Estrutura de Dados Lista

Prof. Joaquim Uchôa
Profa. Juliana Greghi
Prof. Renato Ramos

- Conceitos iniciais
- Visão geral de listas
- Métodos usuais em listas (partes i, ii e iii)
- Exemplos adicionais

Conceitos iniciais

Variáveis homogêneas

Variáveis homogêneas podem ser definidas como um conjunto sequencial de variáveis de um mesmo tipo.

A maioria das linguagens de programação implementam variáveis homogêneas como arranjos (vetores) ou listas.

Variáveis homogêneas

Existe ainda a possibilidade de implementação por meio de vetores associativos (mapas ou dicionários), como em Python. Um vetor associativo geralmente é implementado por meio de tabelas hash.

Arranjos - I

Um arranjo é um conjunto de locais para armazenamento de elementos de um mesmo tipo.

- Estes elementos podem ser selecionados de acordo com sua posição no arranjo.
- Pode ser enxergado como uma variável com várias posições.

2	-3	101	104	0	1	2	3	-3	0

Arranjos - II

O uso de arranjos tem como principal vantagem o acesso rápido a qualquer uma de suas posições.

Como principal desvantagem encontra-se o fato que ele não permite redimensionamento de forma fácil e prática, possuindo tamanho limitado.

Listas - I

Uma lista encadeada (ou linear) é uma coleção ordenada de elementos de um mesmo tipo. É uma estrutura de dados linear e dinâmica.

Coleção ordenada: dado um elemento da coleção podemos identificar seu sucessor e seu predecessor. O primeiro e o último elementos da lista são considerados elementos especiais.

Listas - II

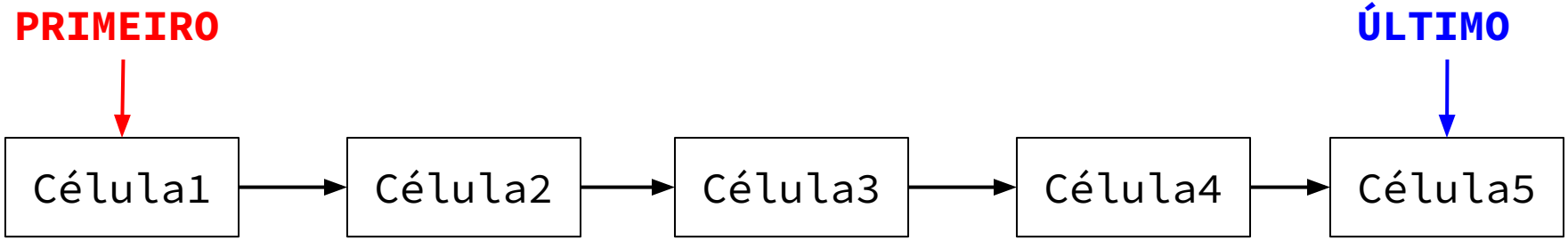
Uma lista encadeada é composta por células que apontam para o próximo elemento da lista.

Uma lista encadeada é criada a partir de seu primeiro elemento e seu último elemento aponta para uma célula nula. Ex:

Célula 1 --> Célula 2 --> Célula 3 --> Célula 4 --> Célula 5 --> (Nulo)

Listas - III

Uma lista geralmente possui indicadores que apontam para o primeiro (obrigatório) e último elemento (para facilitar inserção).



Listas - IV

O uso de listas tem como principal vantagem a possibilidade de redimensionamento, caso não haja limite de memória.

Como principal desvantagem encontra-se o fato que ele não permite acesso ágil aos elementos intermediários, uma vez que, em geral, precisa percorrer os nós, um por um até chegar ao desejado.

Listas - V

Em geral, listas encadeadas são implementadas por meio de indireção (ponteiros ou similares), apesar de existirem alternativas.

Por conta do tamanho dinâmico e do uso de ponteiros (incluindo alocação dinâmica de memória), listas encadeadas também são chamadas de **listas dinamicamente encadeadas**.

Pilhas, Filas e Listas

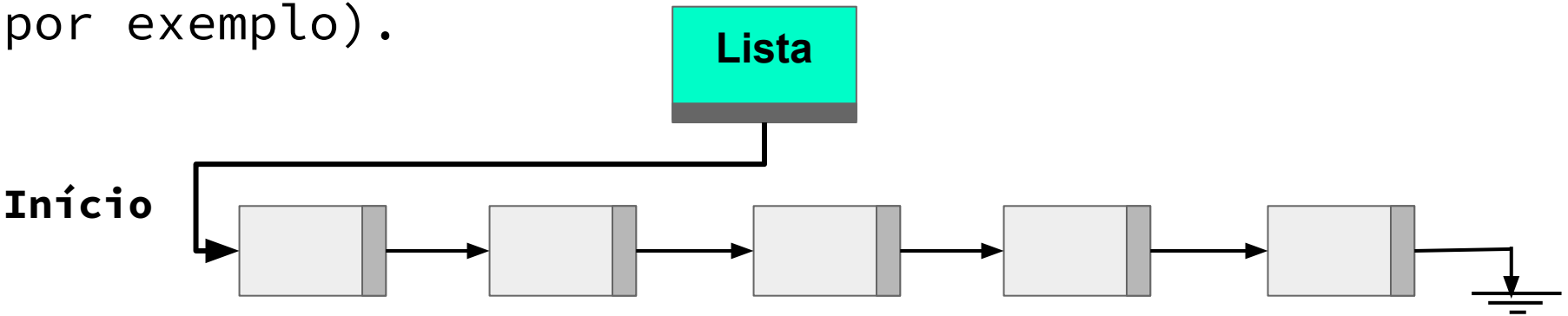
É fácil perceber que pilhas e filas são tipos especiais de listas, que controlam o acesso aos elementos a partir de posições específicas. Acesso a elementos intermediários não são permitidos nessas duas estruturas.

Em uma lista, por outro lado, é possível acessar elementos intermediários sem que haja essa quebra de estrutura. Esse acesso é, inclusive, função básica da lista.

Visão geral de implementação de listas

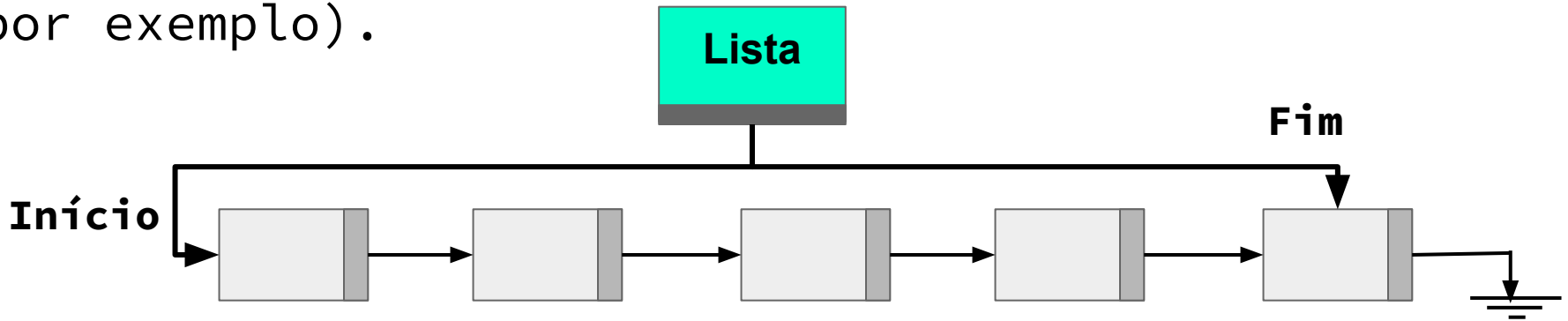
Implementação de listas: visão geral - I

Uma lista possui ao menos um ponteiro, que aponta para seu primeiro elemento. Boa parte das implementações também possui um ponteiro para seu último elemento, para facilitar o acesso à essa posição (em inserção ou remoção no fim da lista, por exemplo).



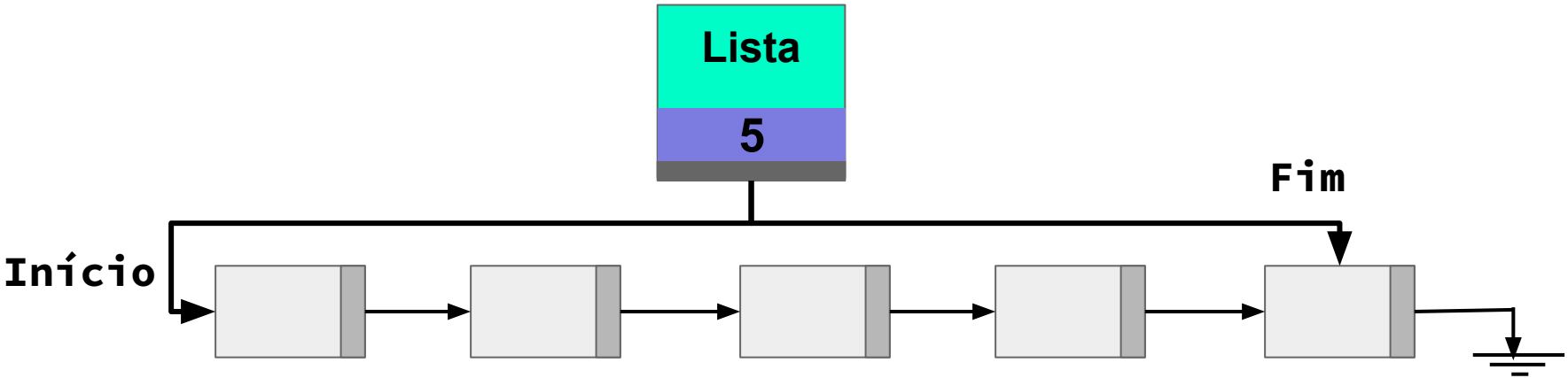
Implementação de listas: visão geral - I

Uma lista possui ao menos um ponteiro, que aponta para seu primeiro elemento. Boa parte das implementações também possui um ponteiro para seu último elemento, para facilitar o acesso à essa posição (em inserção ou remoção no fim da lista, por exemplo).

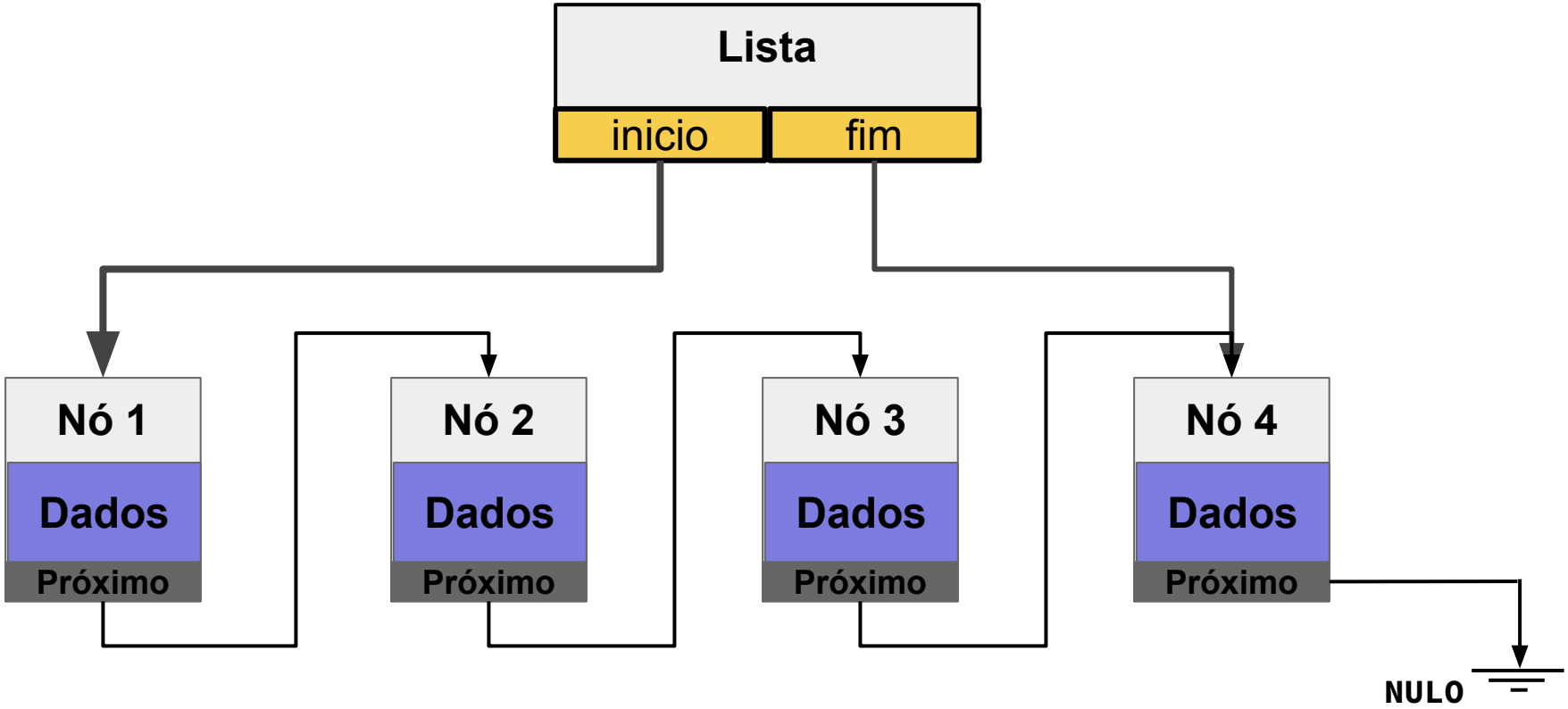


Implementação de listas: visão geral - I

Um atributo opcional, bastante utilizado em implementações é o tamanho da lista. Sem esse atributo, para saber o seu tamanho é necessário percorrê-la por completo.

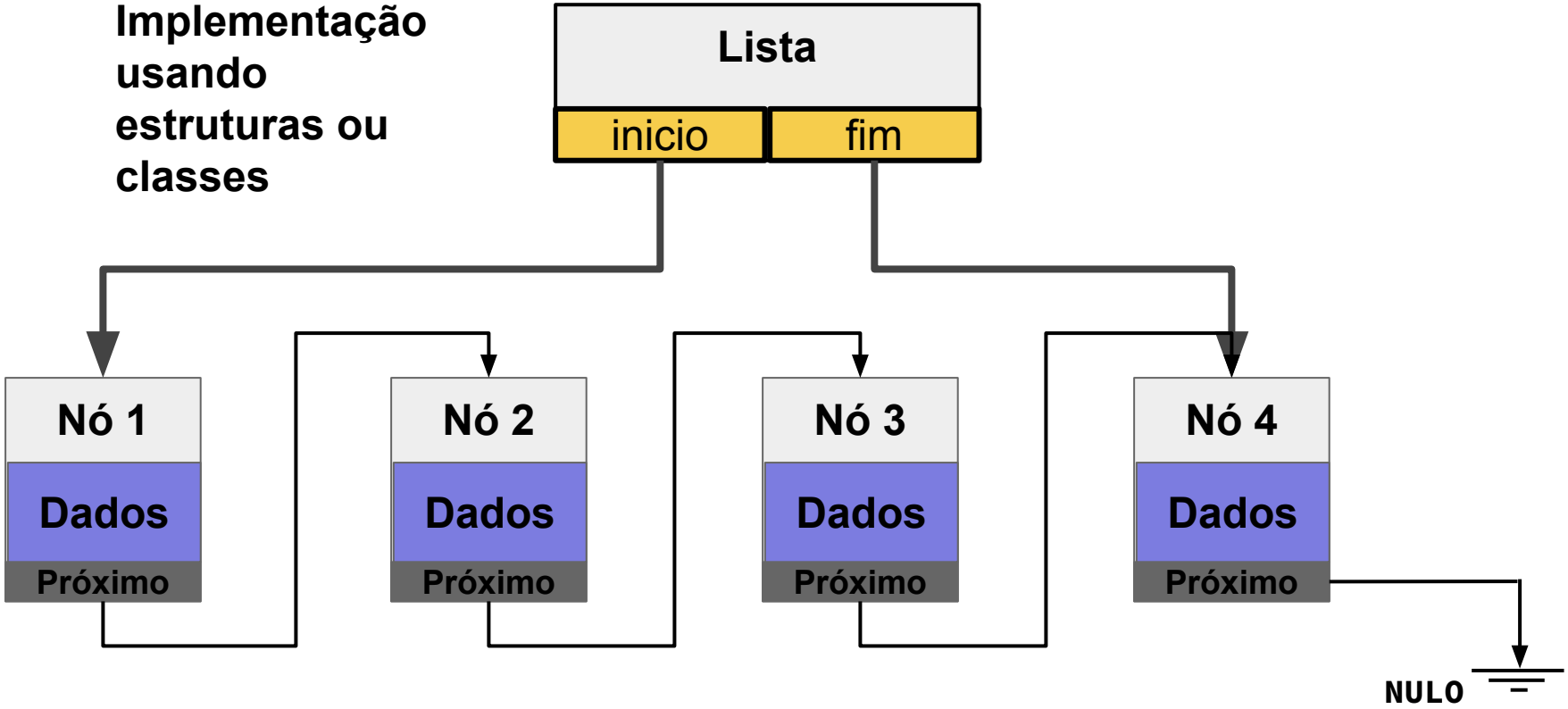


Implementação tradicional de listas



Implementação tradicional de listas

Implementação usando estruturas ou classes

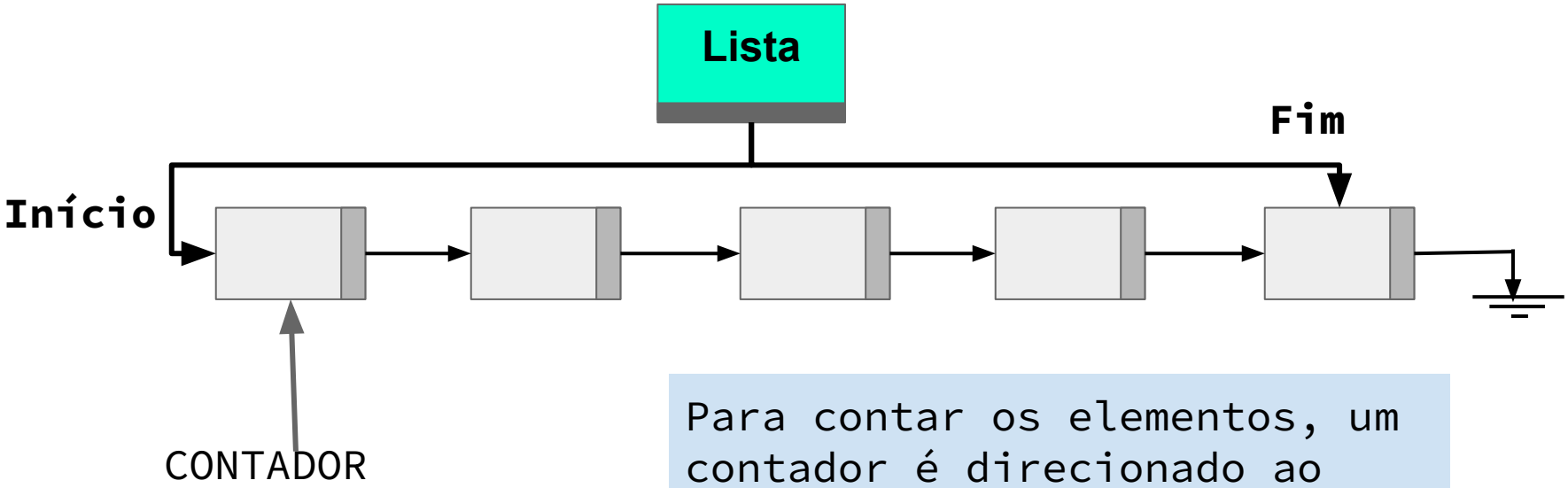


Implementação de listas: visão geral - II

Sem um ponteiro para o último elemento da lista, inserções no final da lista precisam percorrê-la por inteiro antes de efetuar essa tarefa. Caso essa não seja uma operação comum, pode-se evitar esse atributo e o custo de sua manutenção.

A ausência de um atributo para o tamanho torna obrigatório percorrer a lista para descobrir seu tamanho.

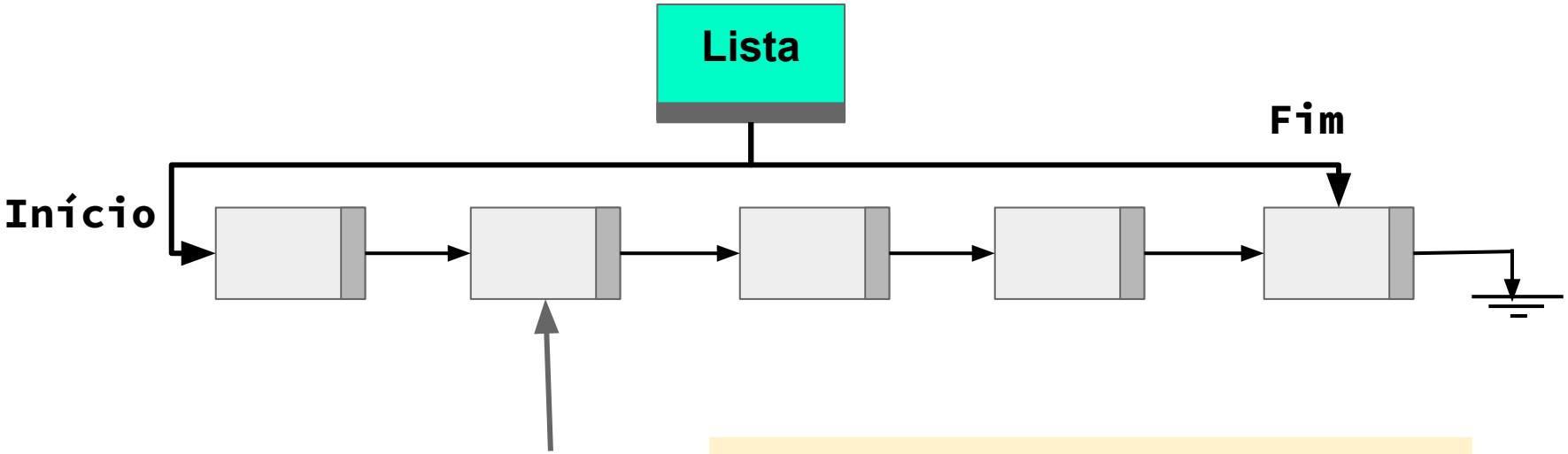
Implementação de listas: Contando elementos



Para contar os elementos, um contador é direcionado ao primeiro elemento da lista:
CONTADOR ← Início

TAMANHO = 1

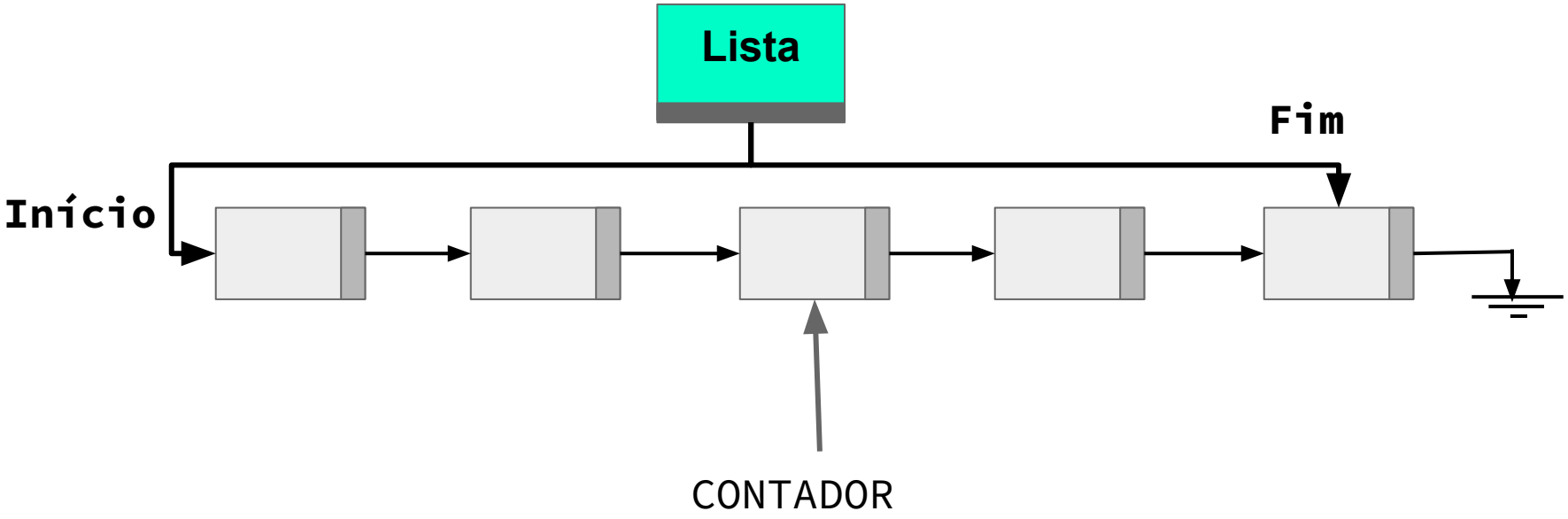
Implementação de listas: Contando elementos



TAMANHO = 2

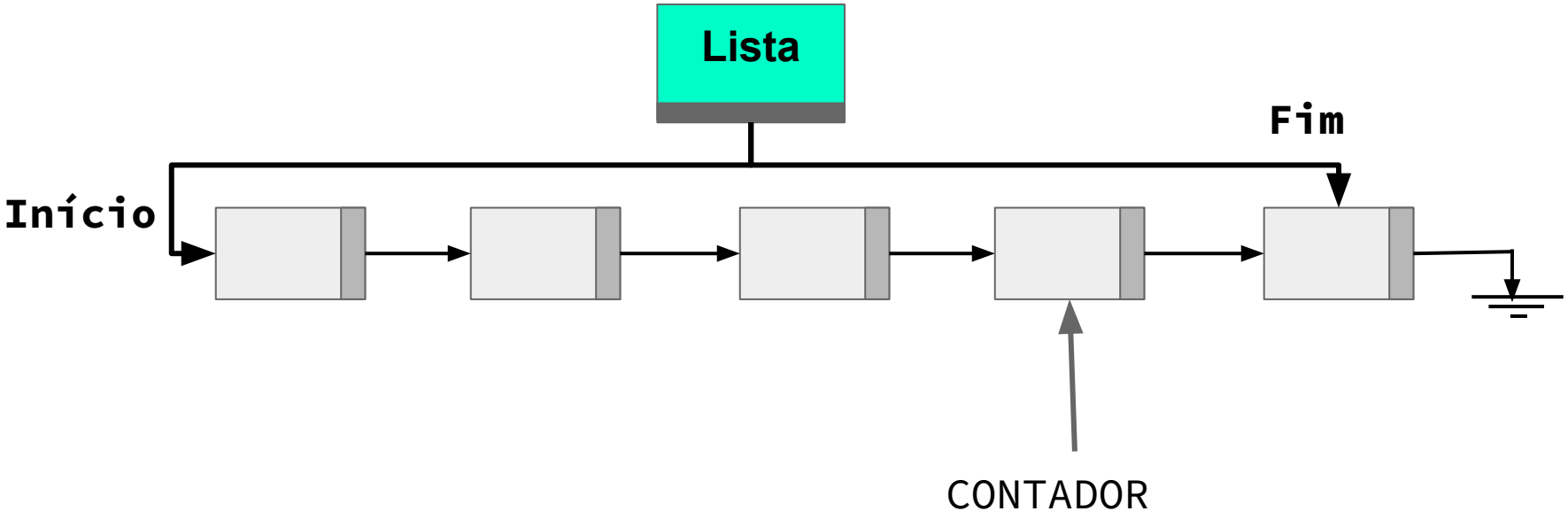
Para avançar, CONTADOR passa a apontar para seu próximo:
CONTADOR ← CONTADOR.próximo

Implementação de listas: Contando elementos



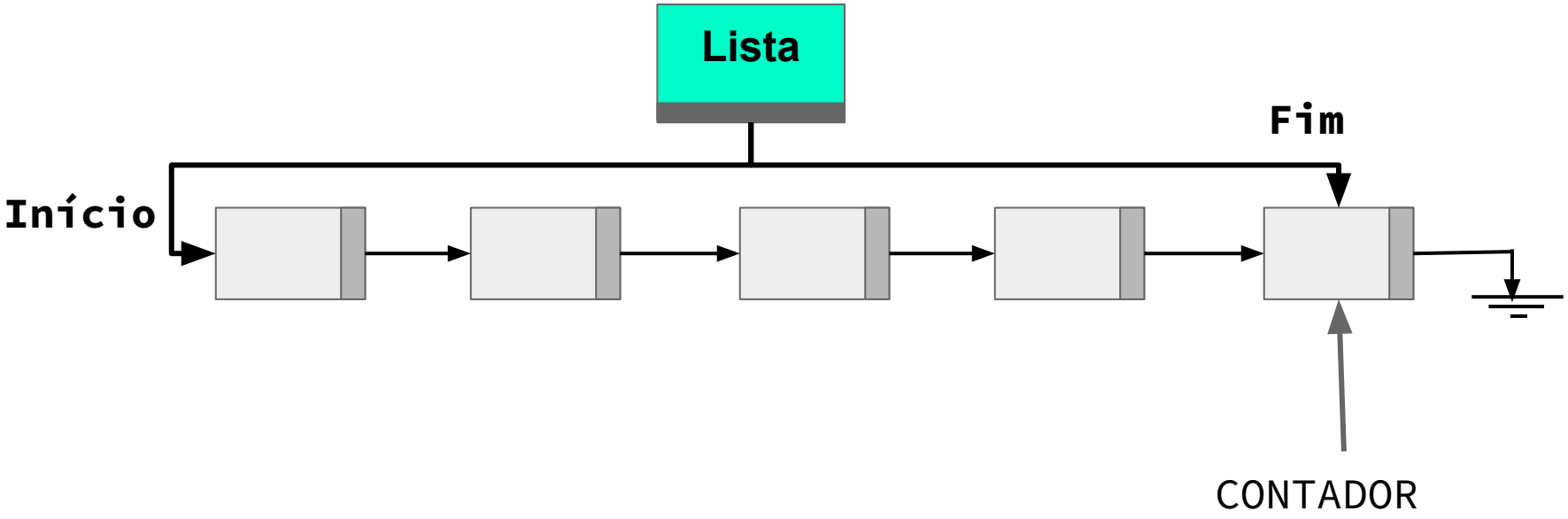
TAMANHO = 3

Implementação de listas: Contando elementos



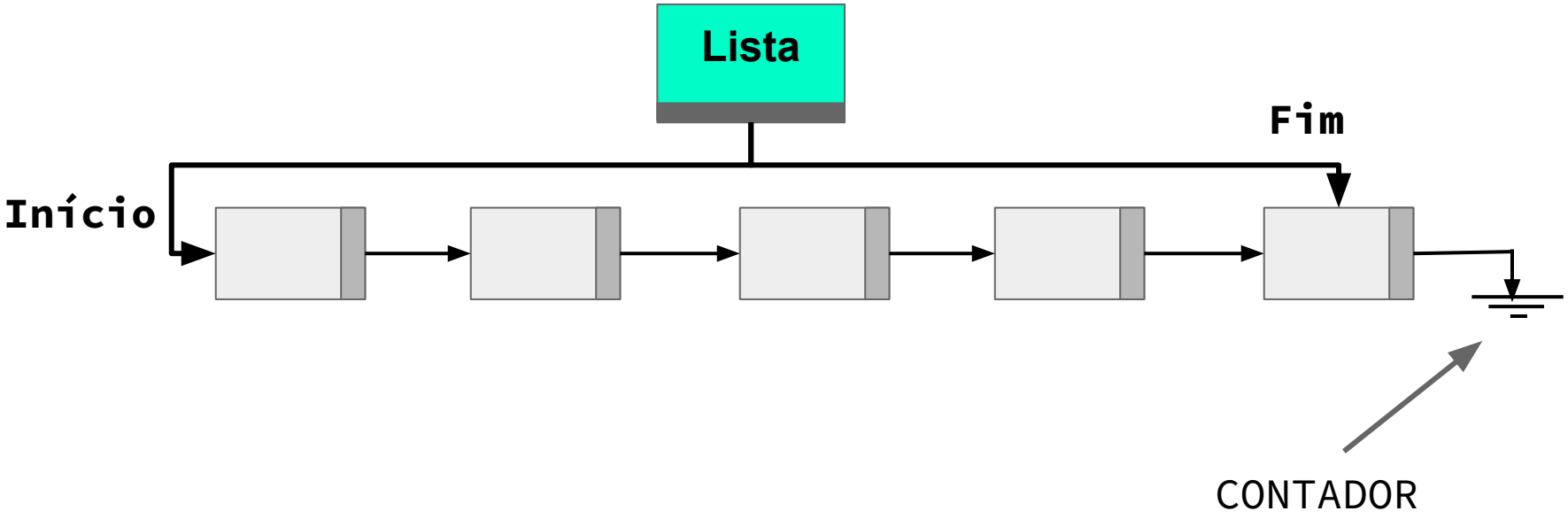
TAMANHO = 4

Implementação de listas: Contando elementos



TAMANHO = 5

Implementação de listas: Contando elementos



TAMANHO = 5

Implementação de listas: visão geral - III

Vários métodos podem ser implementados de maneira recursiva, como a busca, por exemplo: *um nó procura por um valor verificando se não contém o valor ou pedindo ao próximo para fazer o mesmo.*

Maior parte das implementações utiliza métodos iterativos.

Implementação de listas: visão geral - IV

Como nós costumam ser privados, métodos que os utilizam costumam ser auxiliares e privados ou movidos para uma classe que representa os nós da lista.

Algumas listas disponibilizam métodos que retornam iteradores, utilizados para acessar elementos específicos da lista a partir de uma dada posição.

Nós: visão geral

Geralmente os nós armazenam dados (estruturas, por exemplo) com as informações da lista.

Em parte das implementações, é implementado como *estrutura*, mesmo quando se utiliza orientação a objetos.

Dependendo da implementação, métodos auxiliares da lista são implementados diretamente em uma *classe* nó.

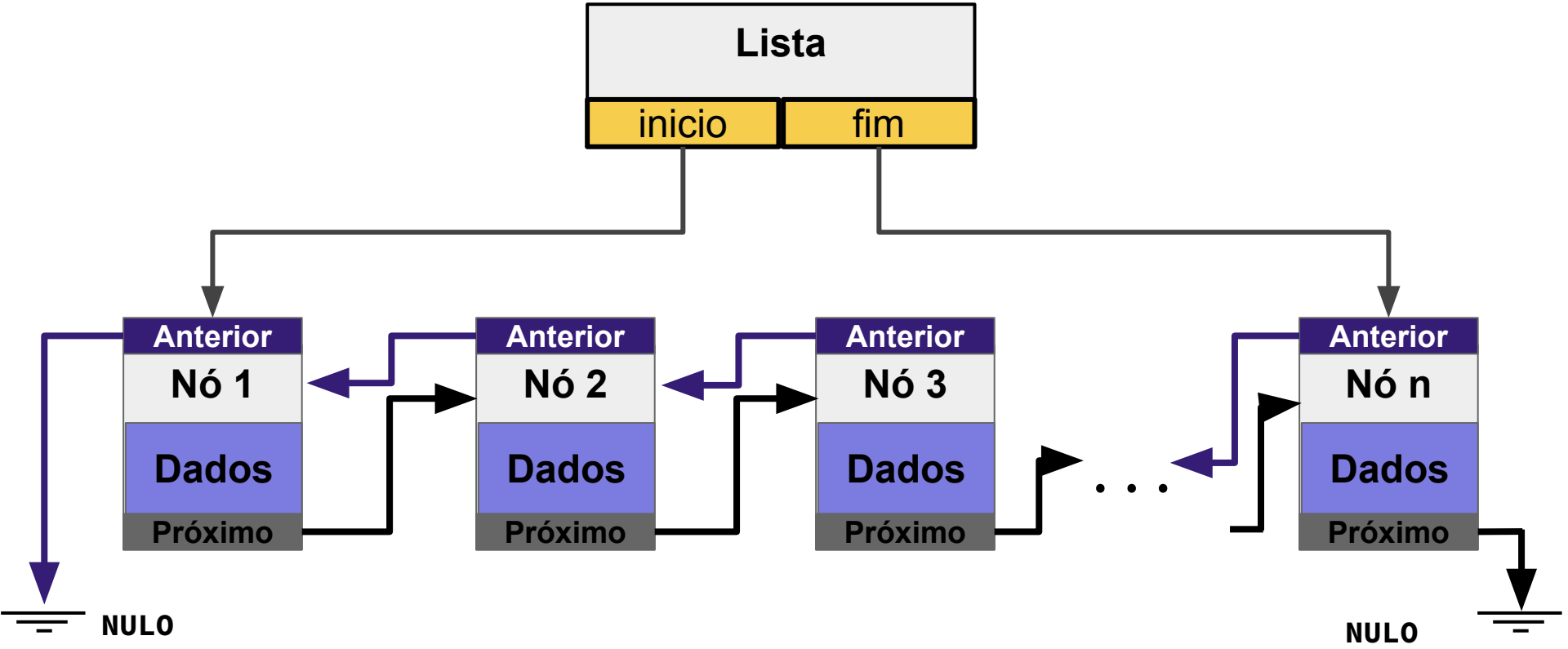
Listas duplamente encadeadas - I

Em várias situações, é interessante percorrer uma lista não apenas em uma direção.

Também pode ocorrer de, a partir de um dado nó, querer retornar ao seu anterior na lista.

Assim, parte das implementações utiliza dois encadeamentos: para o próximo e o anterior.

Listas duplamente encadeadas - II



Listas estaticamente encadeadas - I

Apesar que a implementação tradicional de listas é utilizando indireção, é possível utilizar um arranjo para armazenar os dados de uma lista.

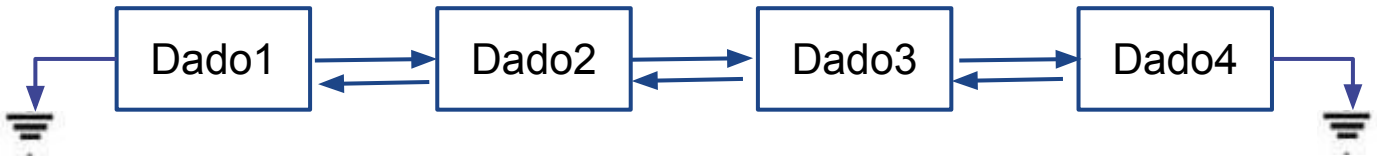
Nesse caso, utiliza-se um vetor de nós, em que cada nó armazena, por exemplo, além de seus dados, a posição do nó anterior e do próximo.

Listas estaticamente encadeadas - II

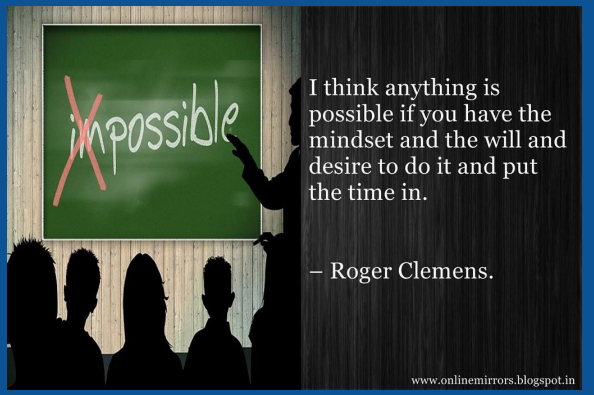
Por exemplo, o arranjo a seguir:

Célula 0		Célula 1		Célula 2		Célula 3		Célula 4	
Dado1	3	NULO	-1	Dado3	4	Dado2	2	Dado4	-1
	-1		-1		3		0		2

Representa a lista:



Métodos Usuais em Listas Parte I



Operações em Listas

Uma lista, em geral, costuma disponibilizar uma série de métodos similares aos existentes em um vetor, como o percorrimento dos dados, por exemplo.

Por outro lado, existem métodos que, mesmo quando são disponibilizados, devem ser evitados, como o acesso direto a uma posição. Em um vetor isso é feito sem custo, em uma lista, é necessário percorrer os nós anteriores.

Lista: métodos usuais - I

Construção: inicializa os dados e aponta os ponteiros primeiro e último para nulo.

Destruição: finaliza a lista, liberando memória que tenha sido alocada (percorre a lista apagando os nós).

Caminhamento/Percorrimento: percorre os nós da lista, um por um, realizando alguma ação.

Lista: métodos usuais - II

Busca: percorre a lista buscando um nó com dado valor.

Acessa posição: percorre os nós da lista, um por um, até uma determinada posição.

Lista: métodos usuais - III

Inserção: possibilita a inserção de um novo nó na lista. Em geral, existem três opções de inserção:

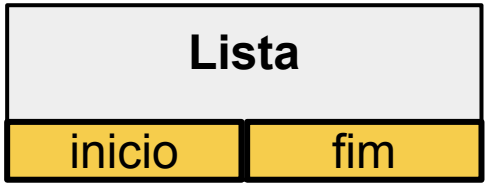
- inserção no início da lista
- inserção no final da lista
- inserção em uma posição determinada da lista (precisa percorrer a lista até chegar à posição desejada)

Lista: métodos usuais - IV

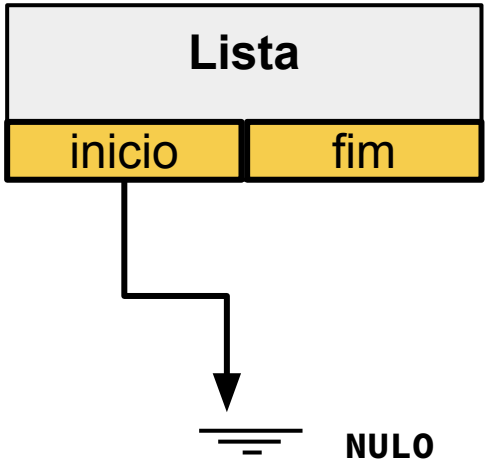
Remoção: possibilita a remoção de um dado nó na lista. Em geral, existem três opções de remoção:

- remoção do início ou fim da lista
- remoção de um nó com um dado valor (é necessário encontrá-lo antes)
- remoção de um nó em uma dada posição da lista (precisa percorrer a lista até chegar à posição desejada)

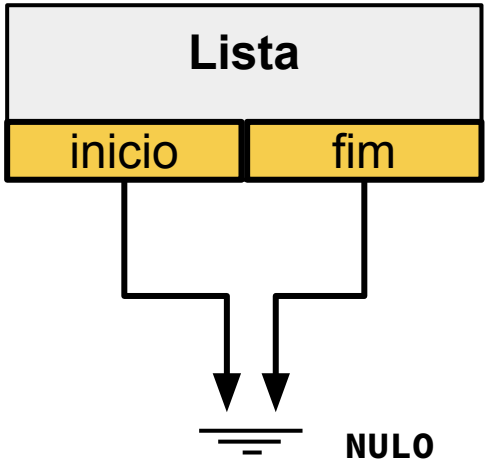
Criação da lista - I



Criação da lista - I



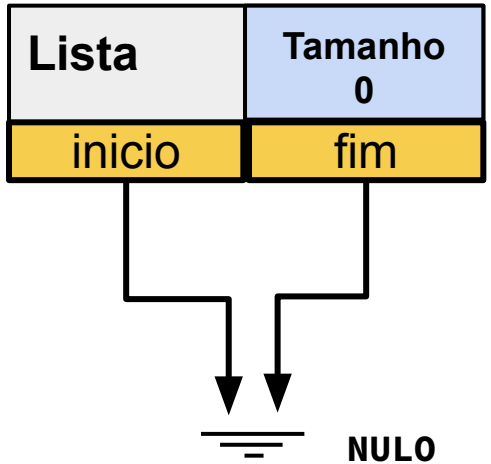
Criação da lista - I



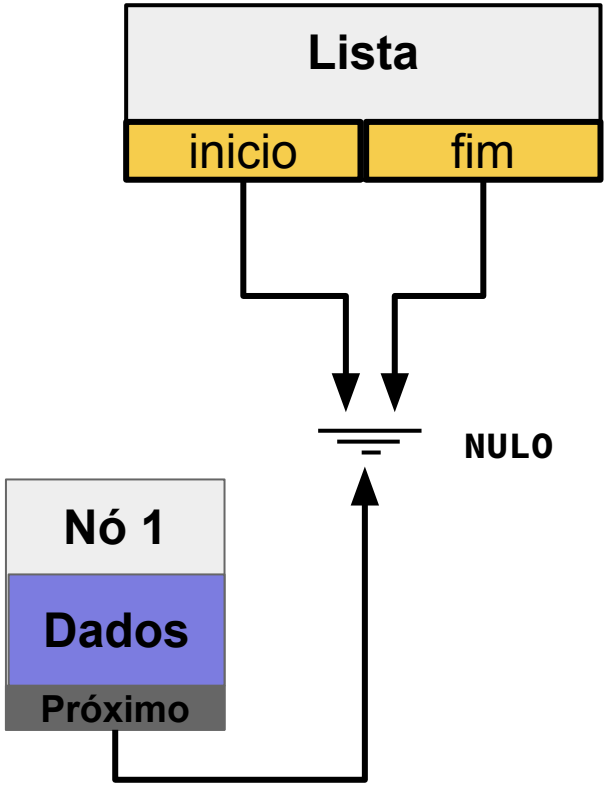
Criação da lista - pseudocódigo

criarLista():

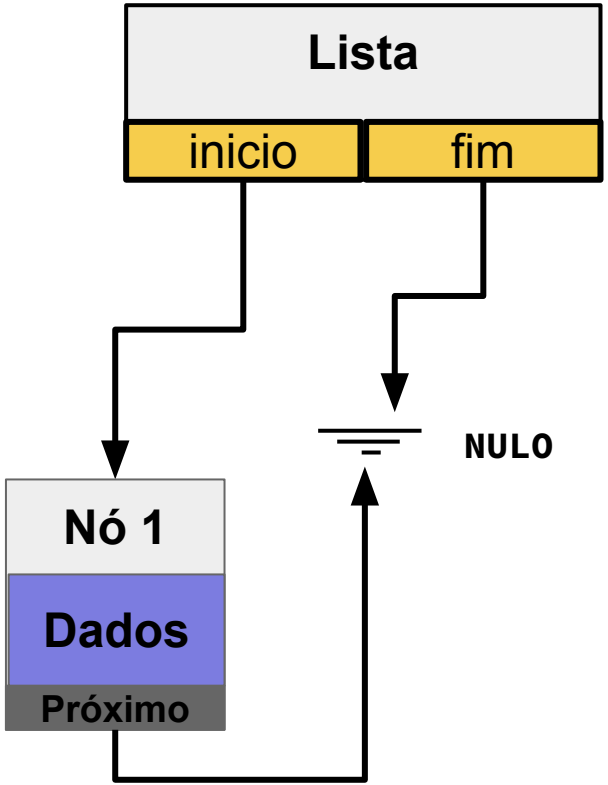
```
inicio ← NULO;
fim ← NULO;

// caso seja interessante armazenar
// o tamanho da lista
tamanho ← 0;
```



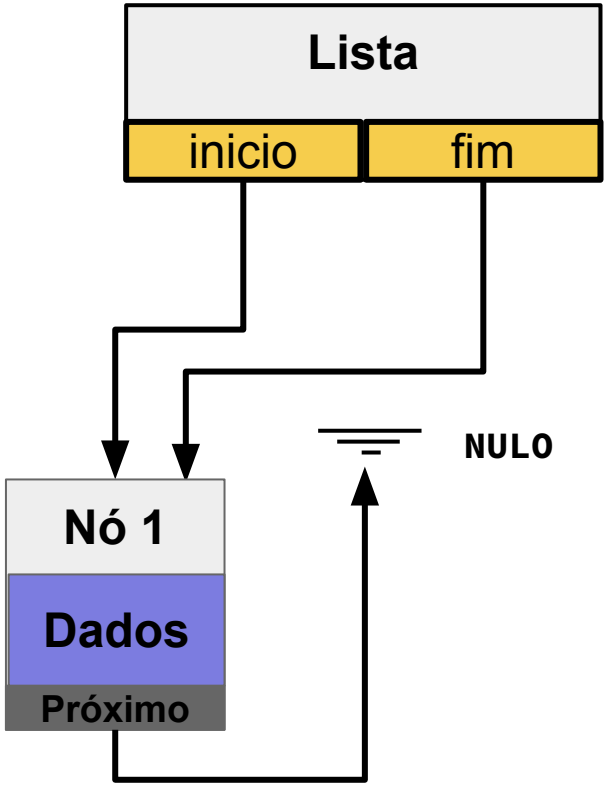
Inserção em lista vazia - I



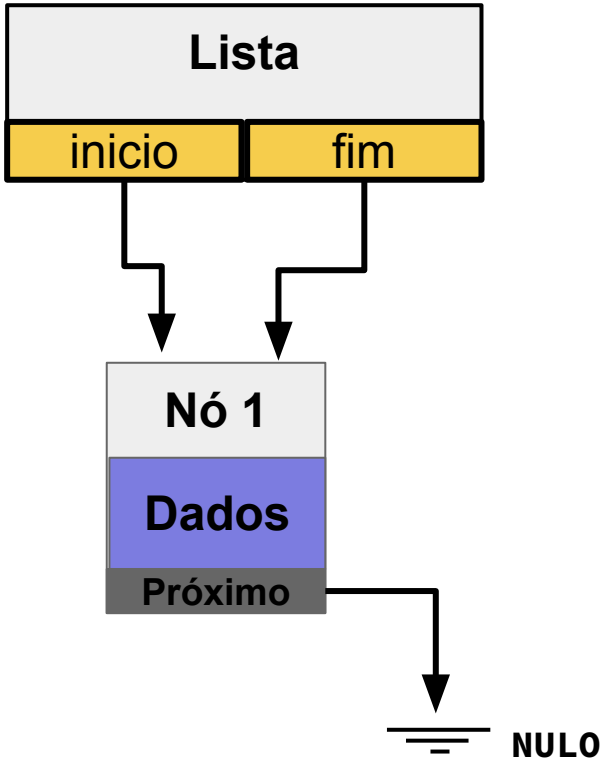
Inserção em lista vazia - I



Inserção em lista vazia - I



Inserção em lista vazia - I



Inserção em lista vazia - Pseudocódigo

inserirEmListaVazia(valor):

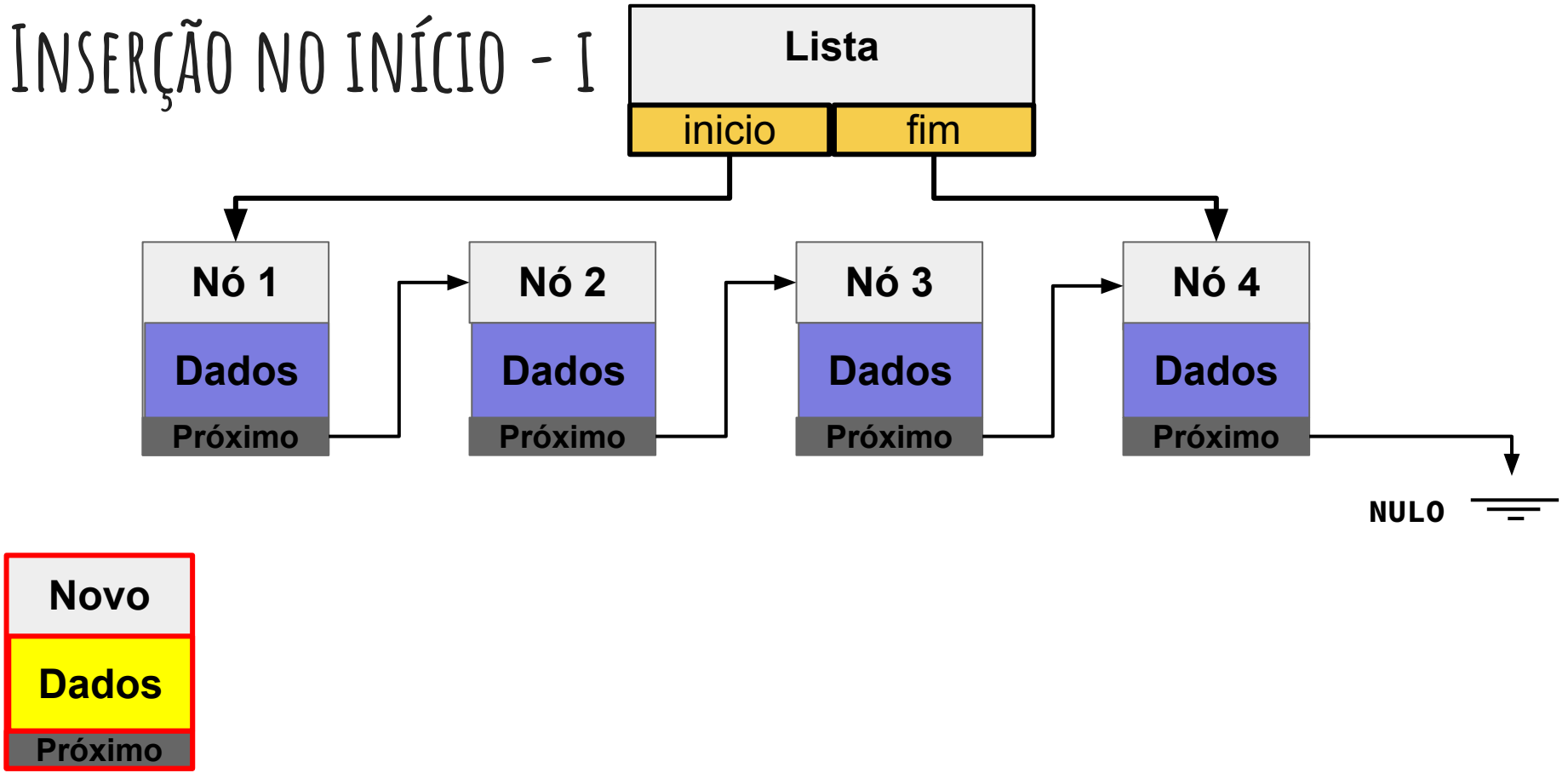
```
novo ← criar_noh(valor); // cria um nó com o valor
inicio ← novo;
fim ← novo;

// caso seja interessante armazenar
// o tamanho da lista
tamanho ← 1;
```

Inserção em lista vazia - Observações

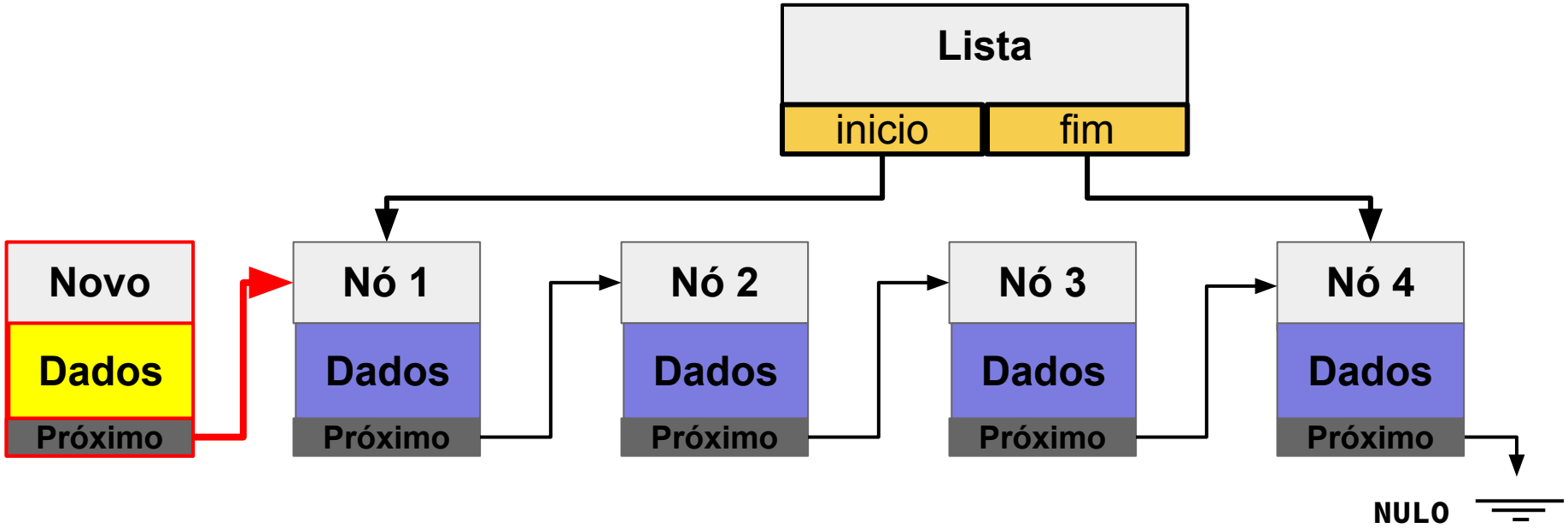
Um método para inserção em lista vazia ou não é implementada ou é inserida como método auxiliar, não invocado pelo usuário.

Em geral, se utilizam métodos mais gerais, como inserir no início, no fim ou em posição específica.

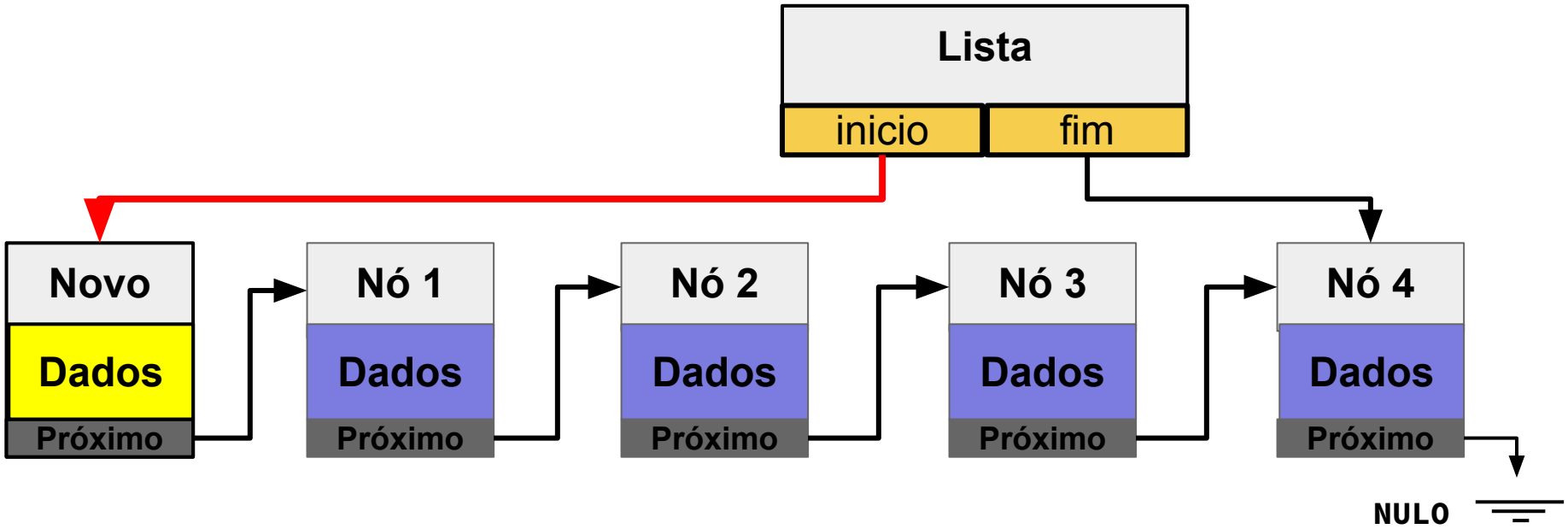
Inserção no início - I



Inserção no início - II



Inserção no início - II



Inserção no início - Observação

O método de inserção no início pode ser chamado em uma lista vazia.

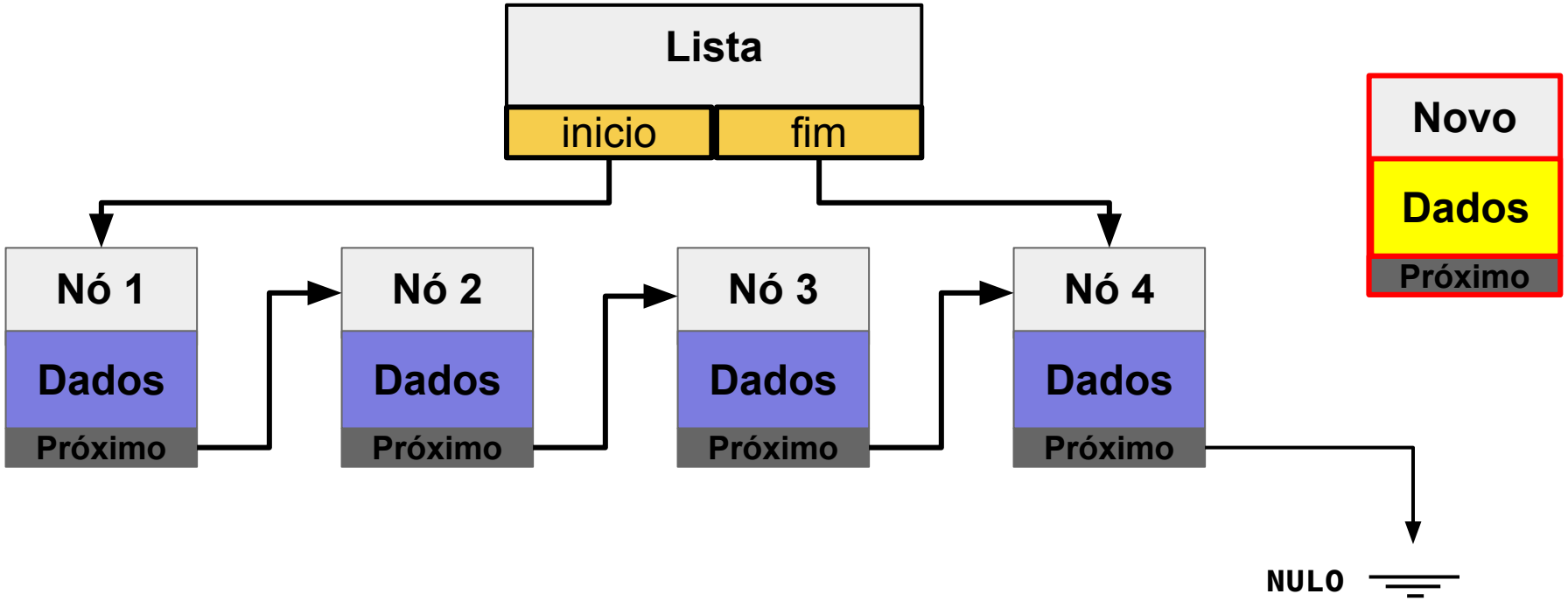
Assim, é necessário verificar e tratar essa situação.

Inserção no início - Pseudocódigo

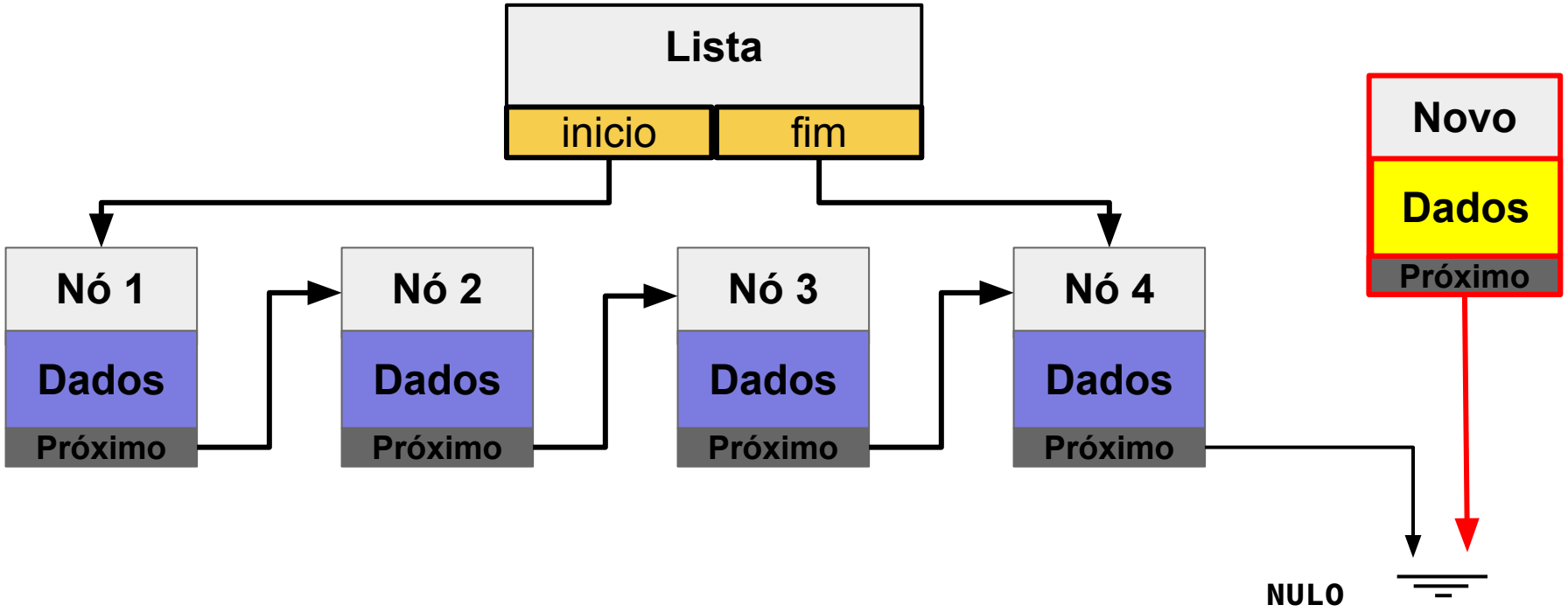
inserirNoInicioDaLista(valor):

```
se (lista.vazia()) {
    inserirEmListaVazia(valor);
} senão {
    novo ← criar_noh(valor); // cria um nó com o valor
    novo.proximo ← inicio;
    inicio ← novo;
    tamanho++; // caso seja interessante o atributo
}
```

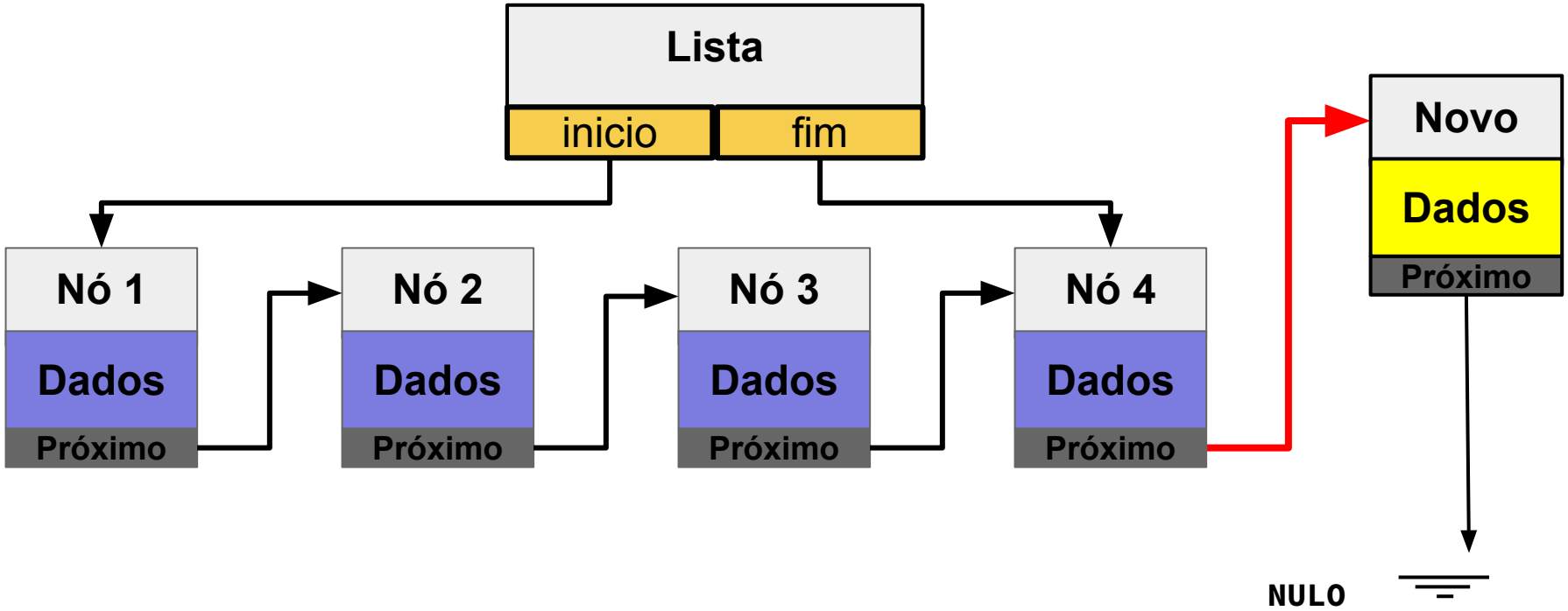
Inserção no fim - I



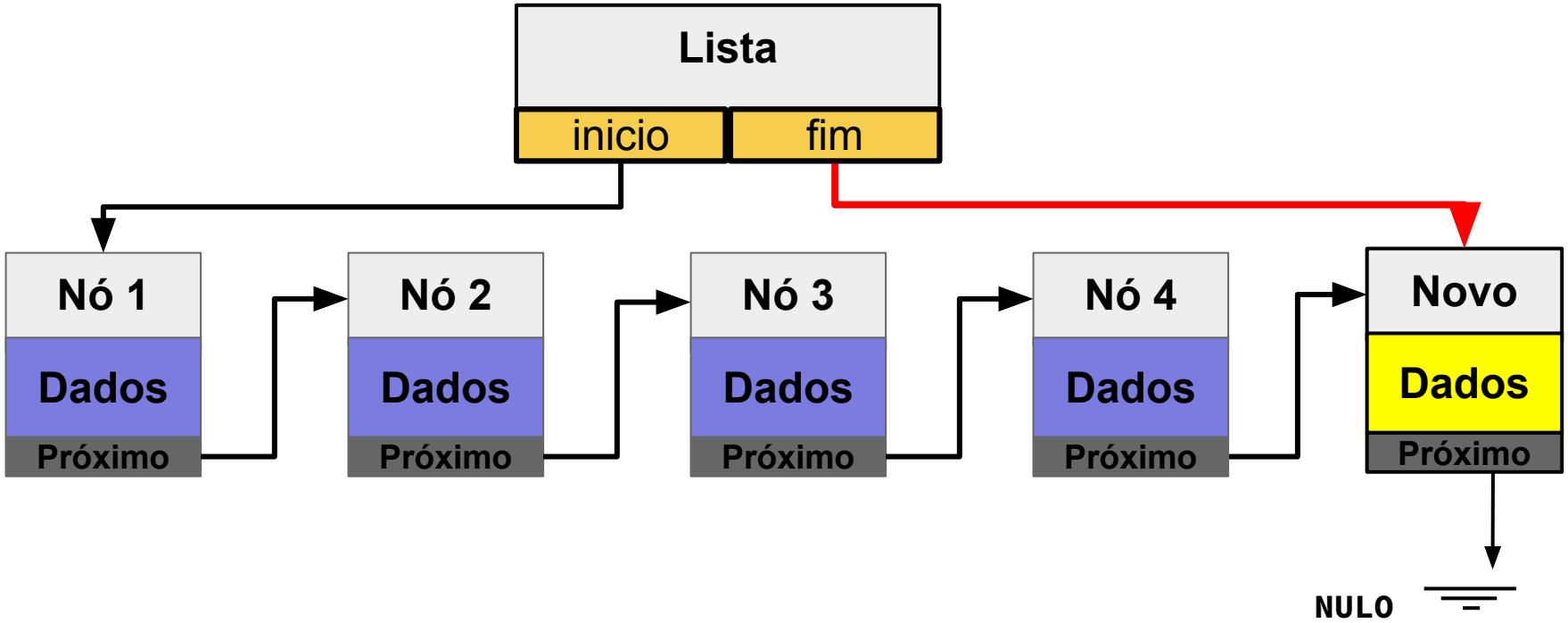
Inserção no fim - II



Inserção no fim - III



Inserção no fim - IV



Inserção no fim - Observação

Assim com na inserção no início, o método de inserção no fim pode ser chamado em uma lista vazia.

Assim, também é necessário verificar e tratar essa situação.

Como seria a inserção no fim sem o ponteiro último? E sem a variável que armazena o tamanho da lista?

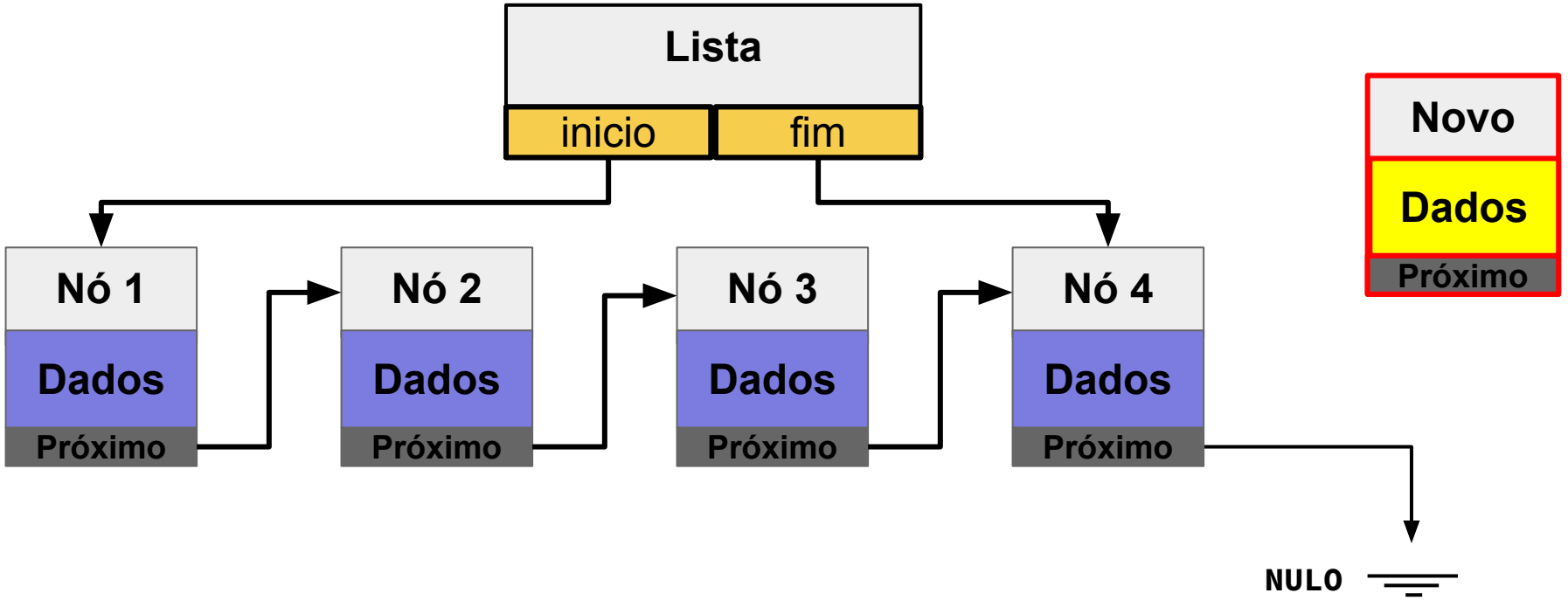
Inserção no fim - Pseudocódigo

inserirNoFimDaLista(valor):

```
se (lista.vazia()) {
    inserirEmListaVazia(valor);
} senão {
    novo ← criar_noh(valor); // cria um nó com o valor
    fim.proximo ← novo;
    fim ← novo;
    tamanho++; // caso seja interessante o atributo
}
```

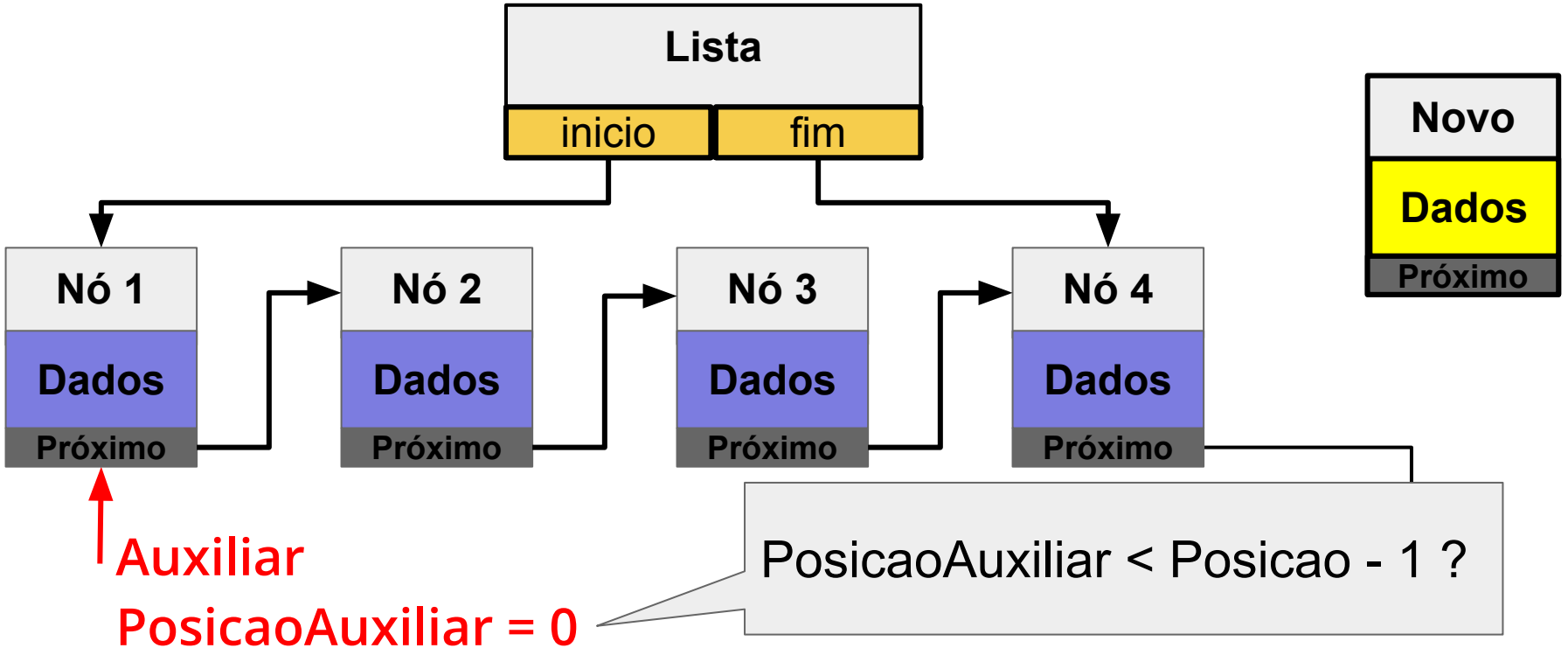
Inserção em posição específica - I

Inserir na posição 2
Posicao = 2



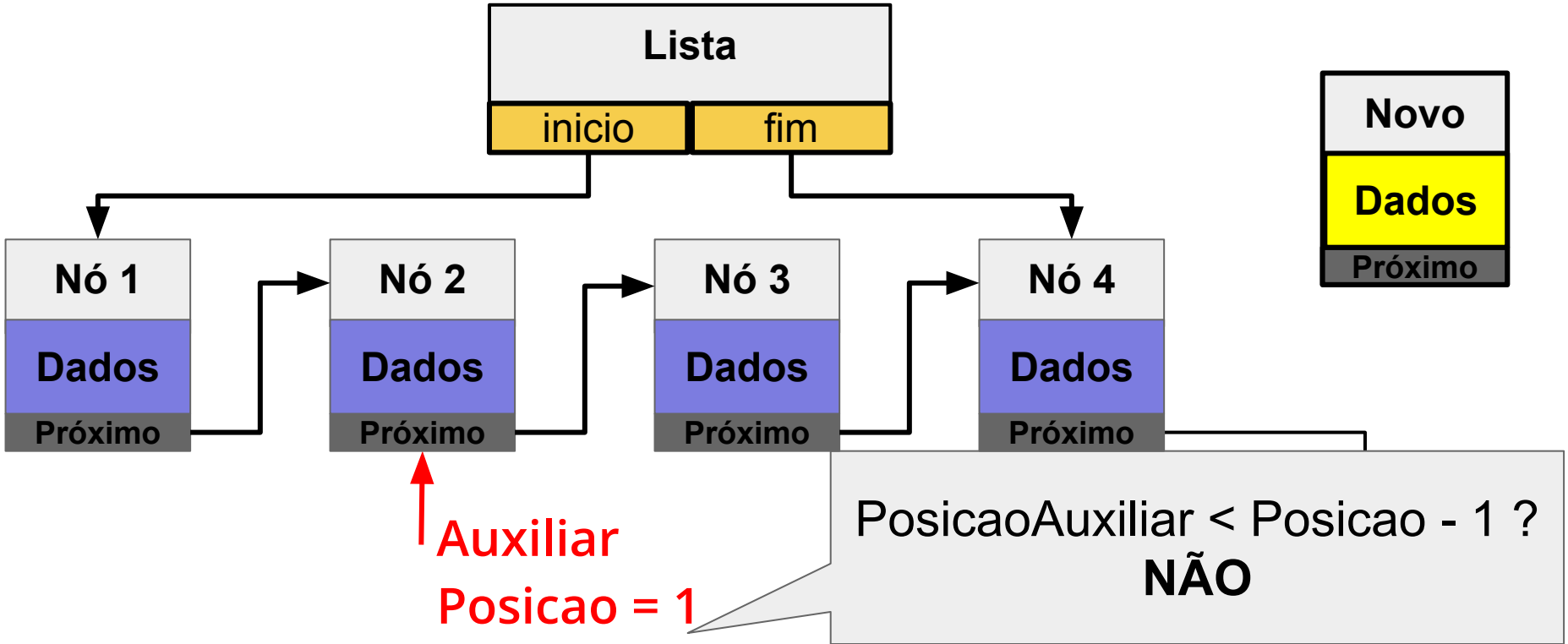
Inserção em posição específica - II

Inserir na posição 2
Posicao == 2



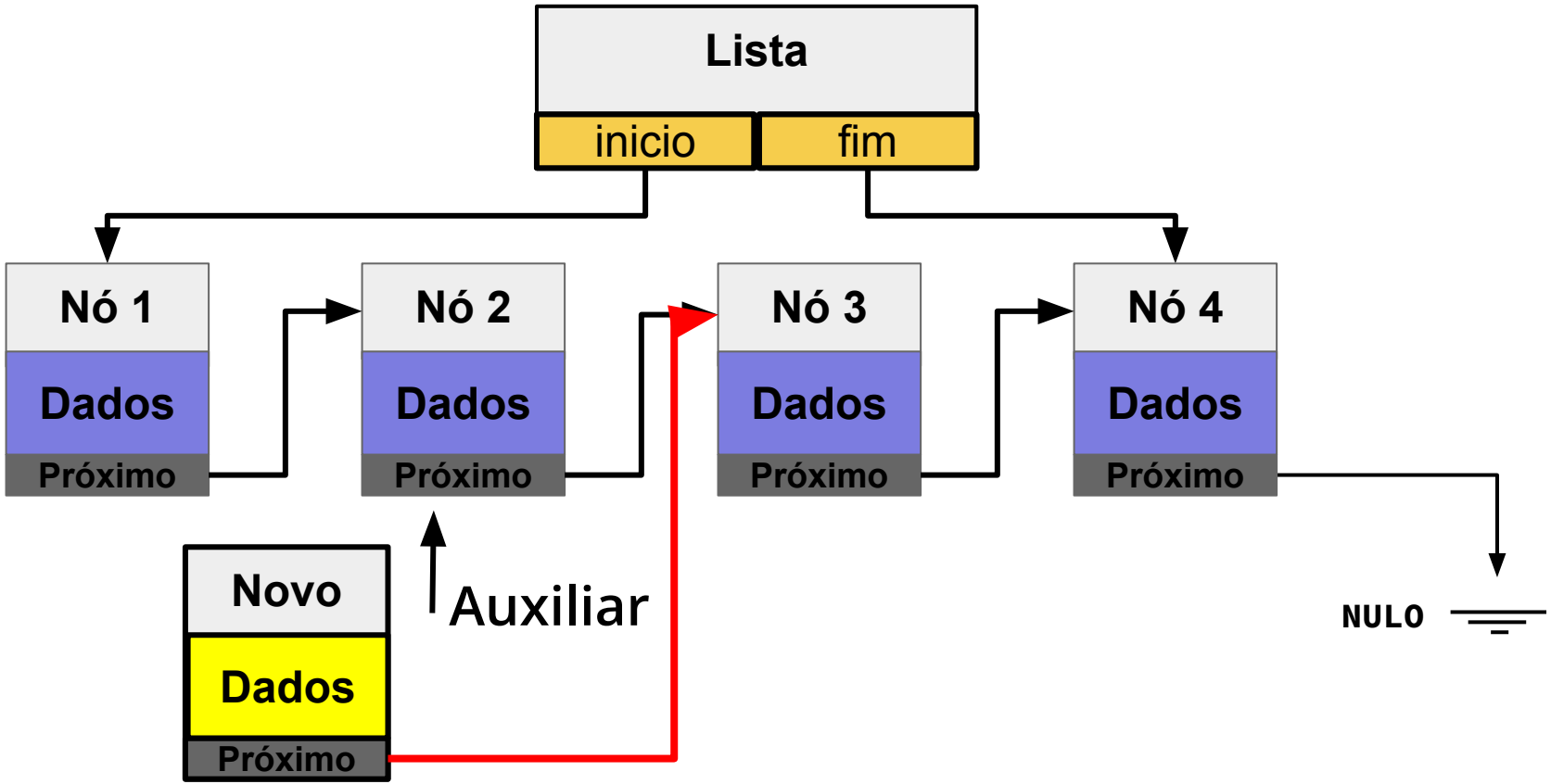
Inserção em posição específica - III

Inserir na posição 2



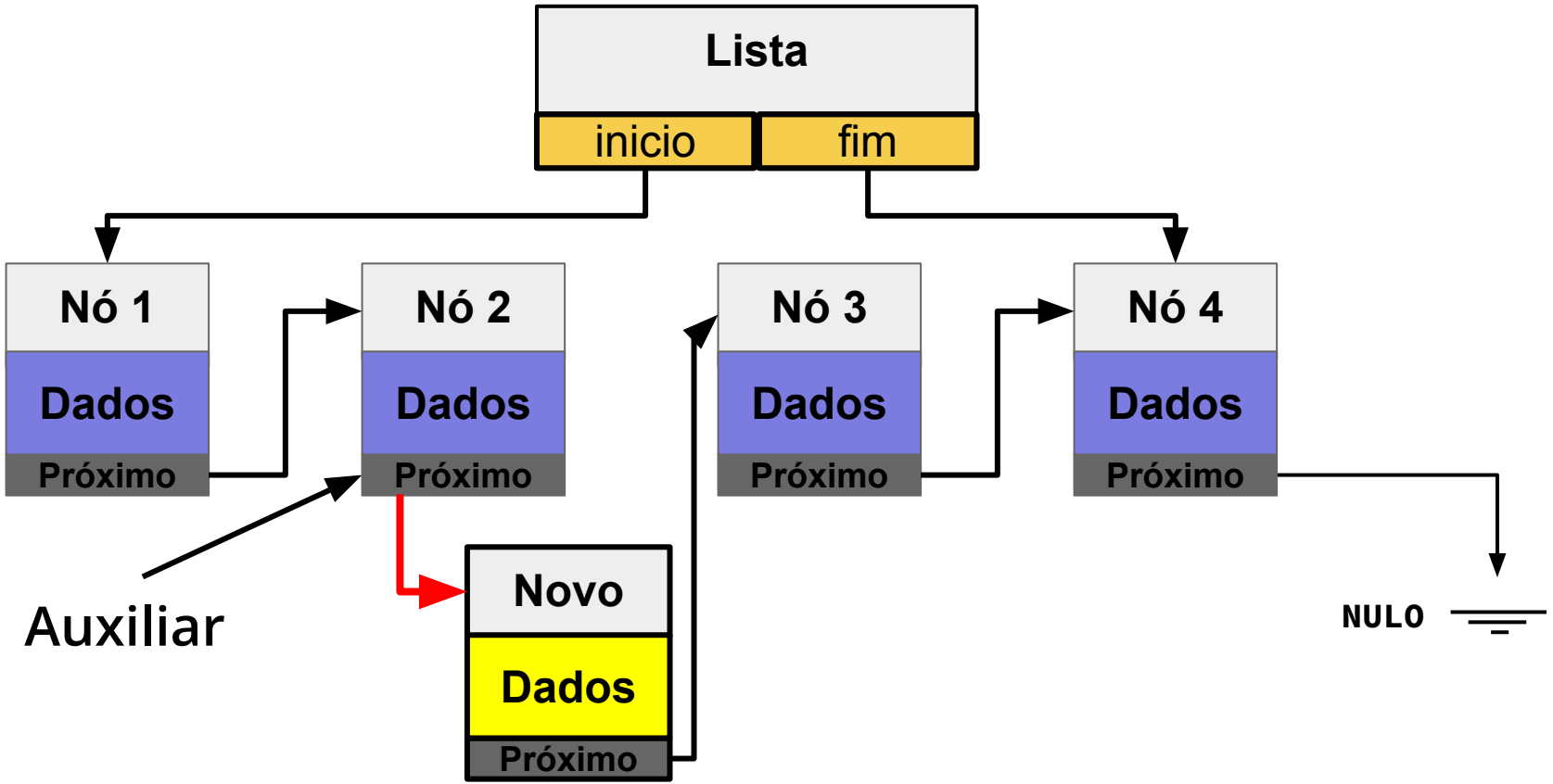
Inserção em posição específica - IV

Inserir na posição 2



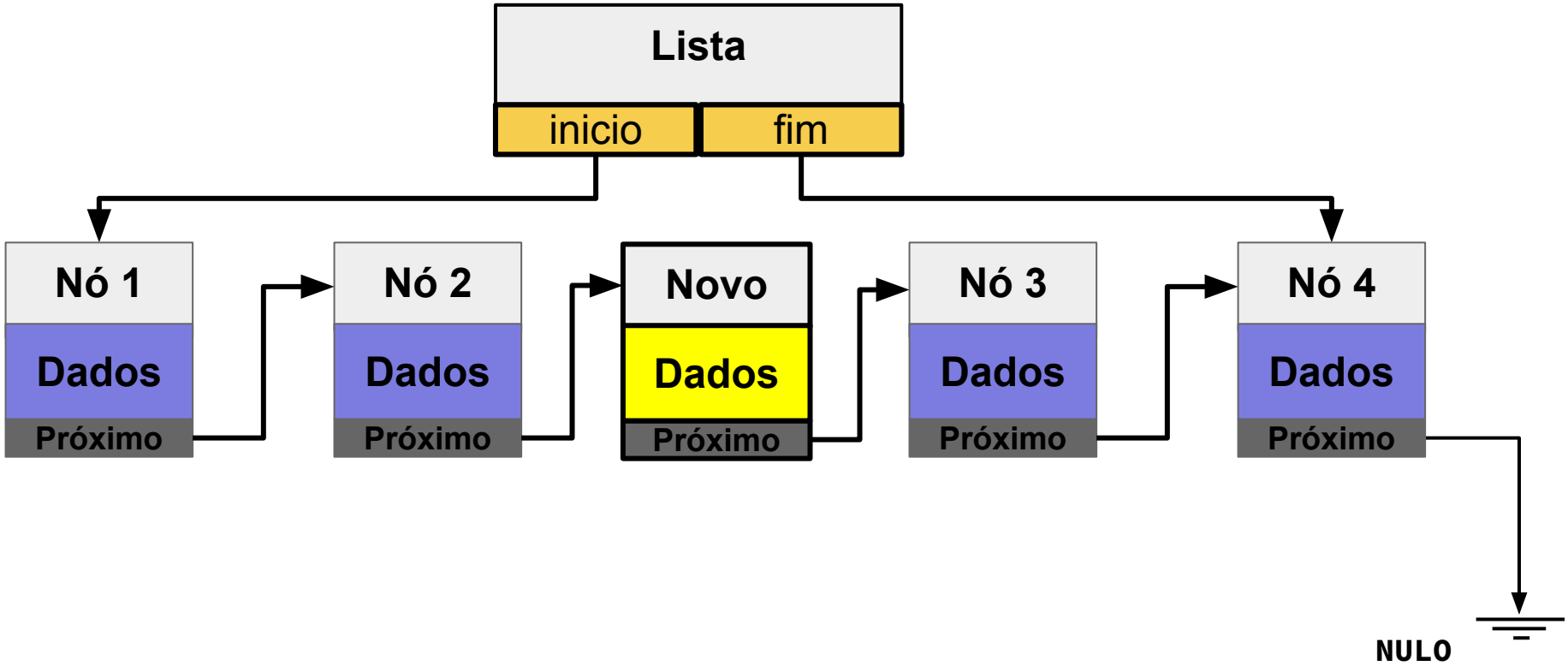
Inserção em posição específica - V

Inserir na posição 2



Inserção em posição específica - VI

Inserir na posição 2



Inserção em posição específica - Observações

Assim, como antes, o método de inserção em posição específica pode ser chamado em uma lista vazia. Assim, é necessário verificar e tratar essa situação.

Adicionalmente, pode ser que a posição seja o início ou fim da lista, o que também exige tratamento diferenciado.

É necessário também avaliar se as posições irão começar em zero ou em um.

Inserção em posição específica - pseudocódigo (i)

inserirEmPosicao(valor,posicao):

```
se (lista.vazia()) {
    inserirEmListaVazia(valor);
} senão se ( posição = 0 ) {
    inserirNoInicioDaLista(valor);
} senão se ( posição = tamanho ) {
    inserirNoFimDaLista(valor);
} senão {
```

Inserção em posição específica - pseudocódigo (II)

```
    novo ← criar_noh(valor); // cria um nó com o valor
    aux ← inicio; // nó auxiliar para percorrer a lista
    posAux ← 0; // posição do nó auxiliar
    // procura o nó anterior ao da posição
    enquanto ( posAux < (posicao - 1) ) {
        aux = aux.proximo;
        posAux++;
    }
    novo.proximo = aux.proximo;
    aux.proximo = novo;
    tamanho++; // caso seja interessante o atributo
}
```

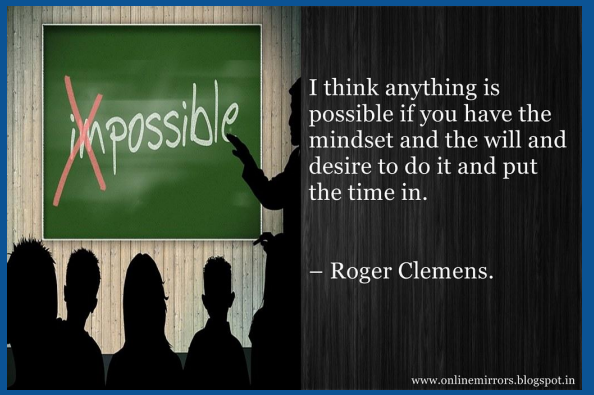
Inserção - Observações gerais

Em geral, implementações tradicionais de listas para uso geral costumam utilizar o duplo encadeamento e apresentar os três métodos de inserção aqui apresentados.

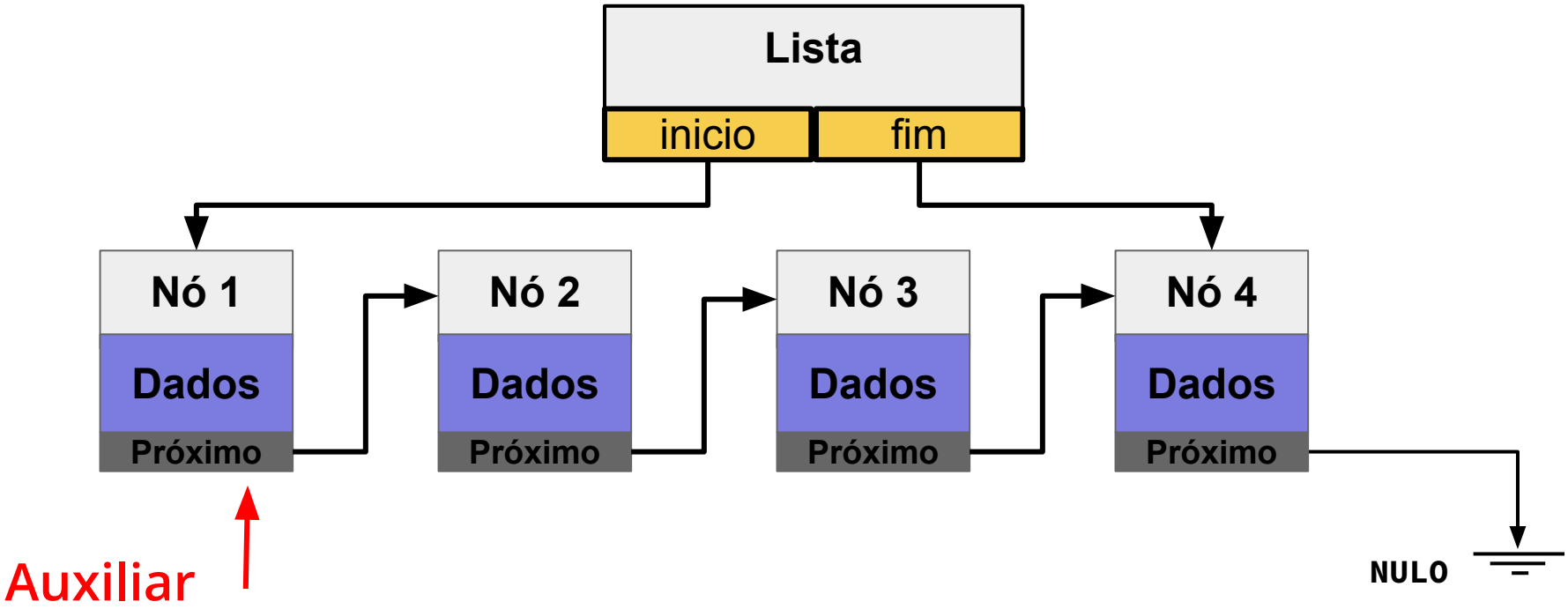
Entretanto, usos específicos podem exigir implementações em que apenas um dos três métodos de inserção sejam realmente implementados e utilizados.

O duplo encadeamento também não é necessário em boa parte dos problemas.

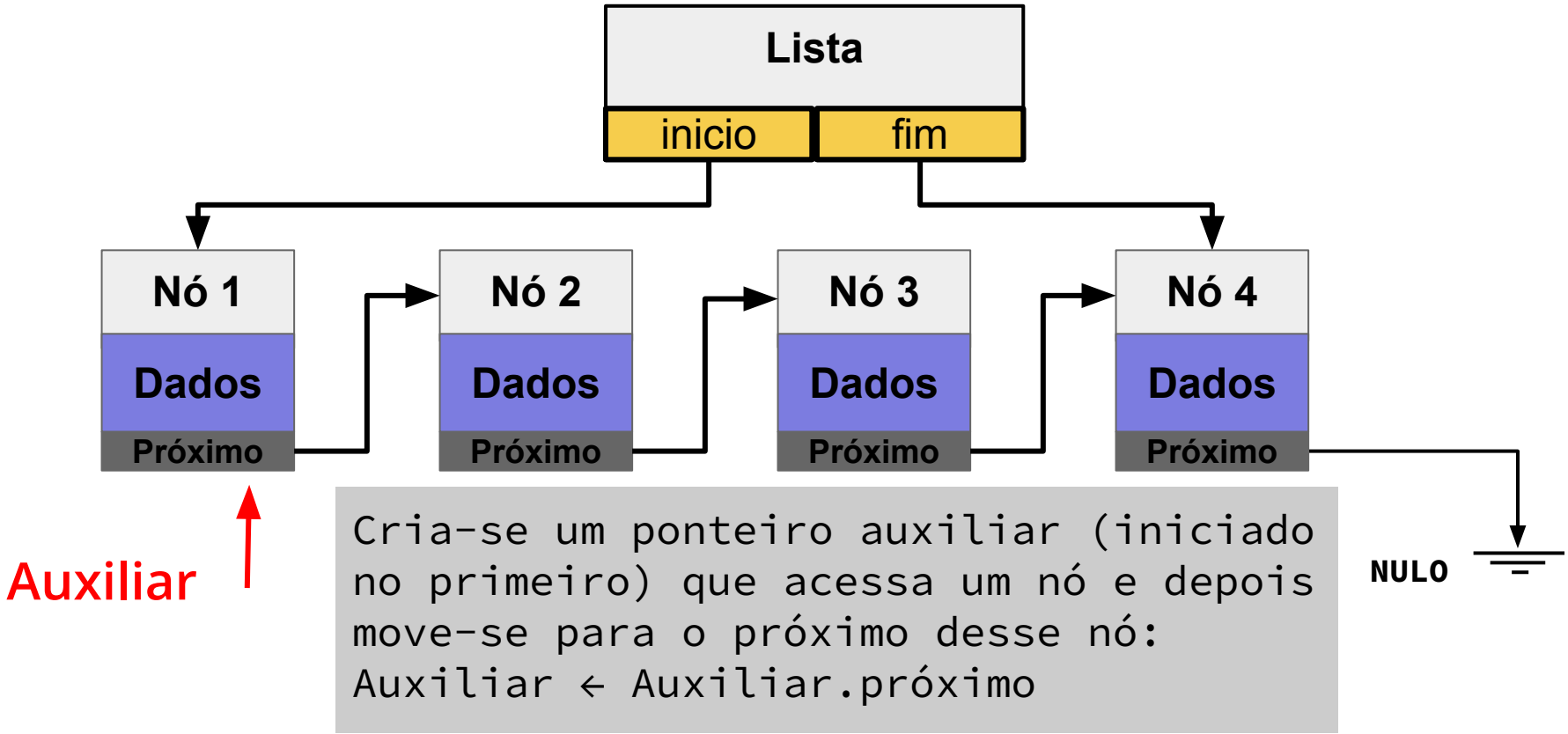
Métodos Usuais em Listas Parte II



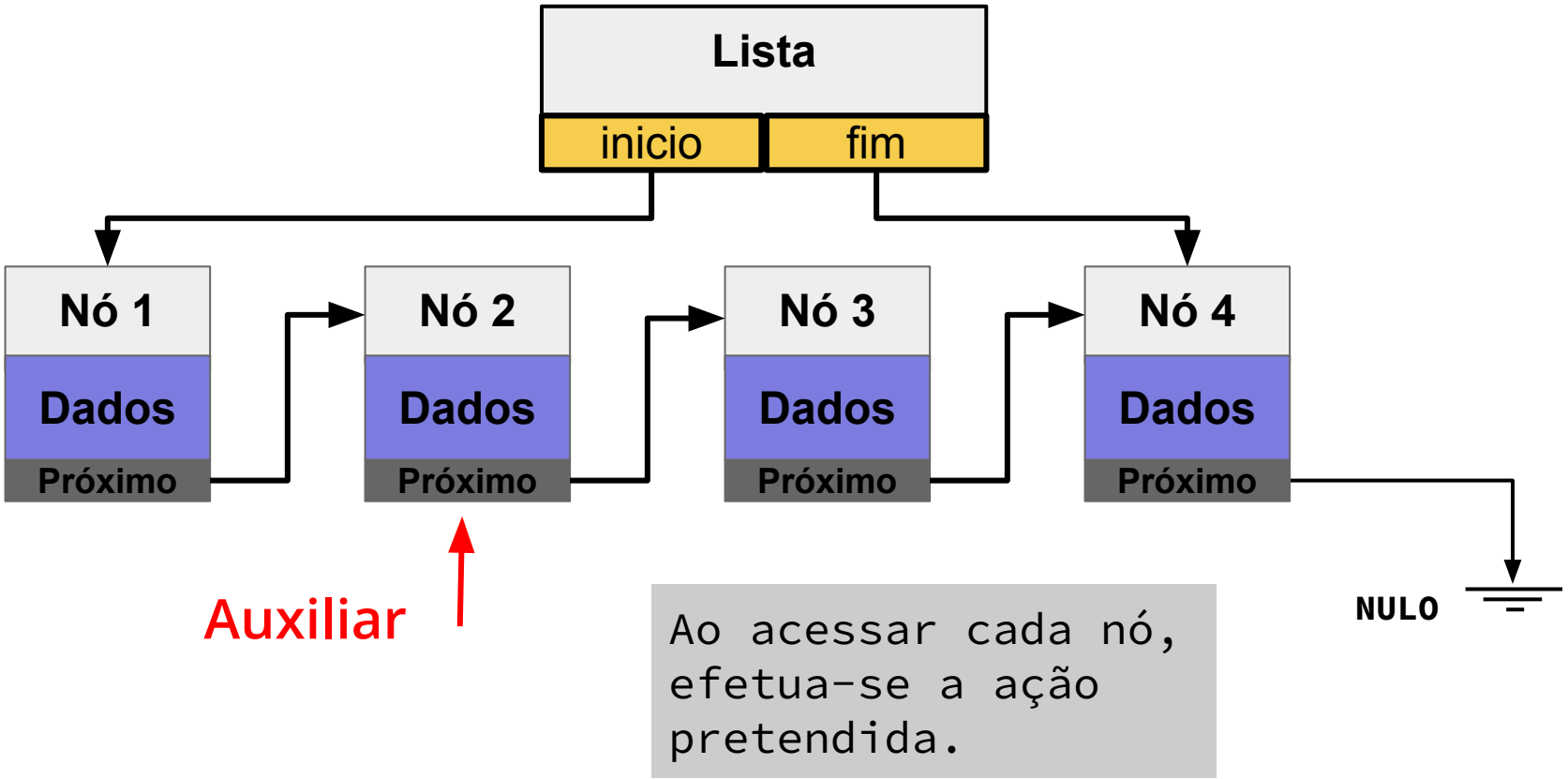
Percorrimento / Caminhamento - 1



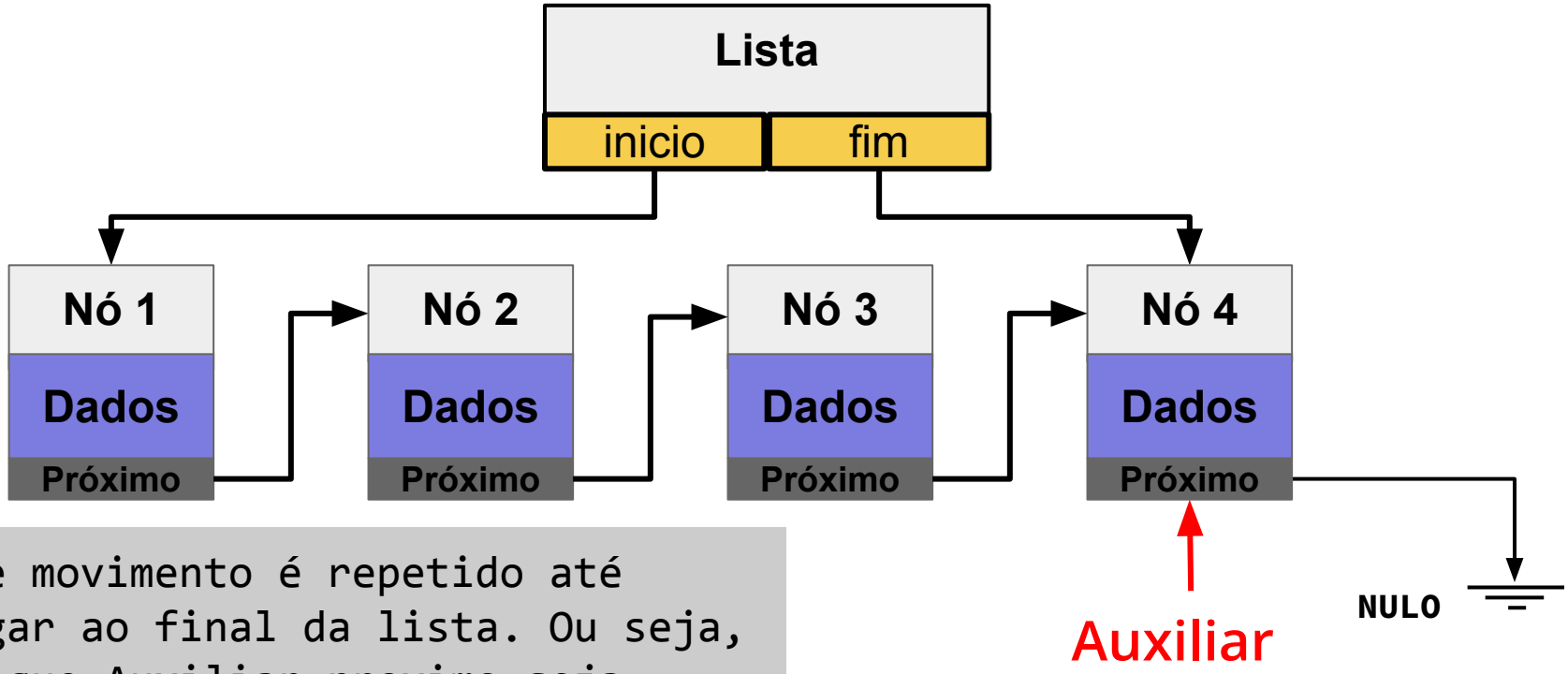
Percorrimento / Caminhamento - I



Percorrimento / Caminhamento - I



Percorrimento / Caminhamento - 1



Esse movimento é repetido até chegar ao final da lista. Ou seja, até que Auxiliar.proximo seja igual a NULO.

Percorrimento - Observação

Didaticamente, iremos imprimir o valor armazenado no nó ao percorrer a lista (isso será mantido nas demais estruturas vistas na disciplina).

Entretanto, outras ações podem ser efetuadas, de forma a atender a necessidade de uso.

Implementação pode ser recursiva ou iterativa.

Percorrimento (iterativo) - Pseudocódigo

percorrerLista():

```
auxiliar ← inicio; // nó auxiliar, começa no primeiro nó

enquanto (auxiliar ≠ NULO) {
    efetuaAcao(auxiliar);  // faz ação desejada
                           // (didaticamente impressão)
    auxiliar ← auxiliar.proximo;
}
```

Percorrimento (recursivo) - Pseudocódigo

percorrerLista():

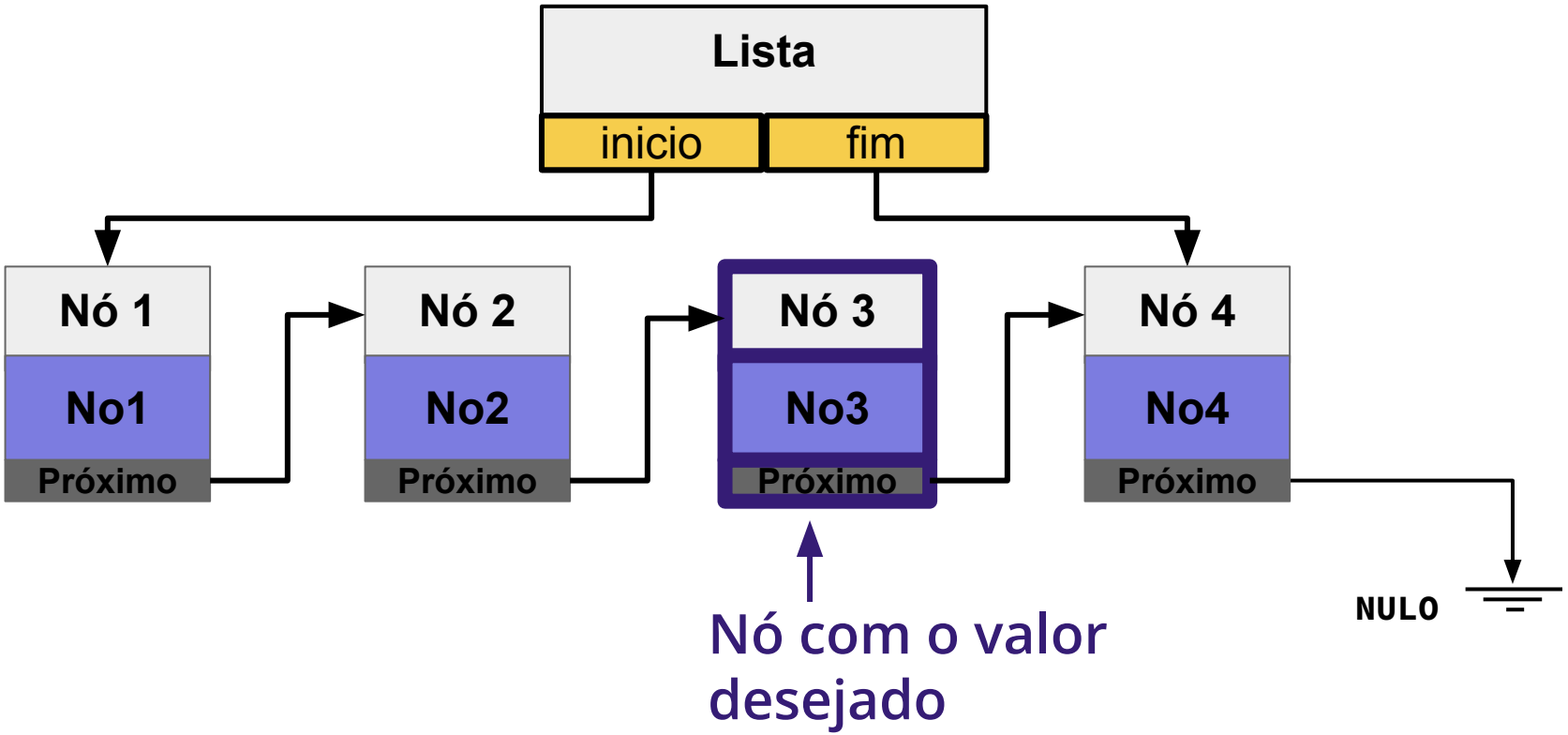
```
percorrerListaAux(inicio);
```

percorrerListaAux(umNoh):

```
se (umNoh ≠ NULO) {
    efetuaAcao(umNoh);  // faz ação desejada
                        // (didaticamente impressão)
    percorrerListaAux(umNoh.proximo);
}
```

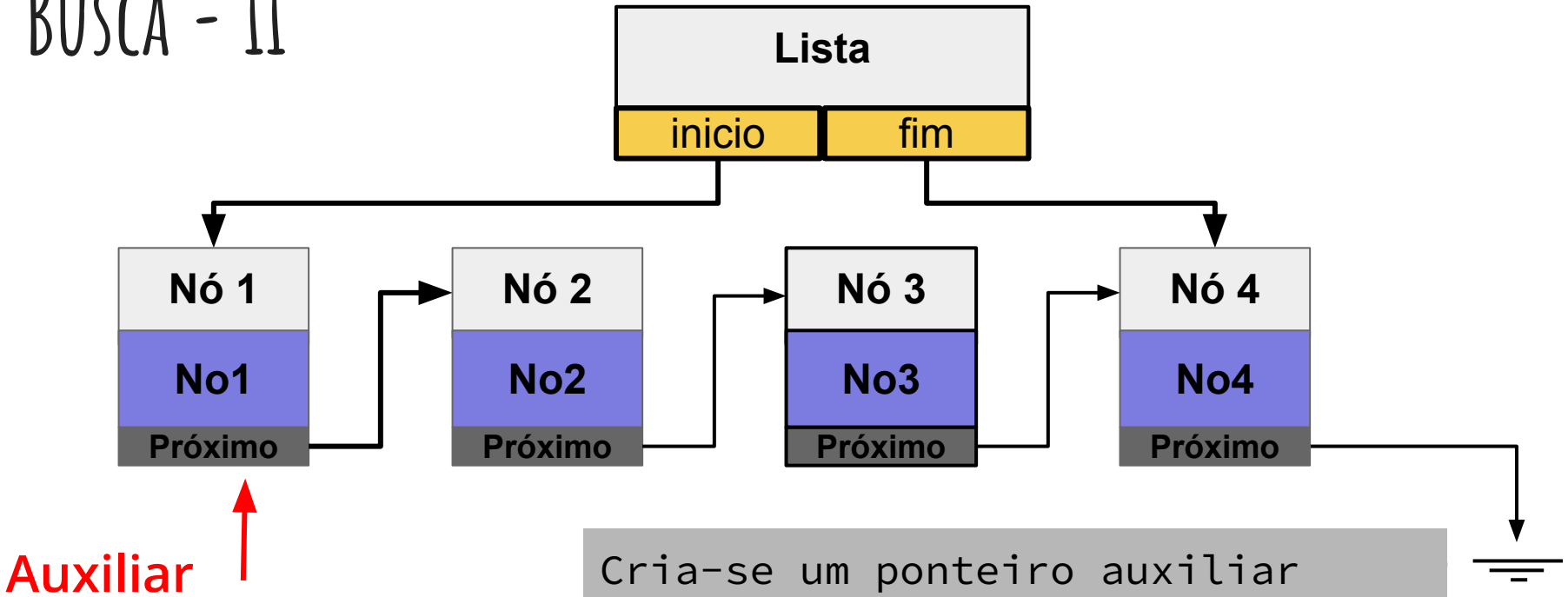
Busca - I

Valor desejado = No3



Busca - II

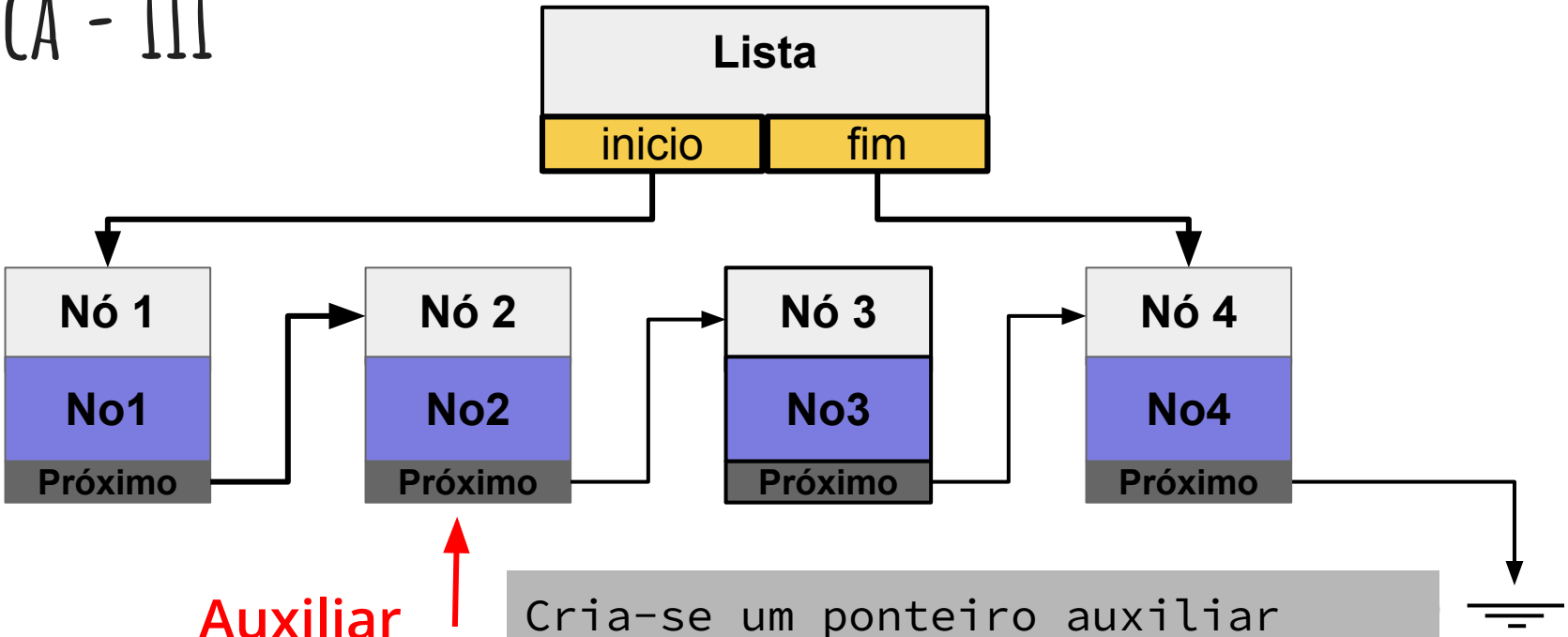
Valor desejado = No3



Cria-se um ponteiro auxiliar (iniciado no primeiro) que acessa um nó e depois move-se para o próximo desse nó:
Auxiliar ← Auxiliar.próximo

Busca - III

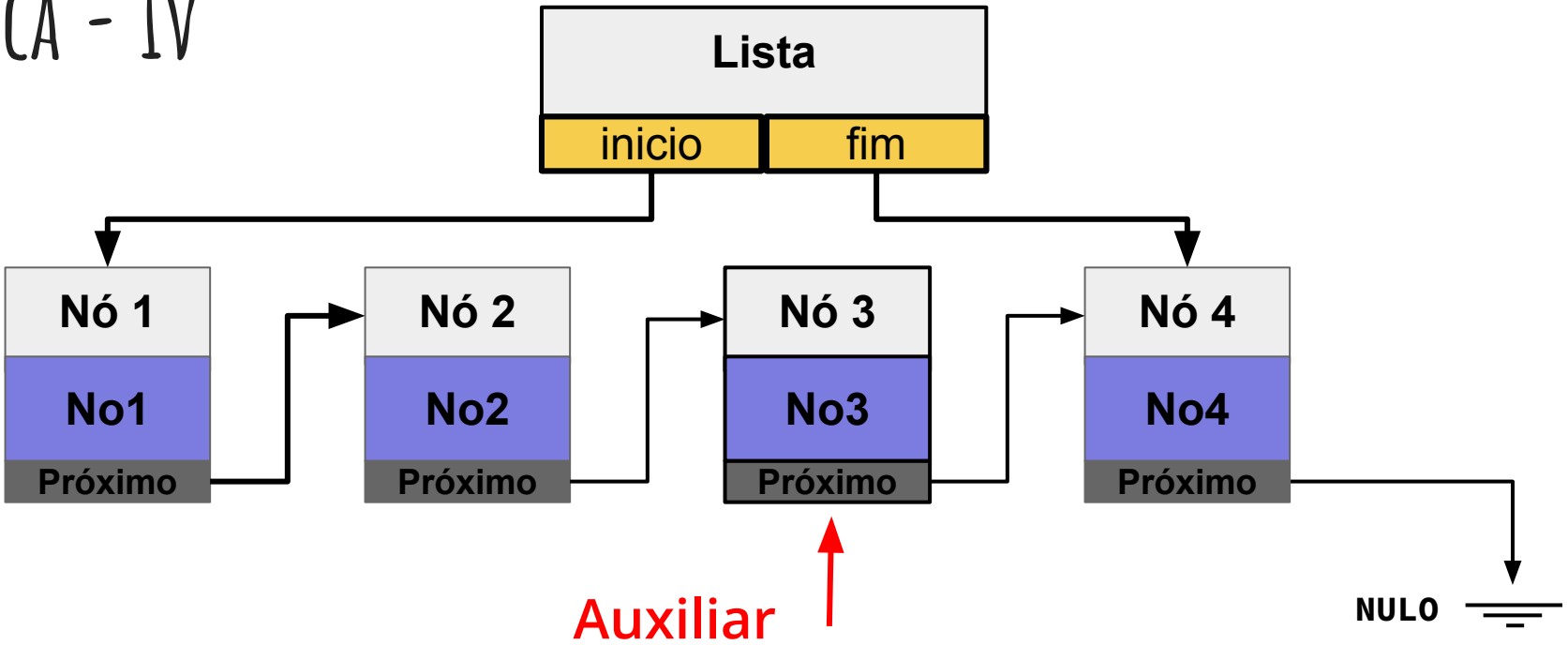
Valor desejado = No3



Cria-se um ponteiro auxiliar (iniciado no primeiro) que acessa um nó e depois move-se para o próximo desse nó:
Auxiliar ← Auxiliar.próximo

Busca - IV

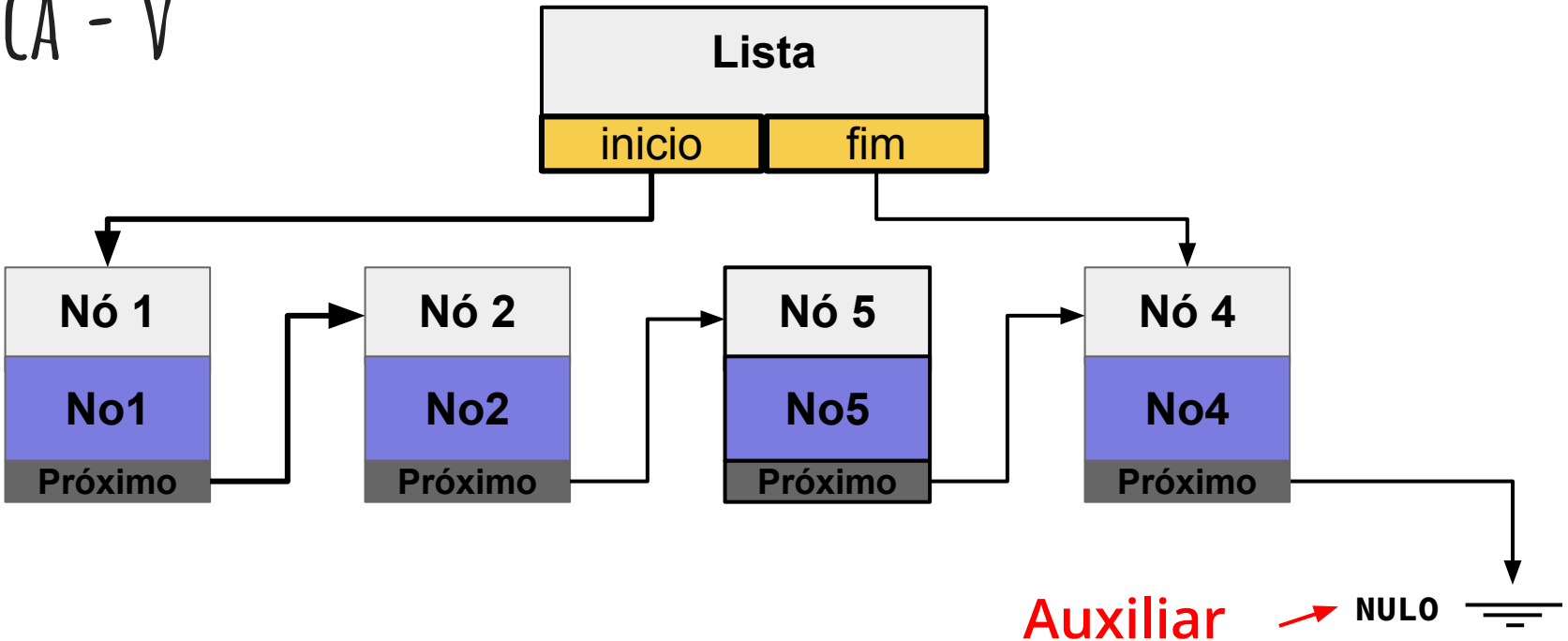
Valor desejado = No3



Movimento é repetido até chegar ao nó com valor desejado ou...

Busca - V

Valor desejado = No3



… chegar ao final da lista sem encontrar o elemento.

Busca - Observações - I

Métodos de busca geralmente são utilizados para acessar ou retornar um elemento com um dado valor.

No caso de final da lista, retorna-se nó nulo, posição inválida, ou erro, de acordo com o desejado.

Em alguns problemas, basta que a busca informe que o valor encontra-se presente na lista, nesse caso basta retornar verdadeiro ou falso de acordo com a situação.

Busca - Observações - II

Em outros problemas é necessário retornar a posição do nó encontrado.

Nesse caso, é preciso ter em mente que isso pode trazer problemas de eficiência, caso o objetivo seja acessar o nó novamente (uma vez que será necessário percorrer novamente a lista).

Busca - Observações - III

Caso seja necessário retornar o dado em si, é possível retornar uma cópia, uma referência ou um ponteiro para o elemento armazenado no nó.

Algumas implementações optam por retornar um ponteiro não para o dado, mas para o nó que contém o elemento buscado. Essa opção pode prejudicar severamente o *encapsulamento* dos dados, se não implementada adequadamente. É melhor retornar apenas o dado armazenado, por cópia, referência ou ponteiro, de acordo com a necessidade.

Busca - Observações - IV

Pode parecer estranho buscar um dado na lista, se você já o passa como parâmetro da busca.

Mas, em geral a busca é feita com uma parte do dado, com o objetivo de obter o todas as informações do elemento procurado.

Assim, procuramos os dados completos de um cliente, passando-se seu nome ou seu CPF, por exemplo.

Busca - Observações - IV

Pode parecer [illegible] a, se você já o passa como parâmetro da busca.

buscaCliente(111.111.111-11)

Mas, em geral a busca é feita com uma parte do dado, com o objetivo de o[illegible]elemento procurado.

Cliente 312
CPF: 111.111.111-11
Nome: João José da Silva Silva
Endereço: Rua dos Bobos, n. 0
...

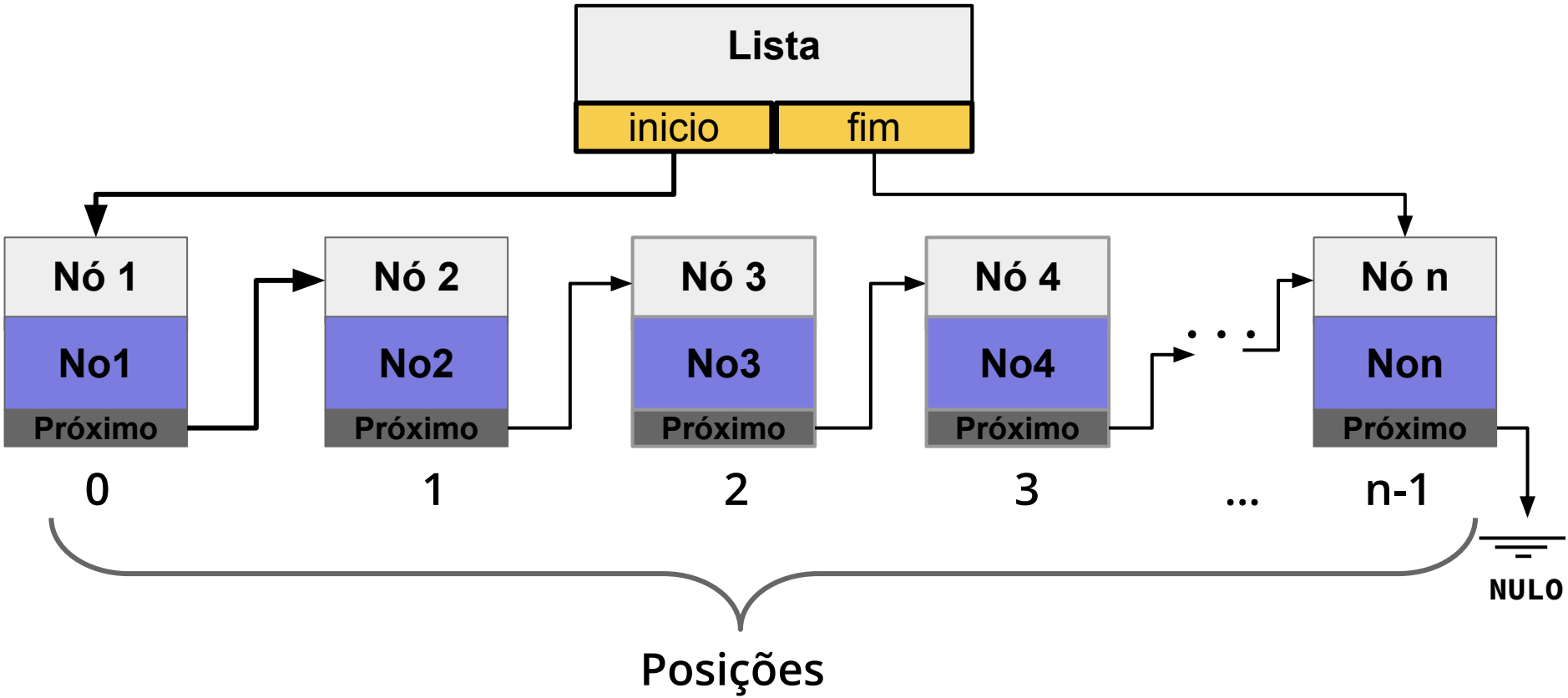
Assim, procur[illegible]liente, passando-se s[illegible].

Busca - Pseudocódigo

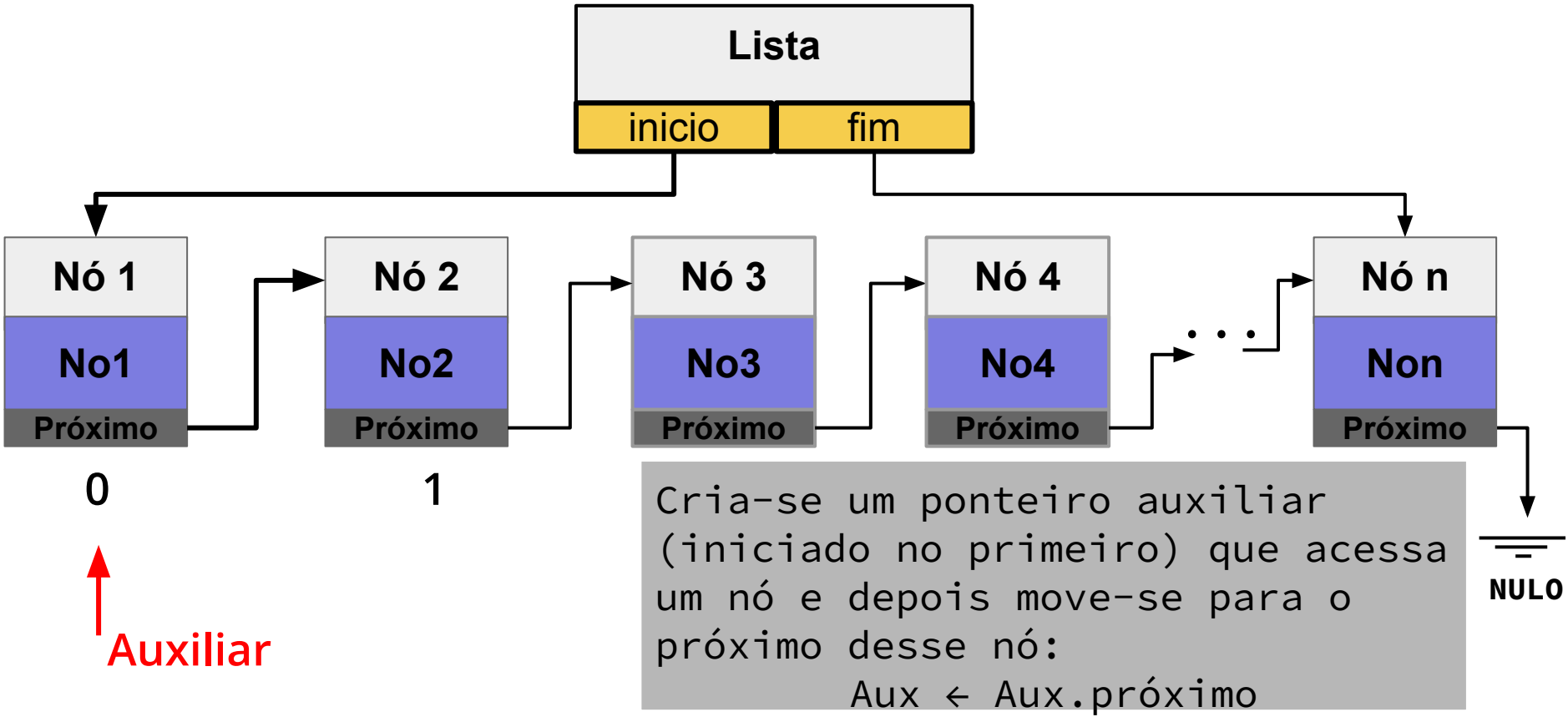
buscarNaLista(valor):

```
auxiliar ← inicio; // nó auxiliar, começa no primeiro nó

enquanto (auxiliar ≠ NULO) {
    se auxiliar.contem(valor) {
        efetuaAcao(auxiliar);  // faz ação desejada
        auxiliar ← NULO; // encerra o laço
    } senão
        auxiliar ← auxiliar.proximo;
}
```

Acesso a Posição - I

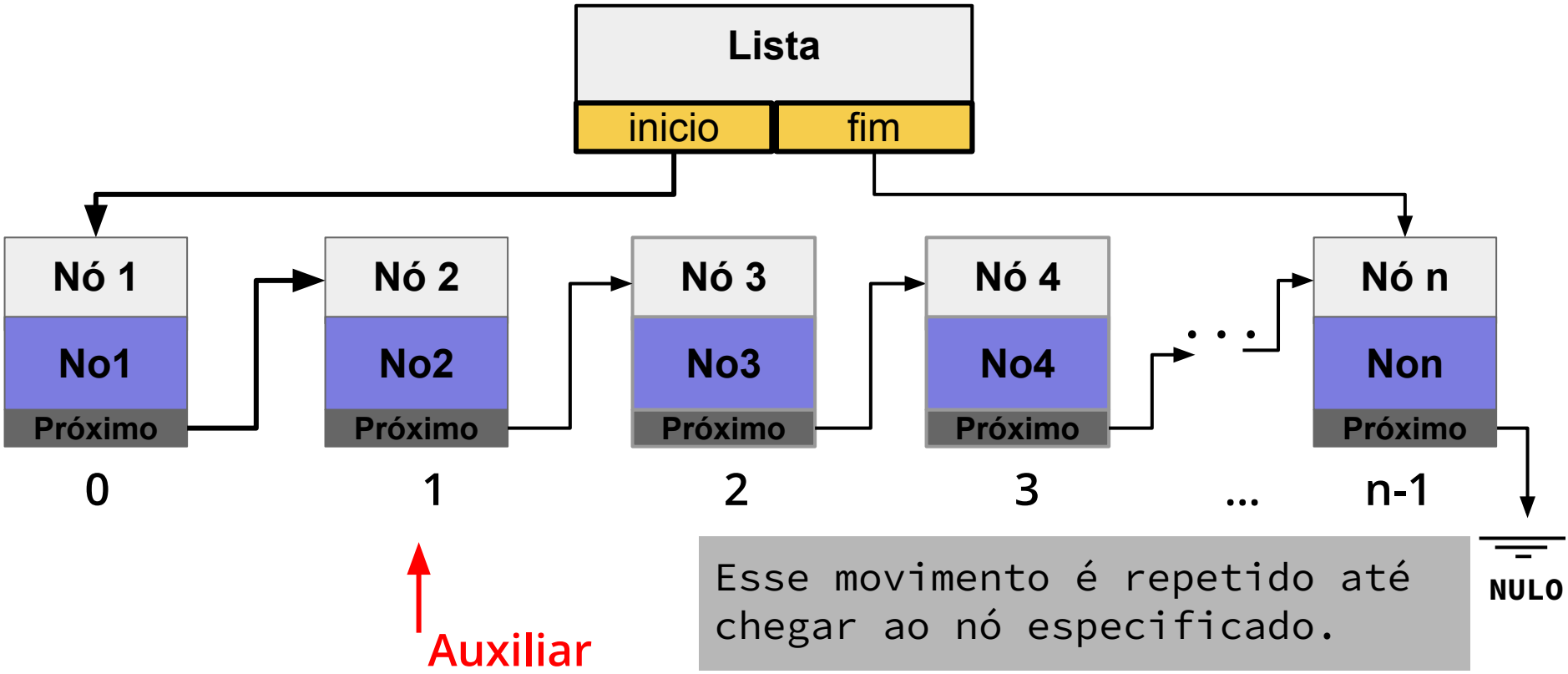


Acesso a Posição - II




```
Cria-se um ponteiro auxiliar
(iniciado no primeiro) que acessa
um nó e depois move-se para o
próximo desse nó:
        Aux ← Aux.próximo
```

Acesso a Posição - II



Acesso a Posição - Observações

O acesso a uma posição geralmente é realizado para efetuar uma ação específica. O operador [] é geralmente sobrecarregado com esse método.

Em alguns casos essa ação implica em retornar um ponteiro ou referência para o nó. Caso seja ponteiro ou referência, deve-se ter cuidado com a quebra de *encapsulamento*.

É necessário definir se posições começam em zero ou um.

Acesso a Posição - Pseudocódigo

acessarPosicao(posicao):

```
se ((posicao < 0) ou (posicao >= tamanho)) sairComErro();

auxiliar ← inicio; // nó auxiliar, começa no primeiro nó
localizacao ← 0;
enquanto (posicao > localizacao) {
    auxiliar ← auxiliar.proximo;
    localizacao++;
}

efetuaAcao(auxiliar);  // faz ação desejada (e.g.: retorno)
```

Iteradores - I

Imagine a seguinte situação: você precisa efetuar alguma ação nos nós nas posições i, i-1 e i+1.

A tendência natural é utilizar o acesso a posição, por meio do operador []. Em uma lista, isso faz com que o desempenho do algoritmo seja severamente prejudicado.

É mais adequado acessar o nó anterior ou próximo, usando iteradores.

Iteradores - II

Em linhas gerais, Um iterador é um objeto que permite percorrer uma coleção de elementos (contêineres), especialmente vetores e listas. Podem ser implementados de diferentes maneiras em diferentes linguagens de programação.

Um iterador pode ser implementado, por exemplo, como um tipo de ponteiro que possui duas operações primárias: referenciar um elemento particular e modificar a si mesmo para apontar para o próximo elemento.

Iteradores - Exemplo - I/II

acessarPosicao(posicao):

```
se ((posicao < 0) ou (posicao >= tamanho)) sairComErro();

auxiliar ← inicio; // nó auxiliar, começa no primeiro nó
localizacao = 0;
enquanto (posicao > localizacao) {
    auxiliar ← auxiliar.proximo;
    localizacao++;
}

retorna Iterador;   // retorna ponteiro ou referência
```

Iteradores - Exemplo - II/II

imprimir(lista):

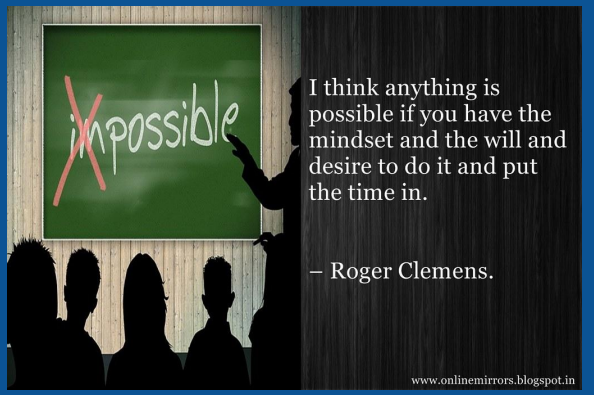
```
// funcões para acessar posição inicial e final,
// variações da acessar posição
inicio ← lista.acessarPosicaoInicial(); // iterador
fim ← lista.acessarPosicaoFinal(); // iterador

enquanto (inicio ≠ fim) {
    imprime(inicio.valor);
    inicio ← inicio.proximo;
}
```

Iteradores - Exemplo em C++:

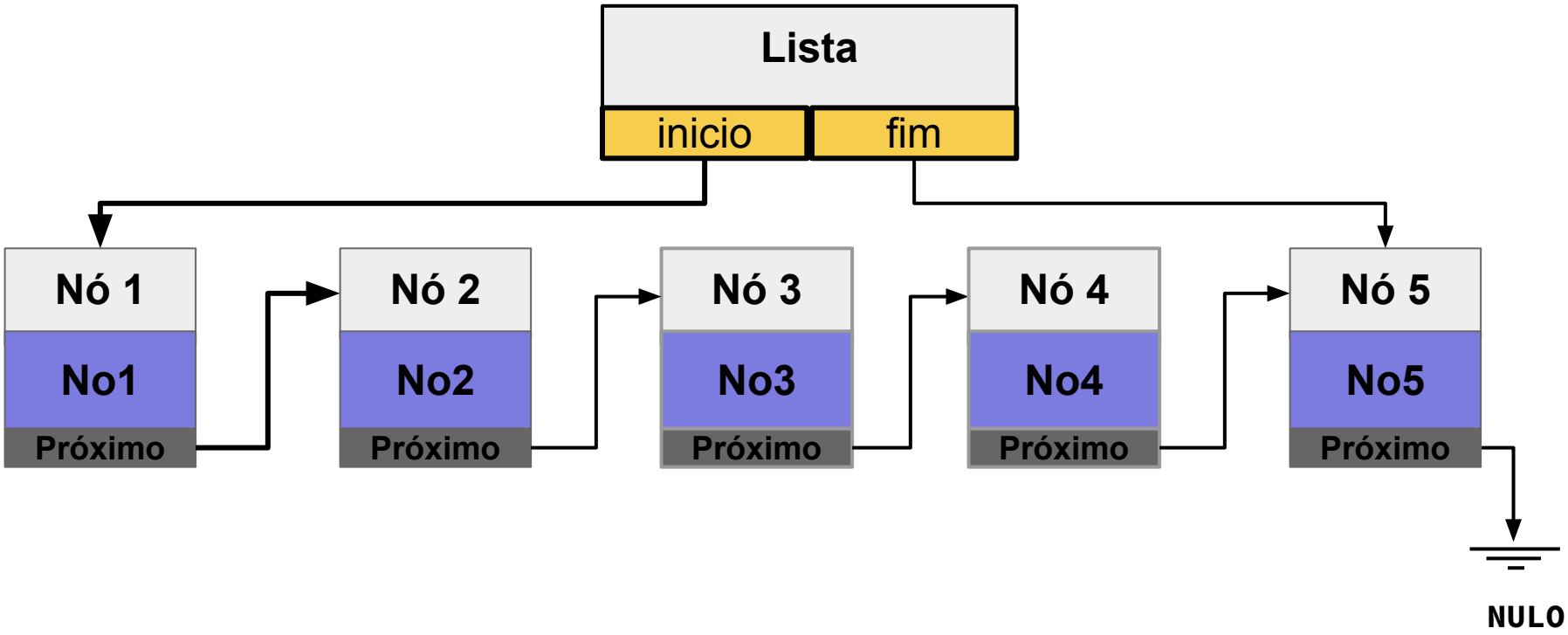
```
list<int> items;

...

for (list<int>::iterator i = items.begin();
      i != items.end(); ++i) {
         cout << *i;
}
```

São iteradores nesse código: `i`, `items.begin()` e `items.end()`

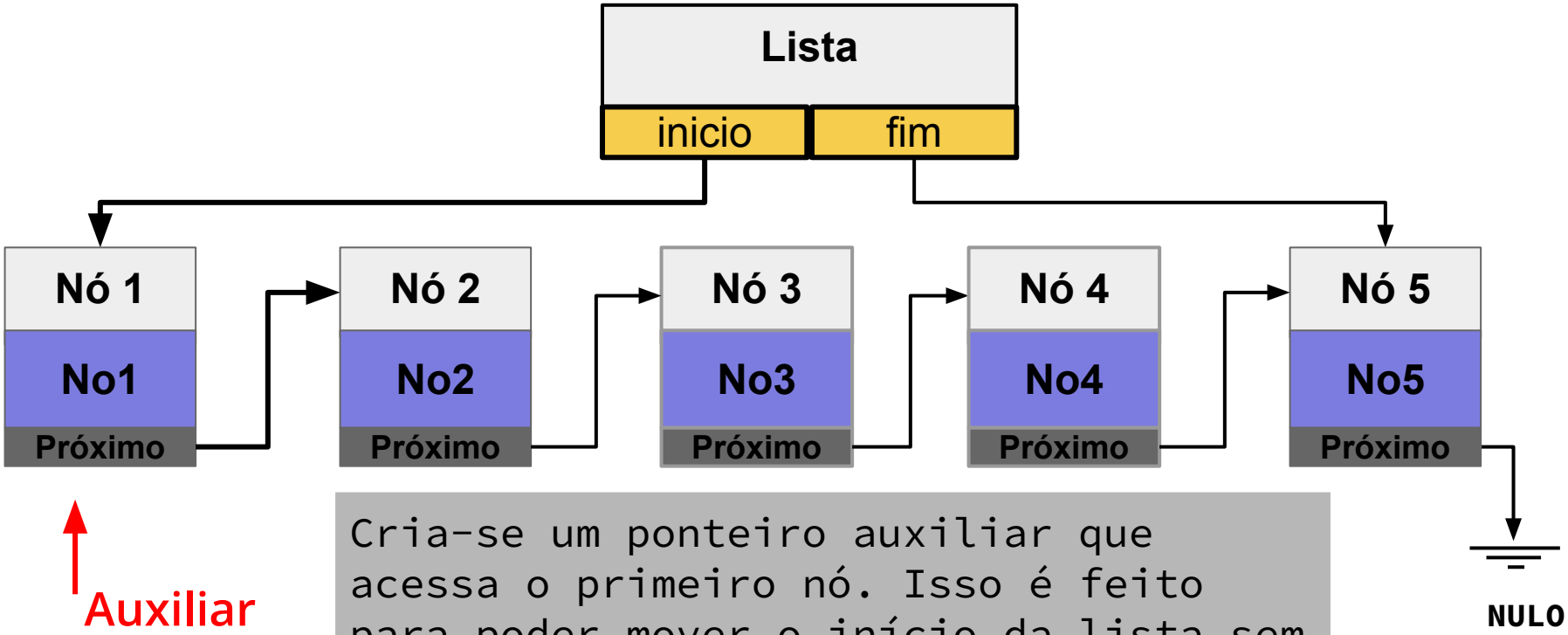
Métodos Usuais em Listas Parte III



Remove no início - I

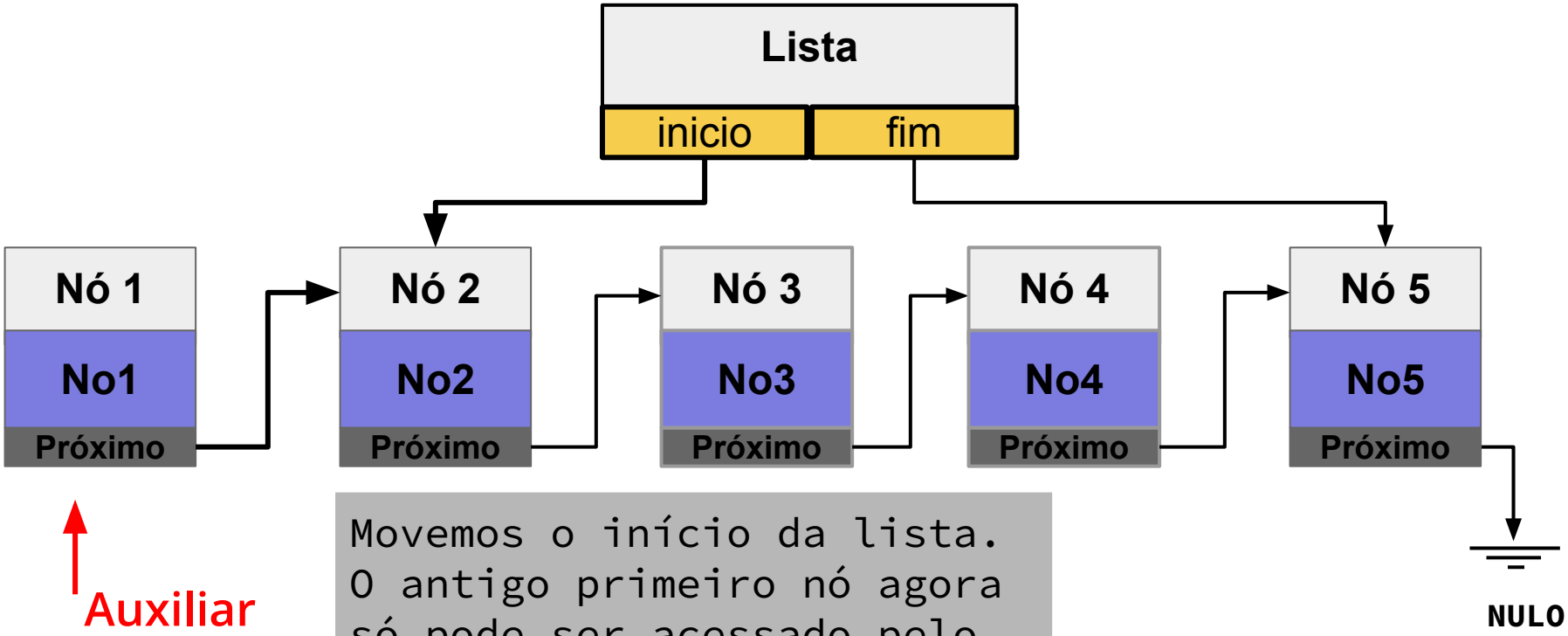


Remove no início - II



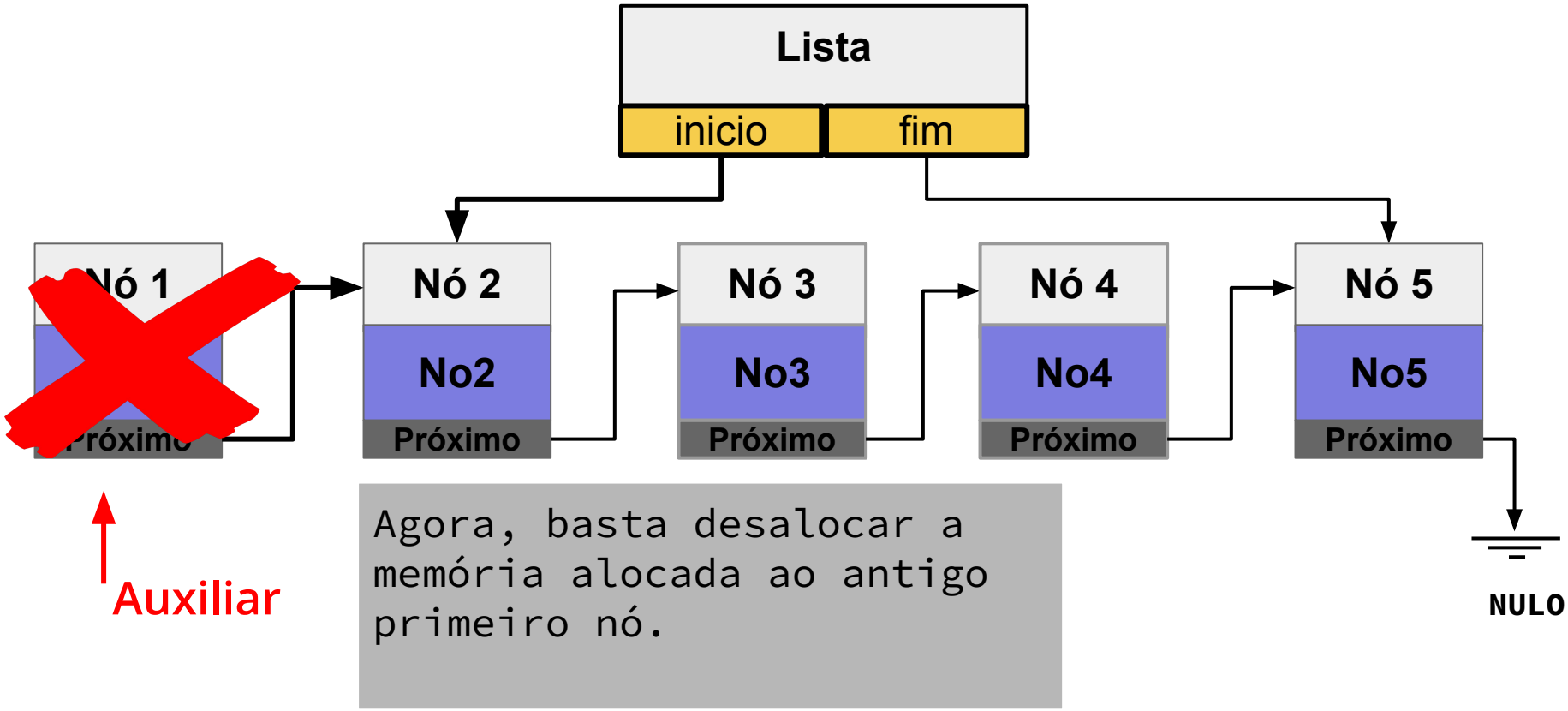
Cria-se um ponteiro auxiliar que acessa o primeiro nó. Isso é feito para poder mover o início da lista sem perder o acesso ao antigo primeiro nó.

Remove no início - III

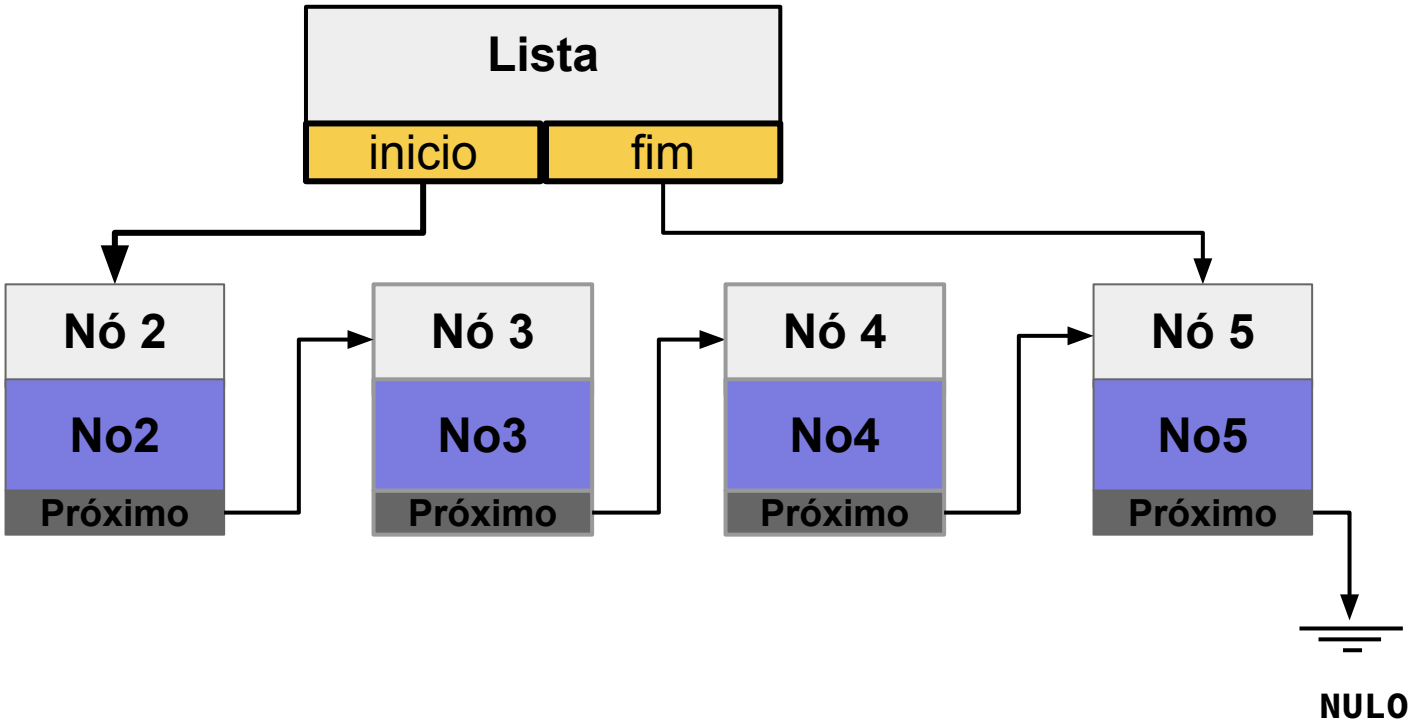


Movemos o início da lista. O antigo primeiro nó agora só pode ser acessado pelo ponteiro auxiliar.

Remove no início - IV



Remove no início - V



Remoção no início - Observações

Caso a remoção seja chamada em uma lista vazia, é necessário gerar erro para a aplicação. O ideal é que isso seja feito usando tratamento de exceções.

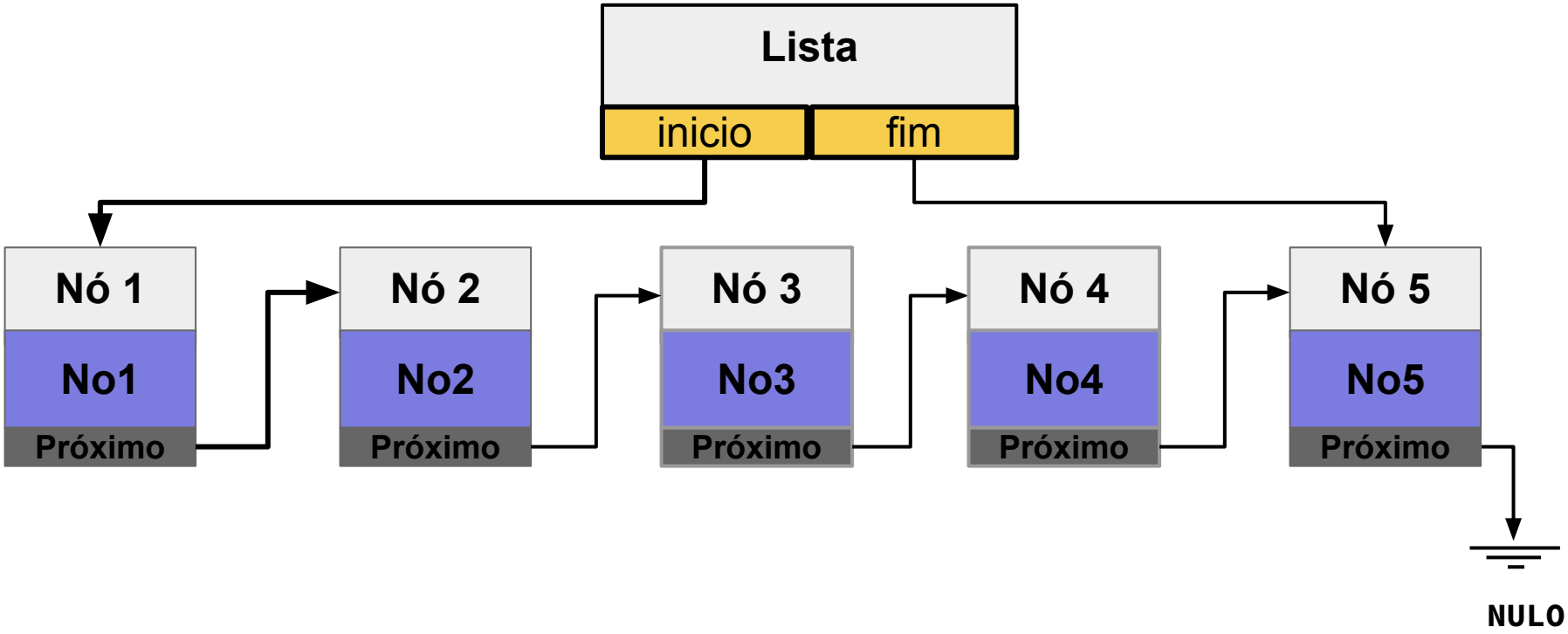
Caso a lista fique vazia após remover o primeiro (e último) elemento, então é necessário fazer o ponteiro fim apontar para NULO.

Remoção no início - Pseudocódigo

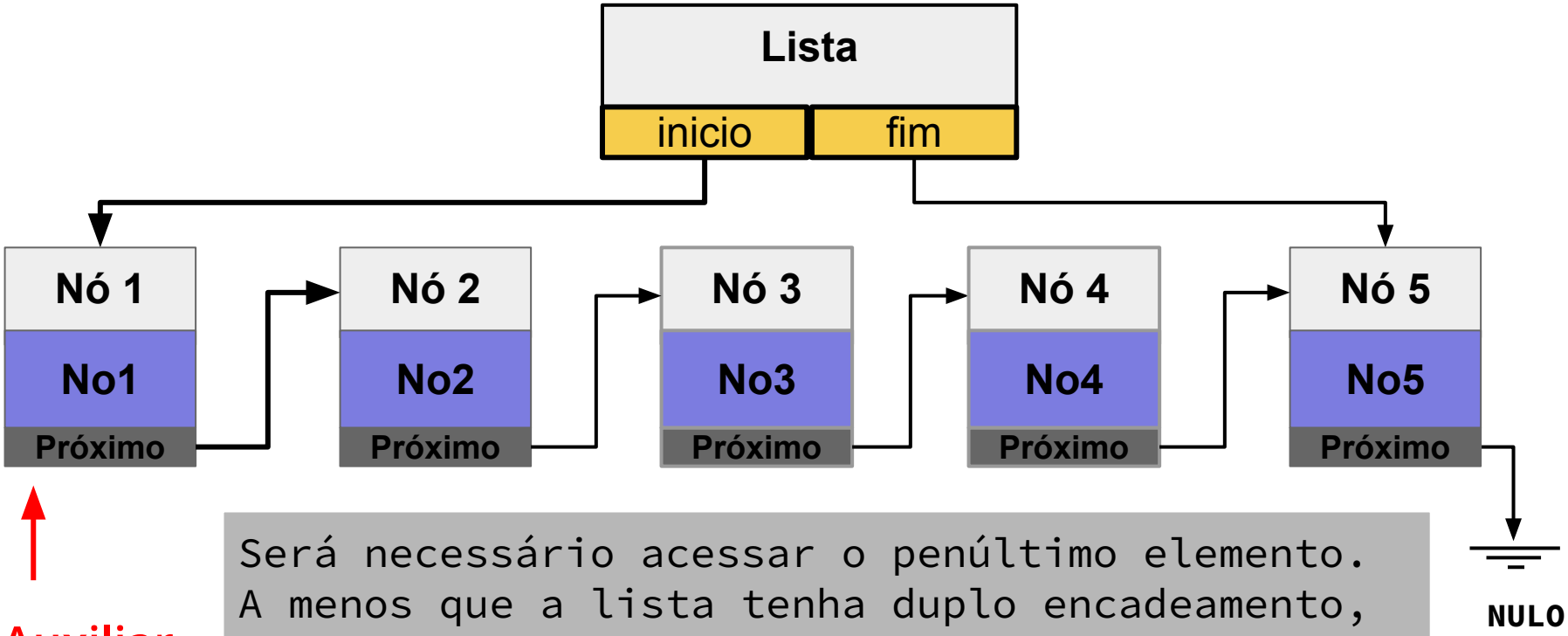
removeNoInicio():

```
se listaVazia() sairComErro()
aux ← inicio;
valor ← aux.dado;
inicio ← aux.proximo;
apagar(aux);
tamanho--;
se listaVazia()
    fim ← NULO;
efetuaAcao(valor); // faz ação desejada (e.g.: retorno)
```

Remove no fim - I

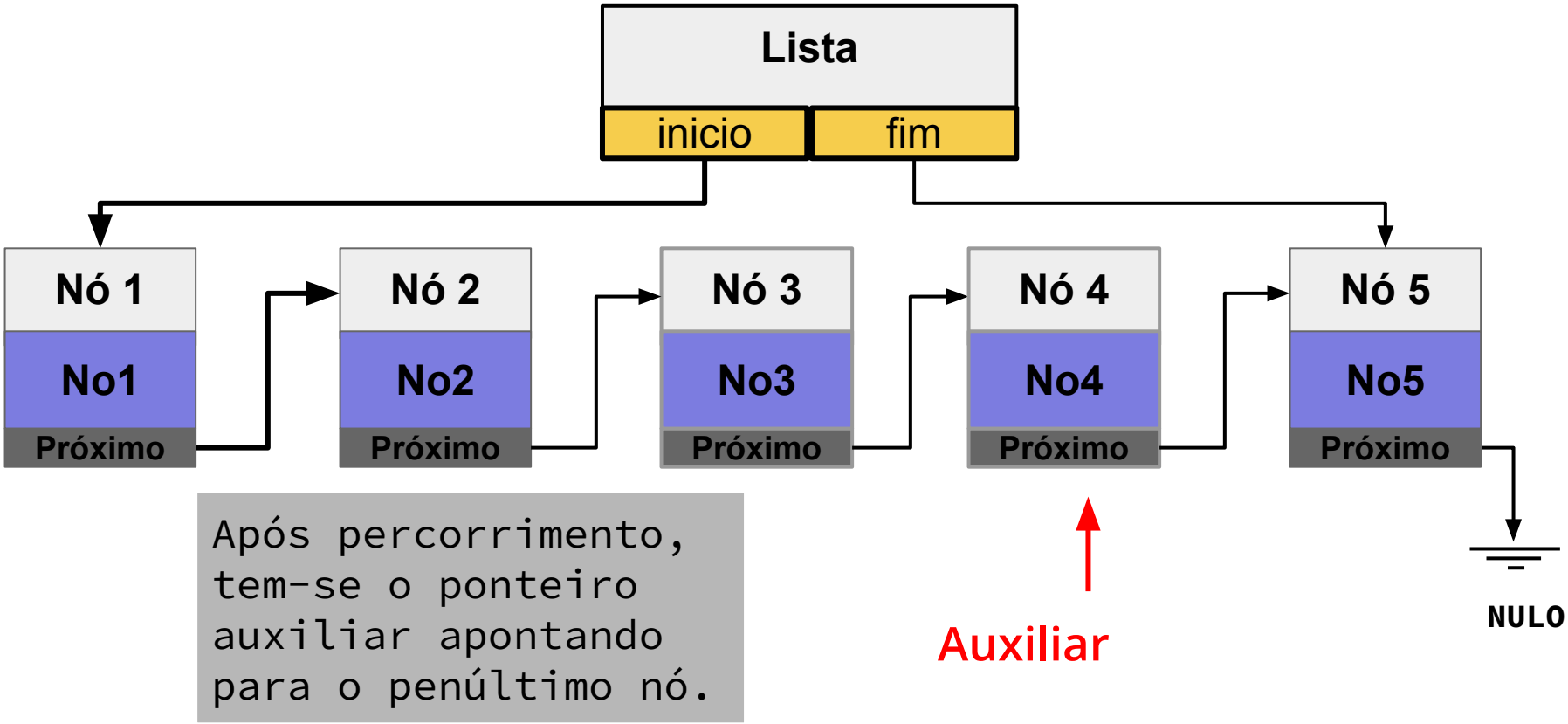


Remove no fim - II



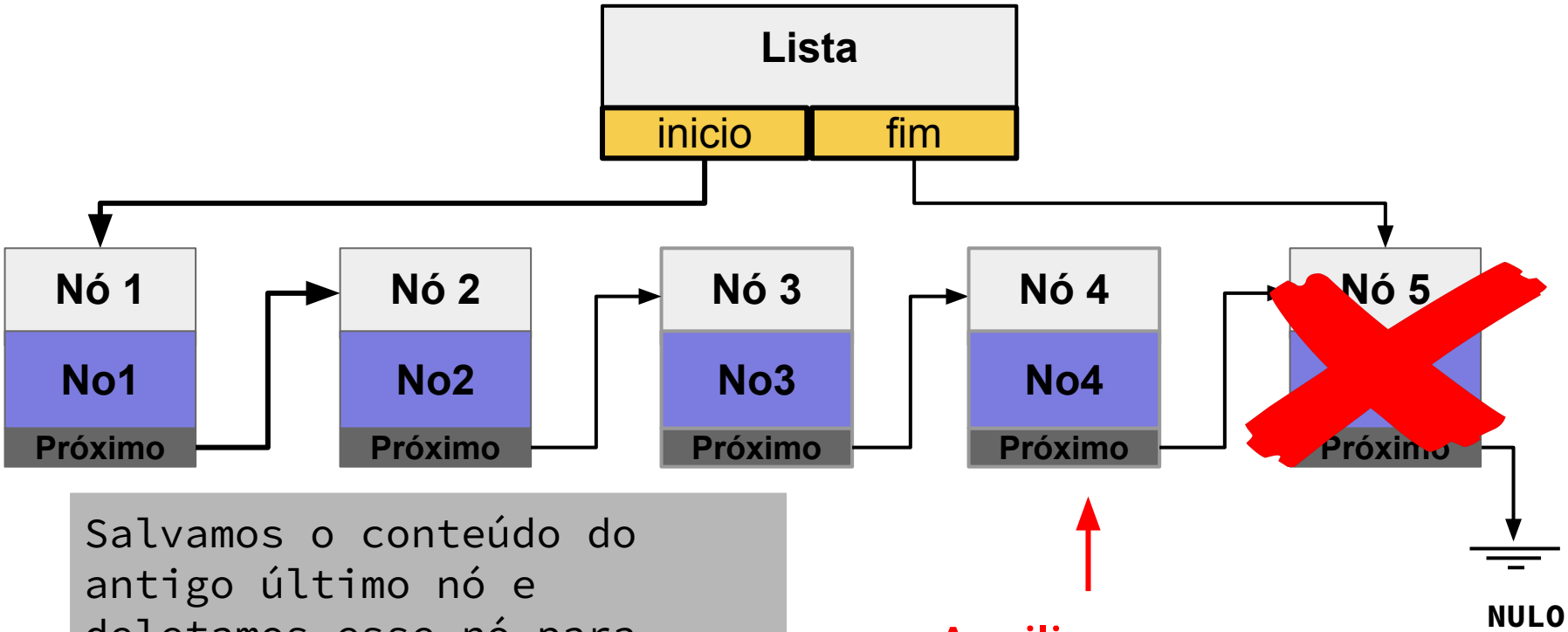
Será necessário acessar o penúltimo elemento. A menos que a lista tenha duplo encadeamento, isso implica em percorrer a lista, usando um ponteiro auxiliar, até chegar ao penúltimo nó.

Remove no fim - III



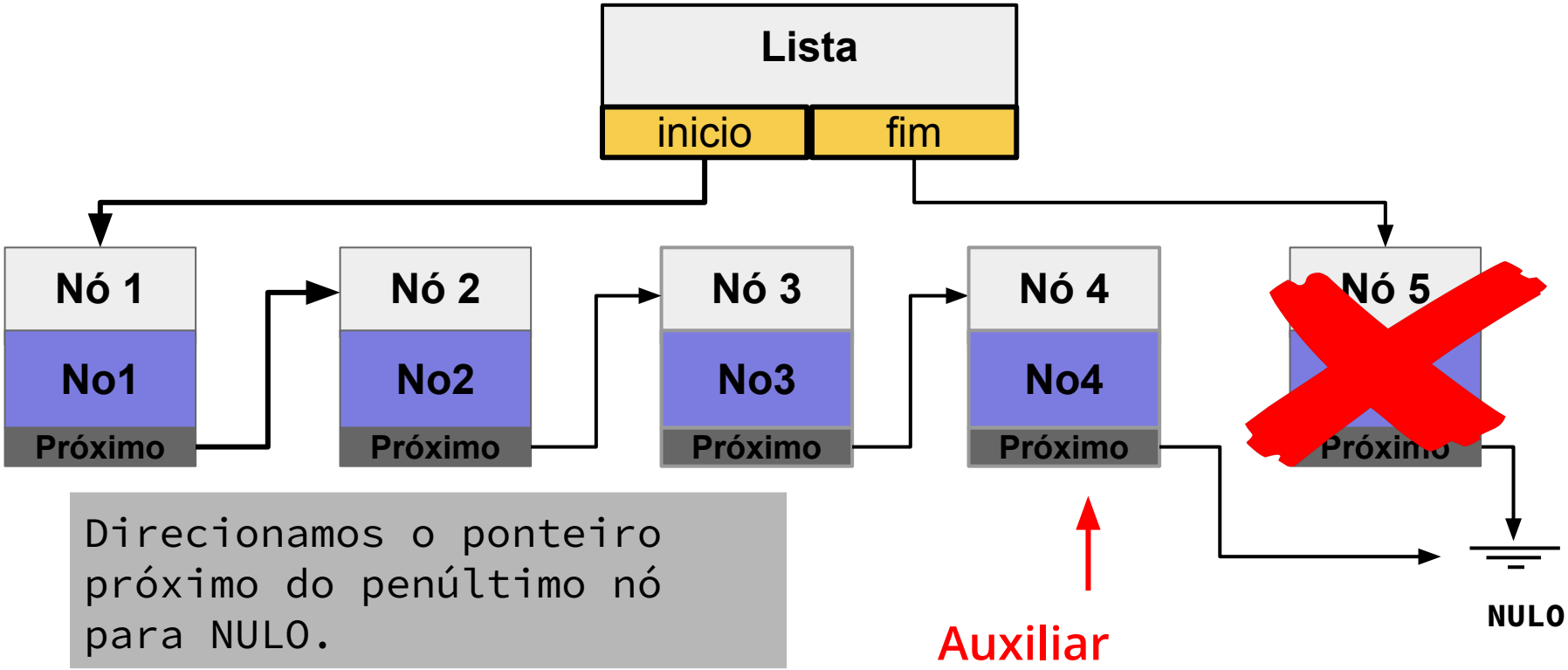
Após percorrimento, tem-se o ponteiro auxiliar apontando para o penúltimo nó.

Remove no fim - iv



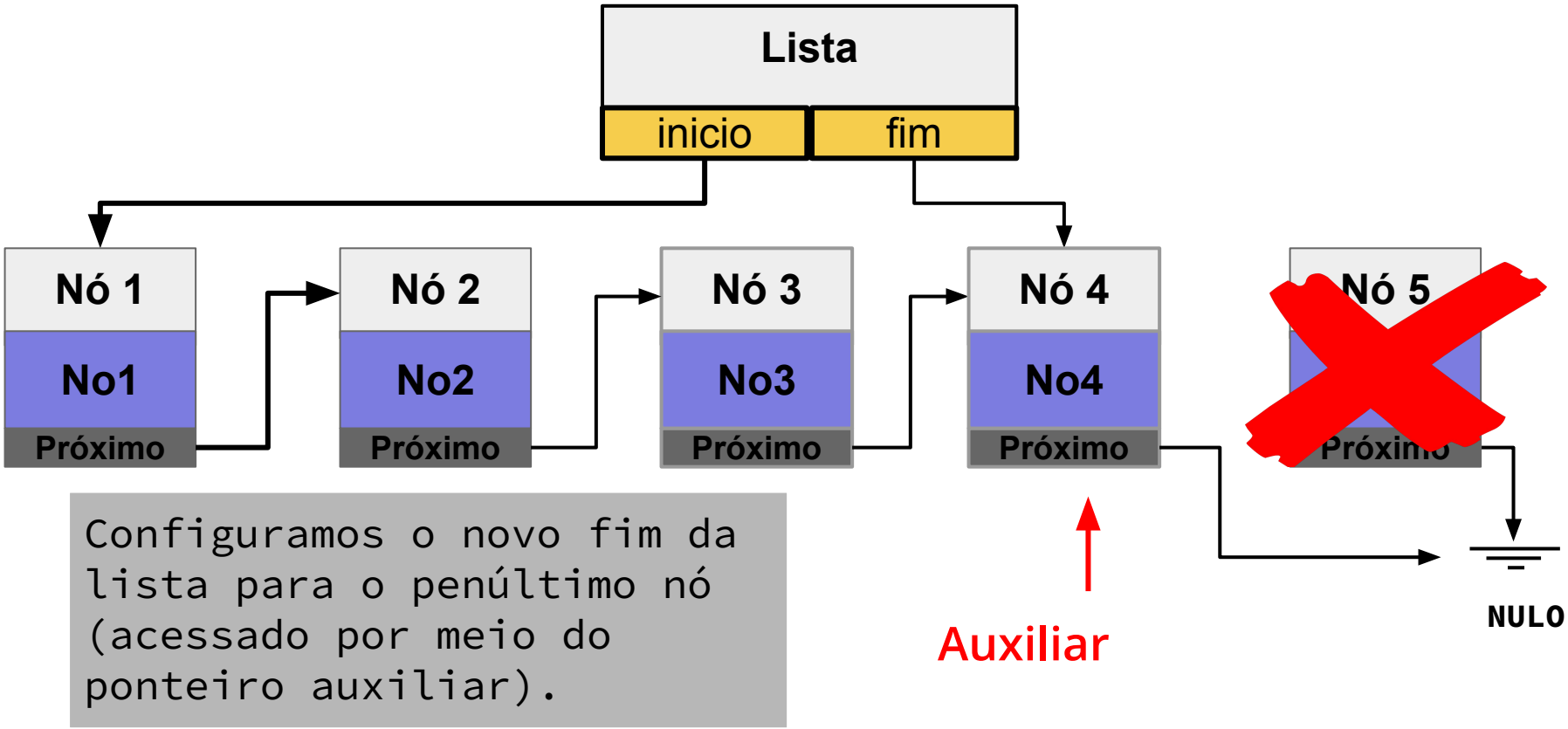
Salvamos o conteúdo do antigo último nó e deletamos esse nó para liberar espaço na memória.

Remove no fim - V



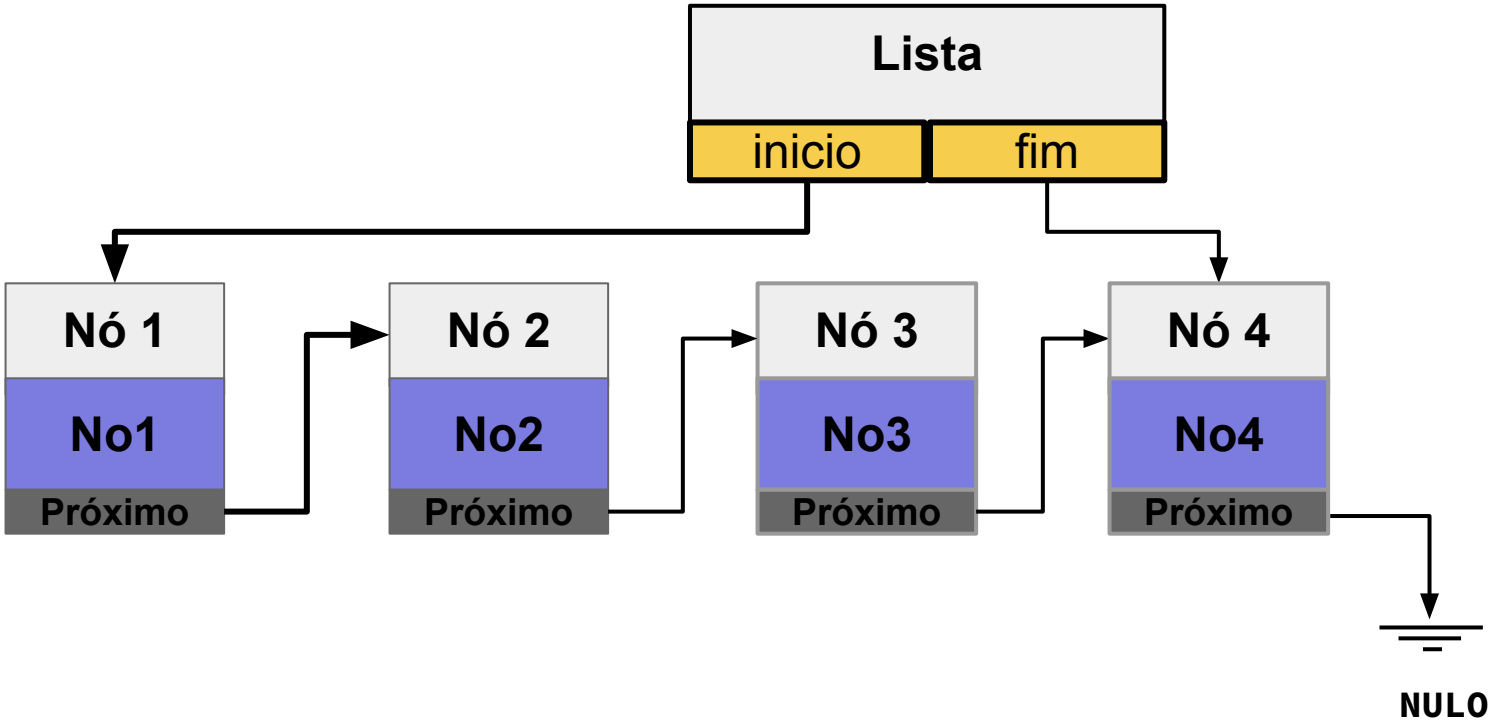
Direcionamos o ponteiro próximo do penúltimo nó para NULO.

Remove no fim - VI



Configuramos o novo fim da lista para o penúltimo nó (acessado por meio do ponteiro auxiliar).

Remove no fim - VII



Remove no fim - Observações

Caso a remoção seja chamada em uma lista vazia, é necessário gerar erro para a aplicação. O ideal é que isso seja feito usando tratamento de exceções.

Caso a lista fique vazia após remover o último (e primeiro) elemento, então é necessário fazer o ponteiro início apontar para NULO.

Remove no fim - Pseudocódigo (I)

removeNoFim():

```
se listaVazia() sairComErro()

aux ← inicio;
anterior ← NULO
enquanto (aux.proximo ≠ NULO) {
      anterior ← aux;
      aux ← aux.proximo
}
```

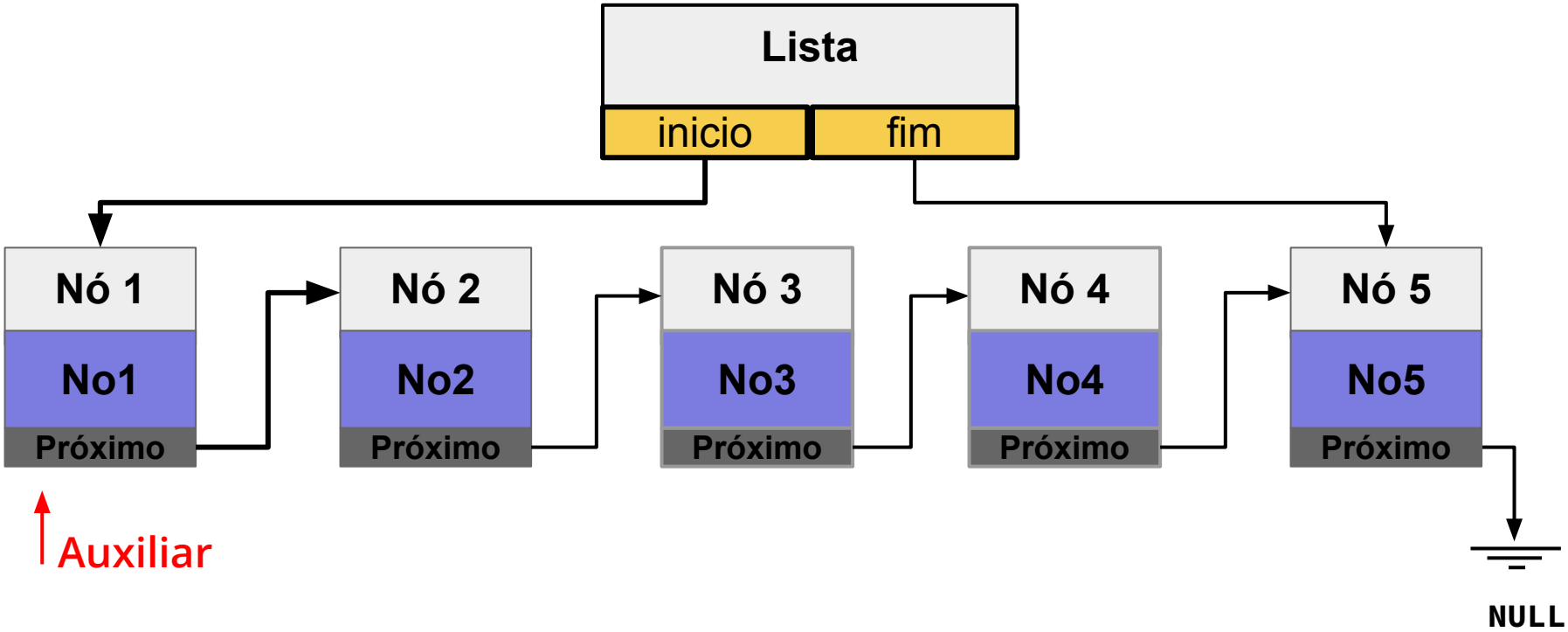
Remove no fim - Pseudocódigo (ii)

```
// paramos no último nó da lista
valor ← aux.dado;

// precisa tratar remoção de lista com um único elemento
// se anterior continua nulo, lista só tinha um elemento
se (anterior = NULO)  // poderia ser (primeiro = ultimo)
   primeiro ← NULO;
senão
   anterior.proximo ← NULO;
```

Remove no fim - Pseudocódigo (iii)

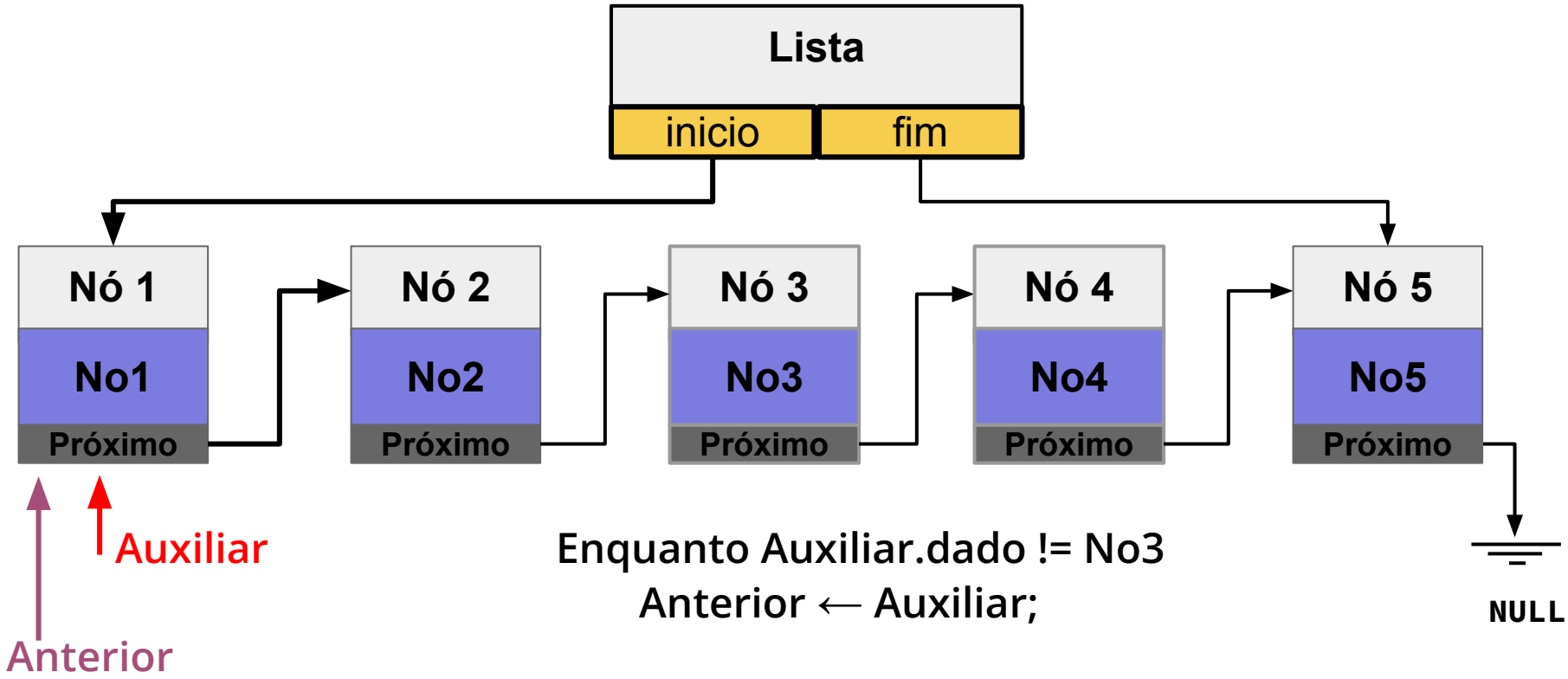
```
// remove o último nó da lista
apagar(fim);
fim ← anterior
tamanho--;

// faz ação desejada (e.g.: retorno)
efetuaAcao(valor);
```

Remove Nó específico - 1

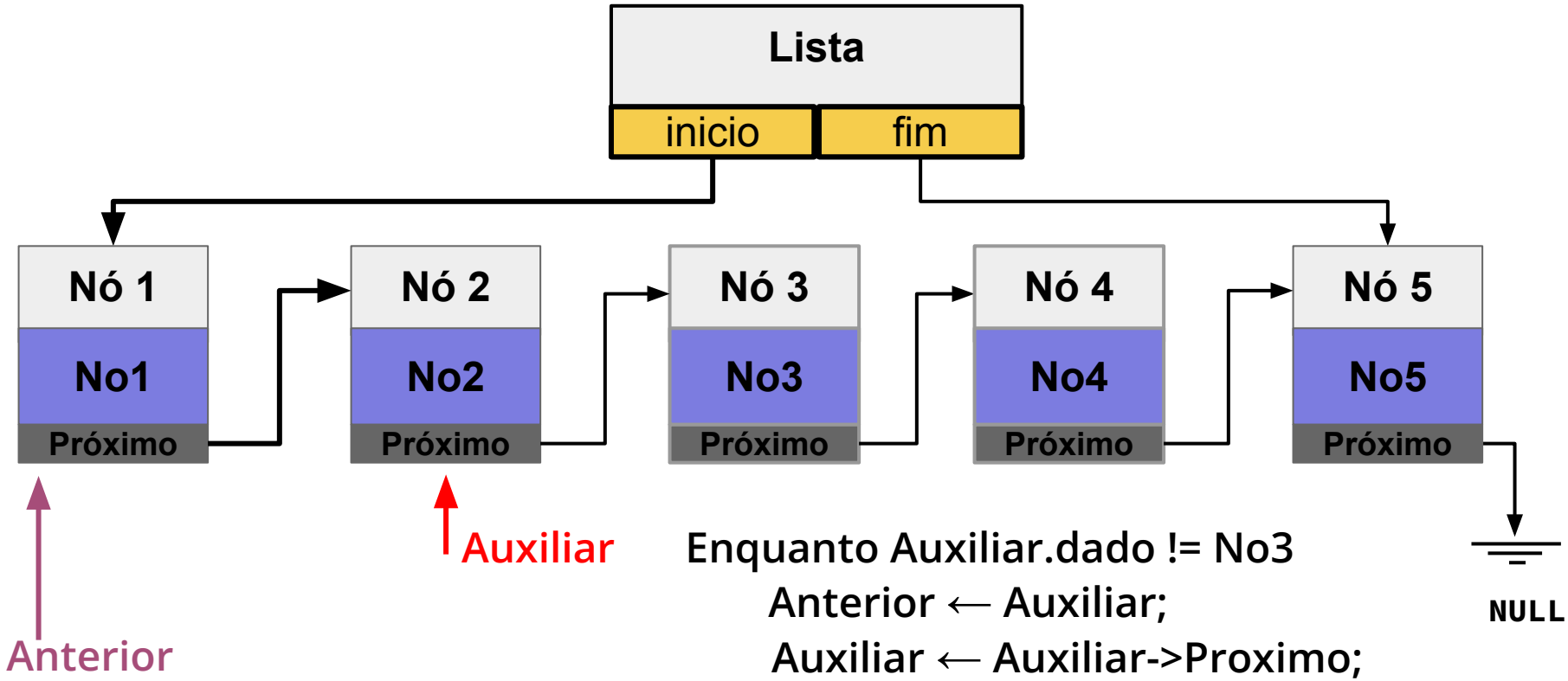


Valor a ser removido → No3

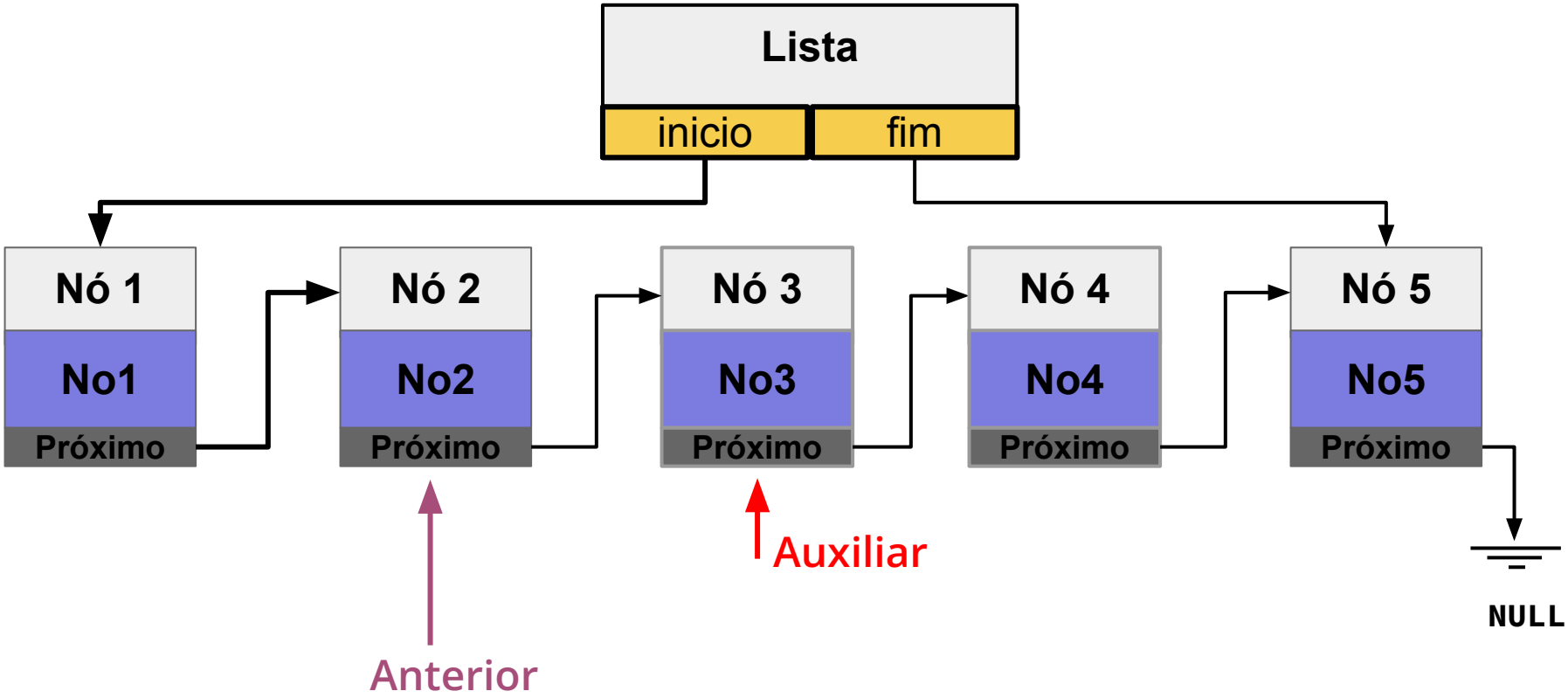
Remove Nó específico - II



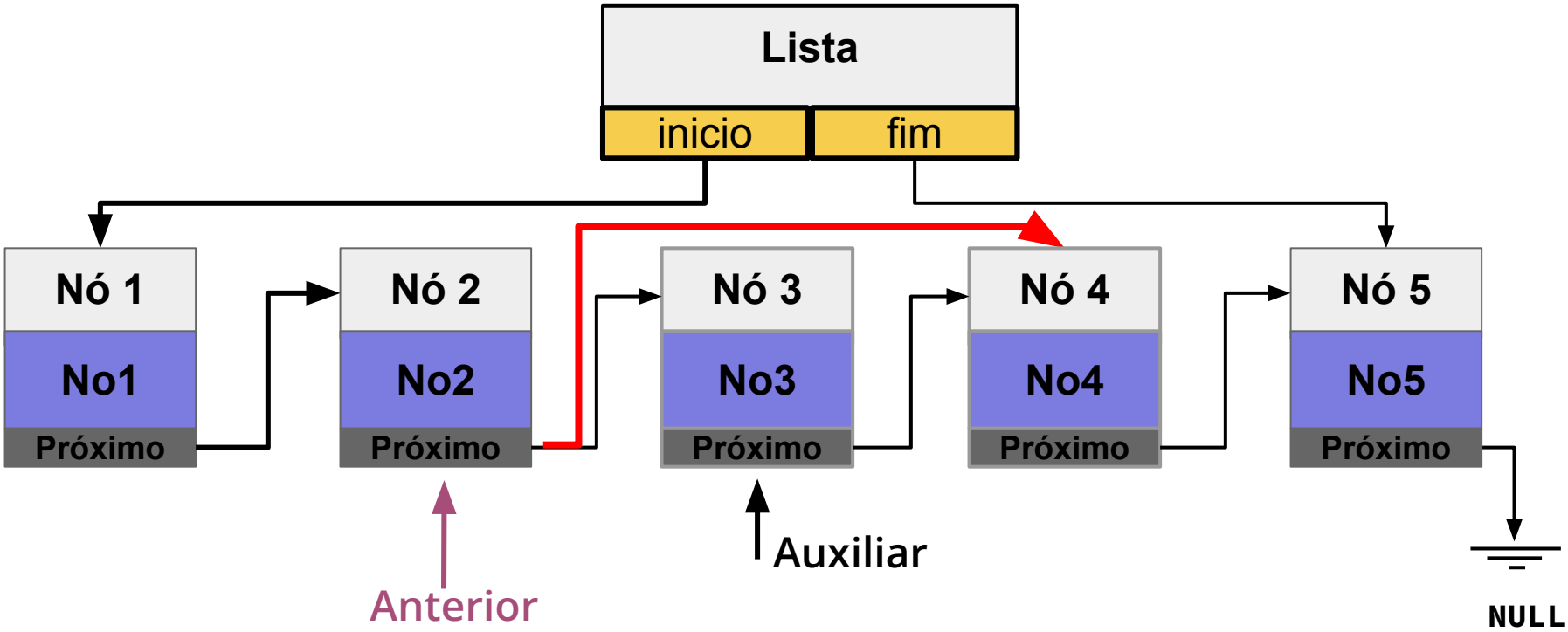
Remove Nó específico - III



Remove Nó específico - IV

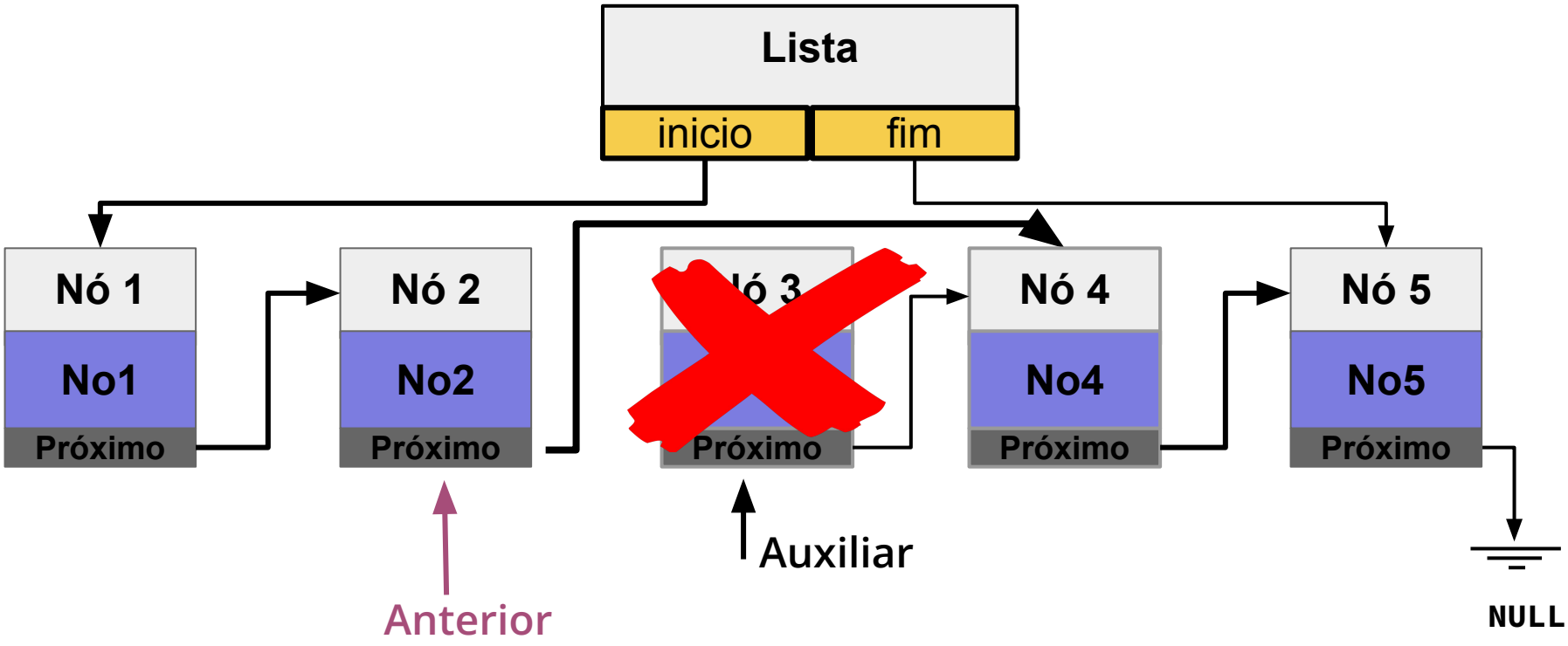


Remove Nó específico - V



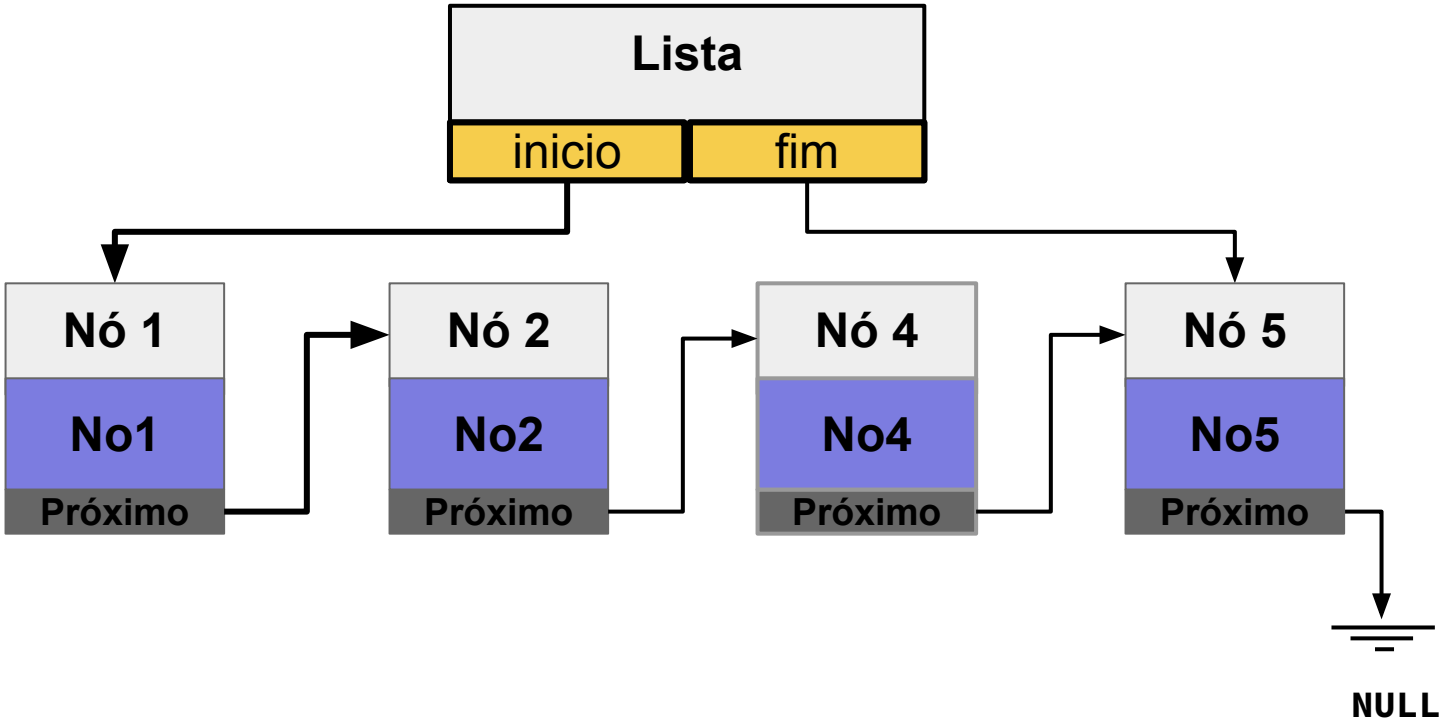
Anterior->Proximo ← Auxiliar->Proximo;

Remove Nó específico - VI



apagar(Auxiliar)

Remove Nó específico - VII



Remove nó específico- Observação

A remoção de um nó específico pode ser feita por valor ou posição. Mostraremos a remoção por valor, sendo que ela pode ser modificada para remover uma posição específica, alterando a forma de busca.

A remoção na primeira posição precisa ser tratada separadamente. Caso a lista também tenha um ponteiro para o último elemento, a remoção da última posição também deve ser tratada separadamente.

Remove nó específico - Pseudocódigo - I

removeNoEspecifico(valor):

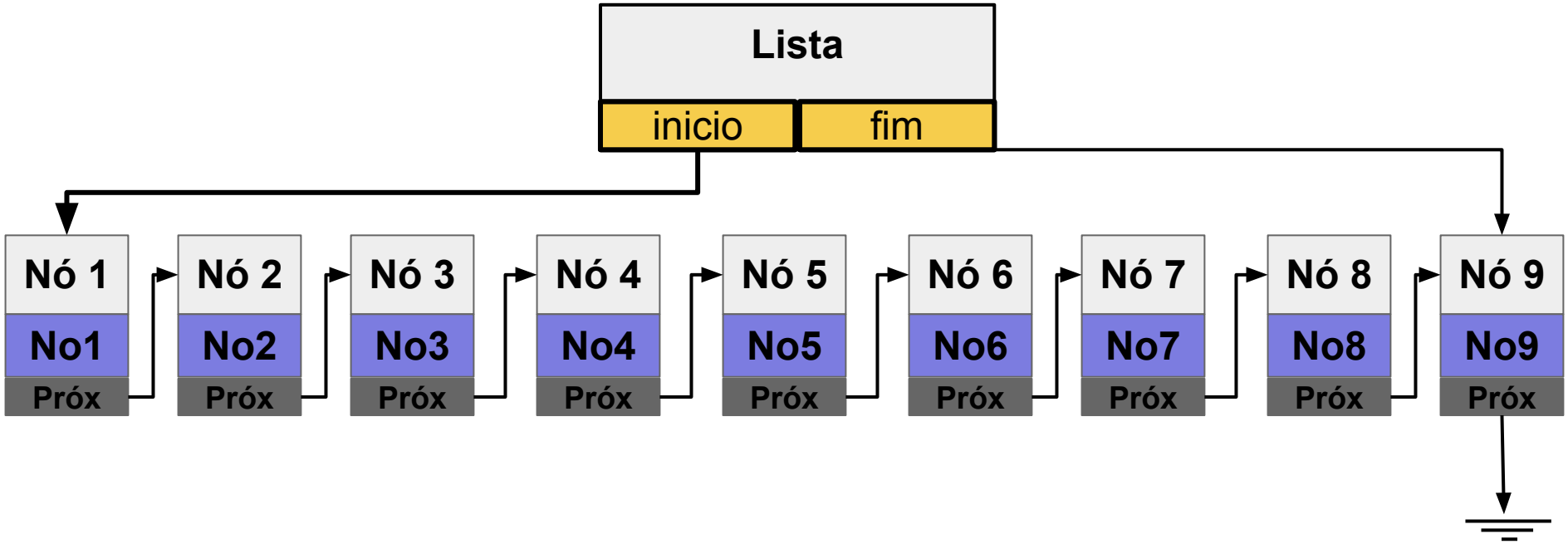
```
// caso seja necessário retorno, é necessário
// pegar o valor do nó sendo removido antes do término
// -> este código apenas remove

se (listaVazia()) sairComErro();

auxiliar← inicio;
enquanto ((auxiliar != NULO) E (auxiliar.dado != valor)){
    anterior ← auxiliar;
    auxiliar ← auxiliar.proximo;
}
```

Remove nó específico - Pseudocódigo - II

```
se (auxiliar != NULO){
    se (auxiliar == inicio){
        removeNoInicio();
    } senao se (auxiliar == fim){
        removeNoFim()
    } senao {
        anterior.proximo ← auxiliar.proximo;
        tamanho--;
        apaga(auxiliar);
    }
} senão sairComErro(“valor não está na lista!”);
```

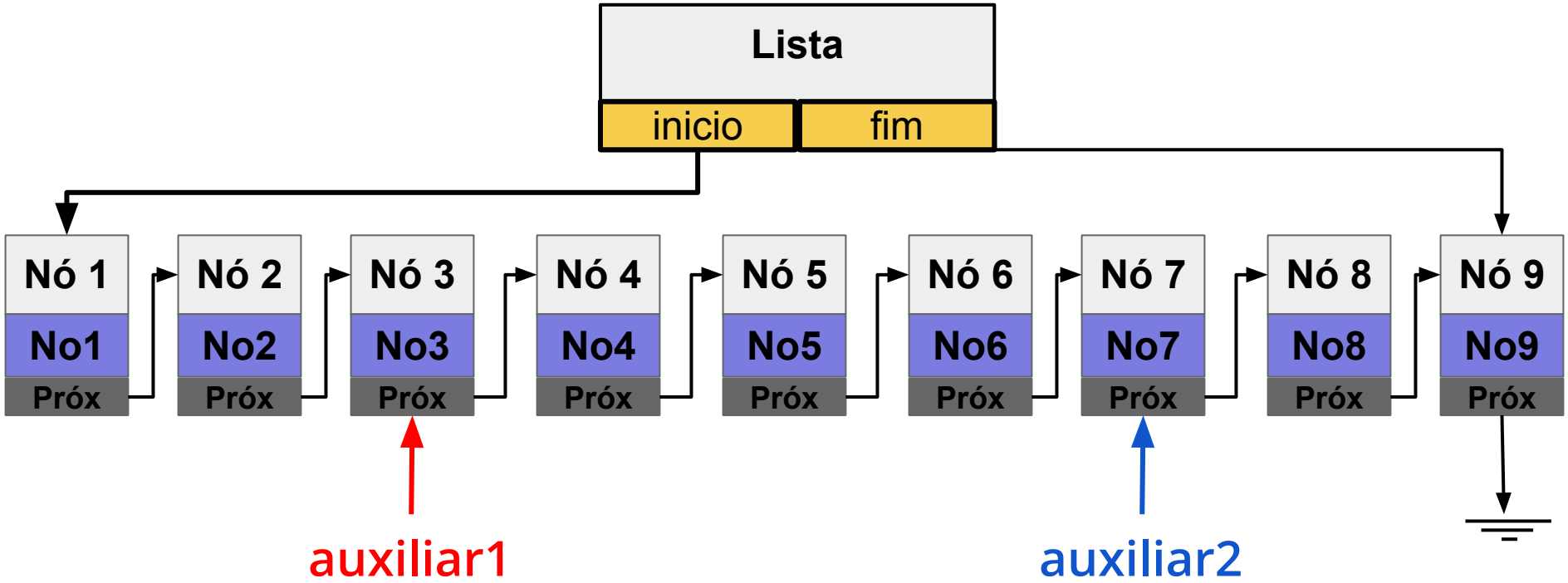
Exemplos Adicionais

Trocar dois elementos de posição - I

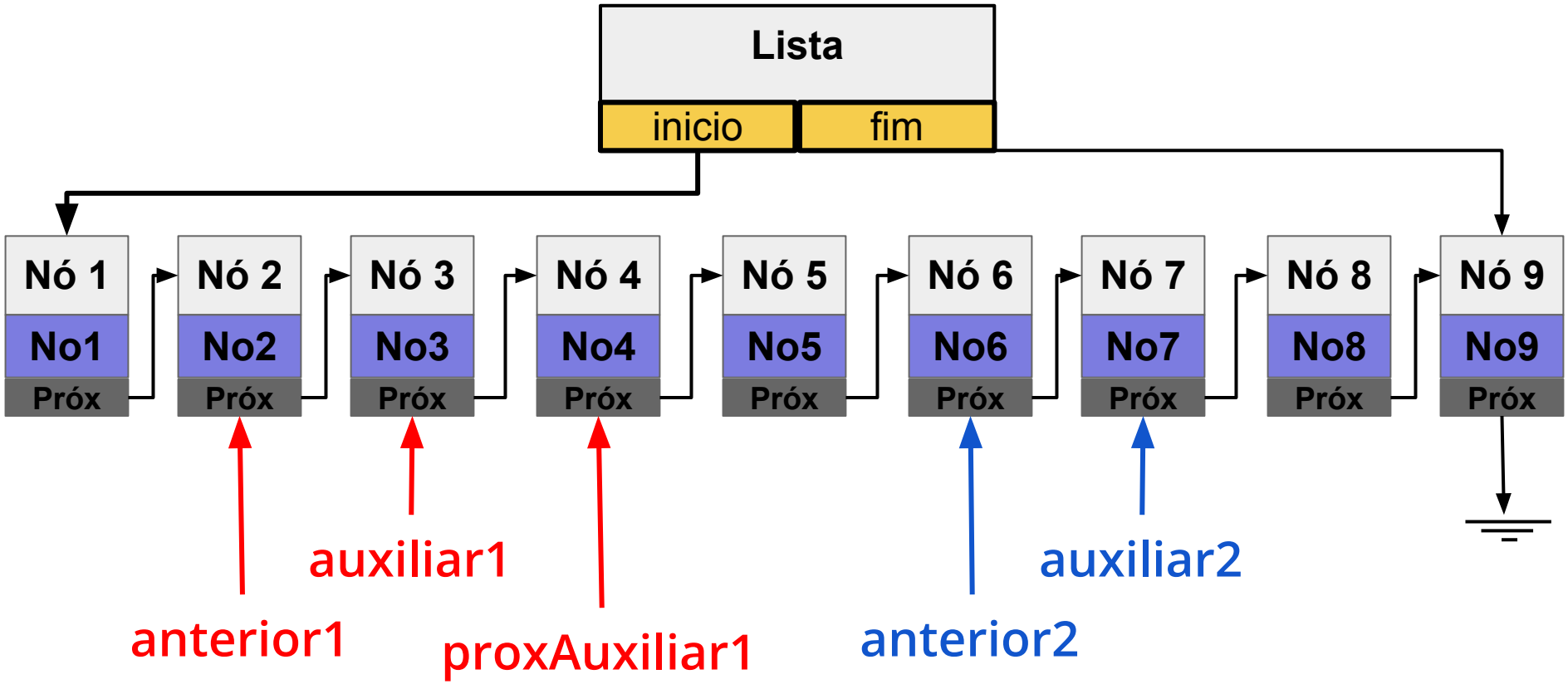


Trocar os nós 3 e 7 de posição

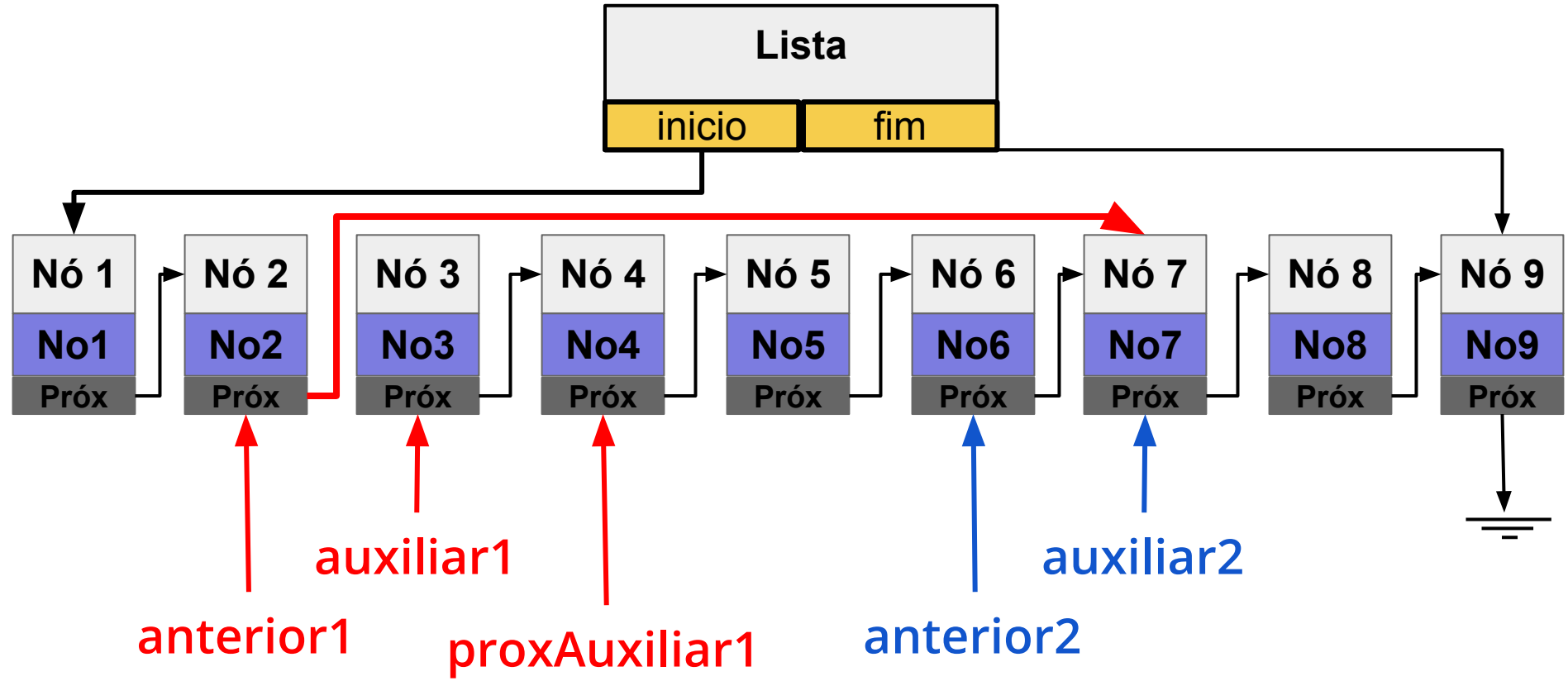
Trocar dois elementos de posição - II



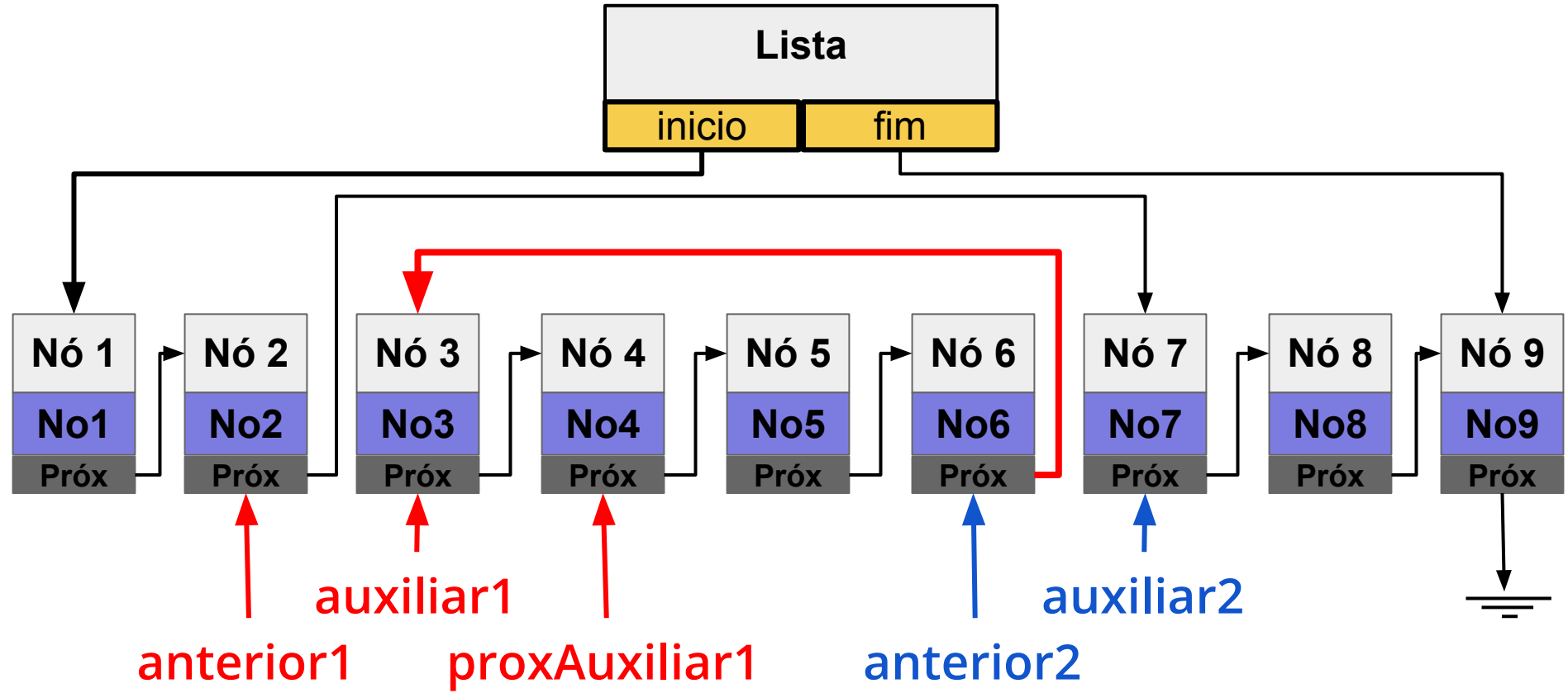
Trocar dois elementos de posição - III



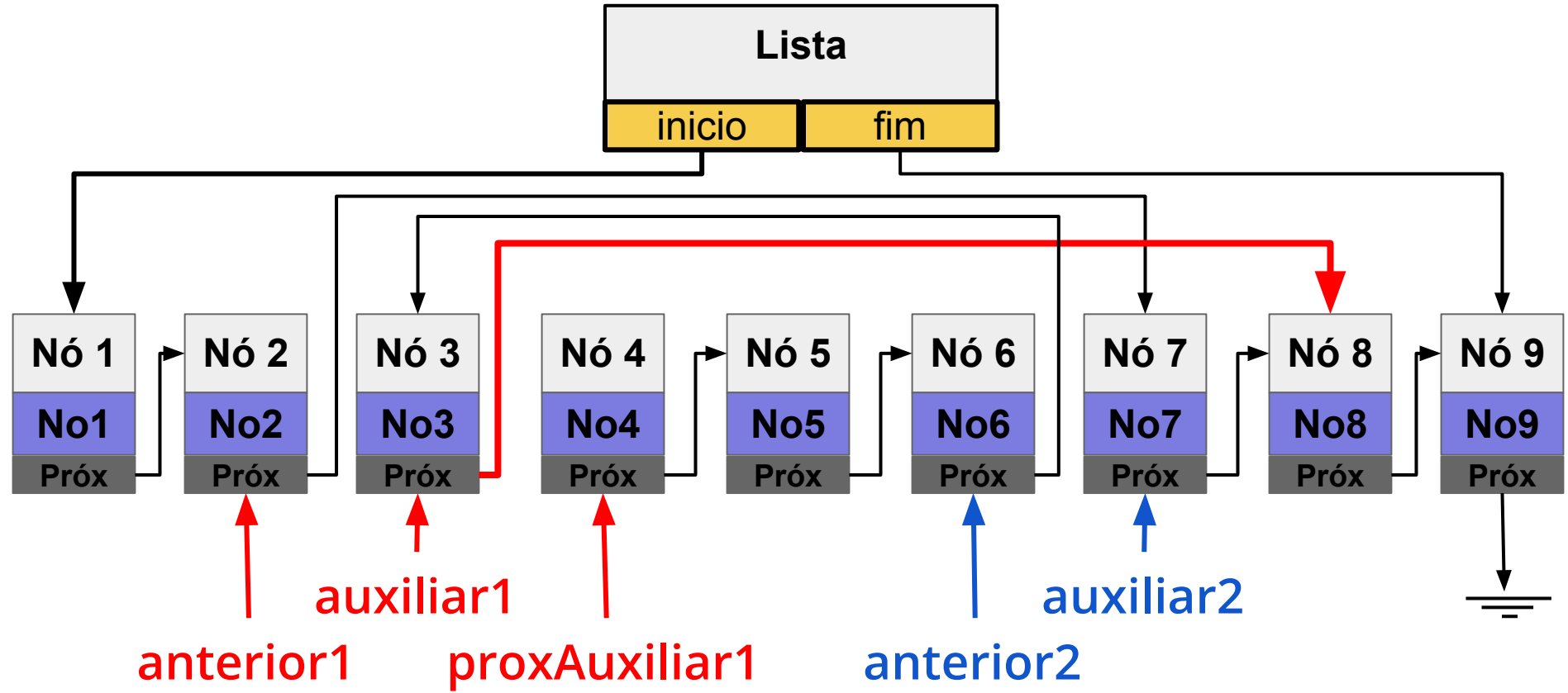
Trocar dois elementos de posição - IV



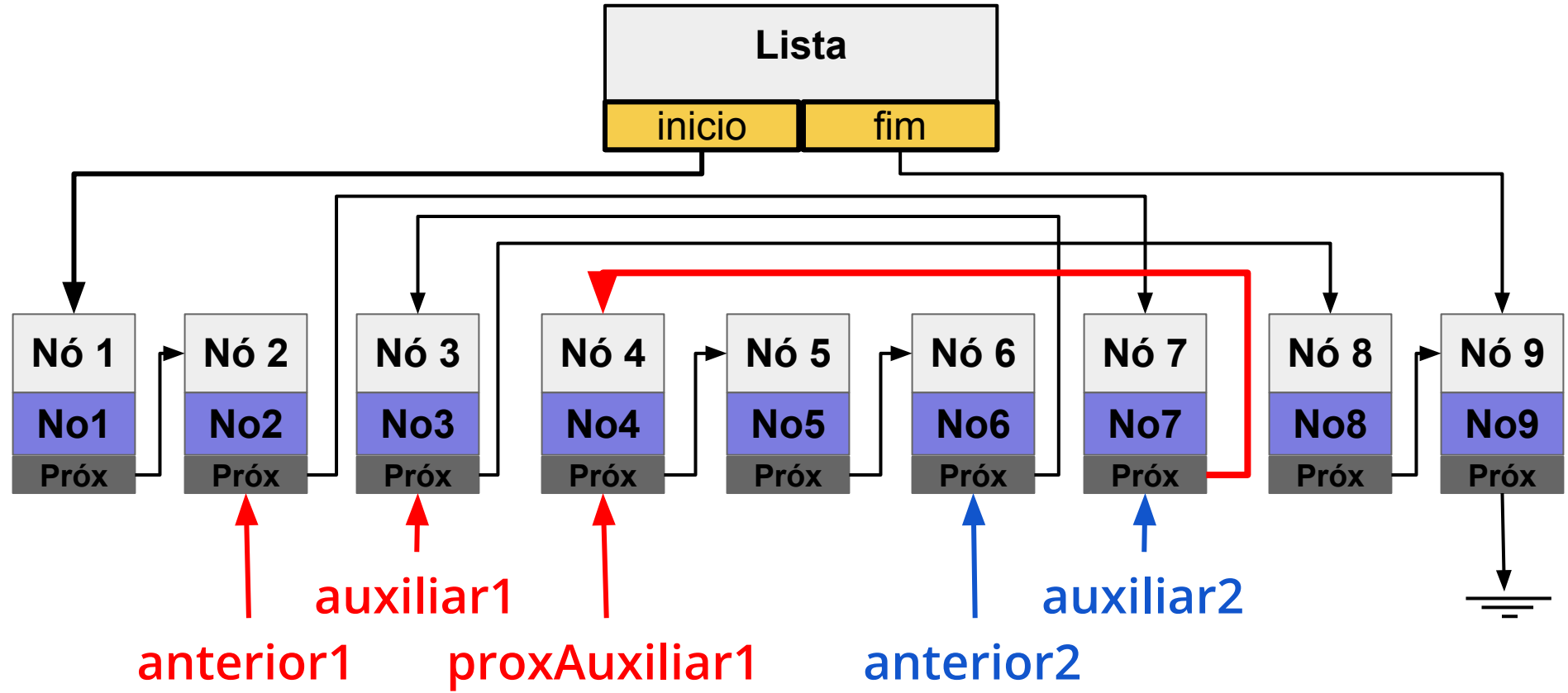
Trocar dois elementos de posição - V



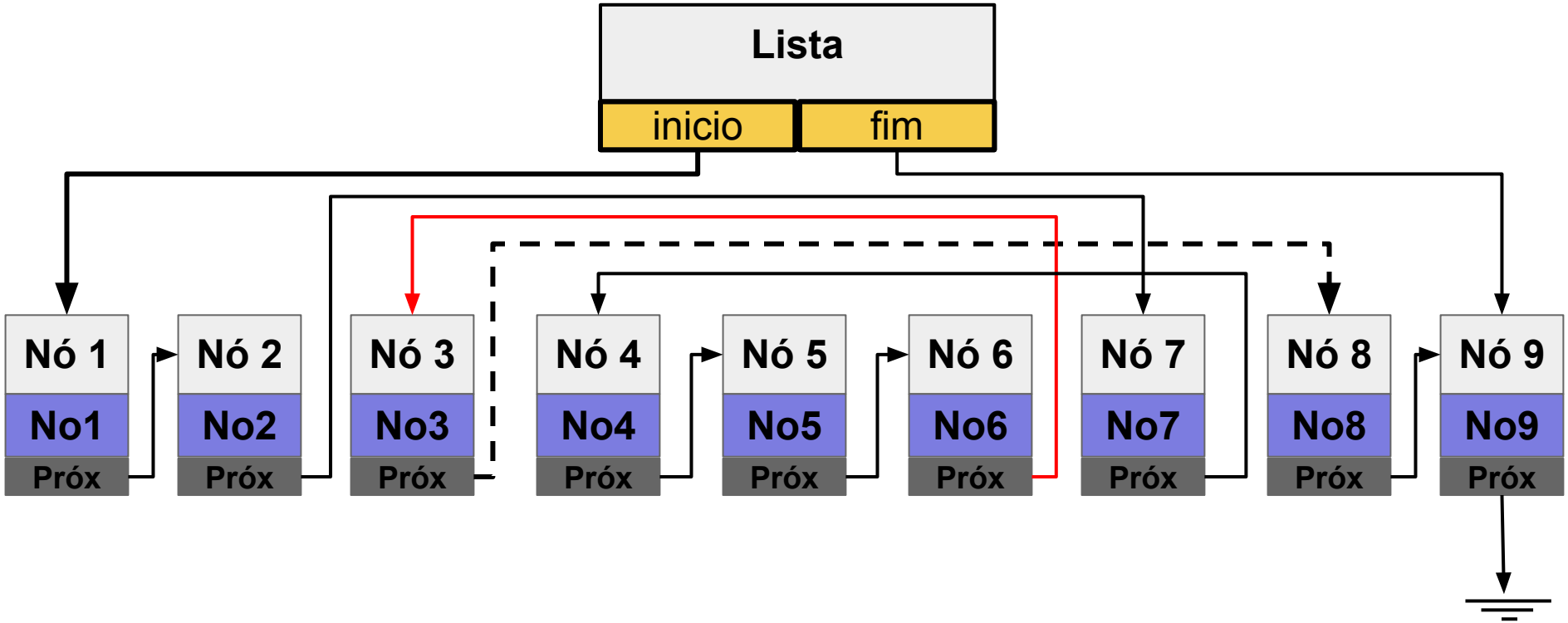
Trocar dois elementos de posição - VI



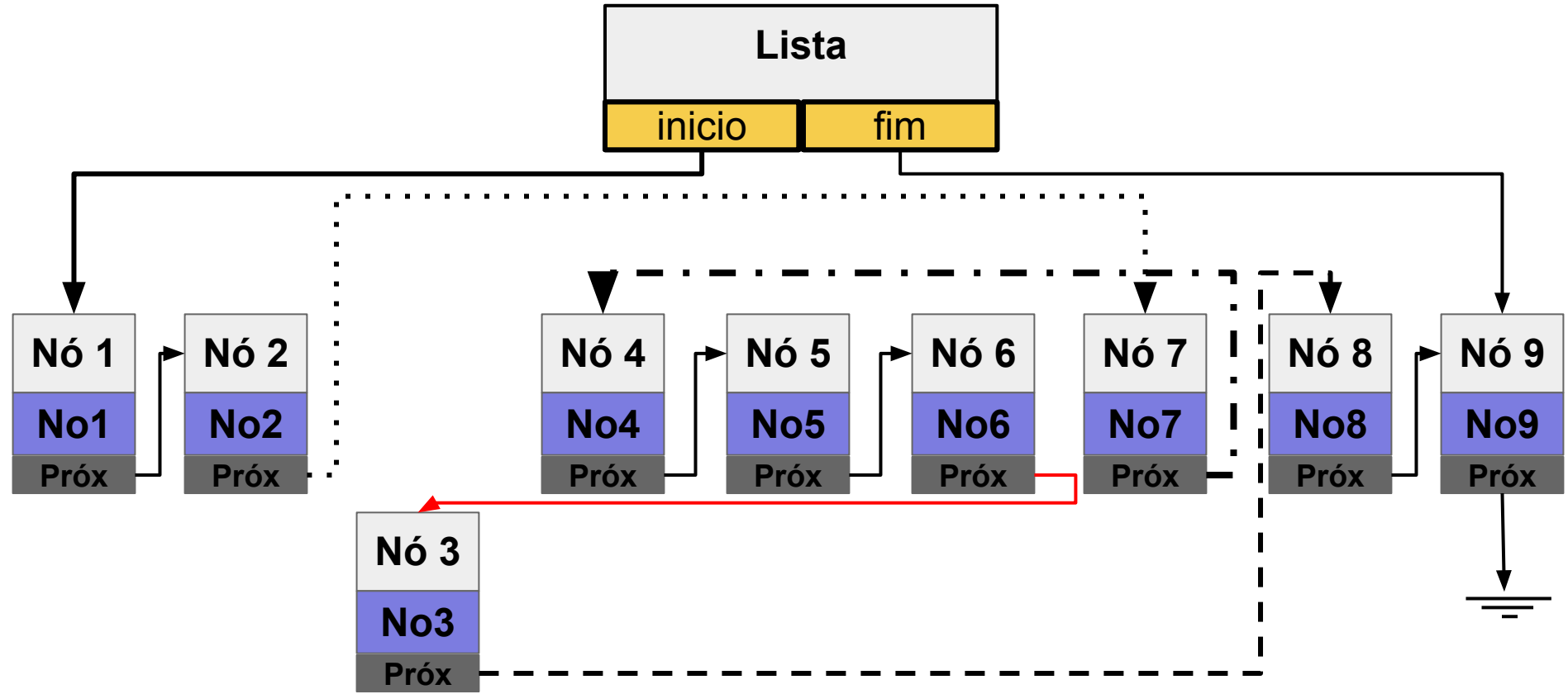
Trocar dois elementos de posição - VII



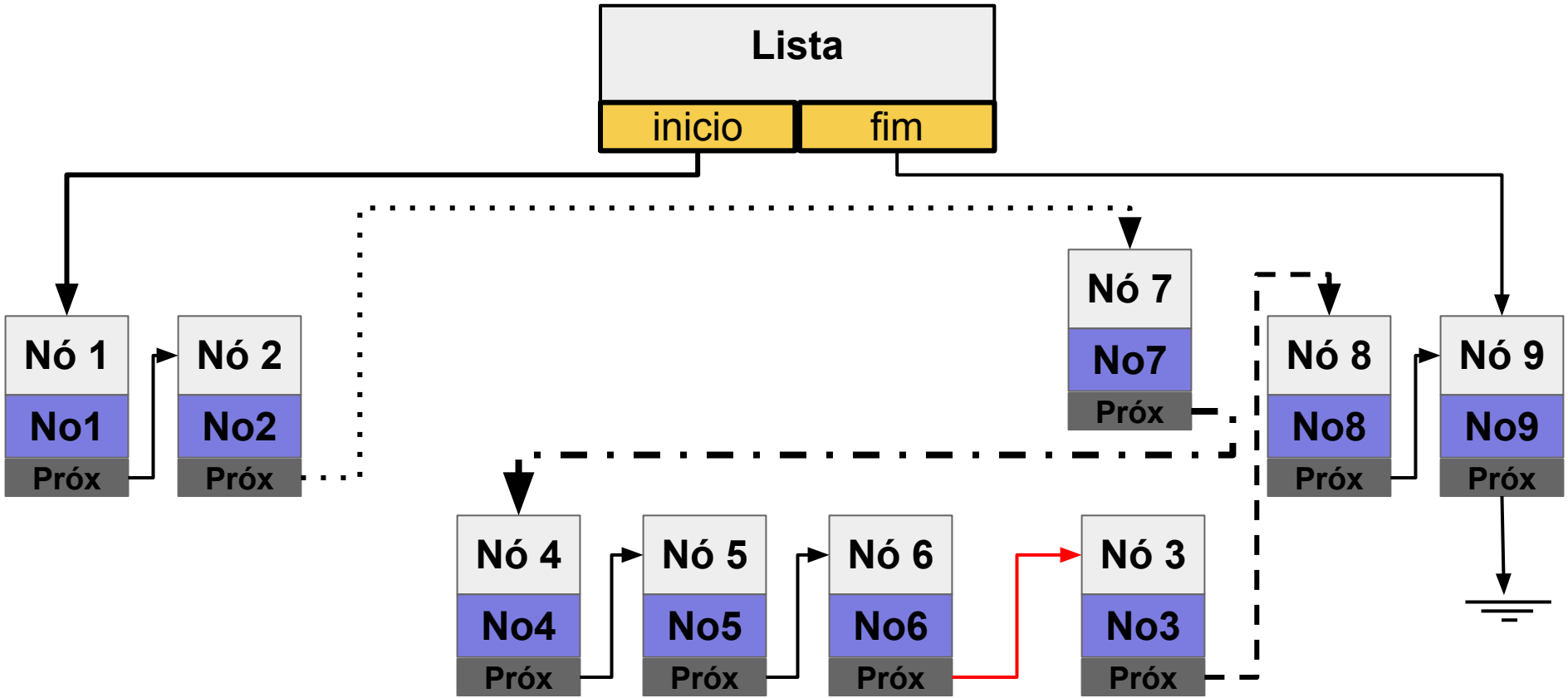
Trocar dois elementos de posição - VIII



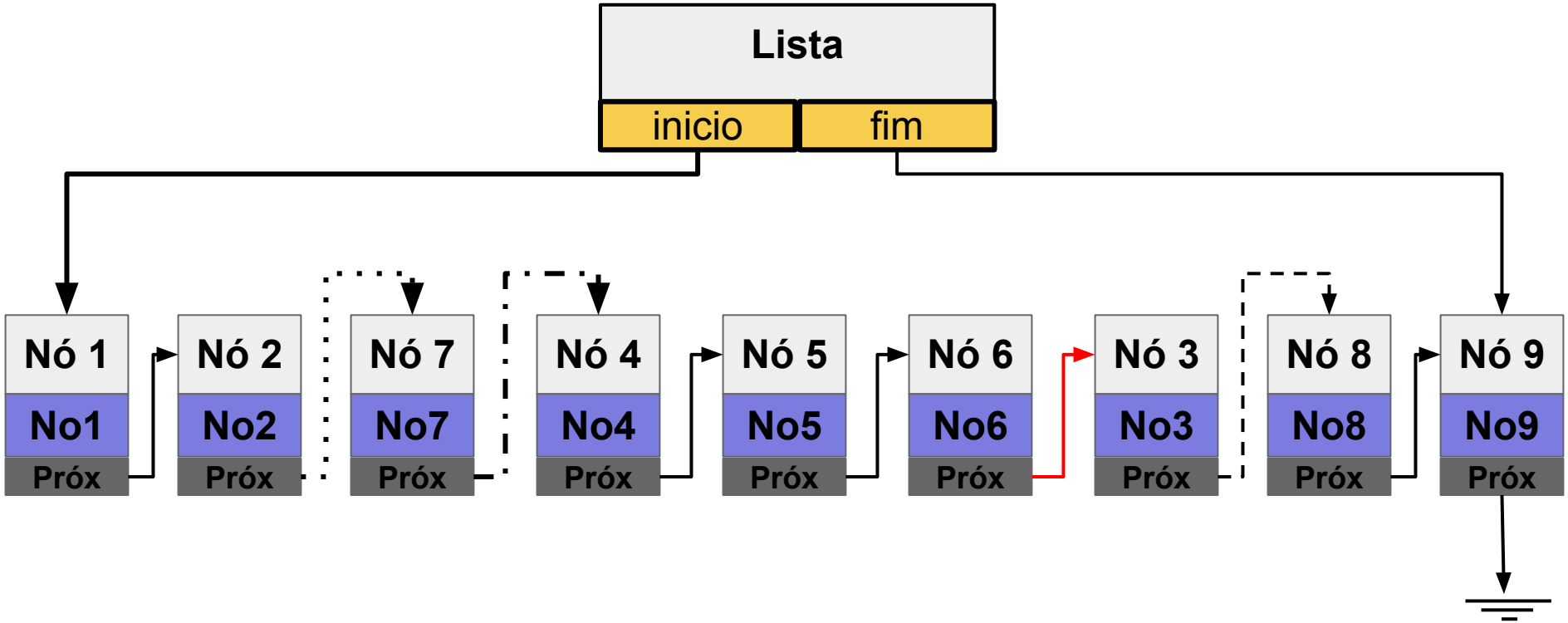
Trocar dois elementos de posição - IX



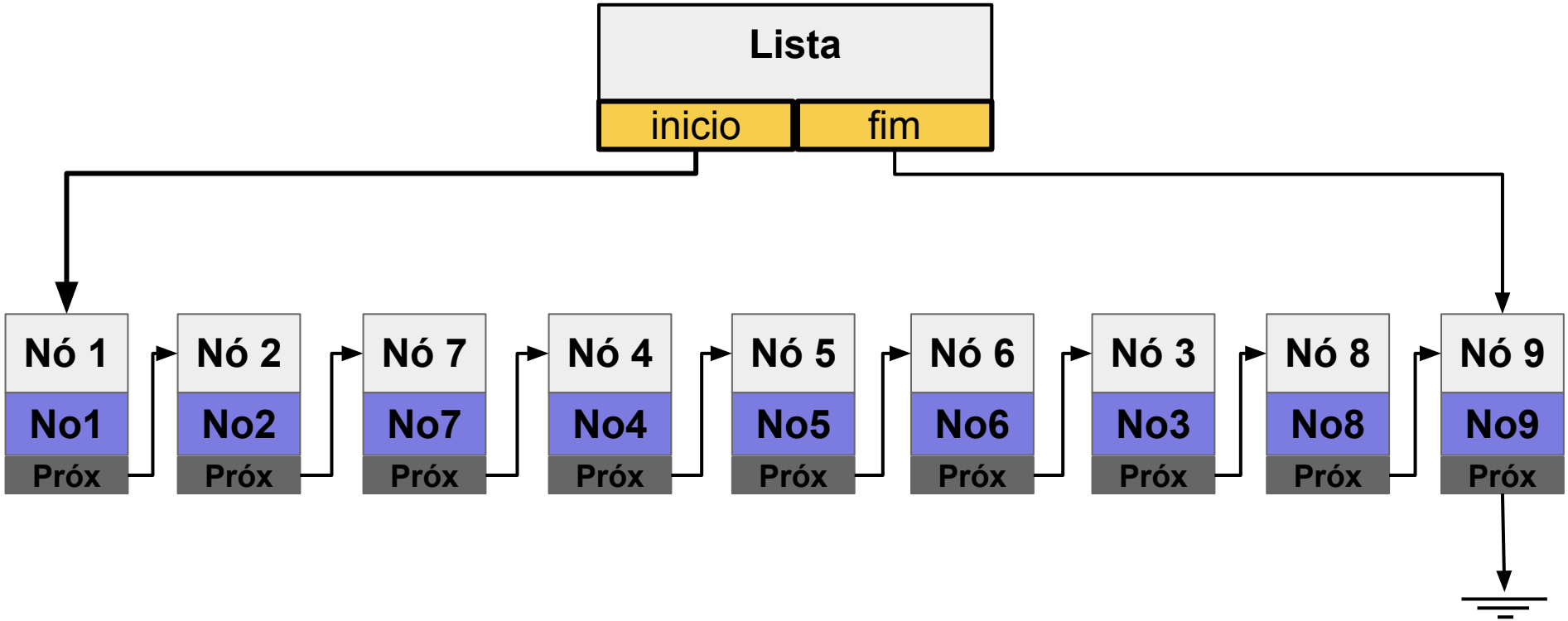
Trocar dois elementos de posição - x



Trocar dois elementos de posição - xi



Trocar dois elementos de posição - xii



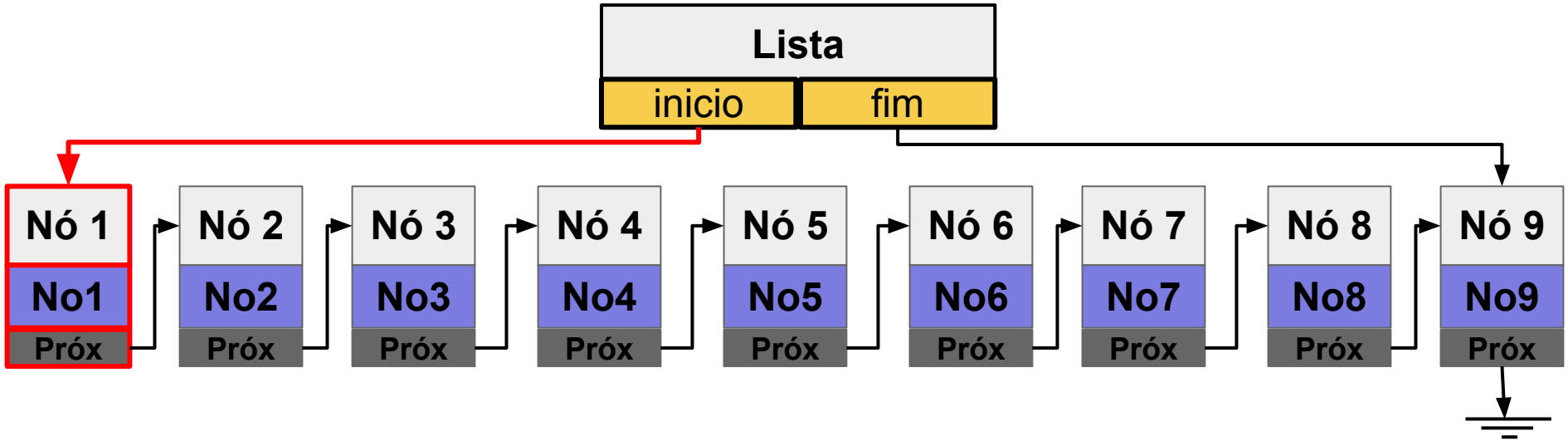
Trocar dois elementos de posição - Pseudocódigo

trocaDoisElementos(posicao1,posicao2):

```
anterior1 ← acessarPosicao(posicao1-1);
auxiliar1 ← anterior1.proximo;
anterior2 ← acessarPosicao(posicao2-1);
auxiliar2 ← anterior2.proximo;
anterior1.proximo ← auxiliar2;
anterior2.proximo ← auxiliar1;
posAuxiliar1 ← auxiliar1.proximo;
auxiliar1.proximo ← auxiliar2.proximo;
auxiliar2.proximo ← posAuxiliar1;
```

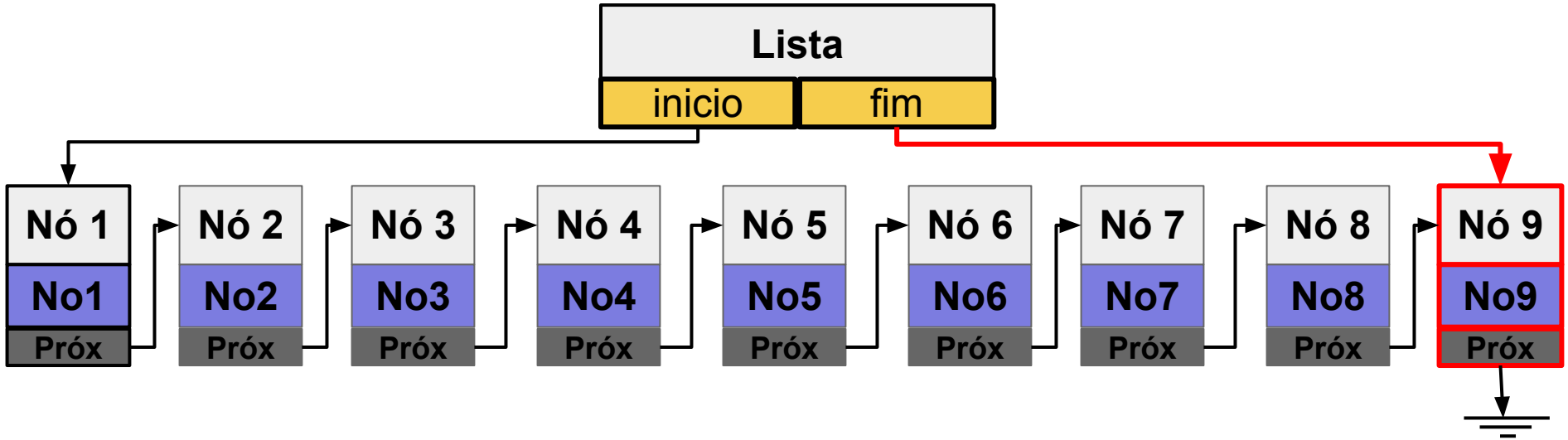
Trocar dois elementos de posição - casos específicos

- um dos elementos a ter sua posição trocada é o primeiro elemento da lista



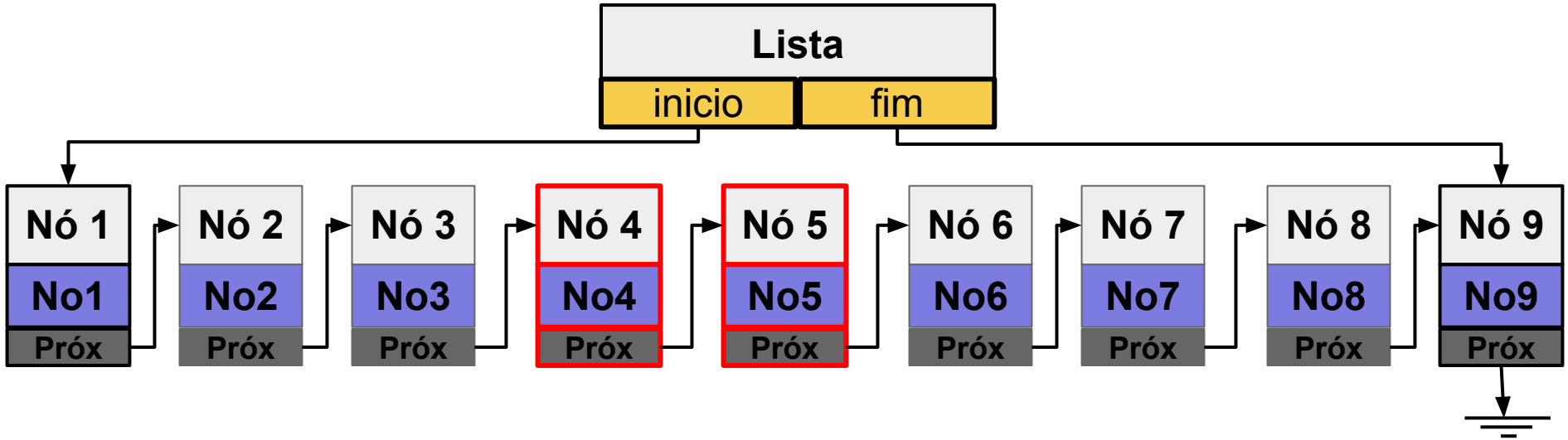
Trocar dois elementos de posição - casos específicos

- um dos elementos a ter sua posição trocada é o último elemento da lista



Trocar dois elementos de posição - casos específicos

- os elementos a terem suas posições trocadas são adjacentes



Trocar dois elementos de posição - Discussão - I

Uma questão a ser feita é porque não apenas trocar os valores dos nós. Não é incomum verificar implementação similar à seguinte:

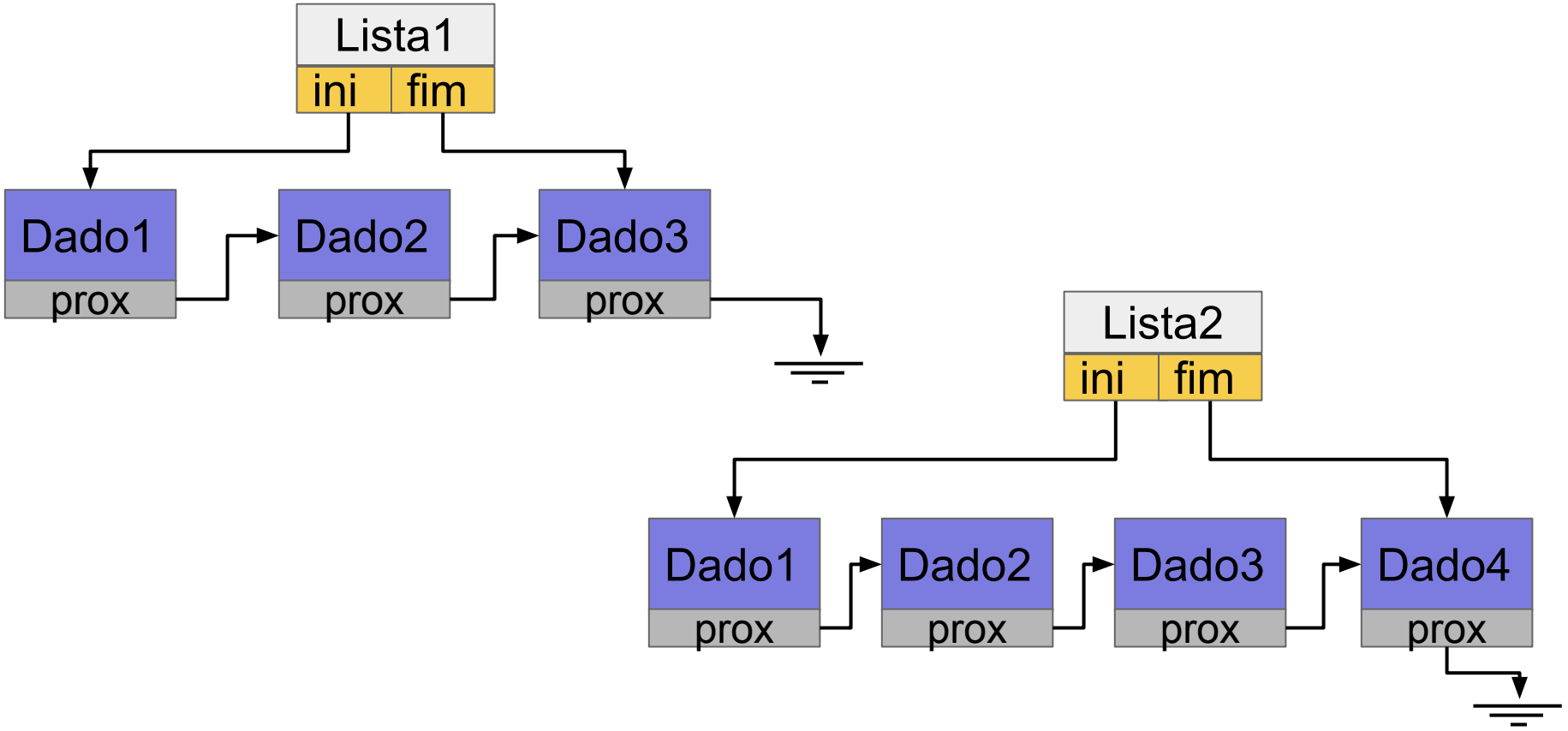
```
aux1 ← acessarPosicao(posicao1);
aux2 ← acessarPosicao(posicao2);
temp ← aux1;
aux1 ← aux2;
aux2 ← temp;
```

Trocar dois elementos de posição - Discussão - II

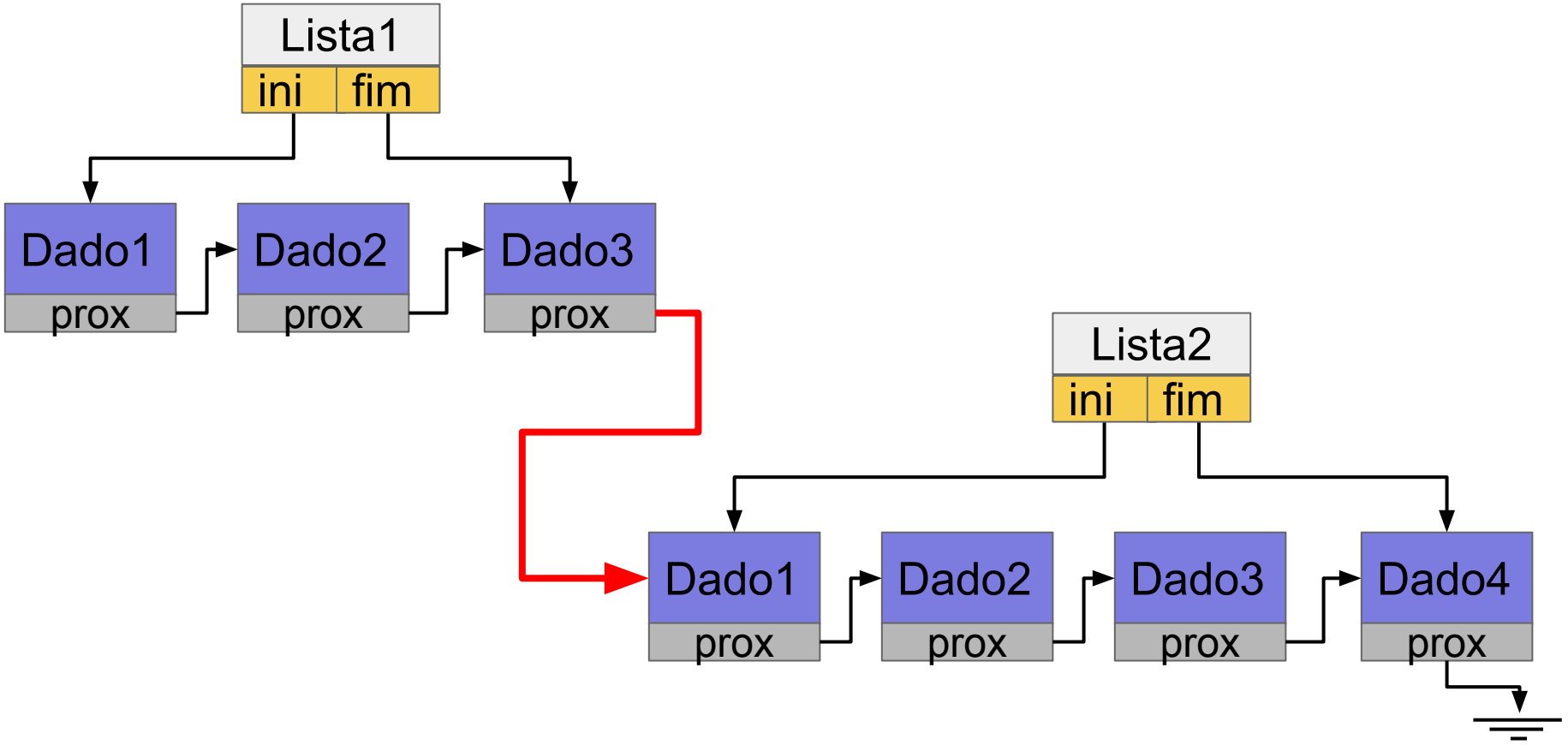
O problema com a troca de valores é que ela geralmente utiliza construtores de cópia e operação de atribuição. Caso o dado armazenado ocupe pouco espaço em memória, isso é feito de forma relativamente rápida e eficiente.

Caso do dado ocupe espaço relativamente grande em memória, a operação de cópia e atribuição irão prejudicar a performance. Assim, a troca dos nós em si é mais adequada e eficiente.

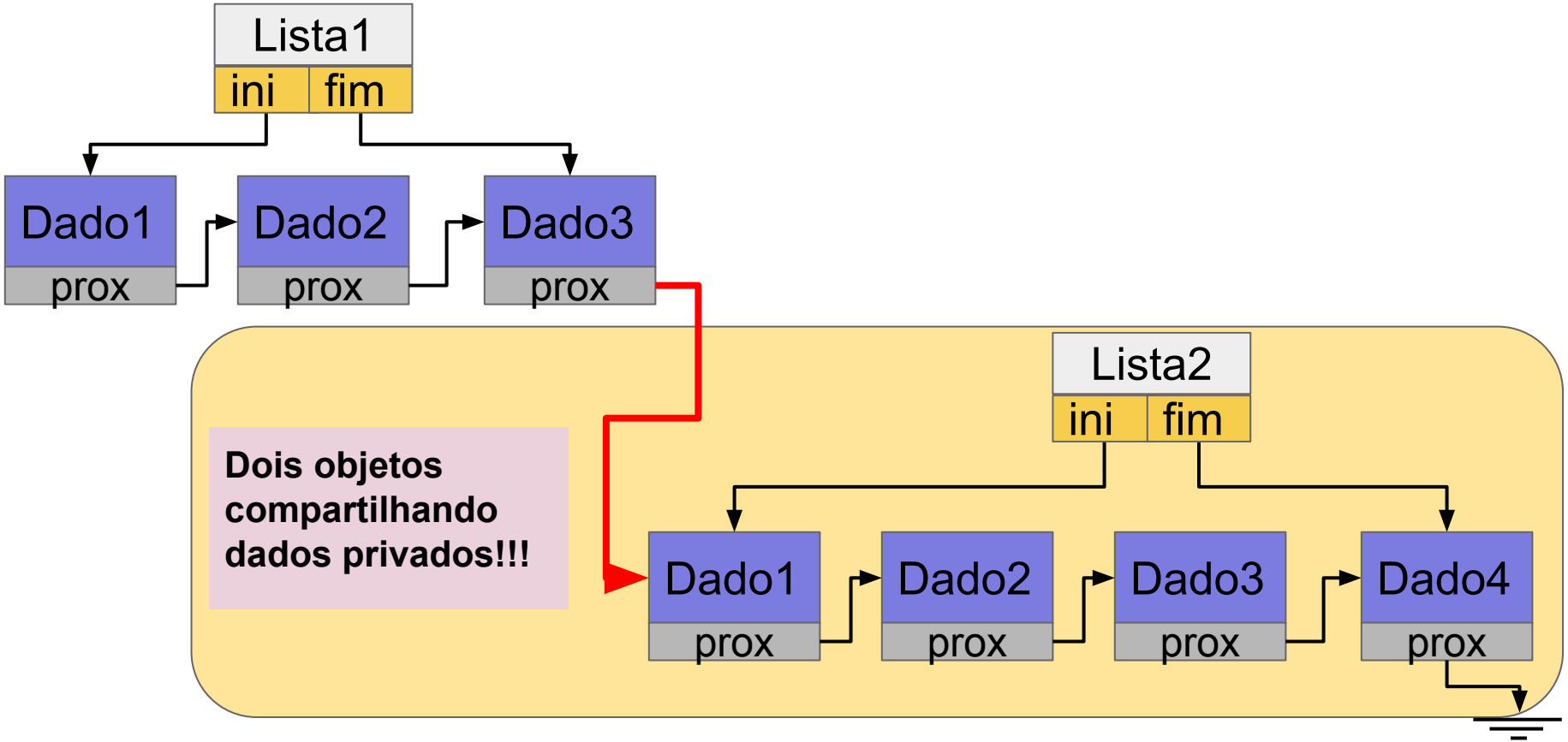
Concatenar duas listas



Concatenar duas listas - Erro comum grave



Concatenar duas listas - Erro comum grave



Concatenar duas listas

Abordagem para concatenação deve ser realizada com cópia dos elementos da segunda lista para a primeira:

- Percorrer os elementos da Lista 2
- Inserir, um a um, os elementos percorridos no final da Lista 1

Concatenar duas listas - Pseudocódigo

```
concatenarDuasLista(lista1,lista2):
auxiliar ← lista2.inicio;

enquanto (auxiliar ≠ NULO) {
    novo ← copiar_noh(auxiliar);
    lista1.fim.proximo ← novo;
    lista1.fim ← novo;
    auxiliar ← auxiliar.proximo;
}
```

Sobre o Material

Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Joaquim Quinteiro Uchôa
- Juliana Galvani Greghi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.