Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

TAD – Listas

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- **Listas Lineares**
  - Uma das formas mais simples de interligar os elementos de um conjunto.
  - Estrutura em que as operações **inserir**, **retirar** e **localizar** são definidas.
  - Podem **crescer** ou **diminuir** de tamanho durante a execução de um programa, de acordo com a demanda.
  - Itens podem ser **acessados**, **inseridos** ou **retirados** de uma lista.
  - Duas listas podem ser **concatenadas** para formar uma lista única, ou uma pode ser **partida** em duas ou mais listas.
  - São úteis em aplicações tais como manipulação simbólica, gerência de memória, simulações e compiladores.

1. Introdução

- **Listas Lineares: Definição**
  - Sequência de zero ou mais itens
    - $x_1, x_2, \ldots, x_n$, 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 $i = 1, 2, \ldots, n-1$
    - 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.

1. Introdução

- **Listas Lineares: TAD**
  - O que deveria conter?
    - Representação do tipo da lista.
    - Conjunto de operações que atuam sobre a lista.
  - Operações que deveriam fazer parte deste conjunto:

O conjunto de operações a ser definido depende de cada aplicação.

1. Introdução

- **Conjunto de operações para a maioria de aplicações:**

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.
5. Combinar duas ou mais listas lineares em uma lista única.
6. Dividir uma lista linear em duas ou mais listas.
7. Fazer uma cópia da lista linear.
8. Ordenar os itens da lista.
9. Pesquisar a ocorrência de um item com um valor particular.

1. Introdução

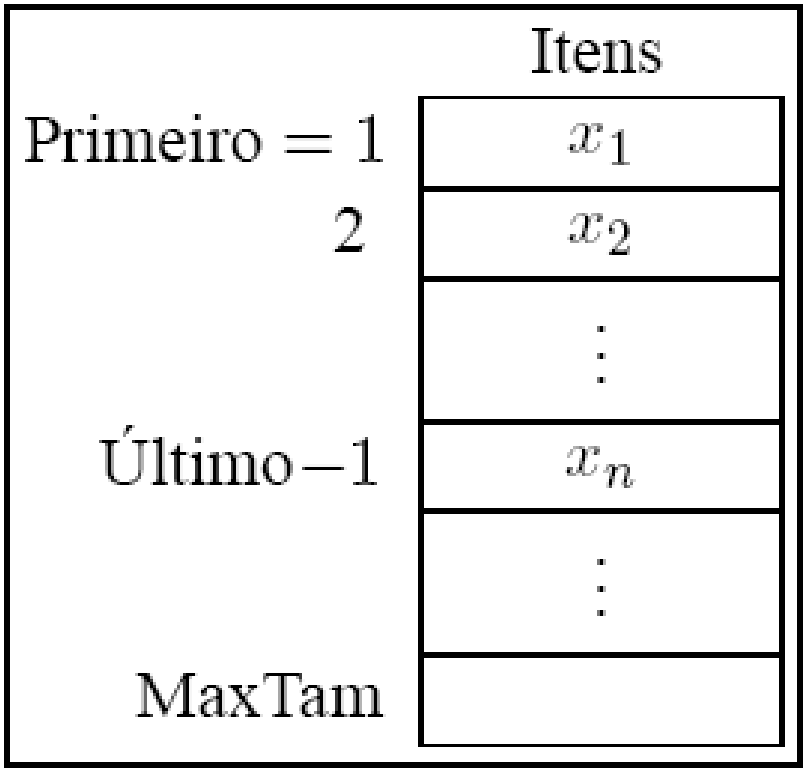
- **Exemplo de Protótipo para Operações:**
  - **`FLVazia(Lista)`**: Faz a lista ficar vazia.
  - **`LInsere(Lista, x)`**: Insere **`x`** após o último item da lista.
  - **`LRetira(Lista, p, x)`**: Retorna o item **`x`** que está na posição **`p`** da lista, retirando-o da lista e deslocando os itens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

1. Introdução

- **Implementações de Listas Lineares**
  - Várias estruturas de dados podem ser usadas para representar listas lineares.
    - Cada uma com vantagens e desvantagens particulares.
  - As duas representações mais utilizadas são:
    - Usando **alocação sequencial** e **estática** (com **vetores**).
    - Usando **alocação não sequencial** e **dinâmica** (com **ponteiros**): Estruturas Encadeadas.

2. Listas com Vetores

- Os itens da lista são armazenados em posições contíguas de memória.
- A lista pode ser percorrida em qualquer direção.
- A inserção de um novo item pode ser realizada após o último item com custo constante.
- A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção.
- Retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.



2. Listas com Vetores

```
#define INICIO 0
#define MAXTAM 1000
typedef int TChave;
typedef int TApontador;

typedef struct {
   TChave chave;
   /* outros componentes */
} TItem;

typedef struct {
   TItem item[MAXTAM];
   TApontador primeiro, ultimo;
} TLista;
```

- Os itens são armazenados em um vetor de tamanho suficiente para armazenar a lista.
- O campo **ultimo** aponta para a posição seguinte a do último elemento da lista.
- O **i-ésimo** item da lista está armazenado na **(i-1)-ésima** posição do vetor, **0 ≤ i < ultimo**.
- A constante **MAXTAM** define o tamanho máximo permitido para a lista.

2. Listas com Vetores

```
void Flvazia (TLista *pLista) {
   pLista->primeiro = INICIO;
   pLista->ultimo = pLista->primeiro;
}

int Lvazia (TLista* pLista) {
   return (pLista->ultimo == pLista->primeiro);
}

int Linsere (TLista* pLista, TItem x){
   if (pLista->ultimo == MAXTAM)
      return 0; /* lista cheia */
   pLista->item[pLista->ultimo++] = x;
   return 1;
}
```

2. Listas com Vetores

```
int Lretira (TLista* pLista, TApontador p, TItem *px) {
   if (Lvazia(pLista) || p >= pLista->ultimo)
     return 0;
   *px = pLista->item[p];
   pLista->ultimo--;
   while (p < pLista->ultimo)
      pLista->item[p] = pLista->item[++p];
   return 1;
}

void Limprime (TLista* pLista) {
   for(int i = pLista->primeiro; i < pLista->ultimo; i++)
      printf("%d\n", pLista->item[i].chave);
}
```

2. Listas com Vetores

- **Vantagem:**
  - Economia de memória (os apontadores são implícitos nesta estrutura).
  - Acesso a qualquer elemento da lista é feito em tempo O(1).
- **Desvantagens:**
  - Custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso;
  - Em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o Pascal ou C pode ser problemática porque neste caso o tamanho máximo da lista tem de ser definido em tempo de compilação.

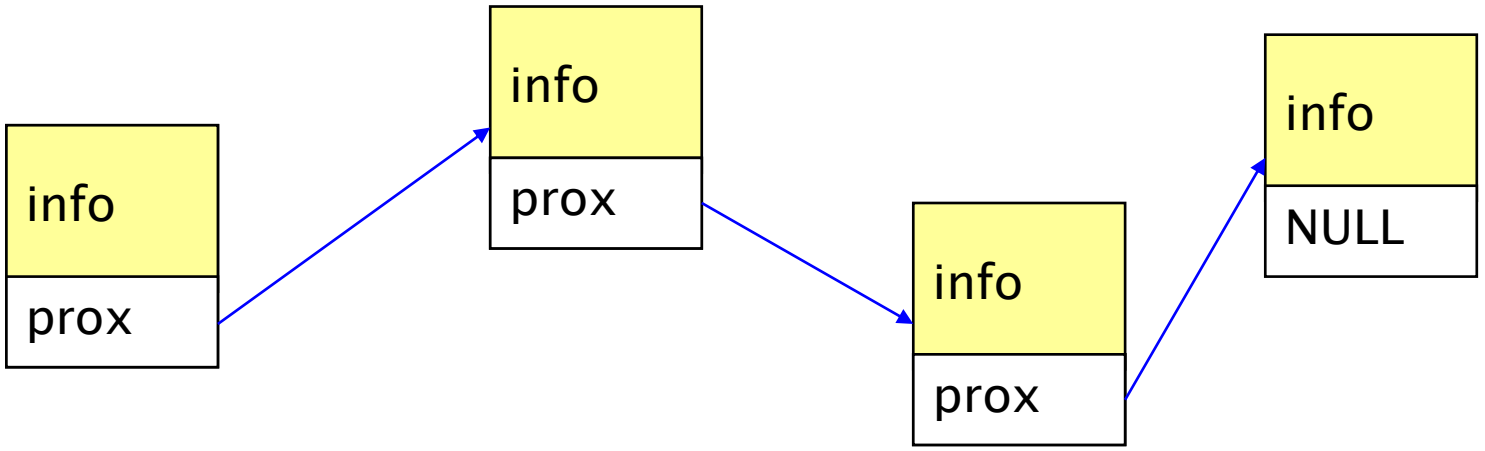
3. Listas com Ponteiros (Encadeadas)

- Um dos principais problema de utilizar vetores para implementar listas diz respeito ao próprio conteúdo da lista:
  - Se a lista aumentar e depois diminuir drasticamente de tamanho, o custo das inserções e remoções pode se tornar um problema.

- **Solução: Listas Encadeadas**
  - O que é?
    - Implementação de uma lista utilizando apenas **ponteiros**.

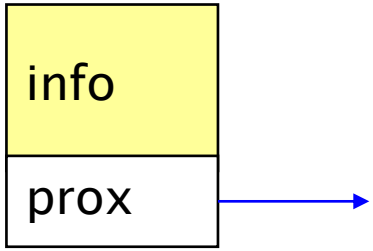
3. Listas com Ponteiros (Encadeadas)

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



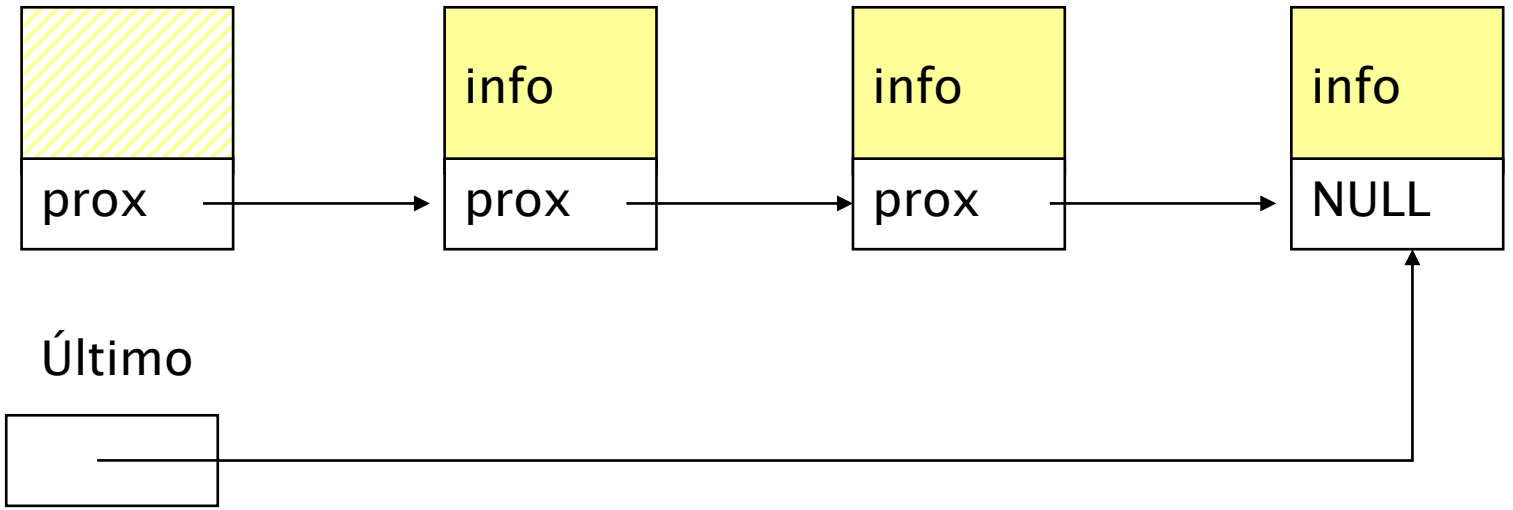
3. Listas com Ponteiros (Encadeadas)

- **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.



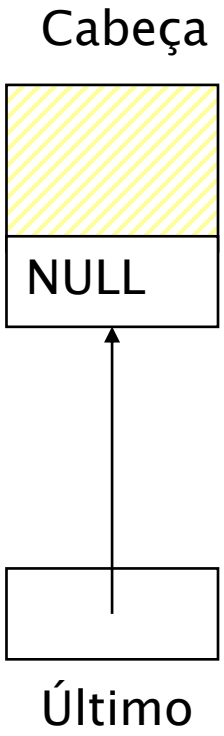
3. Listas com Ponteiros (Encadeadas)

- Uma lista pode ter uma célula **cabeça**, antecedendo o primeiro elemento.
- Pode possuir também um apontador para o **último** elemento.



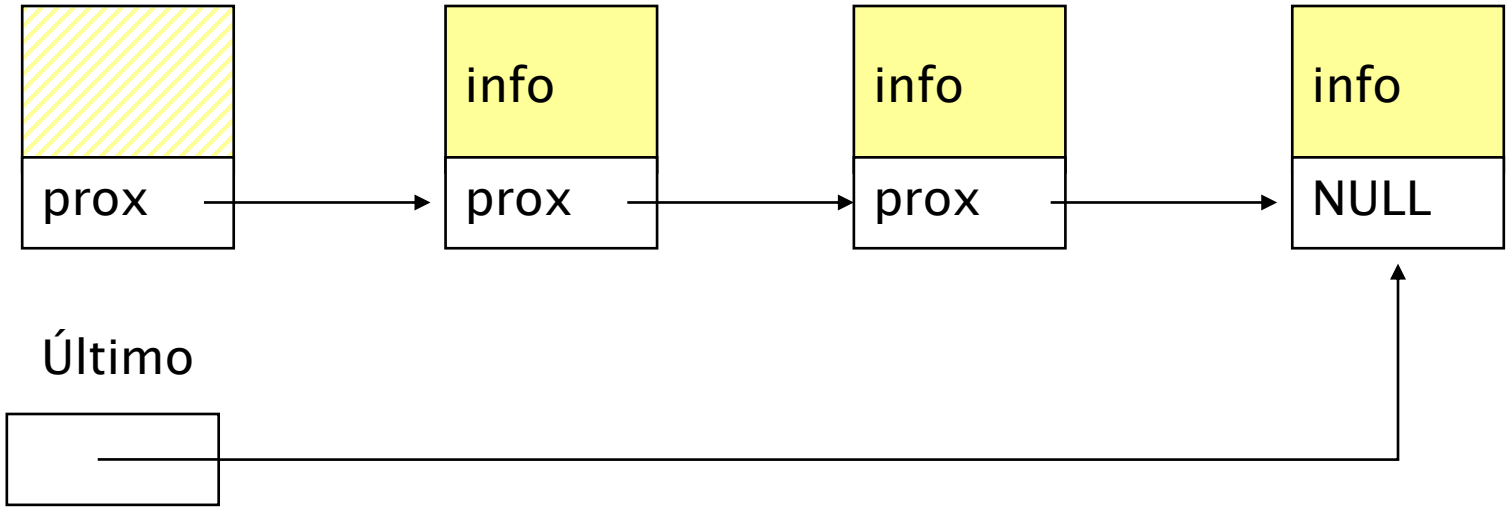
3. Listas com Ponteiros (Encadeadas)

- Cria Lista Vazia



3. Listas com Ponteiros (Encadeadas)

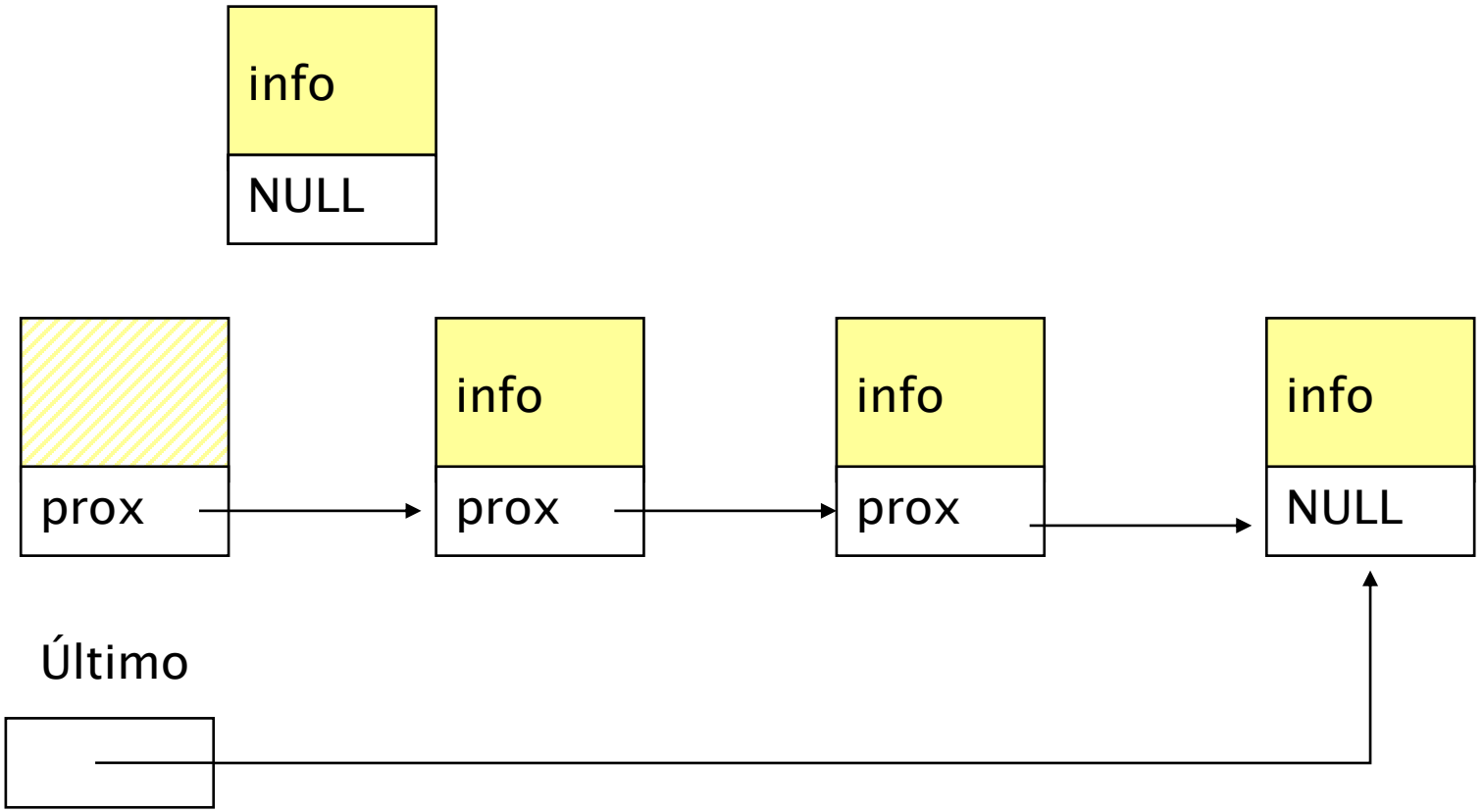
- 3 opções de posições onde pode **inserir**:



  - 1ª posição
  - Última posição
  - Após um elemento **x** qualquer

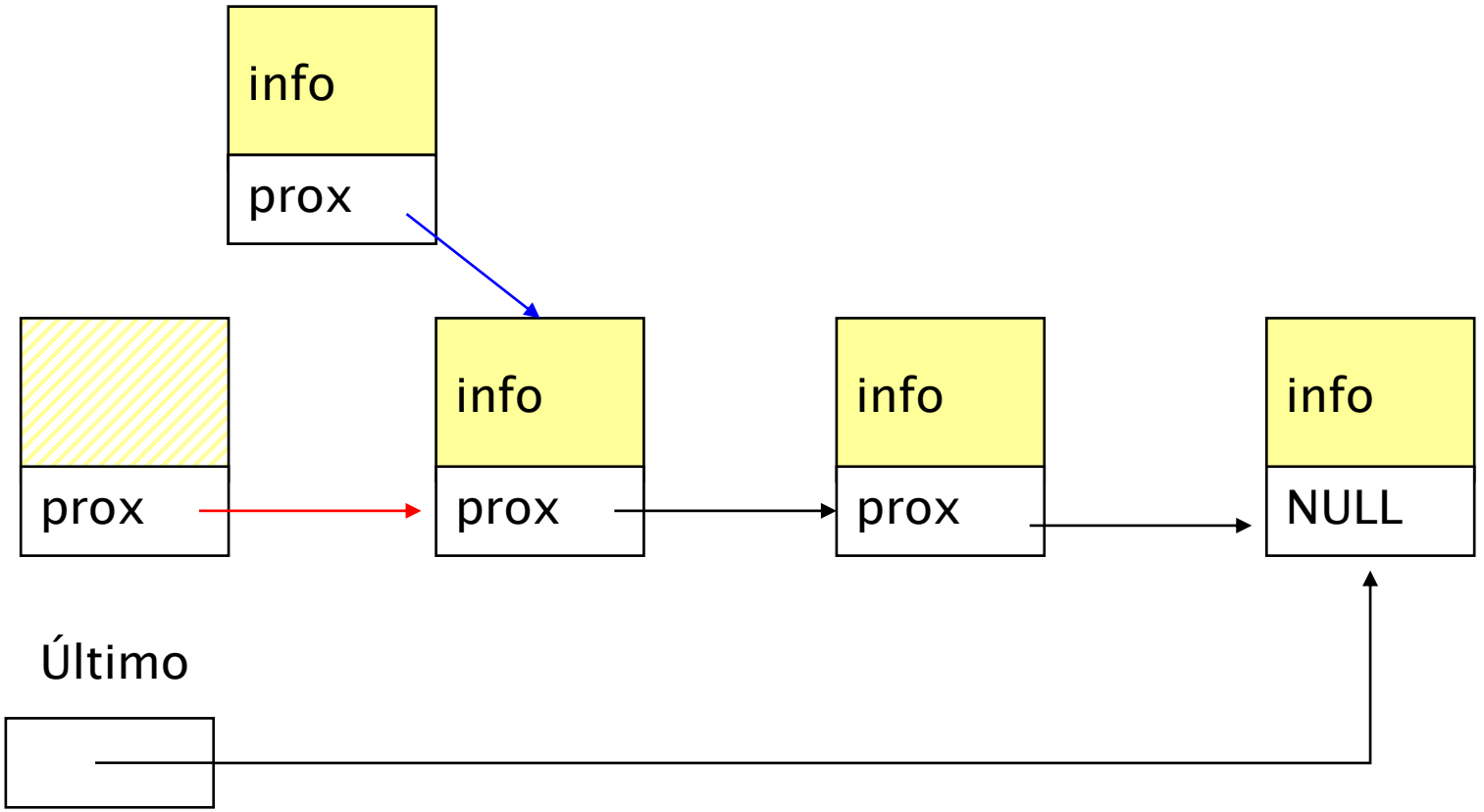
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na **1ª posição (1/3)**



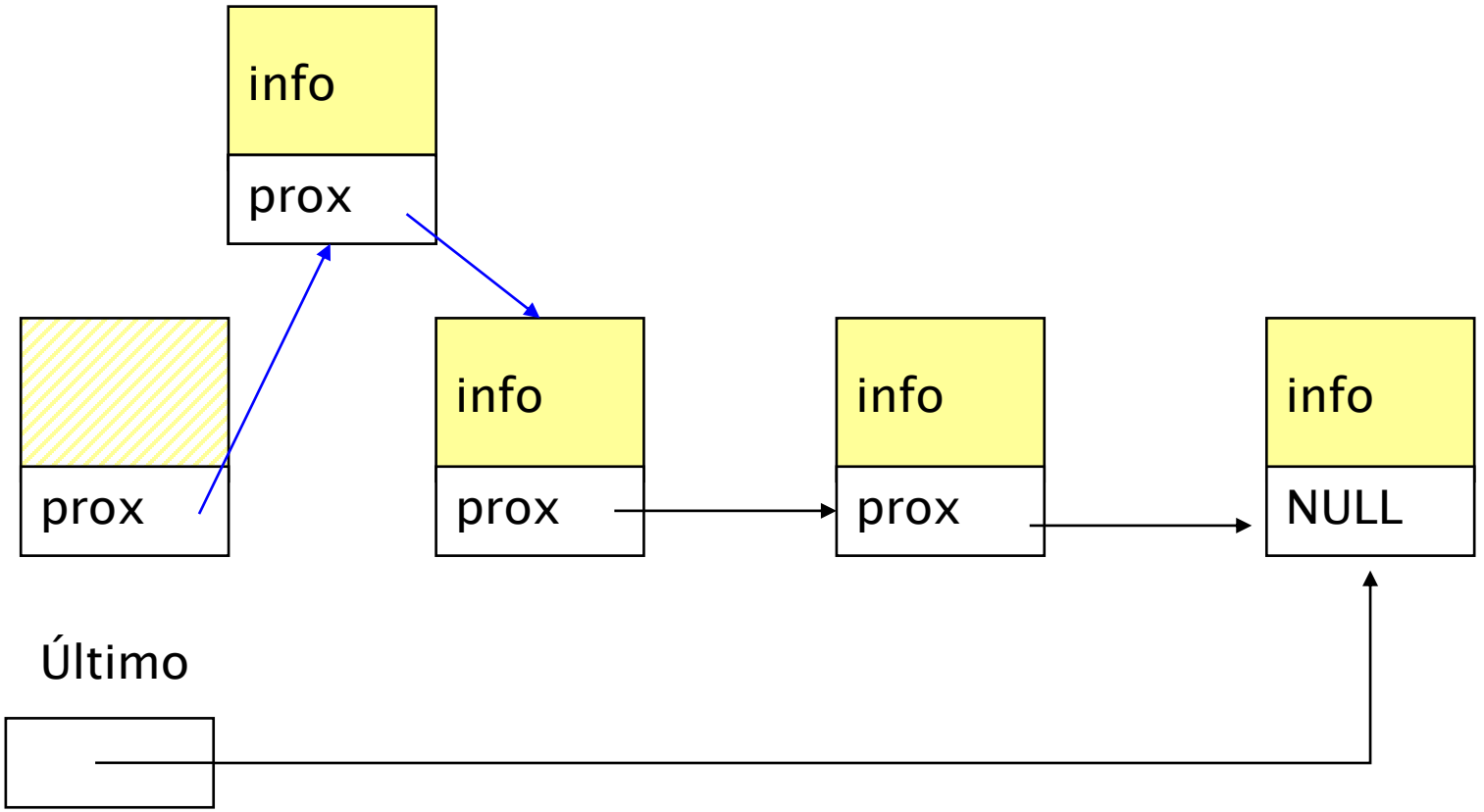
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na 1ª posição (2/3)



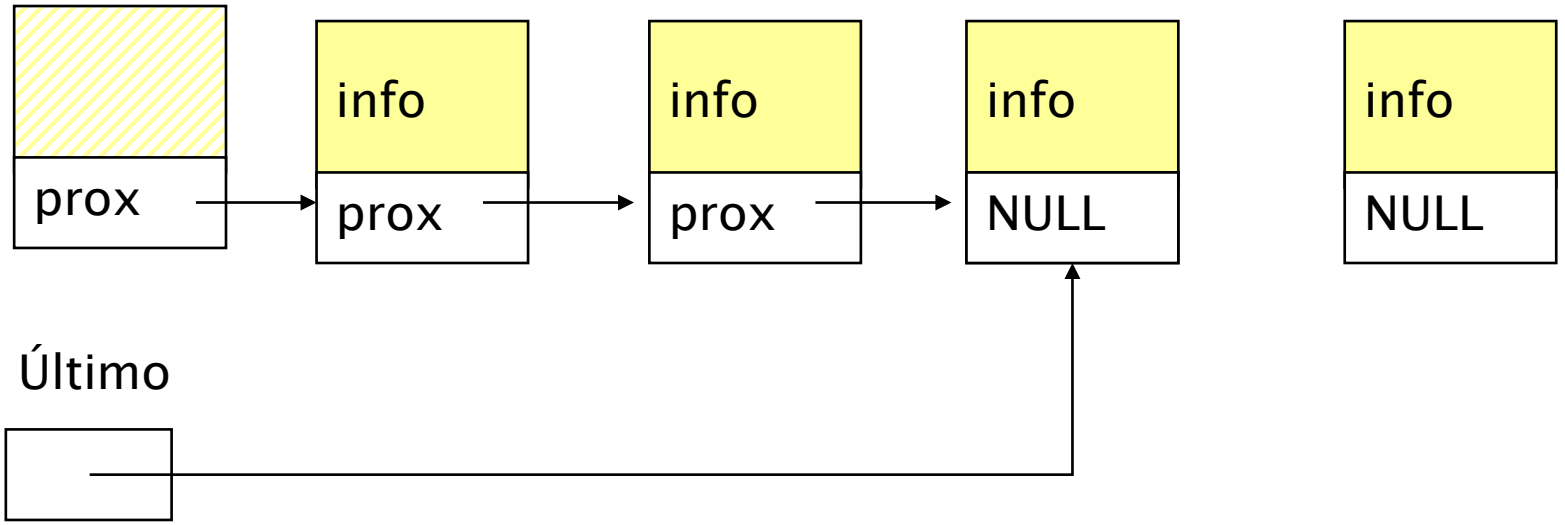
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na 1ª posição (3/3)



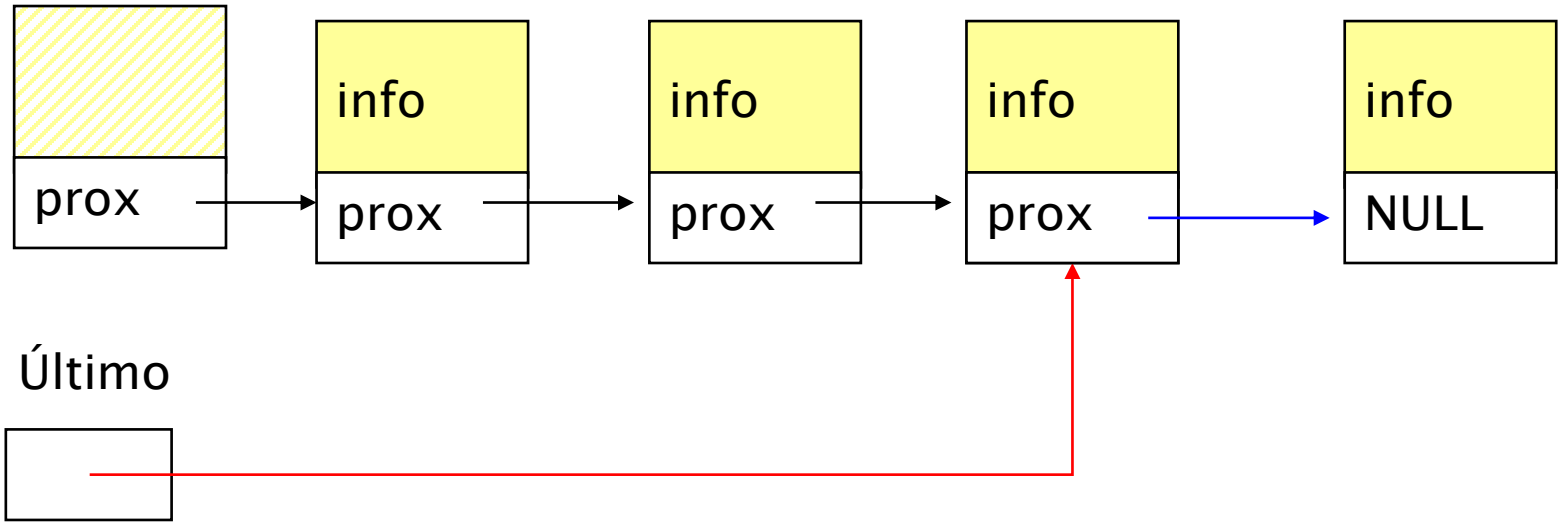
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na **última posição (1/3)**



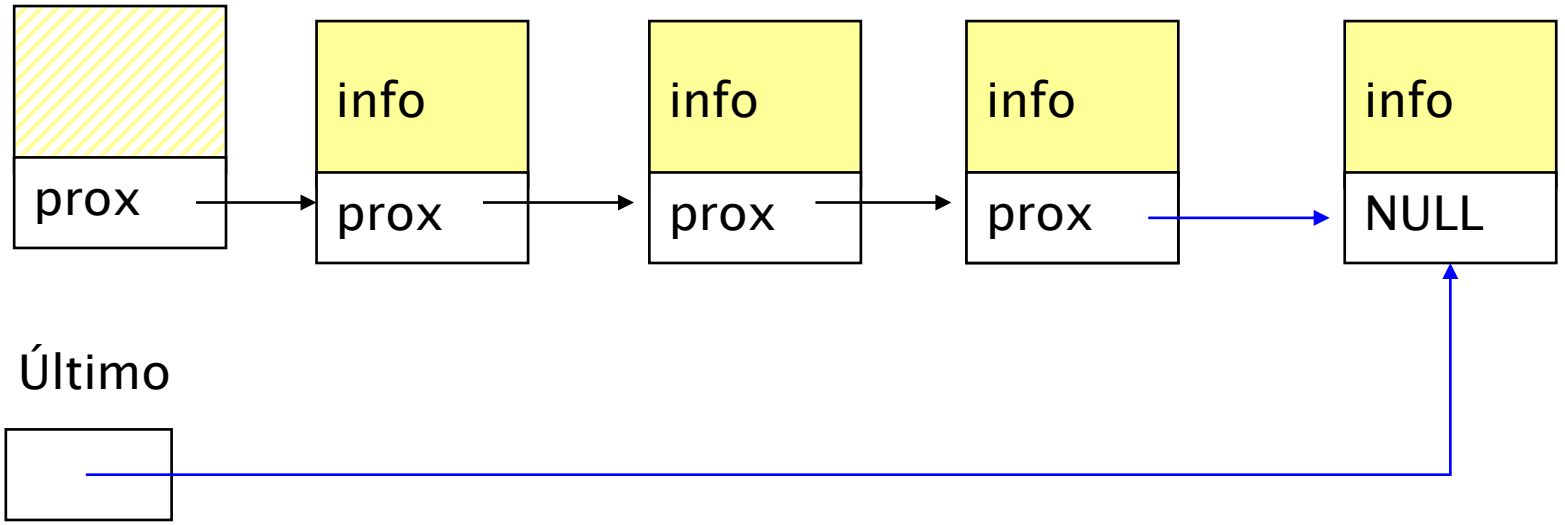
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na **última posição (2/3)**



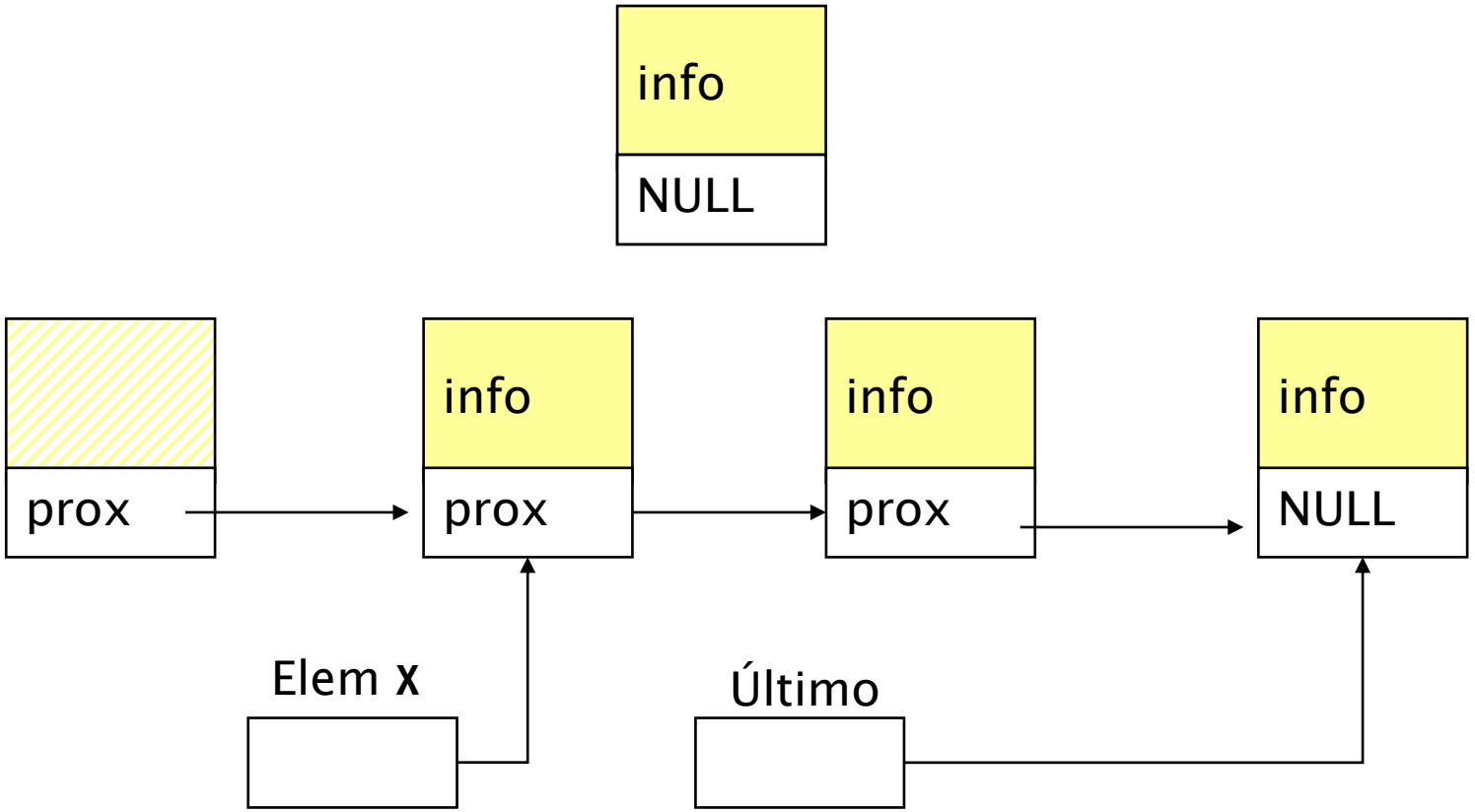
3. Listas com Ponteiros (Encadeadas)

- **Inserir** na **última posição (3/3)**



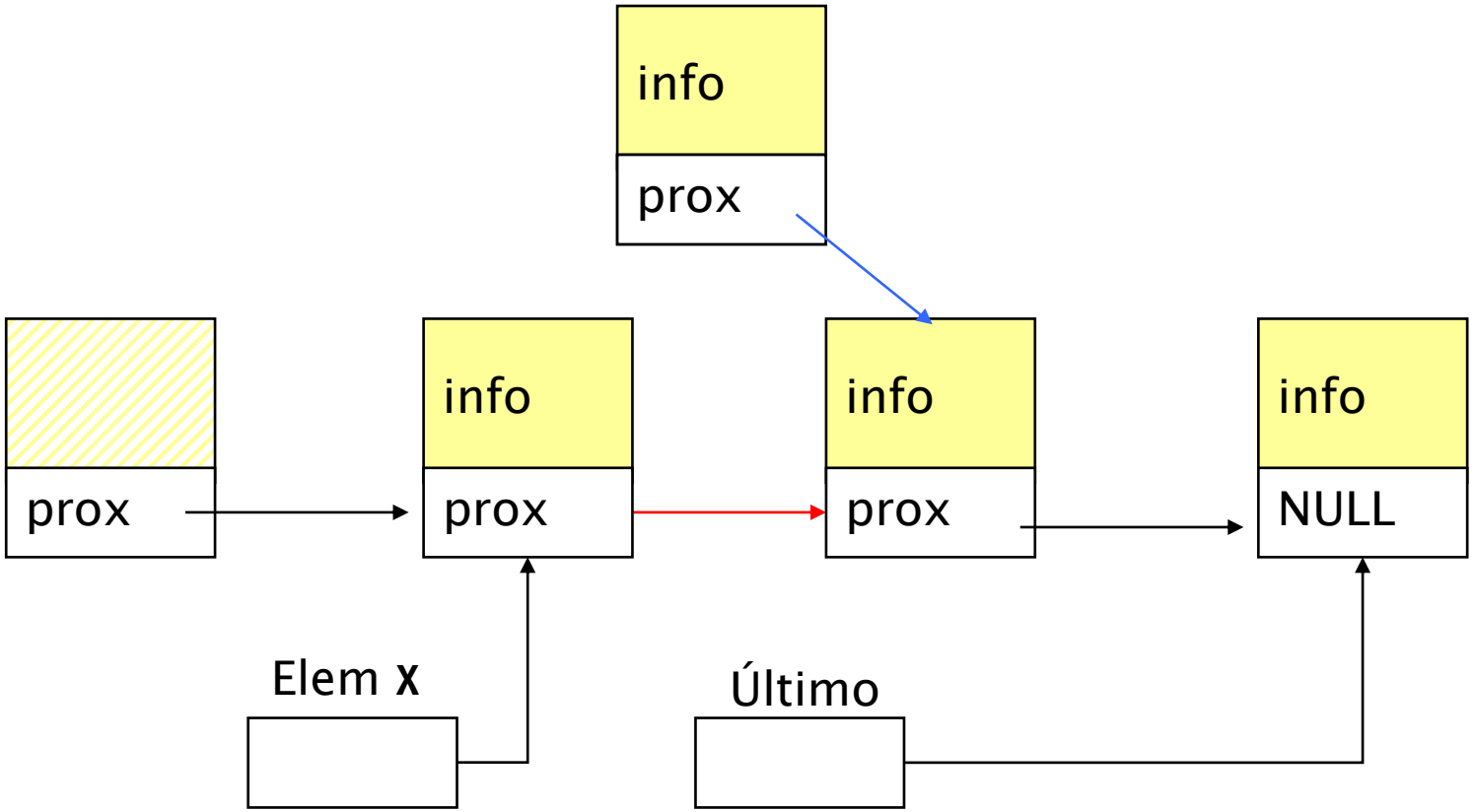
3. Listas com Ponteiros (Encadeadas)

- **Inserir** após um elemento **x** qualquer (1/3)



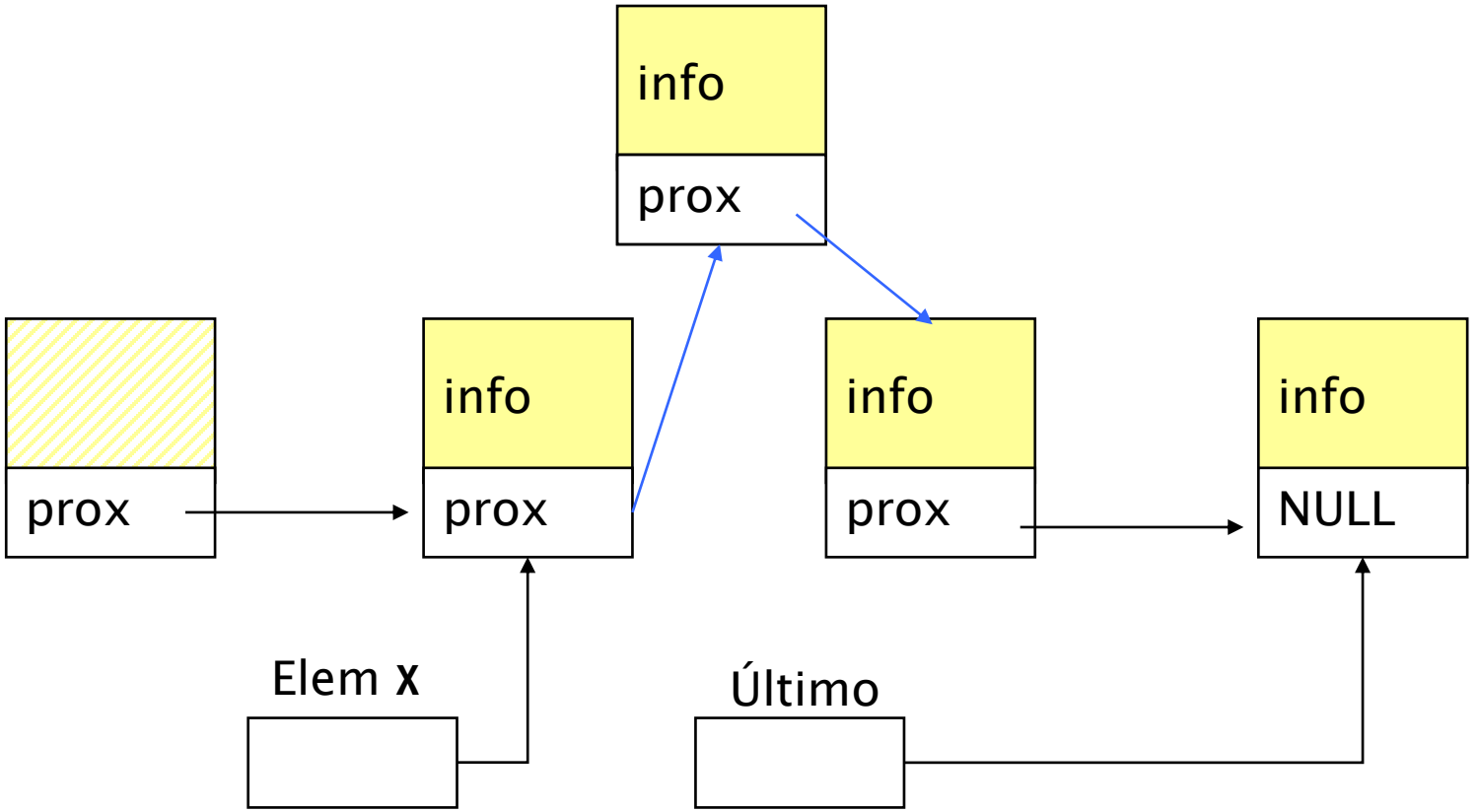
3. Listas com Ponteiros (Encadeadas)

- **Inserir** após um elemento **x** qualquer (2/3)



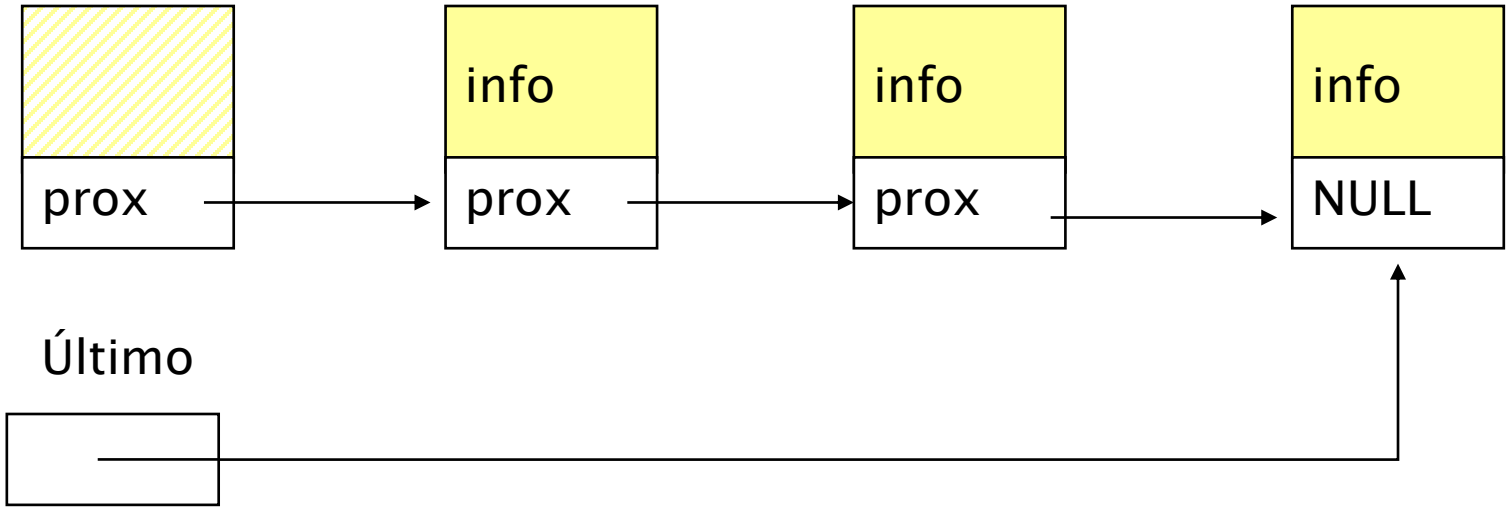
3. Listas com Ponteiros (Encadeadas)

- **Inserir** após um elemento **x** qualquer (3/3)



3. Listas com Ponteiros (Encadeadas)

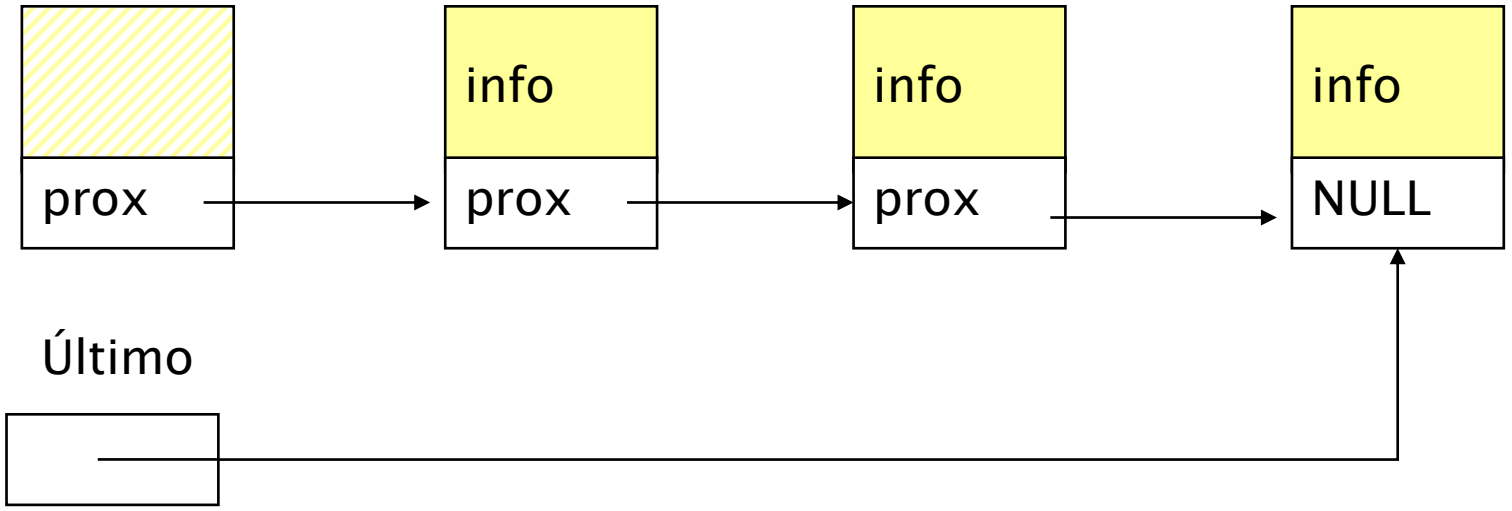
- 3 opções de posições onde pode **retirar**:



  - 1ª posição
  - Última posição
  - Um elemento **x** qualquer

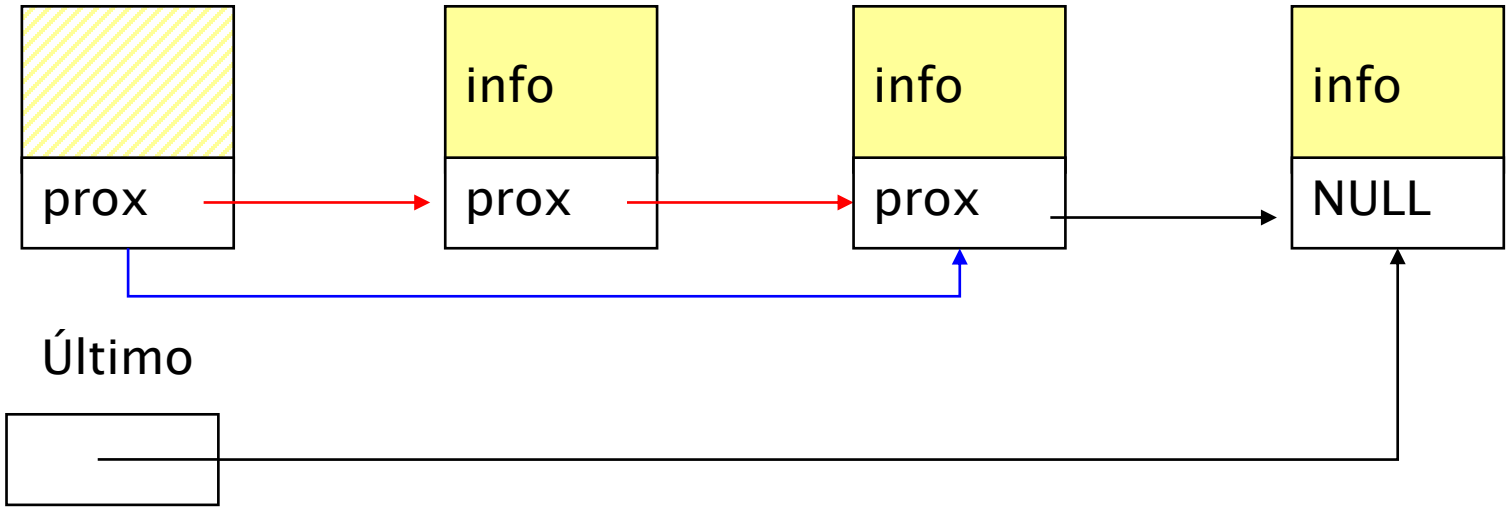
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da **1ª posição (1/3)**



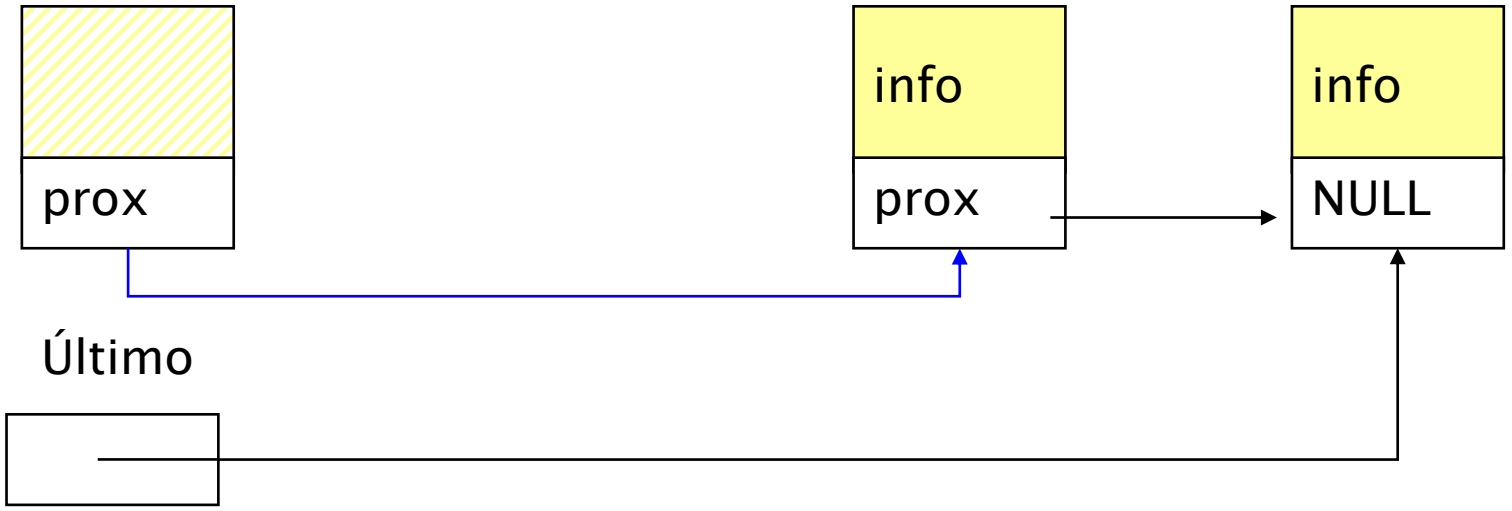
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da 1ª posição (2/3)



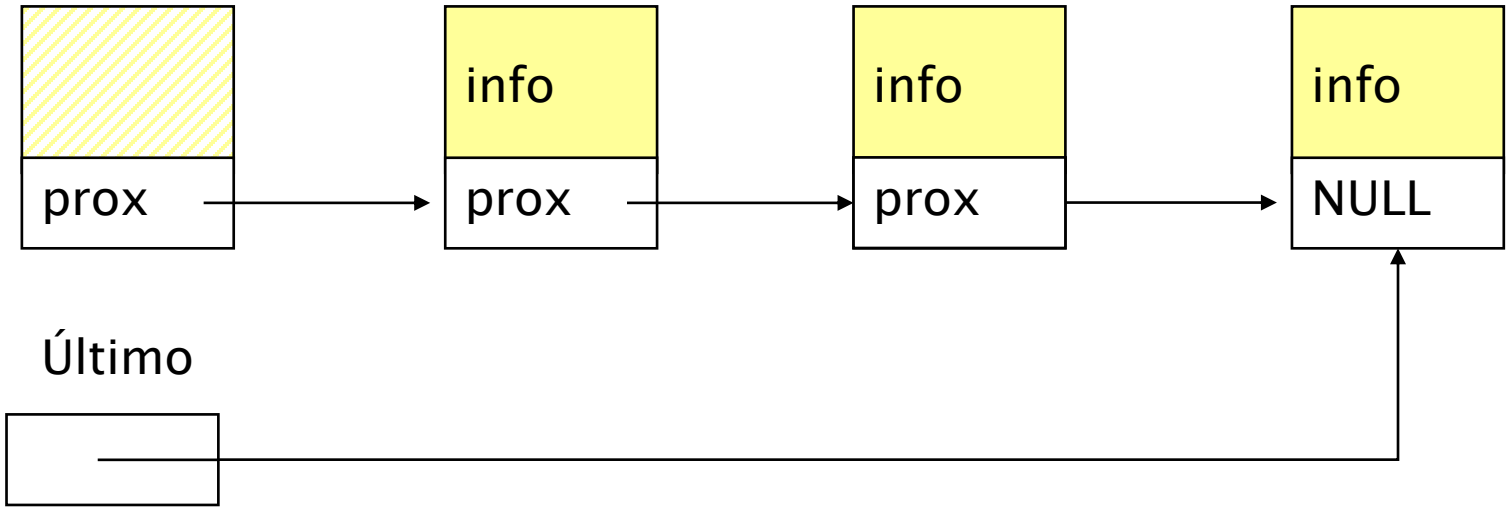
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da 1ª posição (3/3)



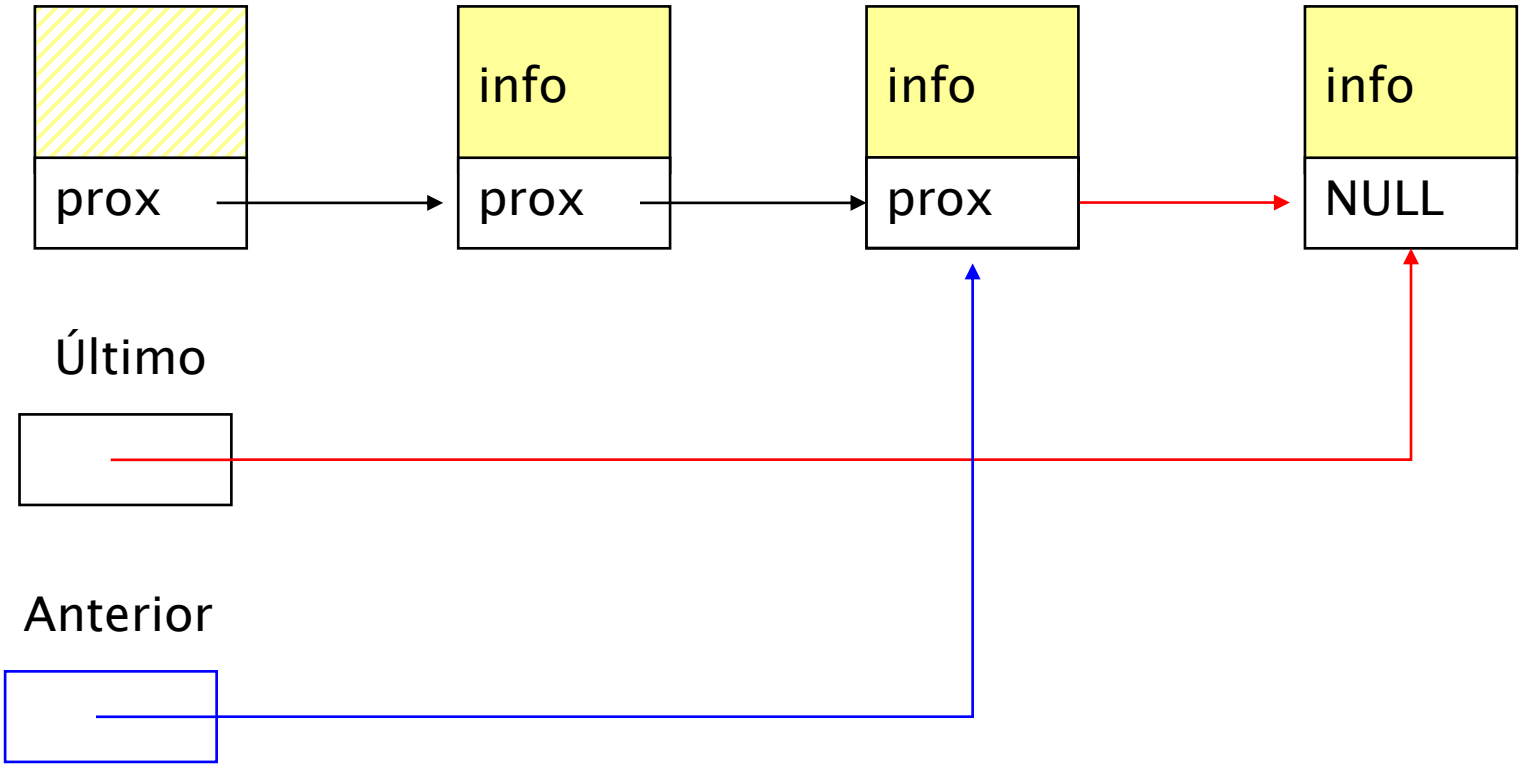
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da **última posição (1/4)**



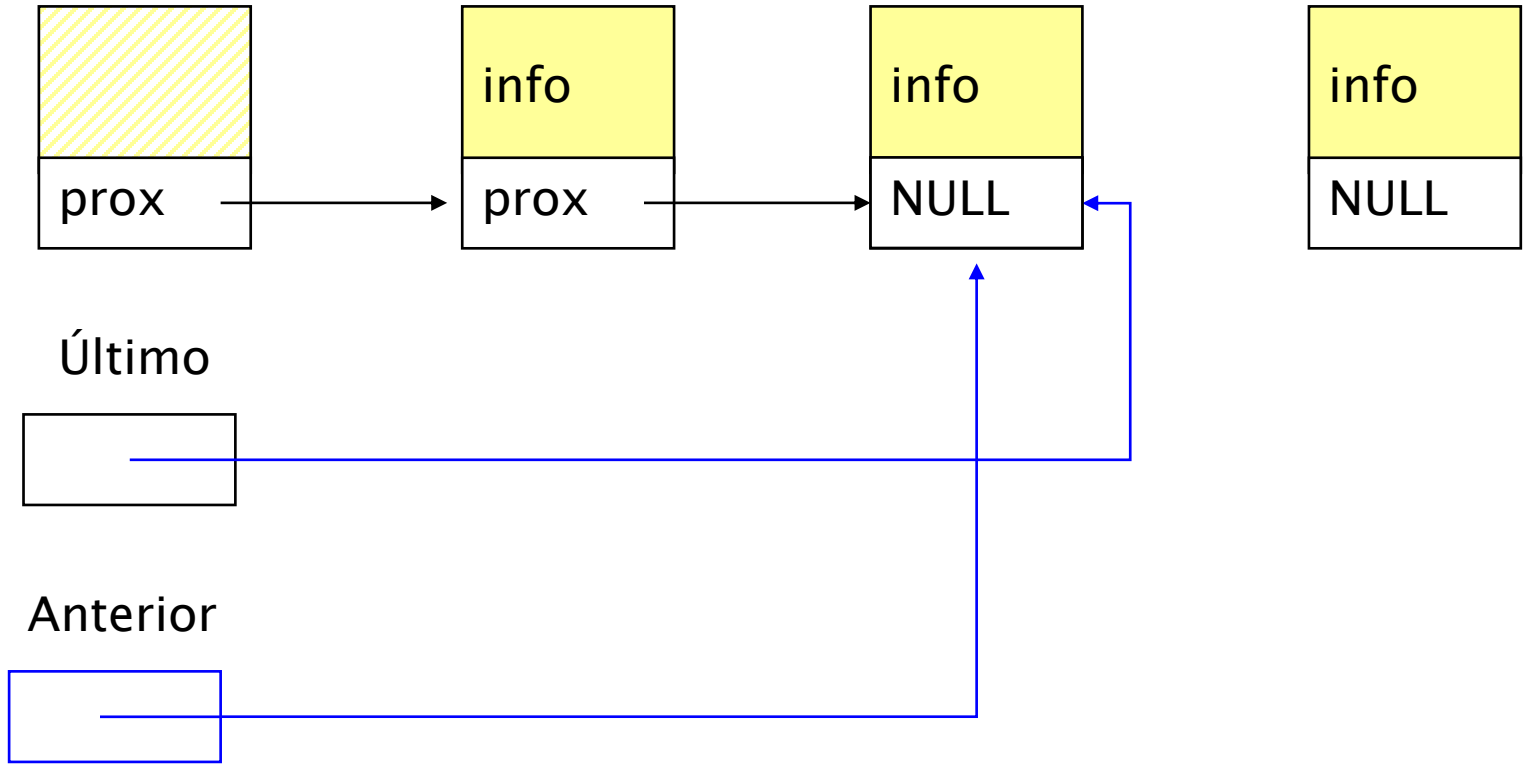
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da **última posição (2/4)**



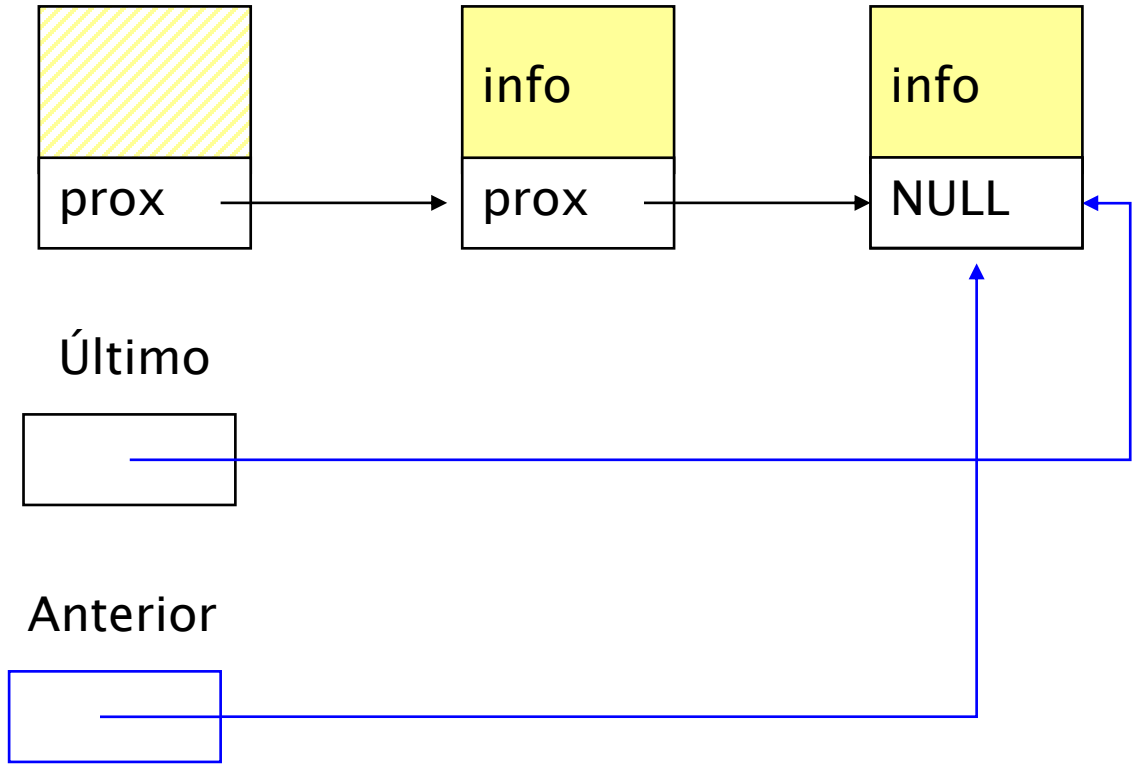
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da **última posição (3/4)**



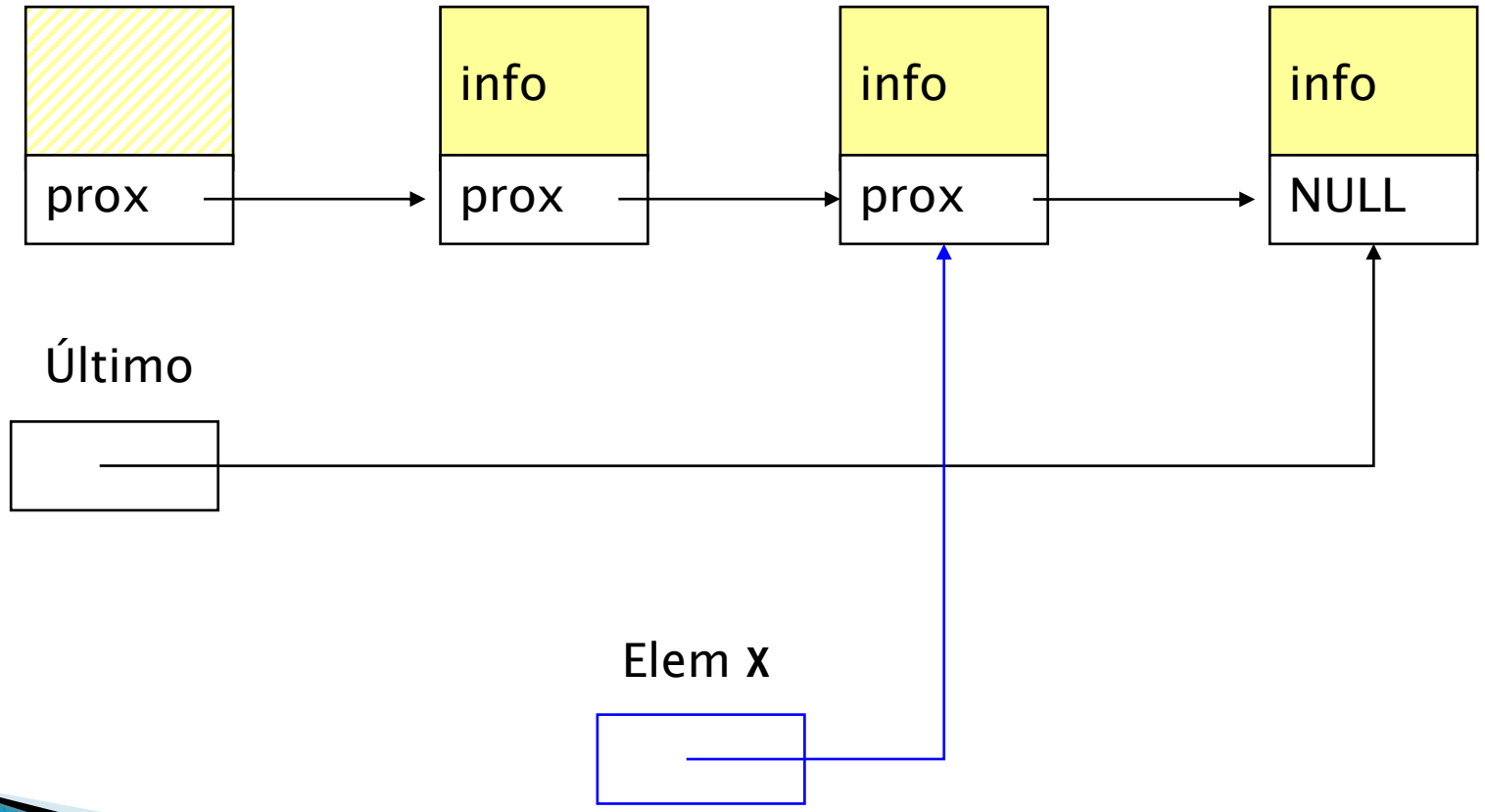
3. Listas com Ponteiros (Encadeadas)

- **Retirar** da **última posição (4/4)**



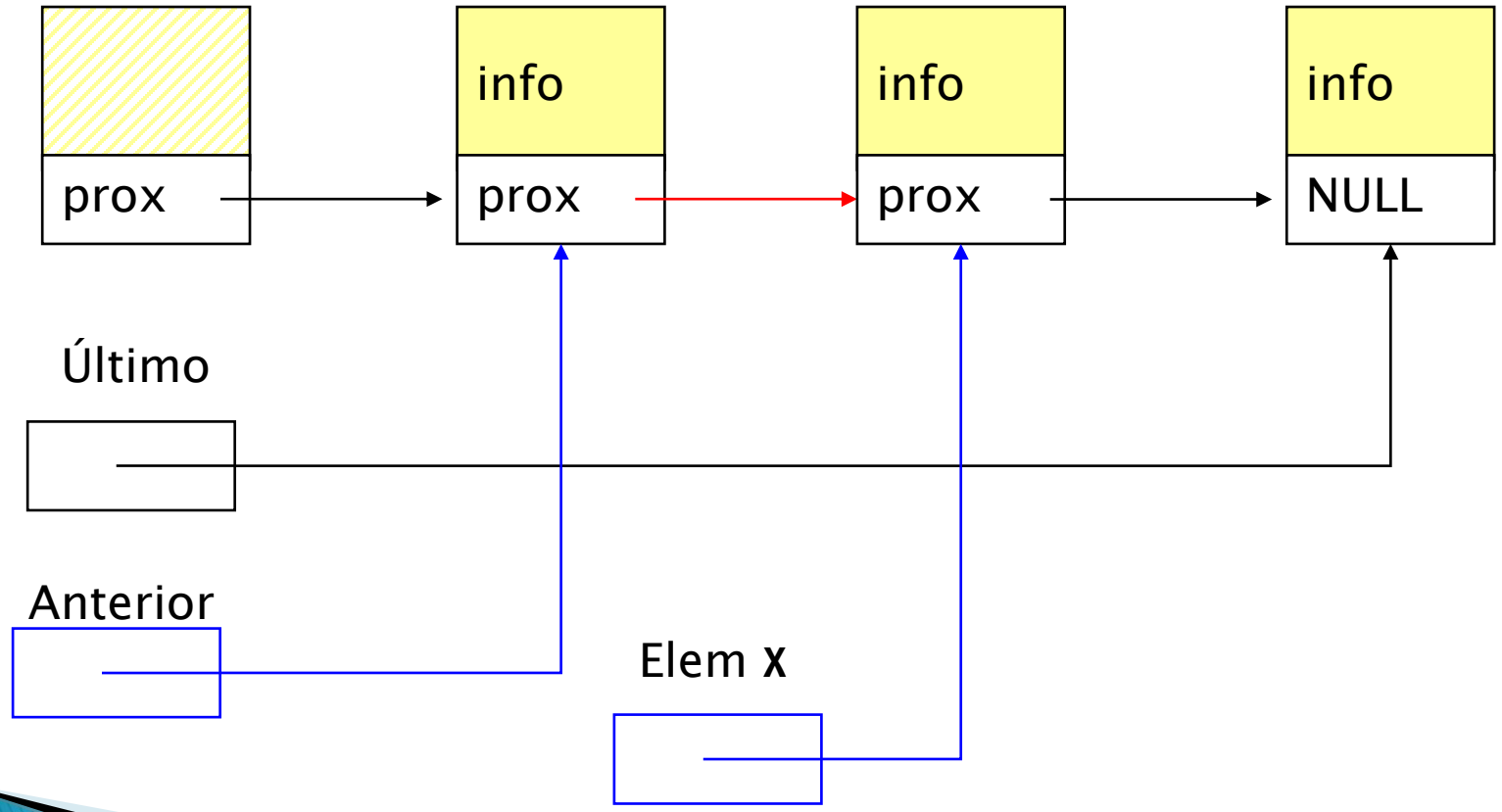
3. Listas com Ponteiros (Encadeadas)

- **Retirar** um **elemento x qualquer (1/4)**



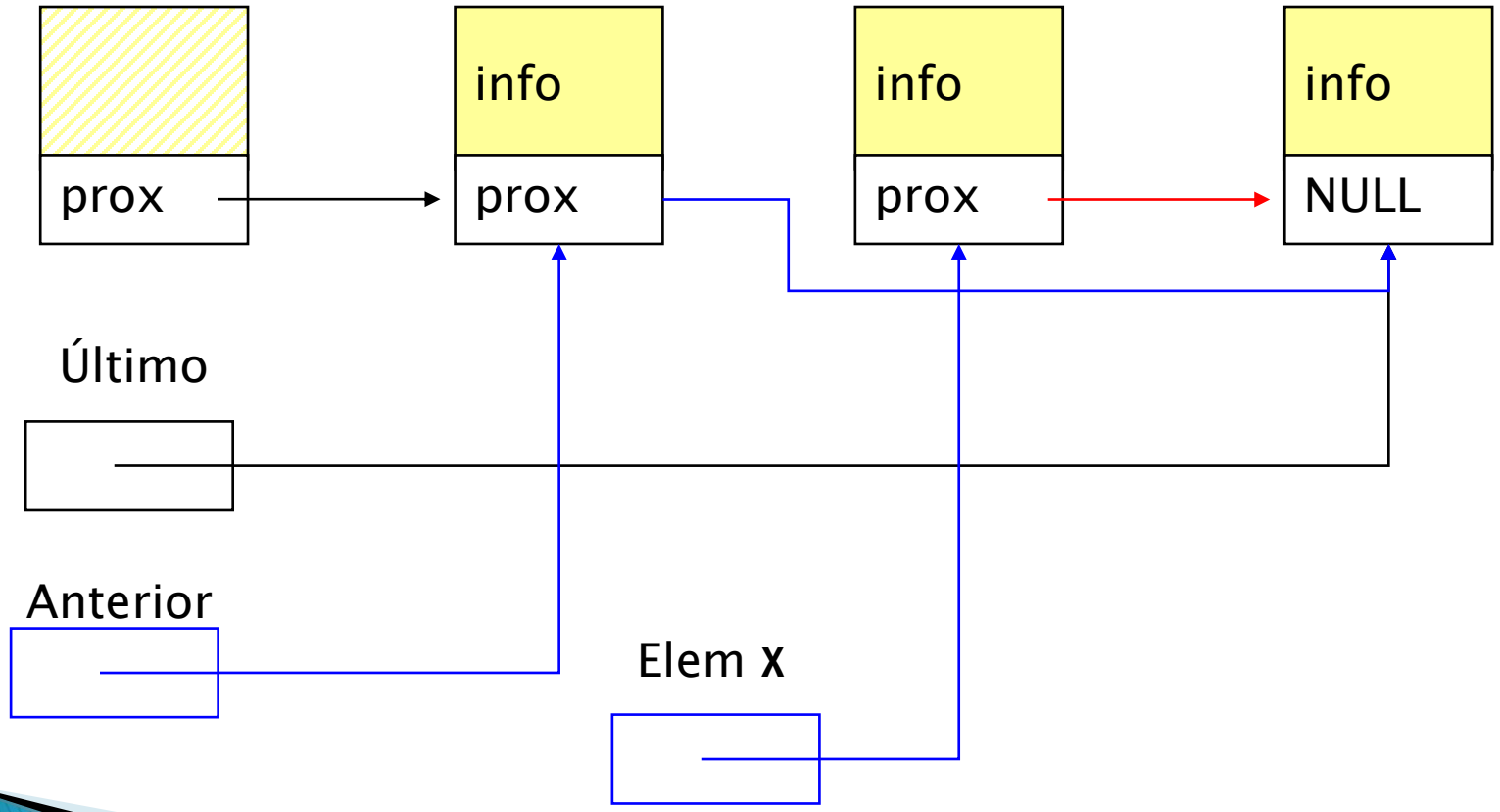
3. Listas com Ponteiros (Encadeadas)

- **Retirar** um **elemento x qualquer (2/4)**



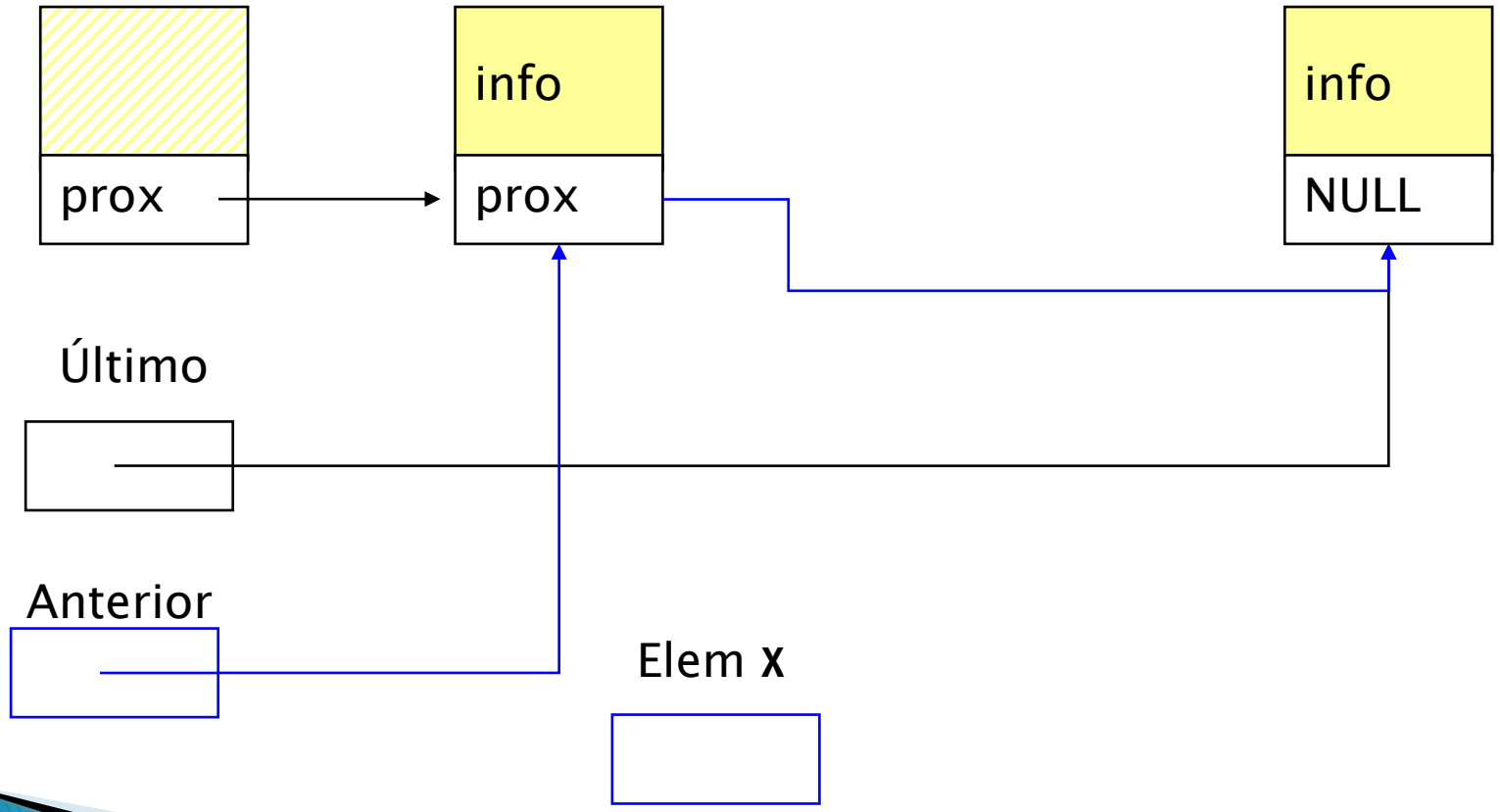
3. Listas com Ponteiros (Encadeadas)

- **Retirar** um **elemento x qualquer (3/4)**



3. Listas com Ponteiros (Encadeadas)

- **Retirar** um **elemento x qualquer (4/4)**



3.1. Estrutura da Lista Encadeada

```
typedef int TChave;

typedef struct {
   TChave chave;
   /* outros componentes */
} TItem;

typedef struct TCelulaEst {
   TItem item;
   struct TCelulaEst* pProx; /* Apontador pProx; */
} TCelula;

typedef struct {
  TCelula* pPrimeiro;
  TCelula* pUltimo;
} TLista;
```

3.2. Operações com Lista Encadeada

- **Com cabeça**

```
void FLVazia (TLista* pLista) {
    pLista->pPrimeiro = (TCelula*)
malloc(sizeof(TCelula));
    pLista->pUltimo = pLista->pPrimeiro;
    pLista->pPrimeiro->pProx = NULL;
}

int Lvazia (TLista* pLista) {
    return (pLista->pPrimeiro == pLista->pUltimo);
}
```

3.2. Operações com Lista Encadeada

- **Sem cabeça**

```
void Flvazia (TLista* pLista) {
    pLista->pPrimeiro = NULL;
    pLista->pUltimo = NULL;
}

int Lvazia (TLista* pLista) {
    return (pLista->pUltimo == NULL);
}
```

3.2. Operações com Lista Encadeada

- **Com cabeça**

```
void Linsere (TLista* pLista, TItem* pItem) {
    pLista->pUltimo->pProx =
            (TCelula*) malloc(sizeof(TCelula));
    pLista->pUltimo = pLista->pUltimo->pProx;
    pLista->pUltimo->item = *pItem;
    pLista->pUltimo->pProx = NULL;
}
```

3.2. Operações com Lista Encadeada

- **Sem cabeça**

```
void Linsere (TLista *pLista, TItem* pItem) {
   if (pLista->pUltimo == NULL){
      pLista->pUltimo = (TCelula*) malloc(sizeof(TCelula));
      pLista->pPrimeiro = pLista->pUltimo; }
   else {
      pLista->pUltimo->pProx =
                        (TCelula*) malloc(sizeof(TCelula));
      pLista->pUltimo = pLista->pUltimo->pProx; }
   pLista->pUltimo->item = *pItem;
   pLista->pUltimo->pProx = NULL;
}
```

3.2. Operações com Lista Encadeada

- **Com cabeça**

```
int Lretira (TLista* pLista, TItem* pItem) {
    TCelula* pAux;
    if (Lvazia(pLista))
        return 0;
    *pItem = pLista->pPrimeiro->pProx->item;
    pAux = pLista->pPrimeiro;
    pLista->pPrimeiro = pLista->pPrimeiro->pProx;
    free(pAux);
    return 1;
}
```

3.2. Operações com Lista Encadeada

- **Sem cabeça**

```
int Lretira (TLista* pLista, TItem* pItem) {
    TCelula* pAux;
    if(Lvazia(pLista))
        return 0;
    *pItem = pLista->pPrimeiro->item;
    pAux = pLista->pPrimeiro;
    pLista->pPrimeiro = pLista->pPrimeiro->pProx;
    free(pAux);
    if(pLista->pPrimeiro == NULL)
        pLista->pUltimo = NULL; /* lista vazia */
    return 1;
}
```

3.2. Operações com Lista Encadeada

- **Com cabeça**

```
void Limprime (TLista* pLista) {
    TCelula* pAux;
    pAux = pLista->pPrimeiro->pProx;
    while (pAux != NULL) {
        printf("%d\n", pAux->item.chave);
        pAux = pAux->pProx; /* próxima célula */
    }
}
```

3.2. Operações com Lista Encadeada

- **Sem cabeça**

```
void Limprime (TLista* pLista) {
    TCelula* pAux;
    pAux = pLista->pPrimeiro;
    while (pAux != NULL) {
        printf("%d\n", pAux->item.chave);
        pAux = pAux->pProx; /* próxima célula */
    }
}
```

3. Listas com Ponteiros (Encadeadas)

- **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.
  - Percorrer a lista, procurando pelo **`i-ésimo`** elemento.
  - Descobrir o elemento anterior (ou anteriores).

3.3. Exemplo: Vestibular

- Num vestibular, cada candidato tem direito a três opções para tentar uma vaga em um dos sete cursos oferecidos.

- Para cada candidato é lido um registro:
  - **Chave**: número de inscrição do candidato.
  - **NotaFinal**: média das notas do candidato.
  - **Opção**: vetor contendo a primeira, a segunda e a terceira opções de curso do candidato.

3.3. Exemplo: Vestibular

- **Problema:**
  - Distribuir os candidatos entre os cursos, segundo a nota final e as opções apresentadas por candidato.
  - Em caso de empate, os candidatos serão atendidos na ordem de inscrição para os exames.

3.3. Exemplo: Vestibular

- **Possível Solução:**
  - Ordenar registros pelo campo **NotaFinal**, respeitando a ordem de inscrição;
  - Percorrer cada conjunto de registros com mesma **NotaFinal**, começando pelo conjunto de **NotaFinal 10**, seguido pelo de **NotaFinal 9**, e assim por diante.
  - Para um conjunto de mesma **NotaFinal** tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções em que houver vaga (se houver).

3.3. Exemplo: Vestibular

- **Primeiro refinamento:**

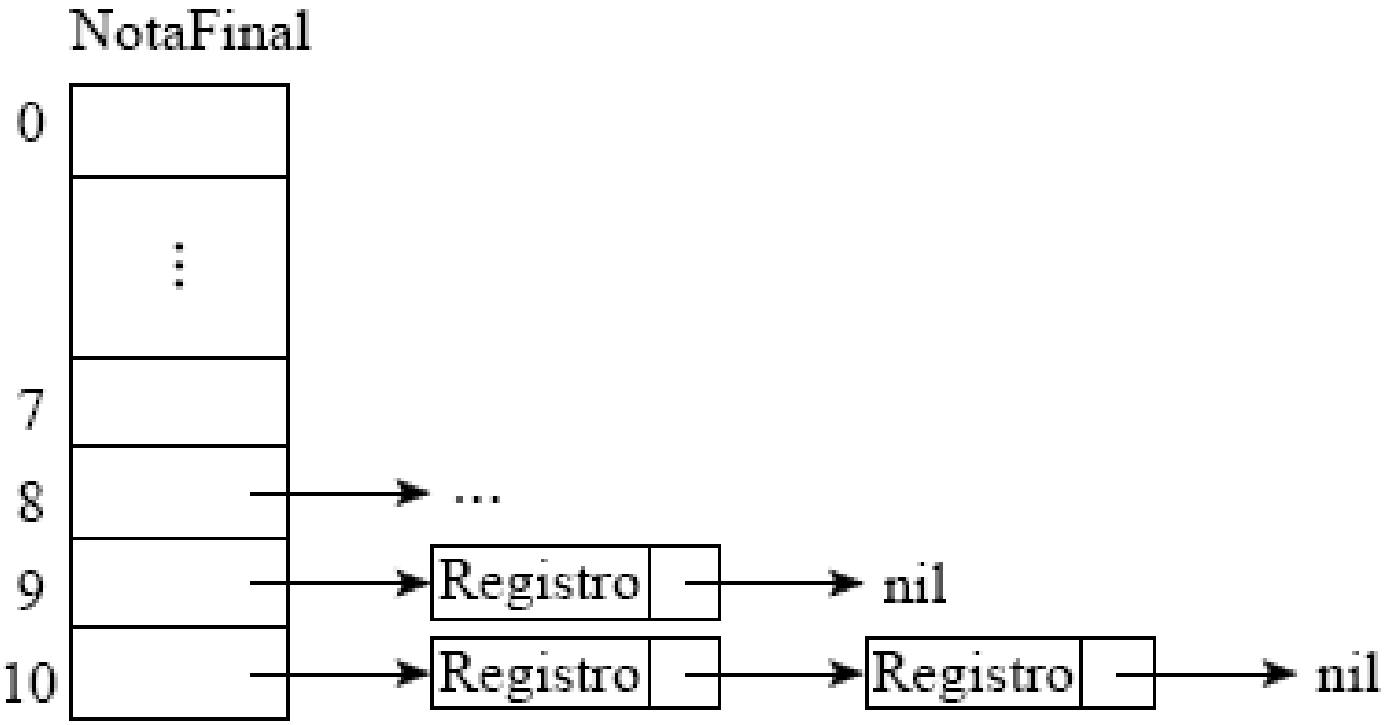
```
main() {
   ordena os registros pelo campo NotaFinal;
   for Nota = 10 até 0 do
      while houver registro com mesma nota do
         if existe vaga em um dos cursos de opcao do
            candidato
         then insere registro no conjunto de aprovados;
         else insere registro no conjunto de reprovados;
   imprime aprovados por curso;
   imprime reprovados;
}
```

3.3. Exemplo: Vestibular

- **Classificação dos Alunos:**
  - Uma boa maneira de representar um conjunto de registros é com o uso de listas.
  - Ao serem lidos, os registros são armazenados em listas para cada nota.
  - Após a leitura do último registro os candidatos estão automaticamente ordenados por **NotaFinal**.
  - Dentro de cada lista, os registros estão ordenados por ordem de inscrição, desde que os registros sejam lidos na ordem de inscrição de cada candidato e inseridos nesta ordem.

3.3. Exemplo: Vestibular

- **Representação da Classificação dos Alunos:**

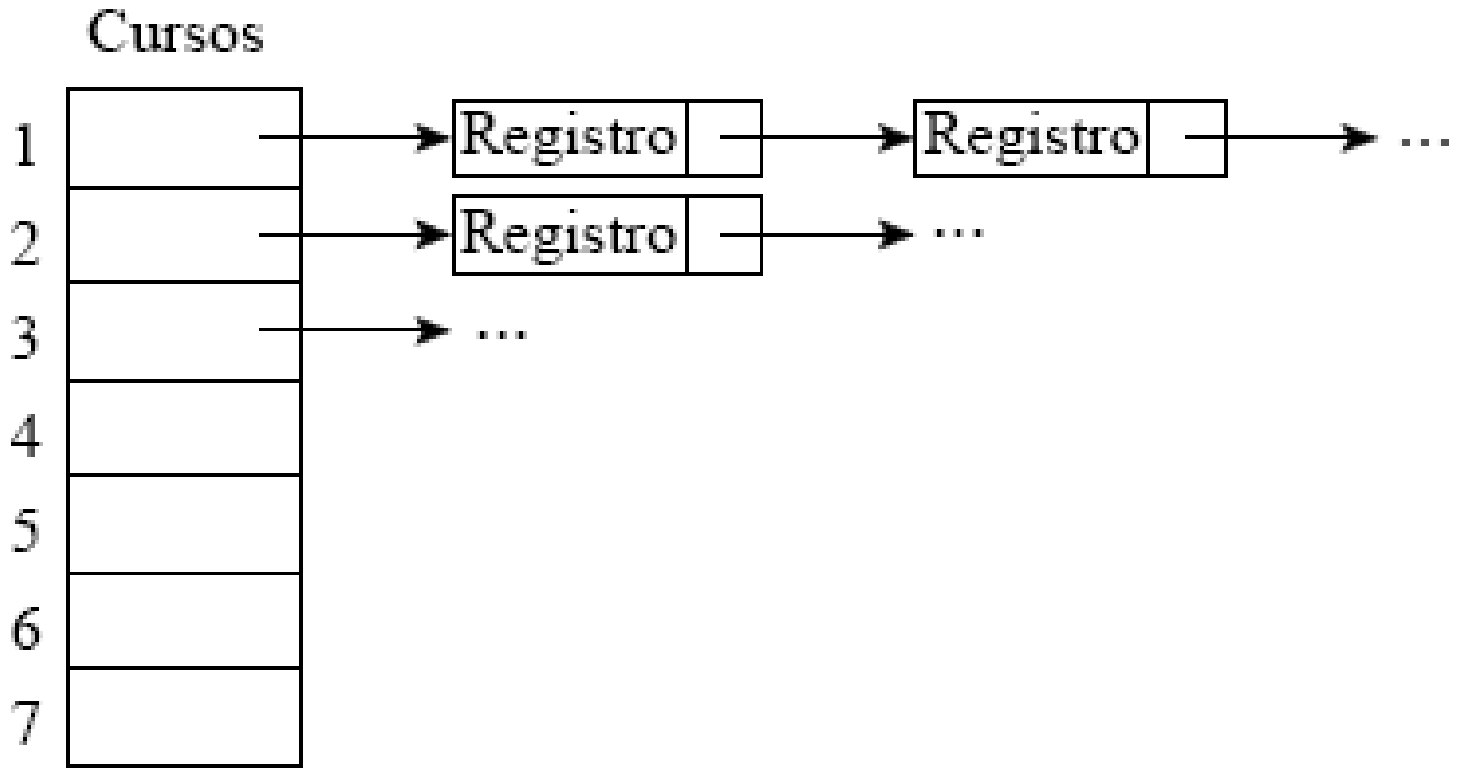


3.3. Exemplo: Vestibular

- **Classificação dos Alunos por Curso:**
  - As listas de registros são percorridas, iniciando-se pela de **NotaFinal 10**, seguida pela de **NotaFinal 9**, e assim sucessivamente.
  - Cada registro é retirado e colocado em uma das listas da abaixo, na primeira das três opções em que houver vaga.
  - Se não houver vaga, o registro é colocado em uma lista de reprovados.
  - Ao final a estrutura acima conterá a relação de candidatos aprovados em cada curso.

3.3. Exemplo: Vestibular

- **Classificação dos Alunos por Curso:**



3.3. Exemplo: Vestibular

- **Segundo refinamento:**

```
main() {
    lê número de vagas para cada curso;
    inicializa listas de classificação de aprovados e
       reprovados;
    lê registro;
    while Chave != 0 do //Ou while Chave do
       insere registro nas listas de classificação, conforme
          nota final;
       lê registro;
```

3.3. Exemplo: Vestibular

- **Segundo refinamento:**

```
for Nota = 10 até 0 do {
   while houver próximo registro com mesma NotaFinal do {
      retira registro da lista;
      if existe vaga em um dos cursos de opção do candidato {
         insere registro na lista de aprovados;
         decrementa o número de vagas para aquele curso; }
      else
         insere registro na lista de reprovados;
      obtém próximo registro;
   }
}
imprime aprovados por curso;
imprime reprovados;
}
```

3.3. Exemplo: Vestibular

- **Estrutura Final da Lista:**

```
#define NOPCOES 3
#define NCURSOS 7
#define FALSE 0
#define TRUE 1

typedef int TipoChave;

typedef struct {
   TipoChave Chave;
   int NotaFinal;
   int Opcao[NOpcoes];
}TipoItem;

typedef struct {
   TipoItem Item;
   struct TipoCelula* pProx;
}TipoCelula;
```

```
typedef struct {
   Celula *pPrimeiro, *pUltimo;
}TipoLista;

TipoItem Registro;
TipoLista Classificacao [11];
TipoLista Aprovados [NCURSOS];
TipoLista Reprovados;
int Vagas [NCURSOS];
int Passou;
int i, Nota;
```

3.3. Exemplo: Vestibular

- **Refinamento Final:**
  - Observe que o programa é completamente independente da implementação do tipo abstrato de dados Lista.

```
void LeRegistro (TipoItem *Registro) {
   /* os valores lidos devem estar separados por brancos */
   int i;
   int TEMP;
   scanf("%d %d", &(Registro->Chave), &TEMP);
   Registro->NotaFinal = TEMP;
   for (i=0; i < NOPCOES; i++) {
      scanf("%d", &TEMP);
      Registro->Opcao[i] = TEMP;
   }
}
```

3.3. Exemplo: Vestibular

- **Refinamento Final:**

```
int main () {
   /* inicializacao */
   for (i = 1; i <= NCursos; i++)
      scanf("%d", &Vagas[i-1]);
   for (i = 0; i <= 10; i++)
      FLVazia(&(Classificacao[i]));
   for (i = 0; i < NCursos; i++)
      FLVazia(&(Aprovados[i]));
   FLVazia(&Reprovados);

   /* leitura dos registros */
   LeRegistro(&Registro);
   while (Registro.Chave != 0) {
      Linsere(&Classificacao[Registro.NotaFinal],&Registro);
      LeRegistro(&Registro);
   }
```

3.3. Exemplo: Vestibular

- **Refinamento Final:**

```
   for (Nota = 10; Nota >= 0; Nota--)
      while (!LVazia(&Classificacao[Nota])) {
         LRetira(&Classificacao[Nota], &Registro);
         Passou = FALSE;
         for (i = 0; i < NOpcoes && !Passou; i++)
            if(Vagas[ Registro.Opcao[i]-1 ] > 0) {
               Linsere(&(Aprovados[ Registro.Opcao[i]-1 ]), &Registro );
               Vagas[ Registro.Opcao[i]-1 ]--;
               Passou = TRUE;
            }
         if (!Passou)
            LInsere(&Reprovados, &Registro);
      }
   for (i = 0; i < NCursos; i++) {
      printf("Relacao dos aprovados no Curso %d\n", i+1);
      Imprime(Aprovados[i]);
   }
   printf("Relacao dos reprovados\n");
   Imprime(Reprovados);
   return 0;
}
```

3.3. Exemplo: Vestibular

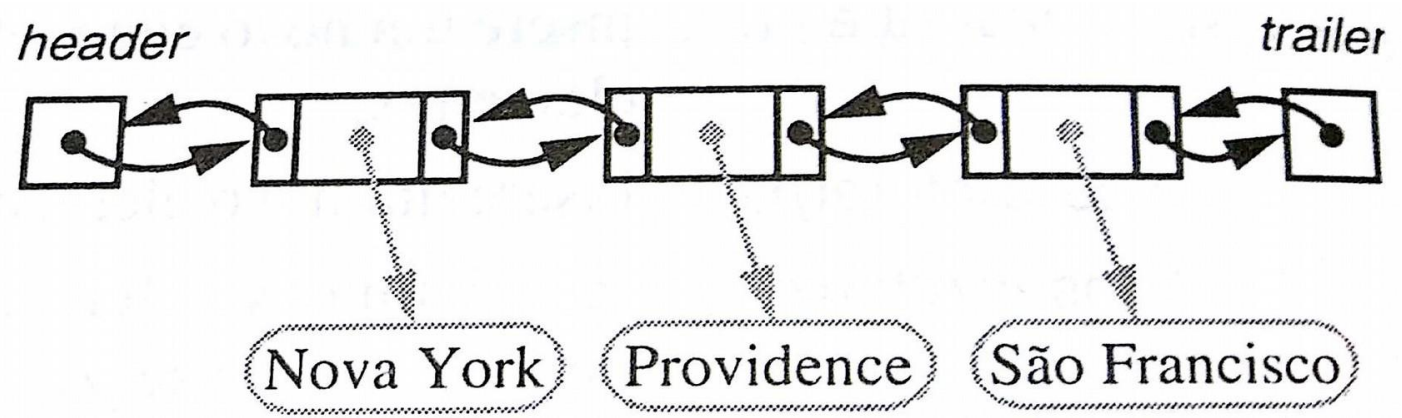
- O exemplo mostra a importância de utilizar tipos abstratos de dados para escrever programas, em vez de utilizar detalhes particulares de implementação.

- Altera-se a implementação rapidamente.
  - Não é necessário procurar as referências diretas às estruturas de dados por todo o código.

- Este aspecto é particularmente importante em programas de grande porte.

4. Listas Duplamente Encadeadas

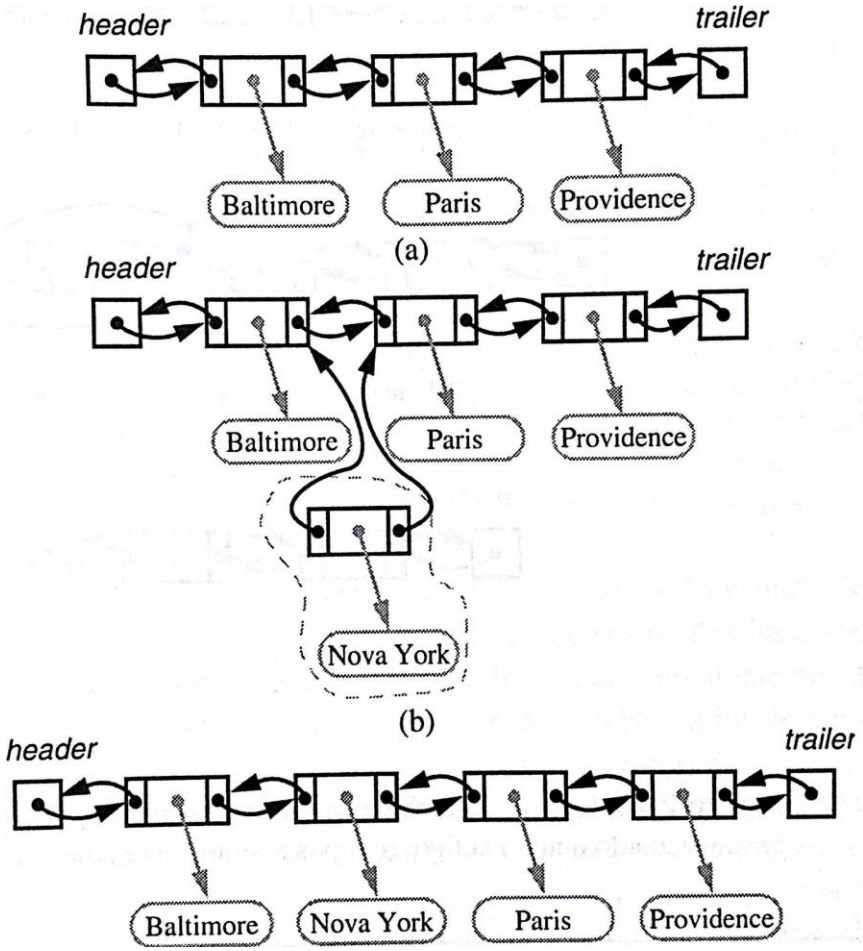
- Quando é necessário navegar pela lista de trás pra frente, ou encontrar elementos anteriores, usa-se o **encadeamento duplo**:
  - Listas com ponteiros para o **próximo** e **anterior**.



4. Listas Duplamente Encadeadas

- **Inserção de elemento:**

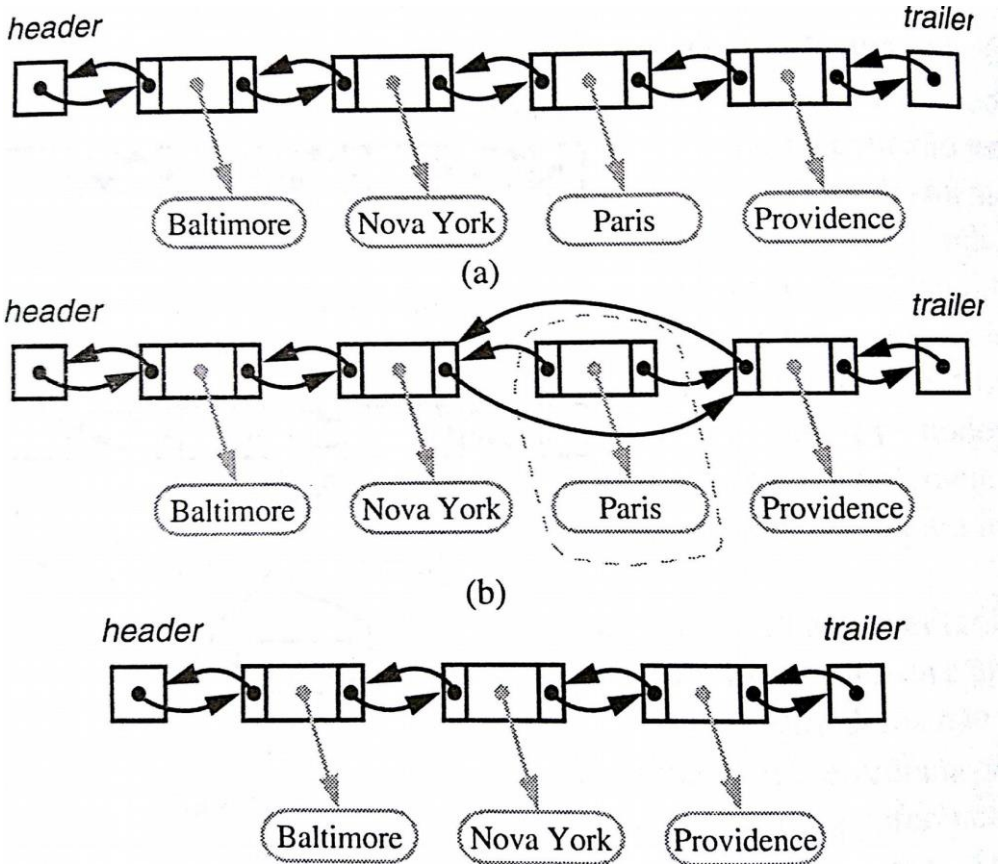
```
Algoritmo insertAfter (p,e):
    Cria um novo nodo v
    v.element ← e
    v.prev ← p
    v.next ← p.next
    (p.next).prev ← v
    p.next ← v
    retorne v
```



4. Listas Duplamente Encadeadas

- **Remoção de elemento:**

```
Algoritmo remove(p):
    t ← p.element
    (p.prev).next ← p.next
    (p.next).prev ← p.prev
    libere p
    retorne t
```



5. Referências

- Material de aula dos Profs. Luiz Chaimowicz e Raquel O. Prates, da UFMG: https://homepages.dcc.ufmg.br/~glpappa/aeds2/AEDS2.1%20Conceitos%20Basicos%20TAD.pdf
- DEITEL, P; DEITEL, H. *C How to Program*. 6a Ed. Pearson, 2010.
- LANGSAM,Y.; AUGENSTEIN, M.J.; TENENBAUM, A.M. Data Structures using C and C++, 2a edição . Prentice Hall of India. 2007.
- CORMEM, T. H.; et al. Introduction to algorithms, 3a edição, The MIT Press.
- DROZDEK A. Estrutura de dados e algoritmos em C++,1a edição Cengage Learning.