Listas encadeadas

Franciny Medeiros

Listas encadeadas

Listas Encadeadas

- As listas ligadas ou encadeadas são conjuntos de elementos encadeados, onde cada elemento contém uma ligação com um ou mais elementos da lista.

- As listas não possuem regras para o acesso de seus elementos.

- Uma lista ecadeada tem necessariamente uma variável ponteiro apontando para o seu primeiro elemento.
  - Essa variável será utilizada sempre, mesmo que a lista esteja vazia.
  - Caso esta primeira variável não seja atualizada corretamente a lista poderá se perder na memória e não ser mais acessível.

Listas Encadeadas

- Cada elemento da lista ligada será composto por 2 partes principais:
  - uma parte conterá as informações.
  - a outra as conexões com outros elementos.

Listas Encadeadas

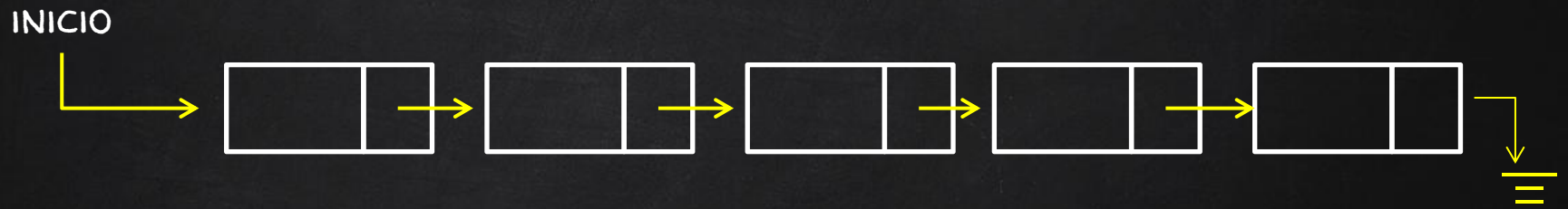
- Cada elemento da lista ligada será composto por 2 partes principais:
  - uma parte conterá as informações.
  - a outra as conexões com outros elementos.

Dados

Conexão para o próximo

```
struct noh{ // nome da estrutura
     char dados; // campo
     struct noh* proximo; //ponteiro
para a proxima celula
}
```

Exemplo

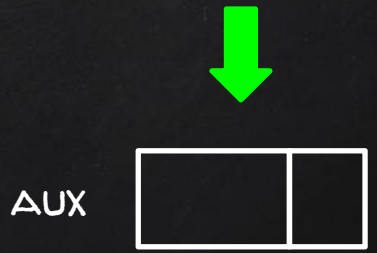


✘ Iniciando a lista

```
struct noh*  inicio;
inicio = NULL;
```

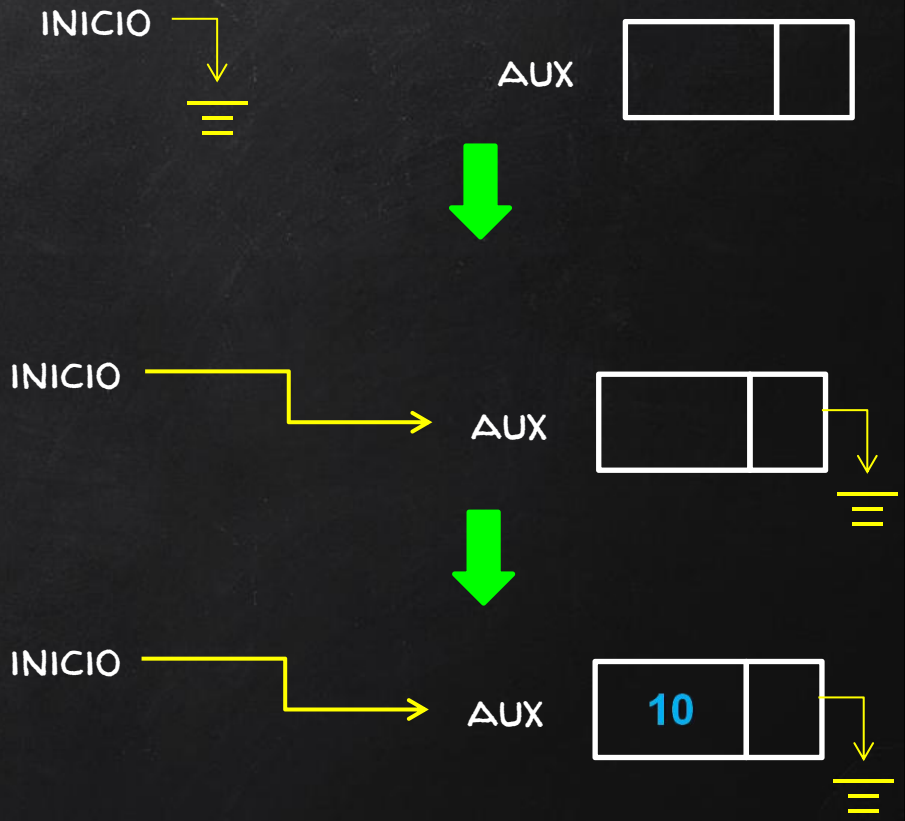
✘ Criando o primeiro nó

```
struct noh* aux;    //variavel auxiliar
// reserva de memória para nova célula com endereço de memória alocada armazenado em aux:

aux = (struct noh*) malloc (sizeof (struct noh));
```



✘ Criando o primeiro nó

```
struct noh* aux;    //variavel auxiliar
// reserva de memória para nova célula com
endereço de memória alocada armazenado
em aux.

aux = (struct noh*) malloc (sizeof (struct noh));
 aux->proximo = 0;
 inicio = aux;   // copia o endereco de aux em inicio
aux->dados = 10;
```



Percorrendo a lista

- O primeiro nó da lista é especialmente fácil de acessar, pois existe sempre uma variável apontando para ele (o *inicio*).

- Todas os nós tem sempre alguém que os aponta.
  - Ou seja, o segundo nó da lista será apontado pelo primeiro, o terceiro pelo segundo e assim por diante até o final da lista.

Percorrendo a lista

- Para percorrer uma lista e acessar cada um dos nós, utiliza-se uma variável auxiliar.

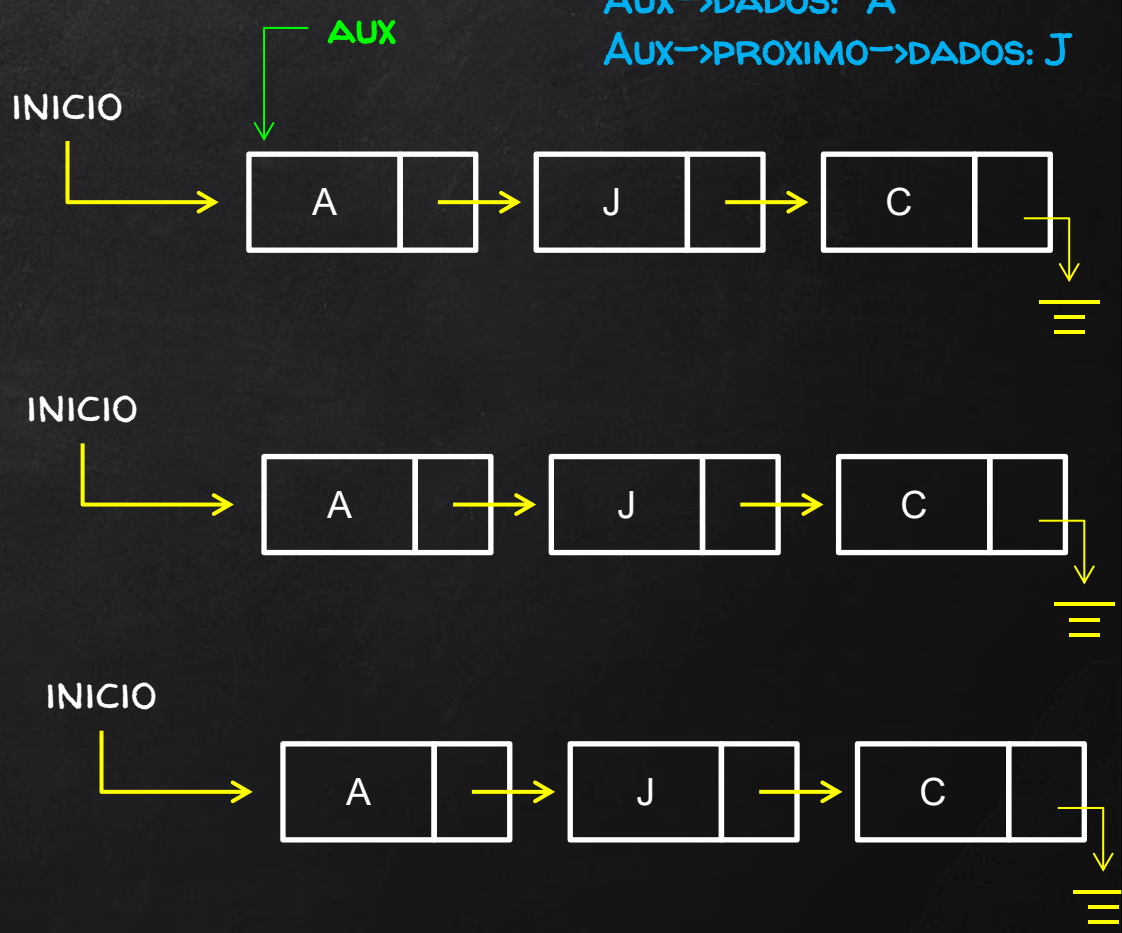
- Essa variável irá receber o endereço de um nó e permitirá que o conteúdo desse nó seja acessado (consultado, modificado, etc).

```
struct noh* aux;    // variável auxiliar para percorrer a lista
```

Percorrendo a lista

- Inicialmente, *aux* deverá apontar para o mesmo lugar que a variável de inicio da lista para que possamos percorrê-la.

- Depois, *aux* receberá o endereço contido no campo ponteiro, ou seja, receberá o endereço do próximo nó: isso acontecerá até *aux* valer 0 (zero) e a lista ter sido completamente percorrida.

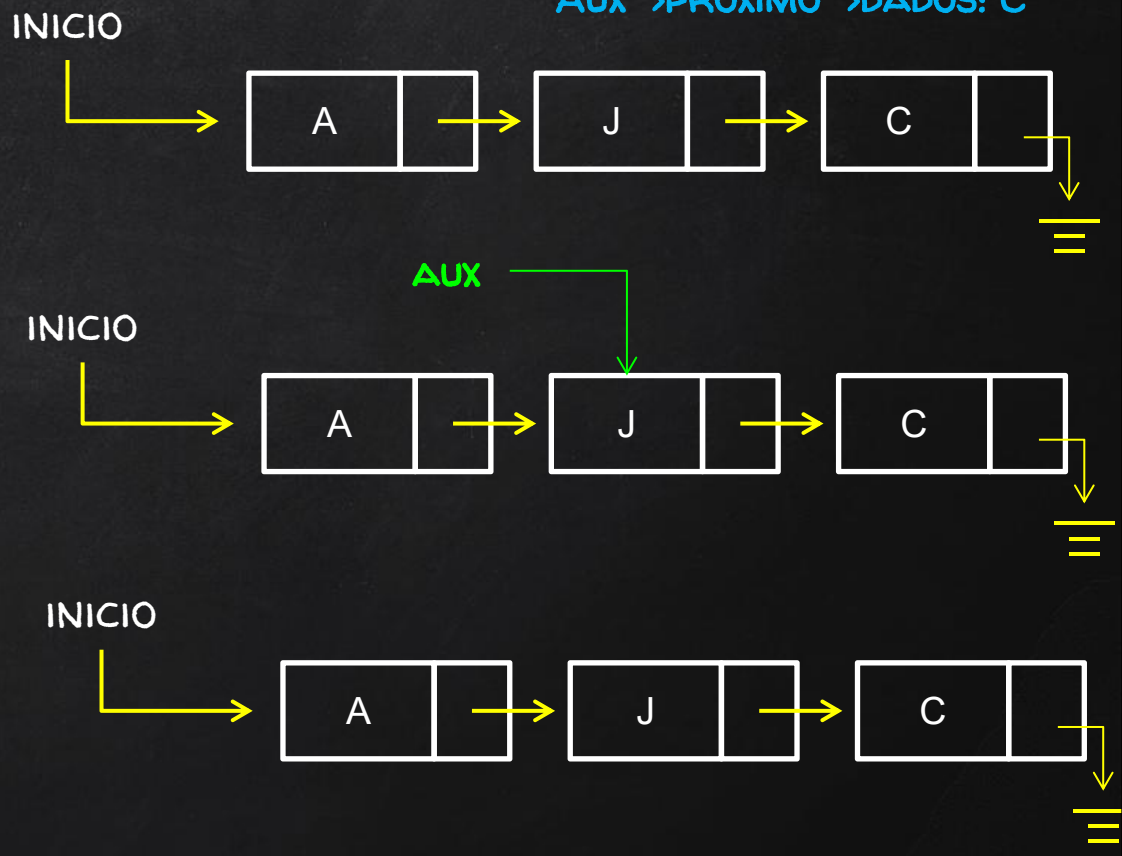


Percorrendo a lista

- Inicialmente, *aux* deverá apontar para o mesmo lugar que a variável de inicio da lista para que possamos percorrê-la.

- Depois, *aux* receberá o endereço contido no campo ponteiro, ou seja, receberá o endereço do próximo nó: isso acontecerá até *aux* valer 0 (zero) e a lista ter sido completamente percorrida.

aux = aux->proximo;
Aux->dados: J
Aux->proximo->dados: C

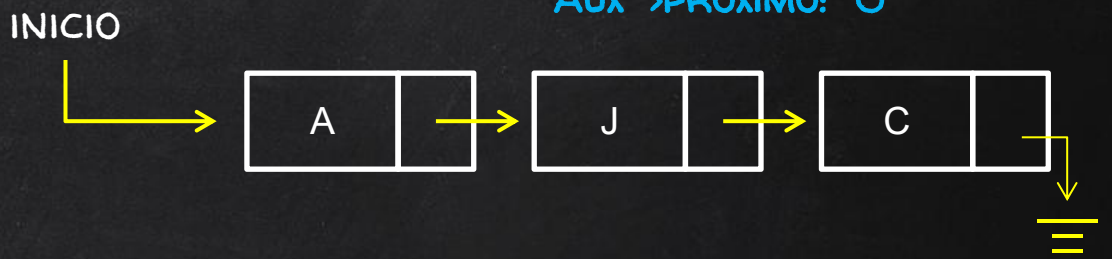


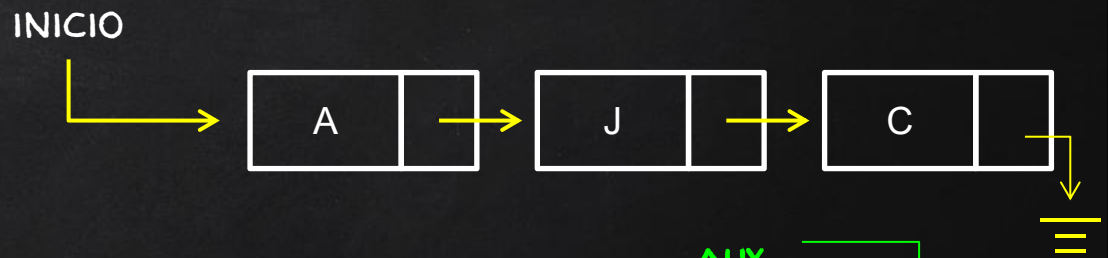
Percorrendo a lista

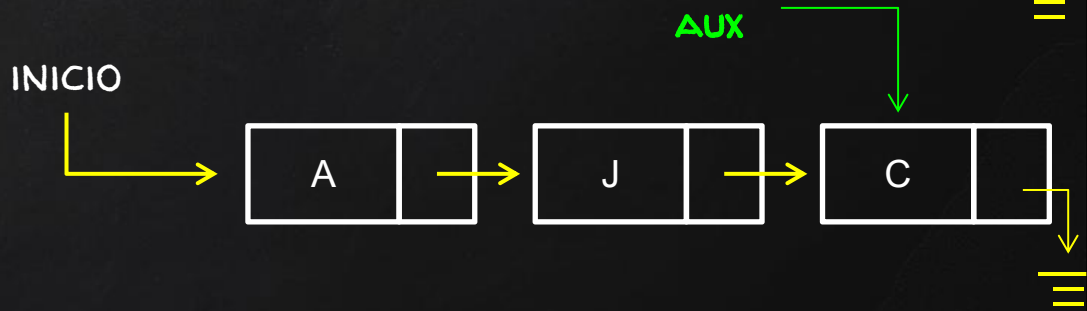
- Inicialmente, *aux* deverá apontar para o mesmo lugar que a variável de inicio da lista para que possamos percorrê-la.

- Depois, *aux* receberá o endereço contido no campo ponteiro, ou seja, receberá o endereço do próximo nó: isso acontecerá até *aux* valer 0 (zero) e a lista ter sido completamente percorrida.

AUX = AUX->PROXIMO;
AUX->DADOS : C
AUX->PROXIMO: 0







Inserindo um nó na lista

- A inclusão de nós numa lista ligada deve ser feita com cuidado para que os ponteiros permaneçam atualizados e parte da lista não se perca.

- Um nó pode ser incluindo em diferentes posições na lista:
  - Início
  - Meio
  - Fim

Inserindo no final

1. Se a lista não é vazia percorrer a lista até o último nó
2. Alocar memória
3. Atribuir dados aos campos de dados
4. Atribuir ao campo ponteiro do último nó célula o endereço do novo nó
5. atribuir 0 ao campo ponteiro do nó incluído

```
struct noh *aux, *nova;
aux = inicio;

if (aux!=0) {
   while(aux->prox imo!= 0)
       aux = aux->proximo;
   novo = (struct noh*)malloc(sizeof(struct noh))
   novo->valor = 'F';
   novo->proximo = aux->proximo;
   aux->proximo = novo;
}
```

Excluindo um nó

- Os passos para exclusão de um nó vão depender da posição do nó dentro da lista. Uma exclusão pode ocorrer no início, final ou no meio da lista.

- Da mesma forma que na inclusão, o principal cuidado é com os ponteiros para que a lista não tenha nós perdidos durante o processo.

Excluindo um nó

- ✘ Percorrer a lista até o nó anterior à do valor que se quer apagar.
- ✘ Atribuir o endereço do nó a um ponteiro auxiliar (essencial para que a memória seja liberada).
- ✘ Alterar o endereço apontado pelo nó anterior para que ele aponte para onde o nó apagado aponta atualmente.
- ✘ Liberar a memória do nó apagado.

```
struct noh *aux, *apaga;
aux = inicio;

while(aux->proximo->valor != 'C')
  aux = aux->proximo;
apaga = aux->proximo;
aux->proximo = apaga->proximo;
free(apaga);
```

Atividade

- Elabore um algoritmo para a inclusão de elementos numa lista encadeada simples a partir do seu início.

- Lembre-se de verificar quando o nó é o primeiro a ser inserida na lista.

- Faça a simulação desenhando os passos da inclusão.

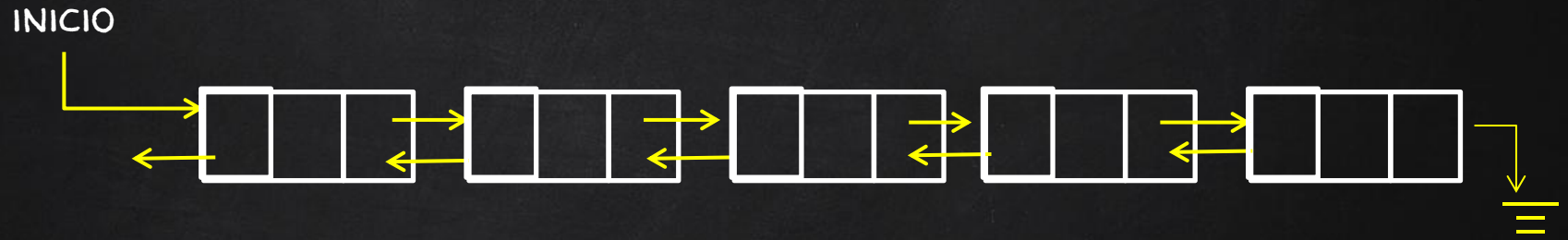
Listas Duplamente Encadeada

- cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior;

- dado um elemento, podemos acessar ambos os elementos adjacentes: o próximo e o anterior;

Listas Duplamente Encadeadas

Exemplo

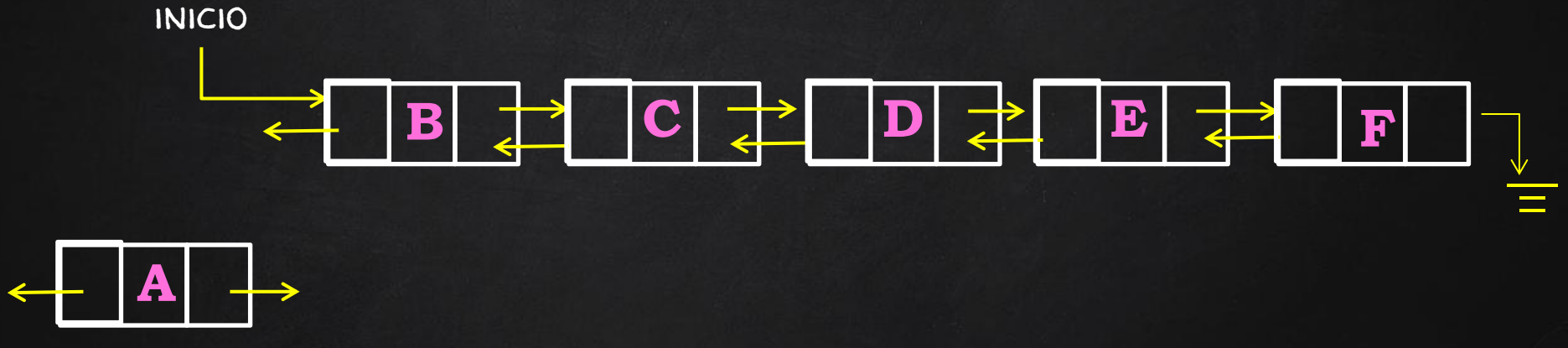


Listas Encadeadas

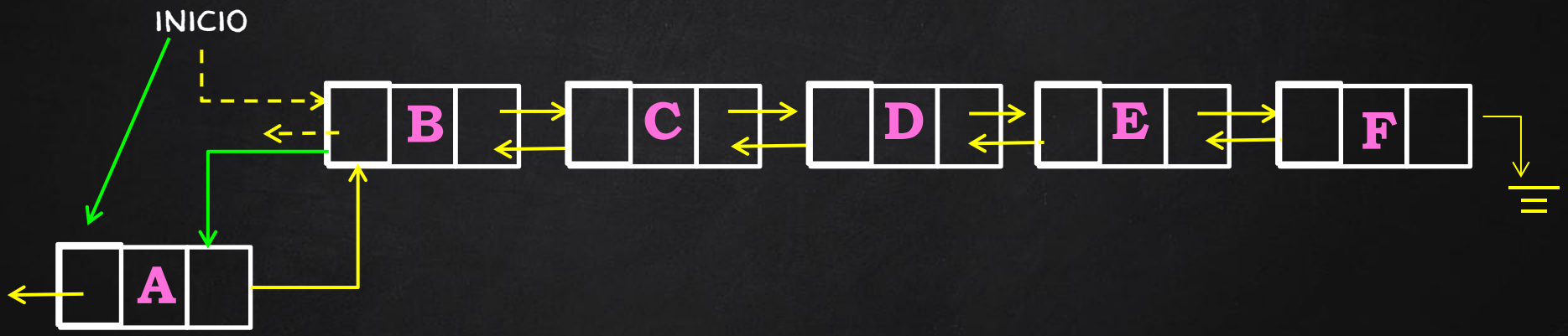
- Cada elemento da lista ligada será composto por 3 partes principais:
  - uma parte conterá as informações.
  - a outra as conexões com elementos próximos.
  - a outra as conexões com elementos anteriores


```
struct noh{ // nome da estrutura
    char dados; // campo
    struct noh* proximo;
    struct noh* anterior;
}
```

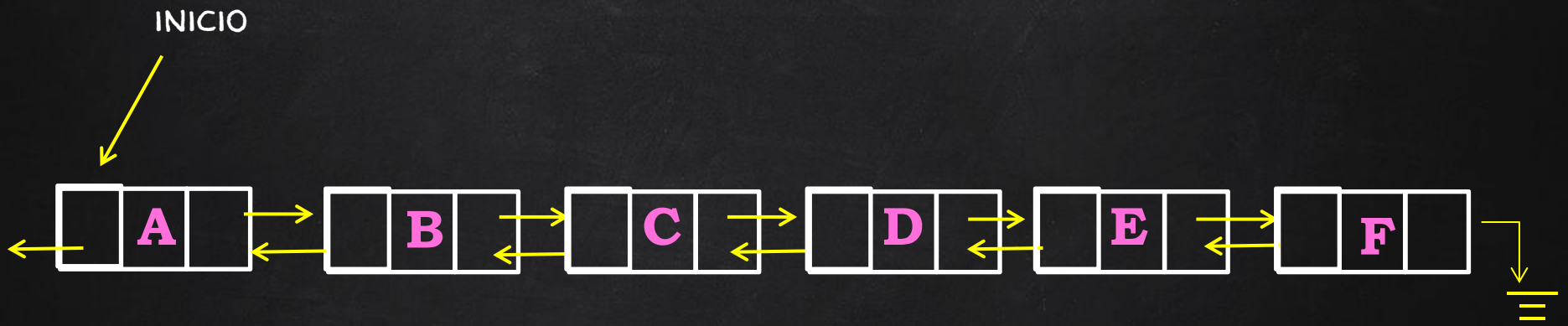
Exemplo de Inserção



Exemplo de Inserção



Exemplo de Inserção



Inserindo no início

```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```

Inserindo no início

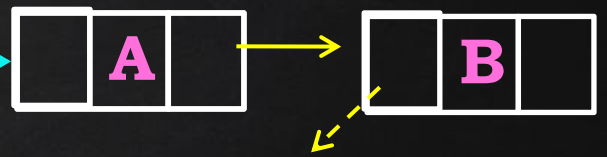
```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```

Inserindo no início

```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```

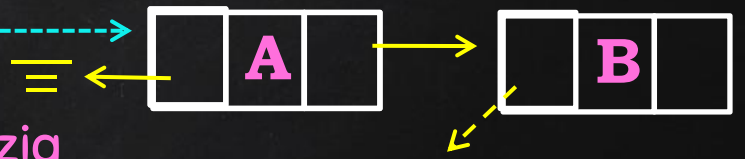
Inserindo no início

```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```



Inserindo no início

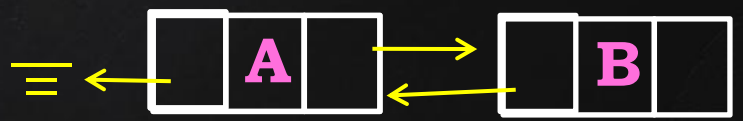
```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```



Inserindo no início

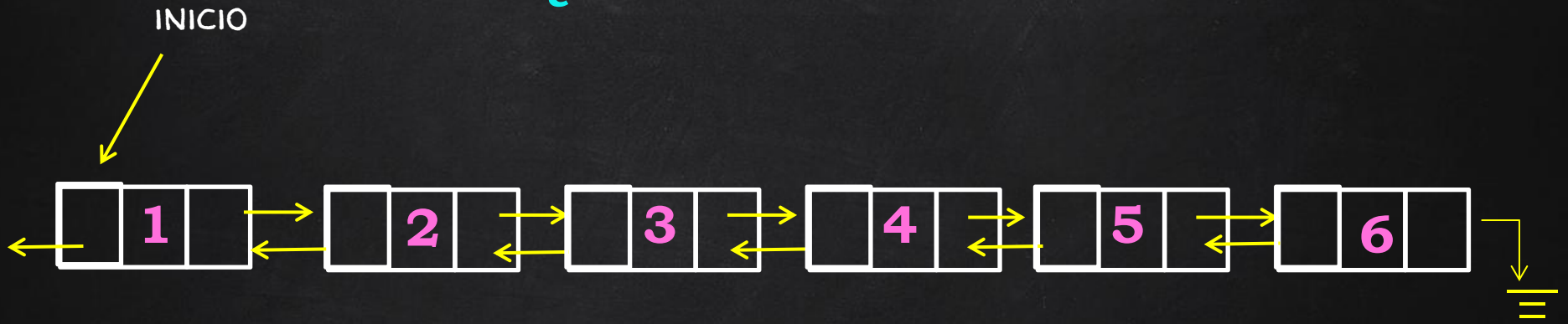
```
noh * insere (noh* l, char v)

    noh* novo = (noh*) malloc (sizeof(noh));
    novo->info = v;
    novo->proximo = l;
    novo->anterior = NULL;

    if ( l != NULL) // verifica se a lista está vazia
       l -> anterior = novo;
    return novo;
```

l = null? não



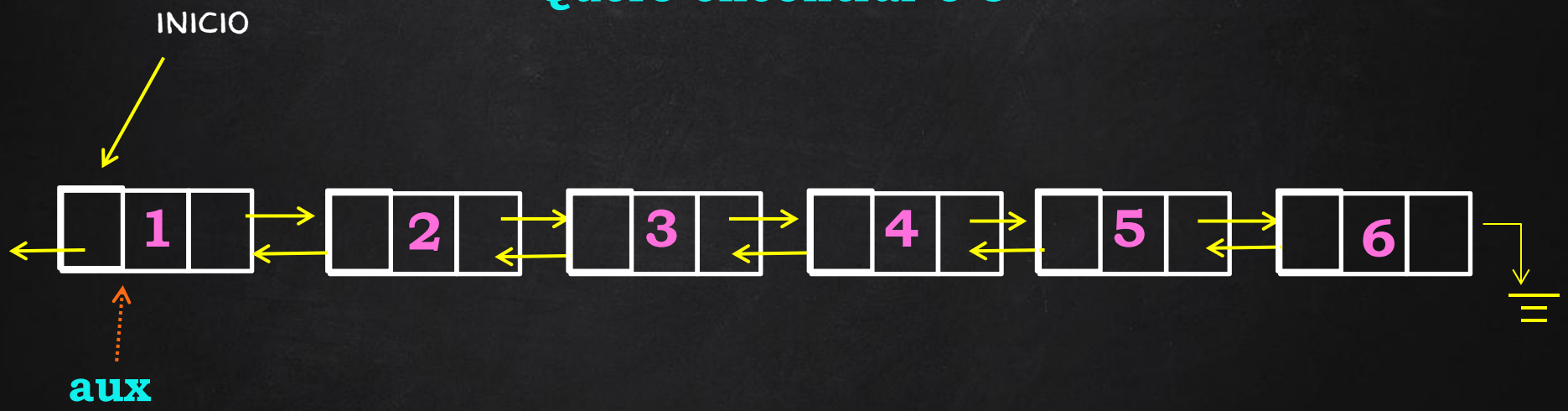
Exemplo de Busca

Quero encontrar o 3



Exemplo de Busca

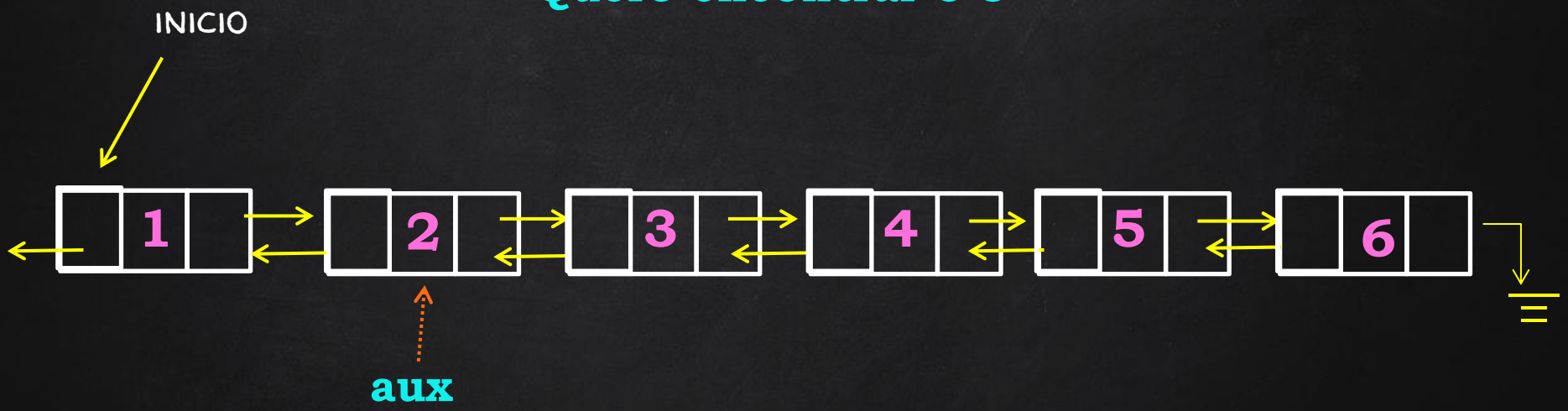
Quero encontrar o 3



aux->info = 3 ?
não!
aux = aux->prox

Exemplo de Busca

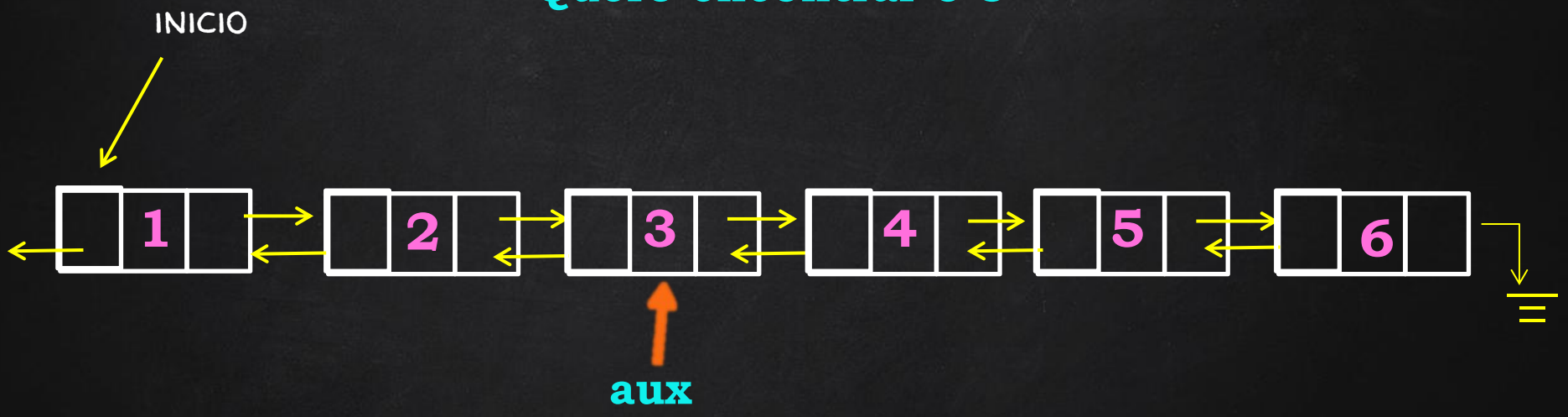
Quero encontrar o 3



aux->info = 3 ?
não!
aux = aux->prox

Exemplo de Busca

Quero encontrar o 3



aux->info = 3 ?
sim!
retorna o elemento encontrado

Buscando um Elemento

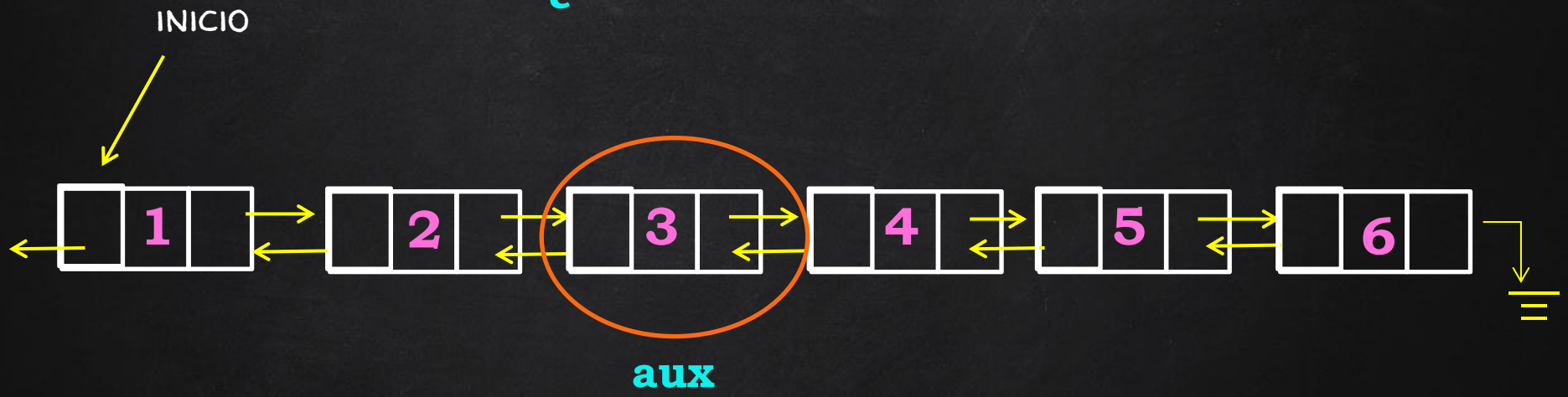
```
noh* busca ( noh* l, int v)

    noh* aux;
    for (aux = l; aux->prox != NULL; aux = aux->prox)

        if (aux ->info == v)
            return aux;

    return NULL;
```

Remover um elemento da lista

- ✘ É preciso acertar os ponteiros do encadeamento
- ✘ Usar a função de busca para achar o elemento e em seguida acertar o encadeamento
- ✘ Liberar o elemento no final

Exemplo de Remoção

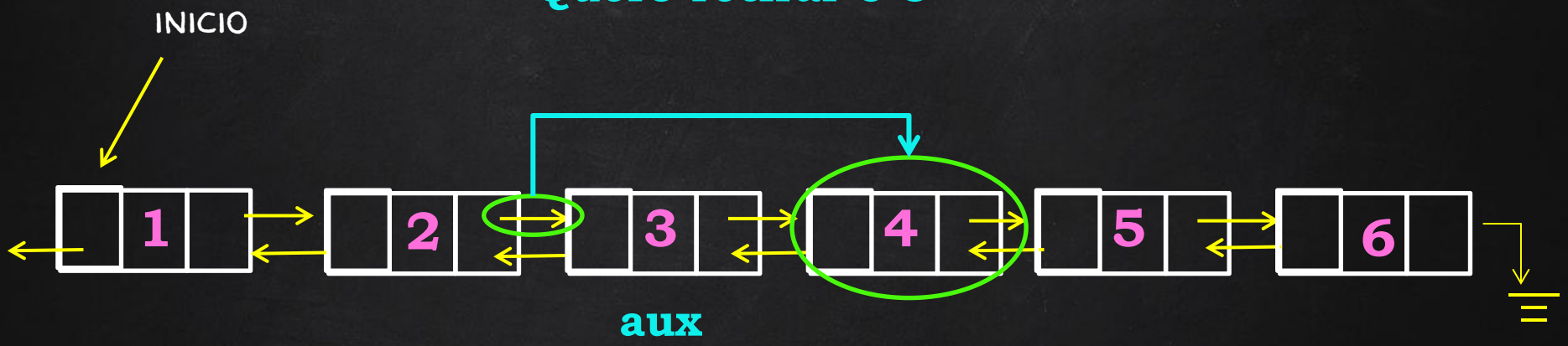
Quero retirar o 3



Para retirar o 3 é preciso acertar os seus ponteiros de próximo e anterior antes de retirá-lo

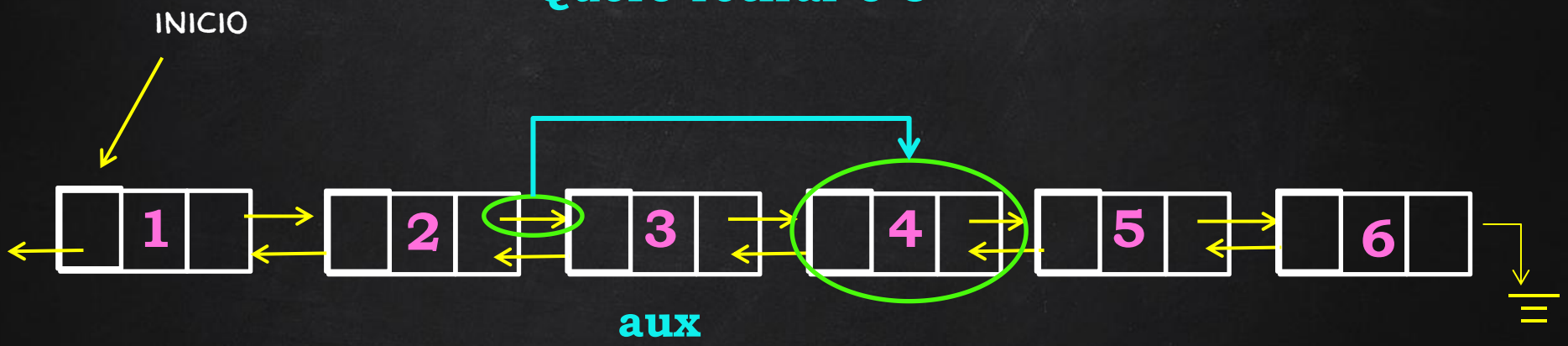
Exemplo de Remoção

Quero retirar o 3

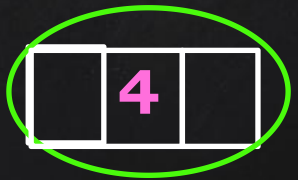


Exemplo de Remoção

Quero retirar o 3



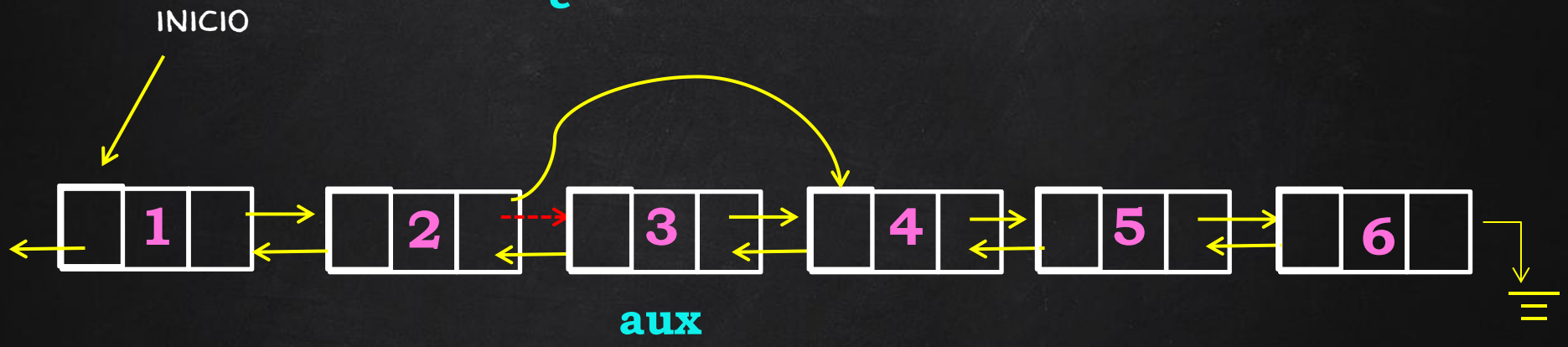
aux->ant->prox



aux->prox

Exemplo de Remoção

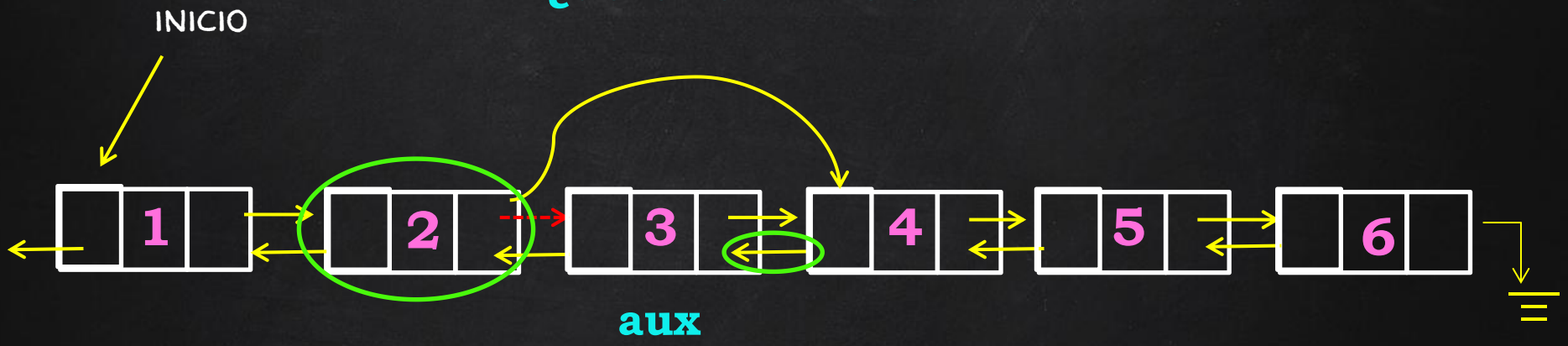
Quero retirar o 3



aux->ant->prox = aux -> prox

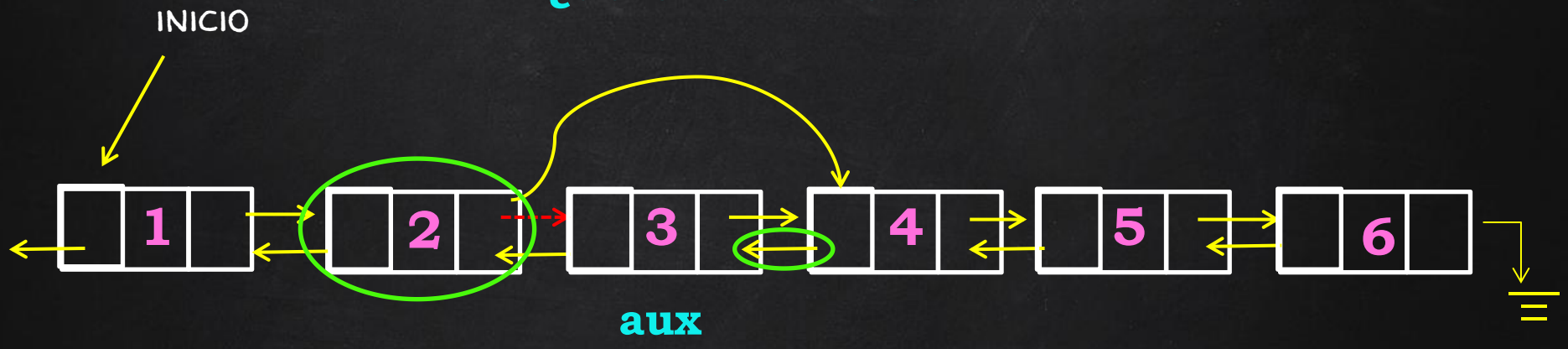
Exemplo de Remoção

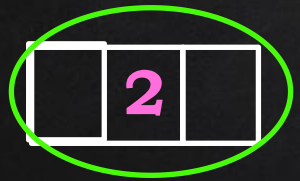
Quero retirar o 3



Exemplo de Remoção

Quero retirar o 3



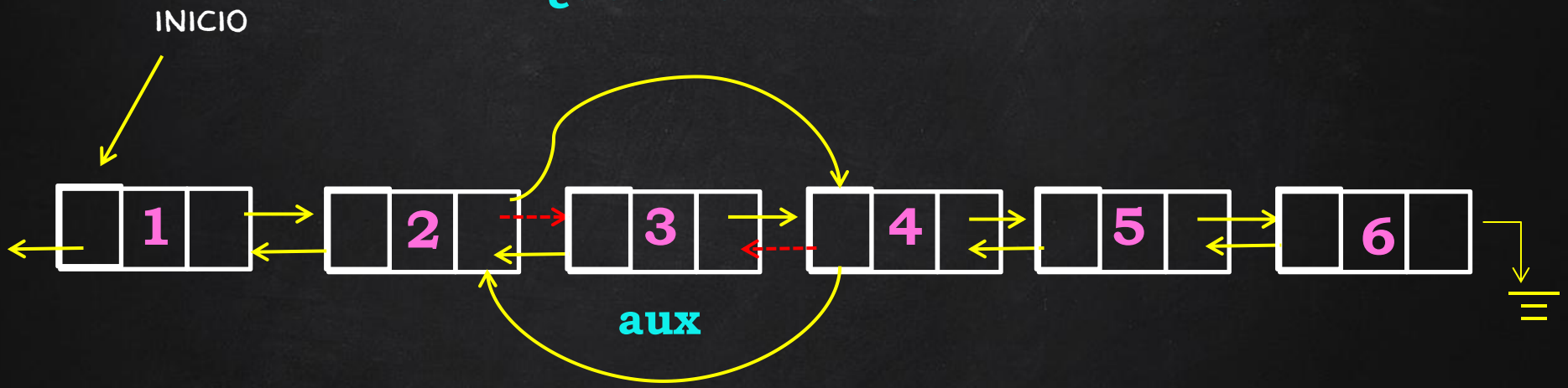


aux->ant

aux->prox->ant

Exemplo de Remoção

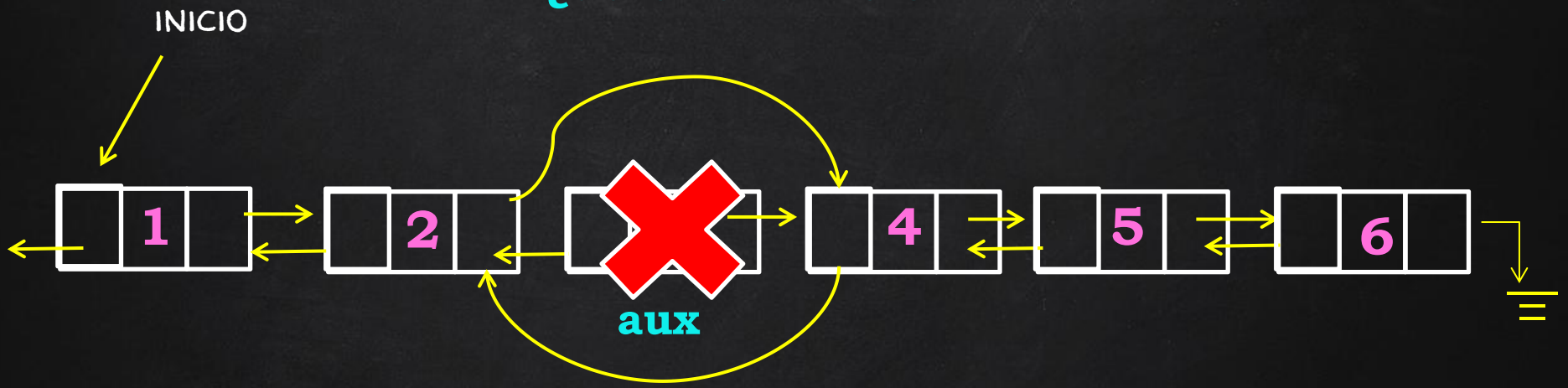
Quero retirar o 3



aux->prox->ant = aux -> ant

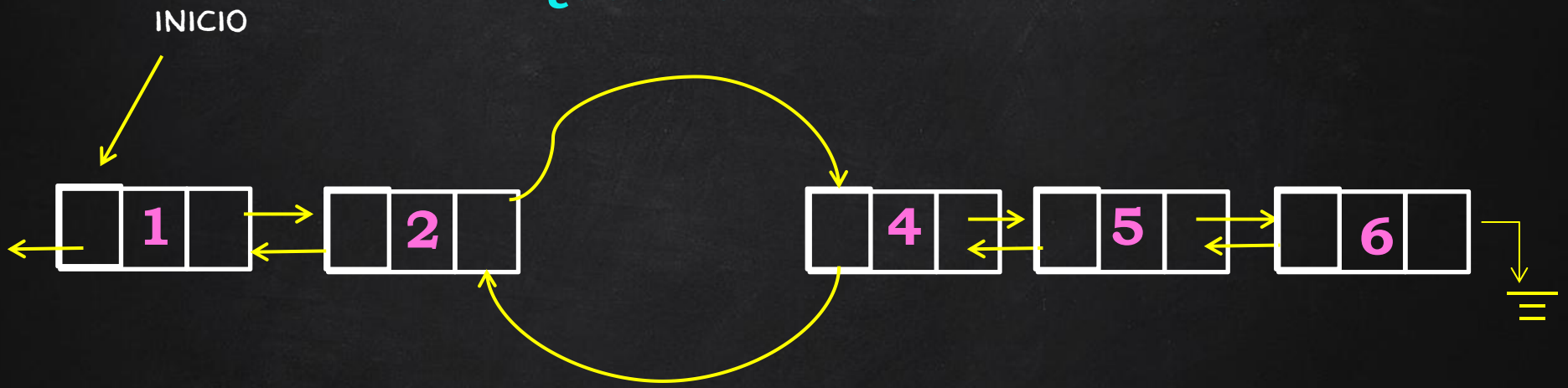
Exemplo de Remoção

Quero retirar o 3



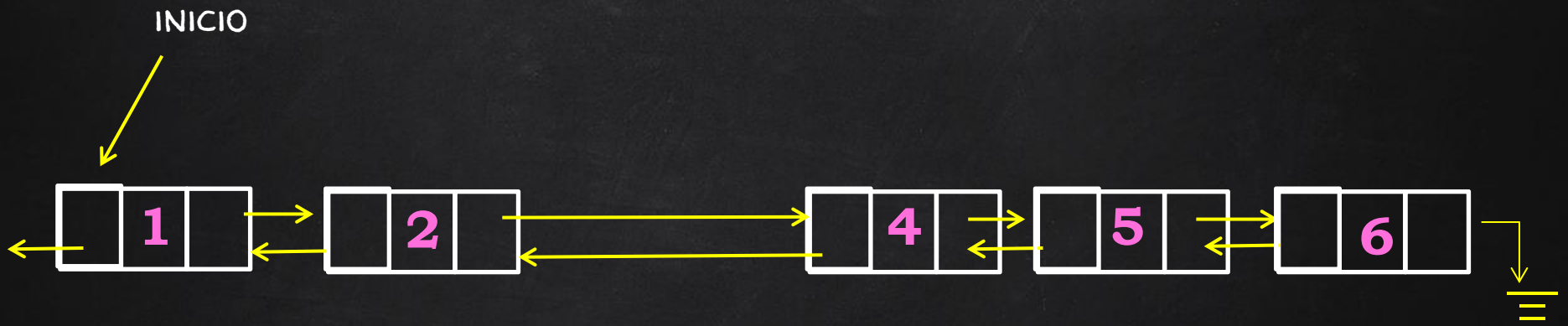
Exemplo de Remoção

Quero retirar o 3



Exemplo de Remoção

Lista sem o elemento 3



Removendo um nó

```
noh* remover (noh* l, int v)
    noh* aux = busca (l,v);

    if ( aux == NULL )
        return l;          //não encontrou o elemento

    if ( l == aux)
        l = aux -> prox;
    else
        aux -> ant -> prox = aux -> prox;
    if (aux -> prox != NULL)
        aux -> prox->ant = aux -> ant;
    free(aux);
    return l;
```

Exercício

1) Implemente uma lista encadeada para simular o comportamento de uma pilha;

2) implemente uma lista duplamente encadeada para simular o comportamento de uma fila;