

ESTRUTURA DE DADOS LISTA

Prof. Joaquim Uchôa
Profa. Juliana Gregghi
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

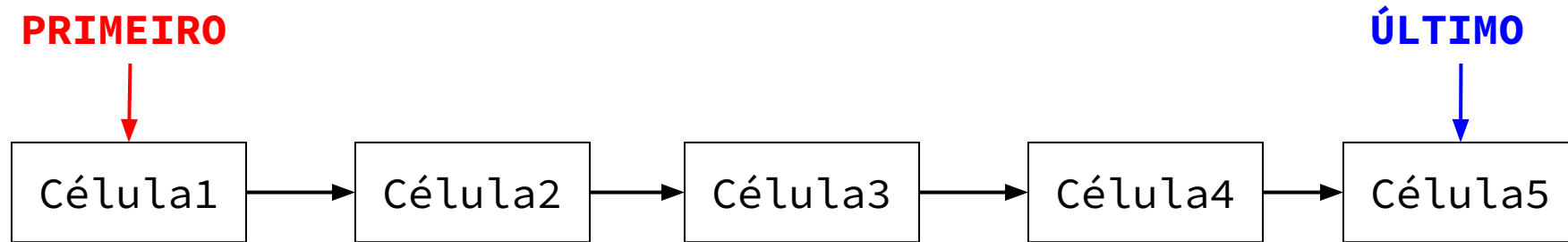
Uma lista encadeada é composta por células que apontam para o próximo elemento da lista.

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.

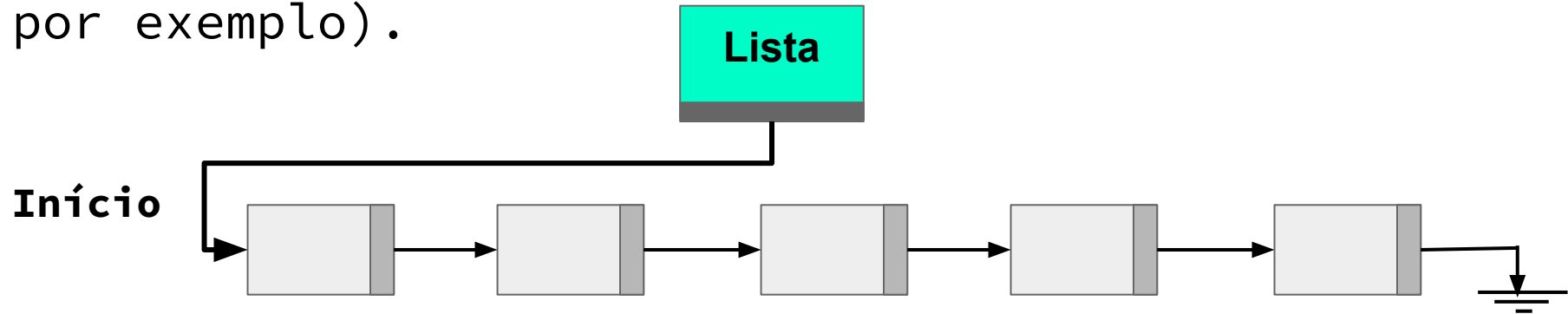
Em uma lista, por outro lado, é possível acessar elementos intermediários sem que haja essa quebra de estrutura. Esse acesso é, inclusive, função básica da lista.

VISÃO GERAL DE IMPLEMENTAÇÃO DE LISTAS



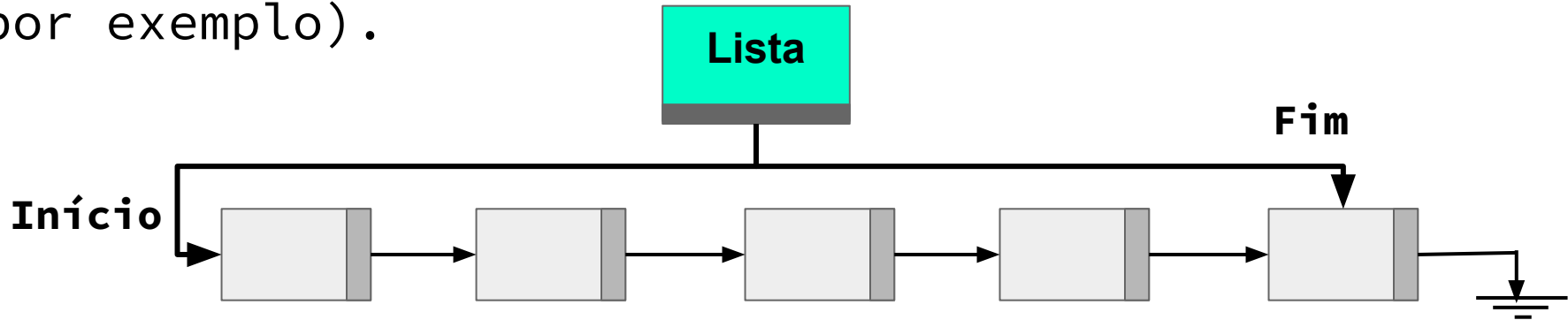
IMPLEMENTAÇÃO DE LISTAS: VISÃO GERAL - I

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).



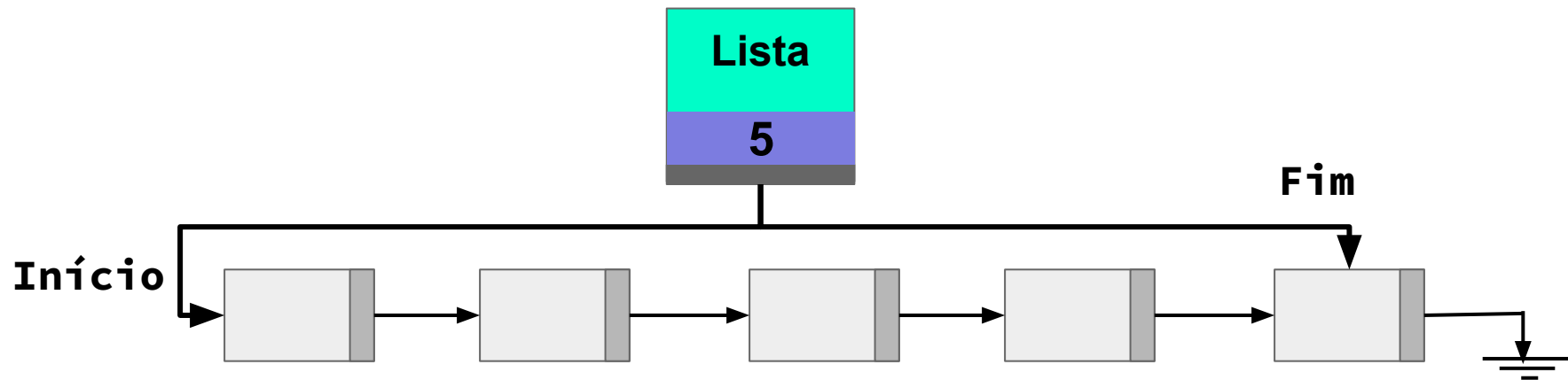
IMPLEMENTAÇÃO DE LISTAS: VISÃO GERAL - I

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).

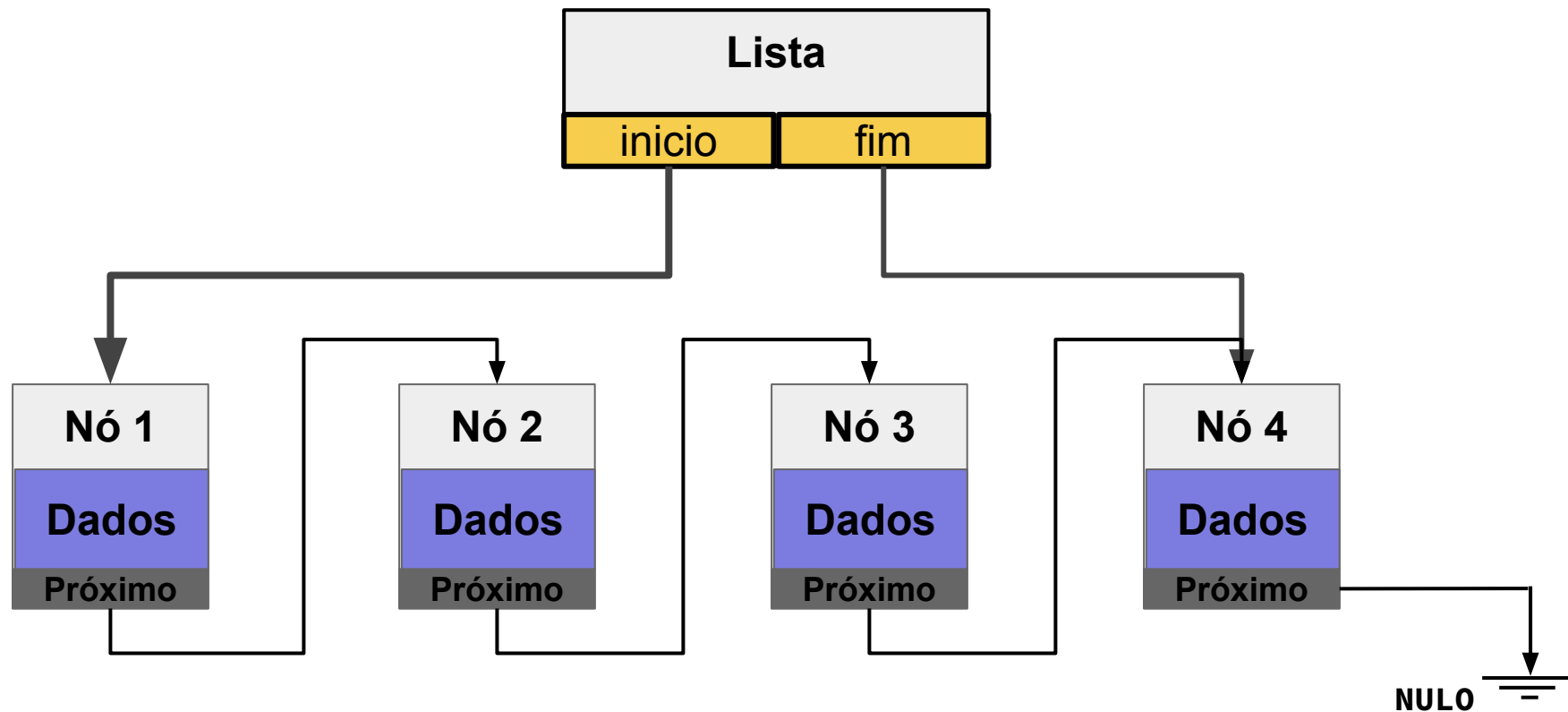


IMPLEMENTAÇÃO DE LISTAS: VISÃO GERAL - I

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.

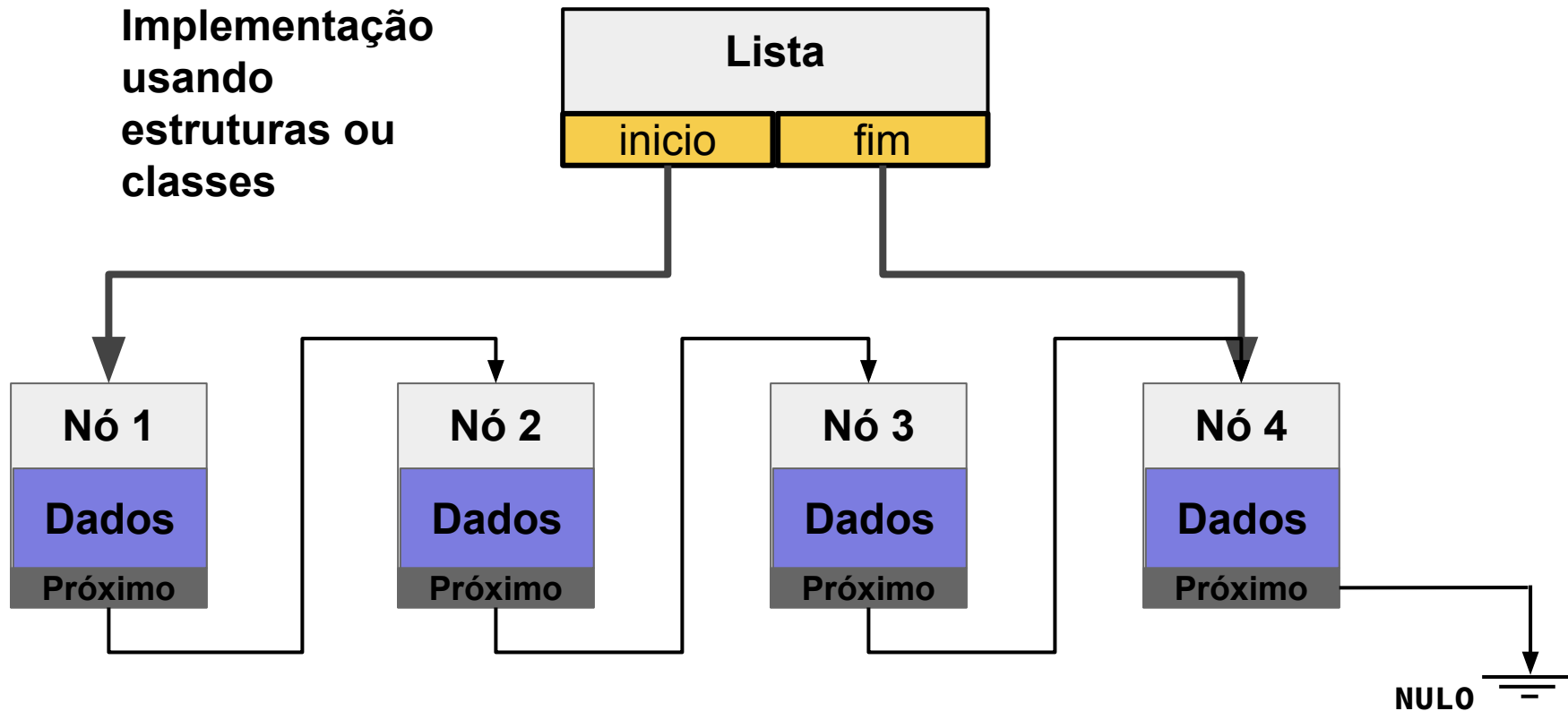


IMPLEMENTAÇÃO TRADICIONAL DE LISTAS



IMPLEMENTAÇÃO TRADICIONAL DE LISTAS

Implementação
usando
estruturas ou
classes

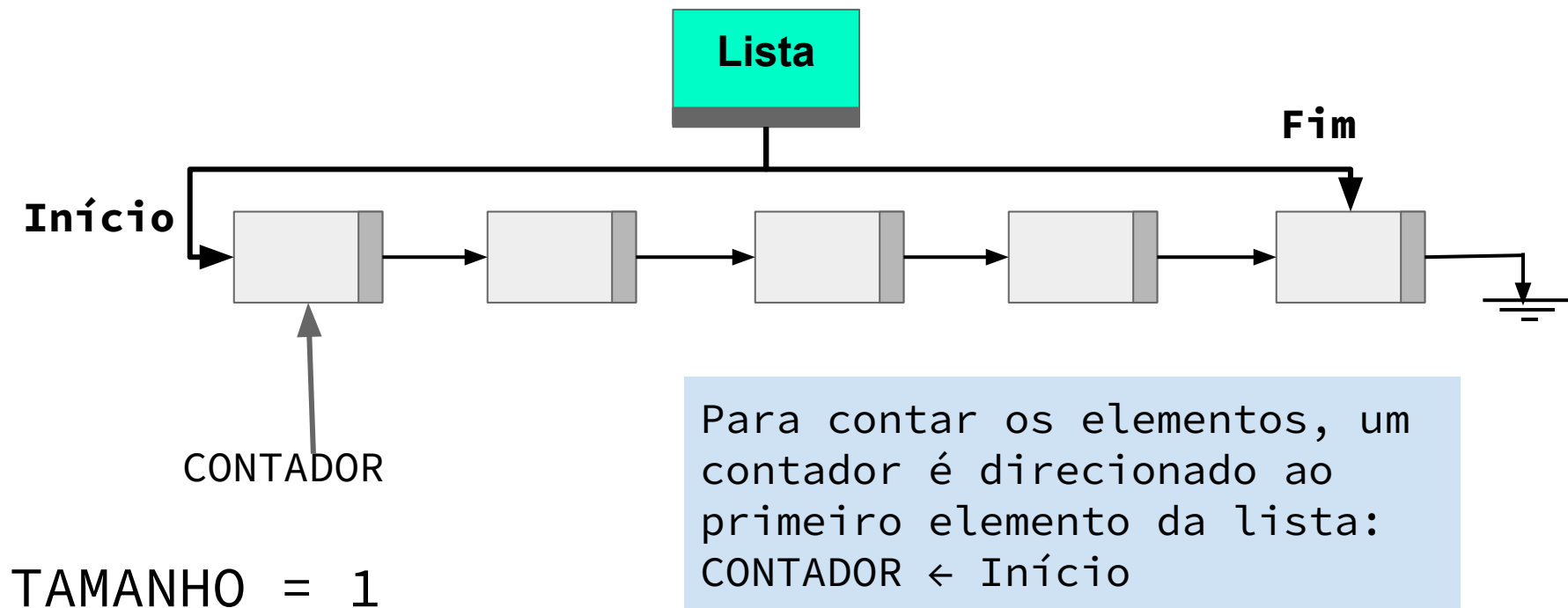


IMPLEMENTAÇÃO DE LISTAS: VISÃO GERAL - II

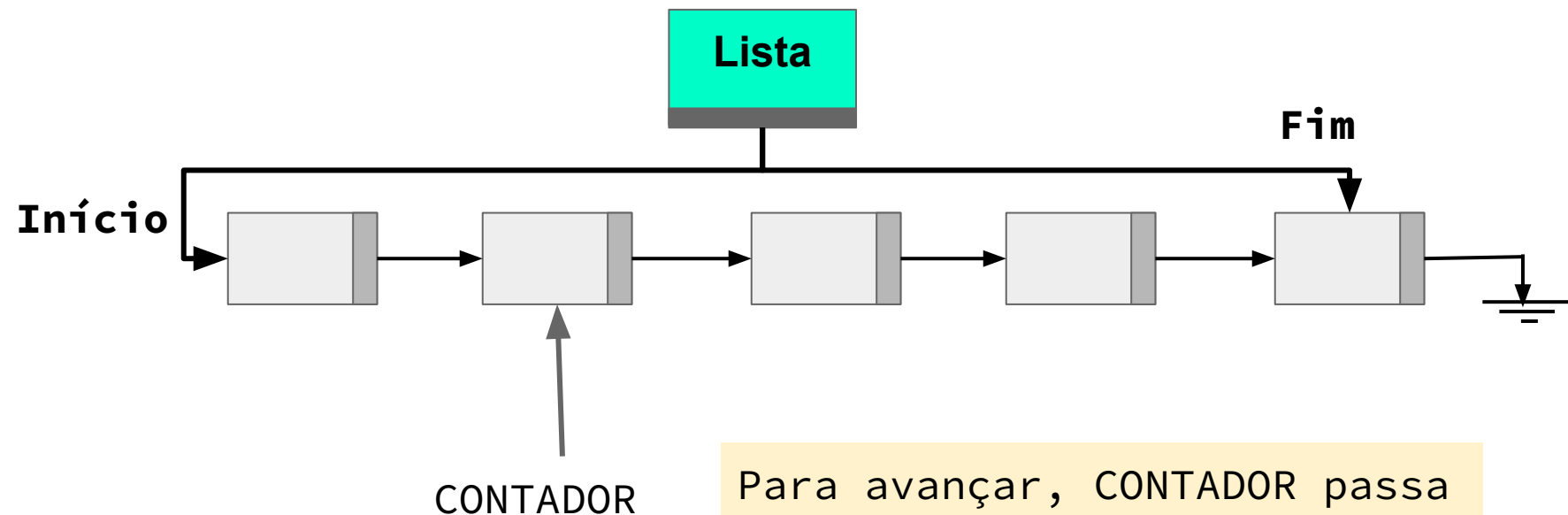
Sem um ponteiro para o último elemento da lista, inserções no final da lista precisam percorrê-la por inteiro antes de efetuar essa tarefa. Caso essa não seja uma operação comum, pode-se evitar esse atributo e o custo de sua manutenção.

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

IMPLEMENTAÇÃO DE LISTAS: CONTANDO ELEMENTOS



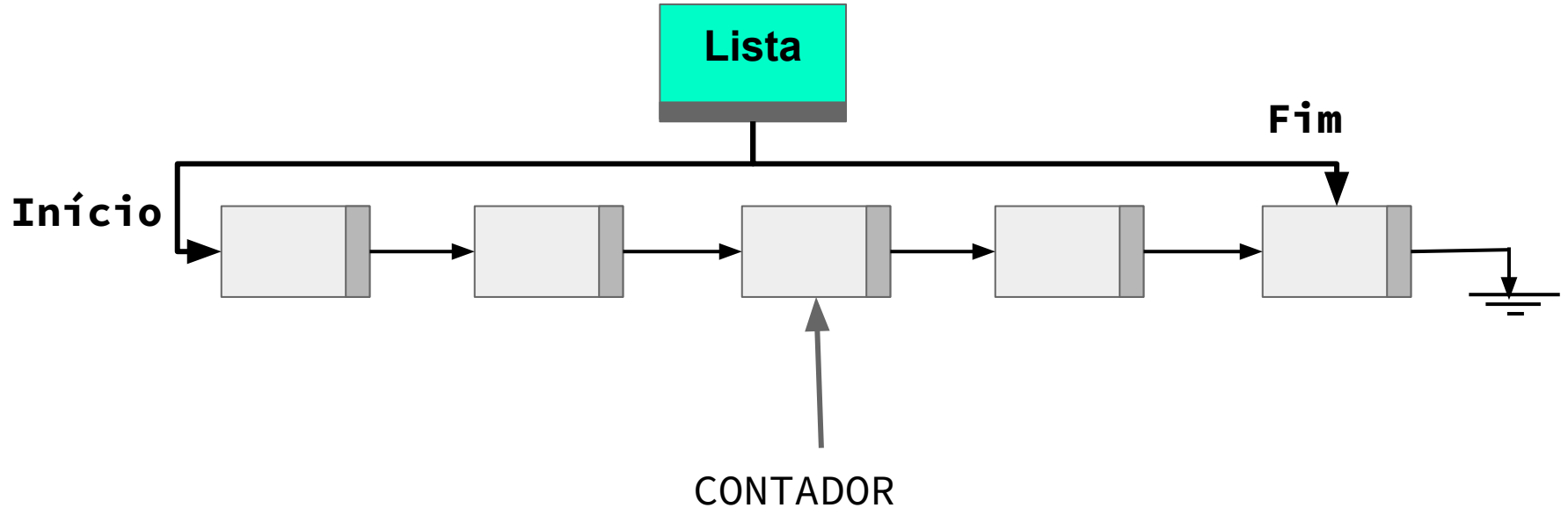
IMPLEMENTAÇÃO DE LISTAS: CONTANDO ELEMENTOS



TAMANHO = 2

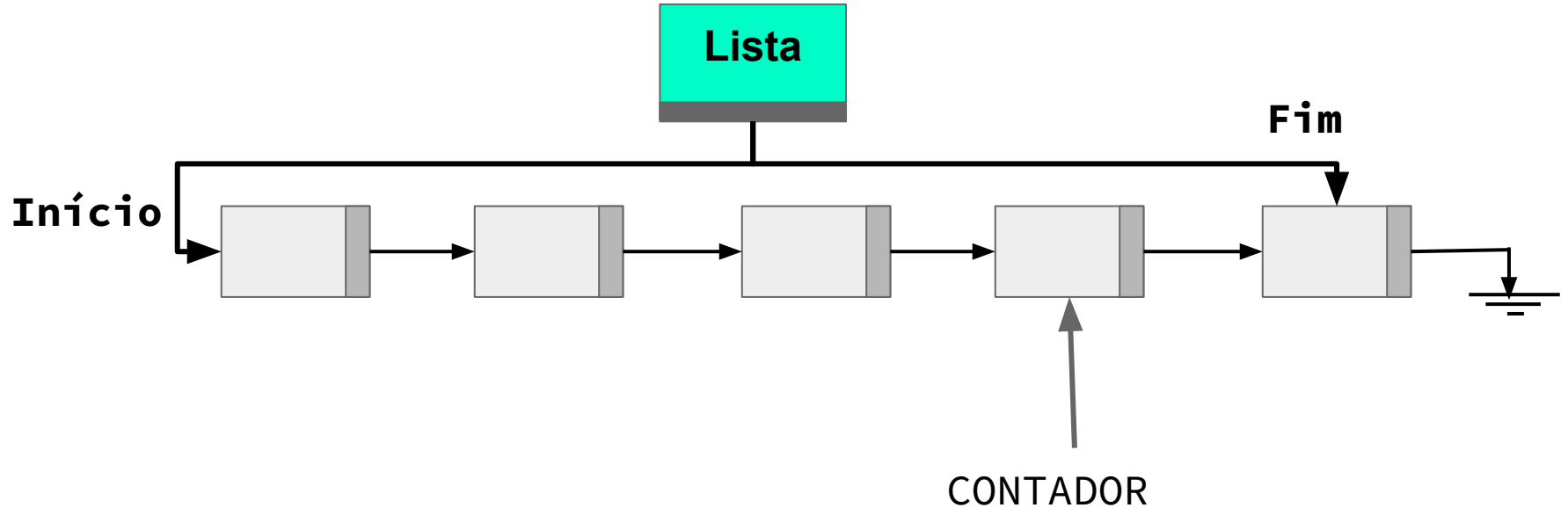
Para avançar, CONTADOR passa a apontar para seu próximo:
 $\text{CONTADOR} \leftarrow \text{CONTADOR.próximo}$

IMPLEMENTAÇÃO DE LISTAS: CONTANDO ELEMENTOS



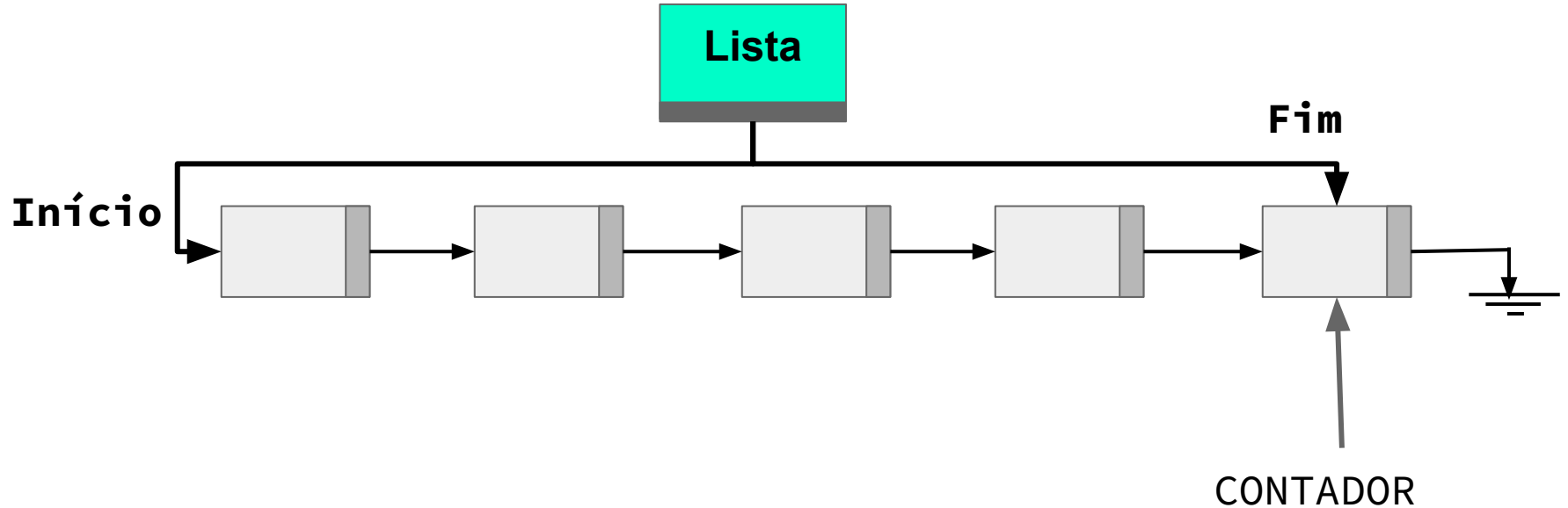
TAMANHO = 3

IMPLEMENTAÇÃO DE LISTAS: CONTANDO ELEMENTOS



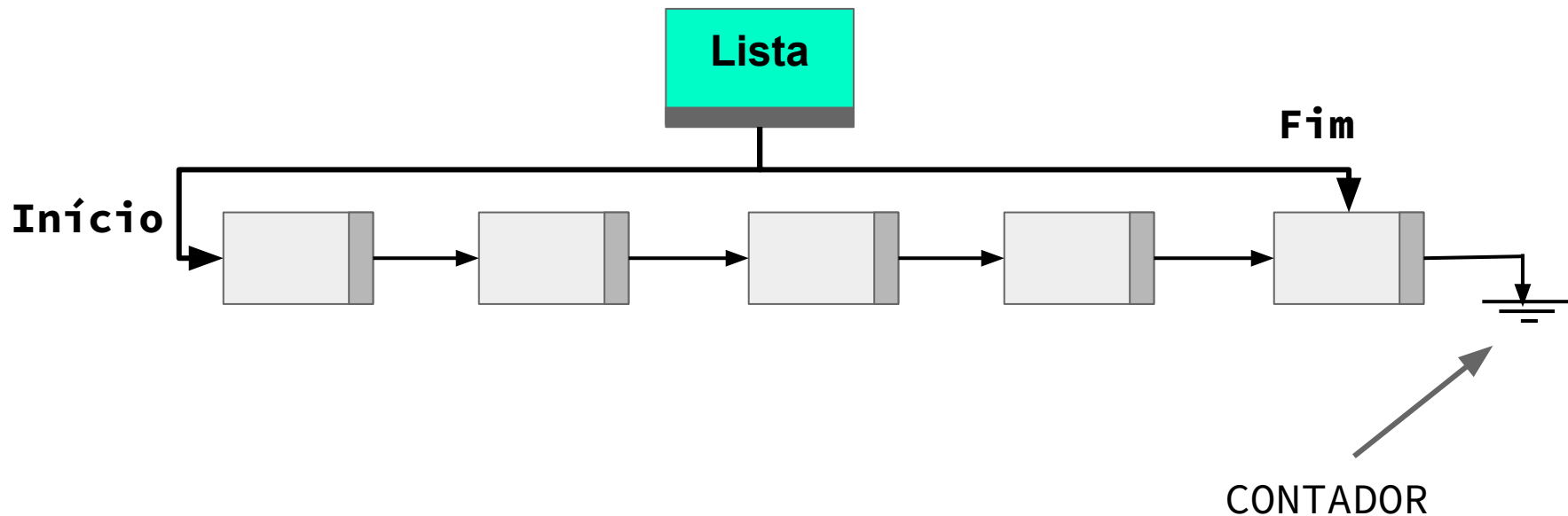
TAMANHO = 4

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.

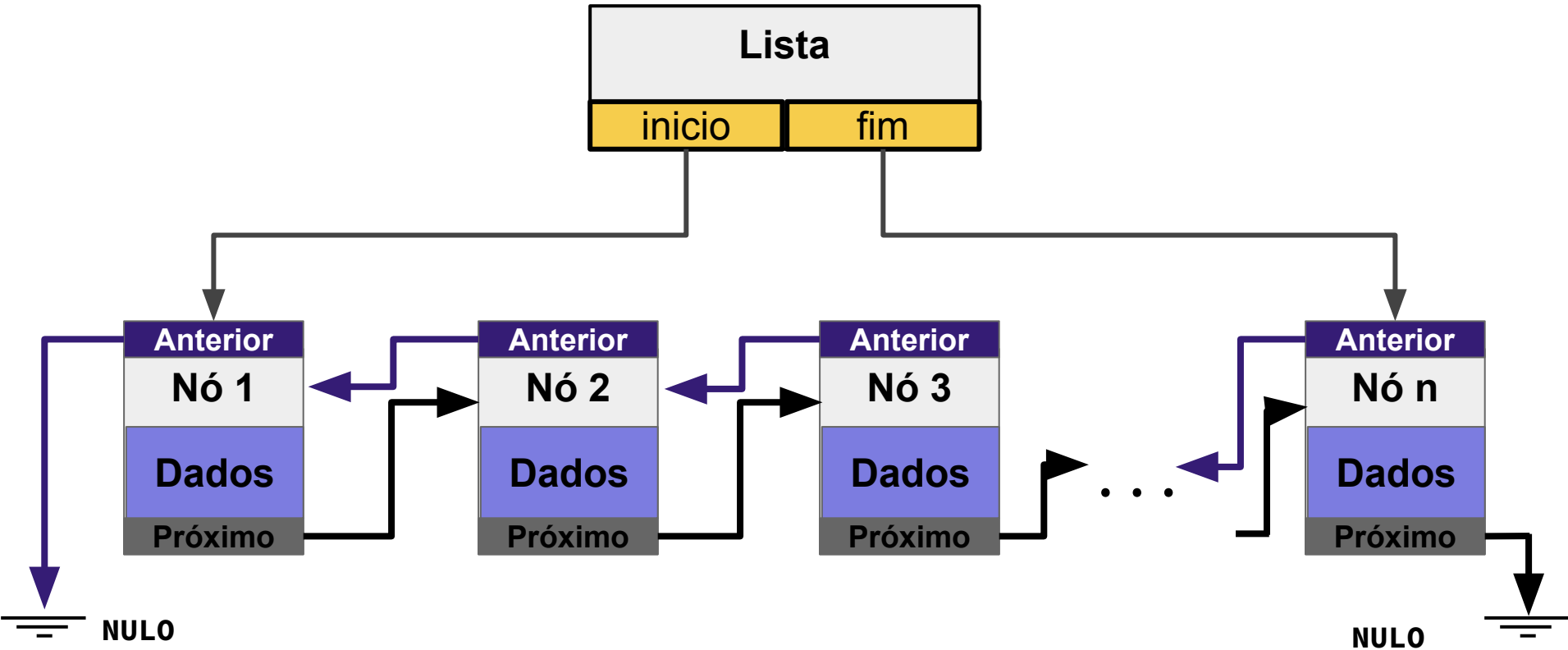
LISTAS DUPLAMENTE ENCADEADAS - I

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.

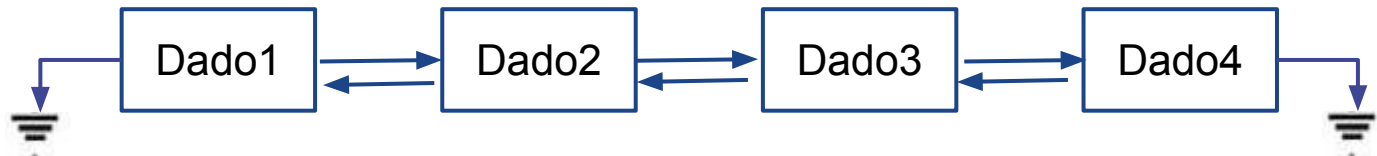
Nesse caso, utiliza-se um vetor de nós, em que cada nó armazena, por exemplo, além de seus dados, a posição do nó anterior e do próximo.

LISTAS ESTATICAMENTE ENCADEADAS - II

Por exemplo, o arranjo a seguir:

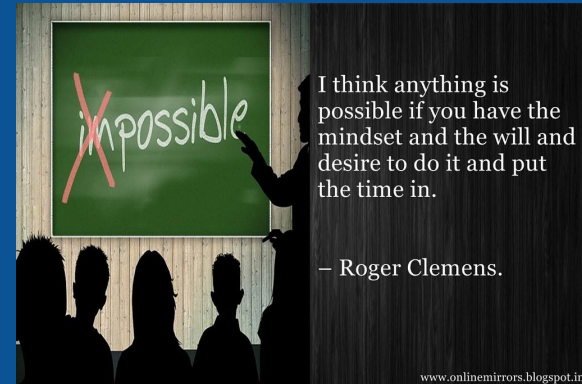
Célula 0		Célula 1		Célula 2		Célula 3		Célula 4	
Dado1	3	NULO	-1	Dado3	4	Dado2	2	Dado4	-1
	-1		-1		3		0		2

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:

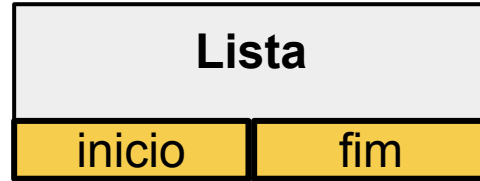
- inserção no início da lista
- inserção no final da lista
- inserção em uma posição determinada da lista
(precisa percorrer a lista até chegar à posição desejada)

LISTA: MÉTODOS USUAIS - IV

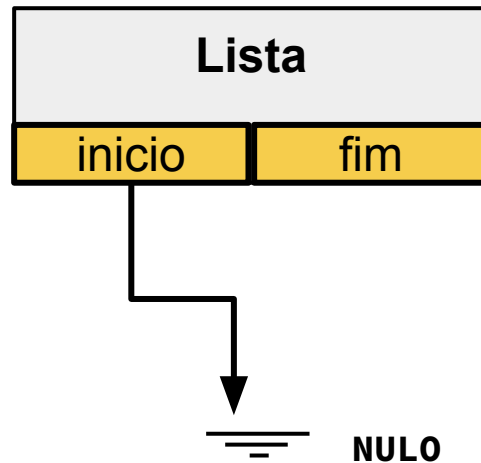
Remoção: possibilita a remoção de um dado nó na lista. Em geral, existem três opções de remoção:

- 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)

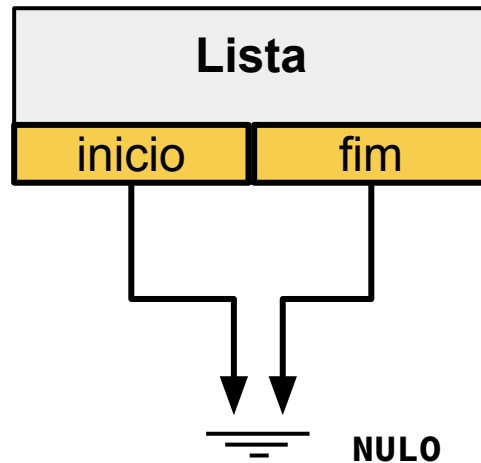
CRIAÇÃO DA LISTA - I



CRIAÇÃO DA LISTA - I



CRIAÇÃO DA LISTA - I



CRIAÇÃO DA LISTA - PSEUDOCÓDIGO

criarLista():

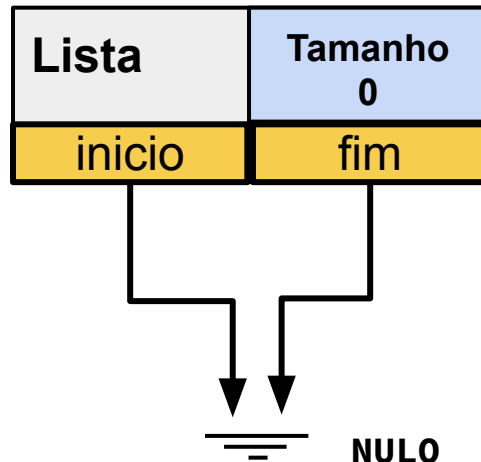
início ← NULO;

fim ← NULO;

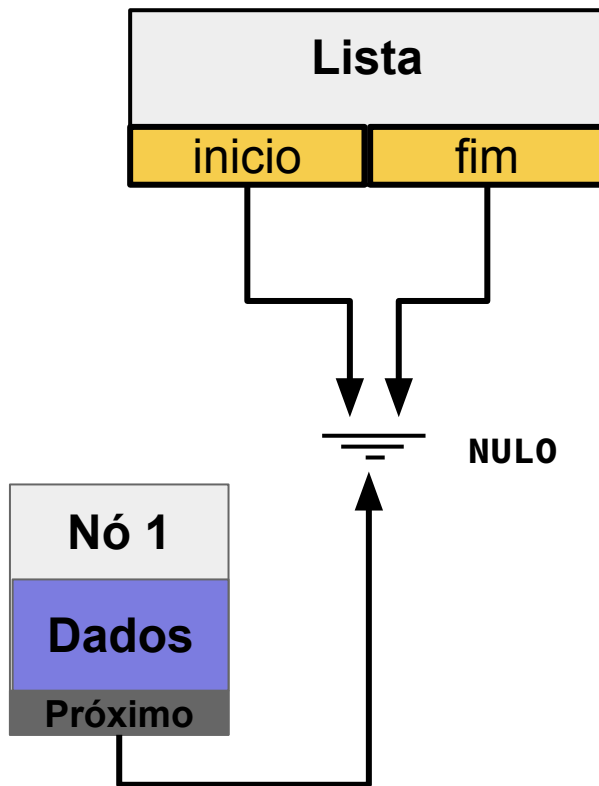
// caso seja interessante armazenar

// o tamanho da lista

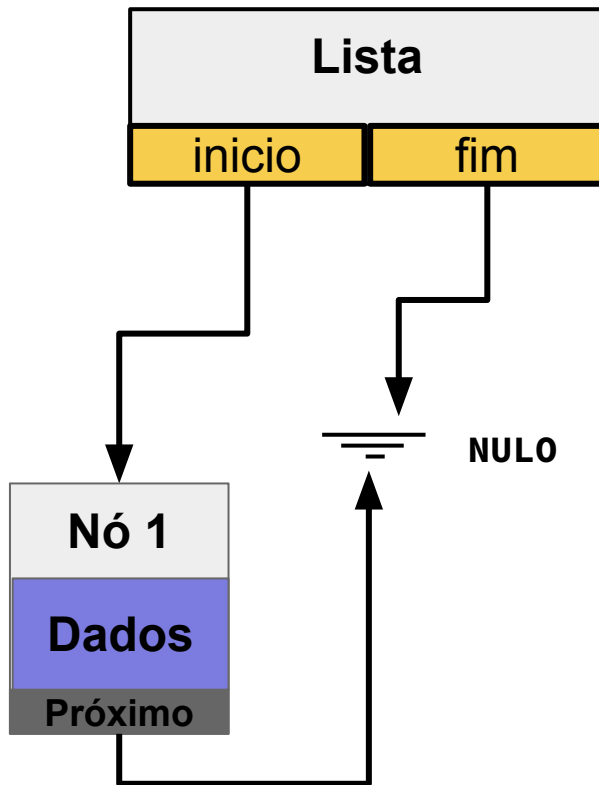
tamanho ← 0;



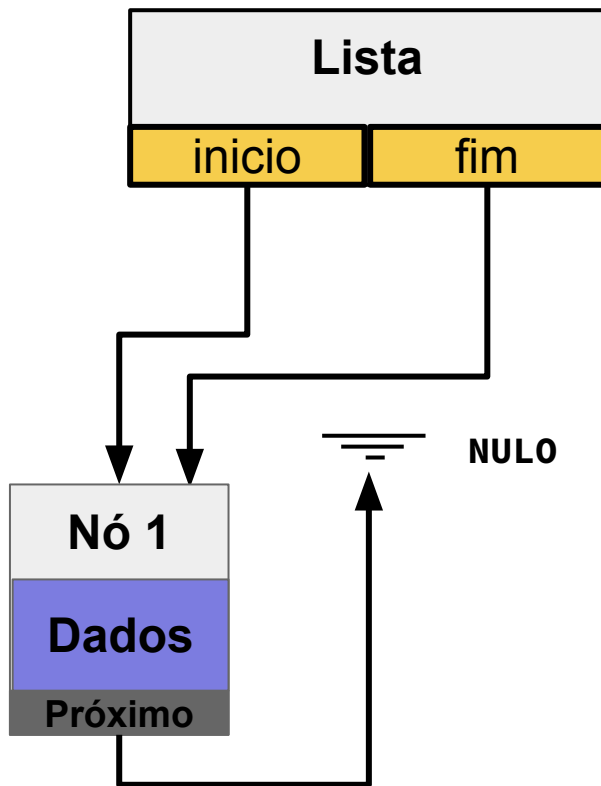
INSERÇÃO EM LISTA VAZIA - I



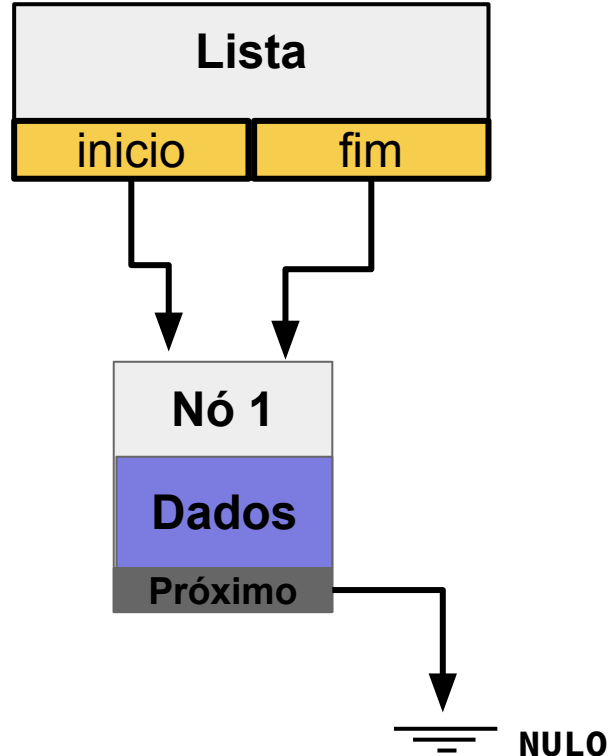
INSERÇÃO EM LISTA VAZIA - I



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

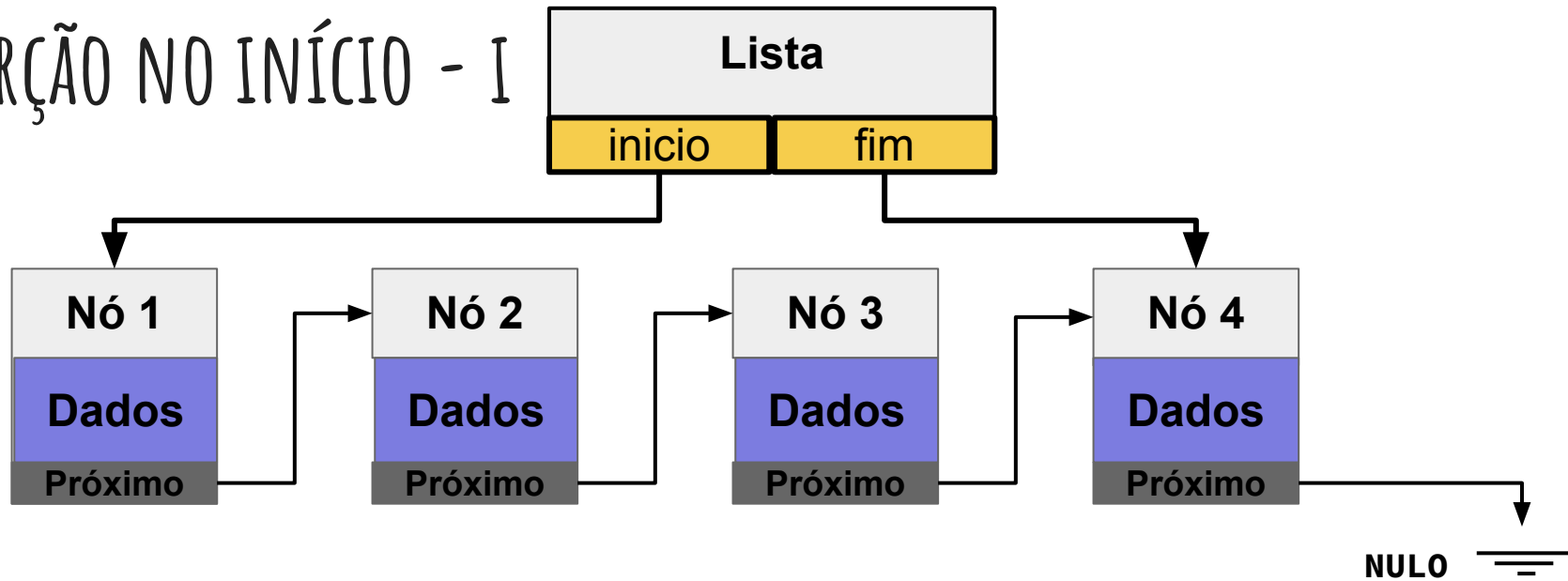
tamanho ← 1;

INSERÇÃO EM LISTA VAZIA - OBSERVAÇÕES

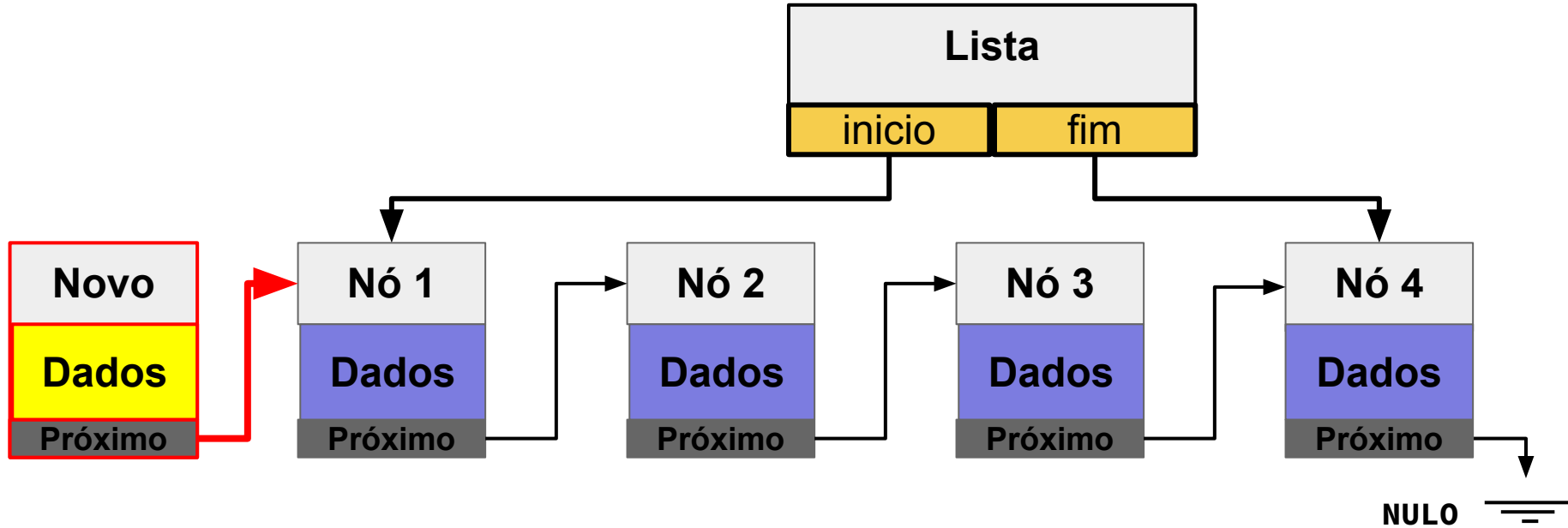
Um método para inserção em lista vazia ou não é implementada ou é inserida como método auxiliar, não invocado pelo usuário.

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

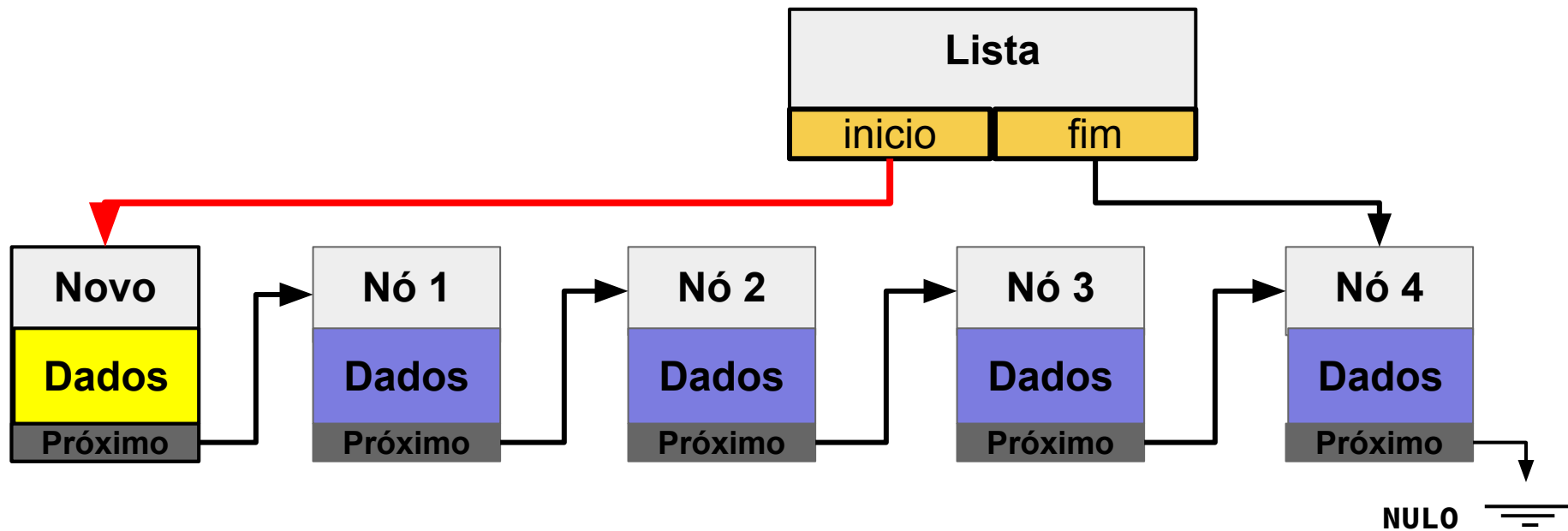
INserção no início - I



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.

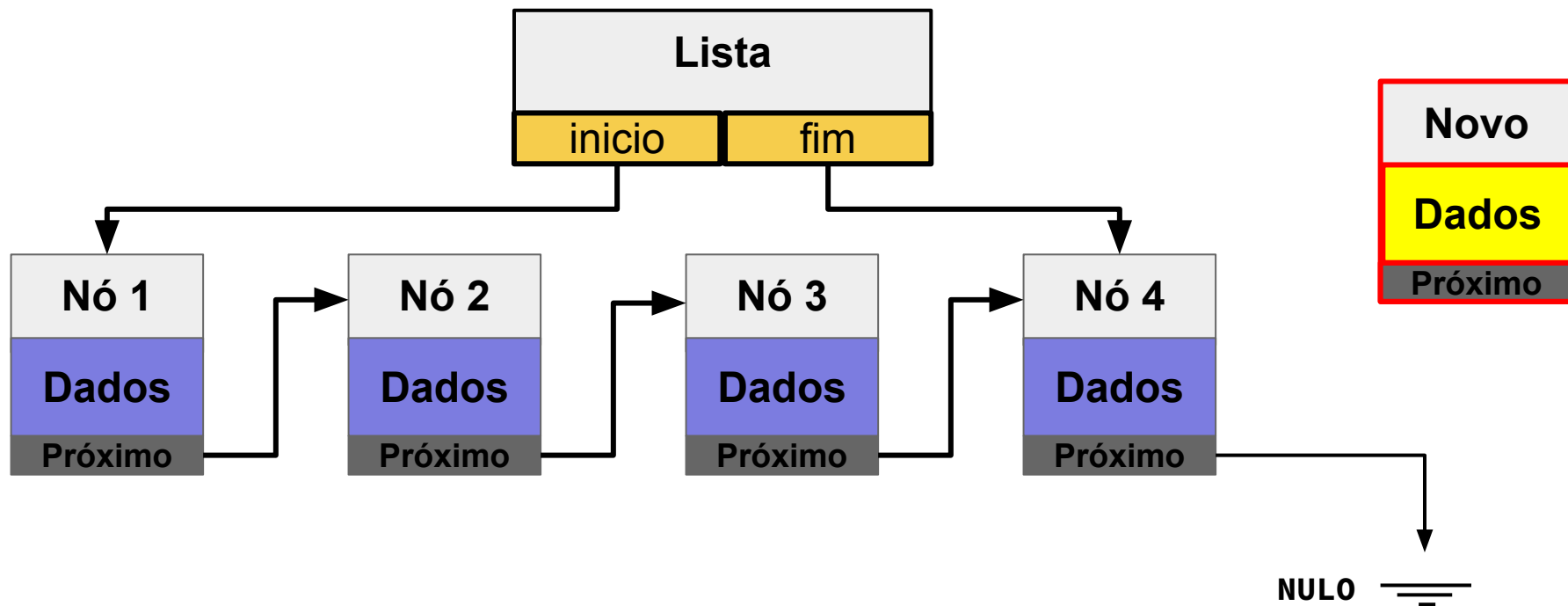
Assim, é necessário verificar e tratar essa situação.

INSERÇÃO NO INÍCIO - PSEUDOCÓDIGO

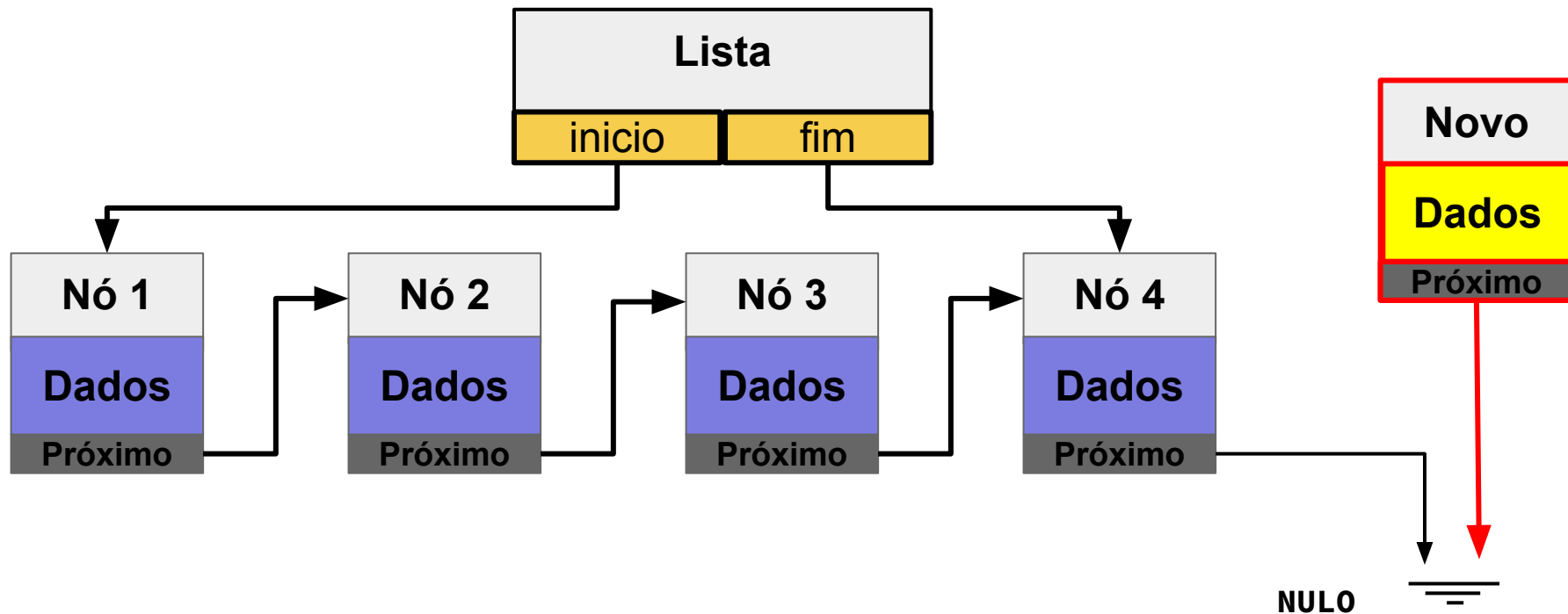
inserirNoInicioDaLista(valor):

```
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  
}
```

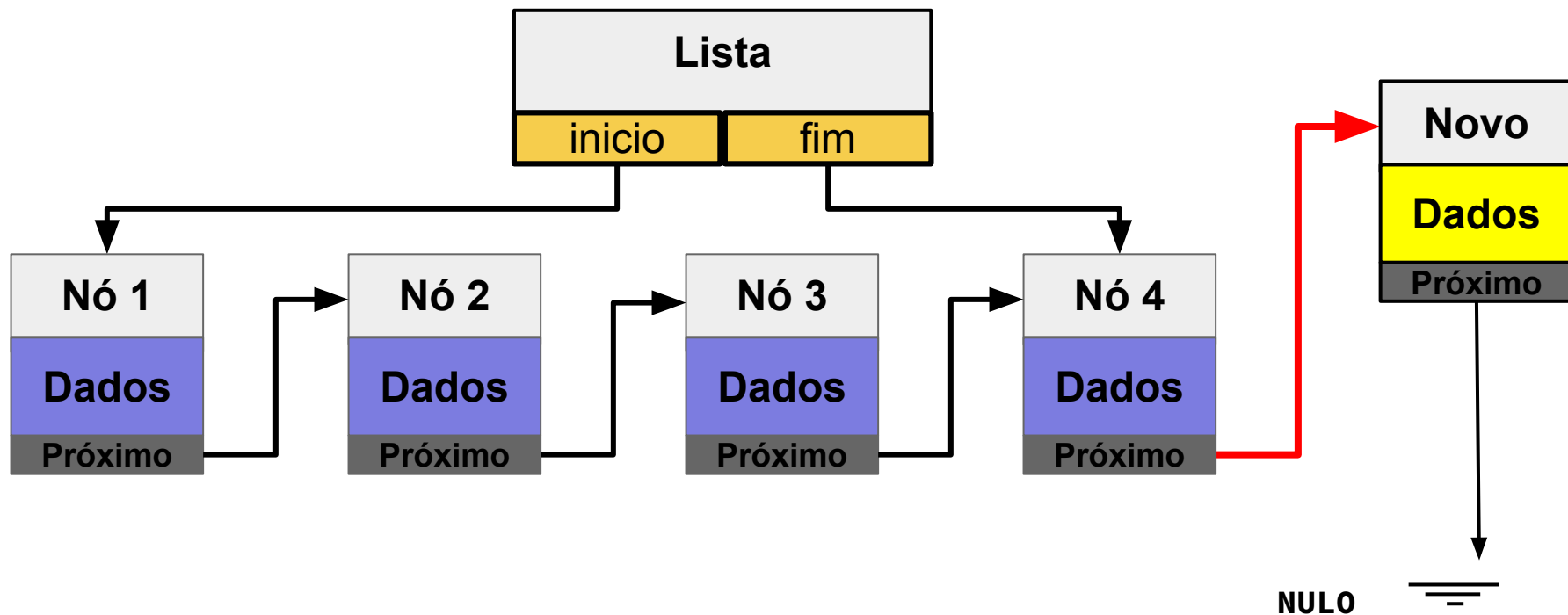
INSERÇÃO NO FIM - I



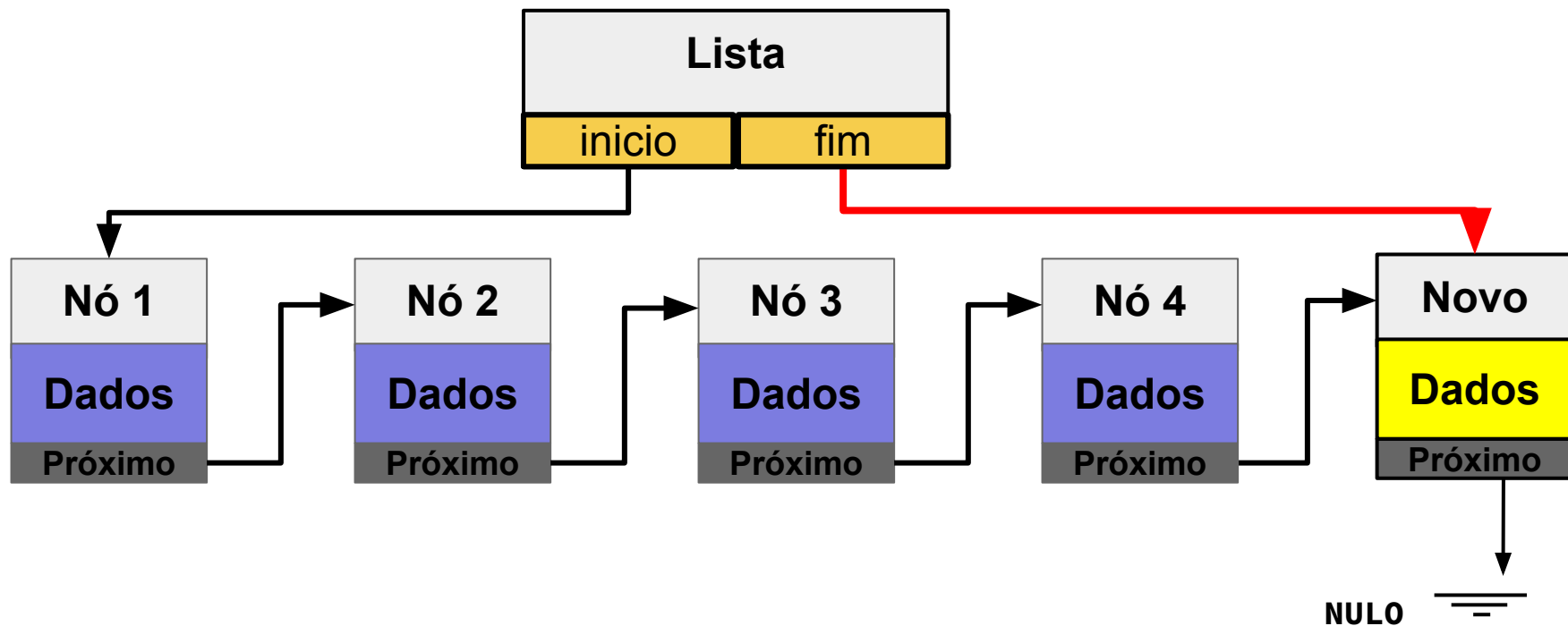
INSERÇÃO NO FIM - II



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?

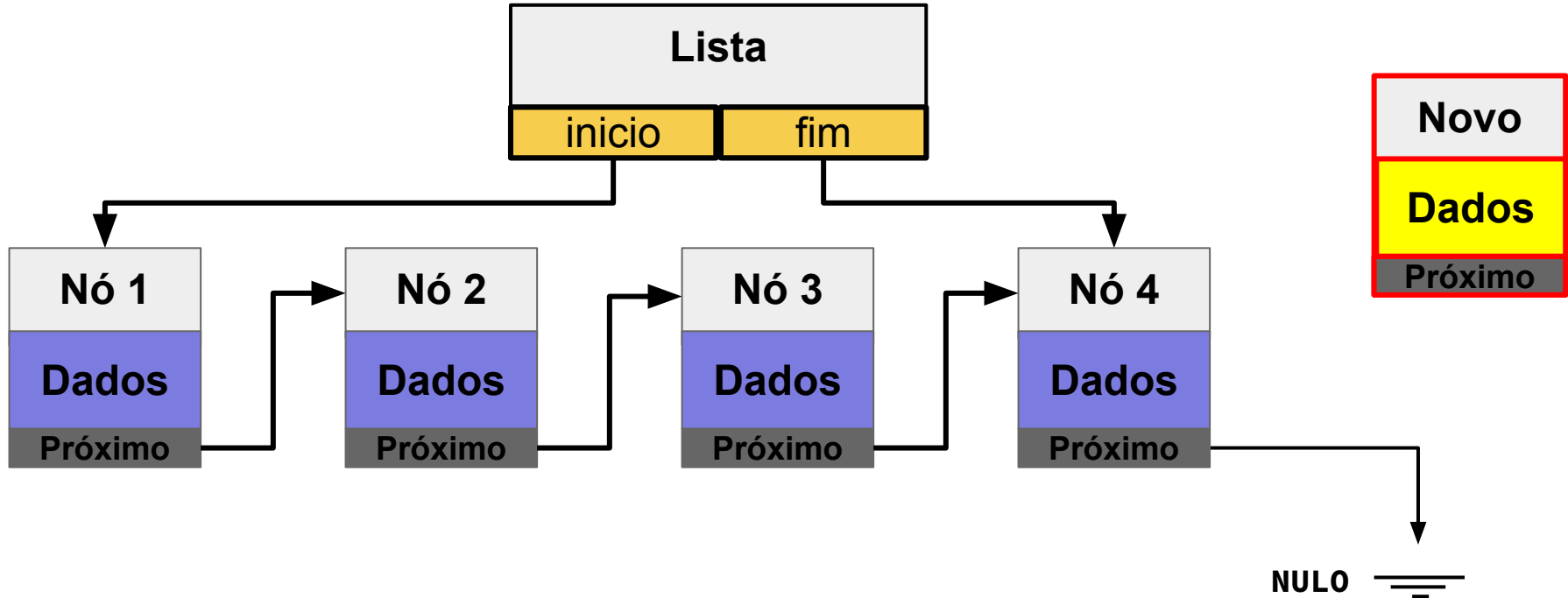
INSERÇÃO NO FIM - PSEUDOCÓDIGO

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  
}
```

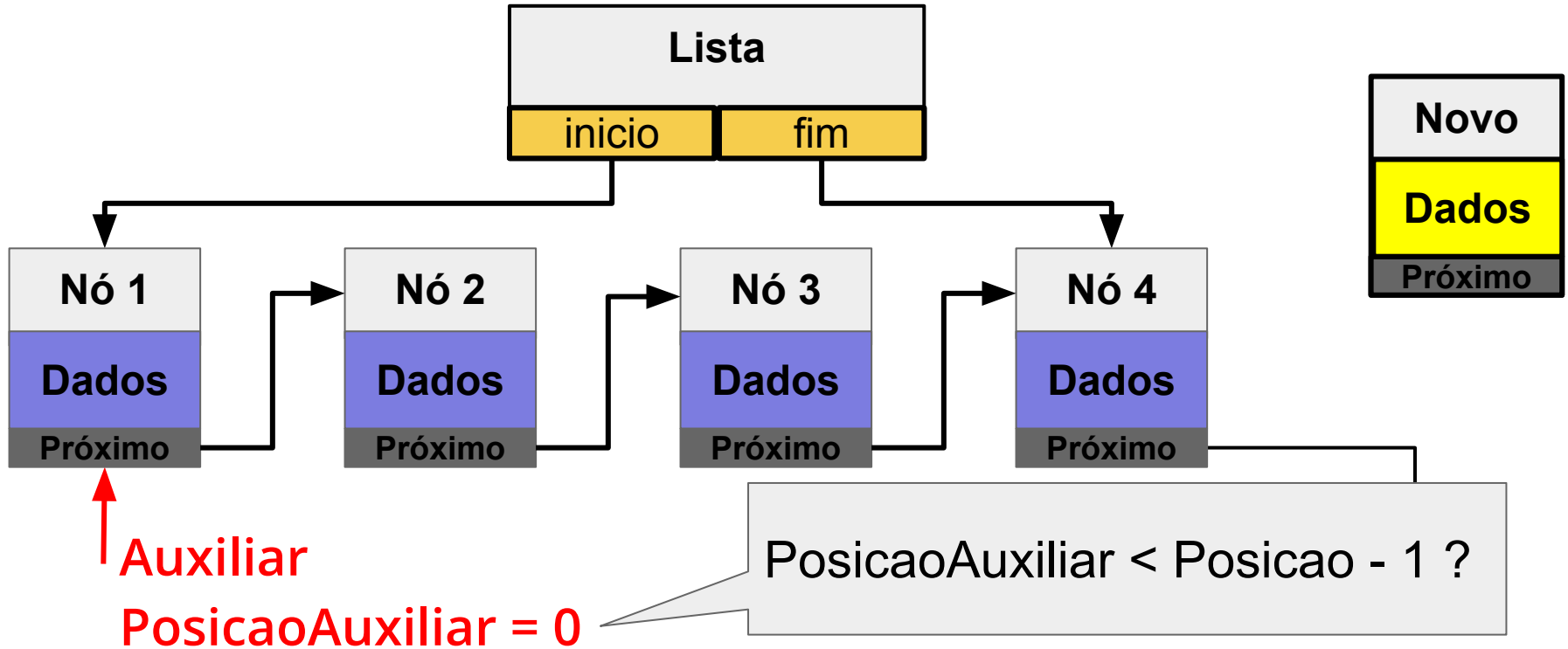
INSERÇÃO EM POSIÇÃO ESPECÍFICA - I

Inserir na posição 2
Posicao = 2



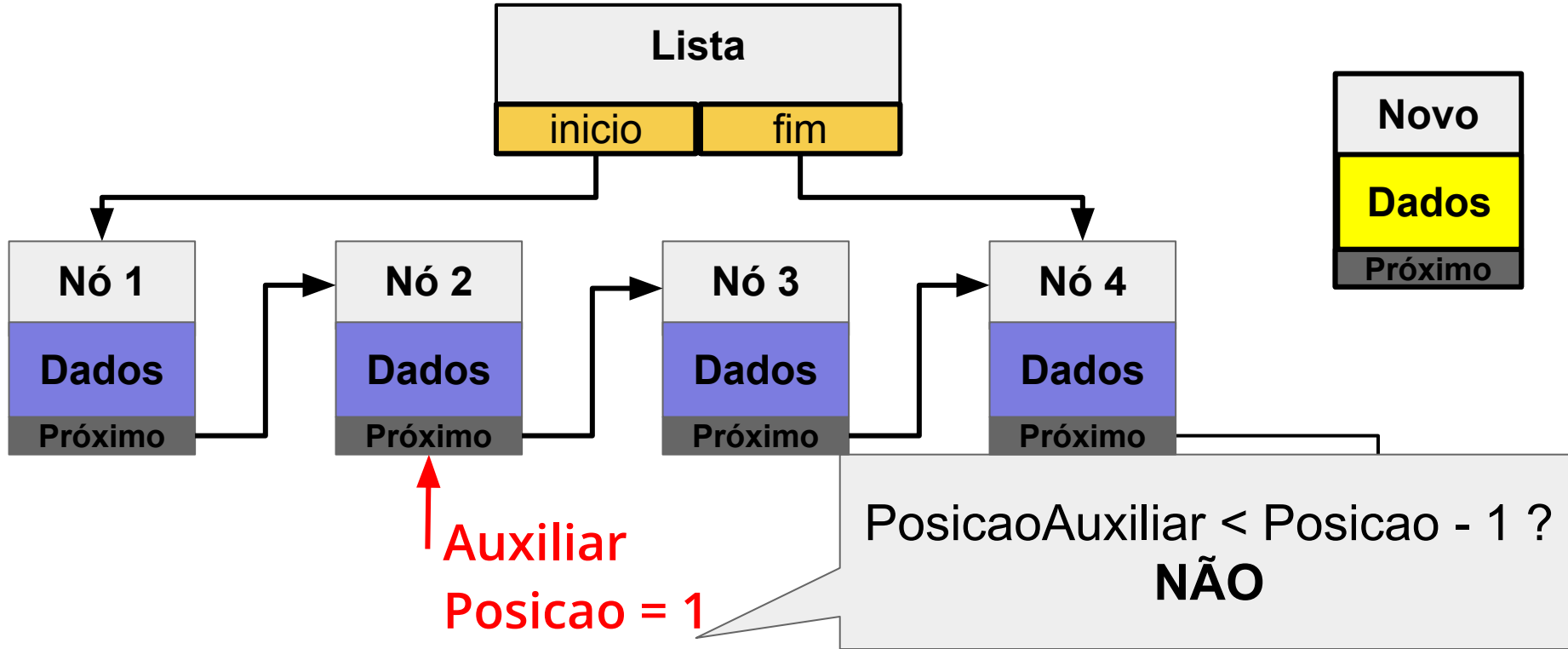
INSERÇÃO EM POSIÇÃO ESPECÍFICA - II

Inserir na posição 2
Posicao == 2



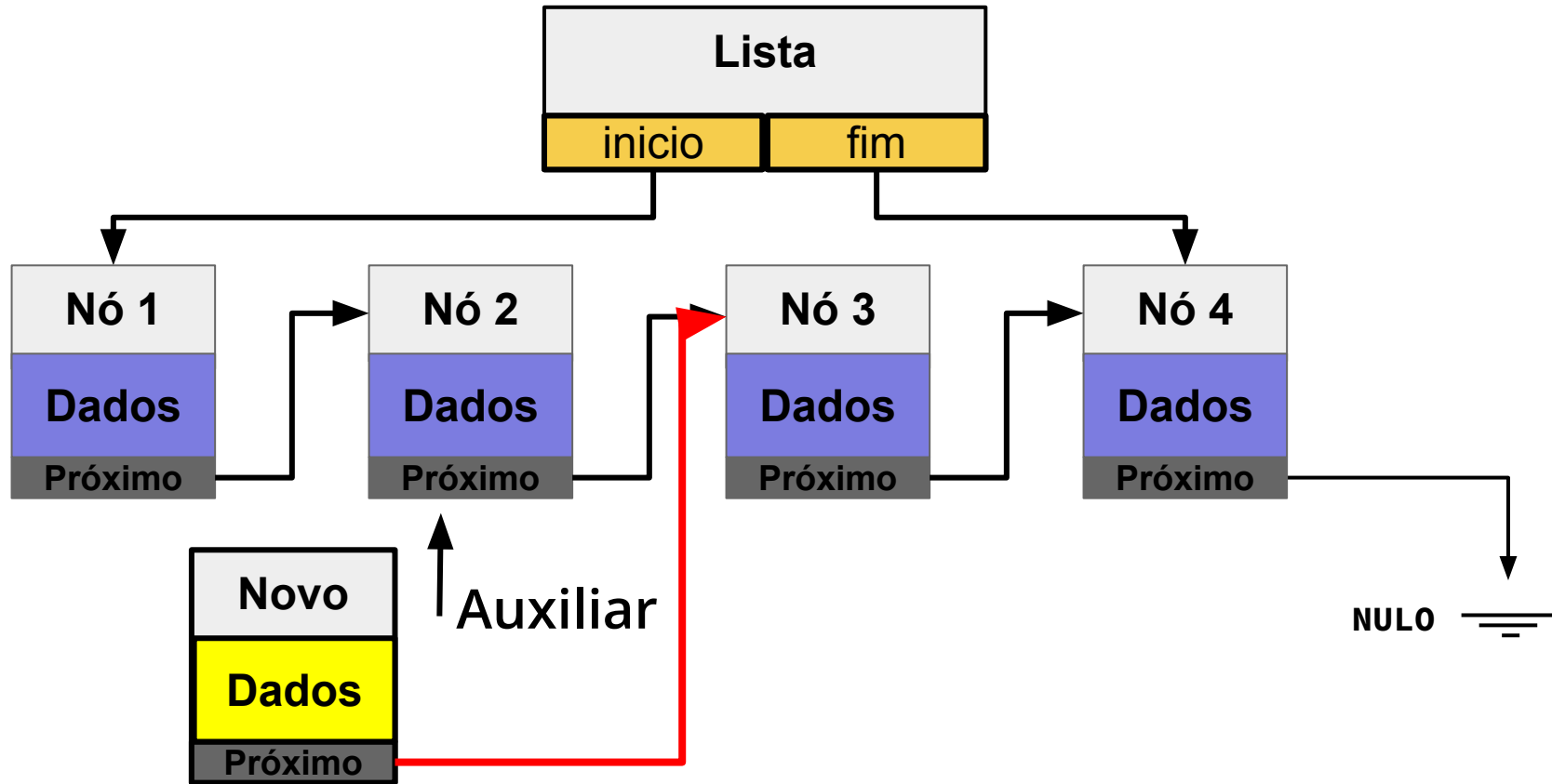
INSERÇÃO EM POSIÇÃO ESPECÍFICA - III

Inserir na posição 2



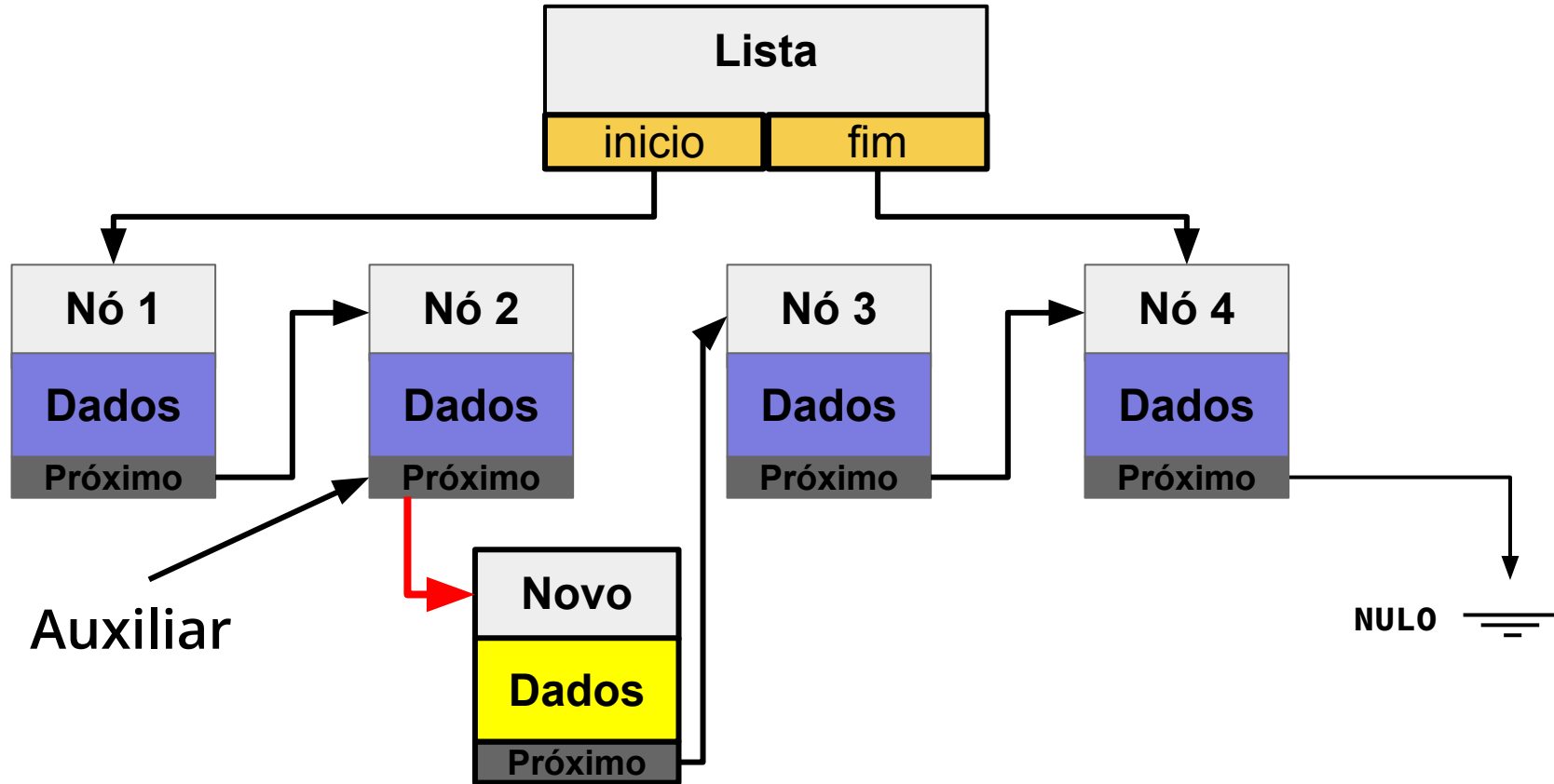
INSERÇÃO EM POSIÇÃO ESPECÍFICA - IV

Inserir na posição 2



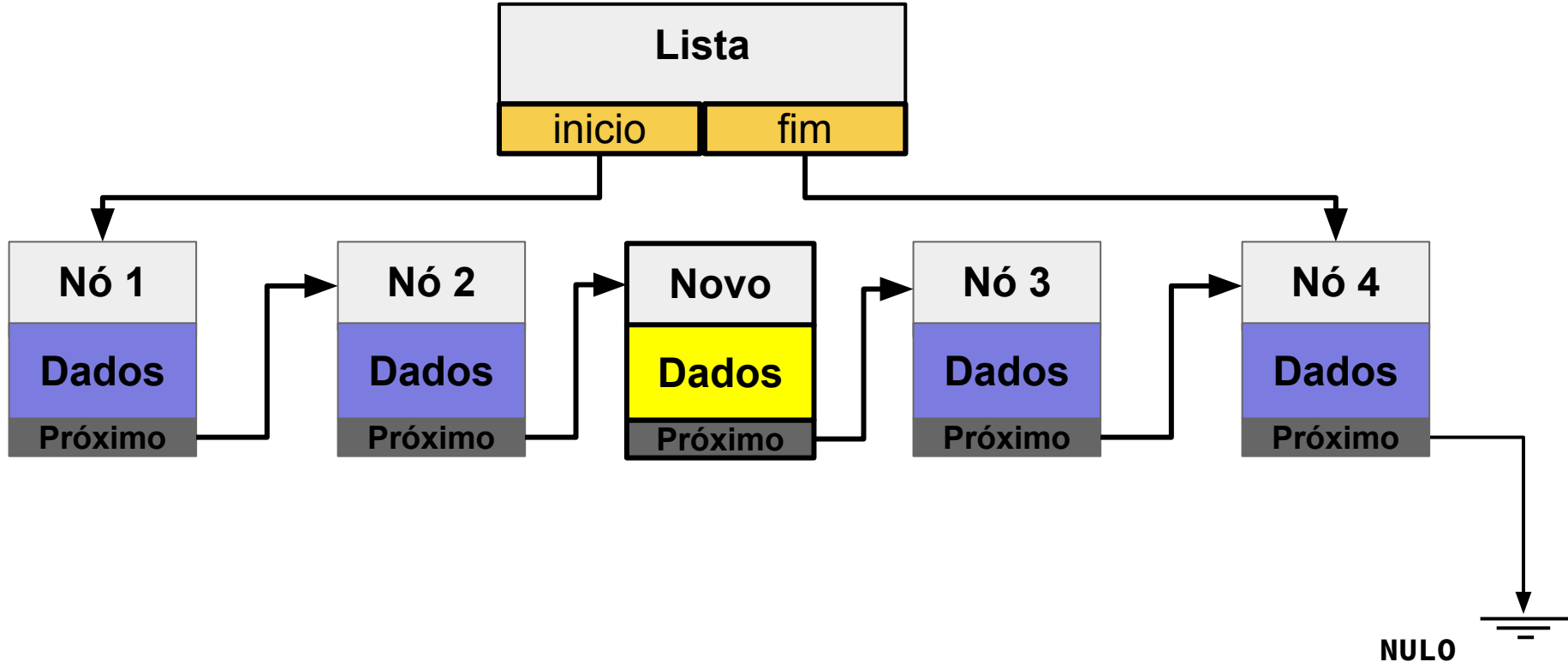
INSERÇÃO EM POSIÇÃO ESPECÍFICA - V

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

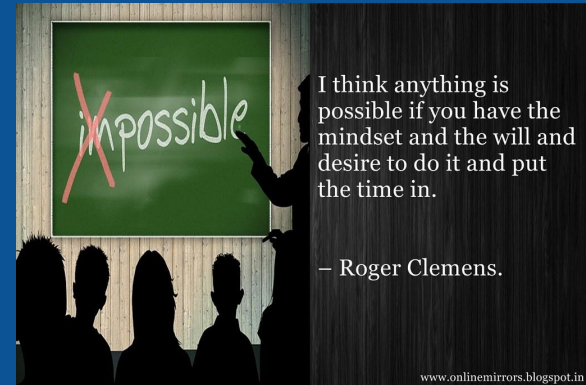
Em geral, implementações tradicionais de listas para uso geral costumam utilizar o duplo encadeamento e apresentar os três métodos de inserção aqui apresentados.

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

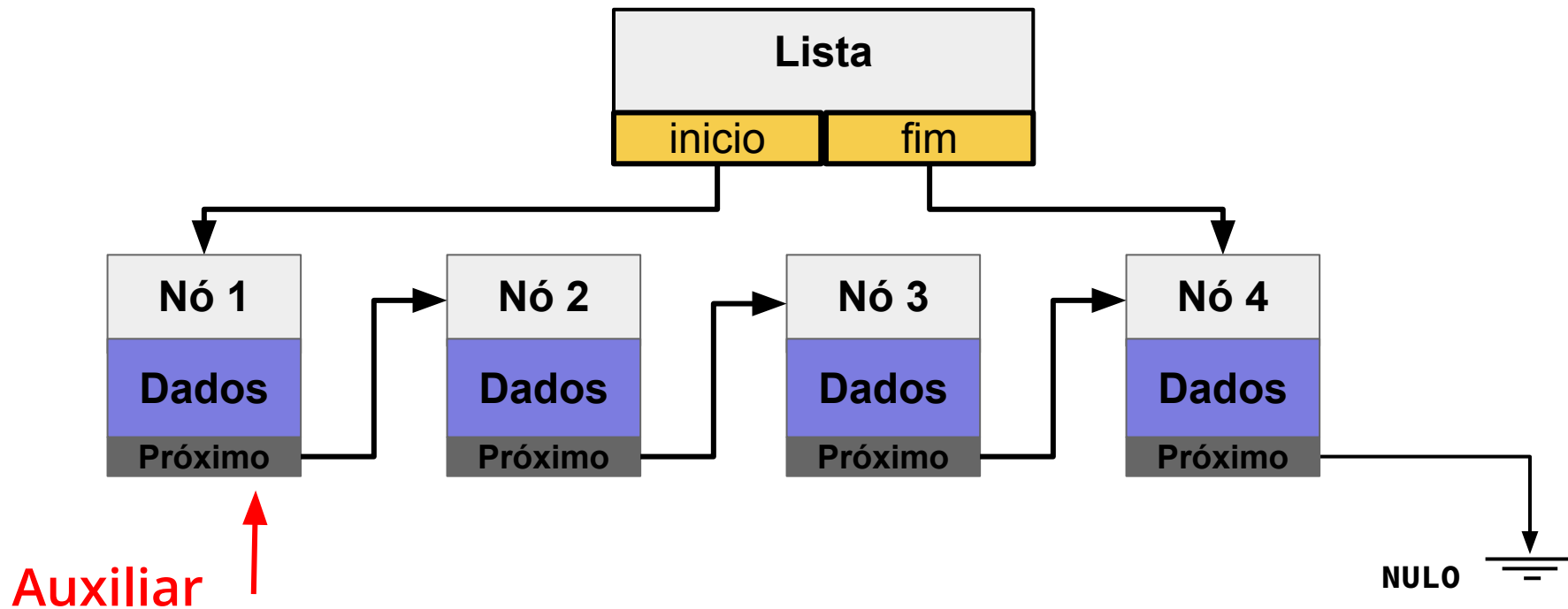
O duplo encadeamento também não é necessário em boa parte dos problemas.

MÉTODOS USUAIS EM LISTAS

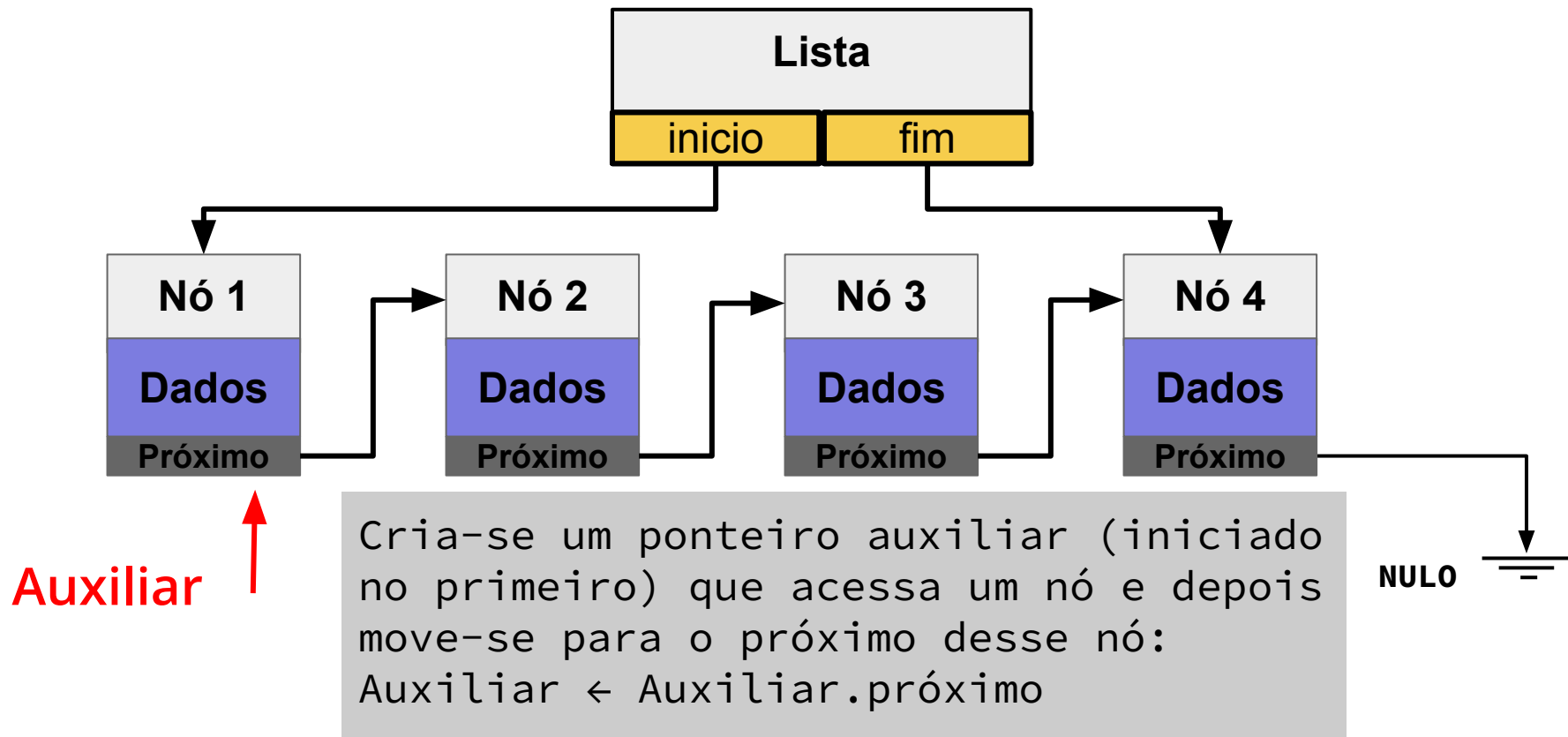
PARTE II



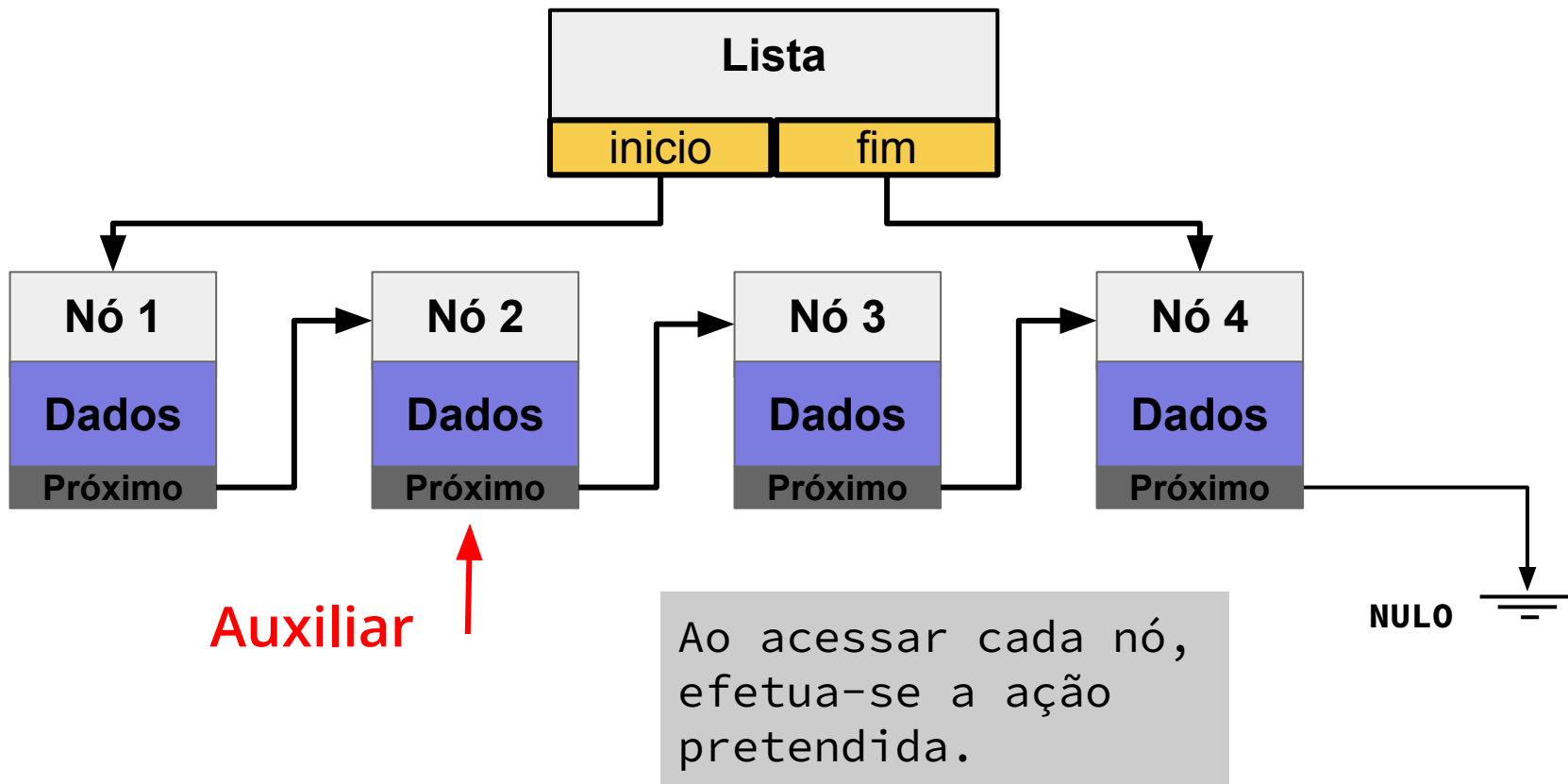
PERCORRIMENTO / CAMINHAMENTO - I



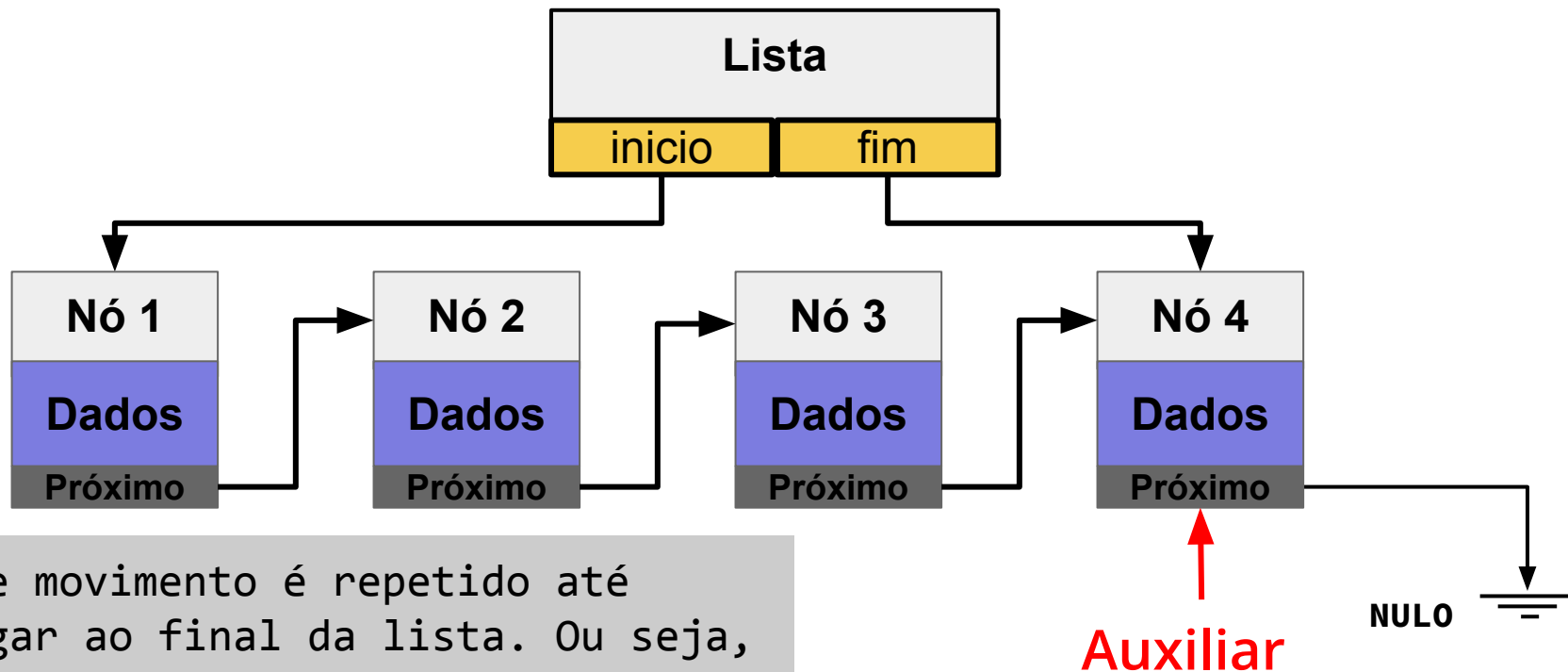
PERCORRIMENTO / CAMINHAMENTO - I



PERCORRIMENTO / CAMINHAMENTO - I



PERCORRIMENTO / CAMINHAMENTO - I



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():

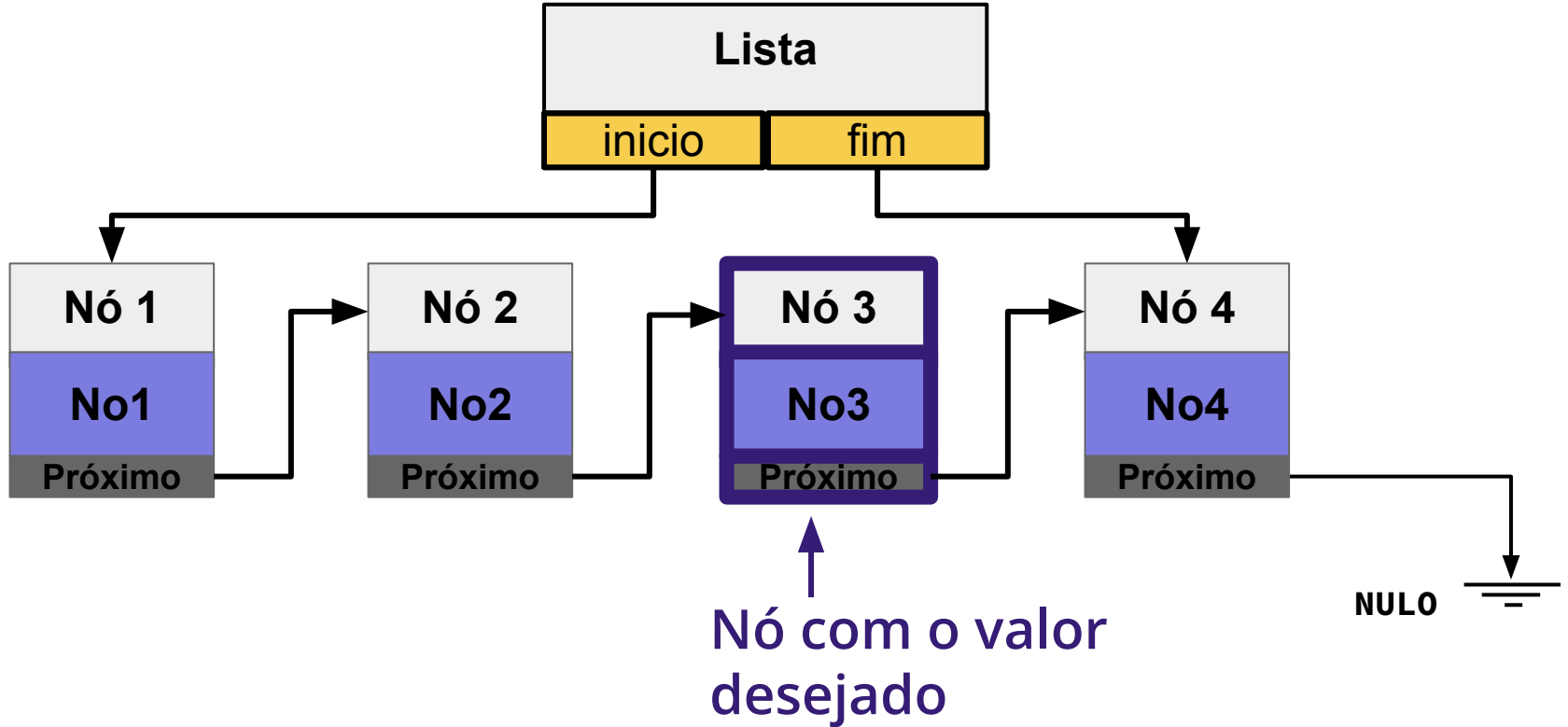
```
percorrerListaAux(inicio);
```

percorrerListaAux(umNoh):

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

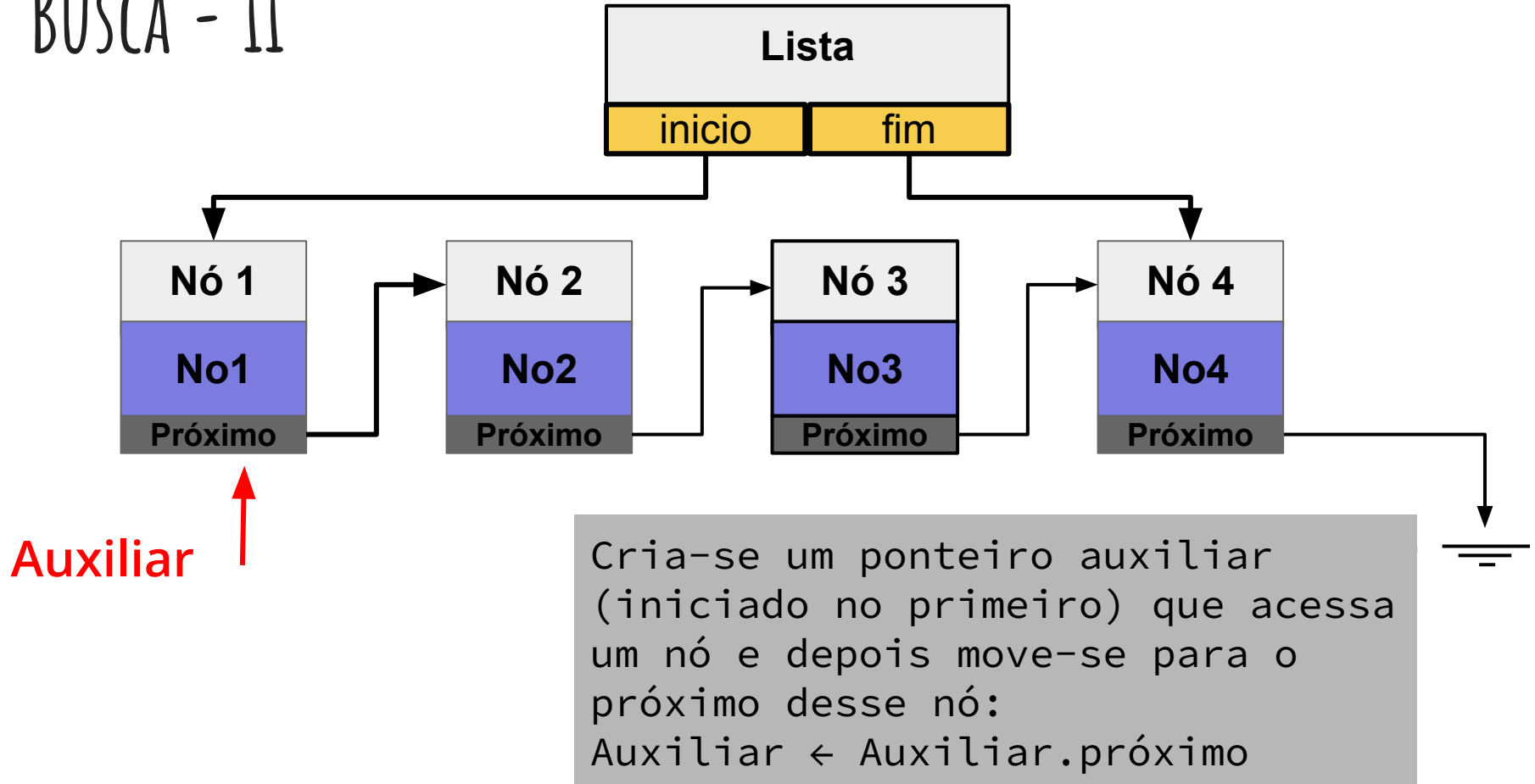
BUSCA - I

Valor desejado = No3



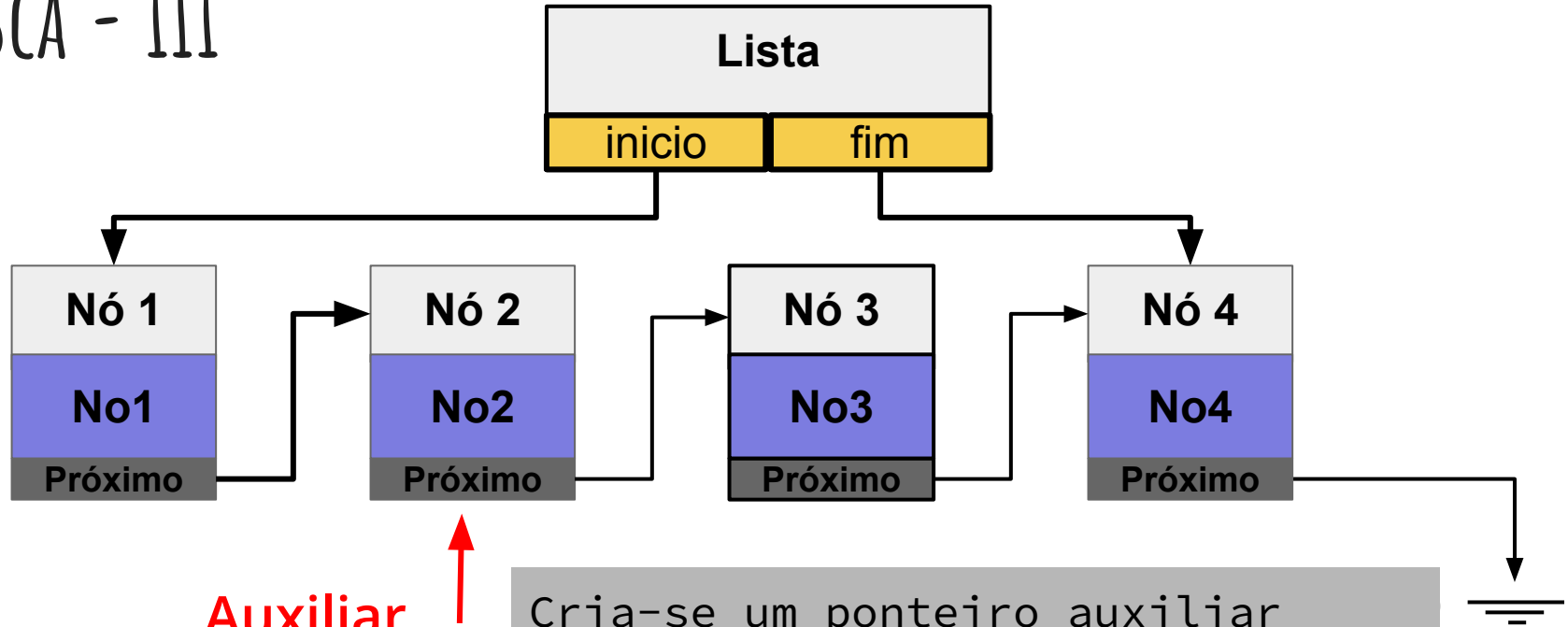
BUSCA - II

Valor desejado = No3



BUSCA - III

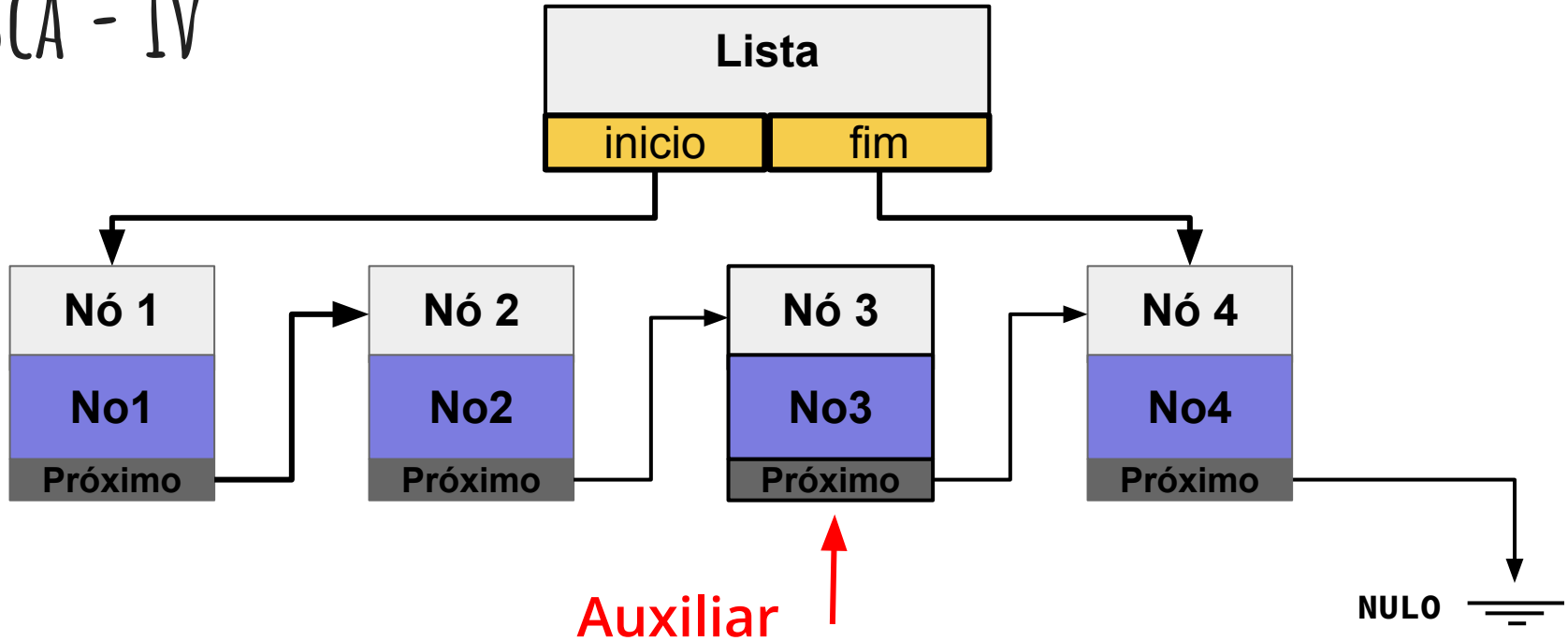
Valor desejado = No3



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

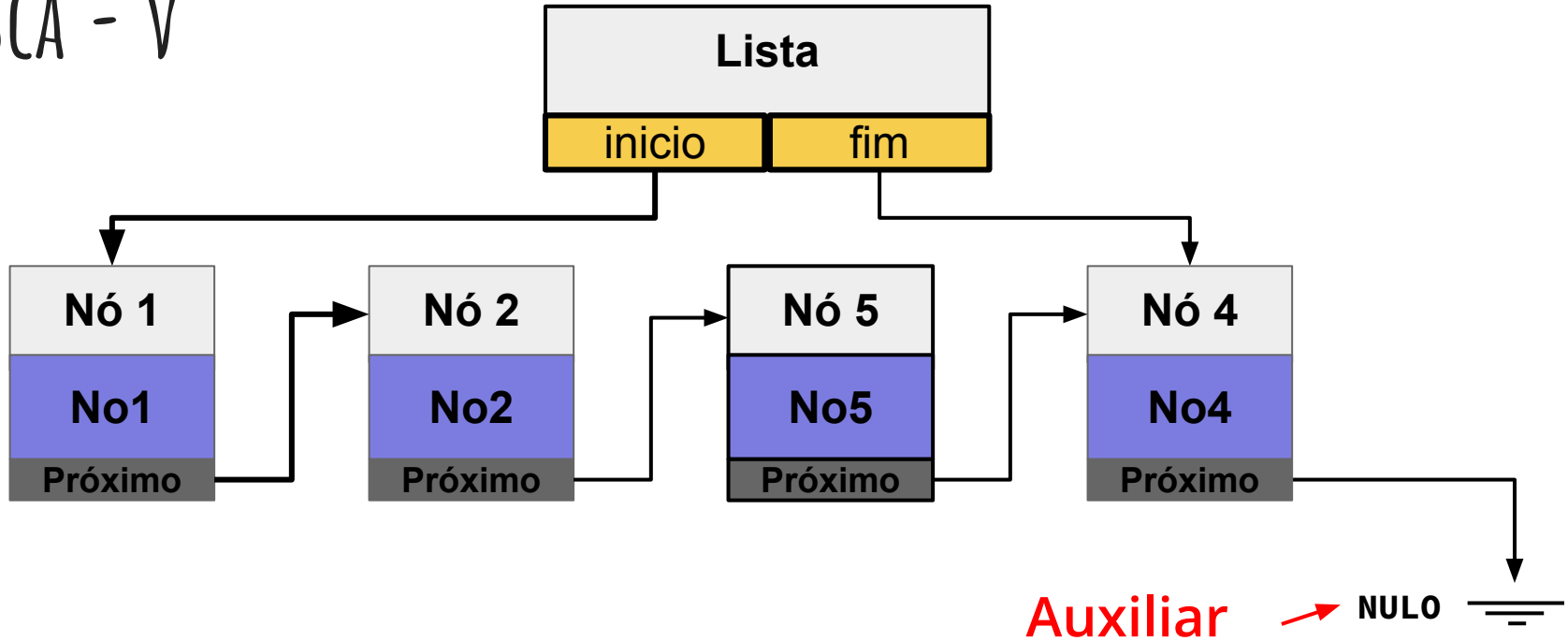
Valor desejado = No3



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 **`buscaCliente(111.111.111-11)`**, 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 elemento procurado.

Assim, procuramos o cliente, passando-se s...

Cliente 312

CPF: 111.111.111-11

Nome: João José da Silva Silva

Endereço: Rua dos Bobos, n. 0

...

BUSCA - PSEUDOCÓDIGO

buscarNaLista(valor):

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

enquanto (auxiliar ≠ NULO) {

 se auxiliar.contem(valor) {

 efetuaAcao(auxiliar); // faz ação desejada

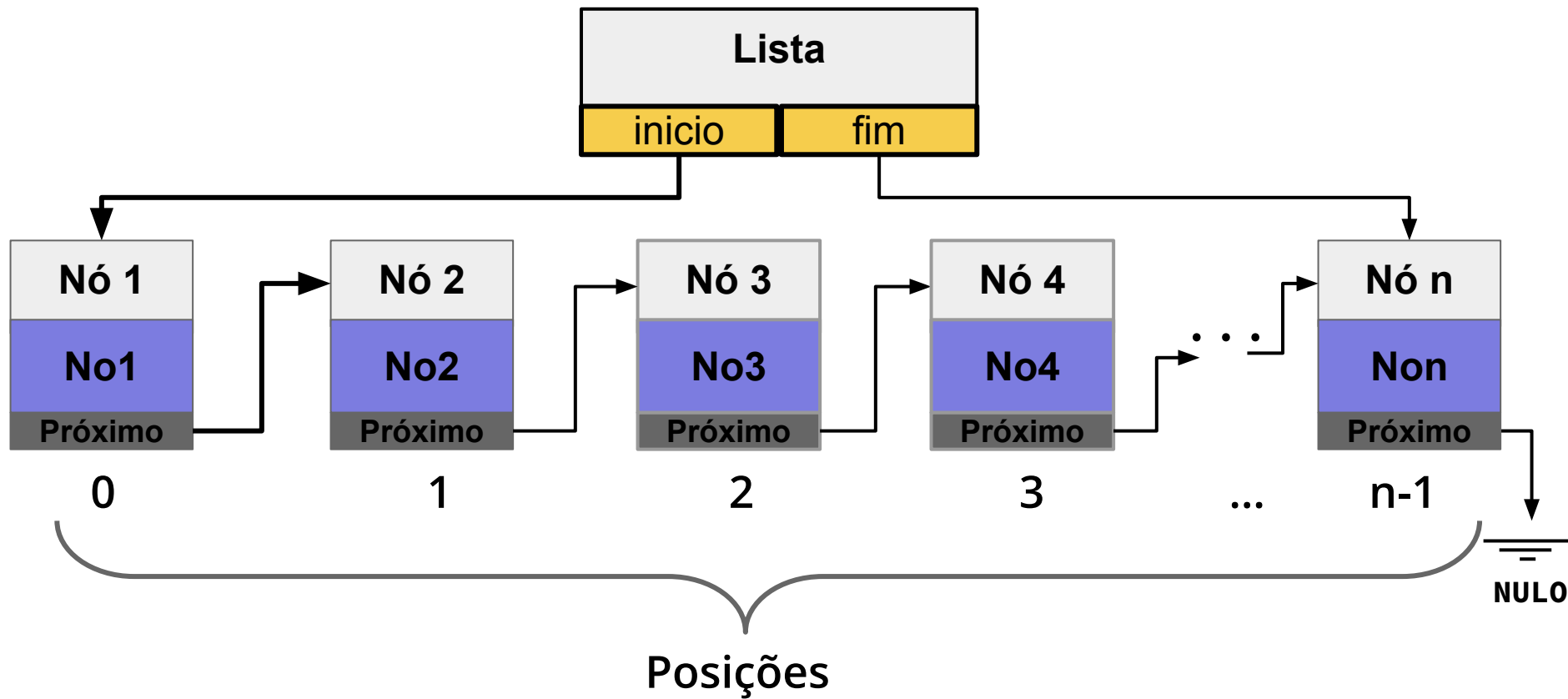
 auxiliar ← NULO; // encerra o laço

 } senão

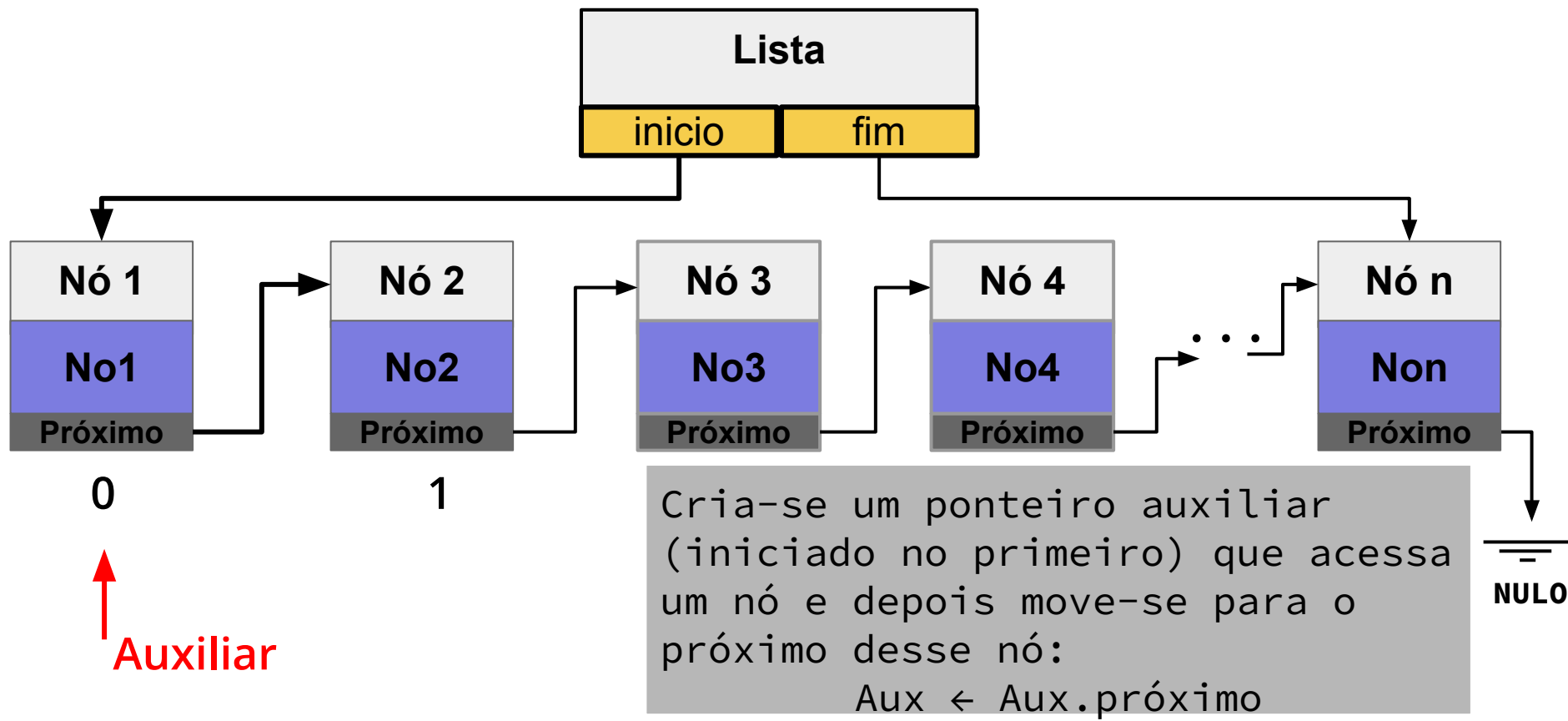
 auxiliar ← auxiliar.proximo;

}

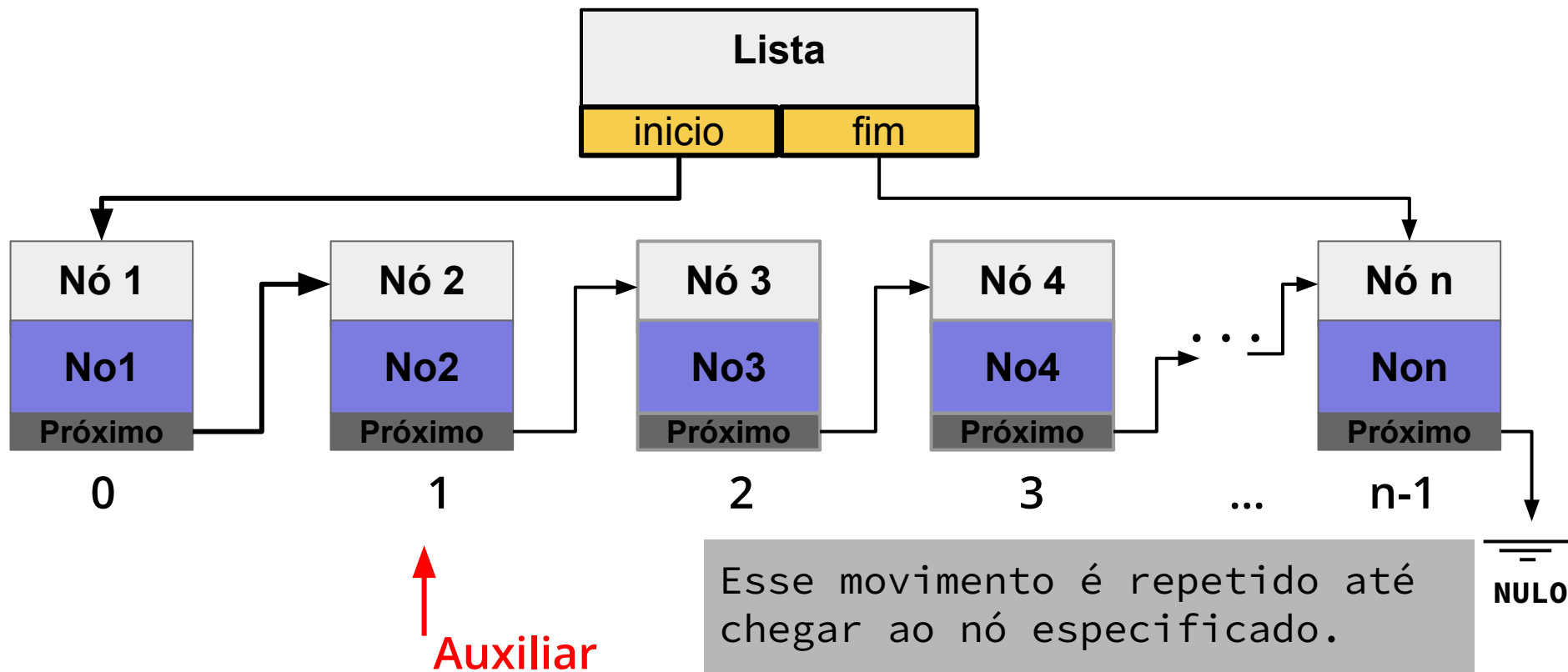
ACESSO A POSIÇÃO - I



ACESSO A POSIÇÃO - II



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):

```
// funções 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;
```

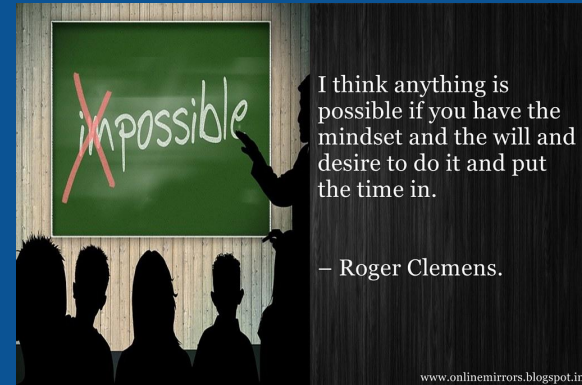
```
...
```

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

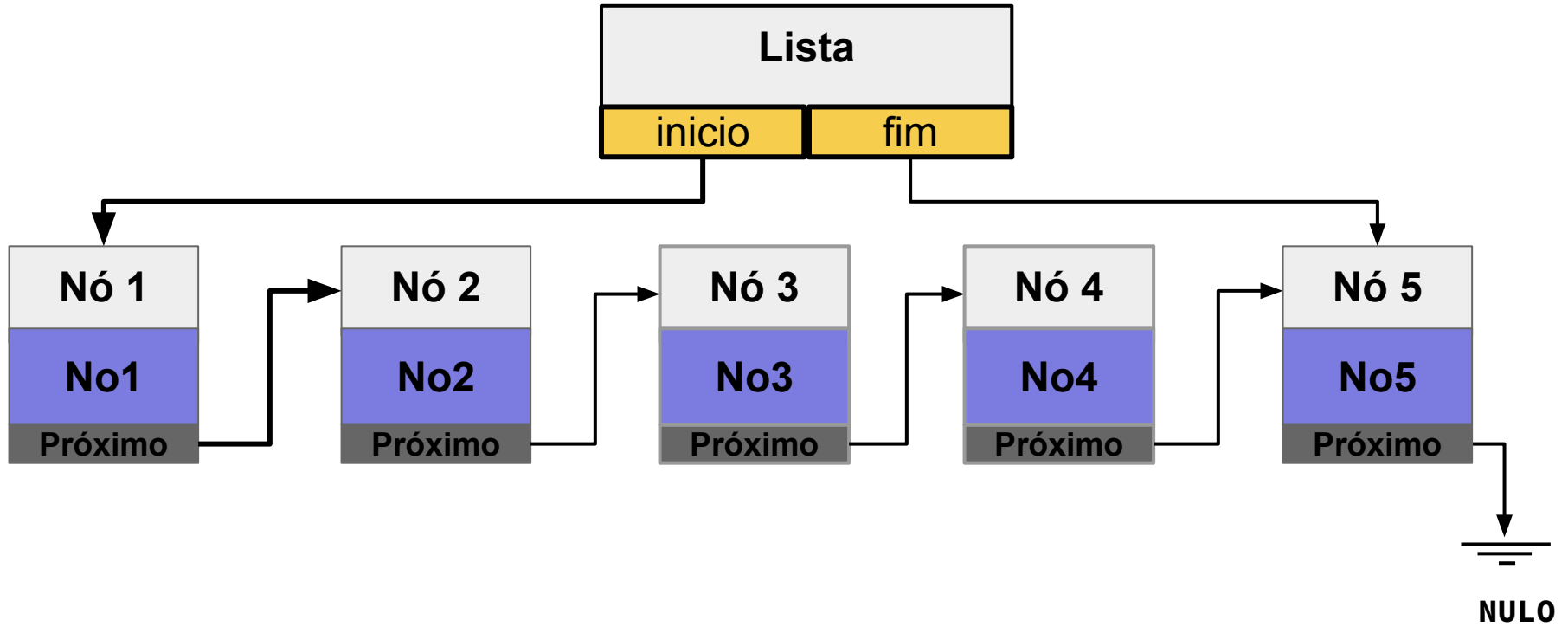
São iteradores nesse código: i, items.begin() e items.end()

MÉTODOS USUAIS EM LISTAS

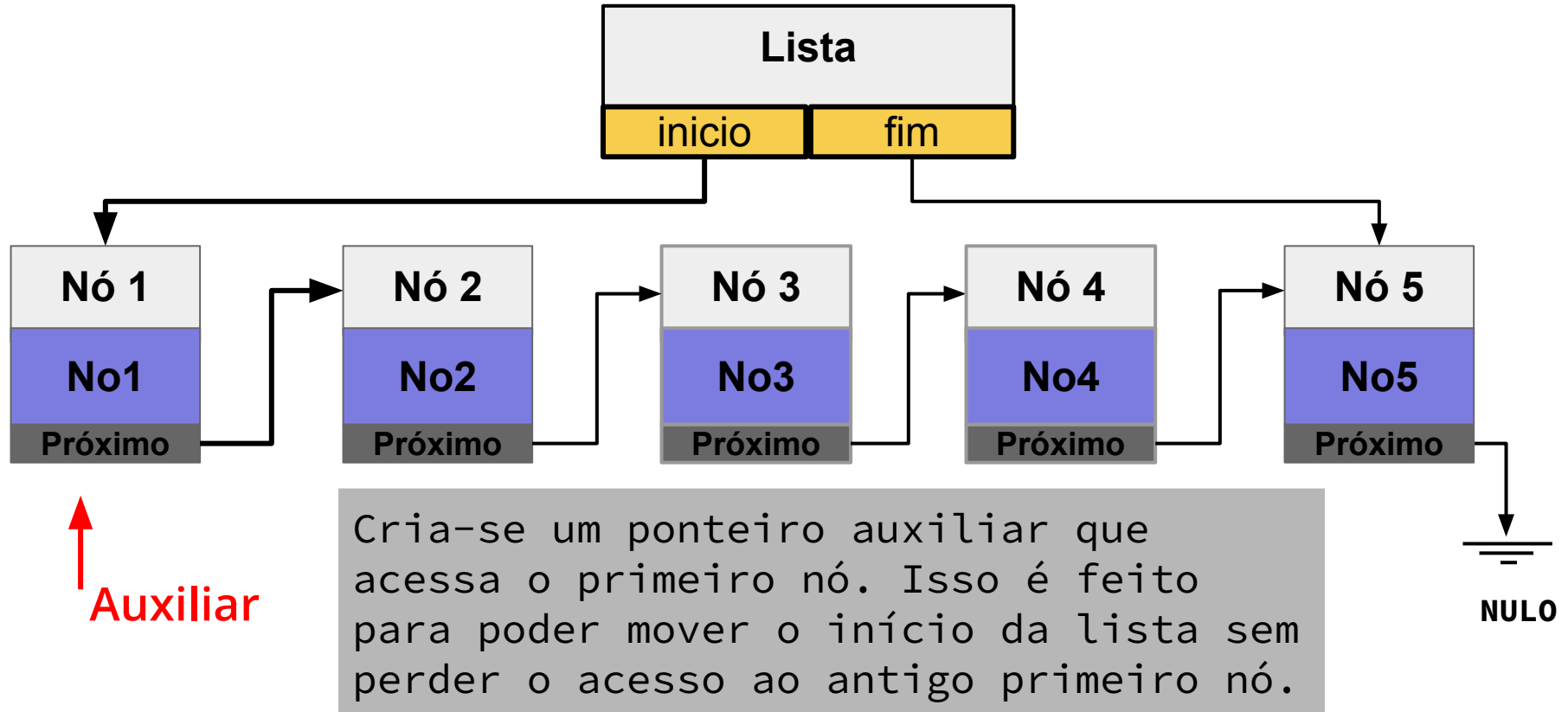
PARTE III



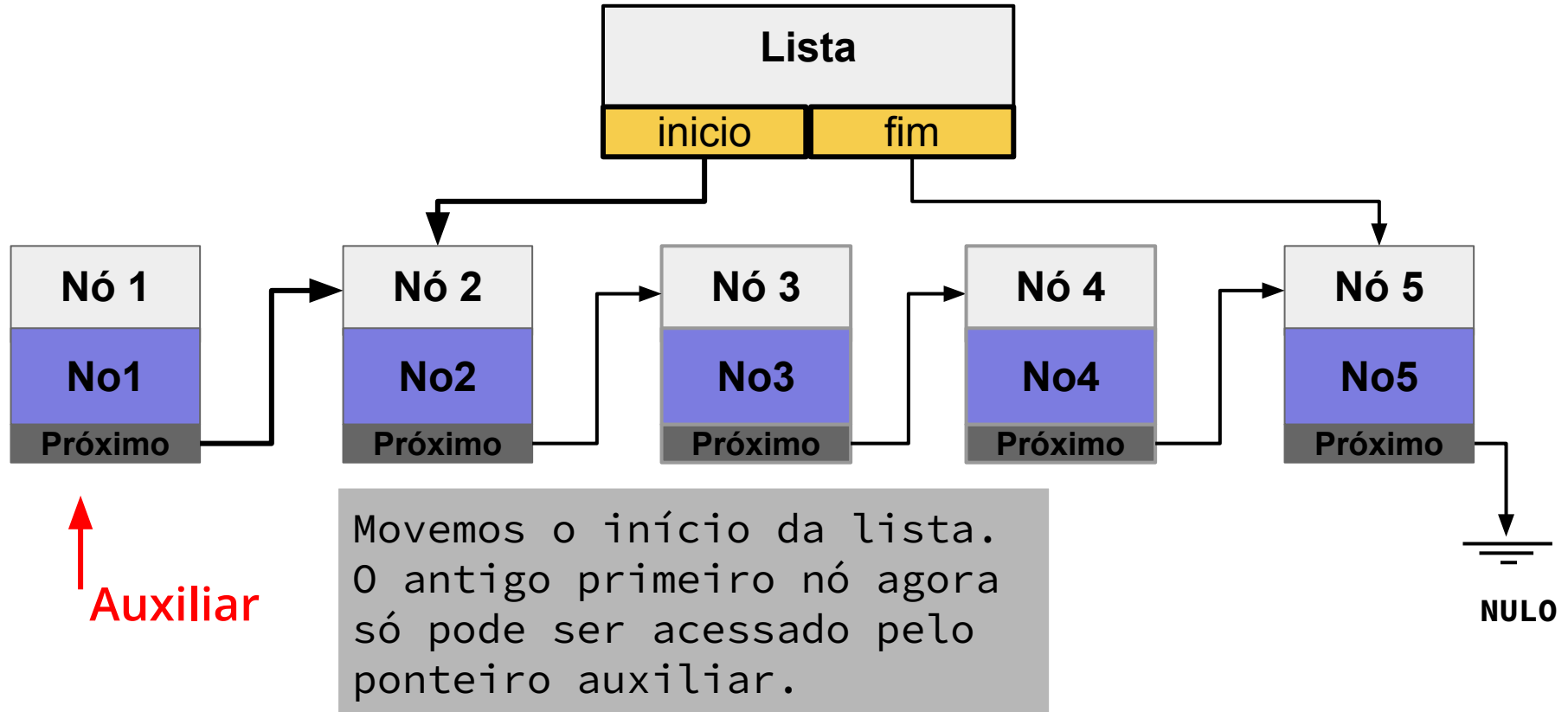
REMOVE NO INÍCIO - I



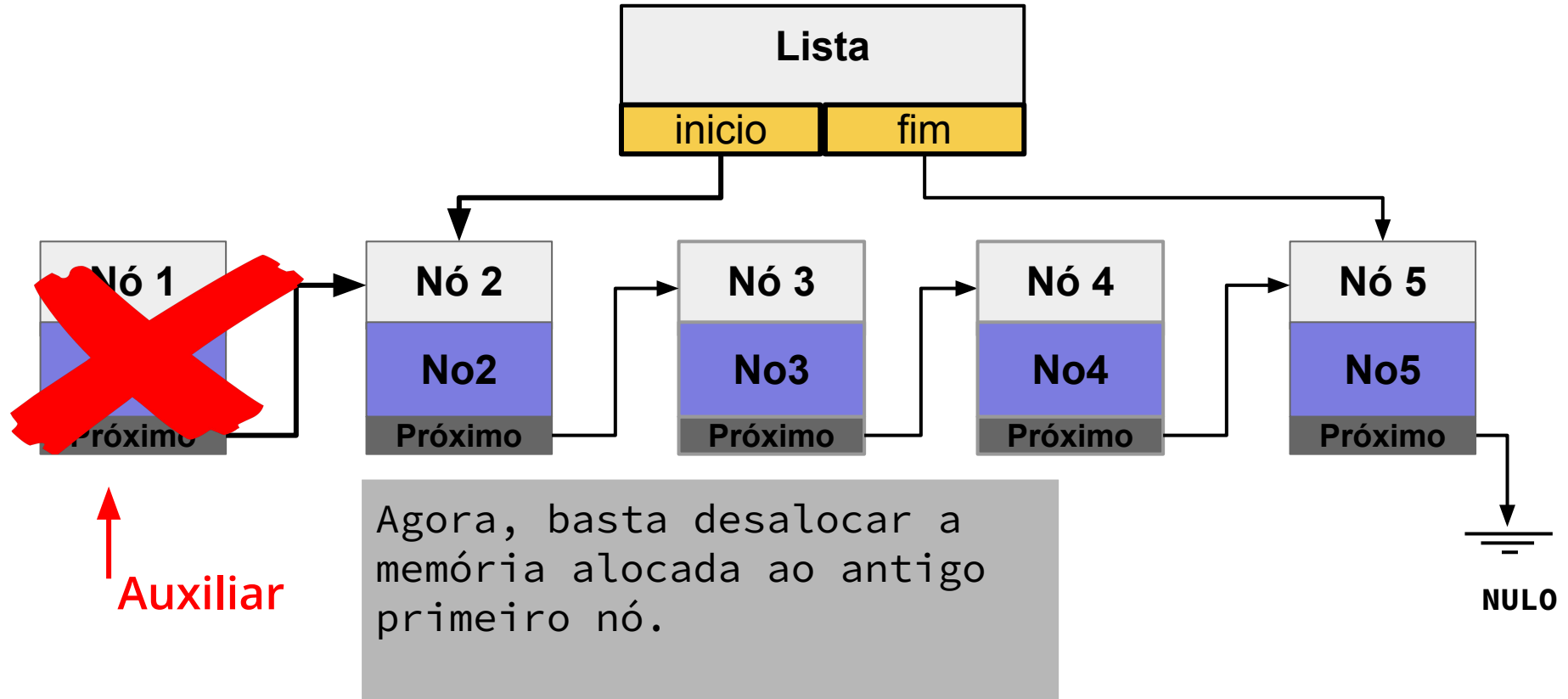
REMOVE NO INÍCIO - II



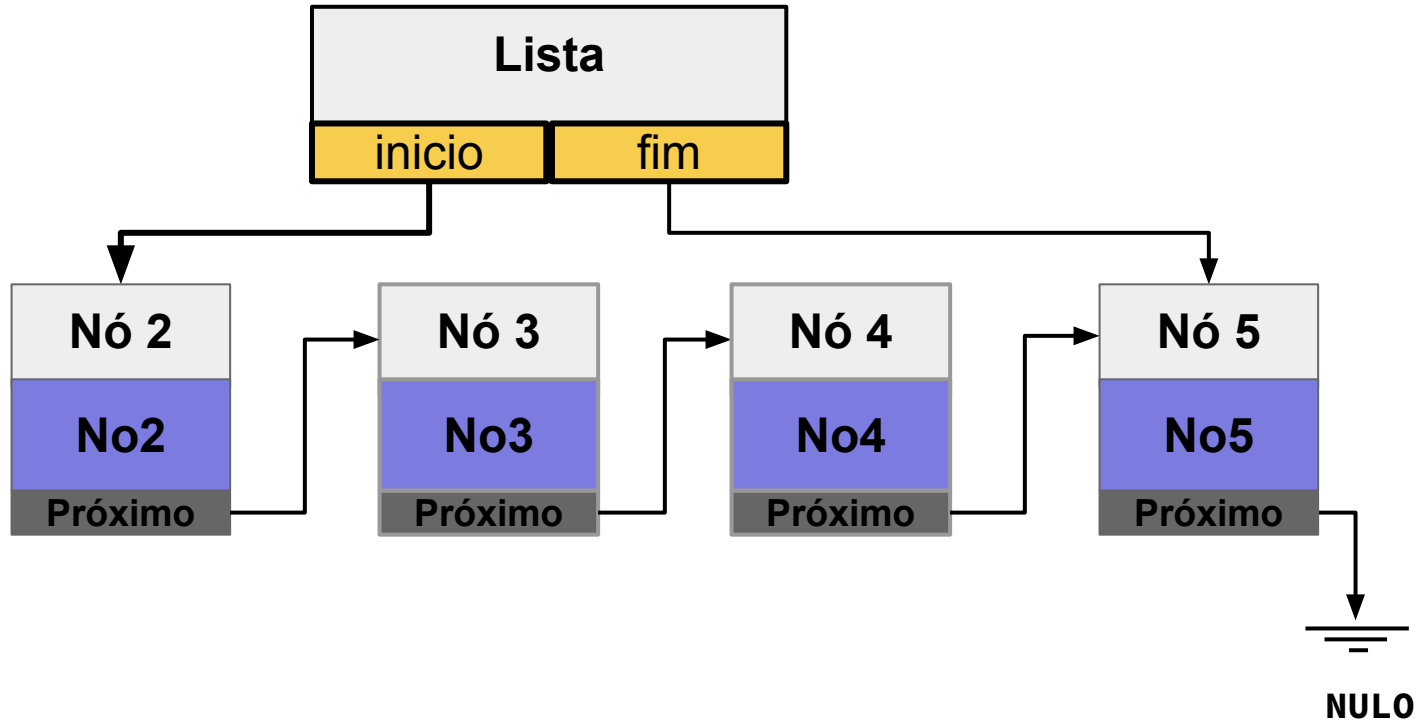
REMOVE NO INÍCIO - III



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);

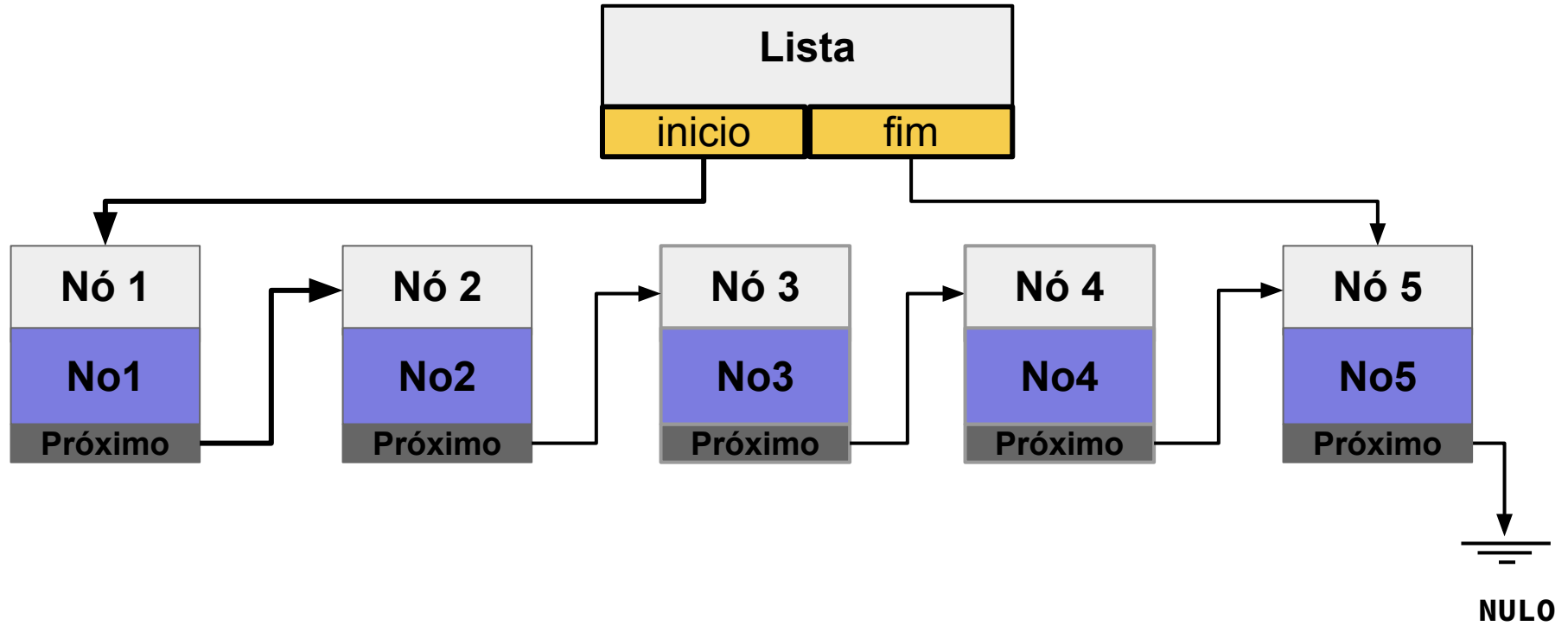
tamanho--;

se listaVazia()

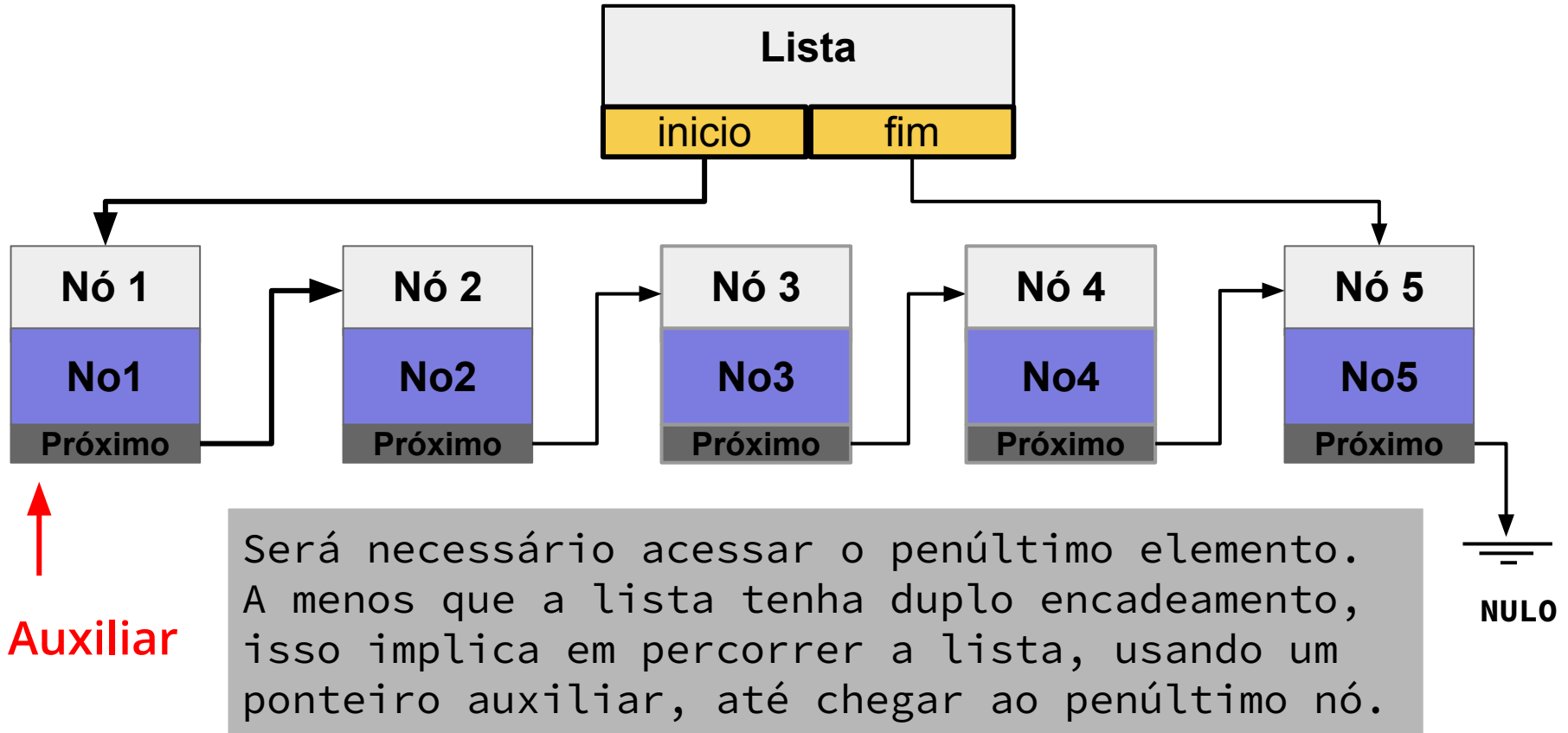
 fim ← NULO;

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

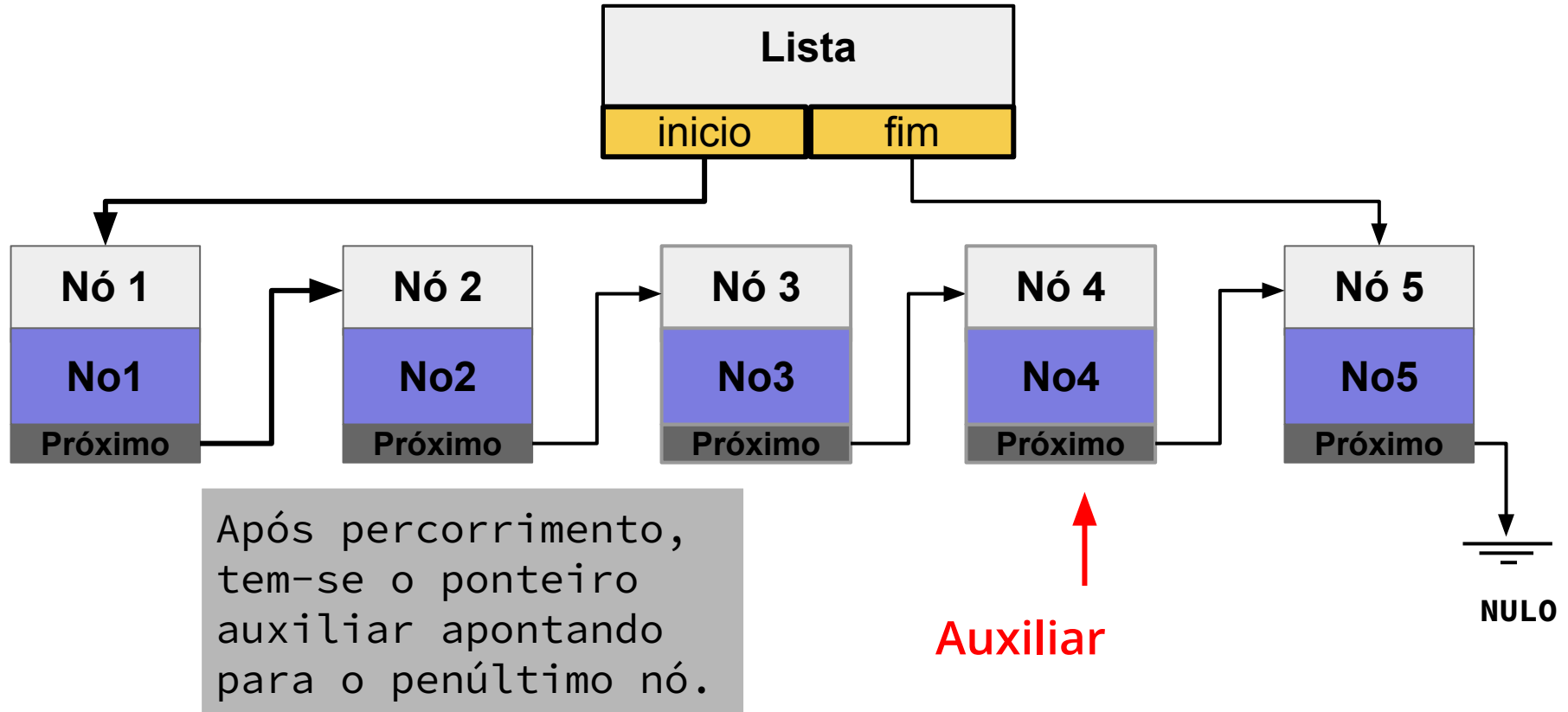
REMOVE NO FIM - I



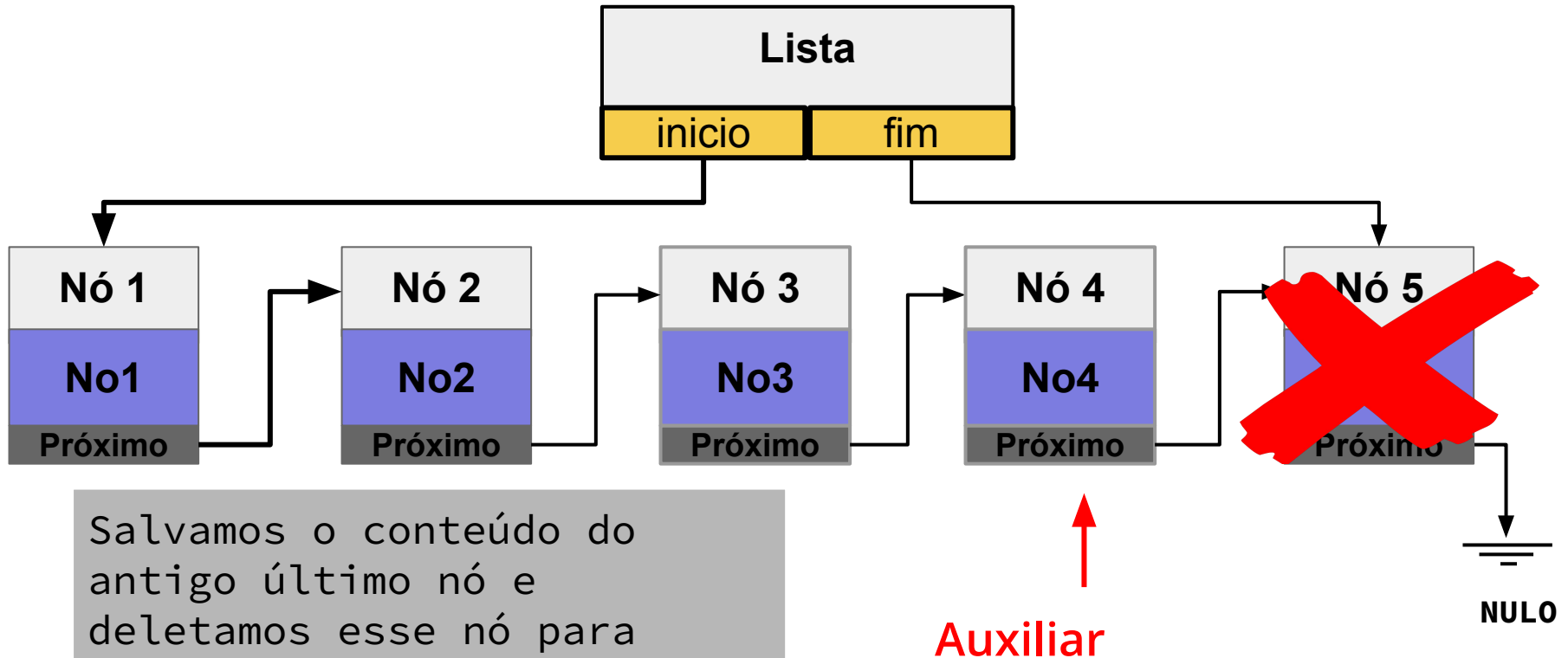
REMOVE NO FIM - II



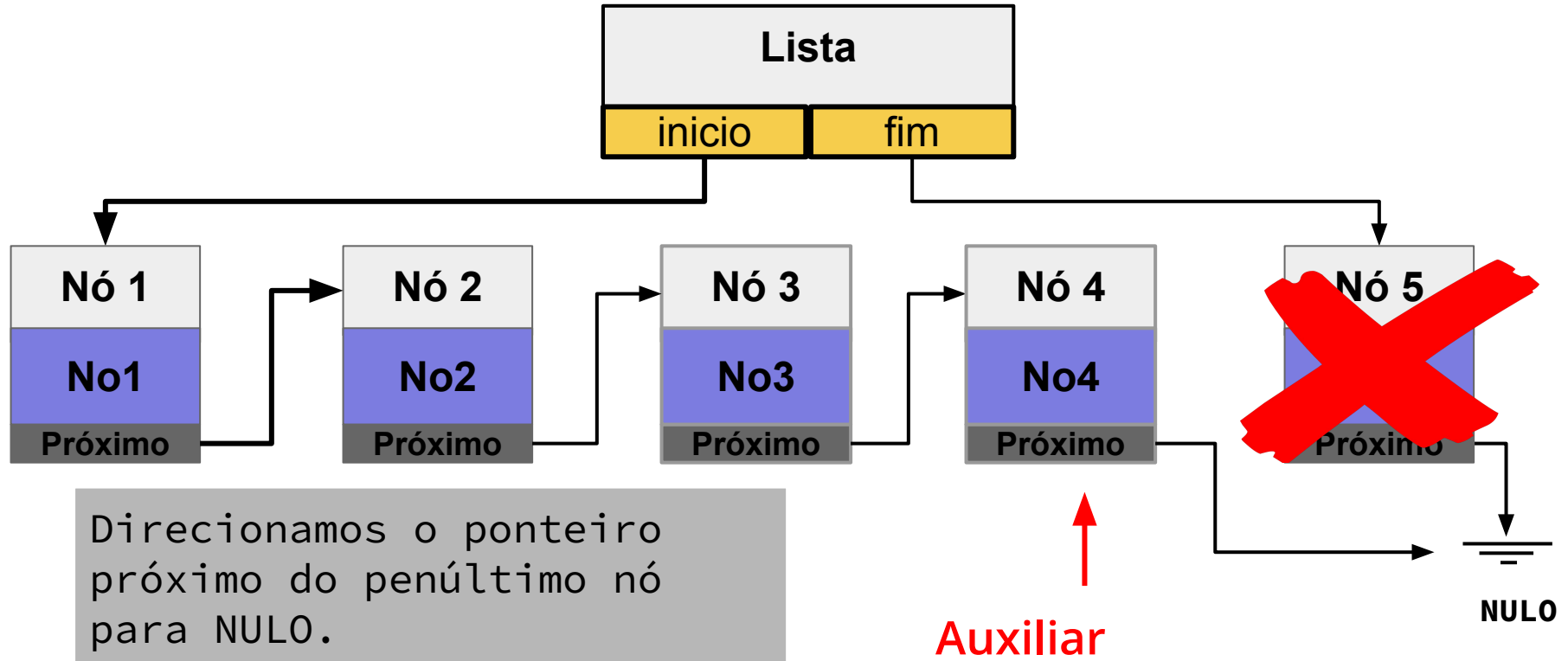
REMOVE NO FIM - III



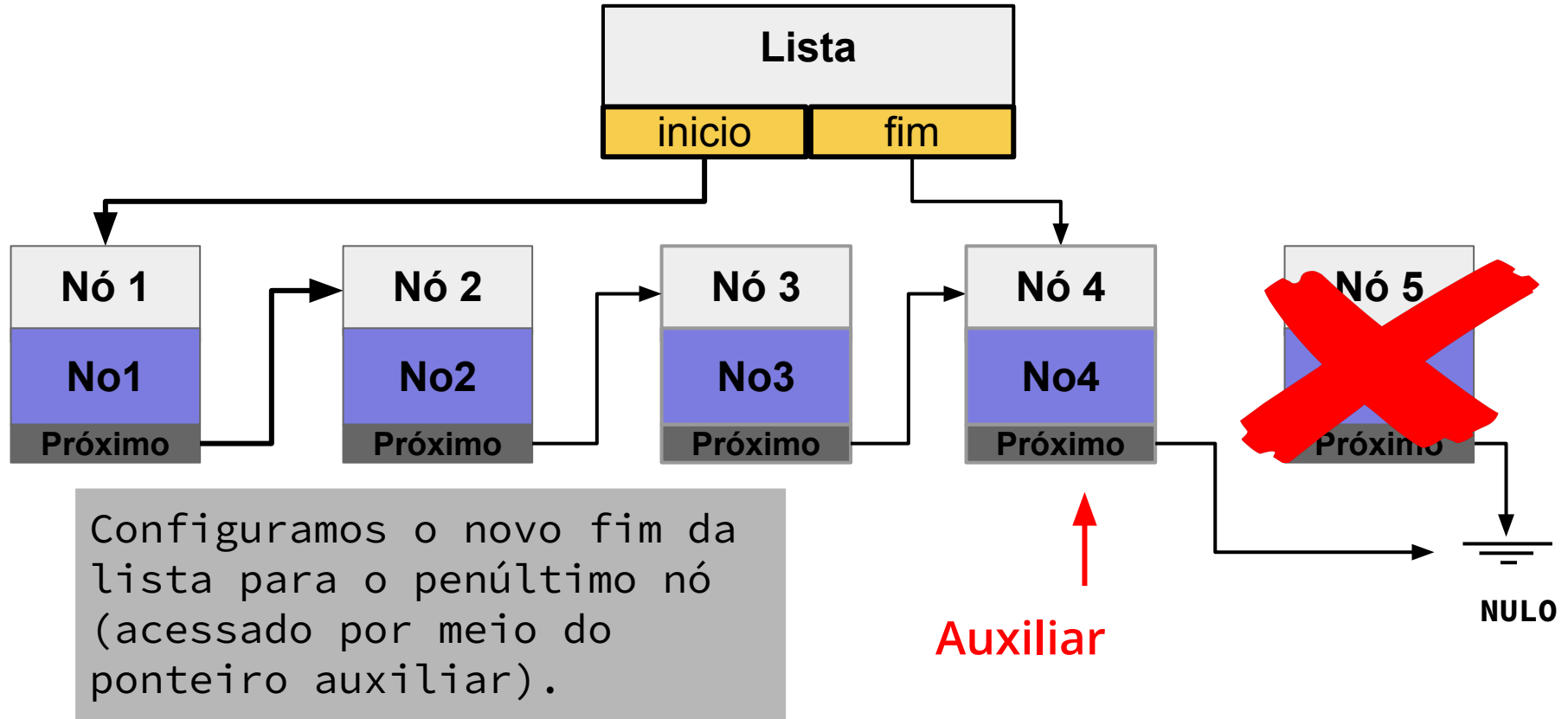
REMOVE NO FIM - IV



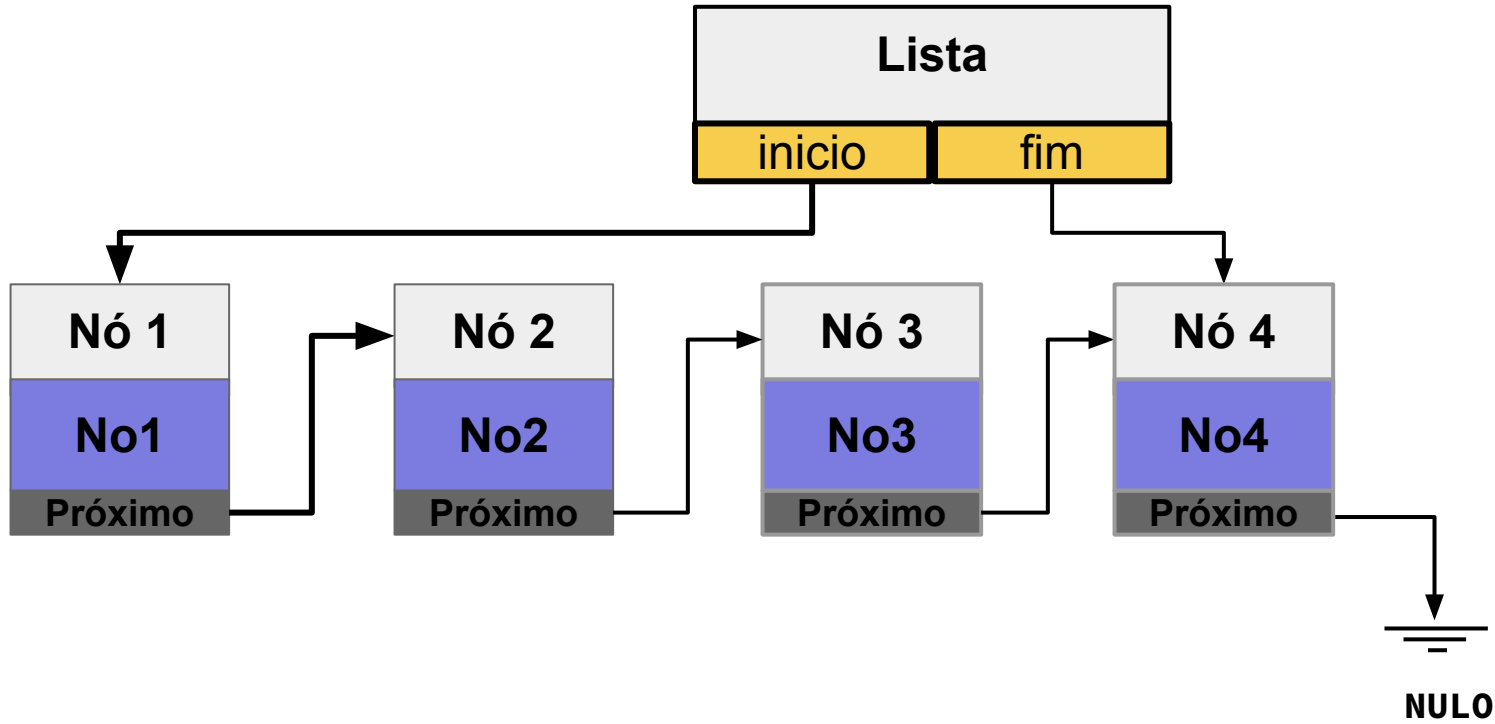
REMOVE NO FIM - V



REMOVE NO FIM - VI



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);
```

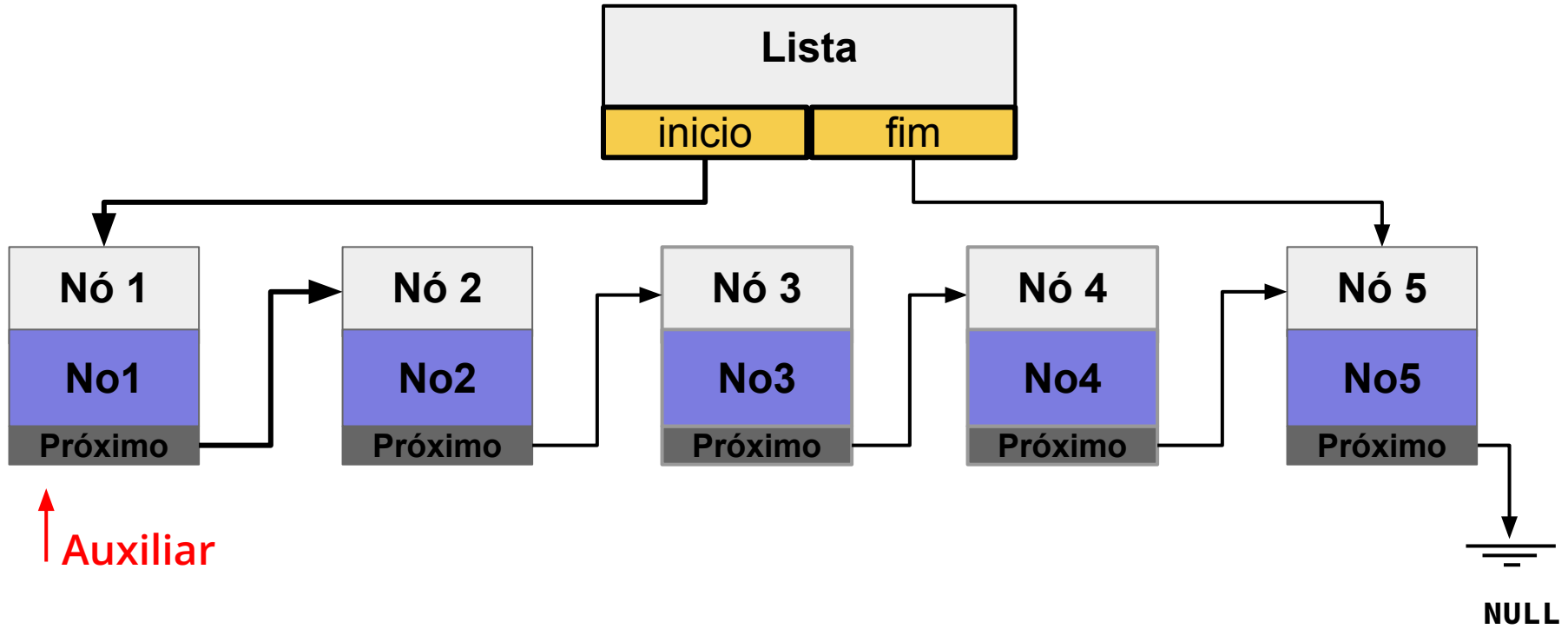
```
fim ← anterior
```

```
tamanho--;
```

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

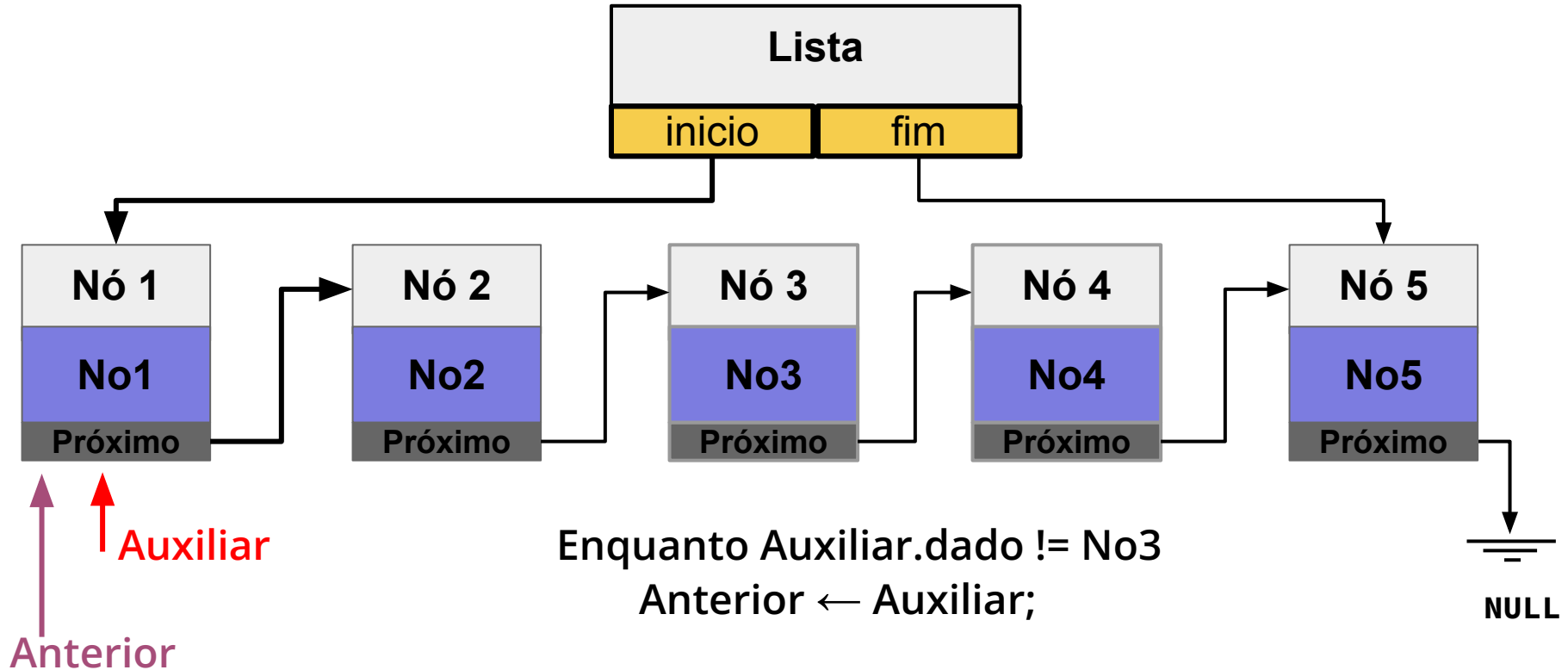
```
efetuaAcao(valor);
```

REMOVE NÓ ESPECÍFICO - I

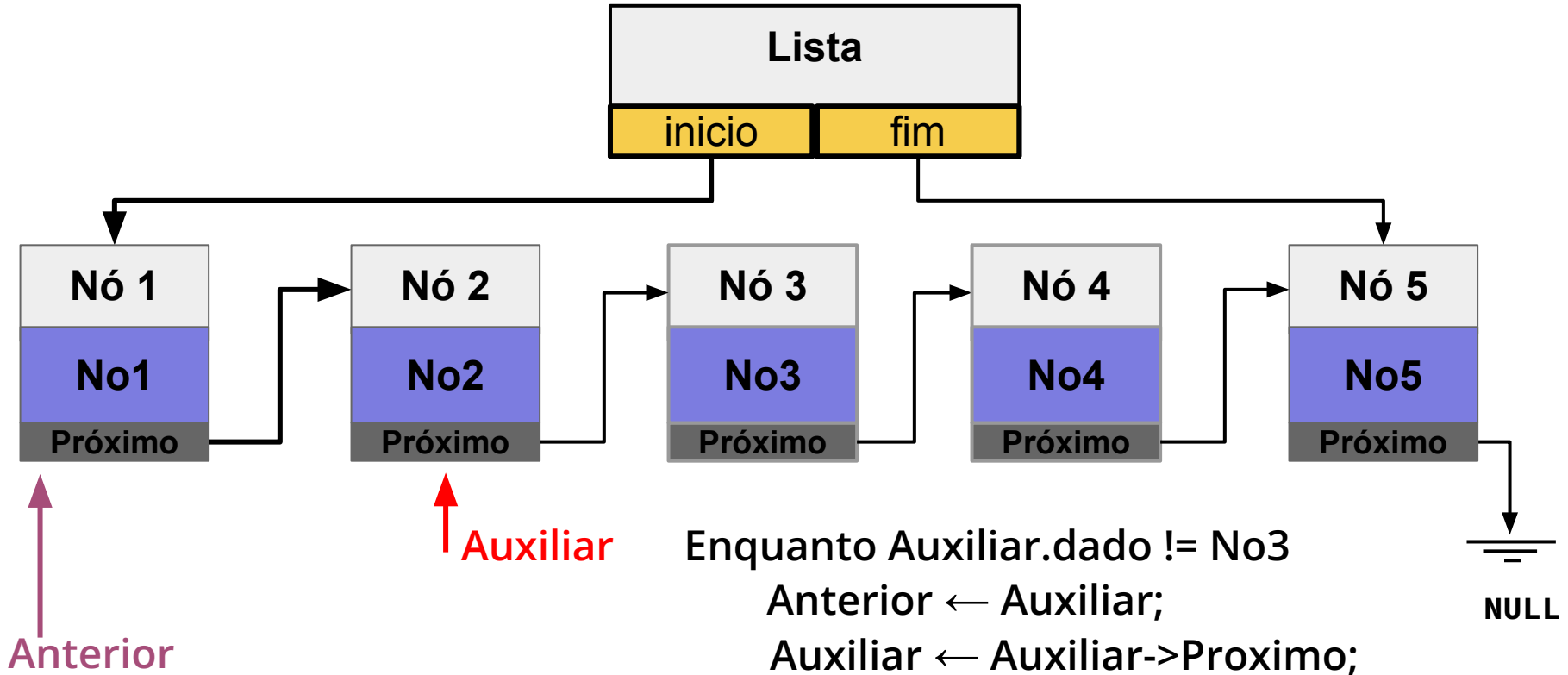


Valor a ser removido → No3

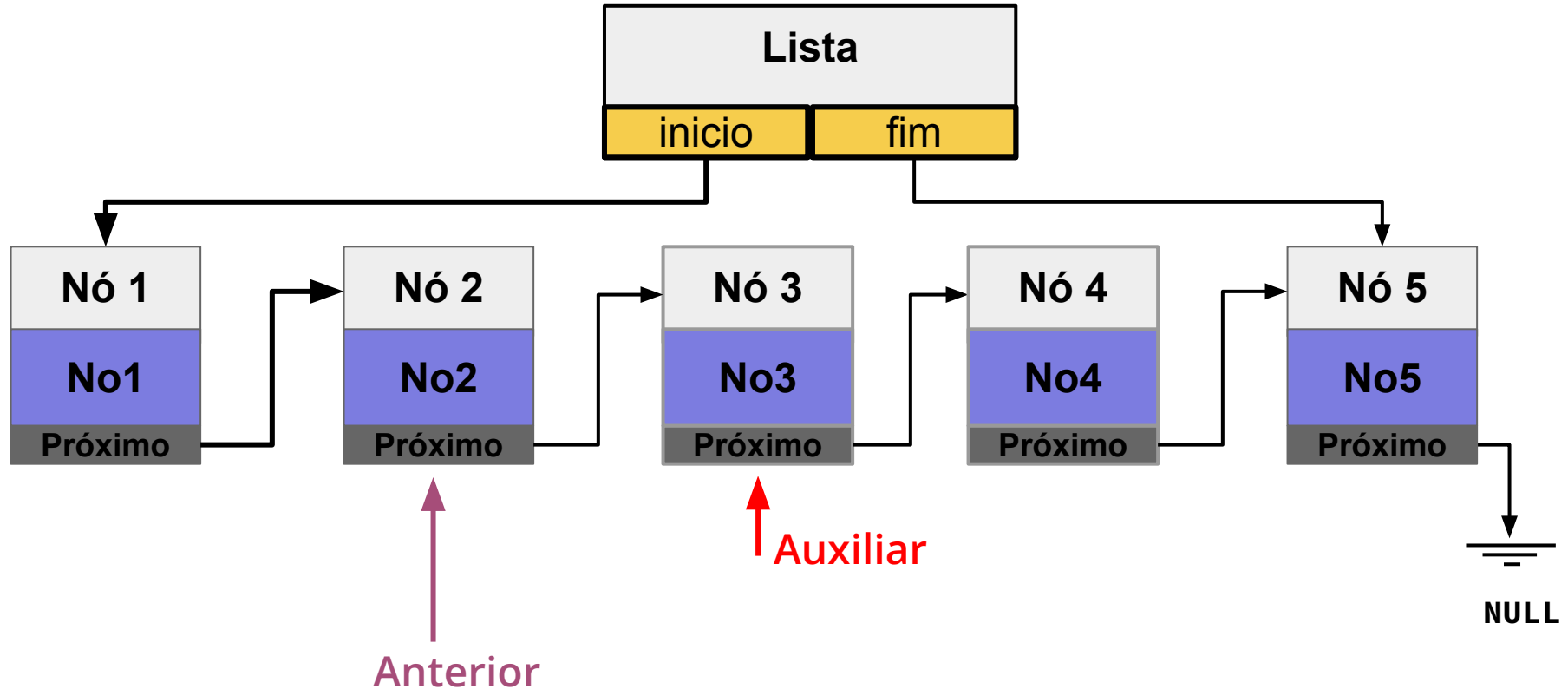
REMOVE NÓ ESPECÍFICO - II



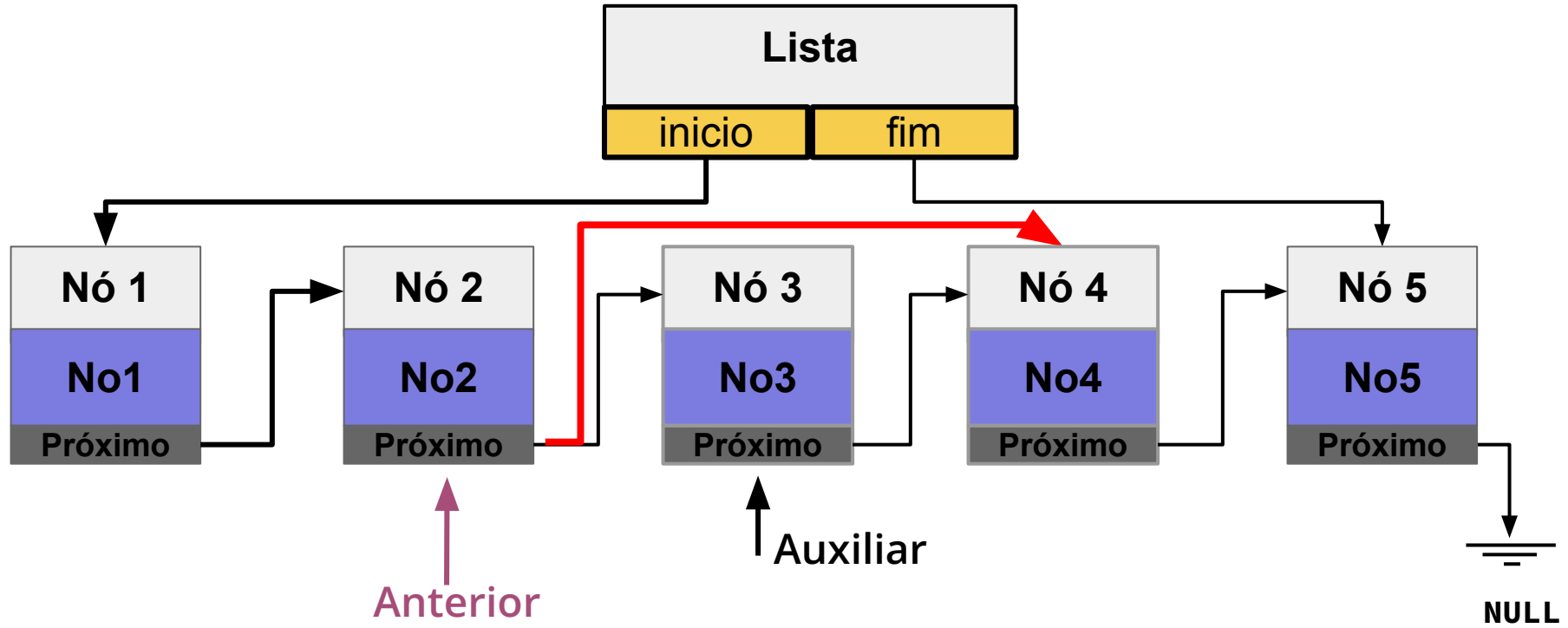
REMOVE NÓ ESPECÍFICO - III



REMOVE NÓ ESPECÍFICO - IV

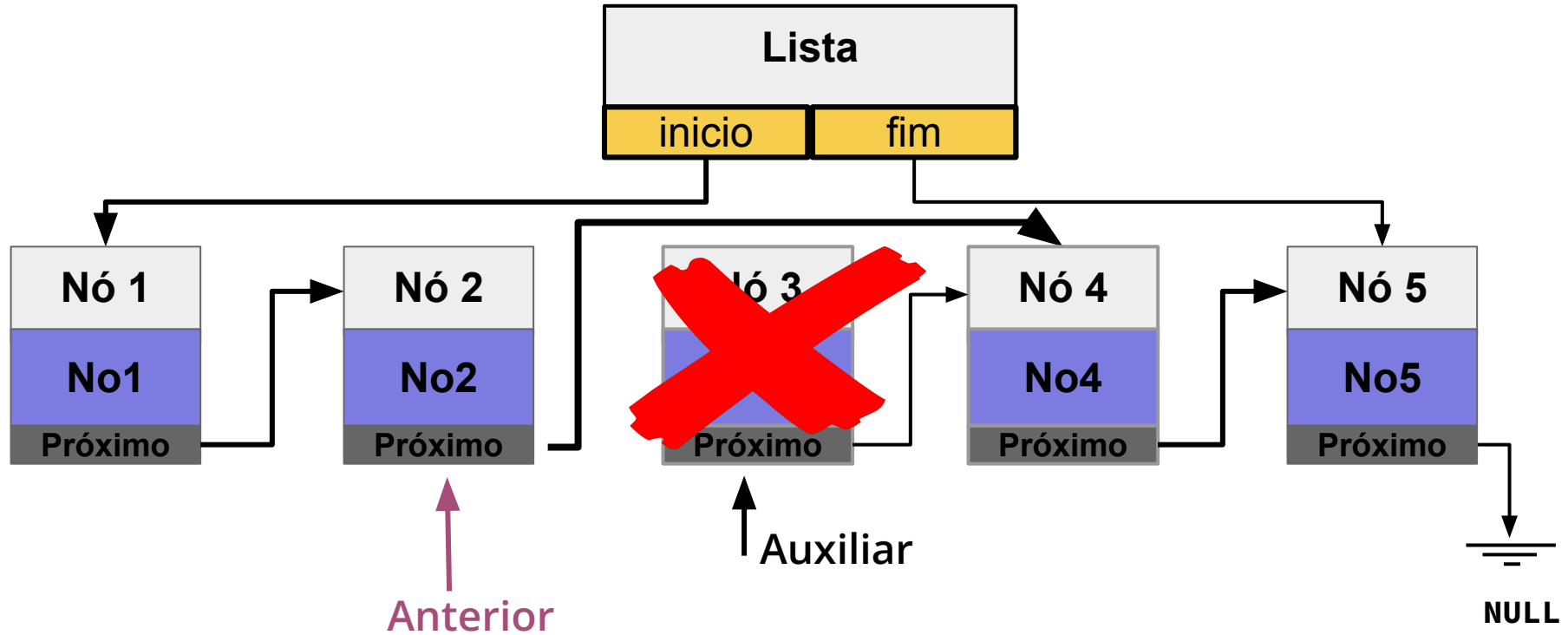


REMOVE NÓ ESPECÍFICO - V



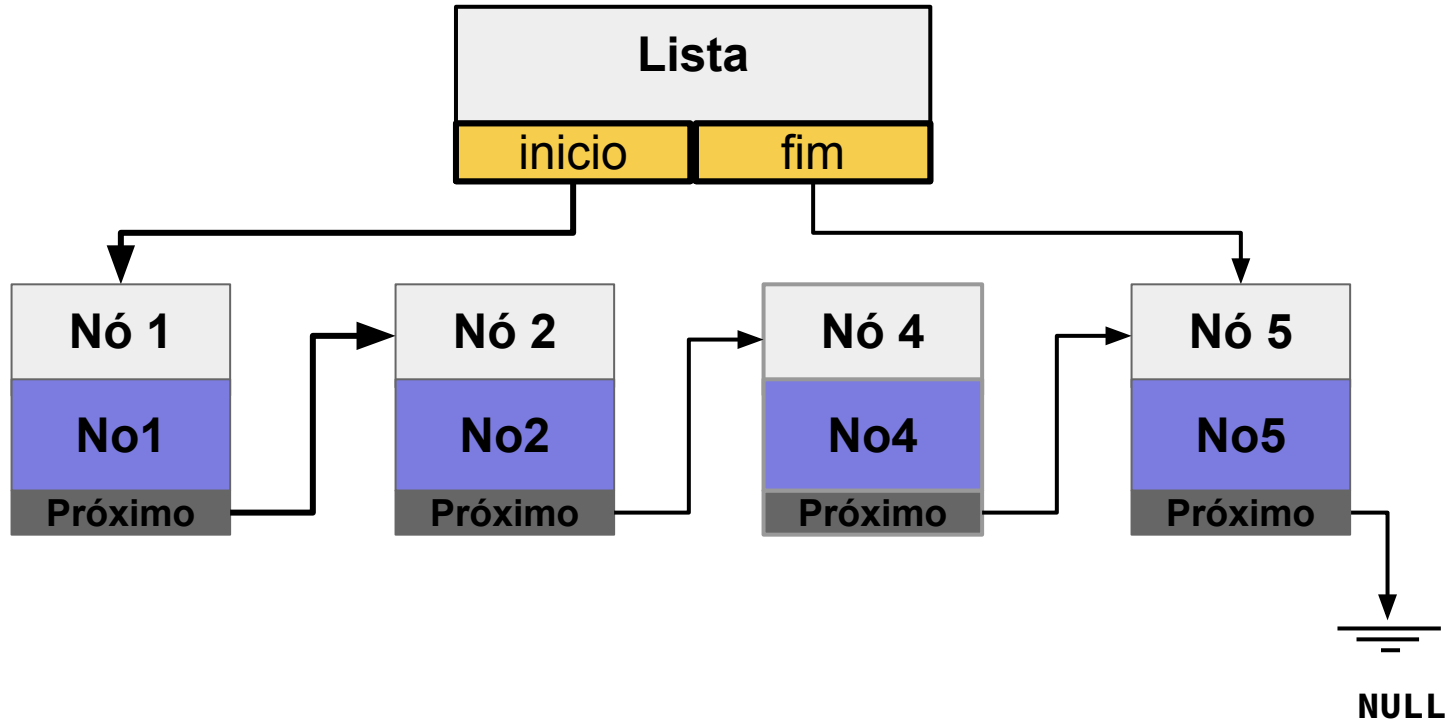
Anterior->Proximo ← Auxiliar->Proximo;

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;

}

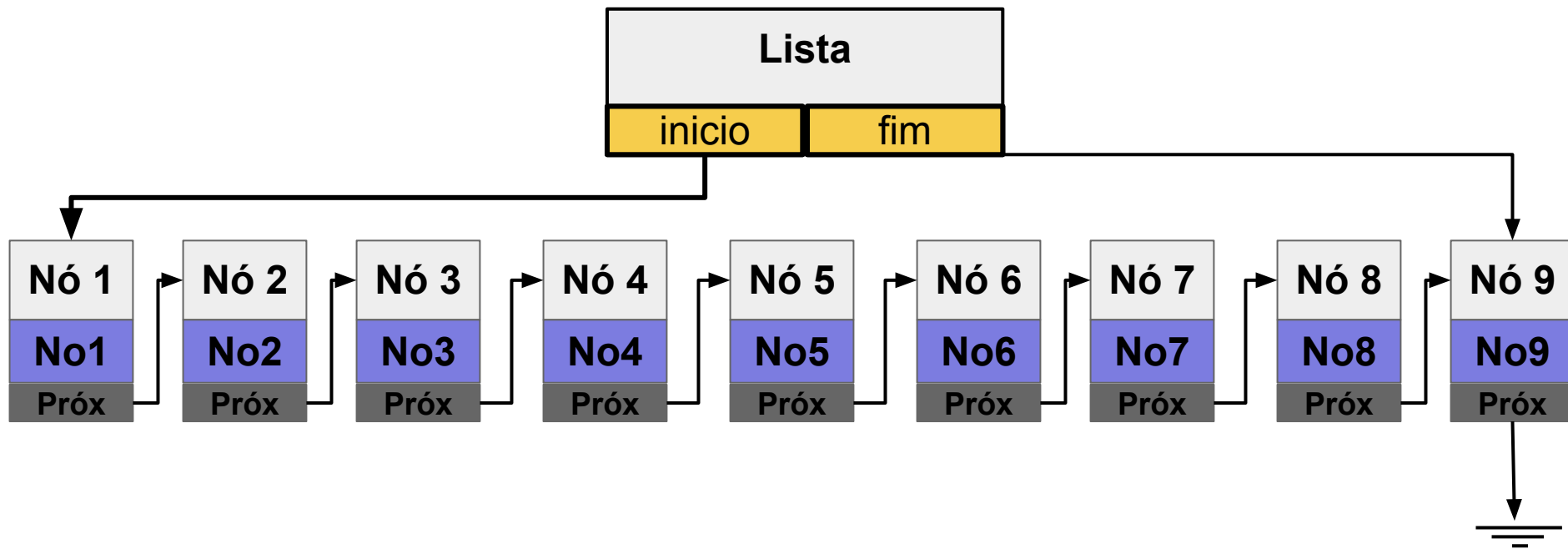
REMOVE NÓ ESPECÍFICO - PSEUDOCÓDIGO - II

```
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

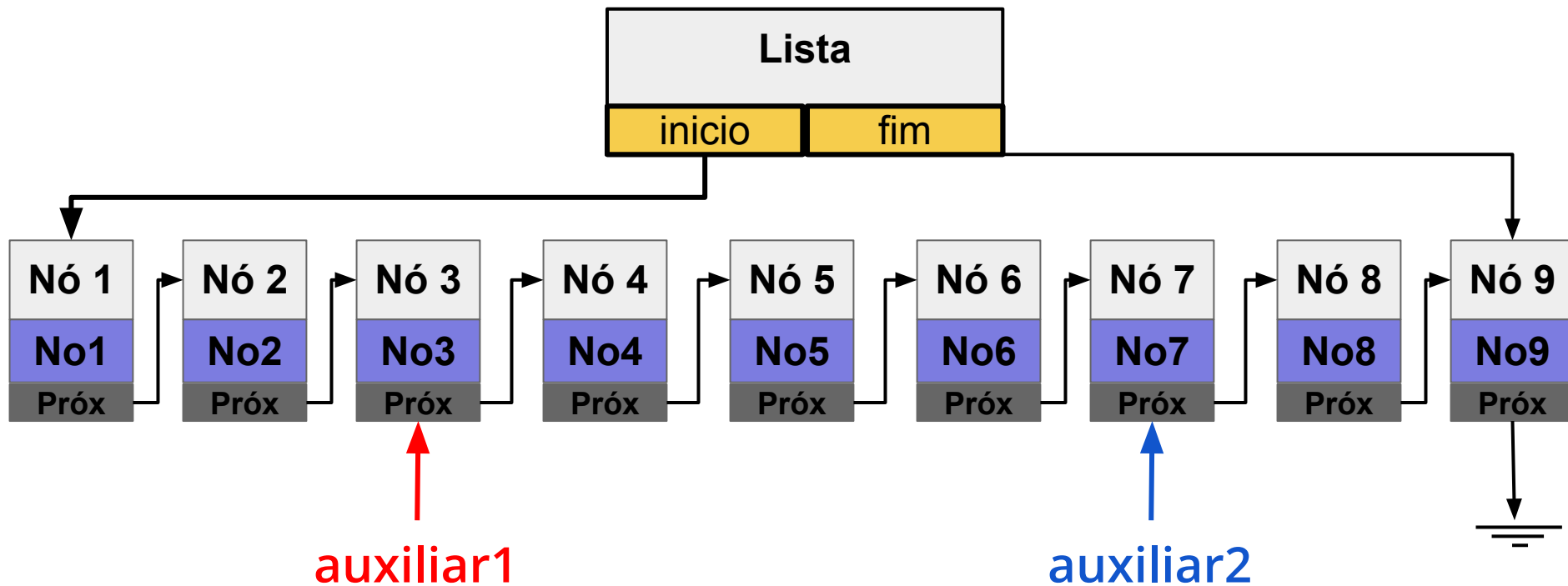


TROCAR DOIS ELEMENTOS DE POSIÇÃO - I

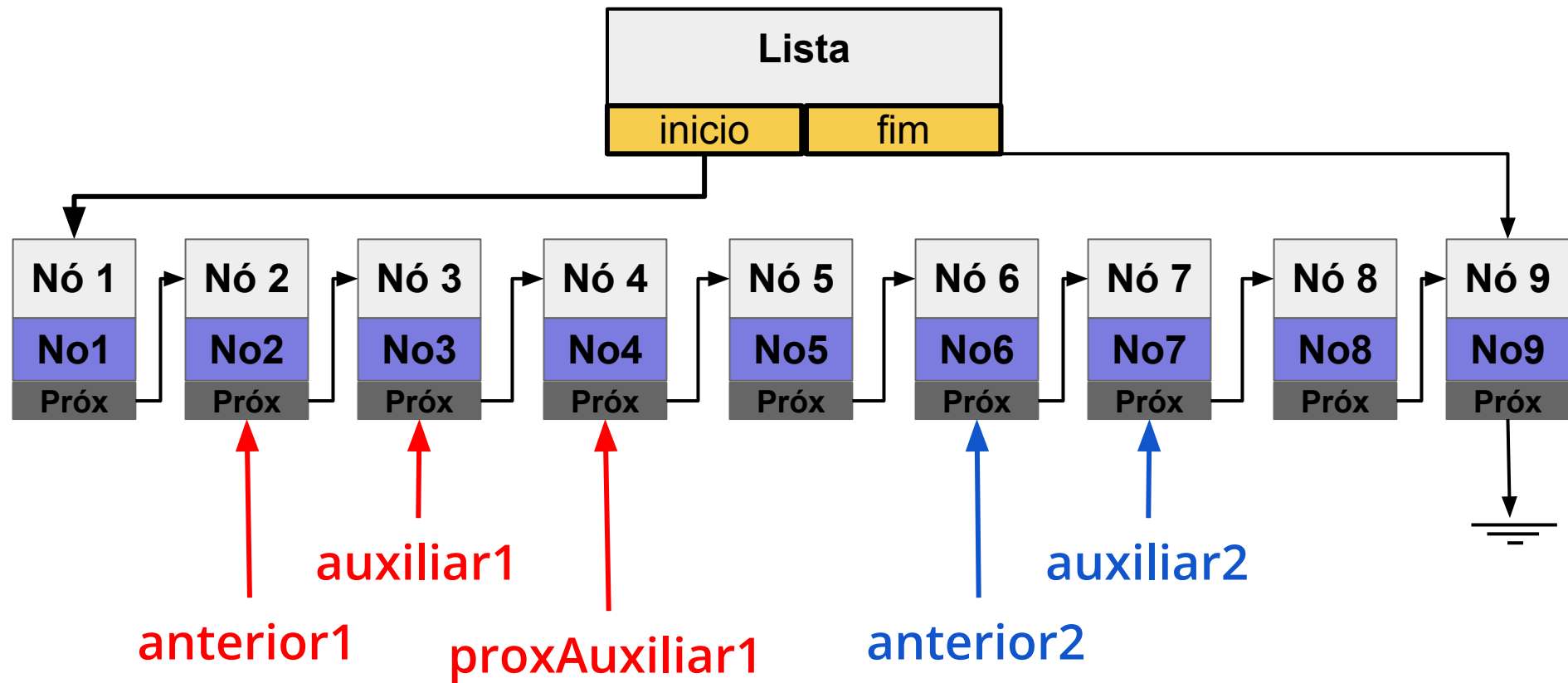


Trocar os nós 3 e 7 de posição

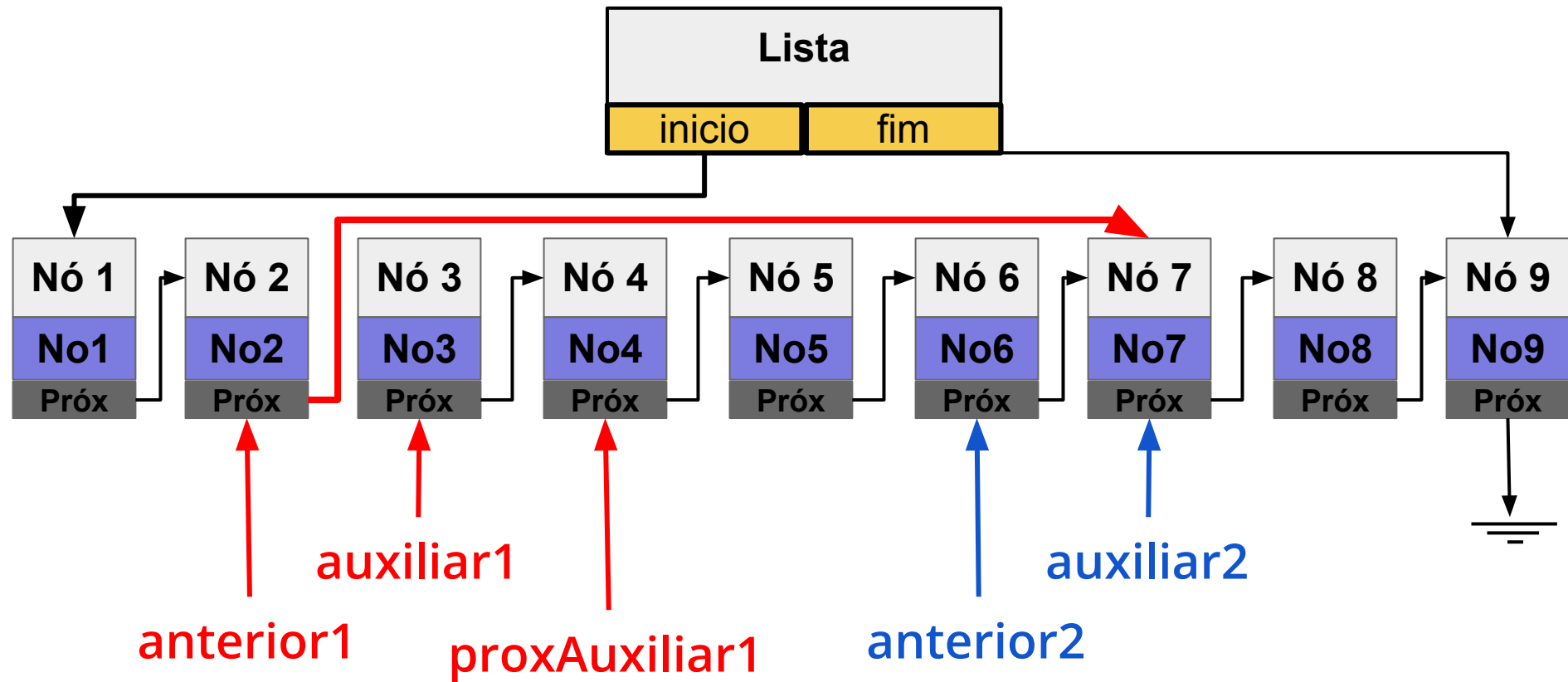
TROCAR DOIS ELEMENTOS DE POSIÇÃO - II



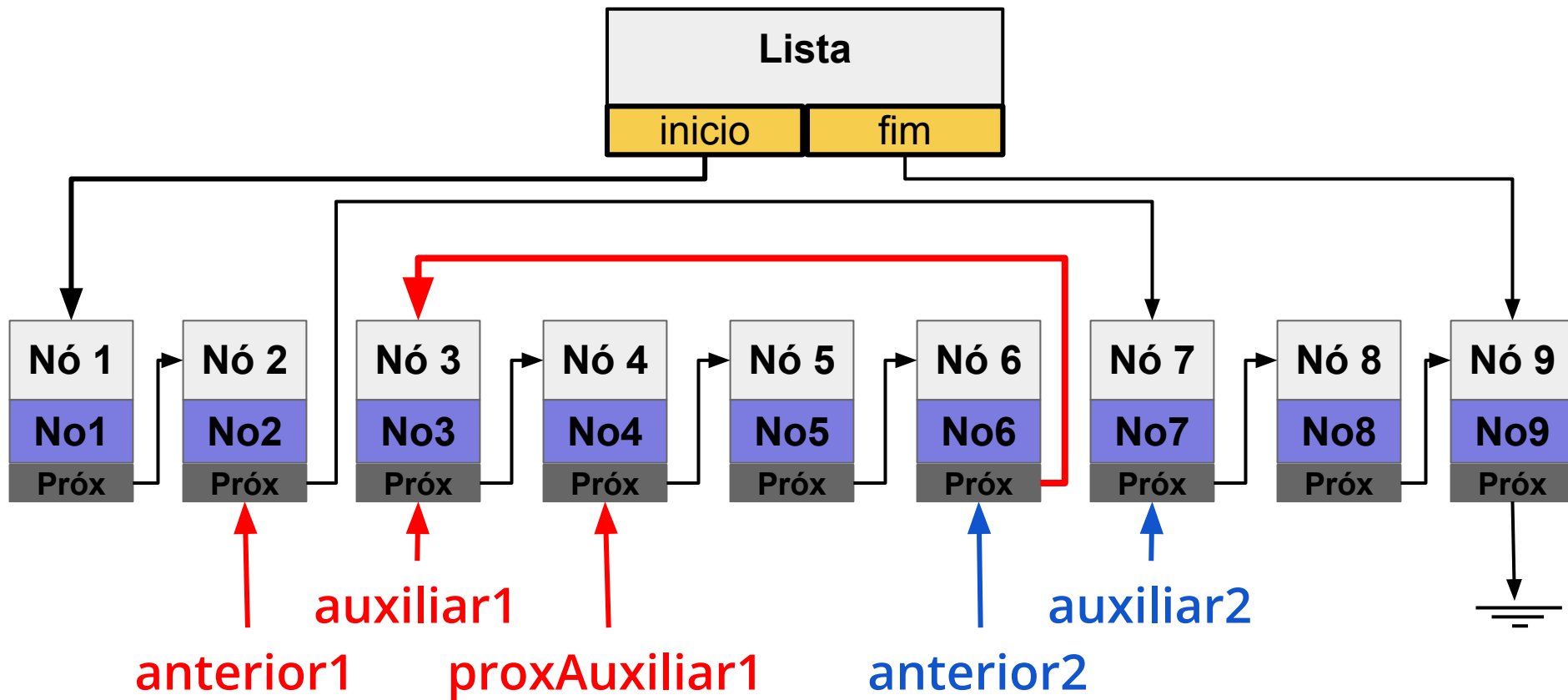
TROCAR DOIS ELEMENTOS DE POSIÇÃO - III



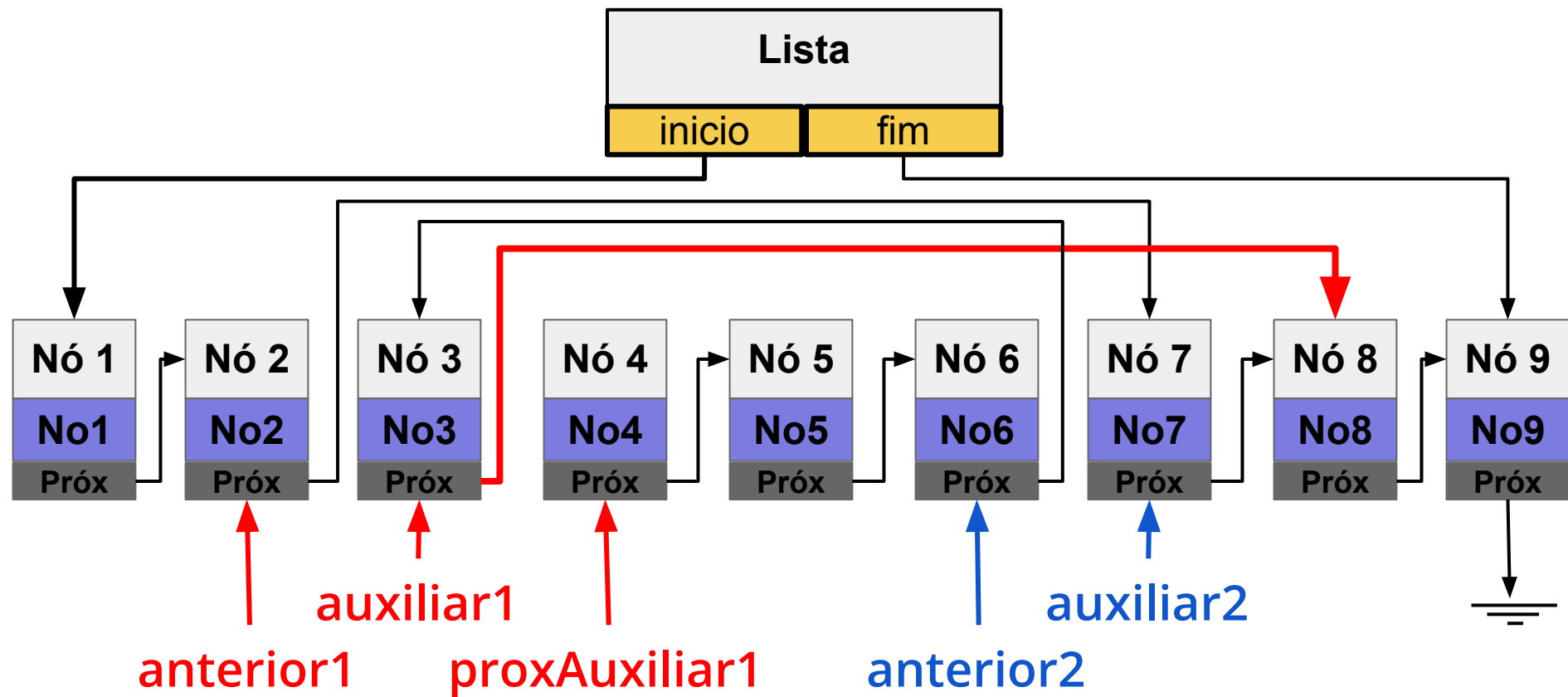
TROCAR DOIS ELEMENTOS DE POSIÇÃO - IV



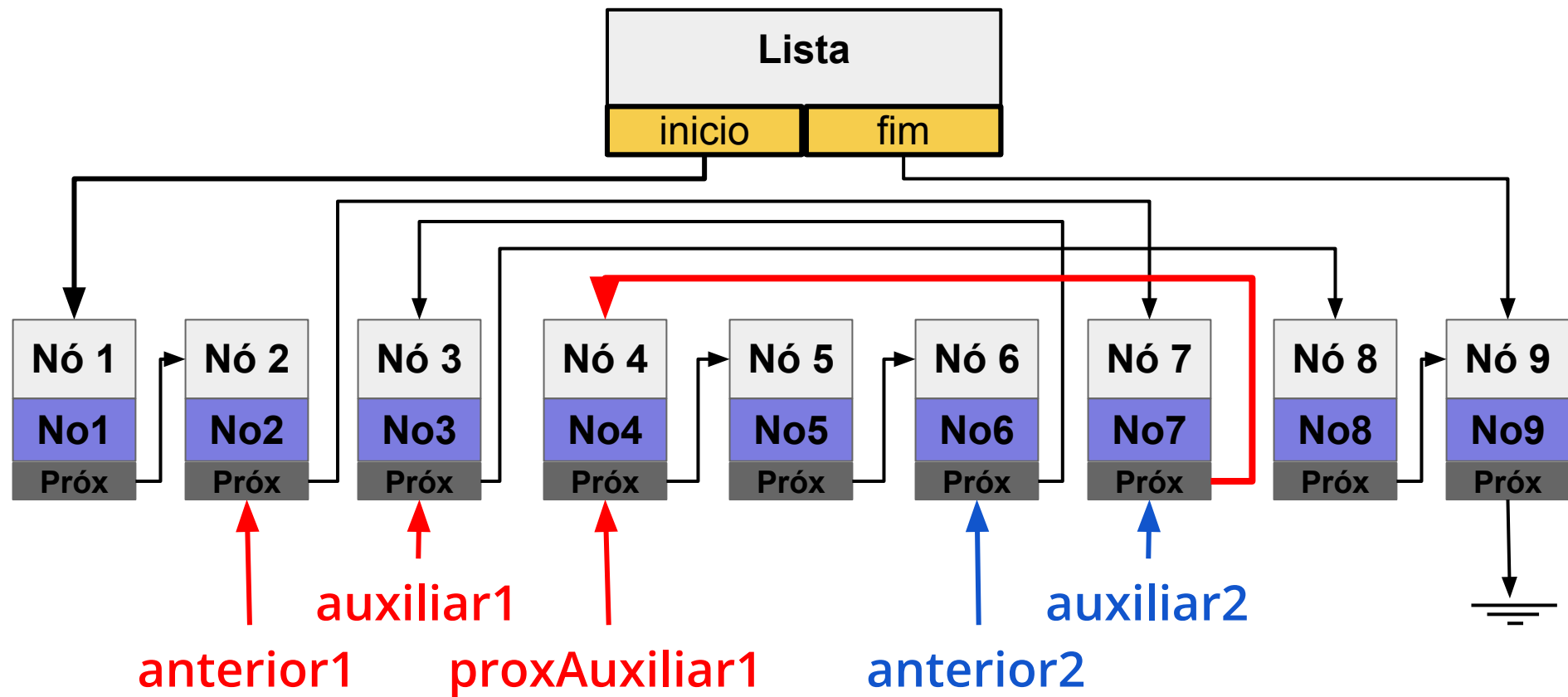
TROCAR DOIS ELEMENTOS DE POSIÇÃO - V



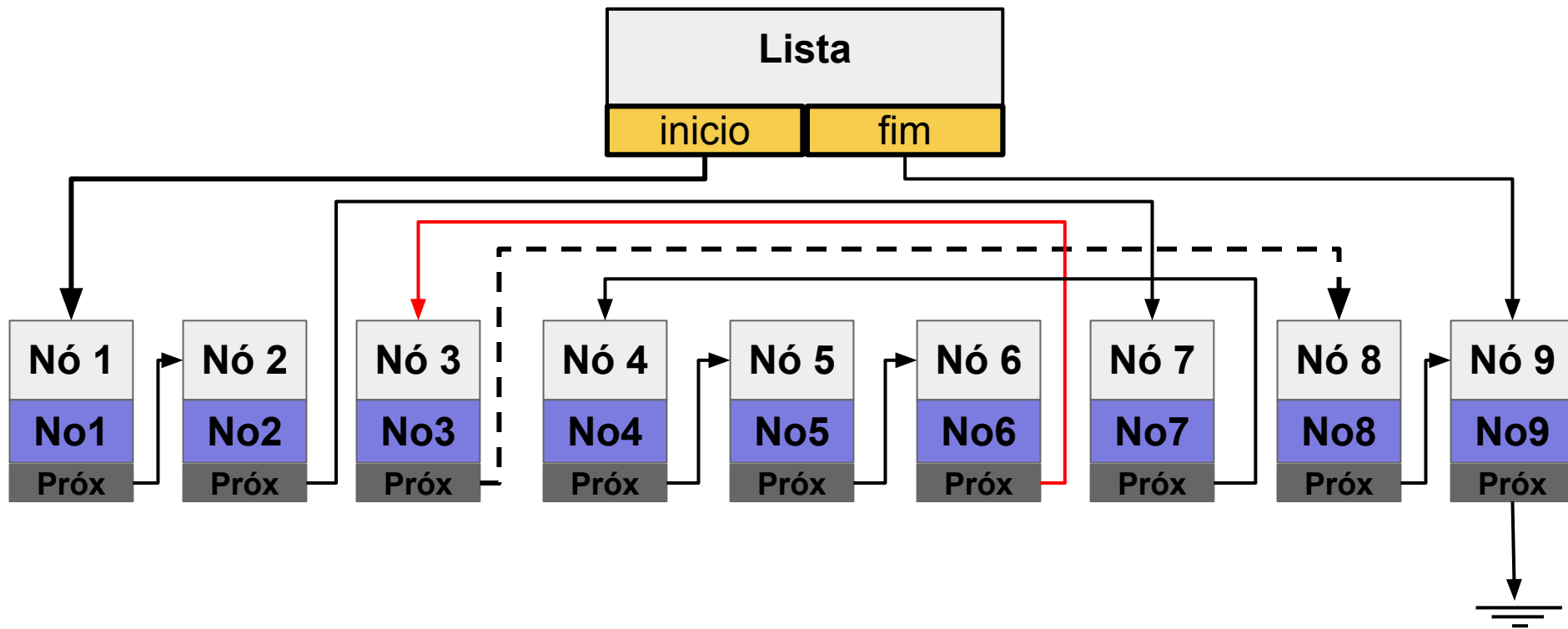
TROCAR DOIS ELEMENTOS DE POSIÇÃO - VI



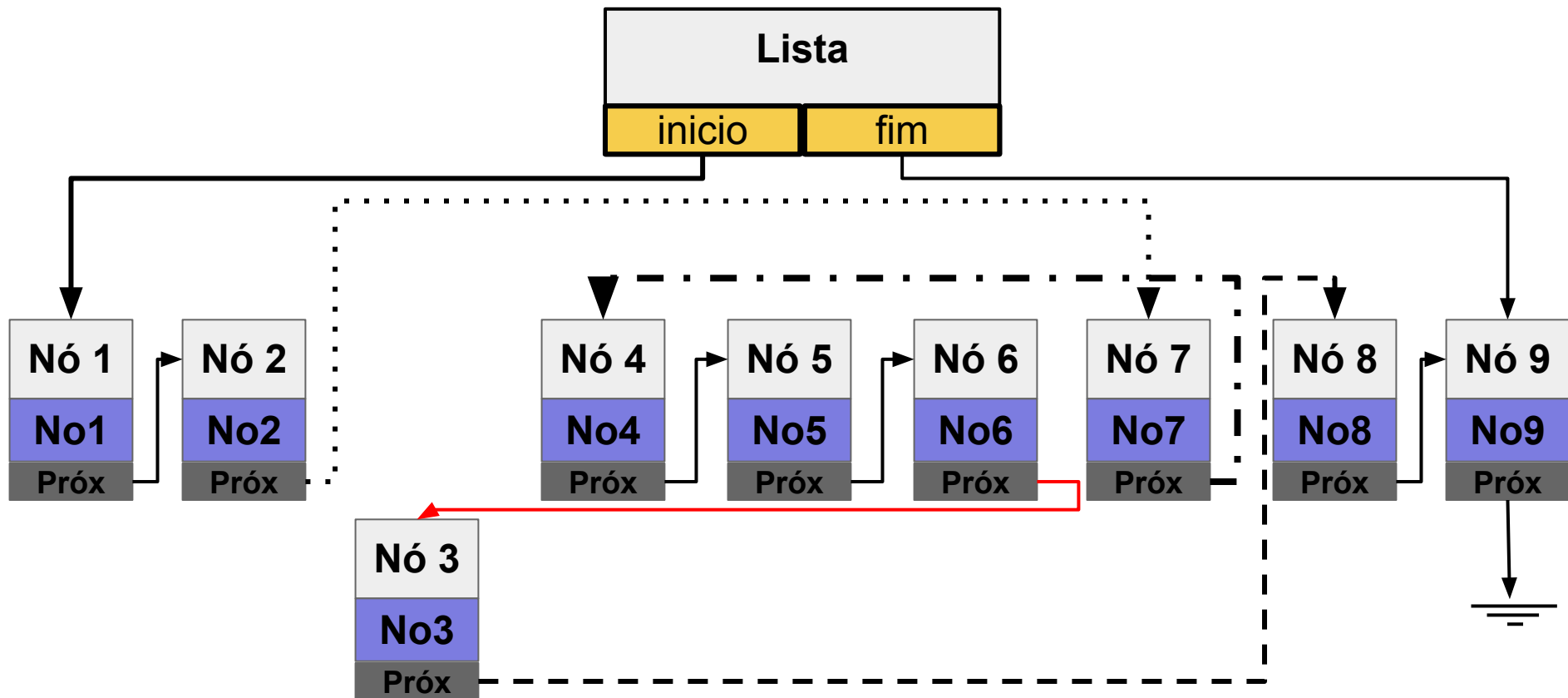
TROCAR DOIS ELEMENTOS DE POSIÇÃO - VII



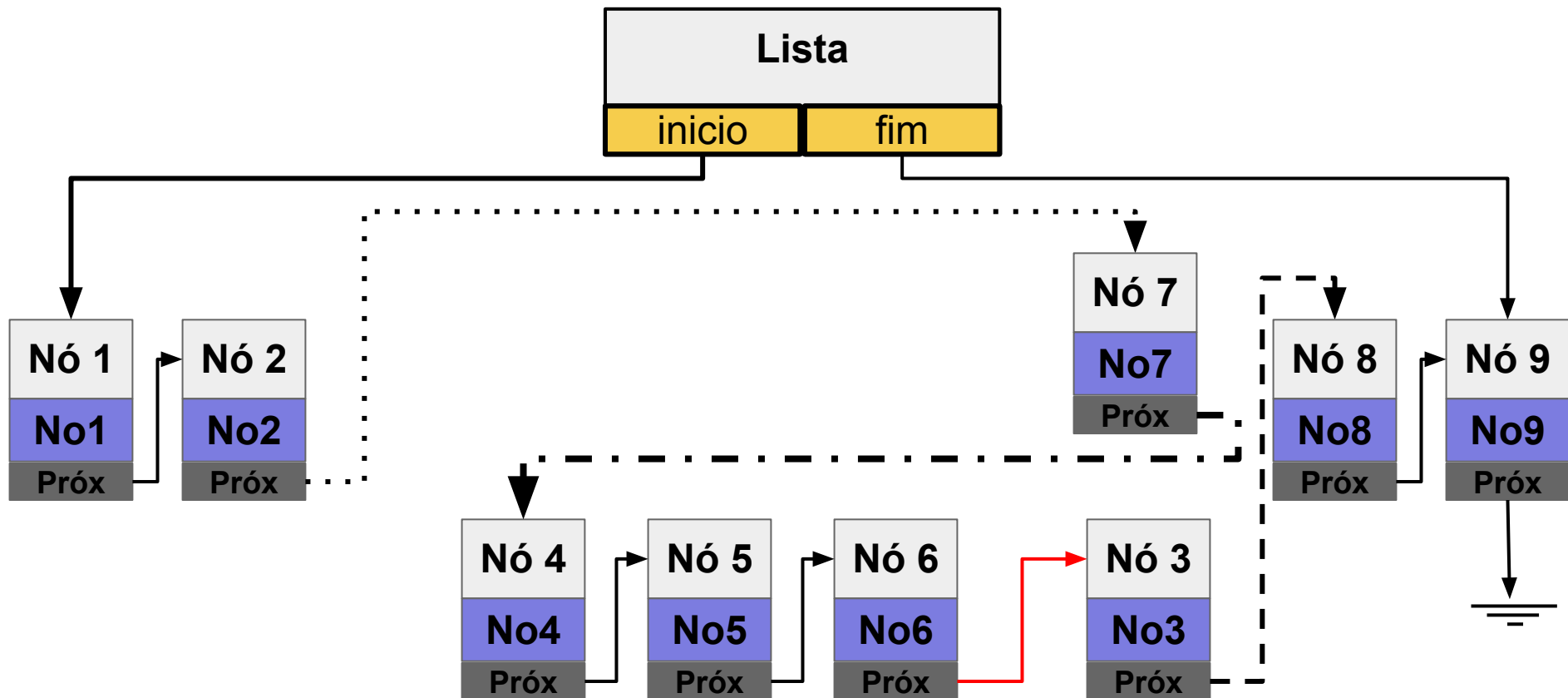
TROCAR DOIS ELEMENTOS DE POSIÇÃO - VIII



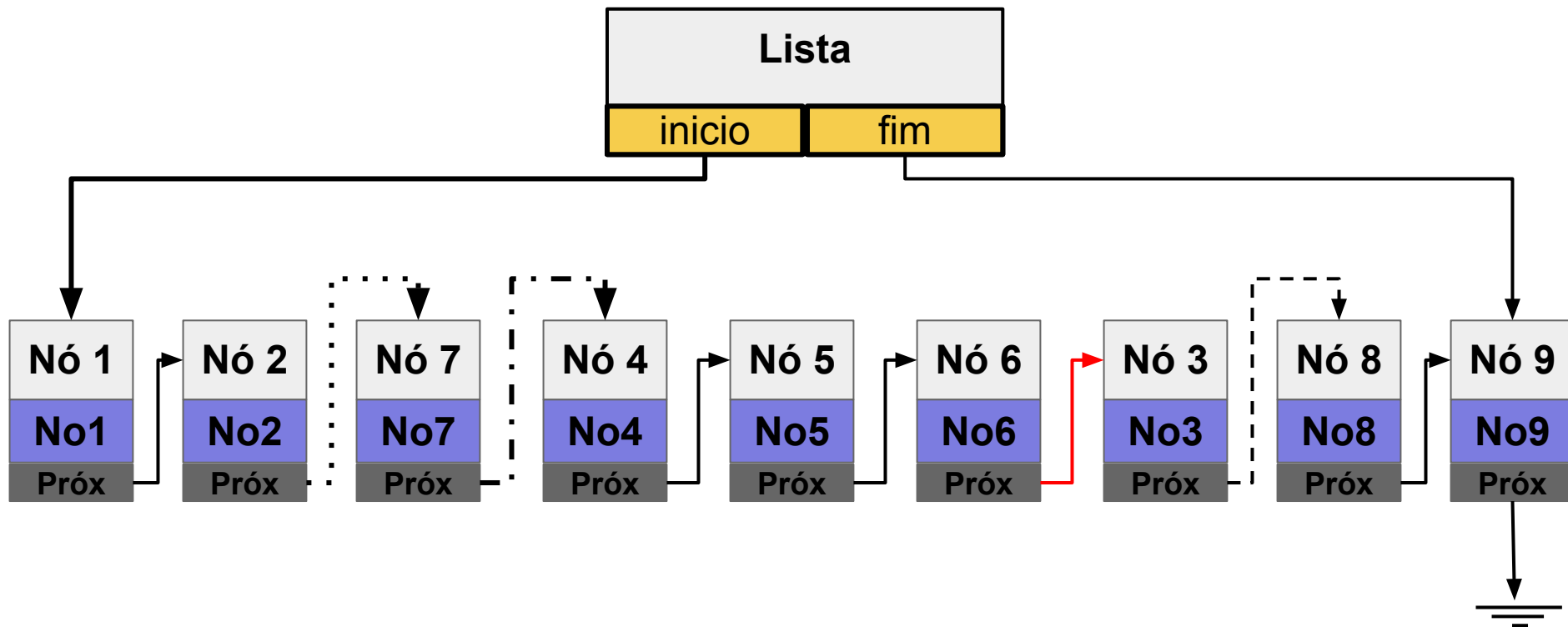
TROCAR DOIS ELEMENTOS DE POSIÇÃO - IX



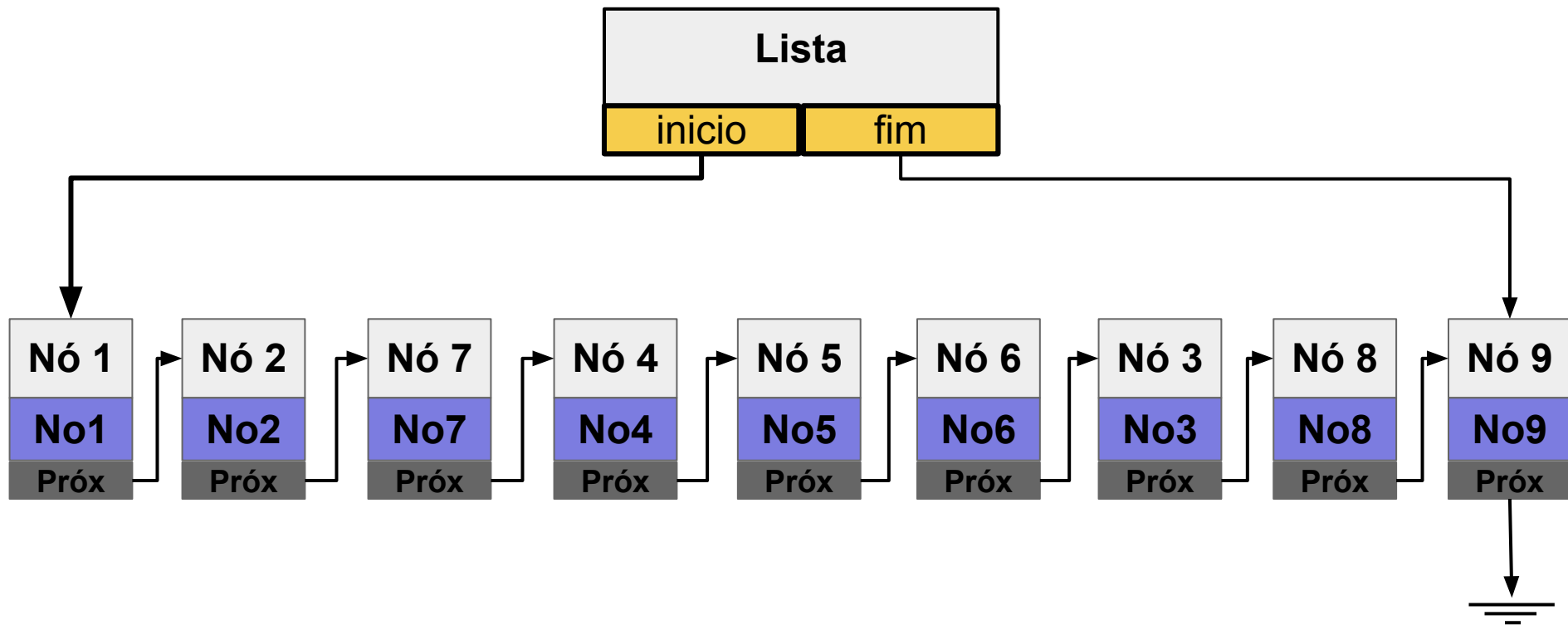
TROCAR DOIS ELEMENTOS DE POSIÇÃO - X



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;

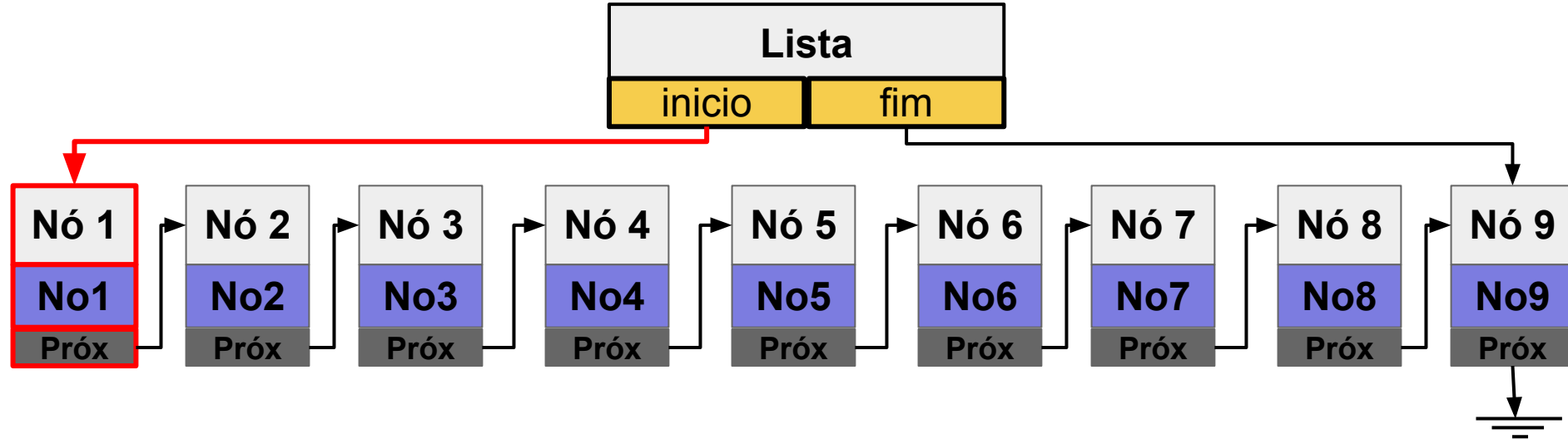
posAuxiliar1 ← auxiliar1.proximo;

auxiliar1.proximo ← auxiliar2.proximo;

auxiliar2.proximo ← posAuxiliar1;

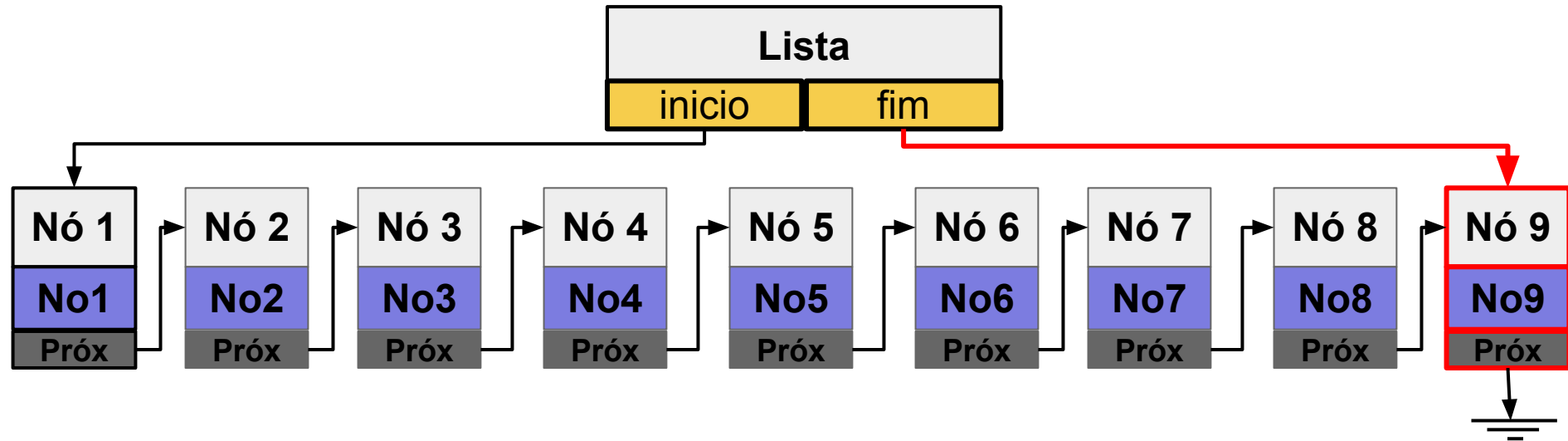
TROCAR DOIS ELEMENTOS DE POSIÇÃO - CASOS ESPECÍFICOS

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



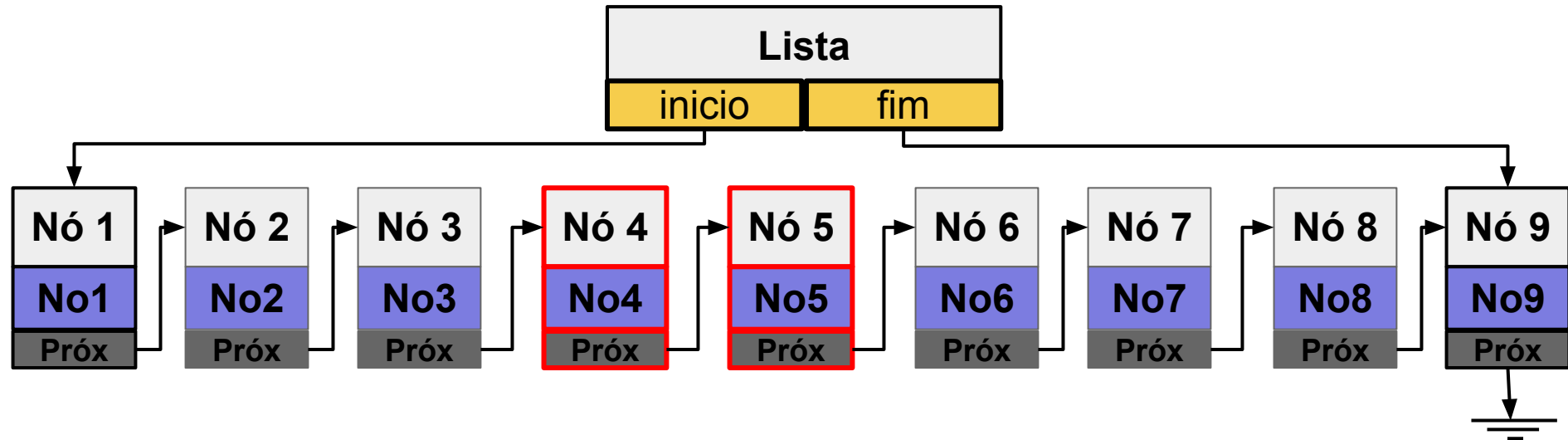
TROCAR DOIS ELEMENTOS DE POSIÇÃO - CASOS ESPECÍFICOS

- 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:

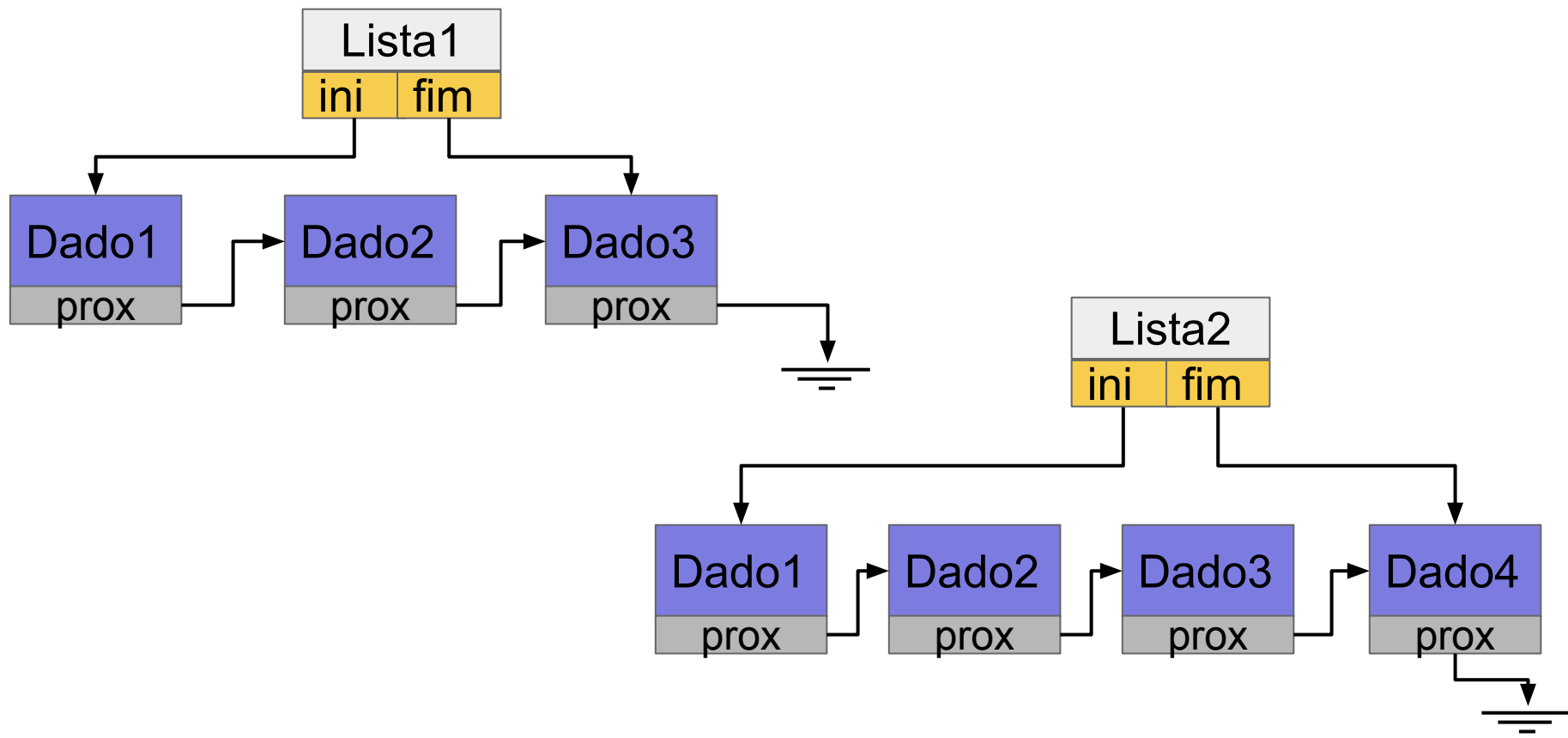
```
aux1 ← acessarPosicao(posicao1);  
aux2 ← acessarPosicao(posicao2);  
temp ← aux1;  
aux1 ← aux2;  
aux2 ← temp;
```

TROCAR DOIS ELEMENTOS DE POSIÇÃO - DISCUSSÃO - II

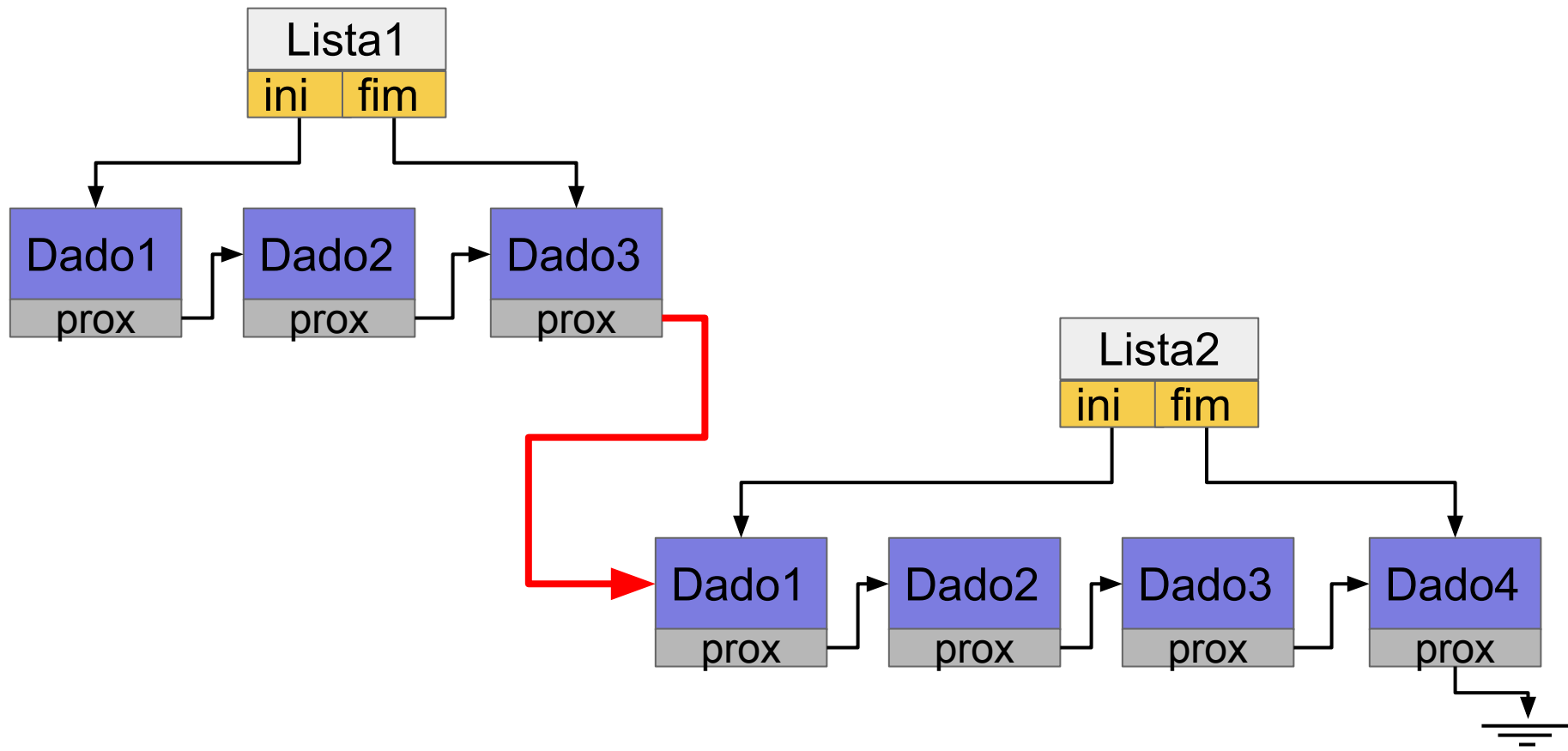
O problema com a troca de valores é que ela geralmente utiliza construtores de cópia e operação de atribuição. Caso o dado armazenado ocupe pouco espaço em memória, isso é feito de forma relativamente rápida e eficiente.

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.

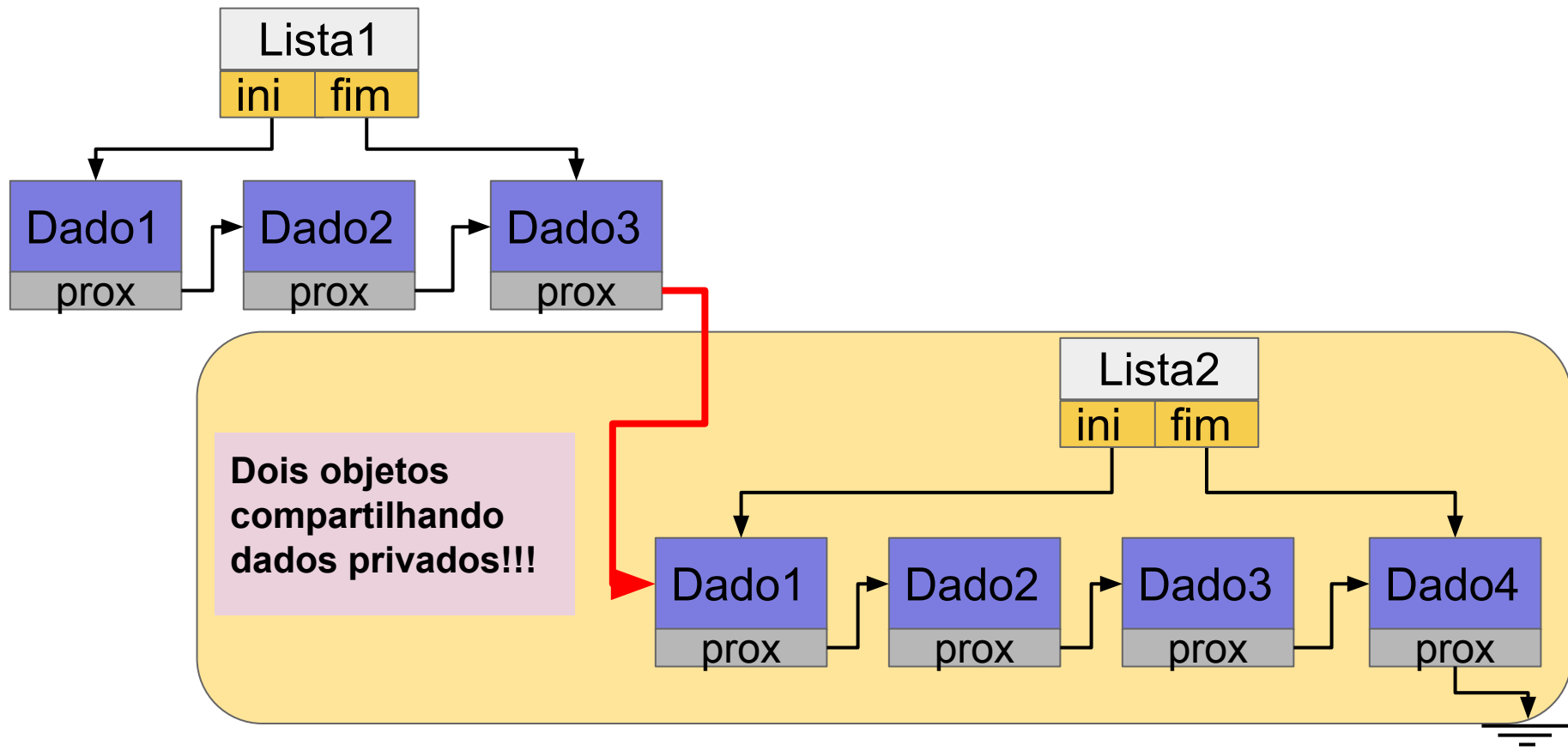
CONCATENAR DUAS LISTAS



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 Greggi
- Renato Ramos da Silva

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

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).