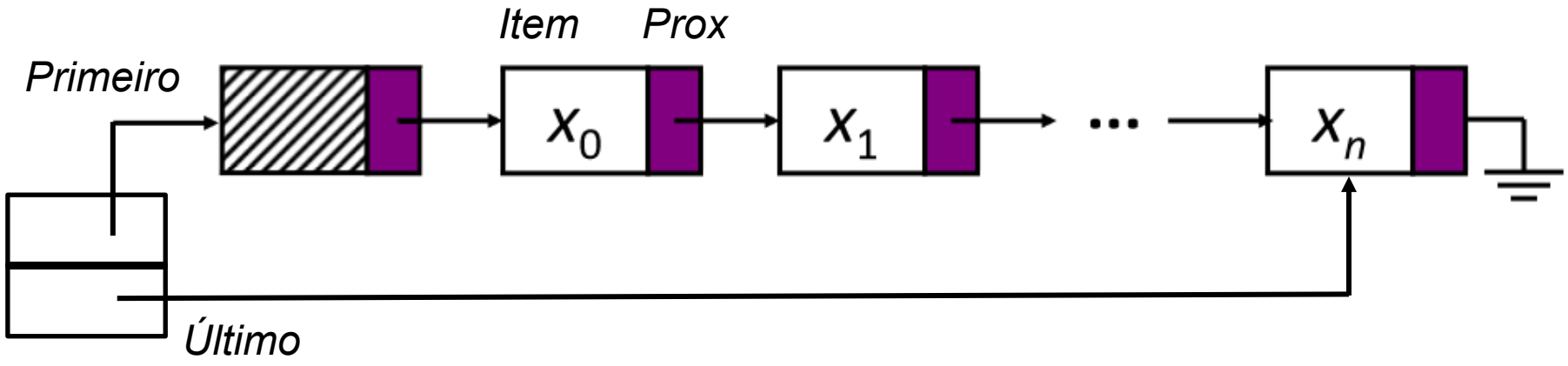
Alocação Encadeada

- A lista é constituída de células
- Cada célula contém um item da lista e um apontador para a célula seguinte
- Uma variável do *TipoLista* é um registro com um apontador para a célula cabeça e um apontador para a última célula da lista.



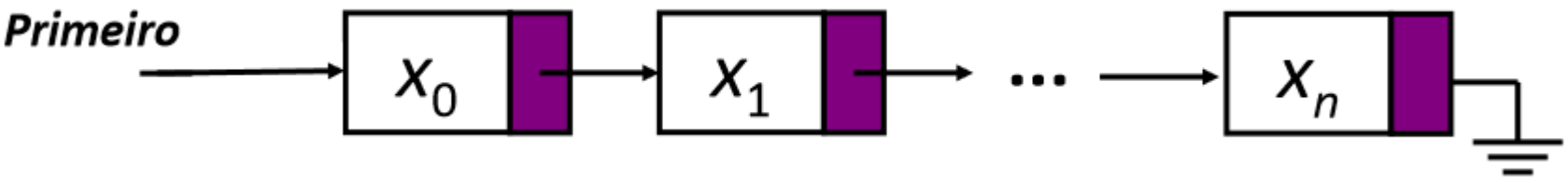
Alocação Encadeada

- Cada elemento (célula) da lista contém
  - Campo **Item**: registro que guarda os dados
  - Campo **Prox**: apontador para a próxima célula
- O campo prox da última célula aponta para NULL
- Novas células são criadas sob demanda em posições não contíguas de memória (heap) e encadeadas na lista



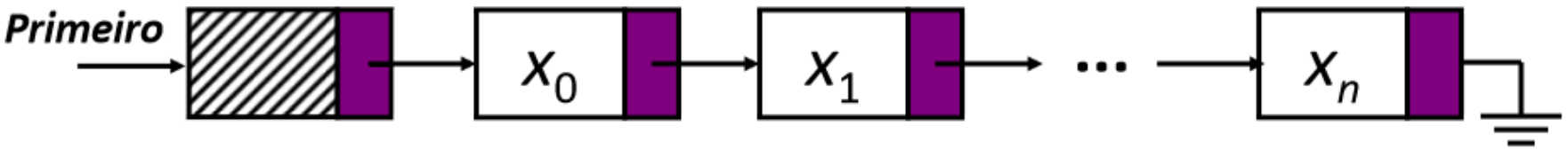
Alocação Encadeada

- Uma lista encadeada pode ser organizada de duas maneiras diferentes:
  - Lista *sem* célula cabeça: A primeira célula contém conteúdo.



Alocação Encadeada

- Uma lista encadeada pode ser organizada de duas maneiras diferentes:
  - Lista *com* célula cabeça: O conteúdo da primeira célula é irrelevante. Essa célula apenas marcar o início da lista.

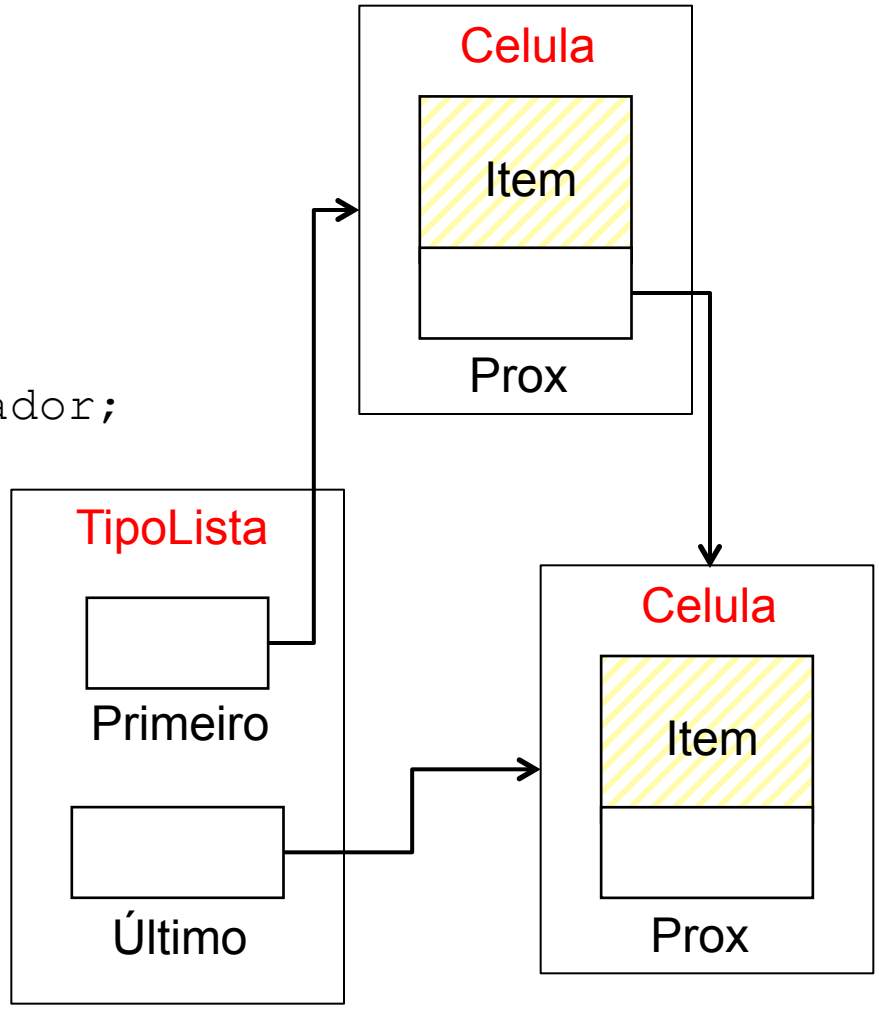


Alocação Encadeada

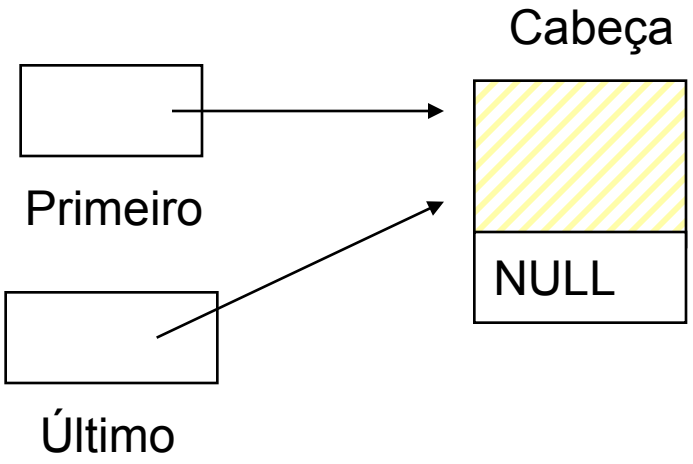
- Lista *sem* cabeça
  - Operações de Inserção e Remoção exigem que seja verificado se ponteiro para a primeiro célula é igual a NULL
- Lista *com* cabeça
  - Evita os testes com a primeira célula e assim facilita o desenvolvimento e melhora o desempenho

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;
```

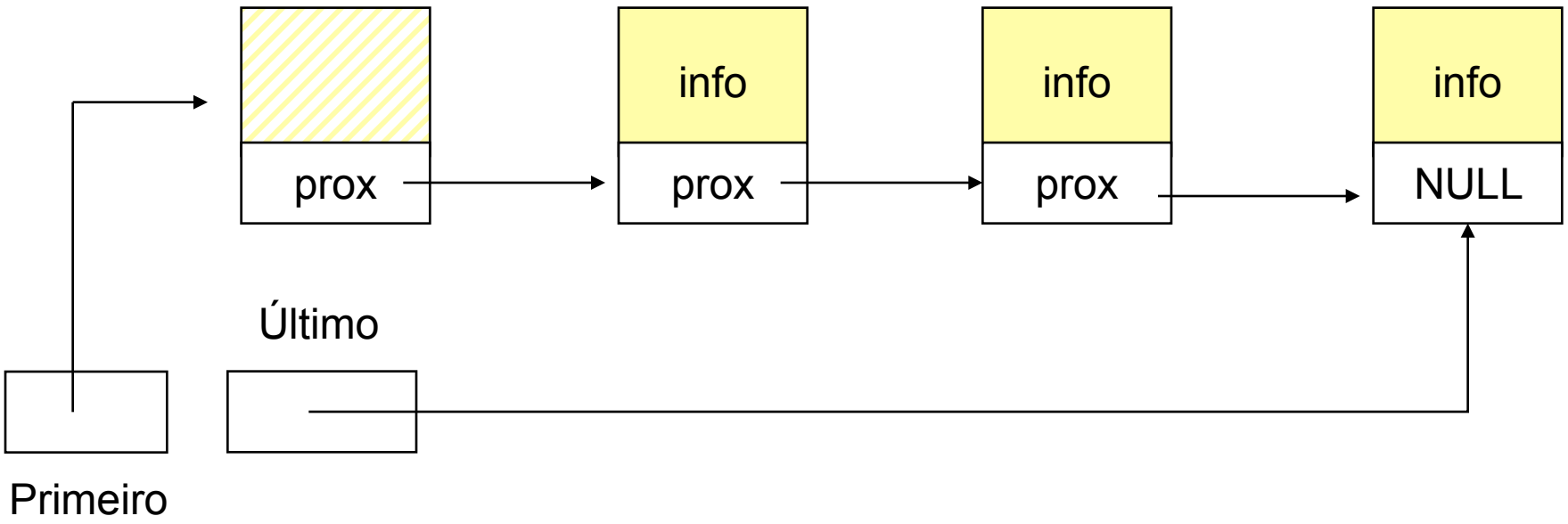


Cria Lista Vazia



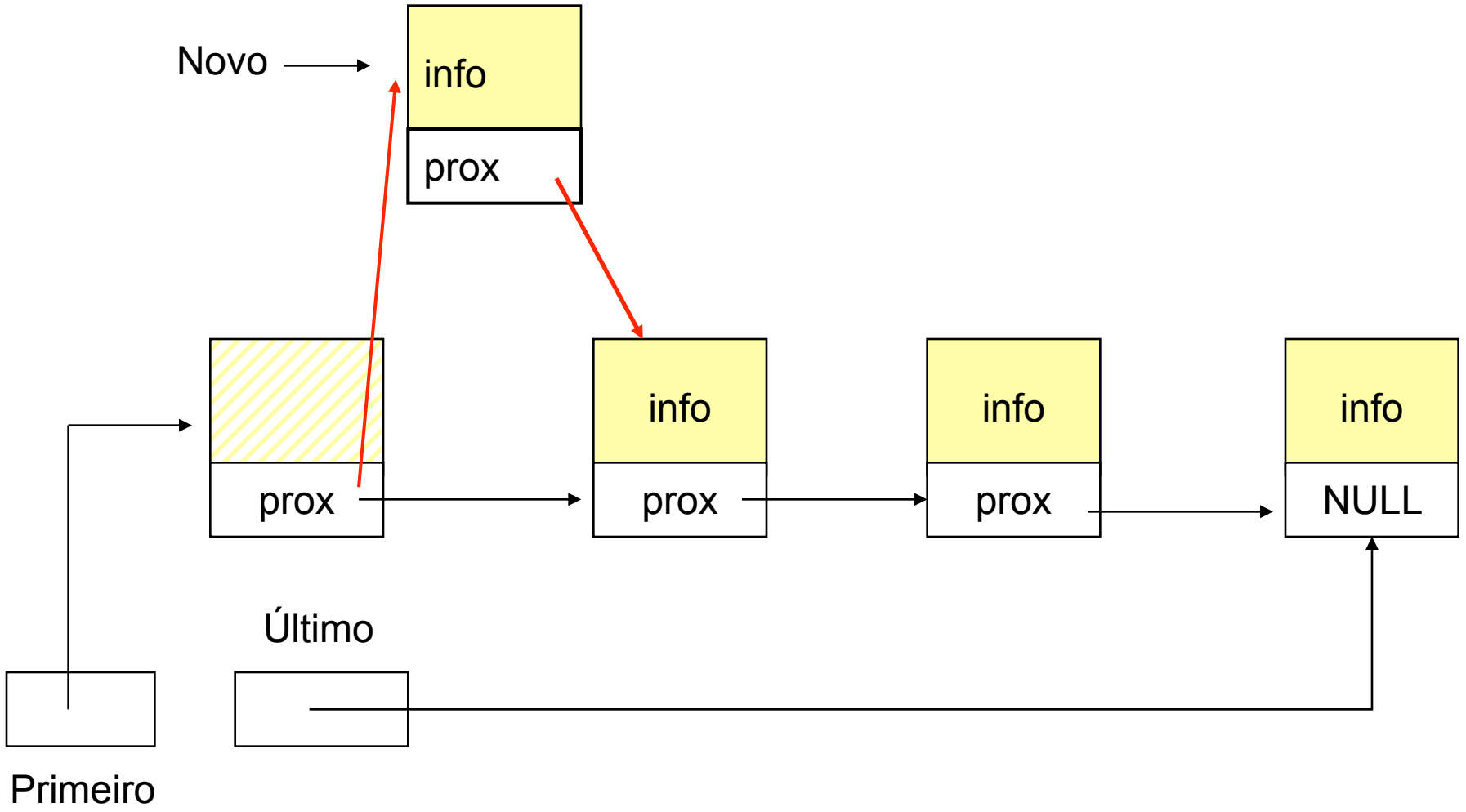

```
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = (Apontador) malloc(sizeof(Celula));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}

int Vazia(const TipoLista *Lista)
{
  return (Lista->Primeiro == Lista->Ultimo);
}
```

Inserção de Elementos na Lista

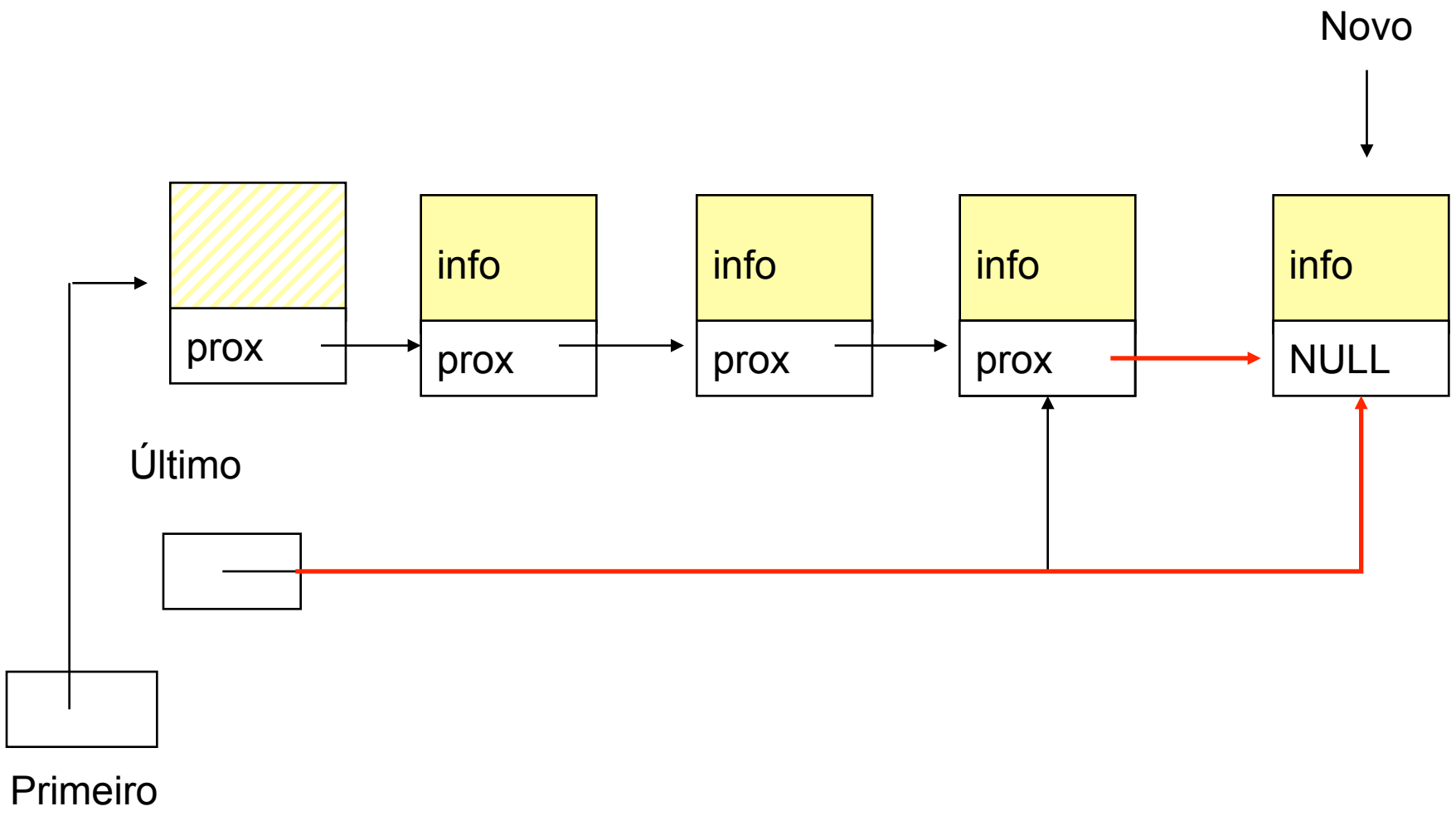


- 3 opções de posição onde pode inserir:
  - 1ª. posição
  - última posição
  - Após um elemento qualquer E

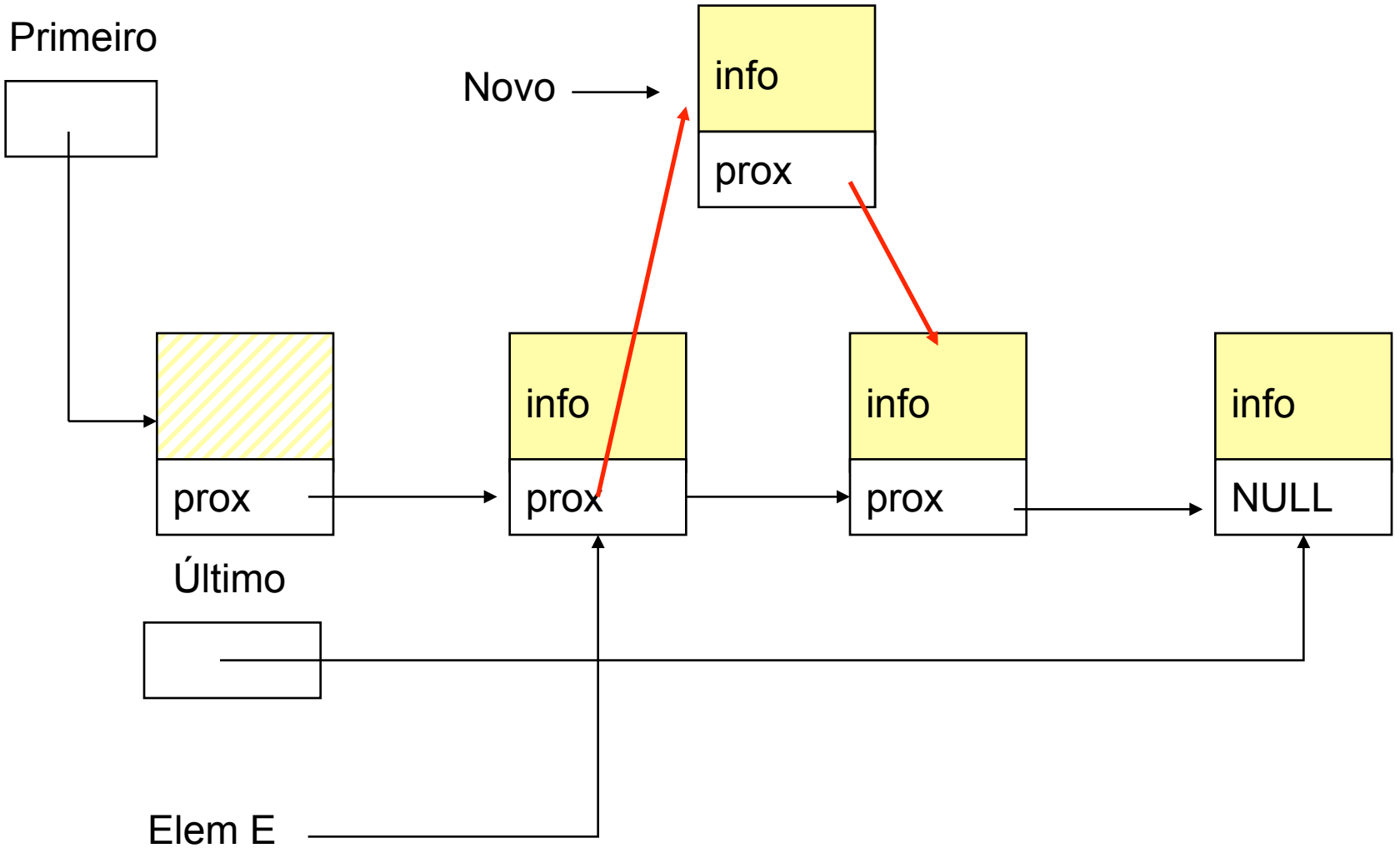
Inserção na Primeira Posição



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 ultimo
    - Necessário atualizar o apontador *último*
  - Após um elemento qualquer E: **p** aponta para E

Inserção na Após o Elemento E

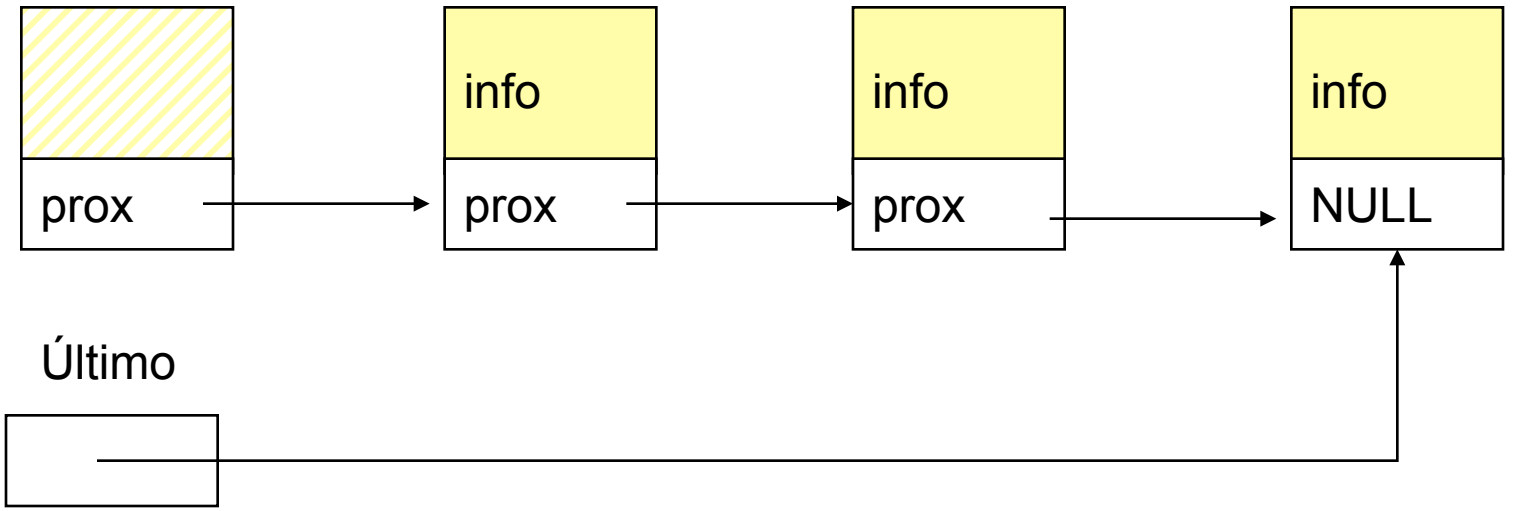
```
void Insere(TipoItem x, TipoLista *lista, Apontador E){
          Apontador novo;
   novo = (Apontador) malloc(sizeof(Celula));
          novo->Item = x;
   novo->prox = E->prox;
   E->prox = novo;
   if(E = Lista->Ultimo)
             Lista->Ultimo = novo
}
```

Inserção após o último

- Mantendo compatibilidade com a função proposta no TAD

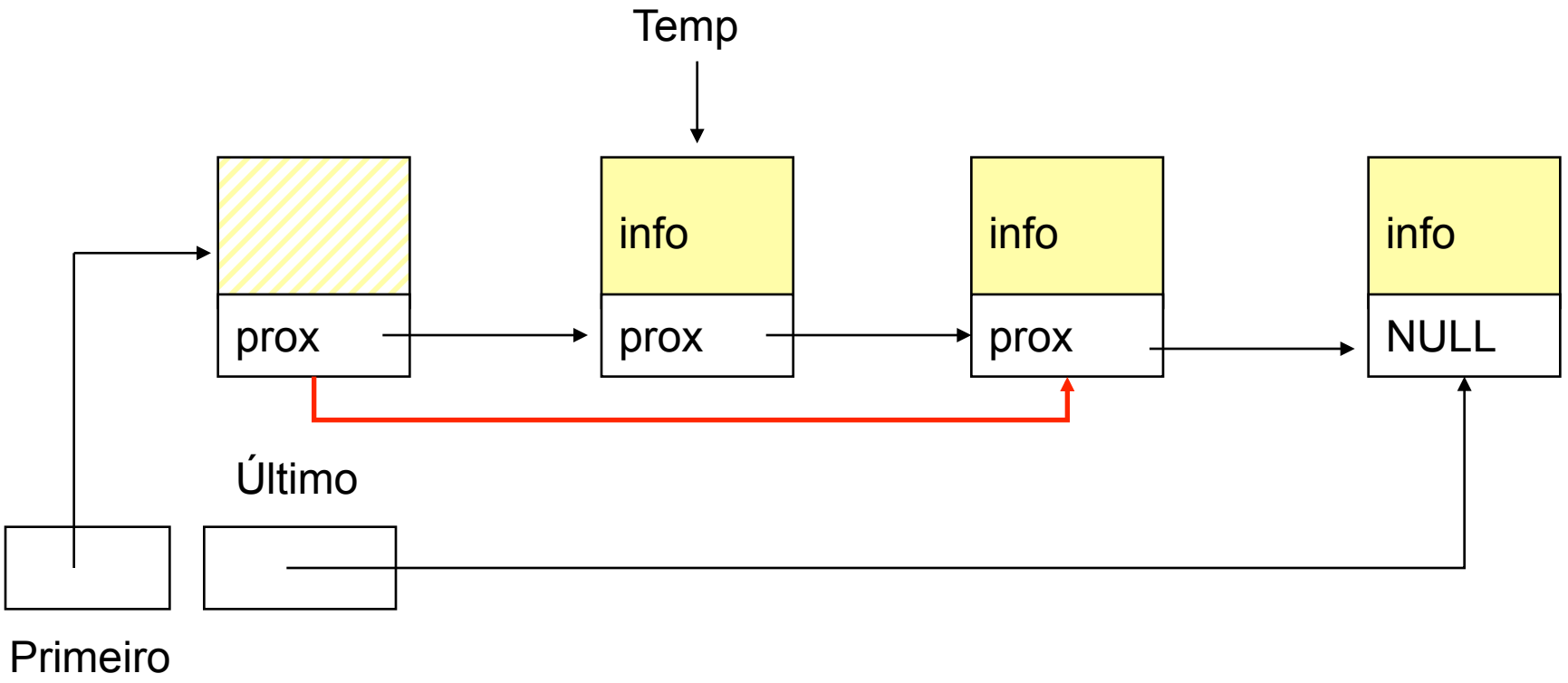
```
void Insere(TipoItem x, TipoLista *Lista)
/* Insere na ultima posição. */
{

   Lista->Ultimo->prox =(Apontador) malloc(sizeof(Celula));
   Lista->Ultimo = Lista->Ultimo->prox;
   Lista->Ultimo->Item = x;
   Lista->Ultimo->prox = NULL;
}
```

Retirada de Elementos na Lista

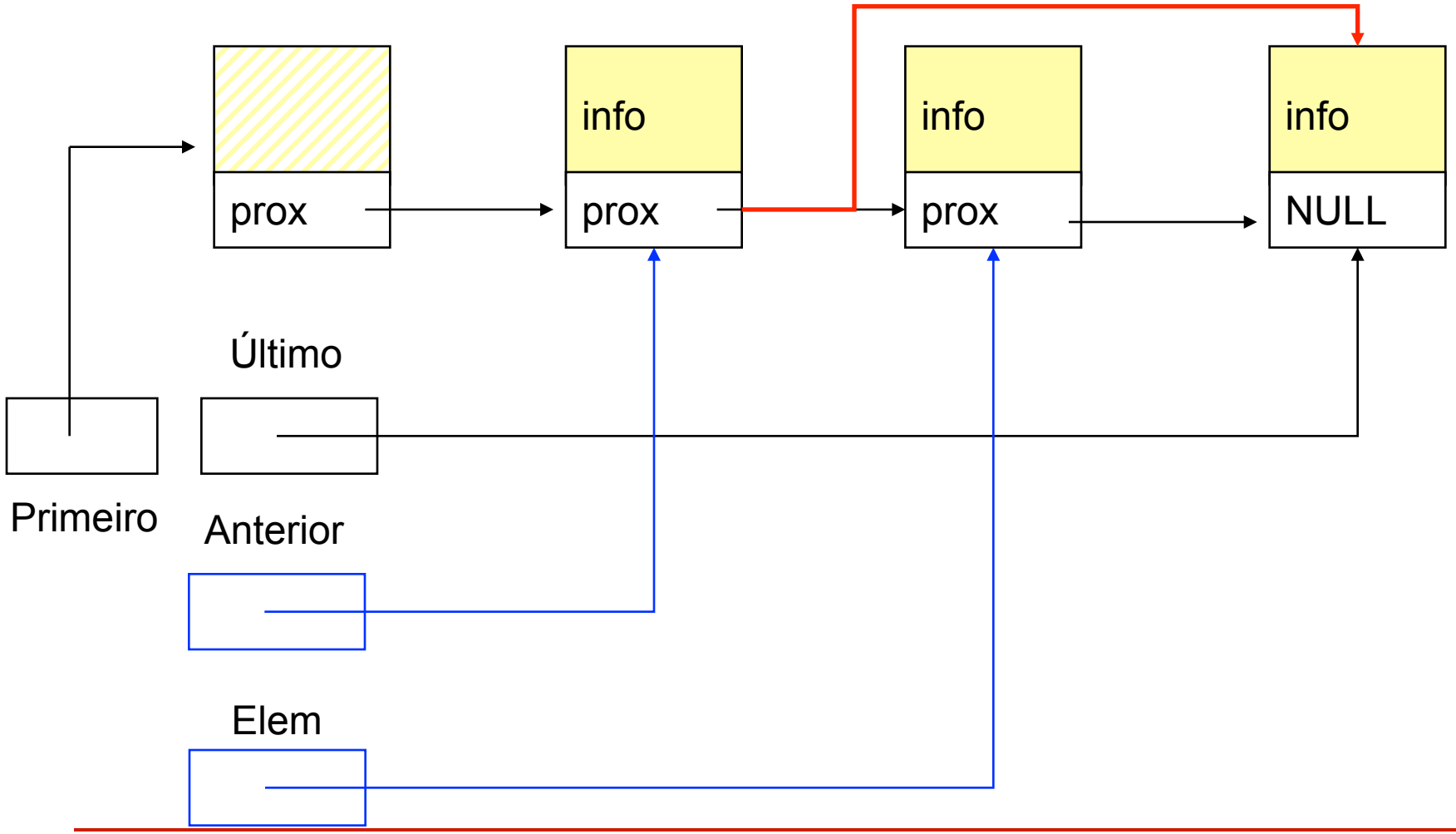


- 3 opções de posição de onde pode retirar:
  - 1ª. posição
  - última posição
  - Um elemento qualquer E

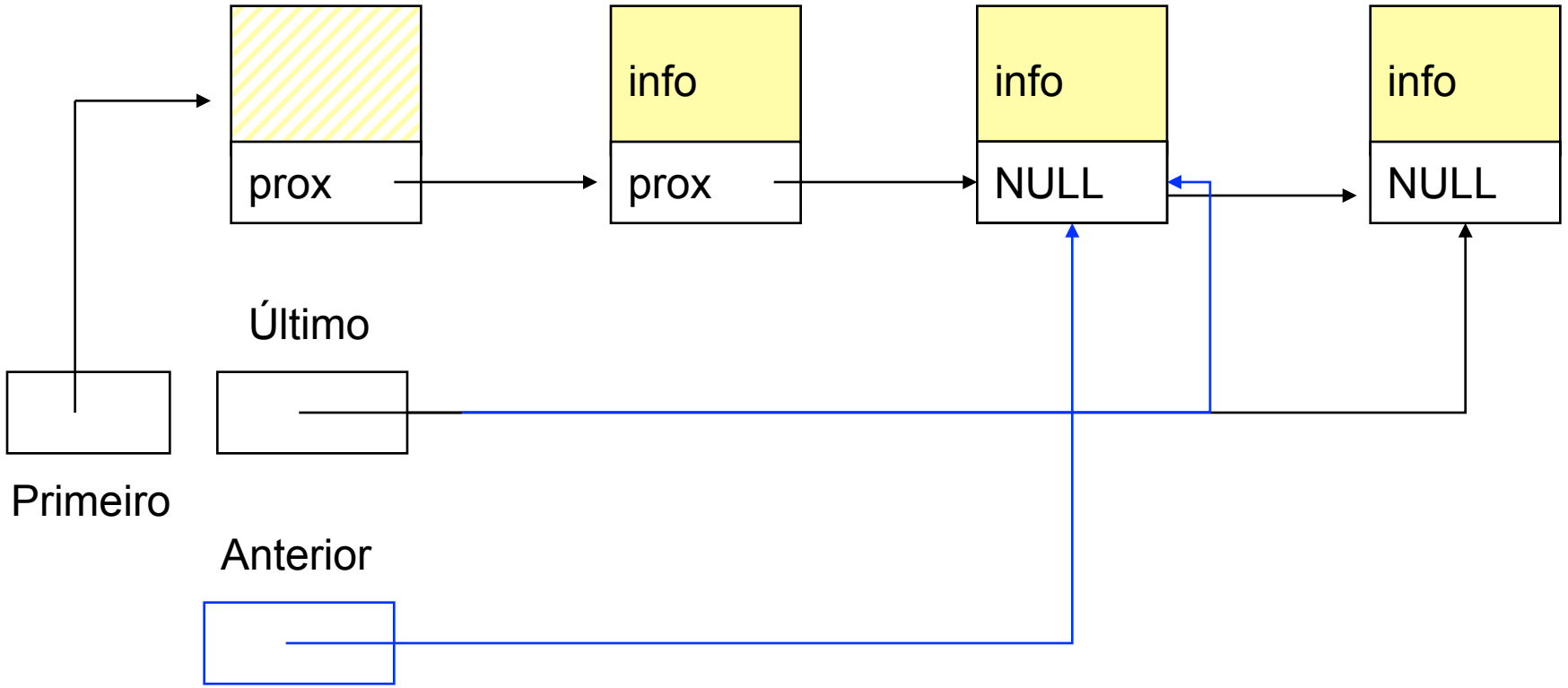
Retirada do Elemento na Primeira Posição da Lista



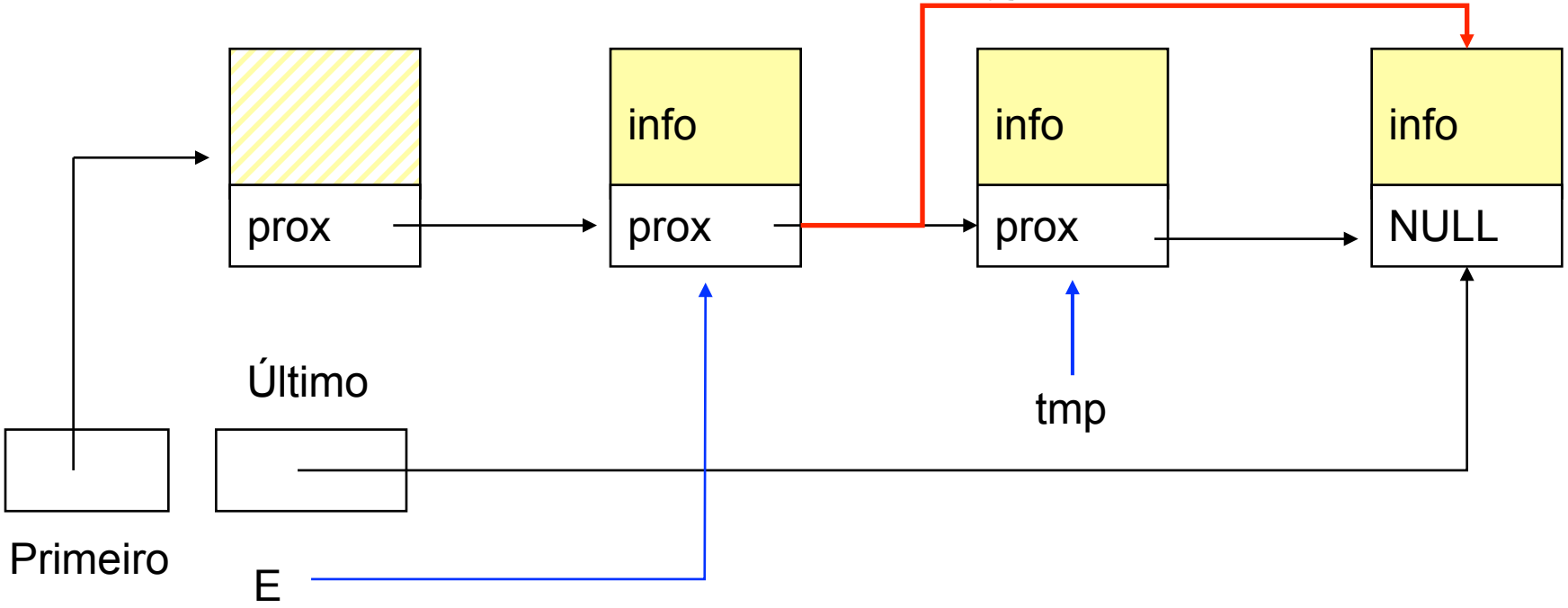
Retirada do Elemento E da Lista



Retirada do Último Elemento da Lista



Retirada do elemento após E da Lista




```
int RemoveProx(TipoLista *lista, Apontador E, TipoItem *item){
   Apontador tmp;
   tmp = E->prox;
   if (tmp != NULL) {
                E->prox = tmp->prox;
                *item = tmp->item;
      free(tmp);
      return 1;
   }
   else return 0;
}
```

Função Retira

```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item)
{ /*- Obs.: o item a ser retirado e o seguinte ao apontado por p - */

   Apontador q;

   if (Vazia(*Lista) || p == NULL || p->Prox == NULL) {
       printf(" Erro Lista vazia ou posição não existe\n");
       return;
   }

   q = p->Prox;
   *Item = q->Item;
   p->Prox = q->Prox;
   if (p->Prox == NULL)
       Lista->Ultimo = p;
   free(q);
}
```

Função Imprime

```
void Imprime(TipoLista Lista)
{
  Apontador Aux;
  Aux = Lista.Primeiro->Prox;
  while (Aux != NULL)
    { printf("%d\n", Aux->Item.Chave);
      Aux = Aux->Prox;
    }
}
```

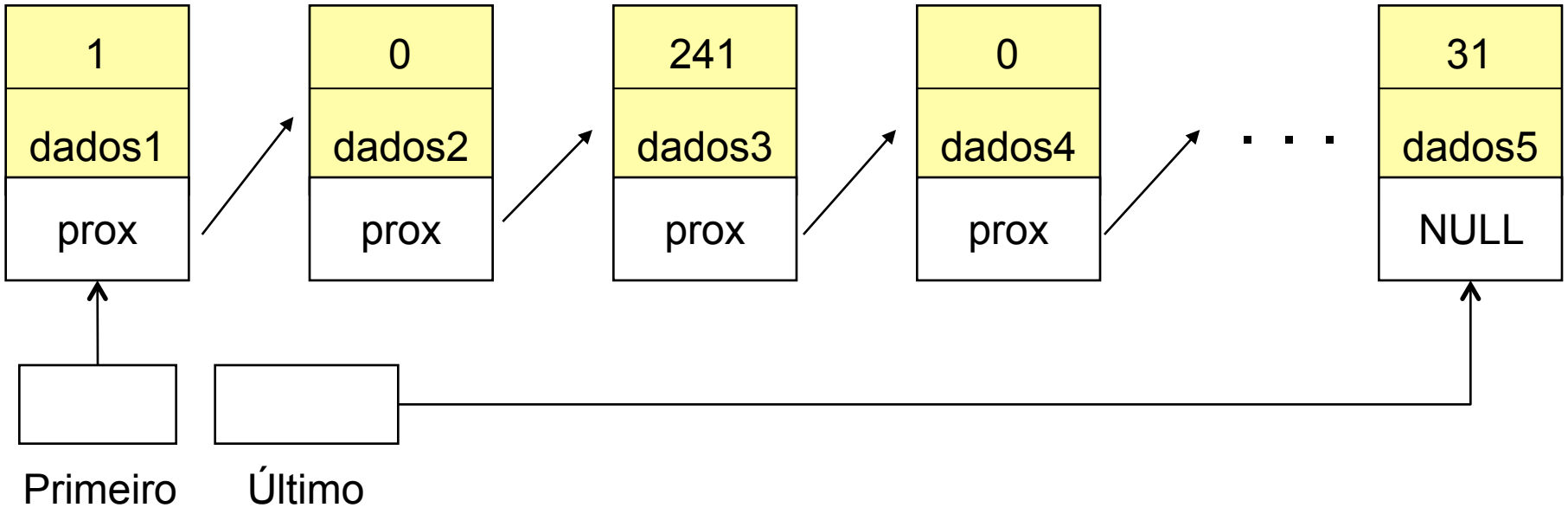
Alocação Encadeada (vantagens e desvantagens)

- 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.
  - **Custo linear** 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.

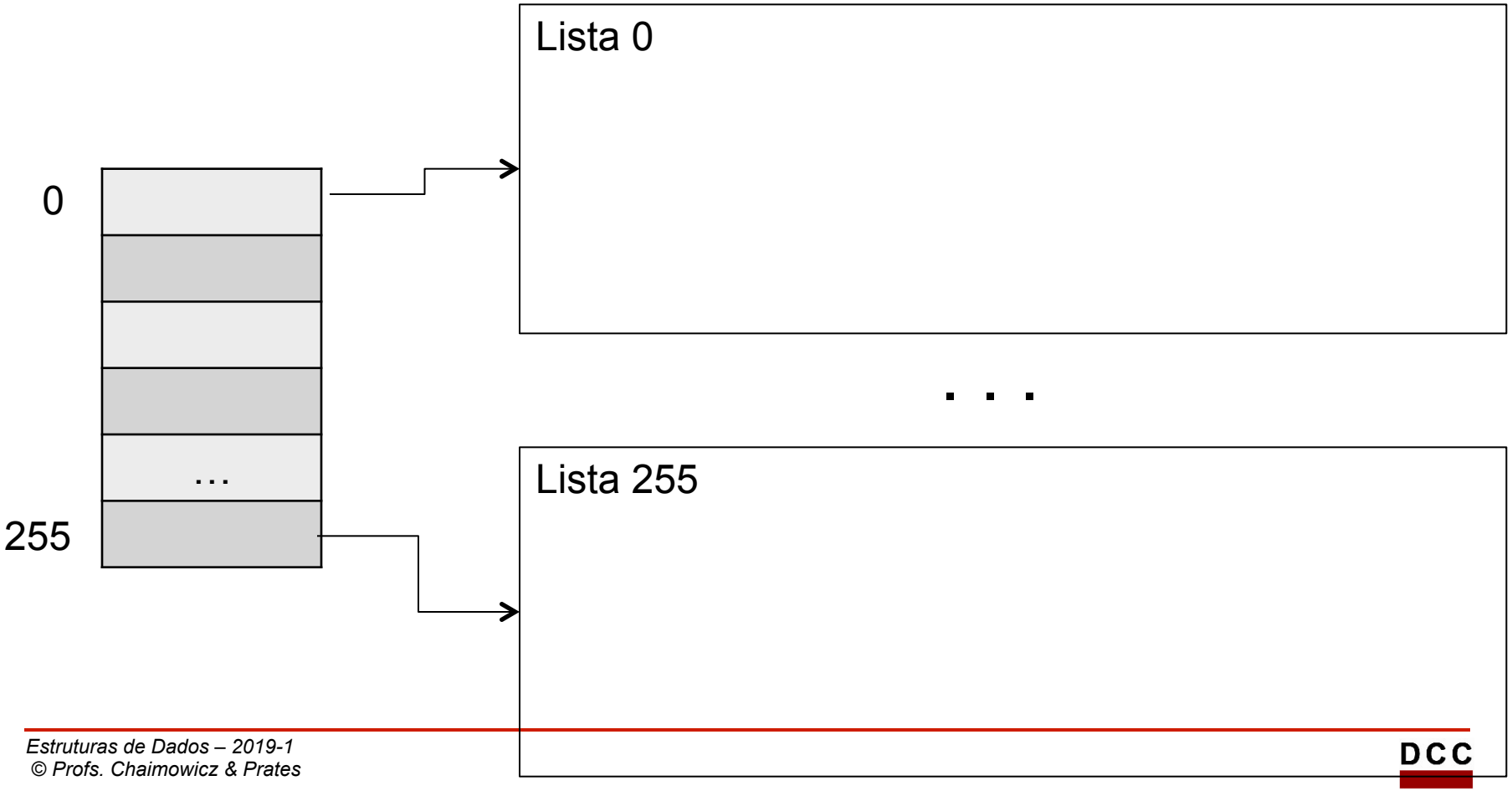


Exemplo: Ordenação

- **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

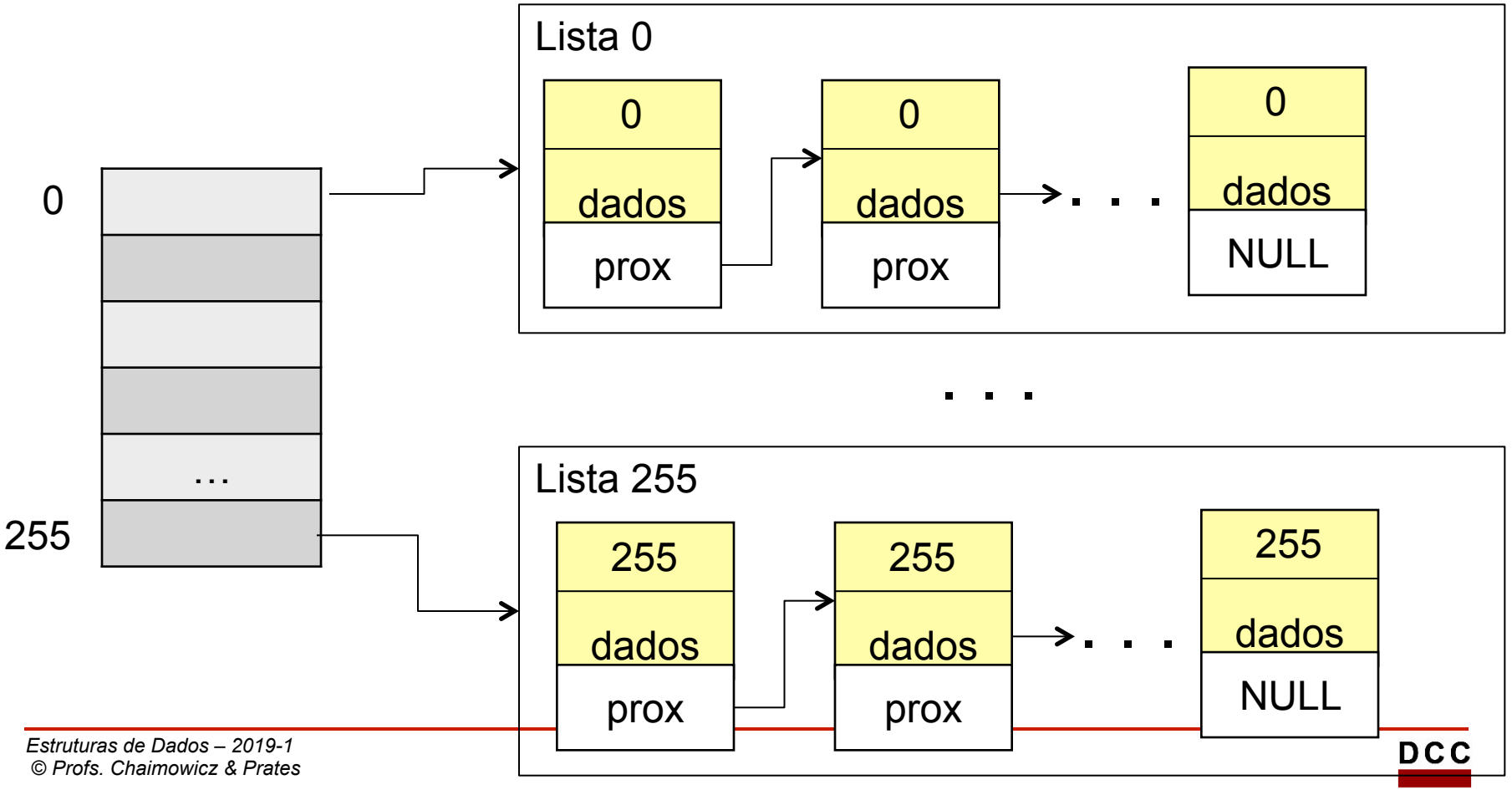
Exemplo: Ordenação

- **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.



Referências

Livro “Projeto de Algoritmos” – Nívio Ziviani

Capítulo 3 – Seção 3.1

http://www2.dcc.ufmg.br/livros/algoritmos/

(adaptado)